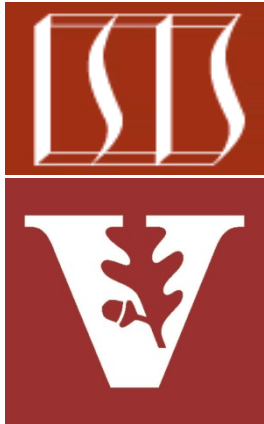


Android Services & Local IPC: Introduction

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

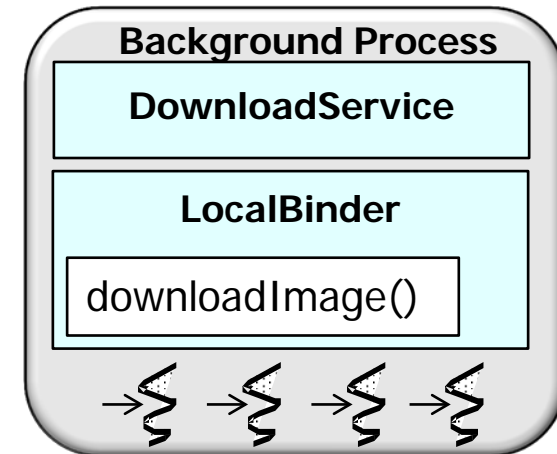
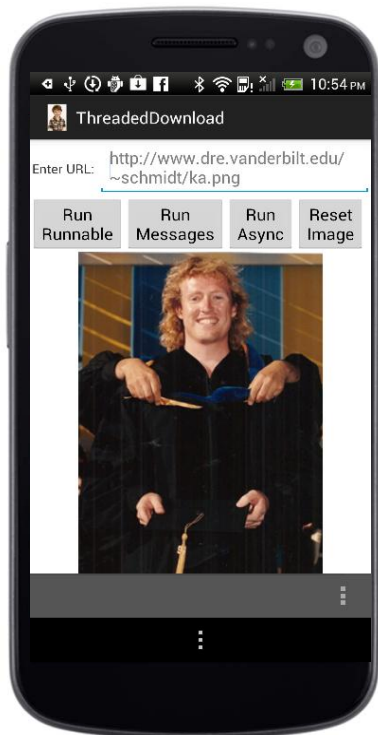
Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



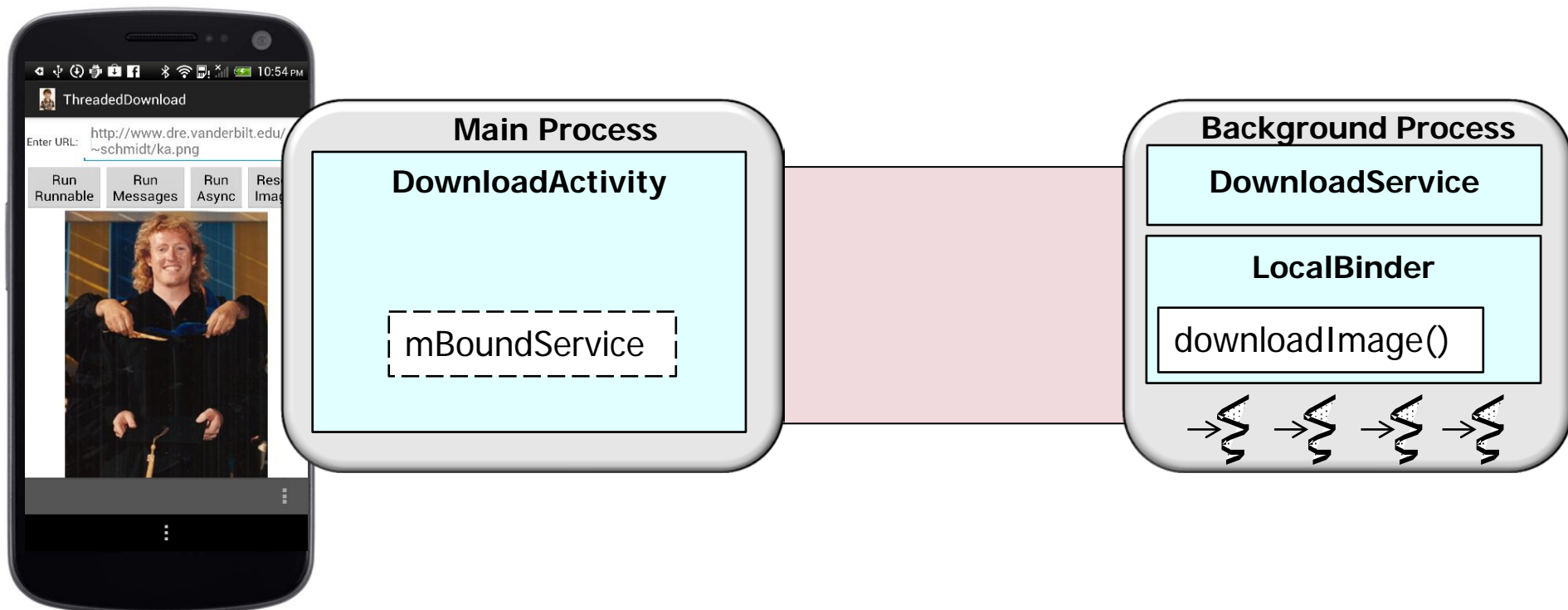
Introduction

- Services don't have a visual user interface & often run in the background in a separate background thread or process



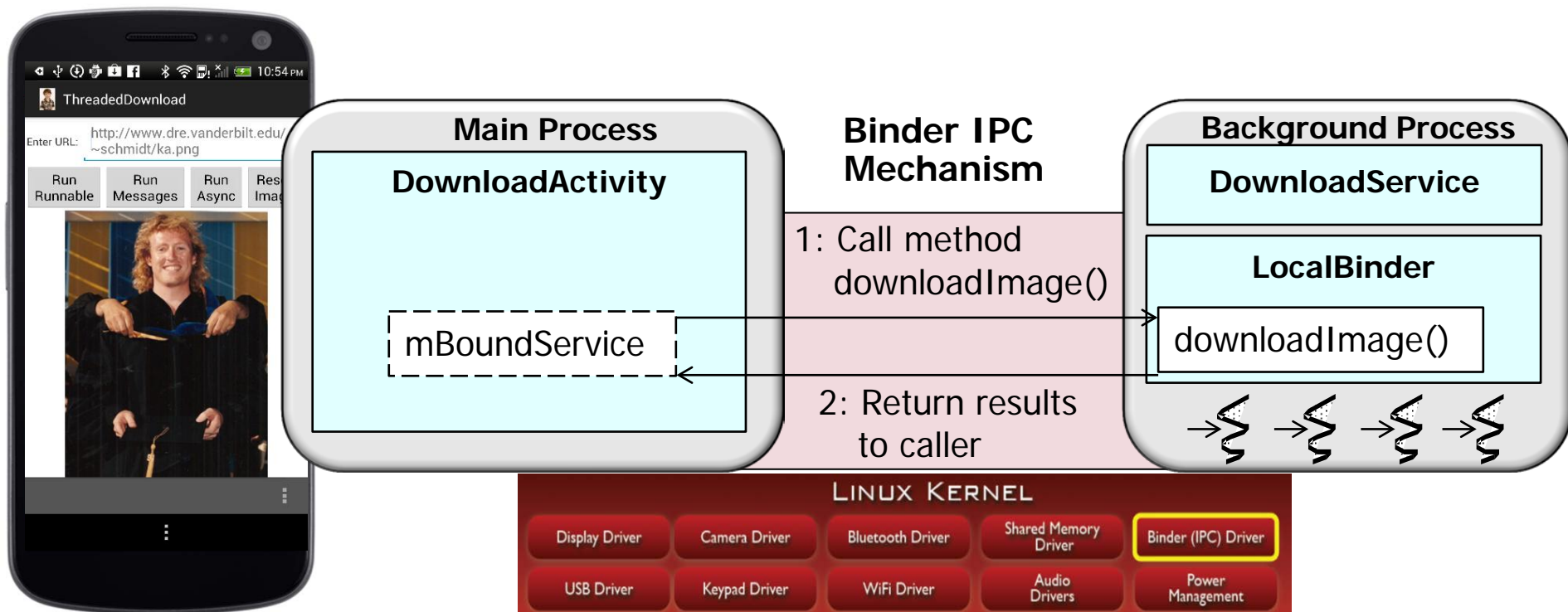
Introduction

- Services don't have a visual user interface & often run in the background in a separate background thread or process
- Activities use Services to perform long-running operations or access remote resources on behalf of users



Introduction

- Services don't have a visual user interface & often run in the background in a separate background thread or process
- Activities & Services interact via IPC mechanisms that are optimized for inter-process communication within a mobile device
 - e.g., the Android Interface Definition Language (AIDL) & Binder framework



Android Services & Local IPC:

Overview of Services

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

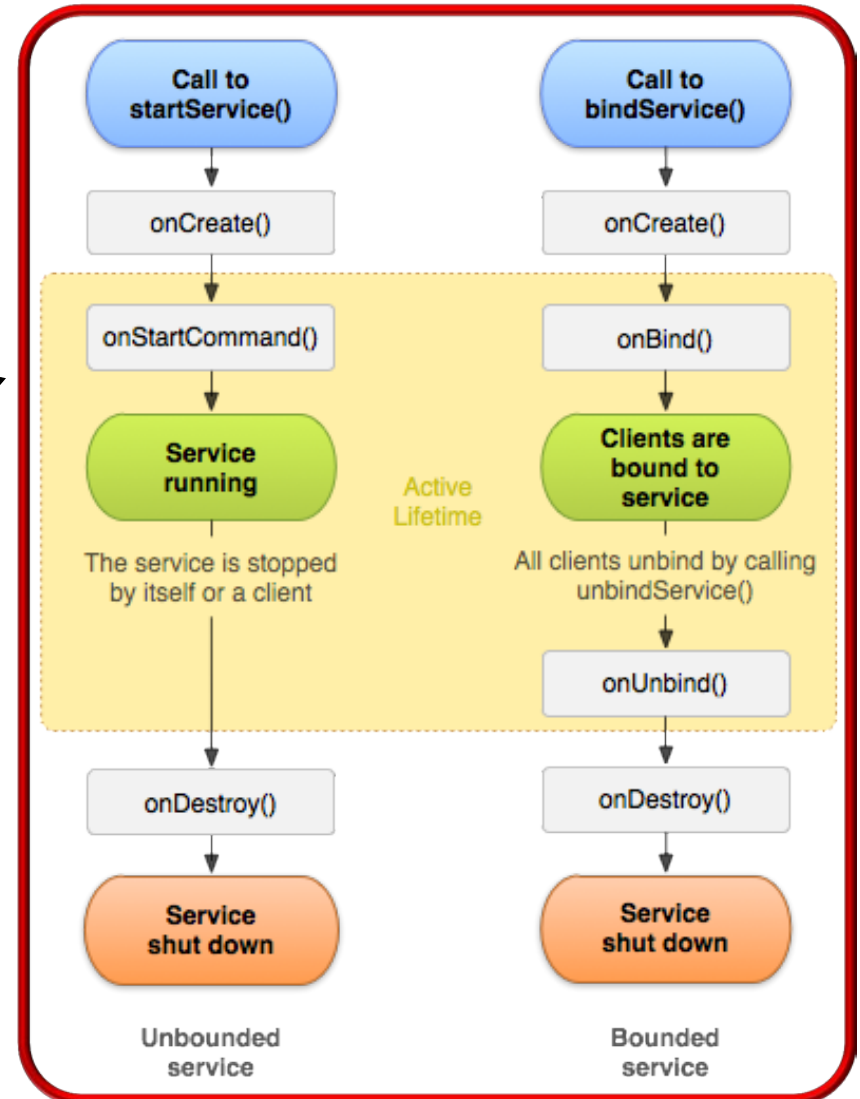
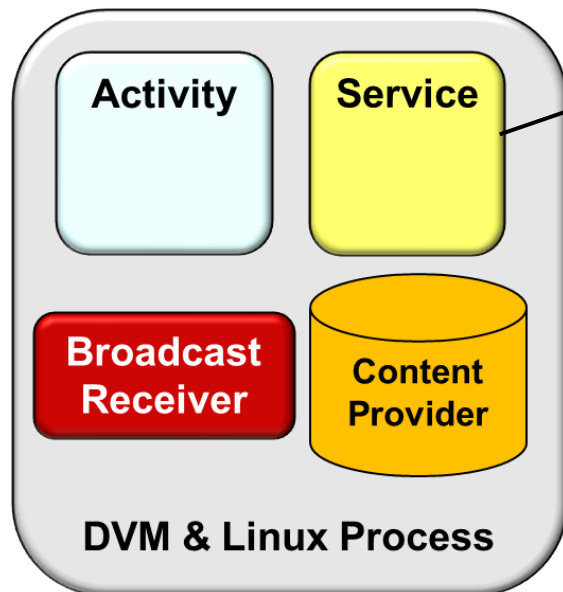
Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



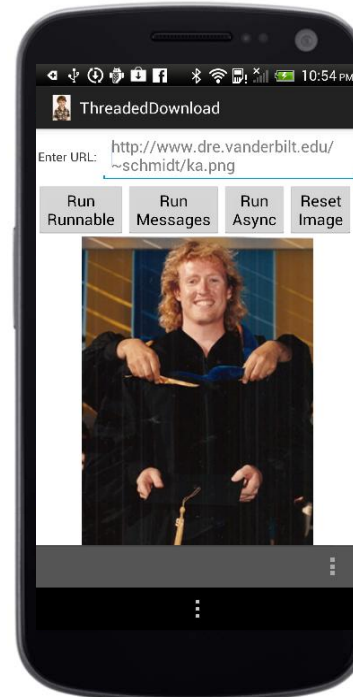
Learning Objectives in this Part of the Module

- Understand what a Service is & what different types of Services Android supports



Overview of a Service

- A Service is an Android component that can perform long-running operations in the background
- e.g., a service might handle e-commerce transactions, play music, **download a file**, interact with a content provider, run tasks periodically, etc.



Download Service

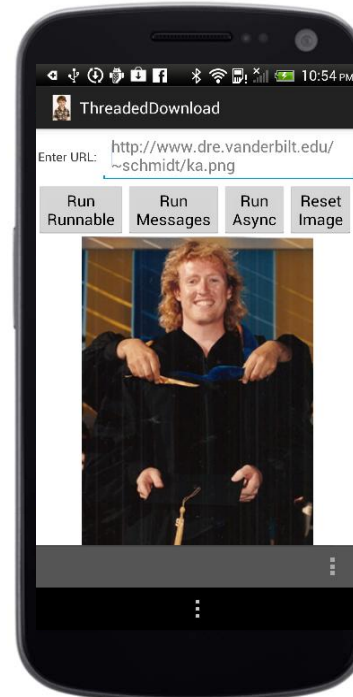
Overview of a Service

- A Service is an Android component that can perform long-running operations in the background
- Another Android component can start a Service
 - It will continue to run in the background even if the user switches to another app/activity

A Service does not provide direct access to the user interface

Download Activity

Download Service

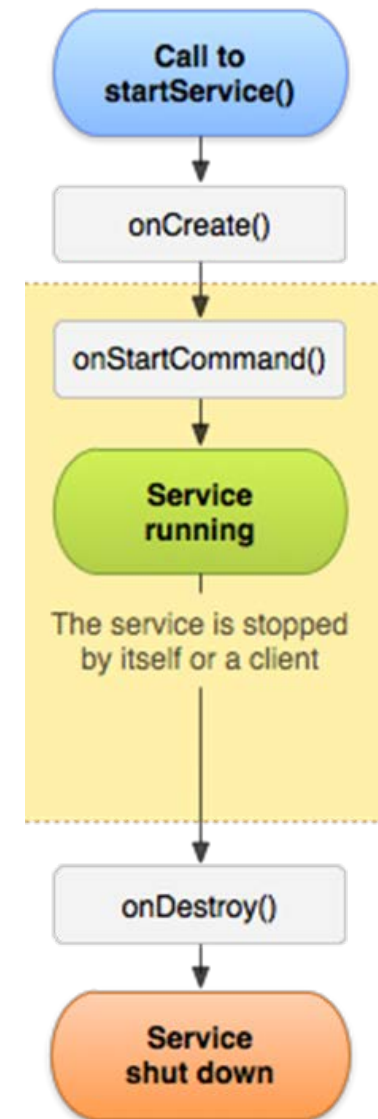
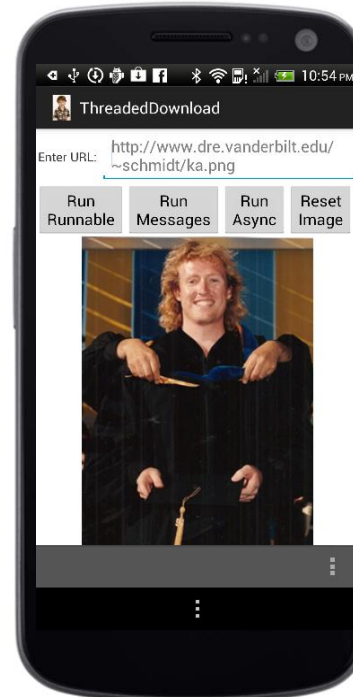


Overview of a Service

- A Service is an Android component that can perform long-running operations in the background
- Another Android component can start a service
- There are two types of Services
 - *Started Service* – Often performs a single operation & might not return a result to the caller directly

Download Activity

Download Service

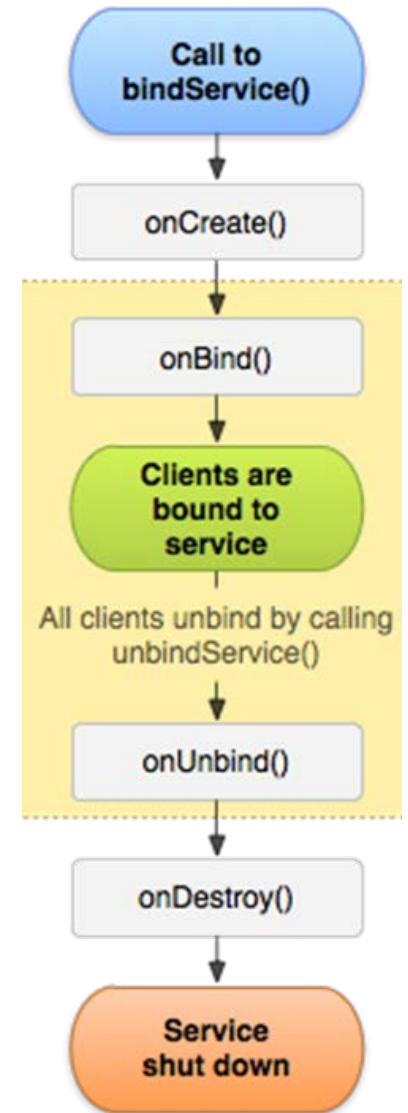
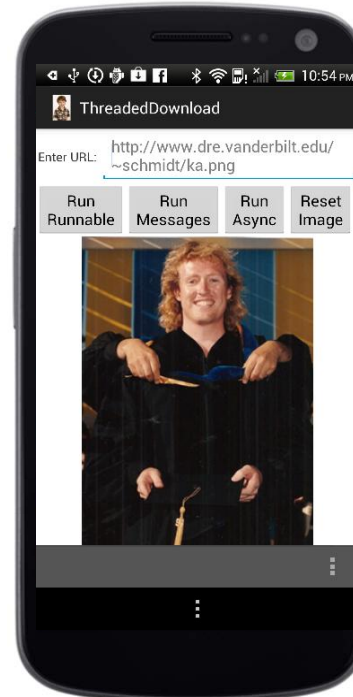


Overview of a Service

- A Service is an Android component that can perform long-running operations in the background
- Another Android component can start a service
- There are two types of Services
 - *Started Service* – Often performs a single operation & might not return a result to the caller directly
 - *Bound Service* – Provides a client-server interface that allows for a conversation with the Service

Download Activity

Download Service



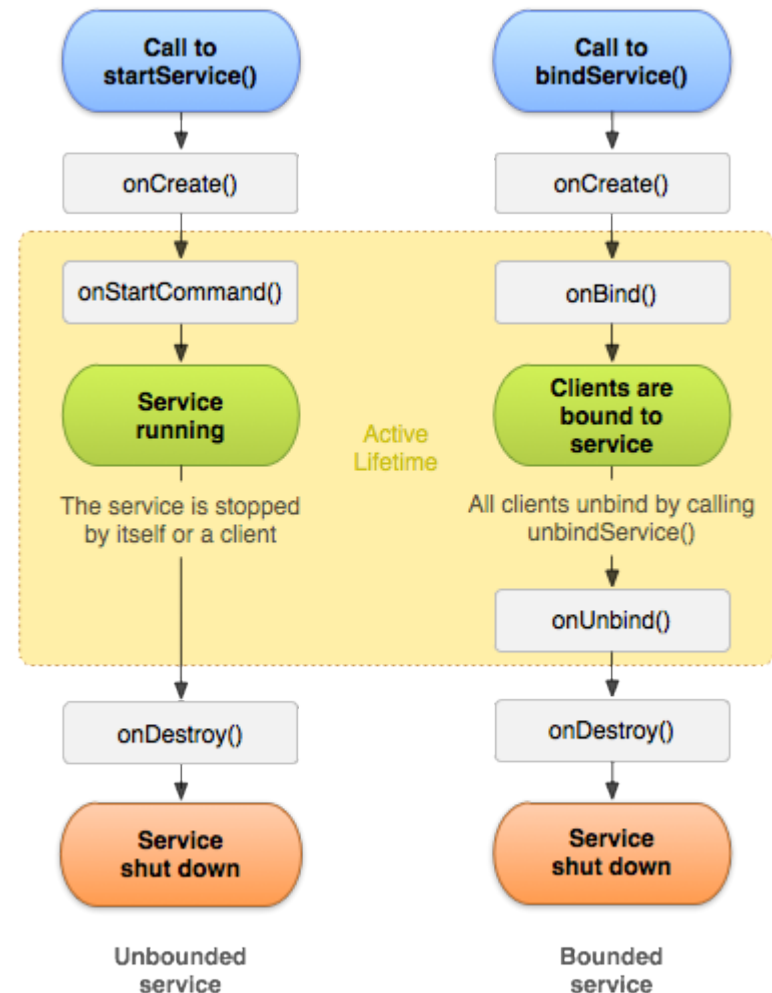
Implementing a Service

- Implementing a Service is similar to implementing an Activity
- e.g., inherit from Android Service class, override lifecycle methods, include Service in the config file AndroidManifest.xml, etc.

```
public class Service extends
    ... {
    public void onCreate();
    public int onStartCommand
        (Intent intent,
         int flags, int startId);
    public abstract IBinder
        onBind(Intent intent);
    public boolean
        onUnbind(Intent intent);
    protected void onDestroy();
    ...
}
```

Implementing a Service

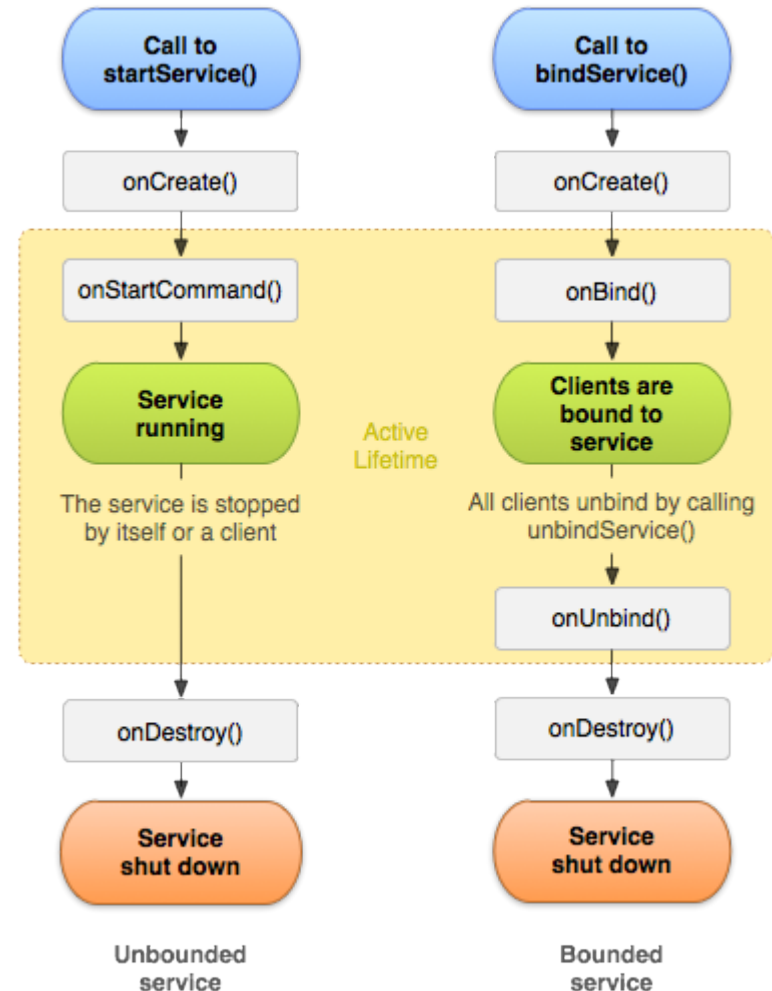
- Implementing a Service is similar to implementing an Activity
- Android communicates state changes to a Service by calling its lifecycle hook methods



