Broadcasts overview

Android apps can send or receive broadcast messages from the Android system and other Android apps, similar to the publish-subscribe (https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern) design pattern. These broadcasts are sent when an event of interest occurs. For example, the Android system sends broadcasts when various system events occur, such as when the system boots up or the device starts charging. Apps can also send custom broadcasts, for example, to notify other apps of something that they might be interested in (for example, some new data has been downloaded).

Apps can register to receive specific broadcasts. When a broadcast is sent, the system automatically routes broadcasts to apps that have subscribed to receive that particular type of broadcast.

Generally speaking, broadcasts can be used as a messaging system across apps and outside of the normal user flow. However, you must be careful not to abuse the opportunity to respond to broadcasts and run jobs in the background that can contribute to a slow system performance, as described in the following video.

About system broadcasts

The system automatically sends broadcasts when various system events occur, such as when the system switches in and out of airplane mode. System broadcasts are sent to all

apps that are subscribed to receive the event.

The broadcast message itself is wrapped in an `Intent` (https://developer.android.com/reference/android/content/Intent.html) object whose action string identifies the event that occurred (for example `android.intent.action.AIRPLANE_MODE`). The intent may also include additional information bundled into its extra field. For example, the airplane mode intent includes a boolean extra that indicates whether or not Airplane Mode is on.

For more information about how to read intents and get the action string from an intent, see Intents and Intent Filters (https://developer.android.com/guide/components/intents-filters.html).

For a complete list of system broadcast actions, see the `BROADCAST_ACTIONS.TXT` file in the Android SDK. Each broadcast action has a constant field associated with it. For example, the value of the constant `ACTION_AIRPLANE_MODE_CHANGED` (https://developer.android.com/reference/android/content/Intent.html#ACTION_AIRPLANE_MODE_CHANGED) is `android.intent.action.AIRPLANE_MODE`. Documentation for each broadcast action is available in its associated constant field.

Changes to system broadcasts

As the Android platform evolves, it periodically changes how system broadcasts behave. Keep the following changes in mind if your app targets Android 7.0 (API level 24) or higher, or if it's installed on devices running Android 7.0 or higher.

Android 9

Beginning with Android 9 (API level 28), The `NETWORK_STATE_CHANGED_ACTION` (https://developer.android.com/reference/android/net/wifi/WifiManager.html#NETWORK_STATE_CHANGED_ACTION) broadcast doesn't receive information about the user's location or personally identifiable data.

In addition, if your app is installed on a device running Android 9 or higher, system broadcasts from Wi-Fi don't contain SSIDs, BSSIDs, connection information, or scan results. To get this information, call `getConnectionInfo()` (https://developer.android.com/reference/android/net/wifi/WifiManager.html#getConnectionInfo()) instead.

Android 8.0

Beginning with Android 8.0 (API level 26), the system imposes additional restrictions on manifest-declared receivers.

If your app targets Android 8.0 or higher, you cannot use the manifest to declare a receiver for most implicit broadcasts (broadcasts that don't target your app specifically). You can still use a context-registered receiver (#context-registered-recievers) when the user is actively using your app.

Android 7.0

Android 7.0 (API level 24) and higher don't send the following system broadcasts:

- `ACTION_NEW_PICTURE` (https://developer.android.com/reference/android/hardware/Camera.html#ACTION_NEW_PICTURE)
- `ACTION_NEW_VIDEO` (https://developer.android.com/reference/android/hardware/Camera.html#ACTION_NEW_VIDEO)

Also, apps targeting Android 7.0 and higher must register the `CONNECTIVITY_ACTION` (https://developer.android.com/reference/android/net/ConnectivityManager.html#CONNECTIVITY_ACTION) broadcast using `registerReceiver(BroadcastReceiver, IntentFilter)` (https://developer.android.com/reference/android/content/Context.html#registerReceiver(android.content.BroadcastReceiver, android.content.IntentFilter)). Declaring a receiver in the manifest doesn't work.

Receiving broadcasts

Apps can receive broadcasts in two ways: through manifest-declared receivers and context-registered receivers.

Manifest-declared receivers

If you declare a broadcast receiver in your manifest, the system launches your app (if the app is not already running) when the broadcast is sent.