Implementing a Service

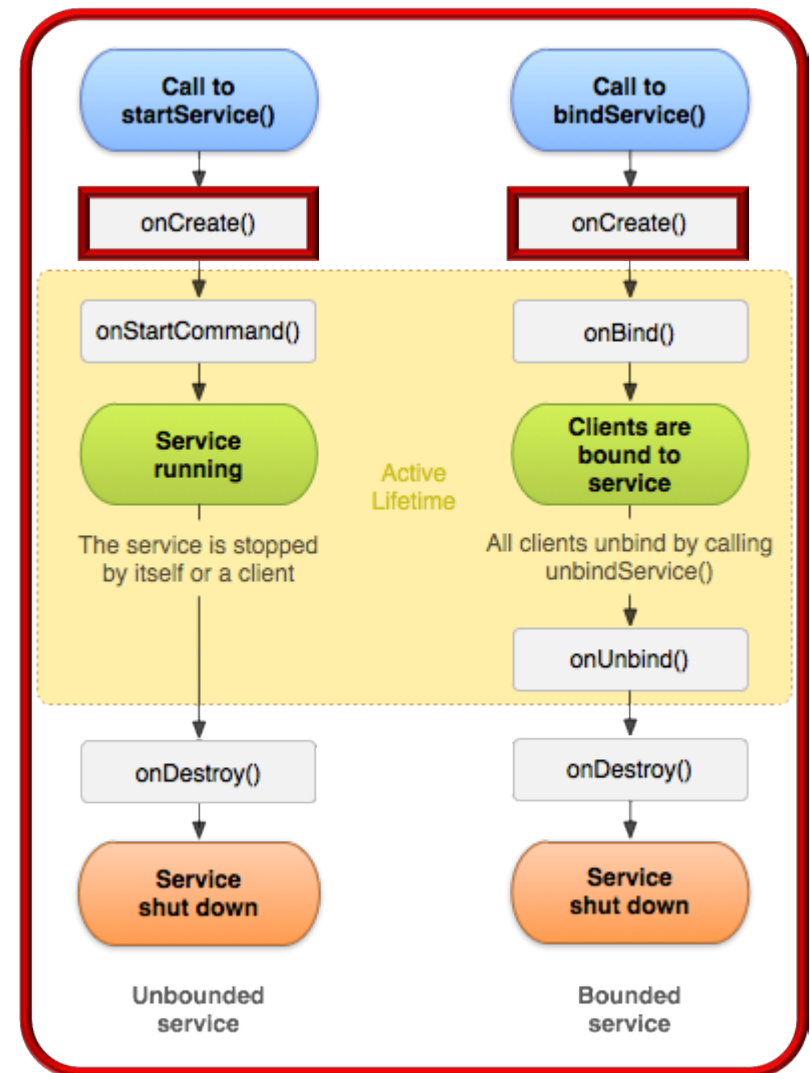
- Implementing a Service is similar to implementing an Activity
- Android communicates state changes to a Service by calling its lifecycle hook methods

- Commonality:** Provides common interface for performing long-running operations that don't interact directly with the user interface
- Variability:** Subclasses can override lifecycle hook methods to perform necessary initialization for *Started* & *Bound* Services



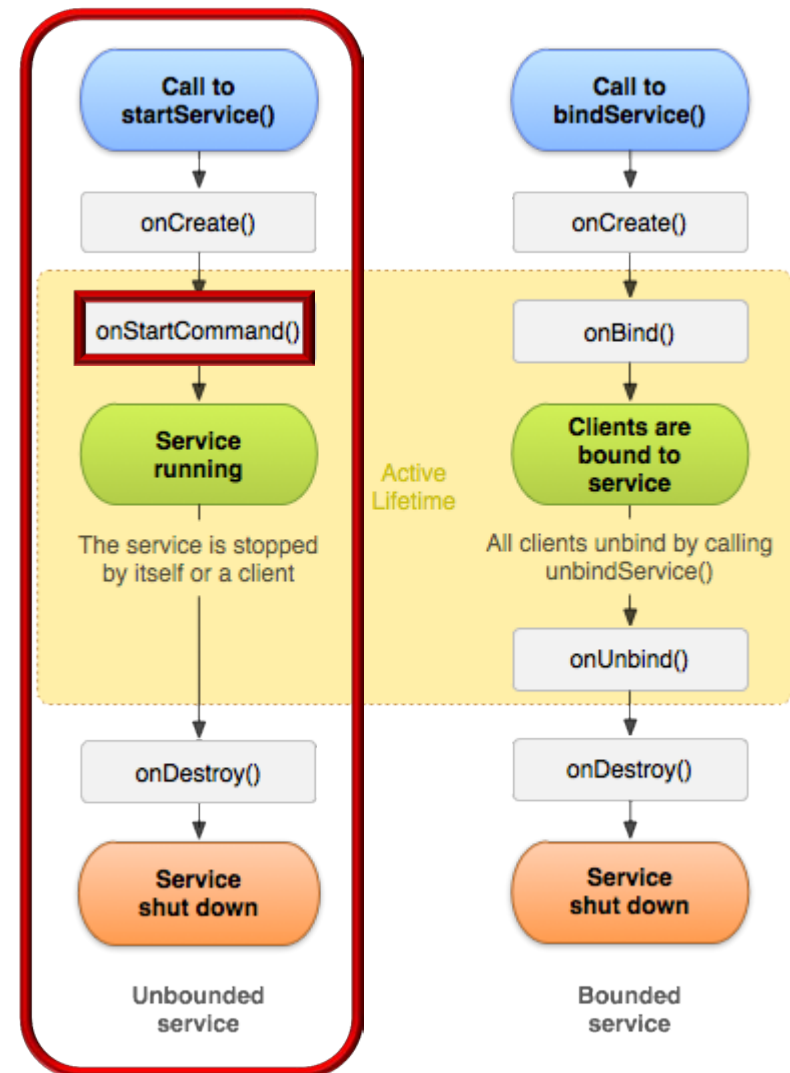
Service Lifecycle Hook Methods

- Services lifecycle methods include
 - onCreate()** – called when Service process is created, by any means



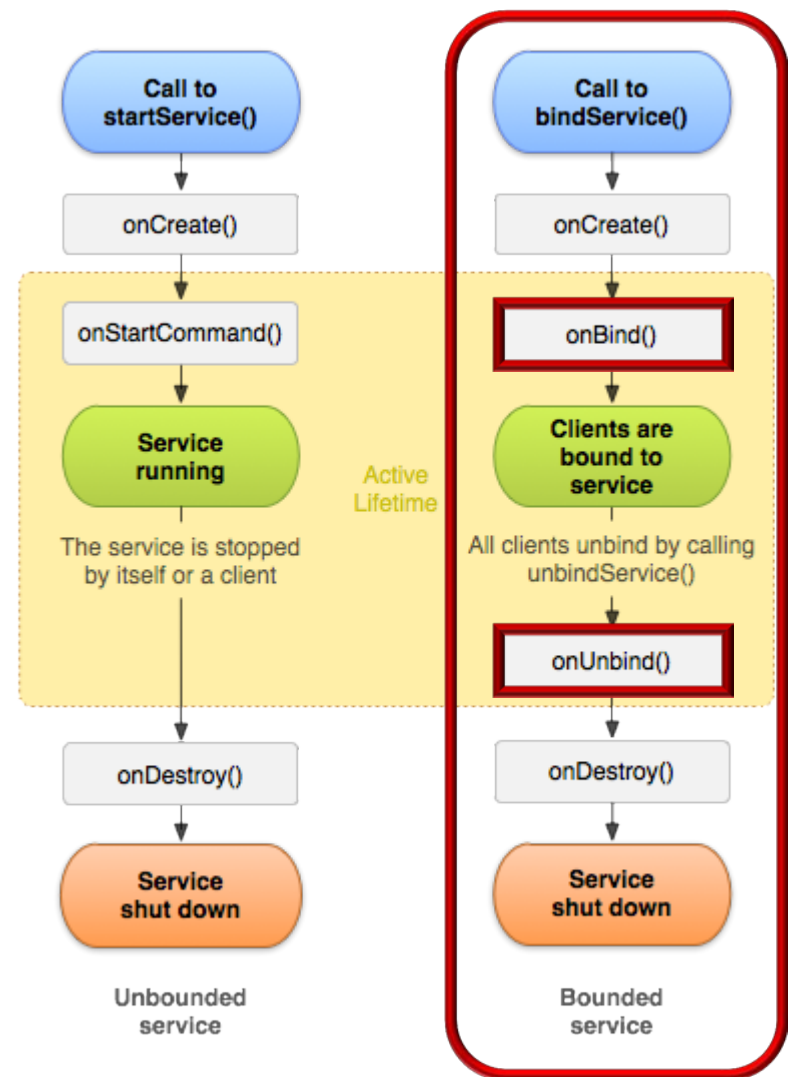
Service Lifecycle Hook Methods

- Services lifecycle methods include
 - onCreate()** – called when Service process is created, by any means
 - onStartCommand()** – called each time a Started Service is sent a command via startService()



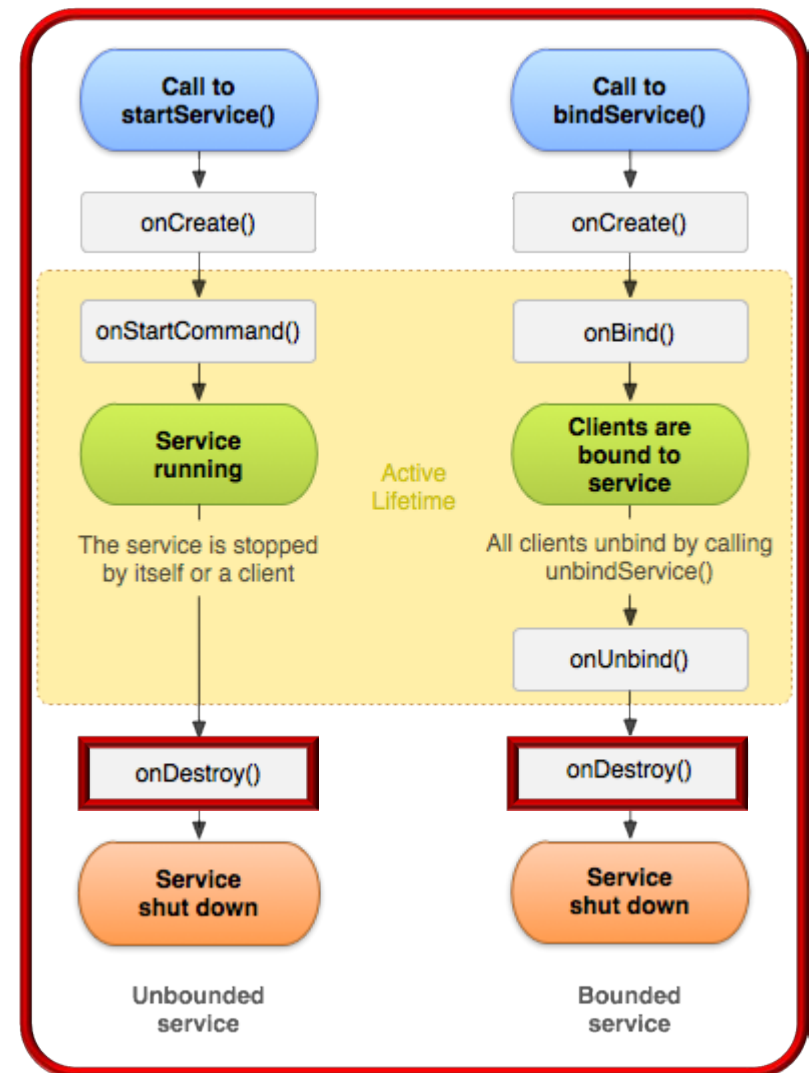
Service Lifecycle Hook Methods

- Services lifecycle methods include
 - onCreate()** – called when Service process is created, by any means
 - onStartCommand()** – called each time a Started Service is sent a command via `startService()`
 - onBind()/onUnbind** – called when a client binds/unbinds to a Bound Service via `bindService()/unBindService()`



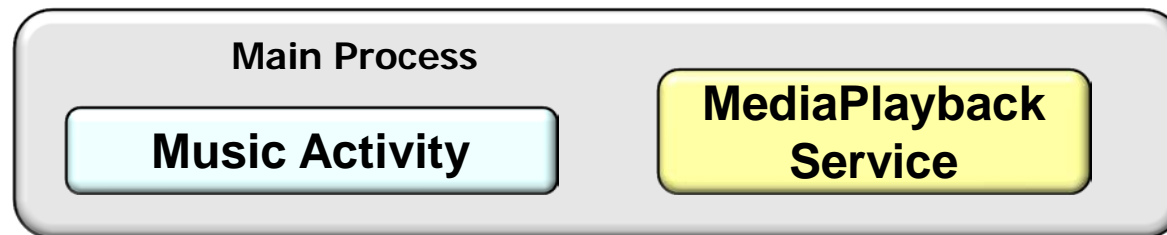
Service Lifecycle Hook Methods

- Services lifecycle methods include
 - onCreate()** – called when Service process is created, by any means
 - onStartCommand()** – called each time a Started Service is sent a command via `startService()`
 - onBind()/onUnbind** – called when a client binds/unbinds to a Bound Service via `bindService()/unBindService()`
 - onDestroy()** – called as Service is being shut down to cleanup resources



Configuring a Service into the Android System

- You need to add a Service to your AndroidManifest.xml file
- Add a <service> element as a child of the <application> element & provide android:name to reference your Service class

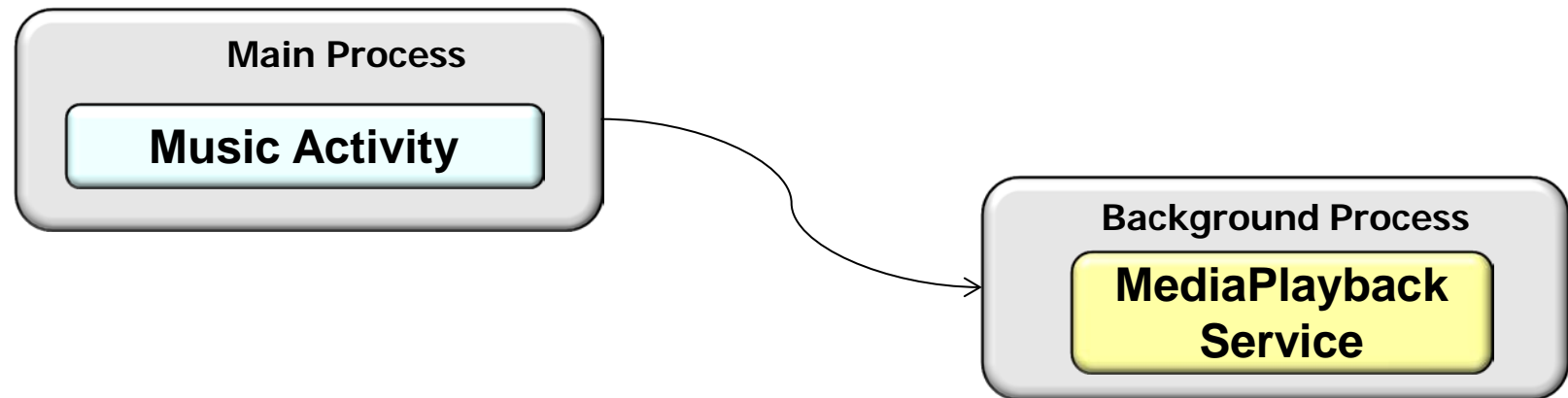


Music Service

```
<service android:name="com.android.music.MediaPlaybackService"
        android:exported="false"/>
```

Configuring a Service into the Android System

- You need to add a Service to your AndroidManifest.xml file
 - Add a `<service>` element as a child of the `<application>` element & provide `android:name` to reference your Service class
 - Use `android:process=":myProcess"` to run the service in its own process

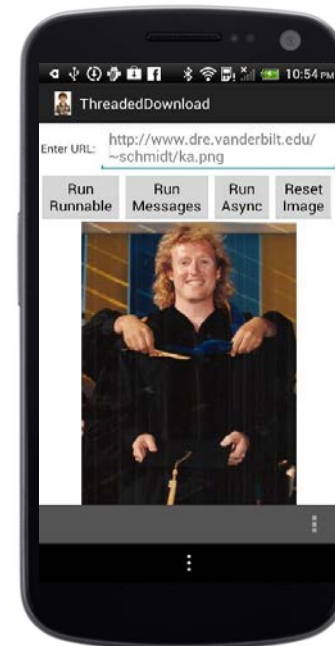
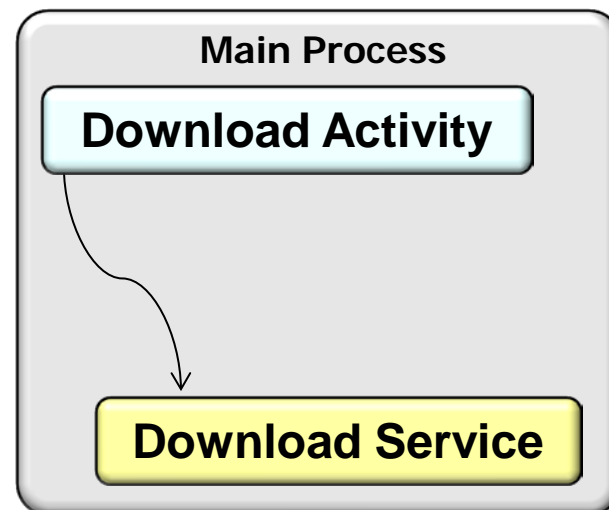


Music Service

```
<service android:name="com.android.music.MediaPlaybackService"  
    android:exported="false"  
    android:process=":myProcess" />
```

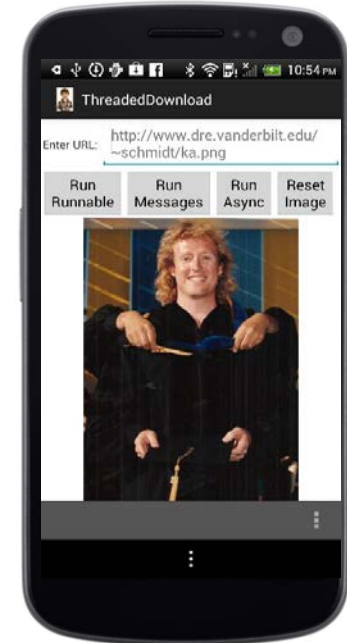
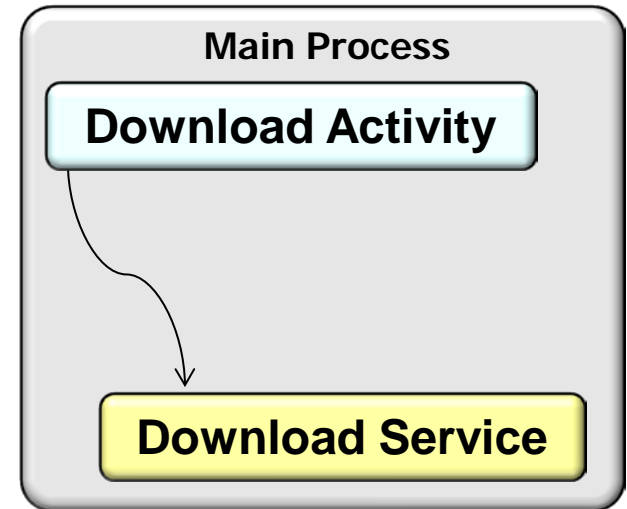
Summary

- Apps can use Services to implement long-running operations in the background
- Unless otherwise specified, a Service runs in the same process/thread as the app it is part of



Summary

- Apps can use Services to implement long-running operations in the background
 - Unless otherwise specified, a Service runs in the same process/thread as the app it is part of
- It keeps running until stopped by itself, stopped by user, or killed by the system if it needs memory



Summary

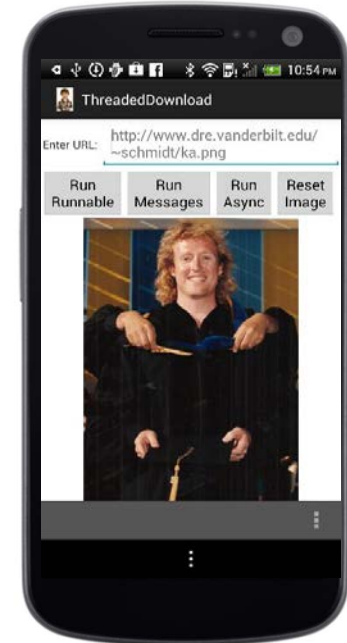
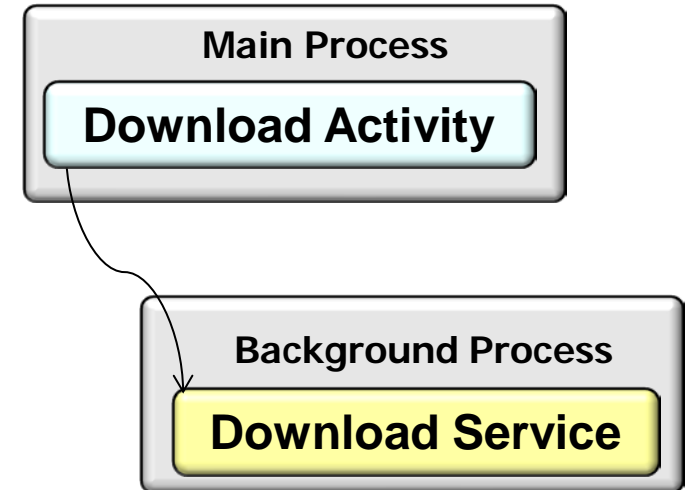
- Apps can use Services to implement long-running operations in the background
- You can configure many properties of Services via an AndroidManifest.xml file

```
<service
```

```
    android:enabled=["true" | "false"]  
    android:exported=["true" | "false"]  
    android:icon="drawable resource"  
    android:isolatedProcess=["true" | "false"]  
    android:label="string resource"  
    android:name="string"  
    android:permission="string"  
    android:process="string" >
```

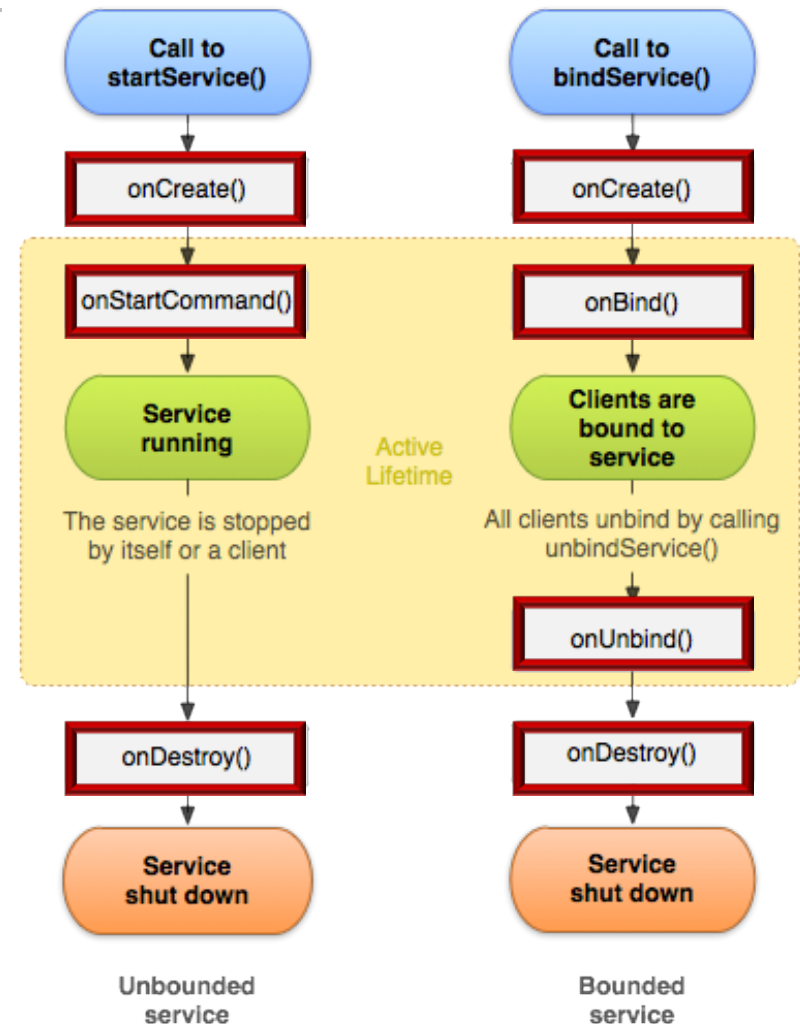
```
    ...
```

```
</service>
```



Summary

- Apps can use Services to implement long-running operations in the background
- You can configure many properties of Services via an AndroidManifest.xml file
- Android calls back on hook methods to control Service processing



Android Services & Local IPC:

Overview of Communicating with Services

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

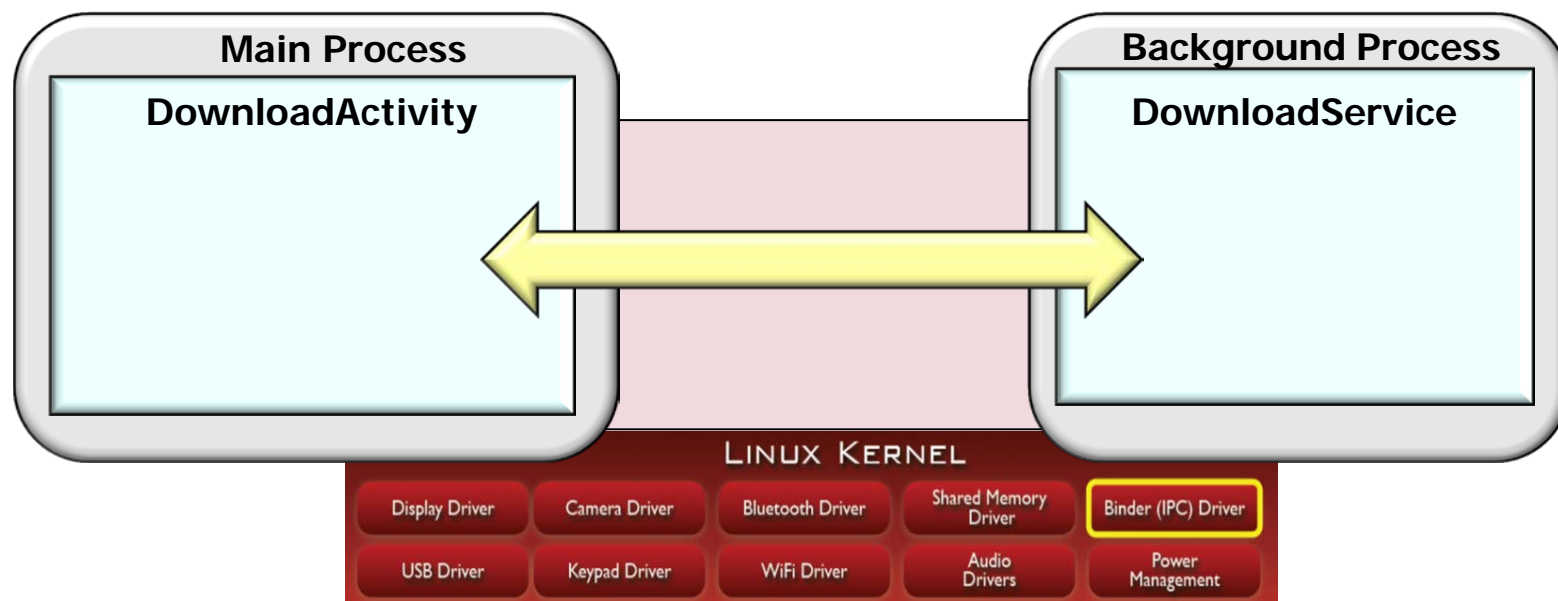
Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



Learning Objectives in this Part of the Module

- Understand various local IPC mechanisms that Activities & Services use to communicate

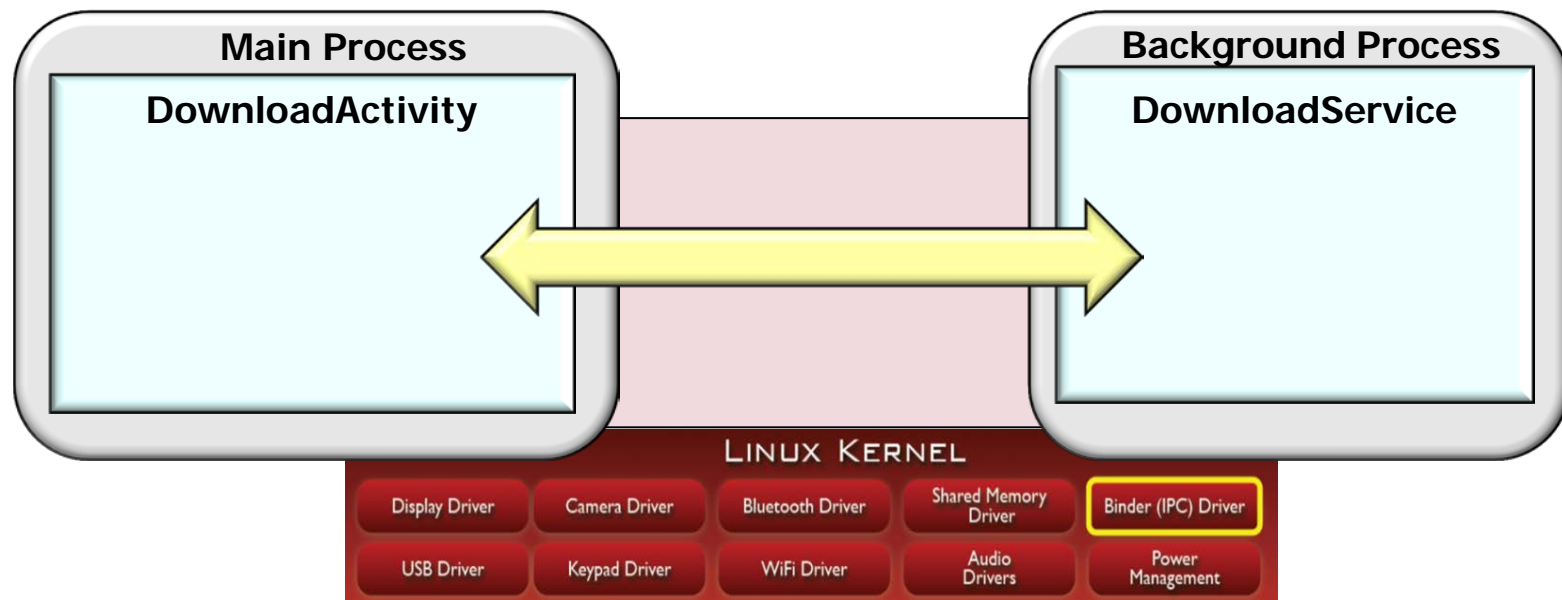


Learning Objectives in this Part of the Module

- Understand various local IPC mechanisms that Activities & Services use to communicate
- Recognize the common patterns used to implement communication with Services

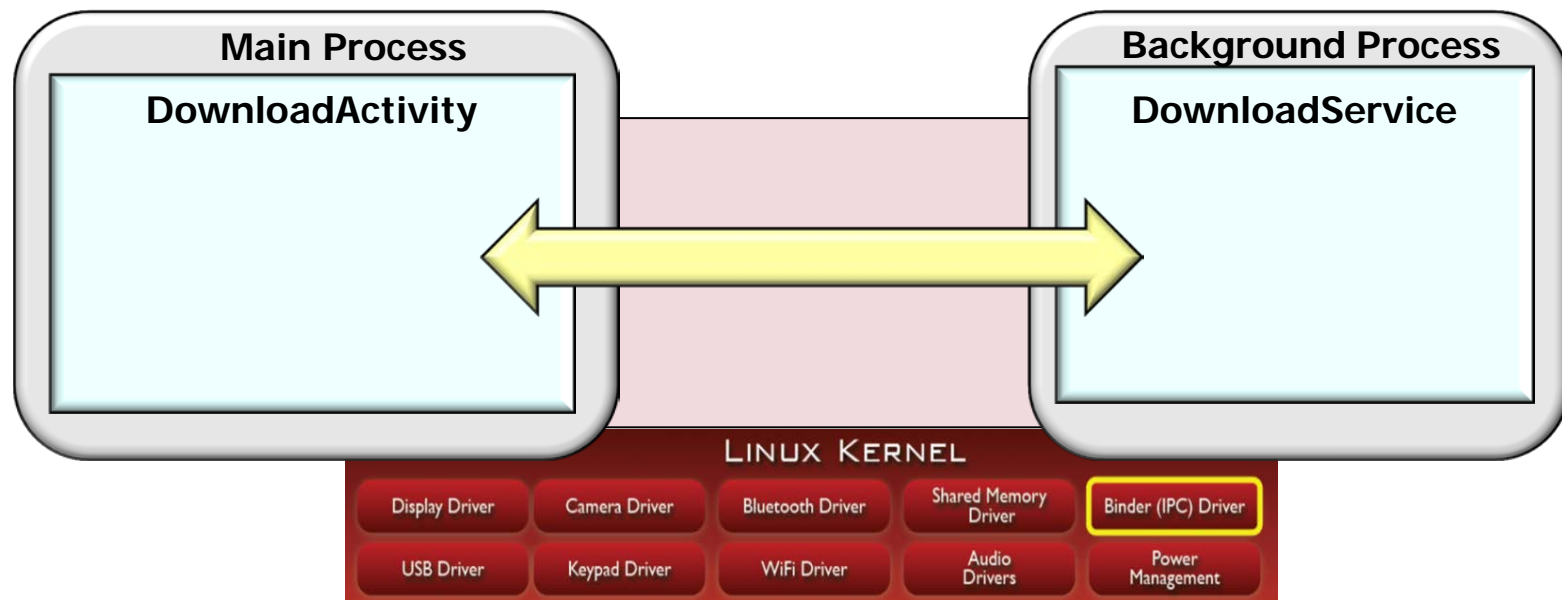


Communicating to Services



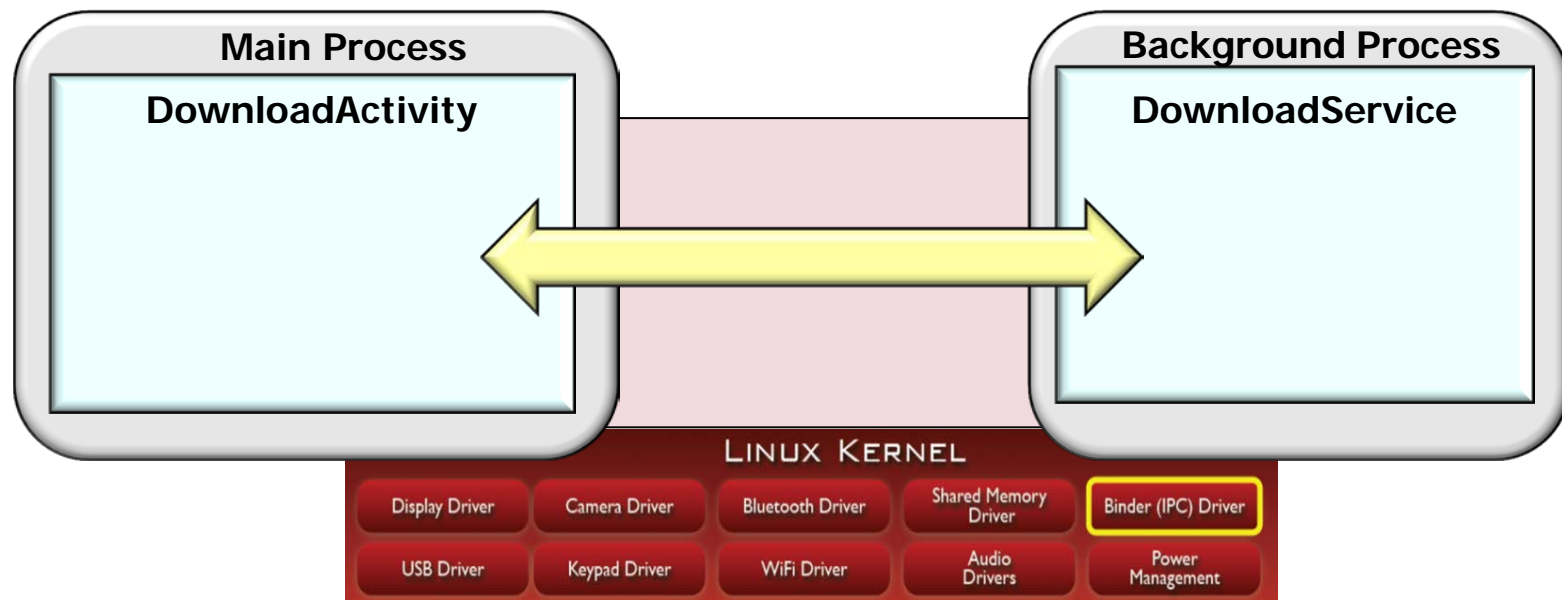
- Activities have two general ways to communicate with a Service
 - Send a command via `startService()`
 - You can add “extras” to the Intent used to start a Service

Communicating to Services



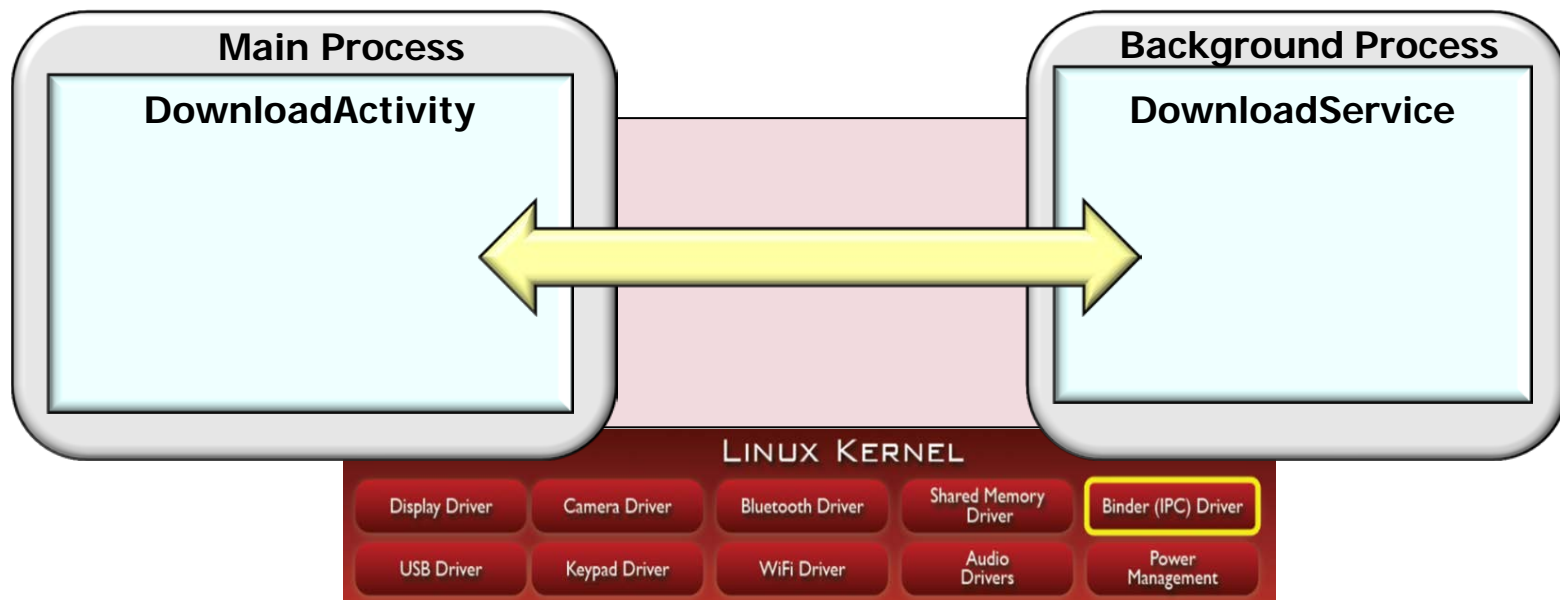
- Activities have two general ways to communicate with a Service
 - Send a command via `startService()`
 - Bind to a Service via `BindService()` & then use the Binder RPC mechanism
 - The Binder supports an object-oriented client/server model defined via the Android Interface Definition Language (AIDL) or Messengers

Communicating to Services



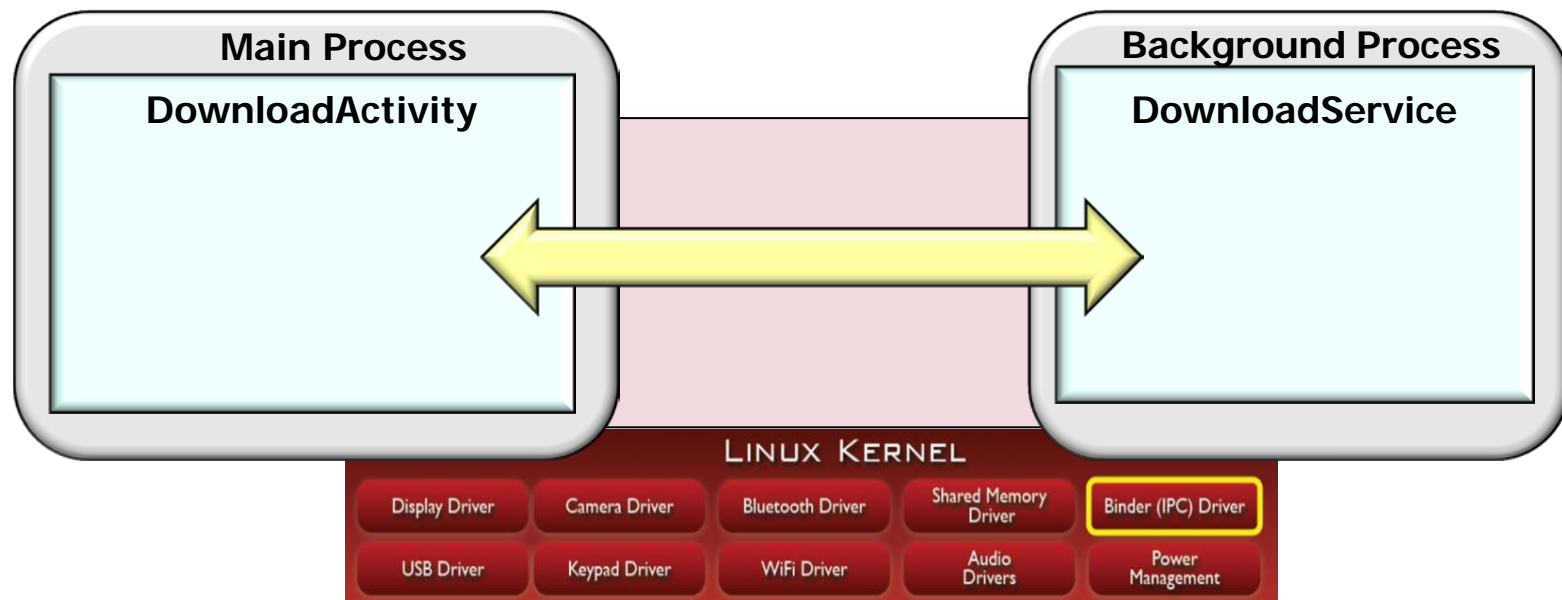
- Activities have two general ways to communicate with a Service
 - Send a command via `startService()`
 - Bind to a Service via `BindService()` & then use the Binder RPC mechanism
 - The Binder supports an object-oriented client/server model defined via the Android Interface Definition Language (AIDL) or Messengers
 - Inter- or intra-process semantics selected by `AndroidManifest.xml` settings

Communicating from Services



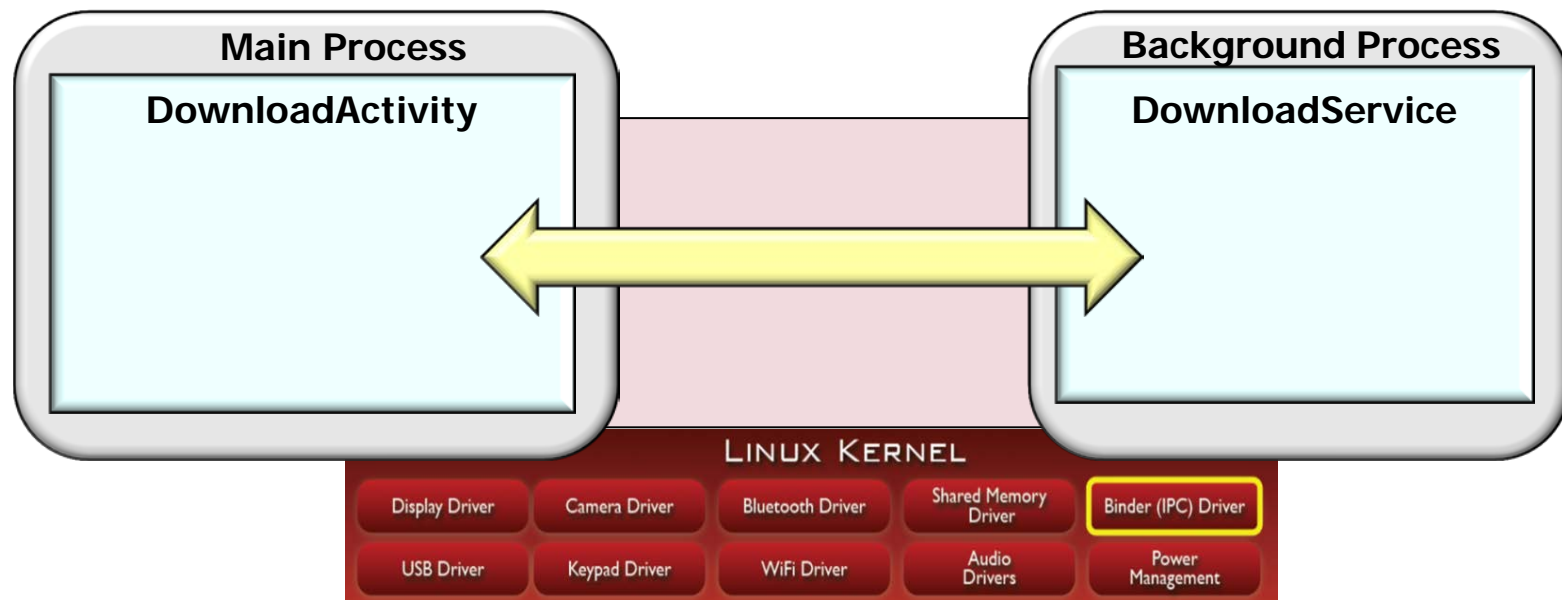
- Services have multiple ways to communicate back to an invoking Activity
 - Use a Messenger object
 - This object can send messages to an Activity's Handler

Communicating from Services



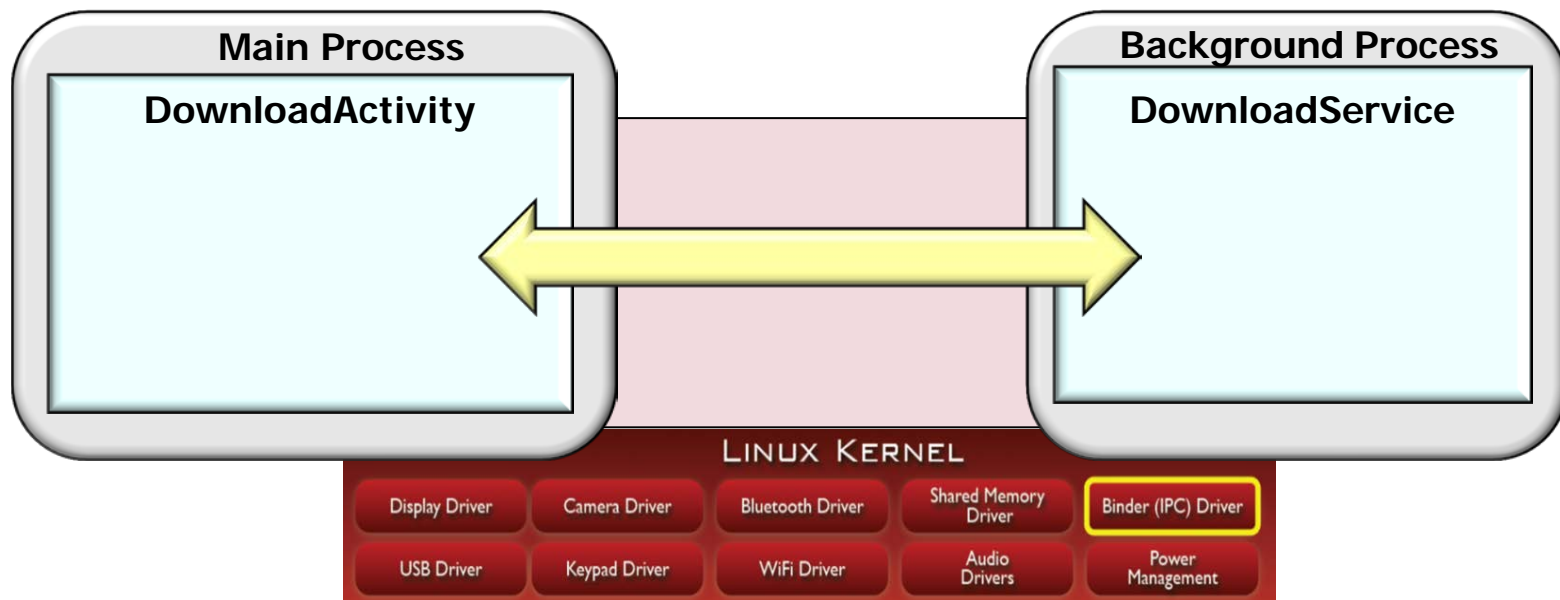
- Services have multiple ways to communicate back to an invoking Activity
 - Use a Messenger object
 - Use Broadcast Intents
 - This requires having the Activity register a BroadcastReceiver

Communicating from Services



- Services have multiple ways to communicate back to an invoking Activity
 - Use a Messenger object
 - Use Broadcast Intents
 - Use a Pending Intent
 - Using a PendingIntent to trigger a call to Activity's `onActivityResult()` method

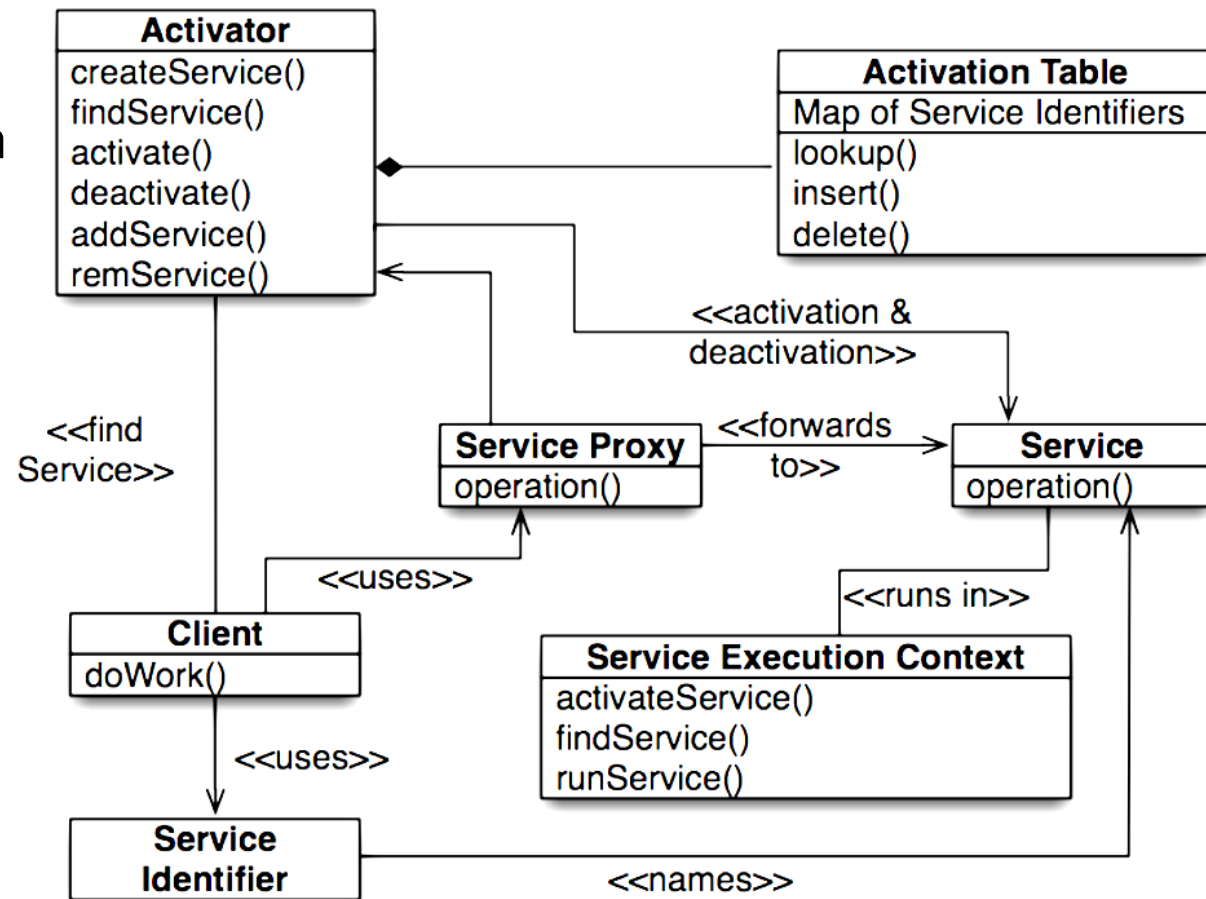
Communicating from Services



- Services have multiple ways to communicate back to an invoking Activity
 - Use a Messenger object
 - Use Broadcast Intents
 - Use a Pending Intent
 - Use an AIDL-based callback object
 - Invoke callback on an AIDL-based object passed to Service via the Binder