Note: If your app targets API level 26 or higher, you cannot use the manifest to declare a receiver for *implicit* broadcasts (broadcasts that do not target your app specifically), except for a few implicit broadcasts that are exempted from that restriction (https://developer.android.com/guide/components/broadcast-exceptions.html). In most cases, you can use scheduled jobs (https://developer.android.com/topic/performance/scheduling.html) instead.

To declare a broadcast receiver in the manifest, perform the following steps:

1. Specify the `<receiver>` (https://developer.android.com/guide/topics/manifest/receiver-element.html) element in your app's manifest.

```
<receiver android:name=".MyBroadcastReceiver"  android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
        <action android:name="android.intent.action.INPUT_METHOD_CHANGED" />
    </intent-filter>
</receiver>
```

   The intent filters specify the broadcast actions your receiver subscribes to.

2. Subclass `BroadcastReceiver` (https://developer.android.com/reference/android/content/BroadcastReceiver.html) and implement `onReceive(Context, Intent)` (https://developer.android.com/reference/android/content/BroadcastReceiver.html#onReceive(android.content.Context, android.content.Intent)). The broadcast receiver in the following example logs and displays the contents of the broadcast:

KOTLIN JAVA

```
public class MyBroadcastReceiver extends BroadcastReceiver {
        private static final String TAG = "MyBroadcastReceiver";
        @Override
        public void onReceive(Context context, Intent intent) {
            StringBuilder sb = new StringBuilder();
            sb.append("Action: " + intent.getAction() + "\n");
            sb.append("URI: " + intent.toUri(Intent.URI_INTENT_SCHEME).toS
            String log = sb.toString();
            Log.d(TAG, log);
```

```
            Toast.makeText(context, log, Toast.LENGTH_LONG).show();
        }
    }
```

The system package manager registers the receiver when the app is installed. The receiver then becomes a separate entry point into your app which means that the system can start the app and deliver the broadcast if the app is not currently running.

The system creates a new `BroadcastReceiver` (https://developer.android.com/reference/android/content/BroadcastReceiver.html) component object to handle each broadcast that it receives. This object is valid only for the duration of the call to `onReceive(Context, Intent)` (https://developer.android.com/reference/android/content/BroadcastReceiver.html#onReceive(android.content.Context, android.content.Intent)). Once your code returns from this method, the system considers the component no longer active.

Context-registered receivers

To register a receiver with a context, perform the following steps:

1. Create an instance of `BroadcastReceiver` (https://developer.android.com/reference/android/content/BroadcastReceiver.html).

   KOTLIN JAVA

   ```
   BroadcastReceiver br = new MyBroadcastReceiver();
   ```

2. Create an `IntentFilter` (https://developer.android.com/reference/android/content/IntentFilter.html) and register the receiver by calling `registerReceiver(BroadcastReceiver, IntentFilter)` (https://developer.android.com/reference/android/content/Context.html#registerReceiver(android.content.BroadcastReceiver, android.content.IntentFilter)):

   KOTLIN JAVA

   ```
   IntentFilter filter = new IntentFilter(ConnectivityManager.CONNECTIVITY_AC
       filter.addAction(Intent.ACTION_AIRPLANE_MODE_CHANGED);
   ```

```
    this.registerReceiver(br, filter);
```

Note: To register for local broadcasts, call `LocalBroadcastManager.registerReceiver(BroadcastReceiver, IntentFilter)` (https://developer.android.com/reference/android/support/v4/content/LocalBroadcastManager.html#registerReceiver(android.content.BroadcastReceiver, android.content.IntentFilter)) instead.

Context-registered receivers receive broadcasts as long as their registering context is valid. For an example, if you register within an `Activity` (https://developer.android.com/reference/android/app/Activity.html) context, you receive broadcasts as long as the activity is not destroyed. If you register with the Application context, you receive broadcasts as long as the app is running.

3. To stop receiving broadcasts, call `unregisterReceiver(android.content.BroadcastReceiver)` (https://developer.android.com/reference/android/content/Context.html#unregisterReceiver(android.content.BroadcastReceiver)). Be sure to unregister the receiver when you no longer need it or the context is no longer valid.

   Be mindful of where you register and unregister the receiver, for example, if you register a receiver in `onCreate(Bundle)` (https://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)) using the activity's context, you should unregister it in `onDestroy()` (https://developer.android.com/reference/android/app/Activity.html#onDestroy()) to prevent leaking the receiver out of the activity context. If you register a receiver in `onResume()` (https://developer.android.com/reference/android/app/Activity.html#onResume()), you should unregister it in `onPause()` (https://developer.android.com/reference/android/app/Activity.html#onPause()) to prevent registering it multiple times (If you don't want to receive broadcasts when paused, and this can cut down on unnecessary system overhead). Do not unregister in `onSaveInstanceState(Bundle)` (https://developer.android.com/reference/android/app/Activity.html#onSaveInstanceState(android.os.Bundle)), because this isn't called if the user moves back in the history stack.