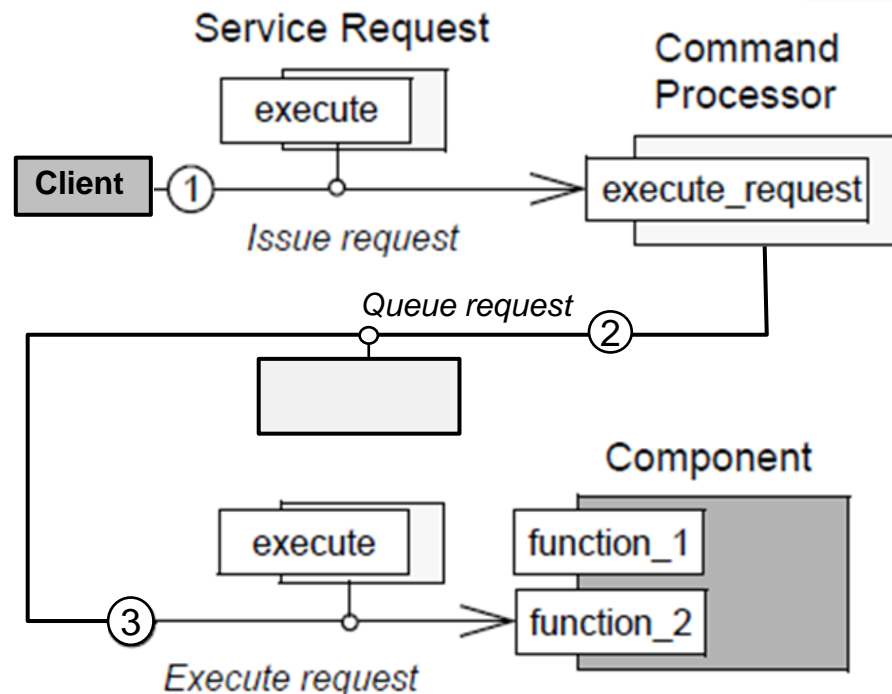
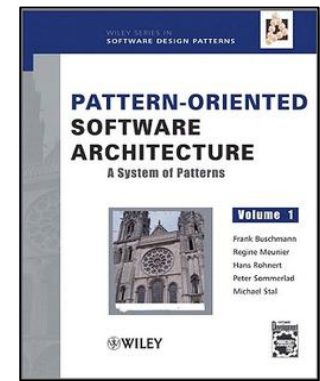
Common Service Communication Patterns

- Several patterns are used to implement communication with Services
- Activator* – Automate the scalable on-demand activation & deactivation of service execution contexts to run services accessed by many clients without consuming resources unnecessarily



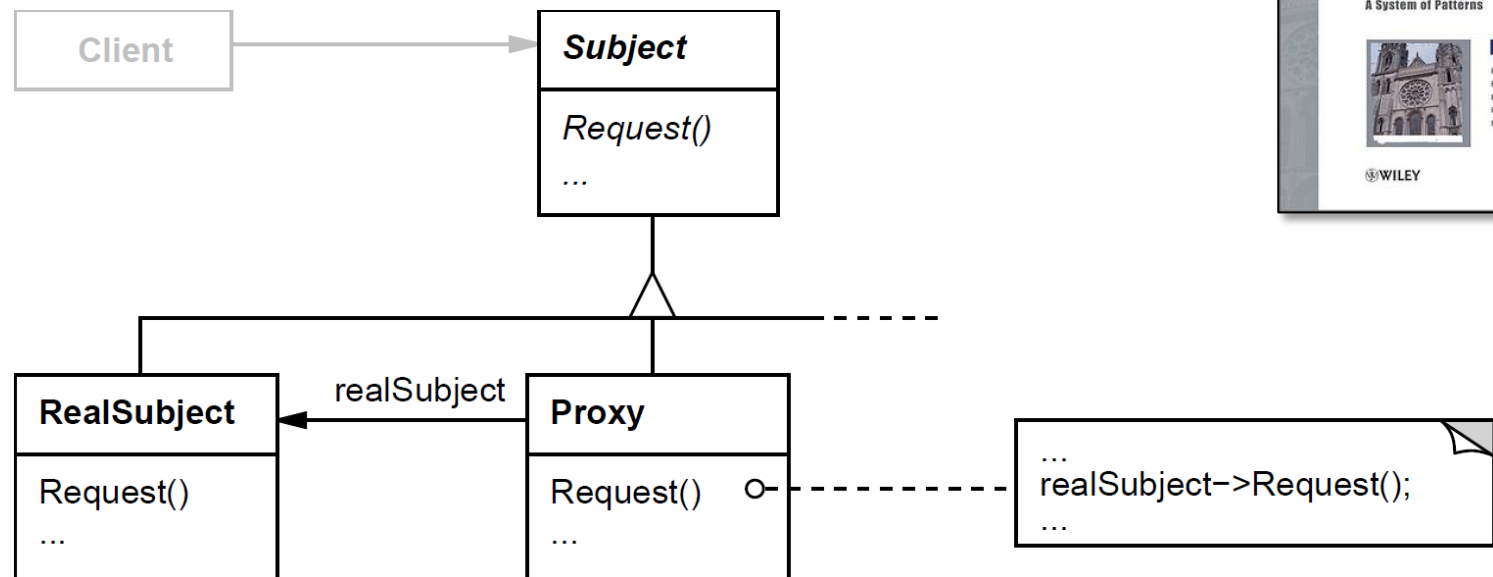
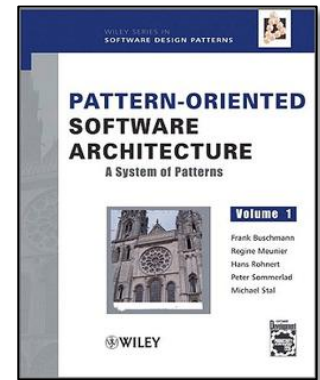
Common Service Communication Patterns

- Services implement several patterns
 - *Activator*
 - *Command Processor* – Encapsulate the request for a service as a command object



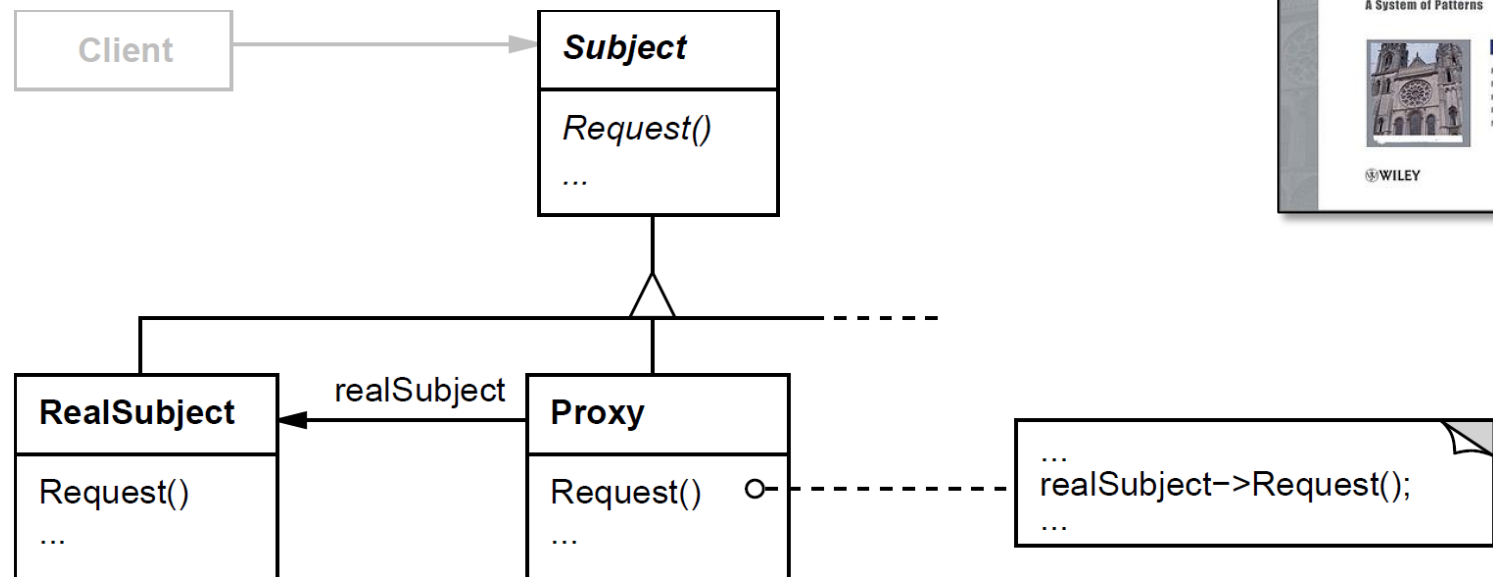
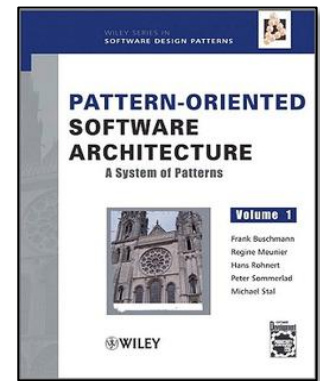
Common Service Communication Patterns

- Services implement several patterns
 - *Activator*
 - *Command Processor*
 - *Proxy* – Provide a surrogate or placeholder for another object to control access to it



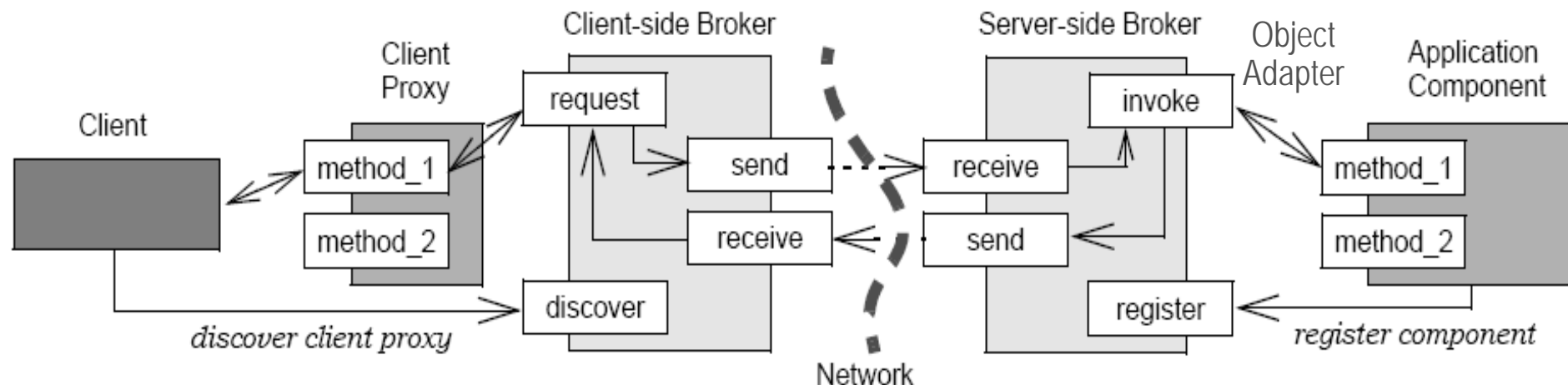
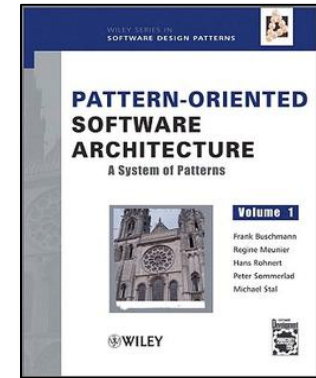
Common Service Communication Patterns

- Services implement several patterns
 - *Activator*
 - *Command Processor*
 - *Proxy* – Provide a surrogate or placeholder for another object to control access to it



Common Service Communication Patterns

- Services implement several patterns
 - *Activator*
 - *Command Processor*
 - *Proxy*
 - *Broker* – Connect clients with remote objects by mediating invocations from clients to remote objects, while encapsulating the details of IPC or network communication



Summary

- There are multiple mechanisms for Activities to communicate with Services

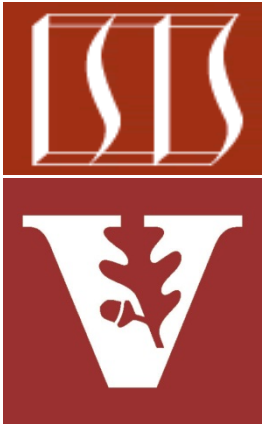


Android Services & Local IPC: Overview of Started Services

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

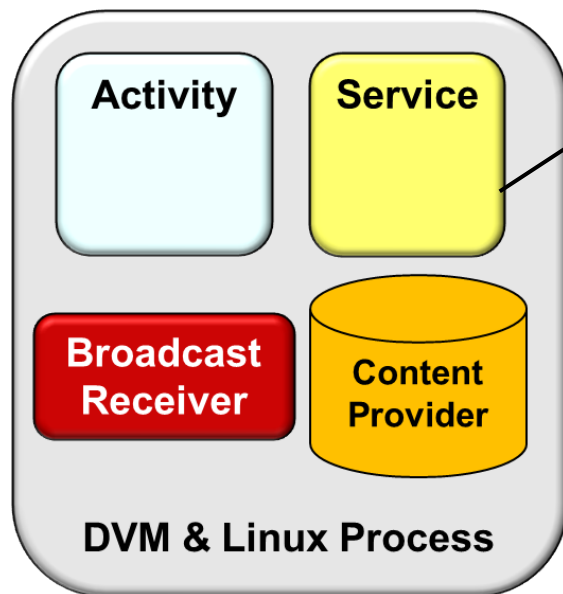
Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

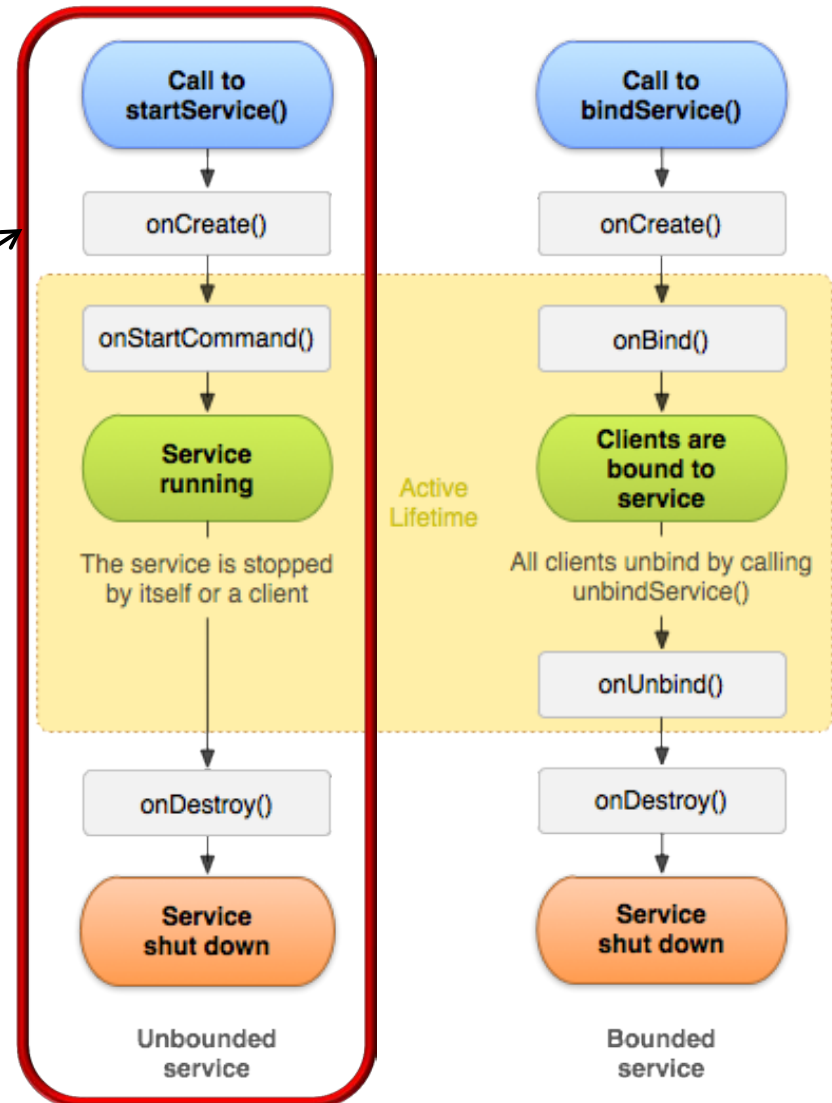


Learning Objectives in this Part of the Module

- Understand how what a Started Service is & what hook methods it defines to manage its various lifecycle states



We'll emphasize commonalities & variabilities in our discussion



Overview of Started Services

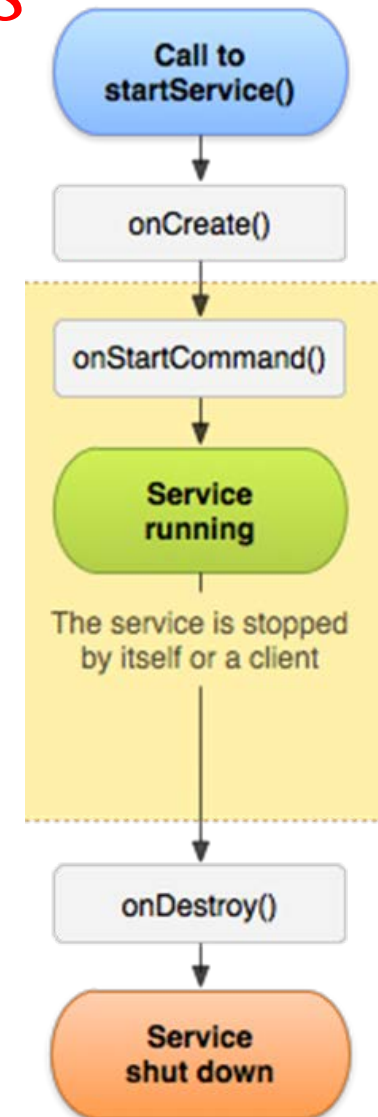
- A Started Service is one that a client component starts by calling `startService()`
 - The Intent identifies the Service to communicate with & supplies parameters (via Intent extras) to tell the Service what to do

```
Intent intent = new Intent  
    (this,  
     ThreadedDownloadService.class));  
intent.putExtra("URL", imageUrl);  
startService(intent);
```

This call doesn't block

**Download
Activity**

**Download
Service**



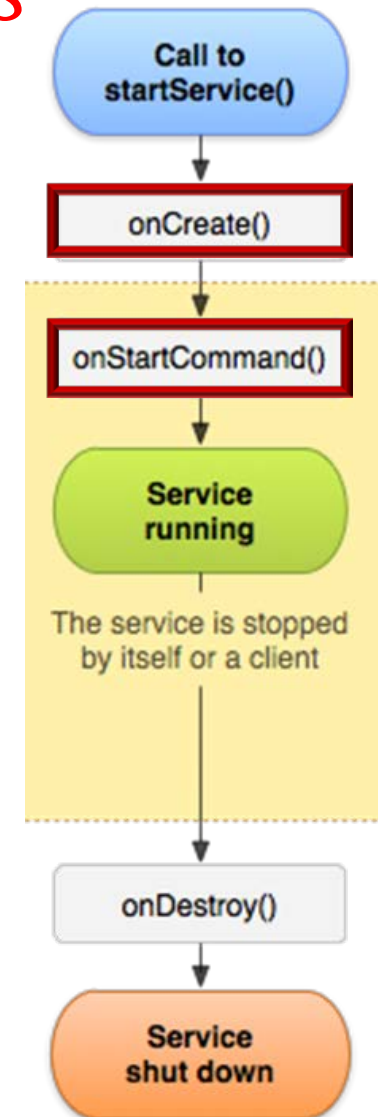
Overview of Started Services

- A Started Service is one that a client component starts by calling `startService()`
- This results in a call to the Service's `onCreate()` & `onStartCommand()` hook methods
 - If the Service is not already running it will be started & will receive the Intent via `onStartCommand()`

```
public class DownloadService extends Service {  
    int onStartCommand(Intent intent,  
                        int flags, int startId)  
    { ... }  
}
```

Download
Activity

Download
Service



Overview of Started Services

- A Started Service is one that a client component starts by calling `startService()`
- This results in a call to the Service's `onCreate()` & `onStartCommand()` hook methods
 - If the Service is not already running it will be started & will receive the Intent via `onStartCommand()`
 - This return a result to Android, but not to client

```
public class DownloadService extends Service {  
    int onStartCommand(Intent intent,  
                        int flags, int startId)  
    { return ...; }  
}
```

Download
Activity

Download
Service



Overview of Started Services

Call to
startService()

Return value tells Android what it should do with the service if its process is killed while it is running

- *START_STICKY – Don't redeliver Intent to onStartCommand() (pass null intent)*
- *START_NOT_STICKY – Service should remain stopped until/unless explicitly started by some client code*
- *START_REDELIVER_INTENT – Restart Service via onStartCommand(), supplying the same Intent as was delivered this time*

```
public class DownloadService extends Service {  
    int onStartCommand(Intent intent,  
                        int flags, int startId)  
    { return ...; }  
}
```

Download
Activity

Download
Service

The service is stopped
by itself or a client

onDestroy()

Service
shut down

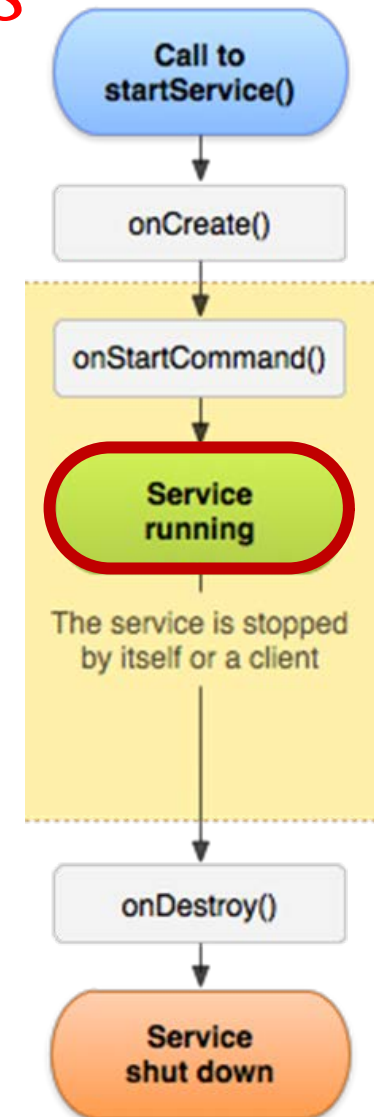
Overview of Started Services

- A Started Service is one that another component starts by calling `startService()`
- This results in a call to the Service's `onCreate()` & `onStartCommand()` hook methods
- A started service often performs a single operation & might not return a result to the caller
 - e.g., it could download or upload a file over TCP

```
public class DownloadService ...  
    String downloadFile (Uri uri) {  
        InputStream in = (InputStream)  
            new URL(uri.toString()).getContent();  
        ...  
    }
```

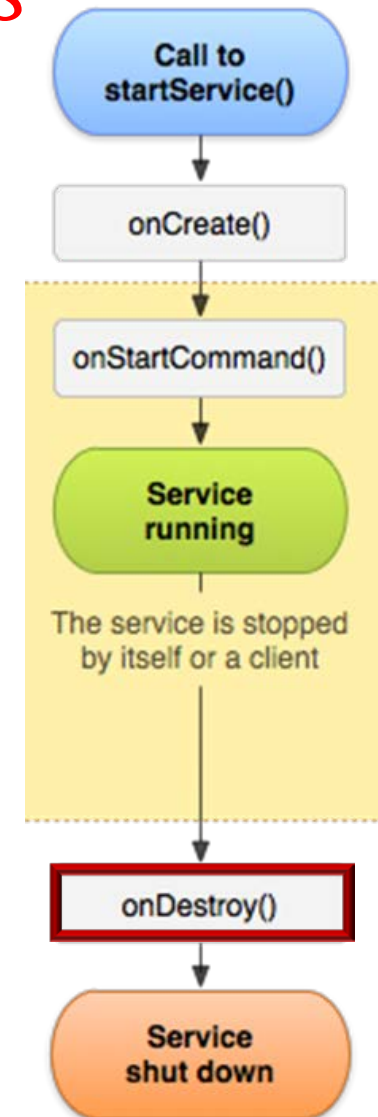
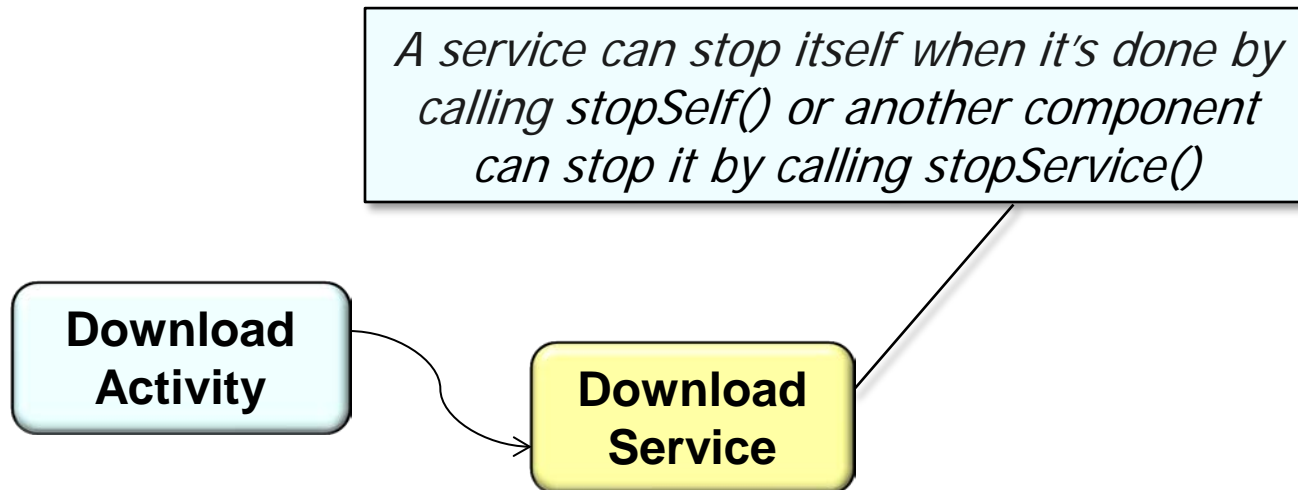
Download
Activity