Effects on process state

The state of your `BroadcastReceiver` (https://developer.android.com/reference/android/content/BroadcastReceiver.html) (whether it is running or not) affects the state of its containing process, which can in turn affect its likelihood of being killed by the system. For example, when a process executes a receiver (that is, currently running the code in its `onReceive()` (https://developer.android.com/reference/android/content/BroadcastReceiver.html#onReceive(android.content.Context, android.content.Intent)) method), it is considered to be a foreground process. The system keeps the process running except under cases of extreme memory pressure.

However, once your code returns from `onReceive()`, the BroadcastReceiver is no longer active. The receiver's host process becomes only as important as the other app components that are running in it. If that process hosts only a manifest-declared receiver (a common case for apps that the user has never or not recently interacted with), then upon returning from `onReceive()`, the system considers its process to be a low-priority process and may kill it to make resources available for other more important processes.

For this reason, you should not start long running background threads from a broadcast receiver. After `onReceive()`, the system can kill the process at any time to reclaim memory, and in doing so, it terminates the spawned thread running in the process. To avoid this, you should either call `goAsync()` (https://developer.android.com/reference/android/content/BroadcastReceiver.html#goAsync()) (if you want a little more time to process the broadcast in a background thread) or schedule a `JobService` (https://developer.android.com/reference/android/app/job/JobService.html) from the receiver using the `JobScheduler` (https://developer.android.com/reference/android/app/job/JobScheduler.html), so the system knows that the process continues to perform active work. For more information, see Processes and Application Life Cycle (https://developer.android.com/guide/topics/processes/process-lifecycle.html).

The following snippet shows a `BroadcastReceiver` (https://developer.android.com/reference/android/content/BroadcastReceiver.html) that uses `goAsync()` (https://developer.android.com/reference/android/content/BroadcastReceiver.html#goAsync()) to flag that it needs more time to finish after `onReceive()` is complete. This is especially useful if the work you want to complete in your `onReceive()` is long enough to cause the UI thread to miss a frame (>16ms), making it better suited for a background thread.

KOTLIN JAVA

```
public class MyBroadcastReceiver extends BroadcastReceiver {
    private static final String TAG = "MyBroadcastReceiver";

    @Override
    public void onReceive(Context context, Intent intent) {
        final PendingResult pendingResult = goAsync();
        Task asyncTask = new Task(pendingResult, intent);
        asyncTask.execute();
    }

    private static class Task extends AsyncTask {

        private final PendingResult pendingResult;
        private final Intent intent;

        private Task(PendingResult pendingResult, Intent intent) {
            this.pendingResult = pendingResult;
            this.intent = intent;
        }

        @Override
        protected String doInBackground(String... strings) {
            StringBuilder sb = new StringBuilder();
            sb.append("Action: " + intent.getAction() + "\n");
            sb.append("URI: " + intent.toUri(Intent.URI_INTENT_SCHEME).toString
            String log = sb.toString();
            Log.d(TAG, log);
            return log;
        }

        @Override
        protected void onPostExecute(String s) {
            super.onPostExecute(s);
            // Must call finish() so the BroadcastReceiver can be recycled.
            pendingResult.finish();
        }
    }
}
```

Sending broadcasts

Android provides three ways for apps to send broadcast:

- The `sendOrderedBroadcast(Intent, String)` (https://developer.android.com/reference/android/content/Context.html#sendOrderedBroadcast(android.content.Intent, java.lang.String)) method sends broadcasts to one receiver at a time. As each receiver executes in turn, it can propagate a result to the next receiver, or it can completely abort the broadcast so that it won't be passed to other receivers. The order receivers run in can be controlled with the android:priority attribute of the matching intent-filter; receivers with the same priority will be run in an arbitrary order.
- The `sendBroadcast(Intent)` (https://developer.android.com/reference/android/content/Context.html#sendBroadcast(android.content.Intent)) method sends broadcasts to all receivers in an undefined order. This is called a Normal Broadcast. This is more efficient, but means that receivers cannot read results from other receivers, propagate data received from the broadcast, or abort the broadcast.
- The `LocalBroadcastManager.sendBroadcast` (https://developer.android.com/reference/android/support/v4/content/LocalBroadcastManager.html#sendBroadcast(android.content.Intent)) method sends broadcasts to receivers that are in the same app as the sender. If you don't need to send broadcasts across apps, use local broadcasts. The implementation is much more efficient (no interprocess communication needed) and you don't need to worry about any security issues related to other apps being able to receive or send your broadcasts.