Download
Service



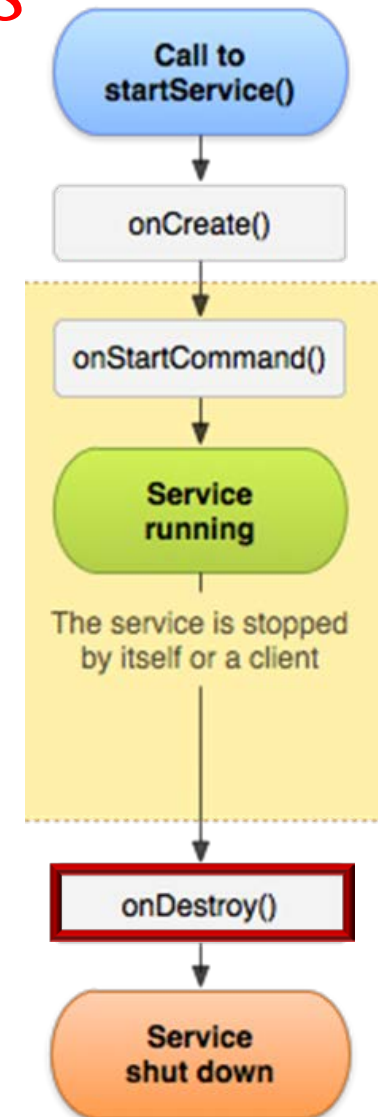
Overview of Started Services

- A Started Service is one that another component starts by calling `startService()`
- This results in a call to the Service's `onCreate()` & `onStartCommand()` hook methods
- A started service often performs a single operation & might not return a result to the caller
- When the operation is done, the service can be stopped



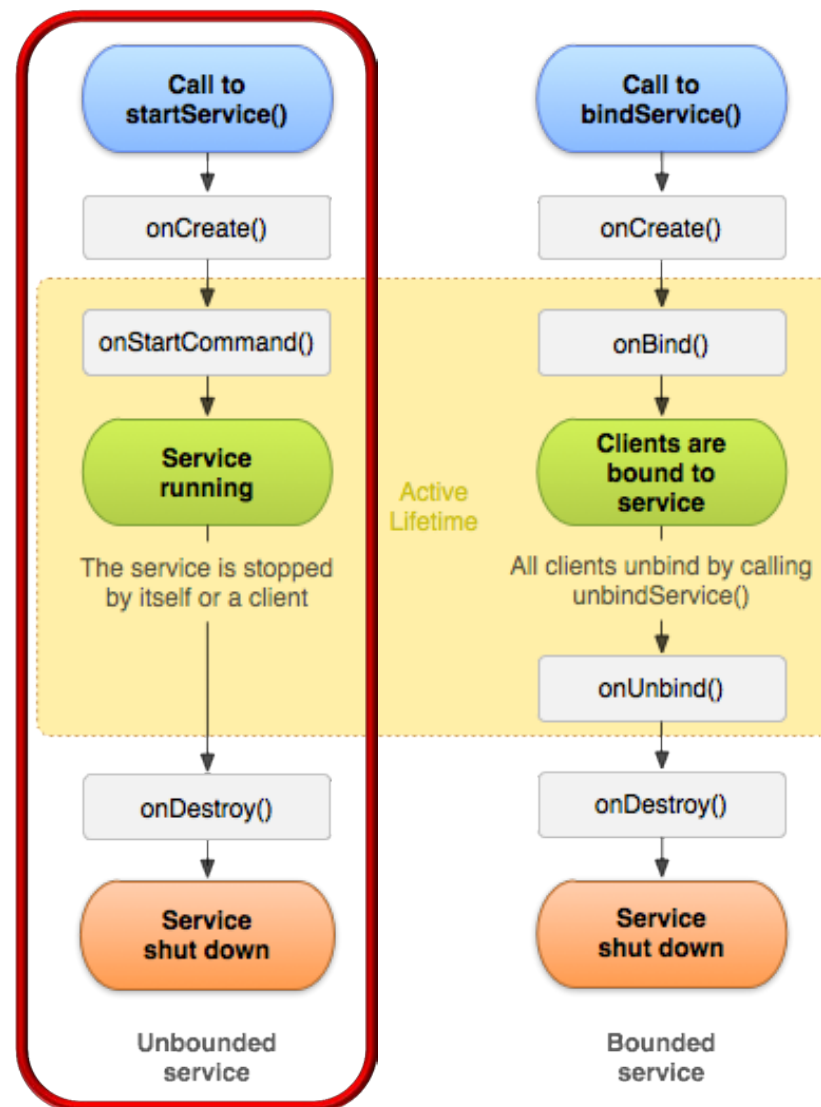
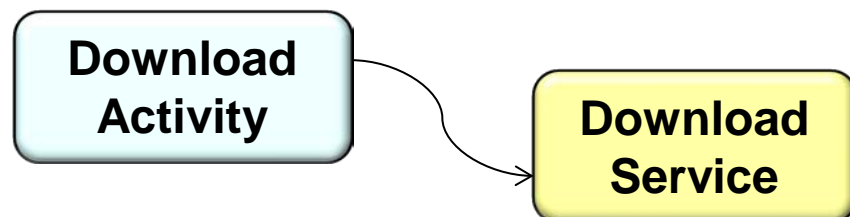
Overview of Started Services

- A Started Service is one that another component starts by calling `startService()`
- This results in a call to the Service's `onCreate()` & `onStartCommand()` hook methods
- A started service often performs a single operation & might not return a result to the caller
- When the operation is done, the service can be stopped
- Examples of Android Started Services
 - *SMS & MMS Services*
 - Manage messaging operations, such as sending data, text, & pdu messages
 - *AlertService*
 - Handle calendar event reminders



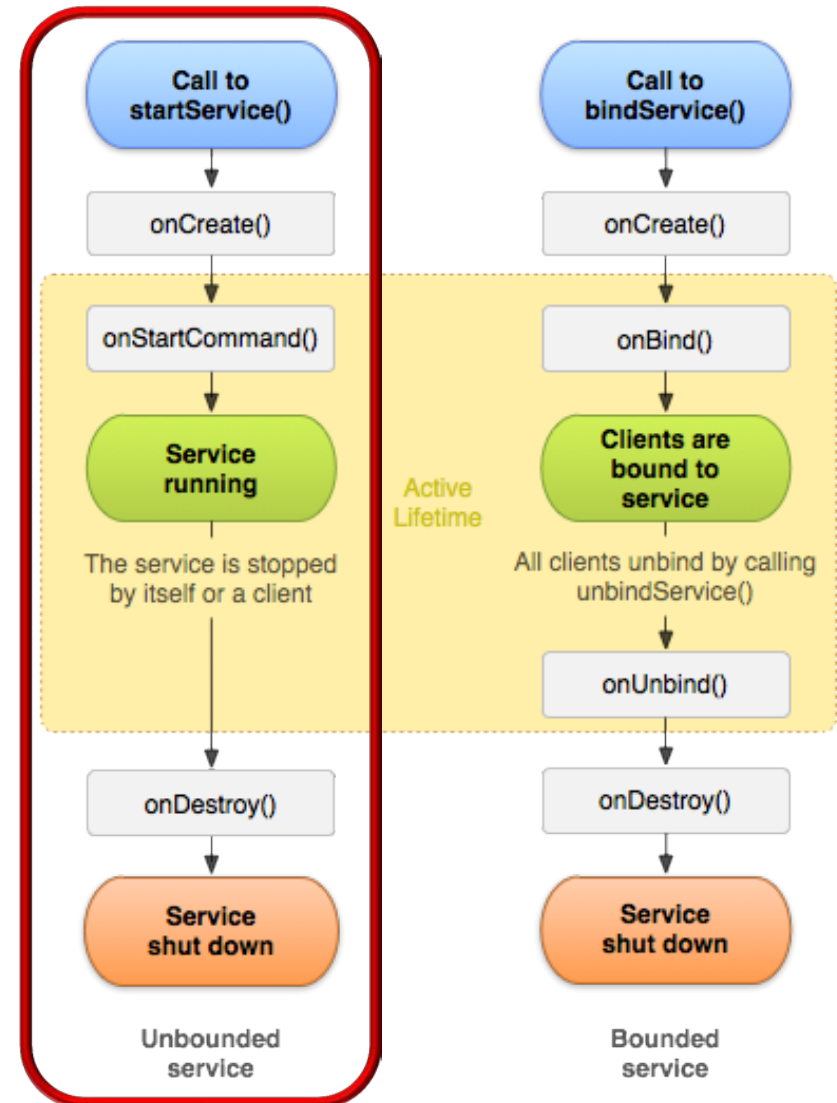
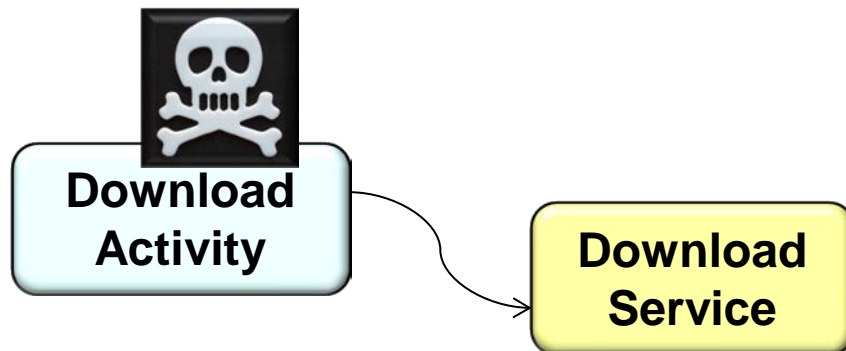
Summary

- When a Started Service is launched, it has a lifecycle that's independent of the component that started it



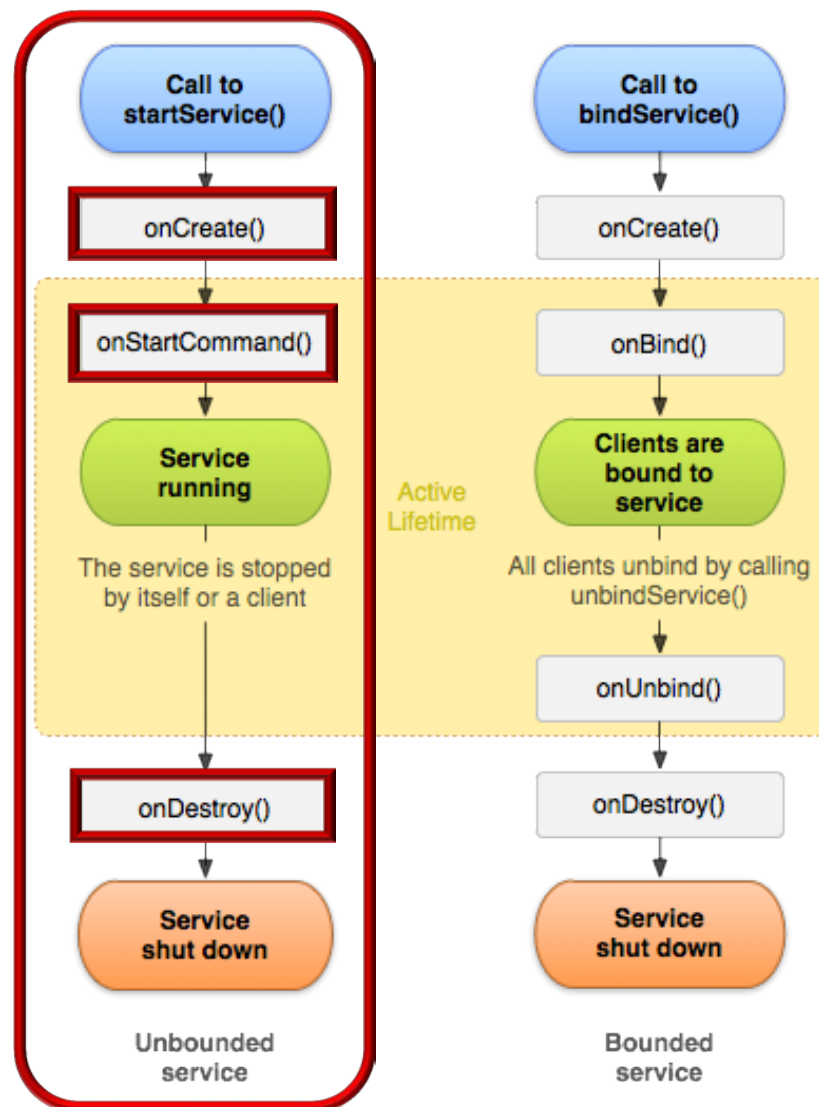
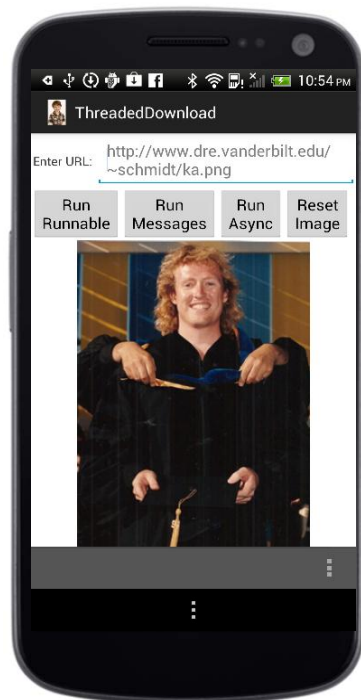
Summary

- When a Started Service is launched, it has a lifecycle that's independent of the component that started it
- The service can run in the background indefinitely, even if the component that started it is destroyed



Summary

- When a Started Service is launched, it has a lifecycle that's independent of the component that started it
- Android's Started Services support inversion of control

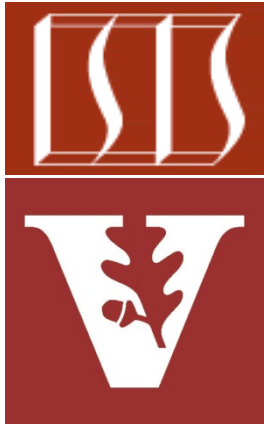


Android Services & Local IPC: Programming Started Services

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

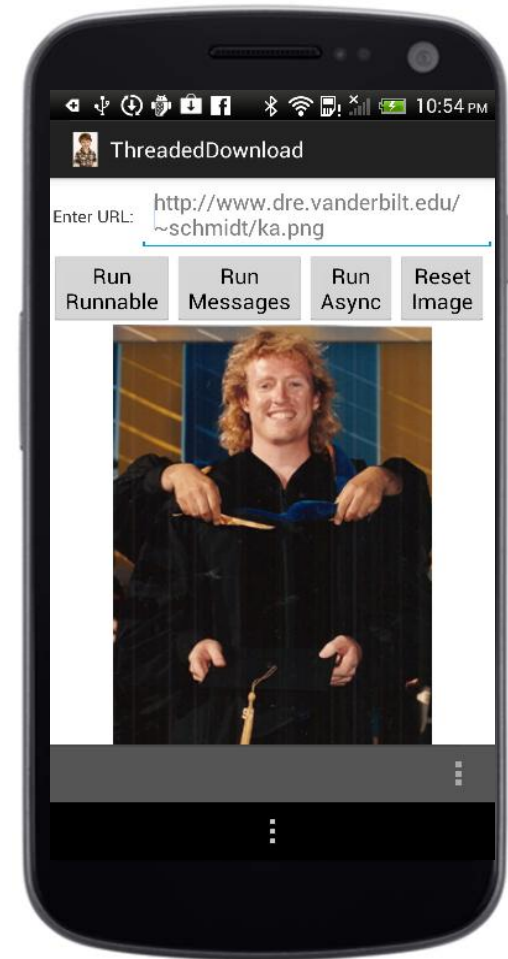
Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



Learning Objectives in this Part of the Module

- Understand how to program Started Services



Programming a Started Service

- Implementing a Started Service is similar to implementing an Activity, e.g.:
 - Inherit from Android Service class

```
public class MusicService
    extends Service {
    public void onCreate() {
        ...
    }
    public int onStartCommand
        (Intent intent,
         int flags, int startId) {
        ...
    }
    protected void onDestroy() {
        ...
    }
    public IBinder
        onBind(Intent intent) {
        return null;
    }
    ...
}
```



Programming a Started Service

- Implementing a Started Service is similar to implementing an Activity, e.g.:
 - Inherit from Android Service class
 - Override lifecycle methods

May need to implement the concurrency model in onStartCommand()

```
public class MusicService
    extends Service {
    public void onCreate() {
        ...
    }
    public int onStartCommand
        (Intent intent,
         int flags, int startId) {
        ...
    }
    protected void onDestroy() {
        ...
    }
    public IBinder
        onBind(Intent intent) {
        return null;
    }
    ...
}
```



Programming a Started Service

- Implementing a Started Service is similar to implementing an Activity, e.g.:
 - Inherit from Android Service class
 - Override lifecycle methods
 - The `onBind()` method & `onUnbind()` aren't used for Started Services

Started Services need to provide a no-op implementation for `onBind()`

```
public class MusicService
    extends Service {
    public void onCreate() {
        ...
    }
    public int onStartCommand
        (Intent intent,
         int flags, int startId) {
        ...
    }
    protected void onDestroy() {
        ...
    }
    public IBinder
        onBind(Intent intent) {
        return null;
    }
    ...
}
```



Programming a Started Service

- Implementing a Started Service is similar to implementing an Activity, e.g.:
 - Inherit from Android Service class
 - Override lifecycle methods
 - Include the Service in the AndroidManifest.xml config file

```
<application ... >
    <activity android:name=
        ".MusicActivity"
        ...
    </activity>

    <service
        android:exported="false"
        android:name=
            ".BGLoggingService"
        ...
    </service>

</application>
```

Music Player App Overview

- MusicActivity can play music via a Started Service



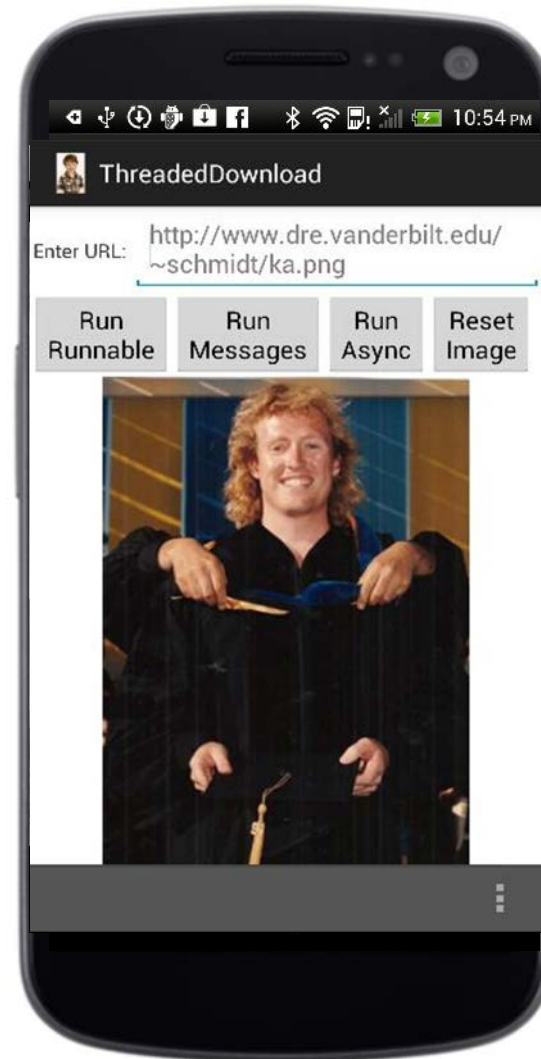
Music Player App Overview

- MusicActivity can play music via a Started Service
- To start the Service a user needs to push the “Play” button



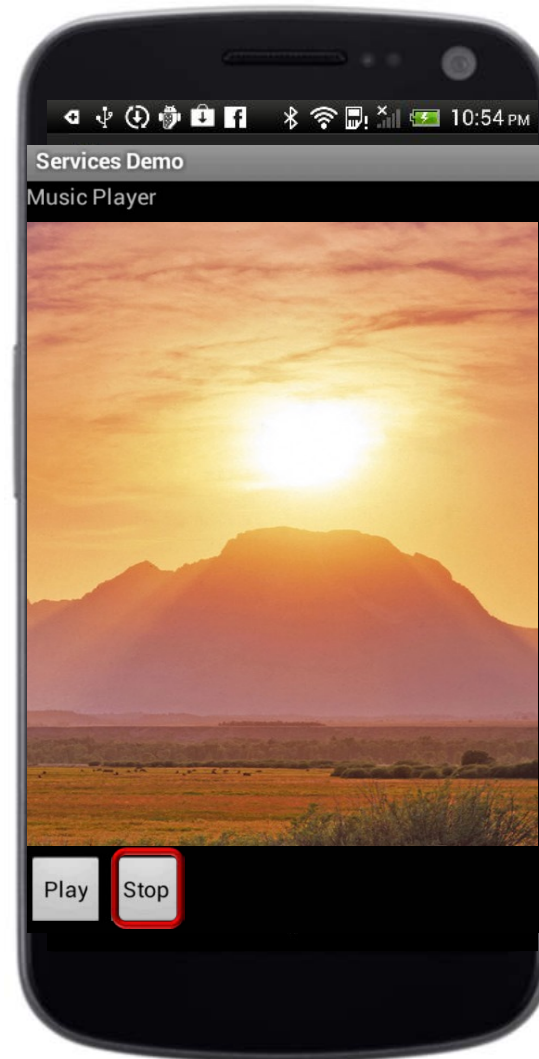
Music Player App Overview

- MusicActivity can play music via a Started Service
- To start the Service a user needs to push the “Play” button
- If music is playing when MusicActivity leaves the foreground, the Music Service will continue playing



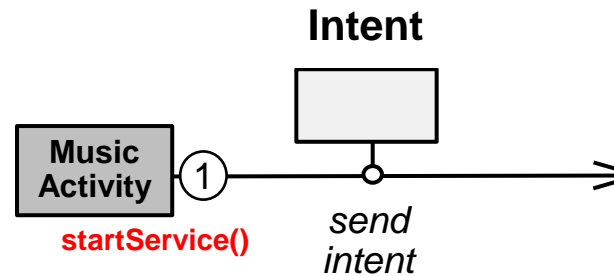
Music Player App Overview

- MusicActivity can play music via a Started Service
- To start the Service a user needs to push the “Play” button
- If music is playing when MusicActivity leaves the foreground, the Music Service will continue playing
- To stop the Service a user needs to explicitly push the “Stop” button



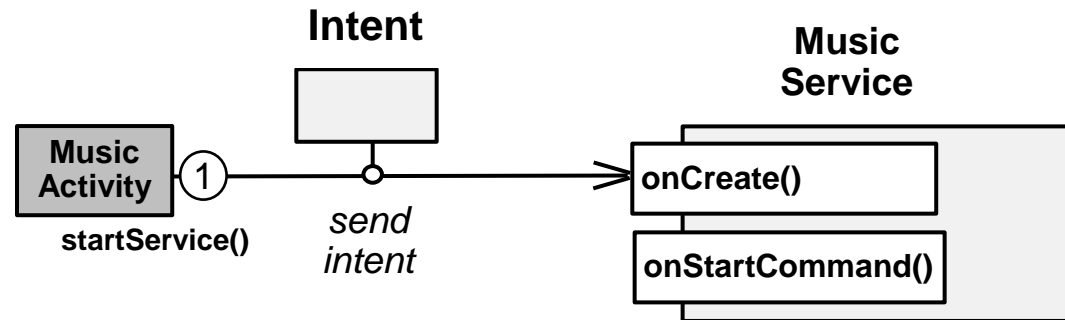
Music Player App Interactions

- MusicActivity send an Intent via a call to `startService()`
- This Intent indicates which song to play



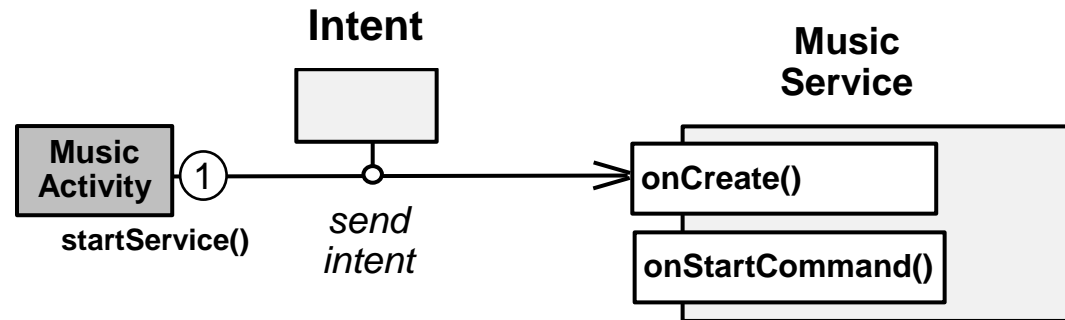
Music Player App Interactions

- MusicActivity send an Intent via a call to startService()
- The MusicService is started on-demand
 - Based on the *Activator* pattern



Music Player App Interactions

- MusicActivity send an Intent via a call to startService()
- The MusicService is started on-demand
- The onStartCommand() starts playing the song requested by the MusicActivity



Music Player Activity Implementation

```
public class MusicActivity extends Activity {  
    ...  
    public void play (View src) {  
        Intent intent = new Intent(MusicActivity.this,  
                                   MusicService.class);  
        intent.putExtra("SongID", R.raw.braincandy);
```

Add the song to play as an “extra”



```
        startService(intent);  
    }
```



Launch the Started Service that handles this Intent

```
    public void stop (View src) {  
        Intent intent = new Intent(MusicActivity.this,  
                                   MusicService.class);  
        stopService (intent);  
    }  
}
```





Stop the Started Service


Clearly, a production music play app wouldn't hard-code the song selection!!


Music Player Service Implementation

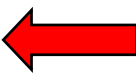
```
public class MusicService extends Service {  
    MediaPlayer player;  
  
    public int onStartCommand (Intent intent,  
                               int flags, int startid) {  
        player = MediaPlayer.create(this,  
                                     intent.getIntExtra("SongID",  
                                                         0));  
        player.setLooping(true);  
        player.start();  
  
        return START_NOT_STICKY;  
    }  
  
    public void onDestroy() { player.stop(); }  
}
```

 **Inherit from Service class**

 **Extract the resid from the "extra" & create a MediaPlayer**

 **Start playing the song (doesn't block)**

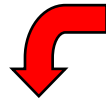
 **Don't restart Service if it shuts down**

 **Stop player when Service is destroyed**

AndroidManifest.xml File

```
<application ... >
```

```
    <activity android:name=".MusicActivity"
              android:label="@string/app_name">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name=
                "android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
```



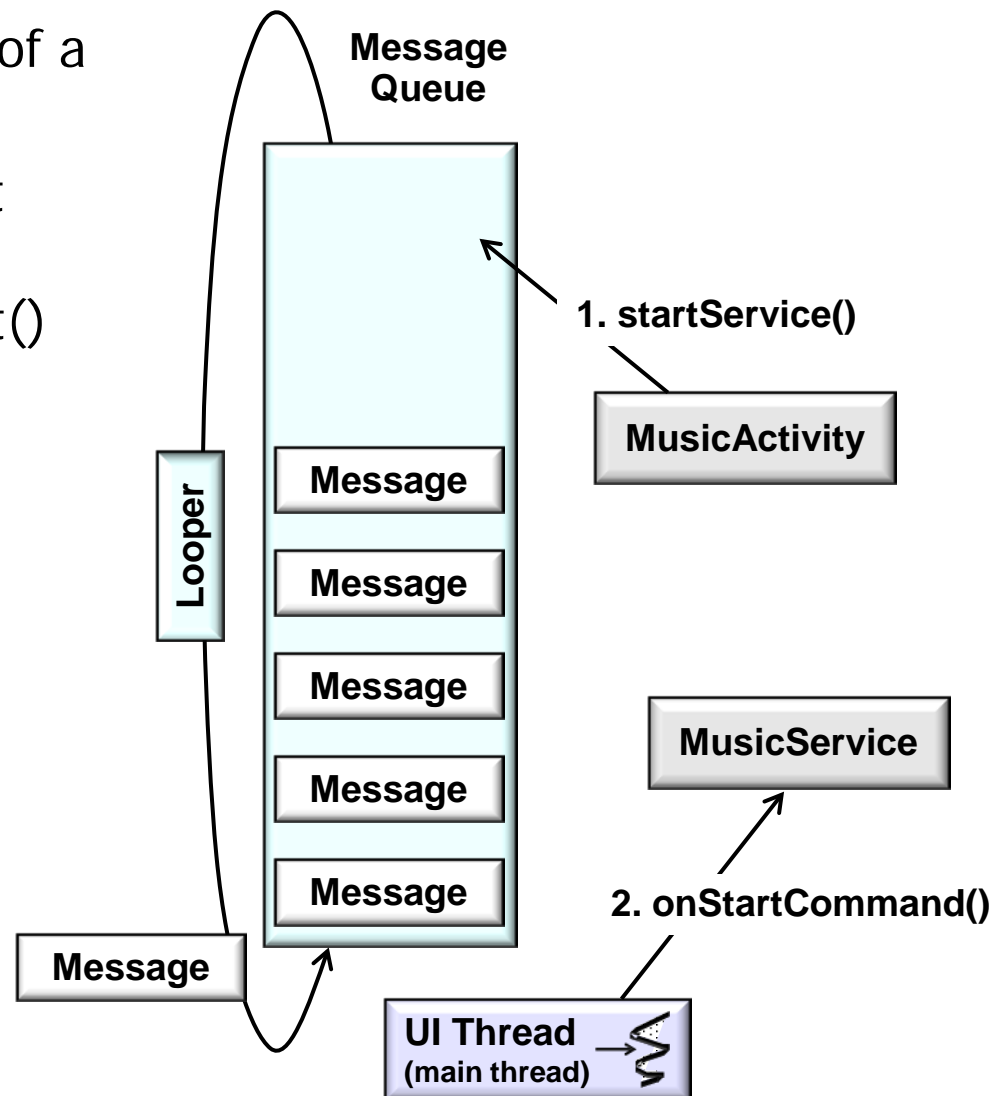
Service is usable by components
external to this application

```
    <service android:exported="true"
            android:name=".MusicService" />
```

```
</application>
```

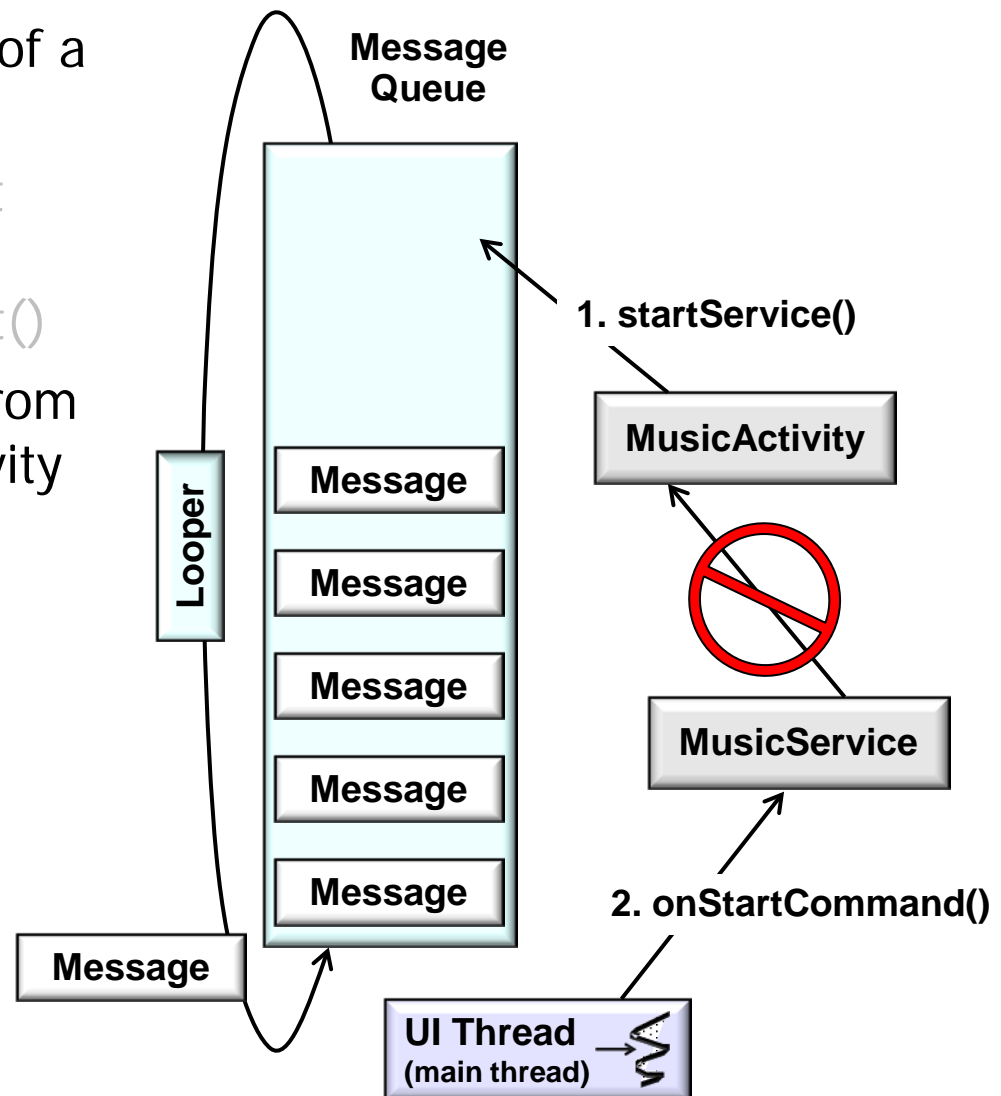
Analysis of the Music Player Service Example

- This is a very simple example of a Started Service, e.g.,
- It runs in the UI Thread, but doesn't block due to the behavior of `MediaPlayer.start()`



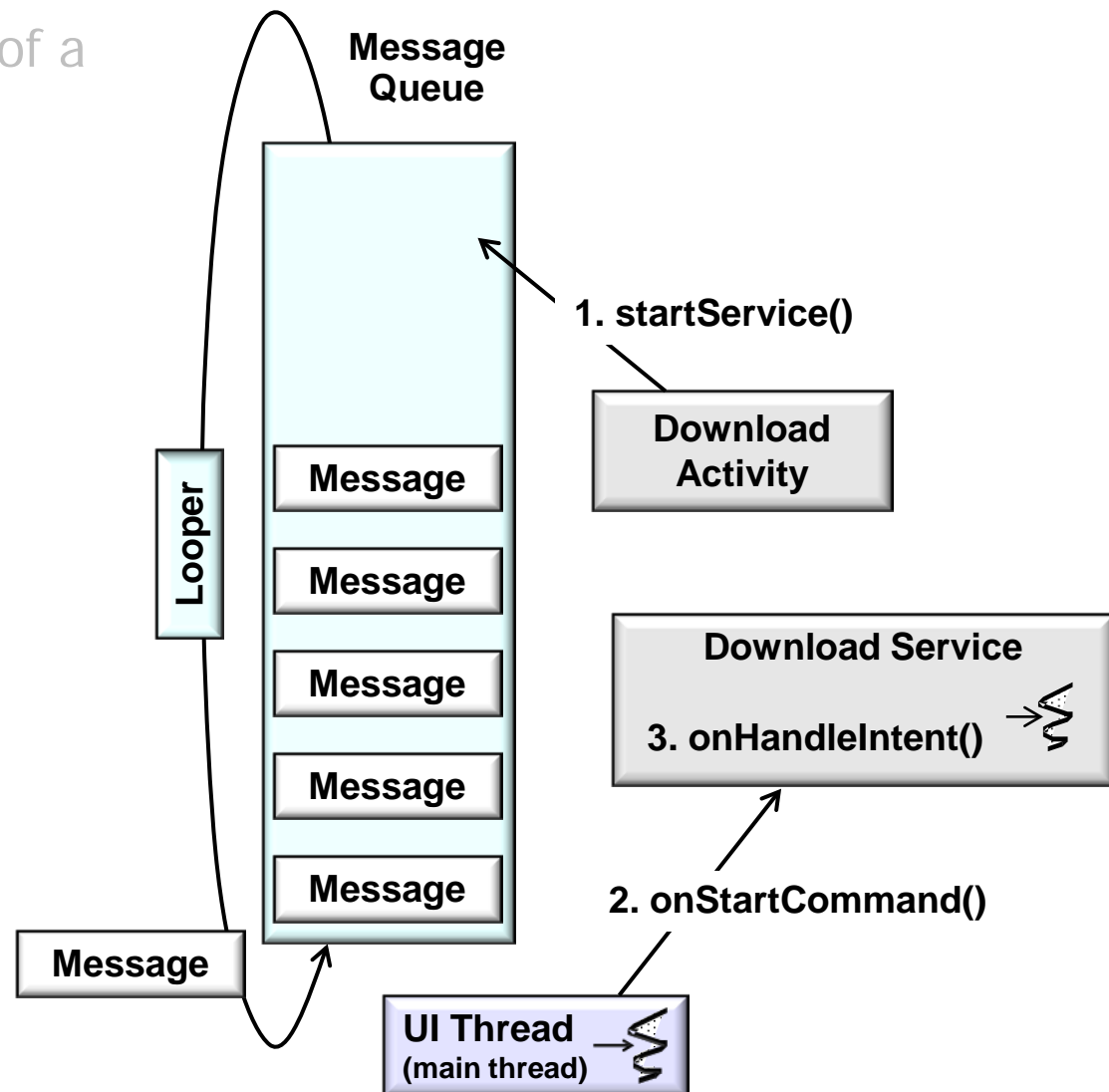
Analysis of the Music Player Service Example

- This is a very simple example of a Started Service, e.g.,
 - It runs in the UI Thread, but doesn't block due to the behavior of `MediaPlayer.start()`
 - There's no communication from the Service back to the Activity that invoked it!



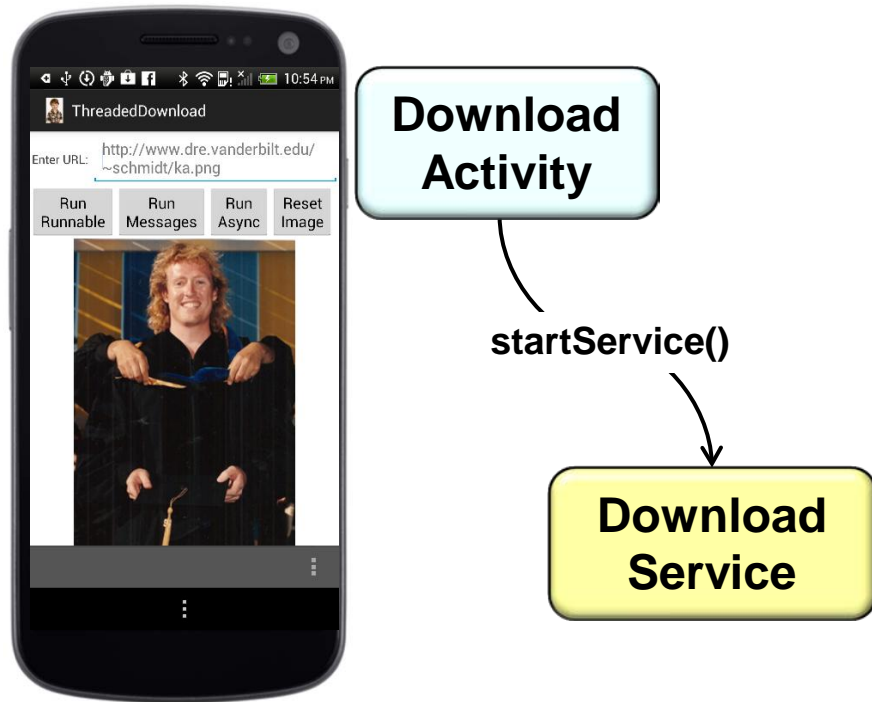
Analysis of the Music Player Service Example

- This is a very simple example of a Started Service
- Services with long-running operations typically need to run in separate Thread(s)



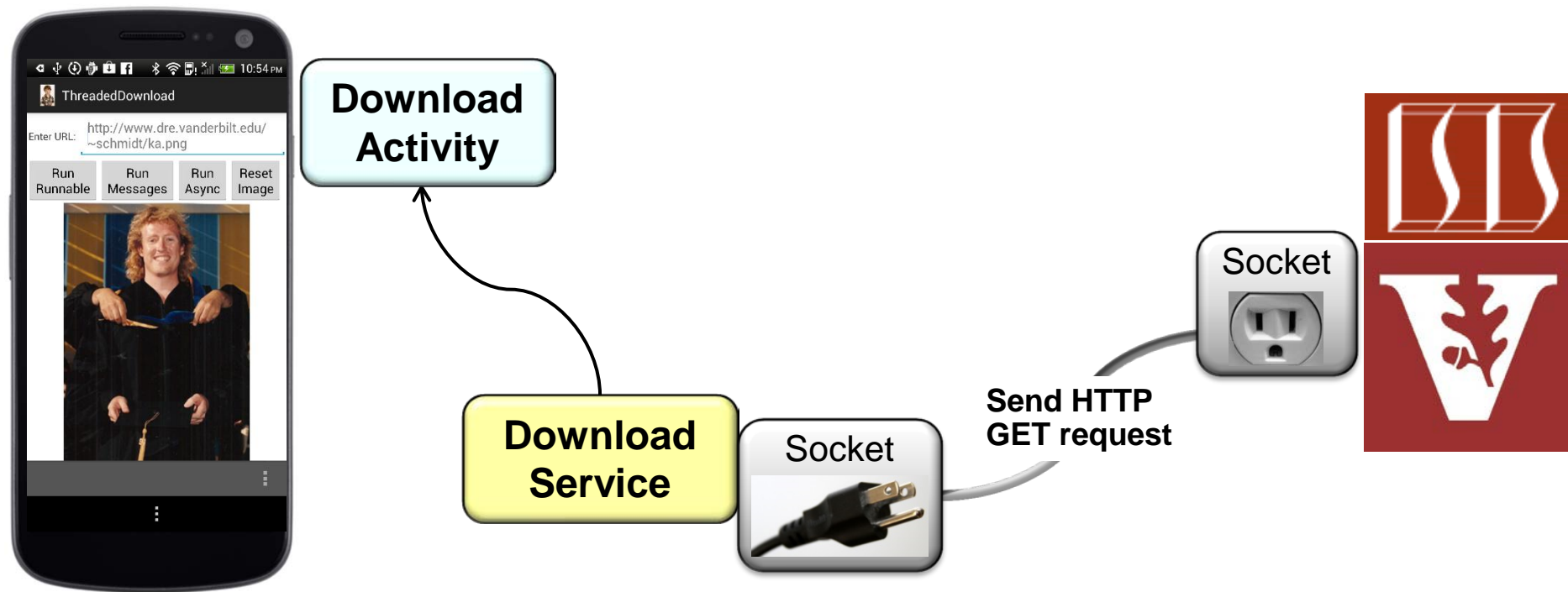
Download App Overview

- DownloadActivity requests a DownloadService to get a file from a server



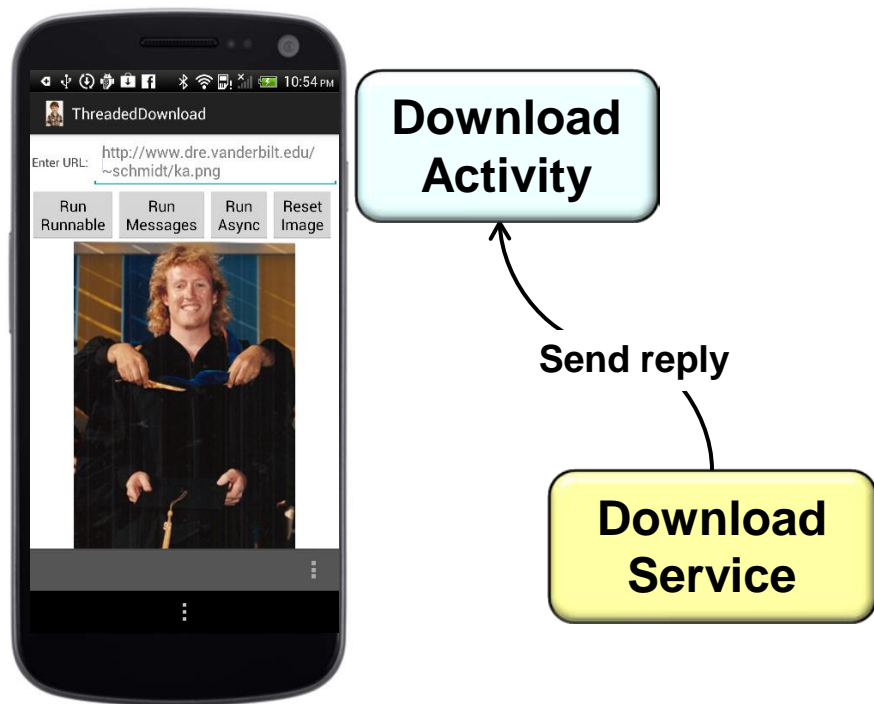
Download App Overview

- DownloadActivity requests a DownloadService to get an image from a server
- The DownloadService downloads the image & stores it in a file on the device



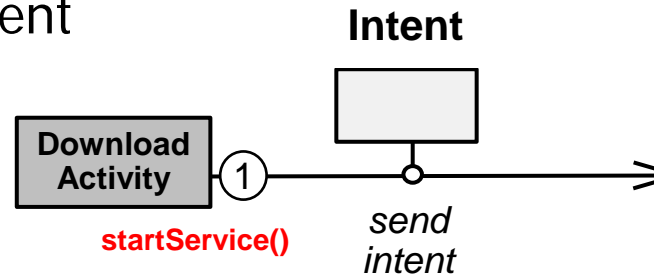
Download App Overview

- DownloadActivity requests a DownloadService to get an image from a server
- The DownloadService downloads the image & stores it in a file on the device
- The DownloadService returns the pathname of the file back to the DownloadActivity, which then displays the image



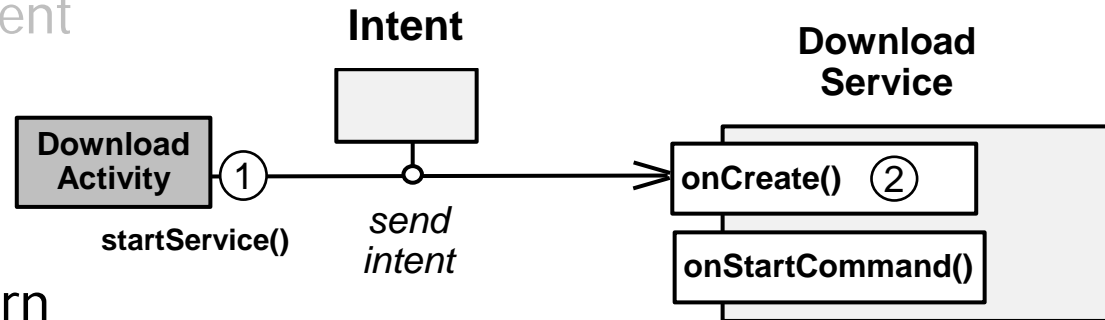
Download App Interactions

- DownloadActivity sends an Intent via a call to `startService()`



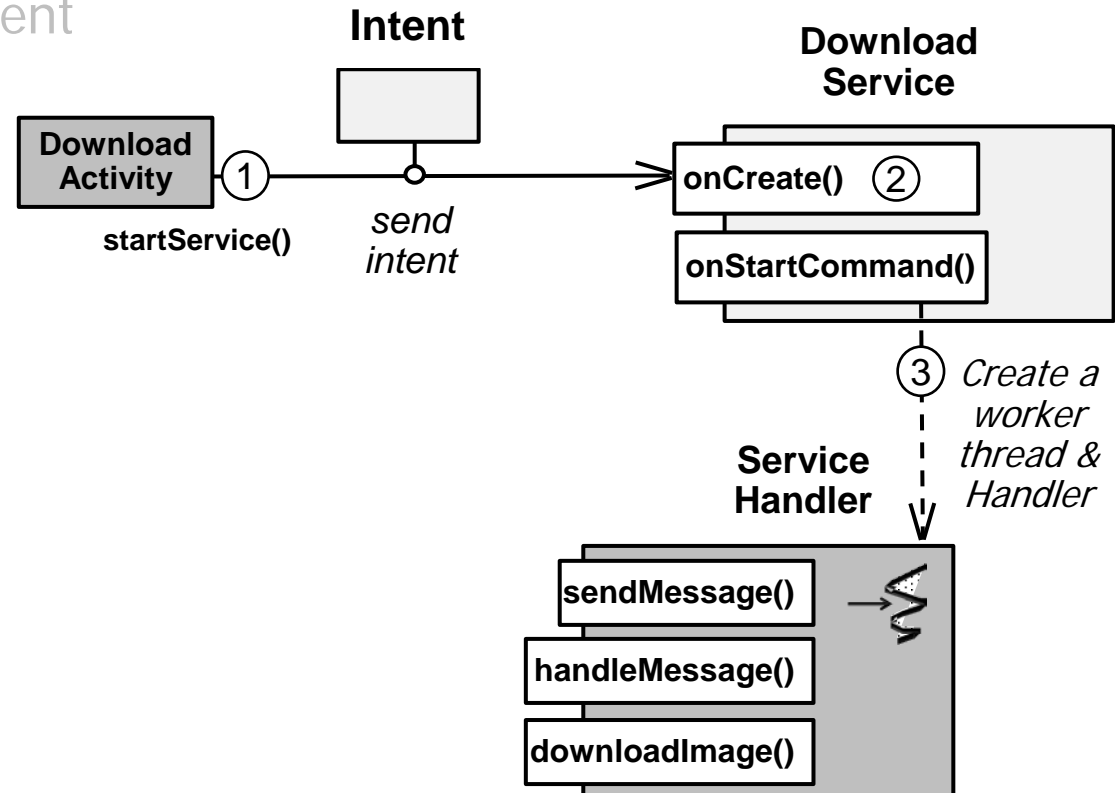
Download App Interactions

- DownloadActivity sends an Intent via a call to startService()
- The DownloadService is started on-demand
 - Based on the *Activator* pattern