The following code snippet demonstrates how to send a broadcast by creating an Intent and calling `sendBroadcast(Intent)` (https://developer.android.com/reference/android/content/Context.html#sendBroadcast(android.content.Intent))

.

KOTLIN JAVA

```
Intent intent = new Intent();
intent.setAction("com.example.broadcast.MY_NOTIFICATION");
intent.putExtra("data","Notice me senpai!");
sendBroadcast(intent);
```

The broadcast message is wrapped in an `Intent` (https://developer.android.com/reference/android/content/Intent.html) object. The intent's action string must provide the app's Java package name syntax and uniquely identify the broadcast

event. You can attach additional information to the intent with `putExtra(String, Bundle)` (https://developer.android.com/reference/android/content/Intent.html#putExtra(java.lang.String, android.os.Bundle)). You can also limit a broadcast to a set of apps in the same organization by calling `setPackage(String)` (https://developer.android.com/reference/android/content/Intent.html#setPackage(java.lang.String)) on the intent.

Note: Although intents are used for both sending broadcasts and starting activities with `startActivity(Intent)` (https://developer.android.com/reference/android/content/Context.html#startActivity(android.content.Intent)), these actions are completely unrelated. Broadcast receivers can't see or capture intents used to start an activity; likewise, when you broadcast an intent, you can't find or start an activity.

Restricting broadcasts with permissions

Permissions allow you to restrict broadcasts to the set of apps that hold certain permissions. You can enforce restrictions on either the sender or receiver of a broadcast.

Sending with permissions

When you call `sendBroadcast(Intent, String)` (https://developer.android.com/reference/android/content/Context.html#sendBroadcast(android.content.Intent, java.lang.String)) or `sendOrderedBroadcast(Intent, String, BroadcastReceiver, Handler, int, String, Bundle)` (https://developer.android.com/reference/android/content/Context.html#sendOrderedBroadcast(android.content.Intent, java.lang.String, android.content.BroadcastReceiver, android.os.Handler, int, java.lang.String, android.os.Bundle)), you can specify a permission parameter. Only receivers who have requested that permission with the tag in their manifest (and subsequently been granted the permission if it is dangerous) can receive the broadcast. For example, the following code sends a broadcast:

KOTLIN JAVA

```
sendBroadcast(new Intent("com.example.NOTIFY"),
              Manifest.permission.SEND_SMS);
```

To receive the broadcast, the receiving app must request the permission as shown below:

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

You can specify either an existing system permission like `SEND_SMS` (https://developer.android.com/reference/android/Manifest.permission.html#SEND_SMS) or define a custom permission with the `<permission>` (https://developer.android.com/guide/topics/manifest/permission-element.html) element. For information on permissions and security in general, see the System Permissions (https://developer.android.com/guide/topics/security/permissions.html).

Note: Custom permissions are registered when the app is installed. The app that defines the custom permission must be installed before the app that uses it.

Receiving with permissions

If you specify a permission parameter when registering a broadcast receiver (either with `registerReceiver(BroadcastReceiver, IntentFilter, String, Handler)` (https://developer.android.com/reference/android/content/Context.html#registerReceiver(android.content.BroadcastReceiver, android.content.IntentFilter, java.lang.String, android.os.Handler)) or in `<receiver>` (https://developer.android.com/guide/topics/manifest/receiver-element.html) tag in your manifest), then only broadcasters who have requested the permission with the `<uses-permission>` (https://developer.android.com/guide/topics/manifest/uses-permission-element.html) tag in their manifest (and subsequently been granted the permission if it is dangerous) can send an Intent to the receiver.