Download App Interactions

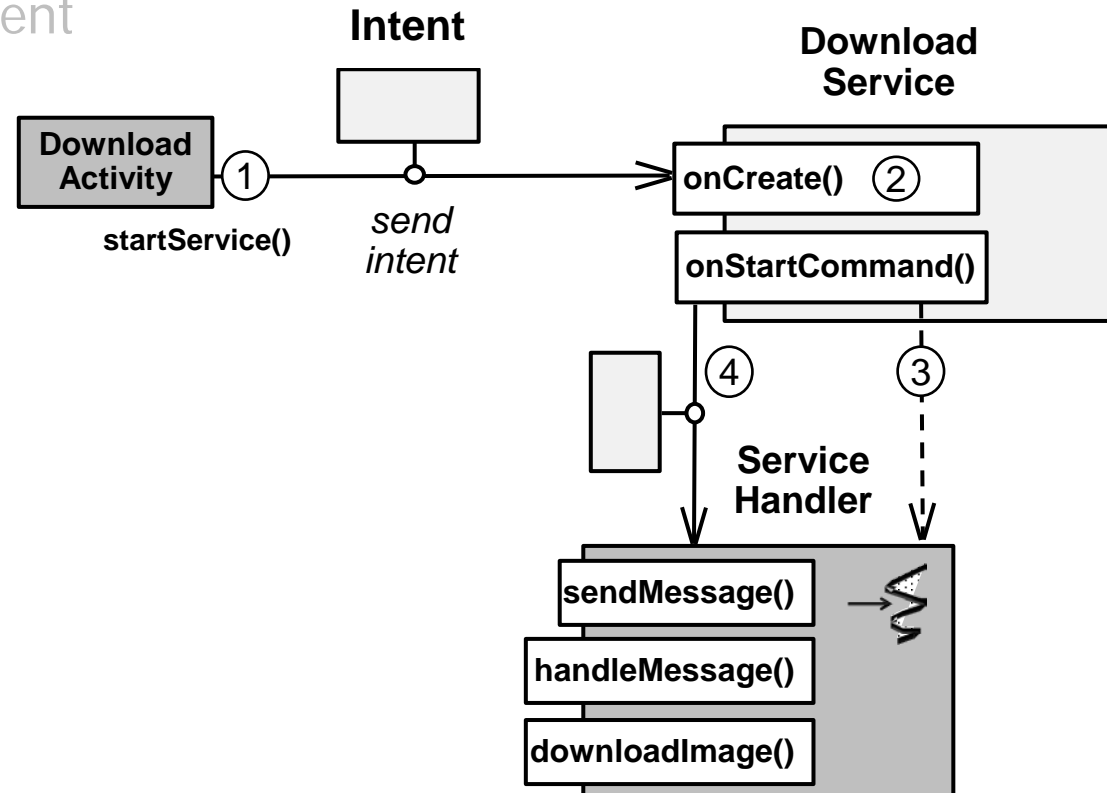
- DownloadActivity sends an Intent via a call to startService()
- The DownloadService is started on-demand
- The DownloadService does several things
 - Creates a ServiceHandler
 - Internally creates a single worker thread



The ServiceHandler is a common idiom in multi-threaded Android Services

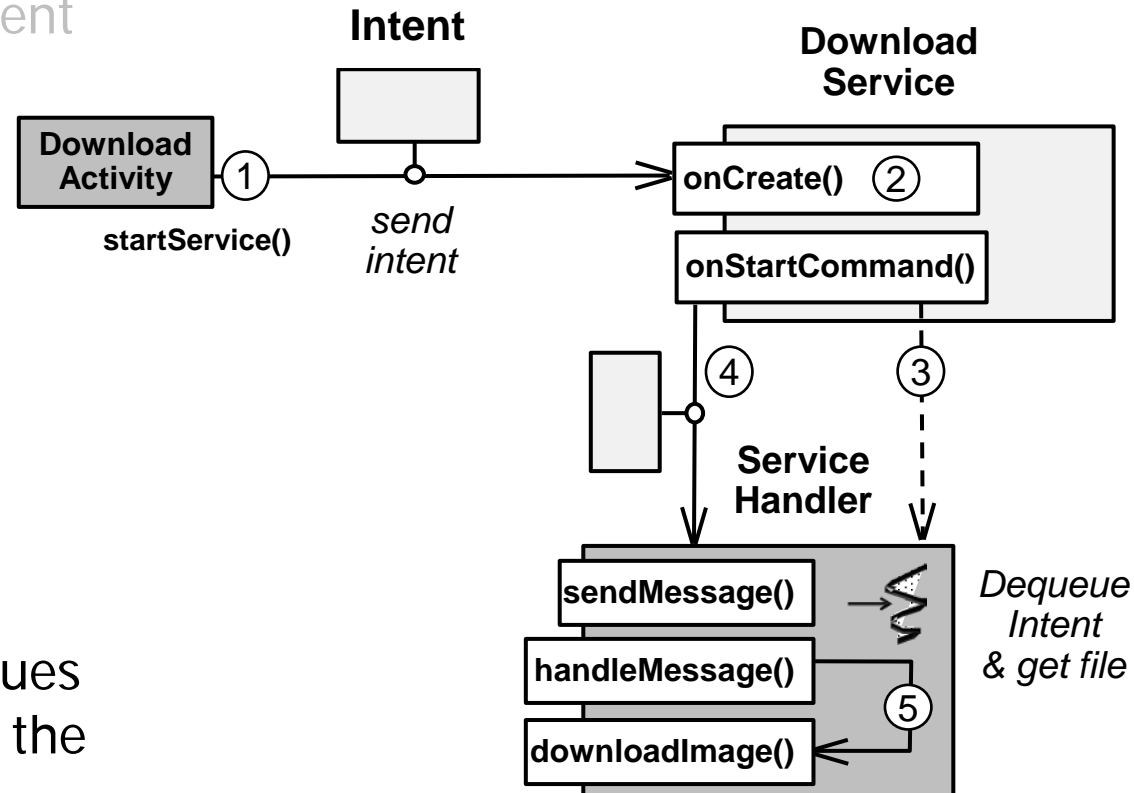
Download App Interactions

- DownloadActivity sends an Intent via a call to startService()
- The DownloadService is started on-demand
- The DownloadService does several things
 - Creates a ServiceHandler
 - Receives & queues Intents in the ServiceHandler



Download App Interactions

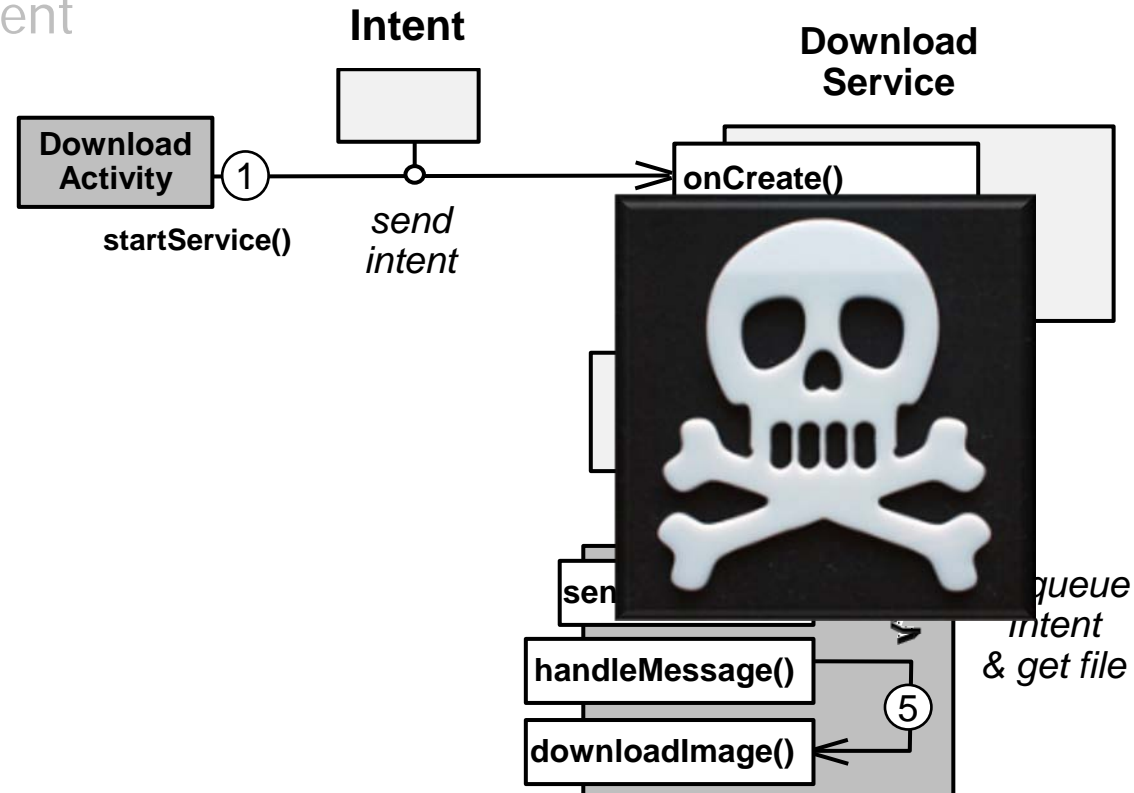
- DownloadActivity sends an Intent via a call to startService()
- The DownloadService is started on-demand
- The DownloadService does several things
 - Creates a ServiceHandler
 - Receives & queues Intents in the ServiceHandler
 - The ServiceHandler dequeues & processes the Intent "in the background" to download the designated image



Later we'll show how the DownloadService passes the file back to the Activity

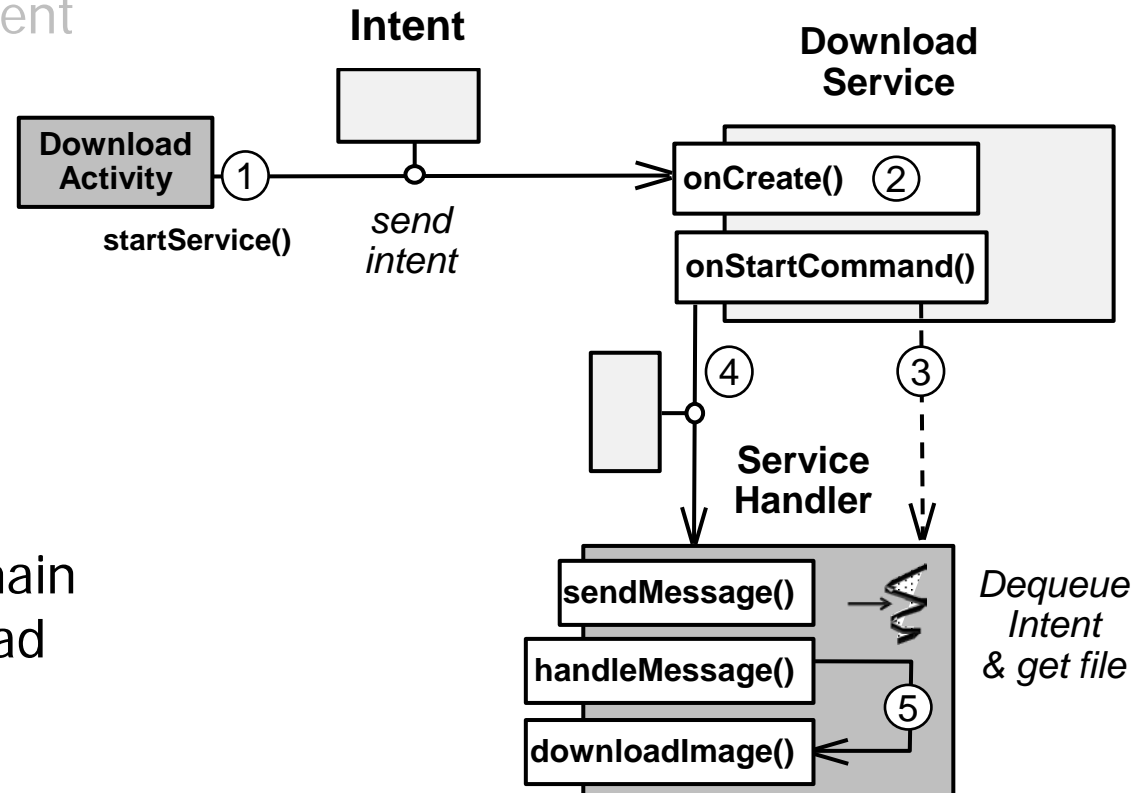
Download App Interactions

- DownloadActivity sends an Intent via a call to startService()
- The DownloadService is started on-demand
- The DownloadService does several things
 - Creates a ServiceHandler
 - Receives & queues Intents in the ServiceHandler
 - Stops the Service when there's no more Intents to handle







Download App Interactions

- DownloadActivity sends an Intent via a call to startService()
- The DownloadService is started on-demand
- The DownloadService does several things
- This implementation of the *Command Processor* pattern offloads tasks from an app's main thread to a single worker thread



Download Activity Implementation

```
public class DownloadActivity extends Activity {  
    ...  
    Create Intent associated with DownloadService   
    public void onClick(View v) {  
        Intent intent = new Intent(DownloadActivity.this,  
                                   DownloadService.class);  
        ...  Some initialization code intentionally omitted  
        intent.setData(Uri.parse(editText.getText().toString()));  
        Add the URI to the download as data   
        startService(intent);  Launch the Started Service  
        that handles this Intent  
    }  
    ...  
    Handler downloadHandler = new Handler() {  
        public void handleMessage(Message msg) { /* ... */ }  
    };  
     Code for processing the downloaded file shown later  
}
```



Download Service Implementation

```
public class DownloadService extends Service {  
    private volatile Looper mServiceLooper;  
    private volatile ServiceHandler mServiceHandler;
```

```
    public void onCreate() {  
        super.onCreate();
```

 **Create/start a separate Thread since the Service normally runs in the process's UI Thread, which we don't want to block**


```
        HandlerThread thread = new HandlerThread("DownloadService");  
        thread.start();
```


 **Get the HandlerThread's Looper & use it for our Handler**


```
        mServiceLooper = thread.getLooper();  
        mServiceHandler = new ServiceHandler(mServiceLooper);  
    }
```


Download Service Implementation

```
public class DownloadService extends Service {  
    ...  
    private final class ServiceHandler extends Handler {  
        public ServiceHandler(Looper looper) { super(looper); }  
  
        public void handleMessage(Message msg) {  
            downloadImage((Intent) msg.obj);  
            stopSelf(msg.arg1);  
        }  
  
        public void downloadImage(Intent intent) { /* ... */ }  
    }  
    ...  
}
```

 **Handler that receives messages from the thread**

 **Dispatch a callback hook method to download a file**

 **Stop the service using the startId, so that we don't stop the service in the middle of handling another job**

 **Download the image & notify the client**

Download Service Implementation

```
public class DownloadService extends Service {  
    ...  
    public int onStartCommand(Intent intent, int f, int startId) {
```

Include start ID in the message to know which request is being stopped when the download completes

```
    Message msg = mServiceHandler.obtainMessage();  
    msg.arg1 = startId;
```

For each Intent, create/send a message to start a download

```
    msg.obj = intent;  
    mServiceHandler.sendMessage(msg);  
    return START_NOT_STICKY;
```

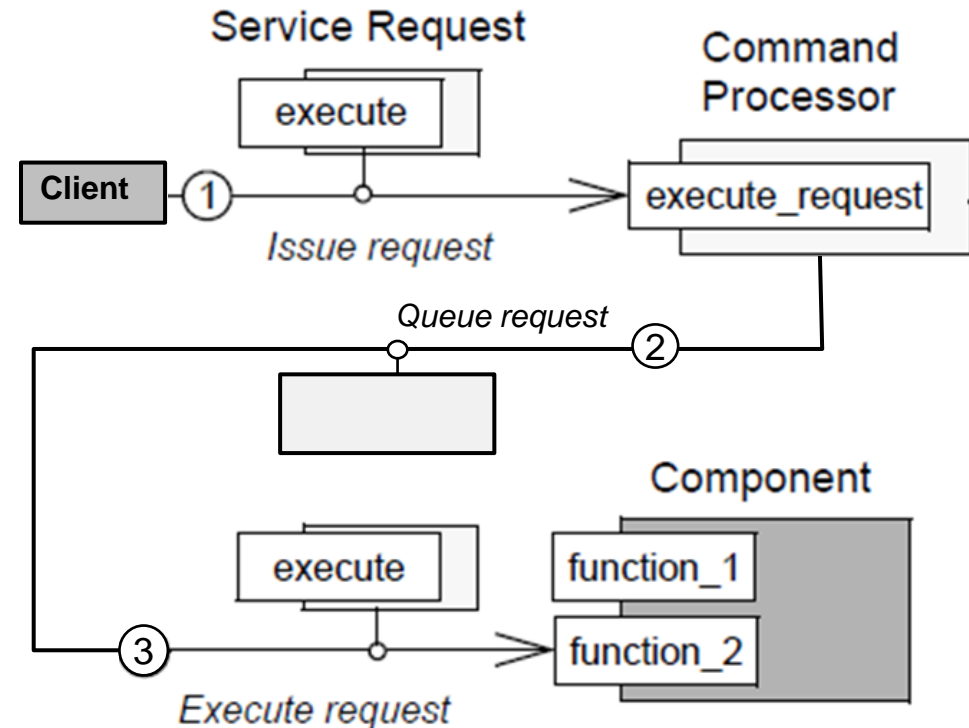
```
    public void onDestroy() {  
        mServiceLooper.quit();  
    }
```

```
}
```

It's instructive to consider how to extend this example to run in a thread pool

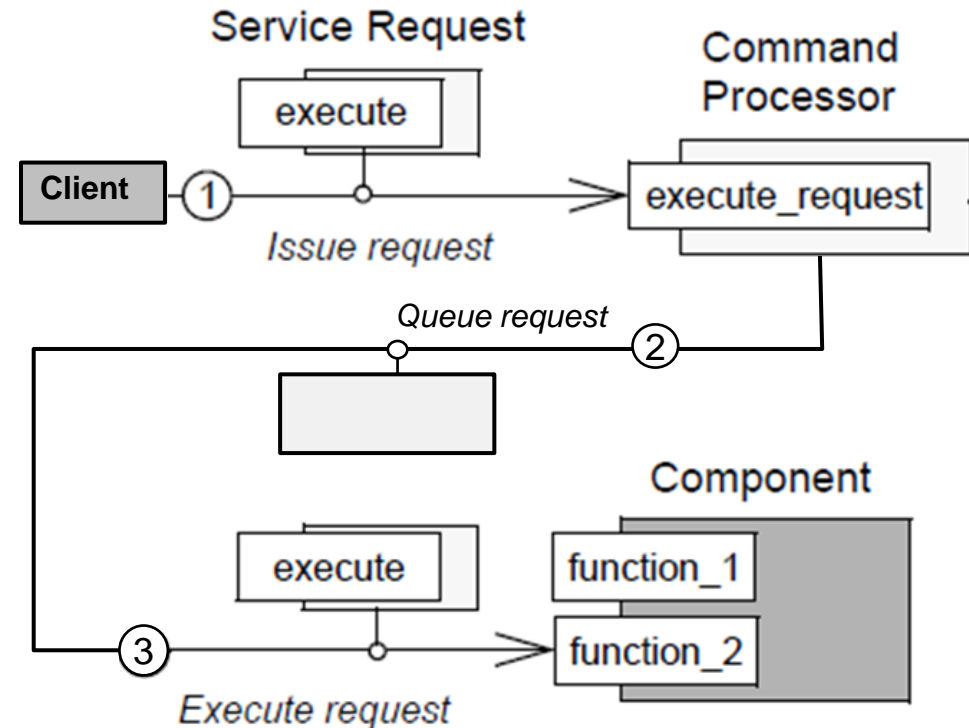
Analysis of the Download Service Example

- The worker thread solution shown here is a common Android Service idiom that implements the *Command Processor* pattern



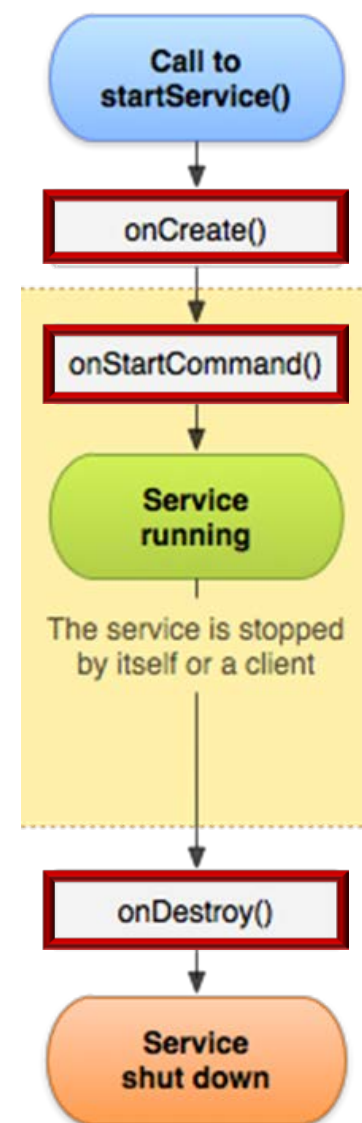
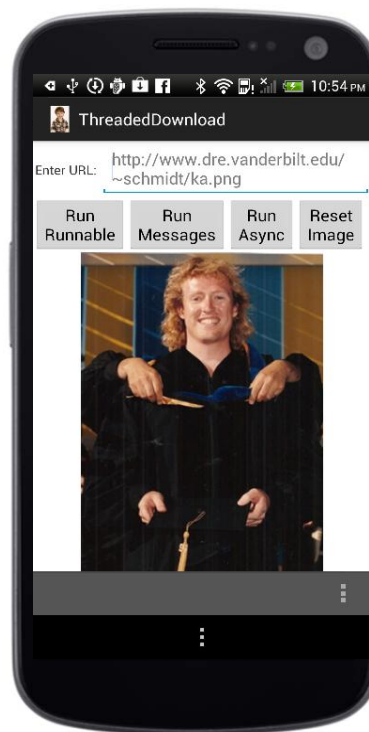
Analysis of the Download Service Example

- The worker thread solution shown here is a common Android Service idiom that implements the *Command Processor* pattern
- This pattern is a good option if you don't require that your service handle multiple requests simultaneously



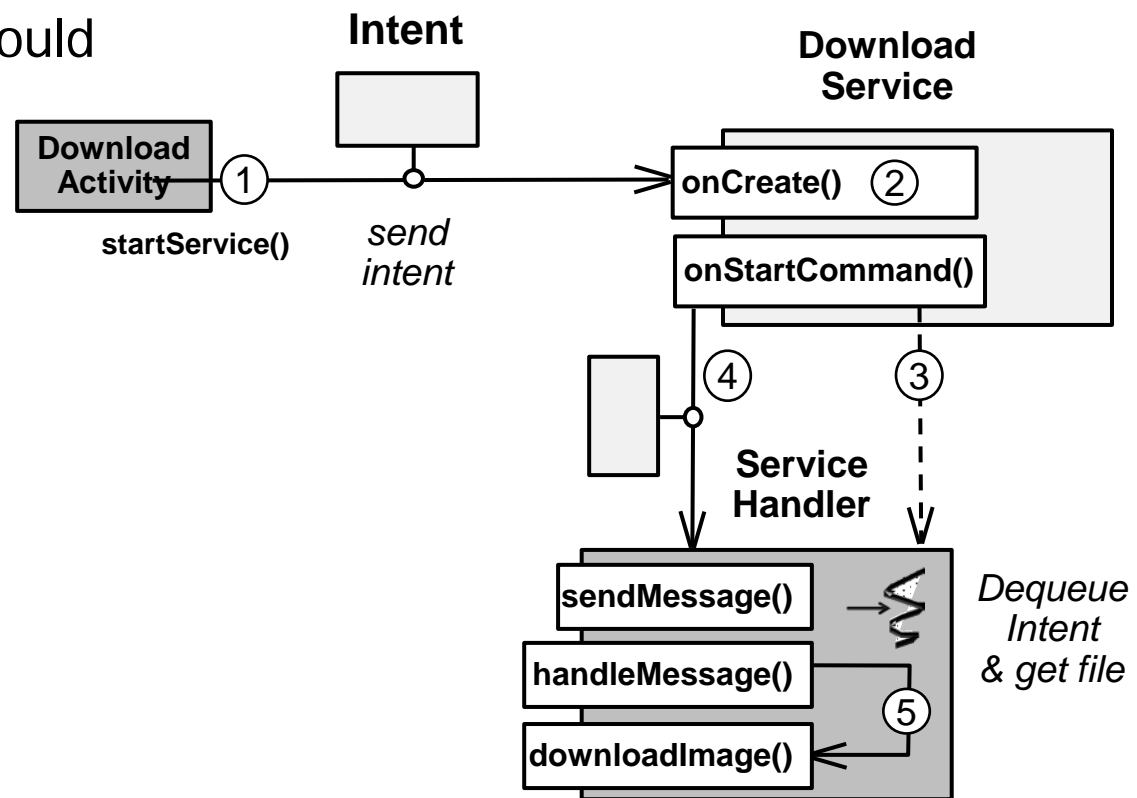
Summary

- Programming Started Services is relatively straightforward
 - e.g., inherit from Service & override various hook methods



Summary

- Programming Started Services is relatively straightforward
- The Service class uses the app's UI Thread by default
- A multi-threaded service should therefore often extend Service directly & spawn one or more threads