For example, assume your receiving app has a manifest-declared receiver as shown below:

```
<receiver android:name=".MyBroadcastReceiver"
          android:permission="android.permission.SEND_SMS">
    <intent-filter>
        <action android:name="android.intent.action.AIRPLANE_MODE"/>
    </intent-filter>
</receiver>
```

Or your receiving app has a context-registered receiver as shown below:

KOTLIN JAVA

```
IntentFilter filter = new IntentFilter(Intent.ACTION_AIRPLANE_MODE_CHANGED);
registerReceiver(receiver, filter, Manifest.permission.SEND_SMS, null );
```

Then, to be able to send broadcasts to those receivers, the sending app must request the permission as shown below:

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

Security considerations and best practices

Here are some security considerations and best practices for sending and receiving broadcasts:

- If you don't need to send broadcasts to components outside of your app, then send and receive local broadcasts with the `LocalBroadcastManager` (https://developer.android.com/reference/android/support/v4/content/LocalBroadcastManager.html) which is available in the Support Library (https://developer.android.com/topic/libraries/support-library/index.html). The `LocalBroadcastManager` (https://developer.android.com/reference/android/support/v4/content/LocalBroadcastManager.html) is much more efficient (no interprocess communication needed) and allows you to avoid thinking about any security issues related to other apps being able to receive or send your broadcasts. Local Broadcasts can be used as a general purpose pub/sub event bus in your app without any overheads of system wide broadcasts.
- If many apps have registered to receive the same broadcast in their manifest, it can cause the system to launch a lot of apps, causing a substantial impact on both device performance and user experience. To avoid this, prefer using context registration over manifest declaration. Sometimes, the Android system itself enforces the use of context-registered receivers. For example, the `CONNECTIVITY_ACTION` (https://developer.android.com/reference/android/net/ConnectivityManager.html#CONNECTIVITY_ACTION) broadcast is delivered only to context-registered receivers.

- Do not broadcast sensitive information using an implicit intent. The information can be read by any app that registers to receive the broadcast. There are three ways to control who can receive your broadcasts:
    - You can specify a permission when sending a broadcast.
    - In Android 4.0 and higher, you can specify a package (https://developer.android.com/guide/topics/manifest/manifest-element.html#package) with `setPackage(String)` (https://developer.android.com/reference/android/content/Intent.html#setPackage(java.lang.String)) when sending a broadcast. The system restricts the broadcast to the set of apps that match the package.
    - You can send local broadcasts with `LocalBroadcastManager` (https://developer.android.com/reference/android/support/v4/content/LocalBroadcastManager.html) .
- When you register a receiver, any app can send potentially malicious broadcasts to your app's receiver. There are three ways to limit the broadcasts that your app receives:
    - You can specify a permission when registering a broadcast receiver.
    - For manifest-declared receivers, you can set the android:exported (https://developer.android.com/guide/topics/manifest/receiver-element.html#exported) attribute to "false" in the manifest. The receiver does not receive broadcasts from sources outside of the app.
    - You can limit yourself to only local broadcasts with `LocalBroadcastManager` (https://developer.android.com/reference/android/support/v4/content/LocalBroadcastManager.html) .
- The namespace for broadcast actions is global. Make sure that action names and other strings are written in a namespace you own, or else you may inadvertently conflict with other apps.
- Because a receiver's `onReceive(Context, Intent)` (https://developer.android.com/reference/android/content/BroadcastReceiver.html#onReceive(android.content.Context, android.content.Intent)) method runs on the main thread, it should execute and return quickly. If you need to perform long running work, be careful about spawning threads or starting background services because the system can kill the entire process after `onReceive()` returns. For

more information, see Effect on process state (#effects-on-process-state) To perform long running work, we recommend:

- Calling `goAsync()` (https://developer.android.com/reference/android/content/BroadcastReceiver.html#goAsync()) in your receiver's `onReceive()` method and passing the `BroadcastReceiver.PendingResult` (https://developer.android.com/reference/android/content/BroadcastReceiver.PendingResult.html) to a background thread. This keeps the broadcast active after returning from `onReceive()`. However, even with this approach the system expects you to finish with the broadcast very quickly (under 10 seconds). It does allow you to move work to another thread to avoid glitching the main thread.
- Scheduling a job with the `JobScheduler` (https://developer.android.com/reference/android/app/job/JobScheduler.html). For more information, see Intelligent Job Scheduling (https://developer.android.com/topic/performance/scheduling.html).

- Do not start activities from broadcast receivers because the user experience is jarring; especially if there is more than one receiver. Instead, consider displaying a notification (https://developer.android.com/guide/topics/ui/notifiers/notifications.html).

Last updated September 27, 2018.