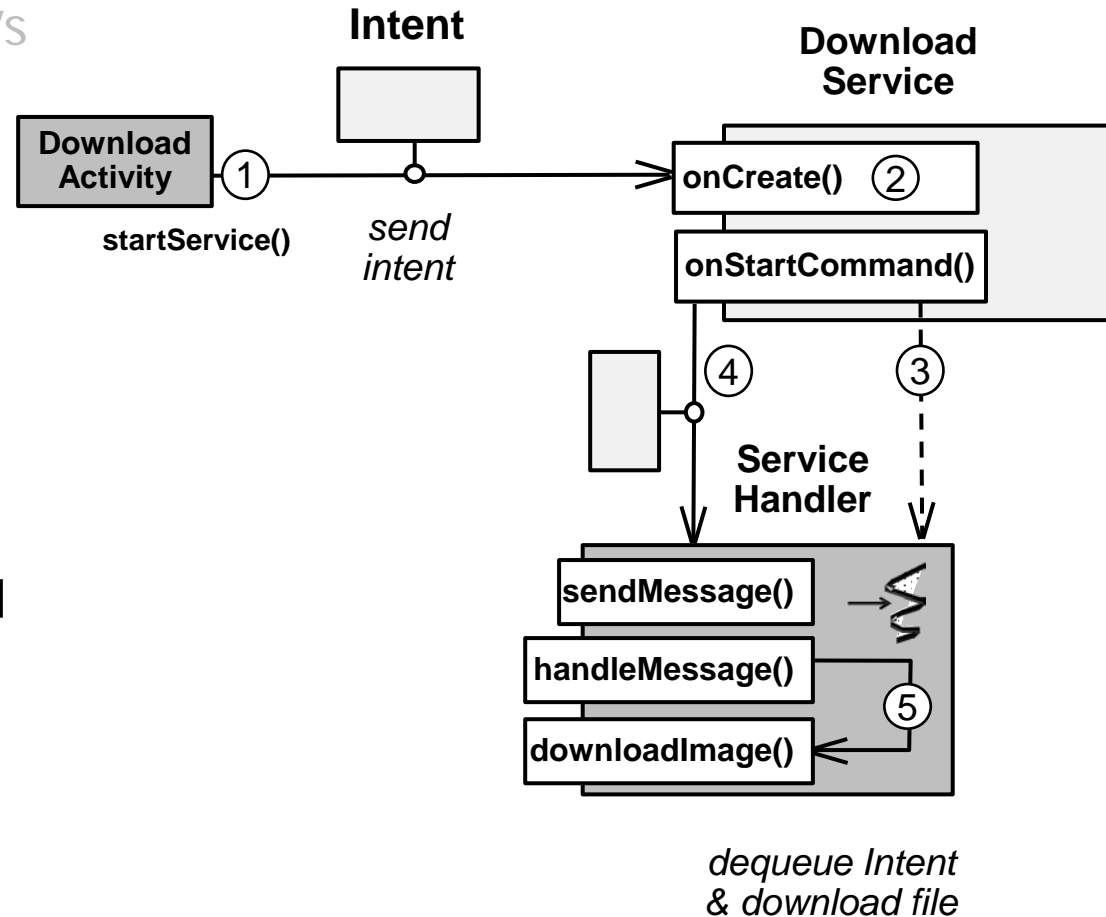
Summary

- Programming Started Services is relatively straightforward
- The Service class uses the app's UI Thread by default
- A Service is *not* a Thread
 - It doesn't automatically do work off the UI Thread & avoid "Application Not Responding" errors)



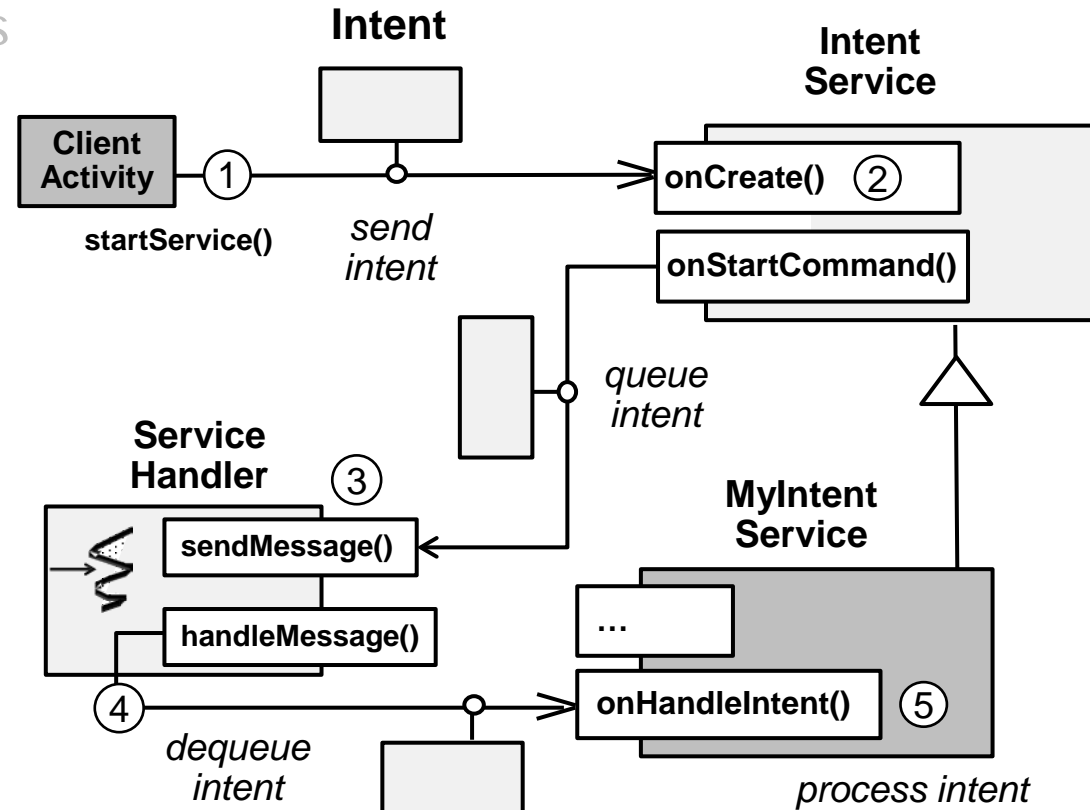
Summary

- Programming Started Services is relatively straightforward
- The Service class uses the app's UI Thread by default
- A Service is *not* a Thread
 - It doesn't automatically do work off the UI Thread & avoid "Application Not Responding" errors)
- A Service with compute- or I/O-intensive tasks should run its work in a background thread or process



Summary

- Programming Started Services is relatively straightforward
- The Service class uses the app's UI Thread by default
- A Service is *not* a Thread
 - It doesn't automatically do work off the UI Thread & avoid "Application Not Responding" errors)
- A Service with compute- or I/O-intensive tasks should run its work in a background thread or process
- The Android IntentService class automates this type of behavior via *Command Processor* pattern



Android Services & Local IPC: Overview of IntentService Framework

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



Overview of IntentService

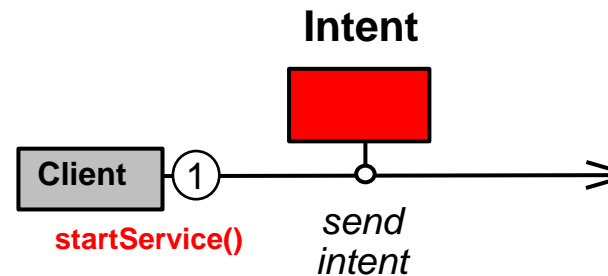
- The most common Service subclass is IntentService

```
public class IntentService extends Service {  
    public int onStartCommand(Intent intent,  
                               int flags,  
                               int startId) {  
  
        ...  
    }  
  
    protected abstract void onHandleIntent(Intent intent);  
}
```

This hook method must be implemented by subclasses to handle an Intent in a worker thread

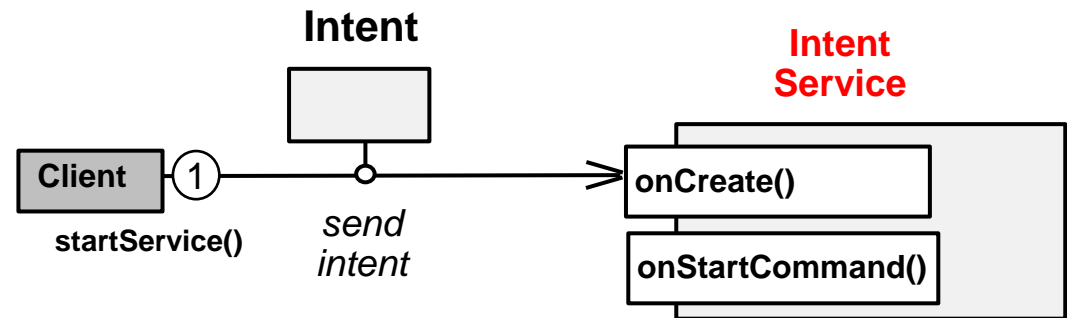
Overview of IntentService

- The most common Service subclass is IntentService
- Clients send Intents via calls to `startService()`
- Clients can pass data & objects to the Service by putting “extras” into the Intents



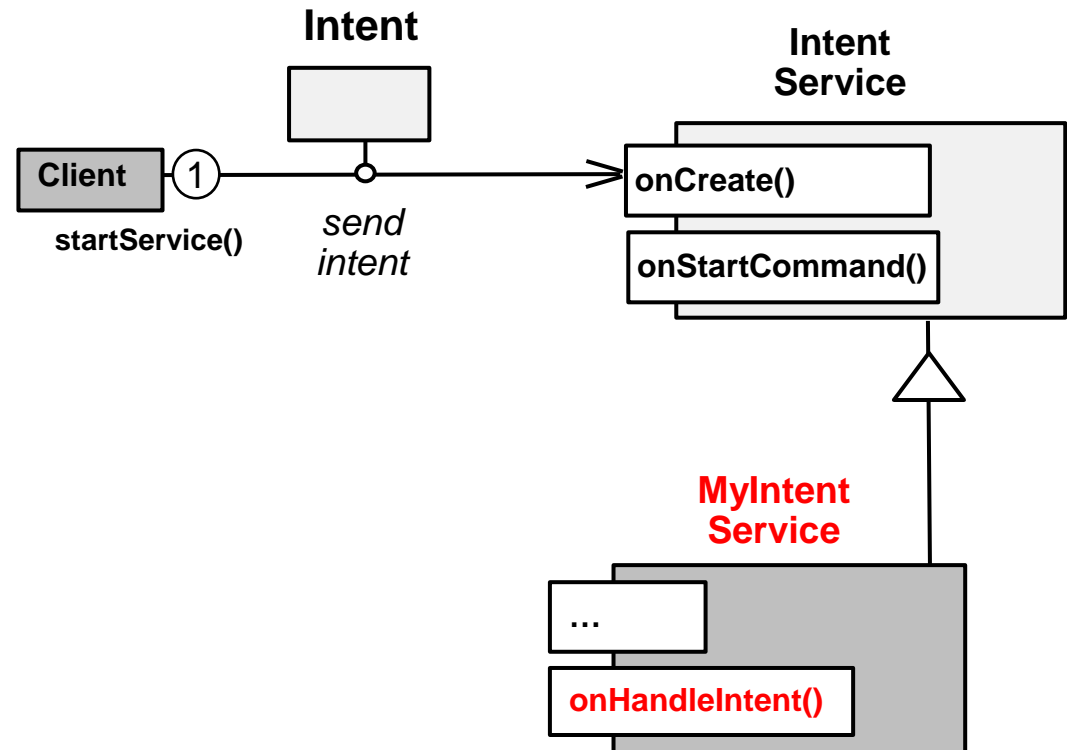
Overview of IntentService

- The most common Service subclass is IntentService
- Clients send Intents via calls to `startService()`
 - Clients can pass data & objects to the Service by putting “extras” into the Intents
- The IntentService is started on-demand via the *Activator* pattern



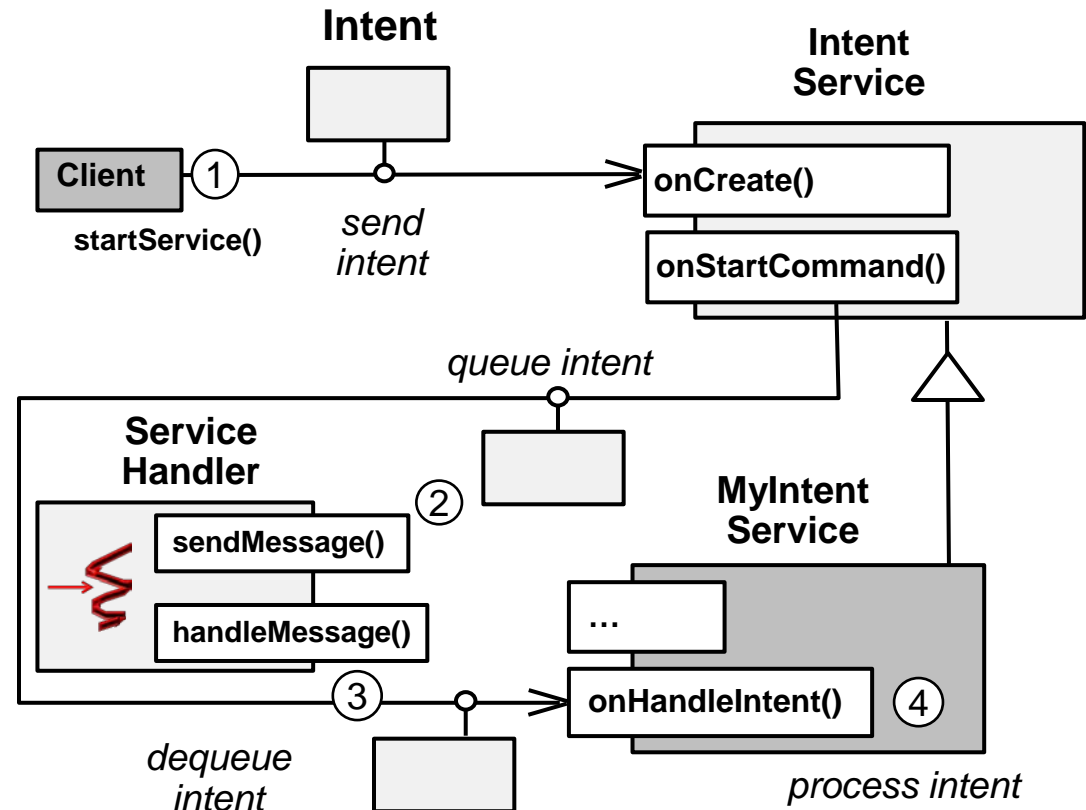
Overview of IntentService

- The most common Service subclass is IntentService
- Clients send Intents via calls to `startService()`
- A subclass of IntentService implements the hook method `onHandleIntent()`
- This hook method processes the Intent sent by the client



Overview of IntentService

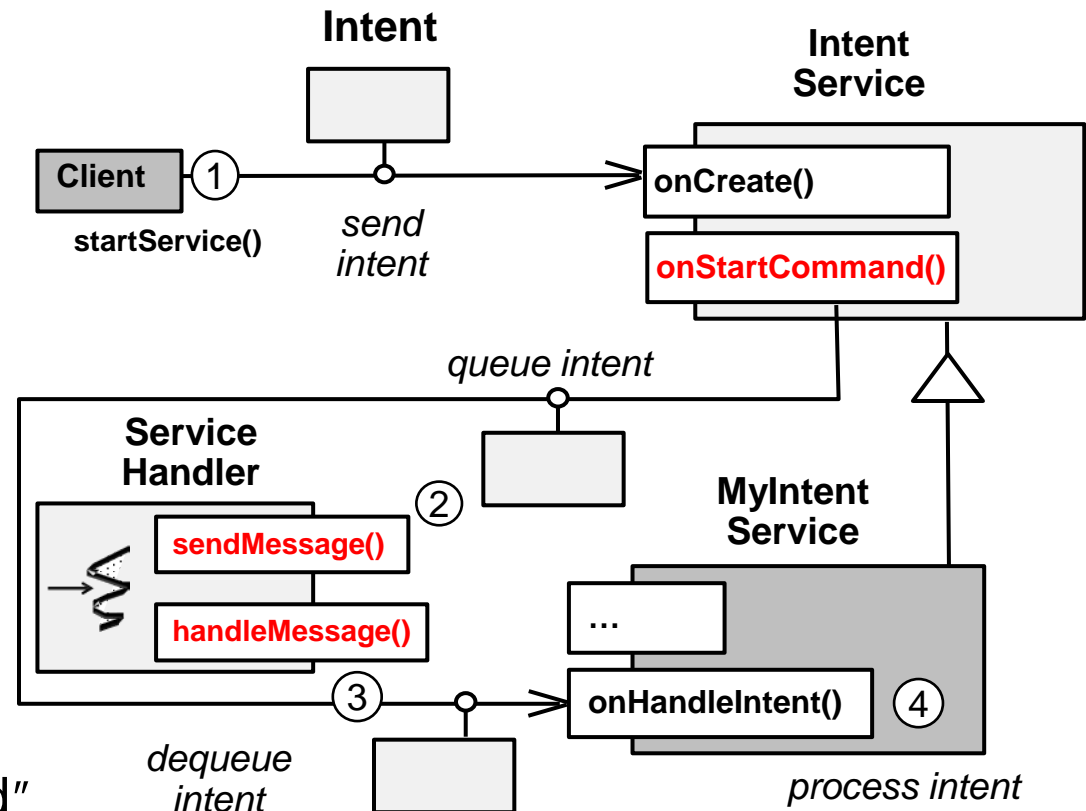
- The most common Service subclass is IntentService
- Clients send Intents via calls to `startService()`
- A subclass of IntentService implements the hook method `onHandleIntent()`
- The IntentService does several things
 - Creates a ServiceHandler
 - Internally creates a single worker thread



The ServiceHandler is a common idiom in multi-threaded Android Services

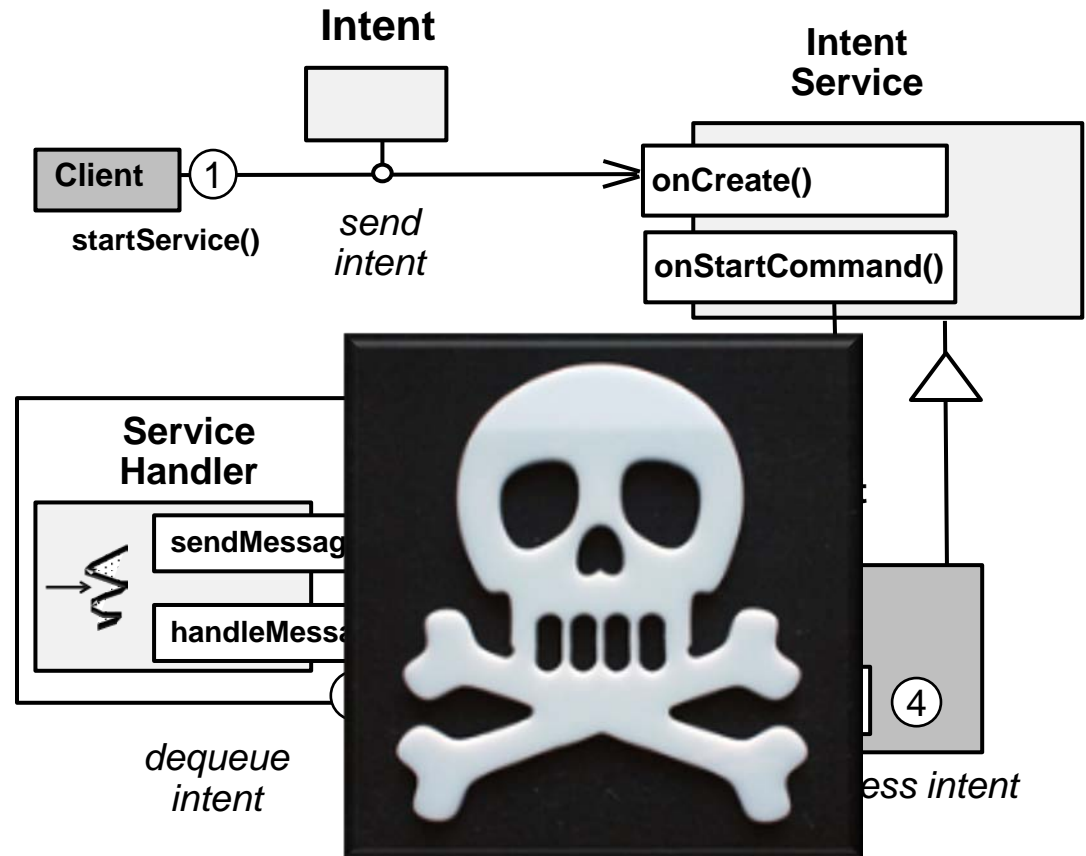
Overview of IntentService

- The most common Service subclass is IntentService
- Clients send Intents via calls to `startService()`
- A subclass of `IntentService` implements the hook method `onHandleIntent()`
- The `IntentService` does several things
 - Creates a `ServiceHandler`
 - Receives & queues Intents in `ServiceHandler`
 - Processes the queue of Intents “in the background”



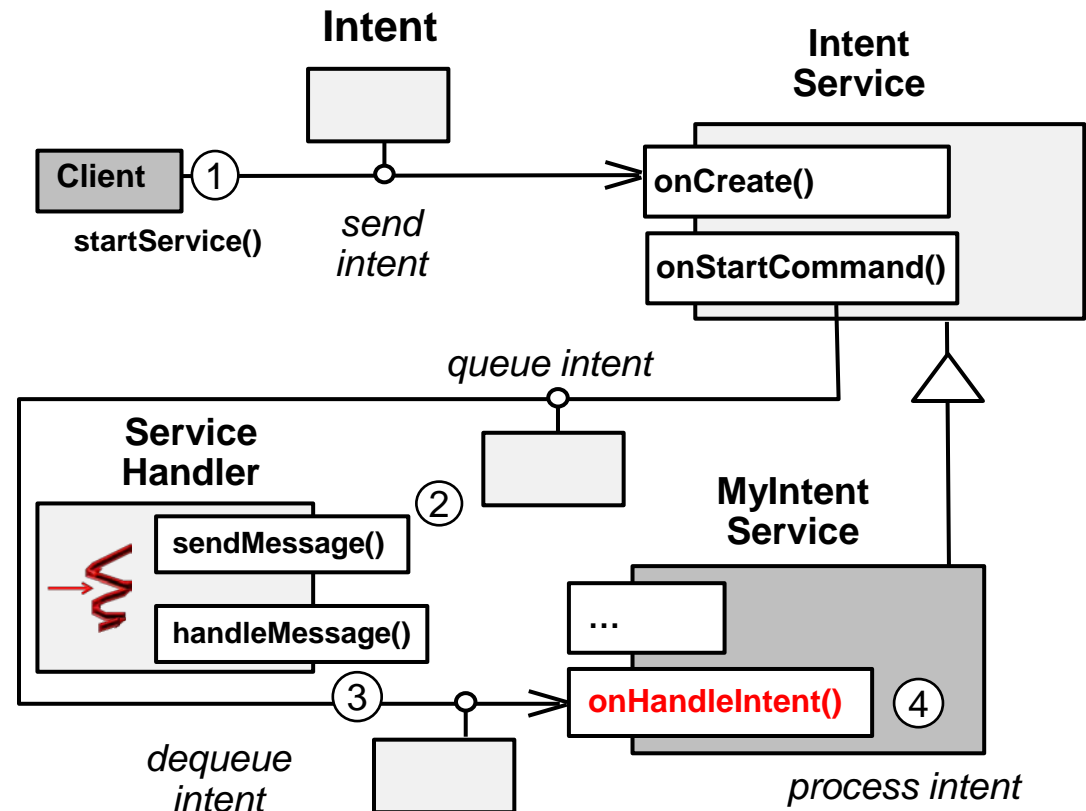
Overview of IntentService

- The most common Service subclass is IntentService
- Clients send Intents via calls to `startService()`
- A subclass of `IntentService` implements the hook method `onHandleIntent()`
- The `IntentService` does several things
 - Creates a `ServiceHandler`
 - Receives & queues Intents in `ServiceHandler`
 - Stops the Service when there are no more Intents to handle



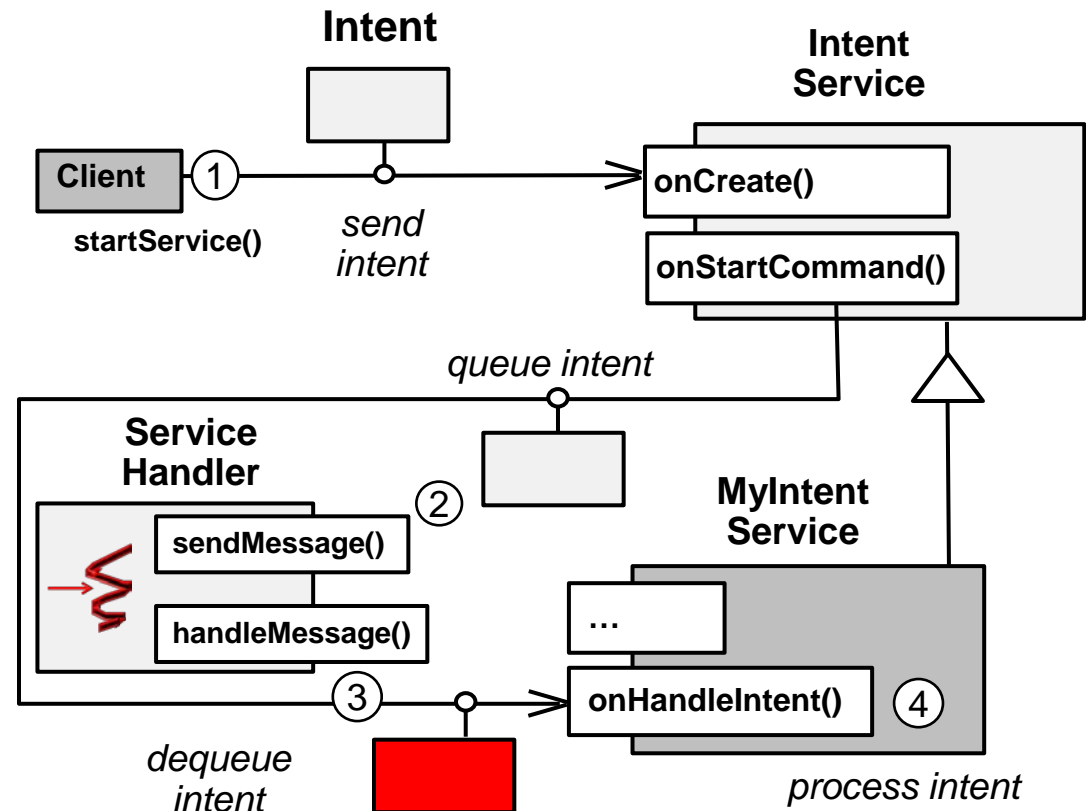
Overview of IntentService

- The most common Service subclass is IntentService
- Clients send Intents via calls to `startService()`
- A subclass of IntentService implements the hook method `onHandleIntent()`
- The IntentService does several things
- All Intents are handled in the ServiceHandler's worker thread
- They may take as long as necessary (& will not block the app's UI Thread loop)



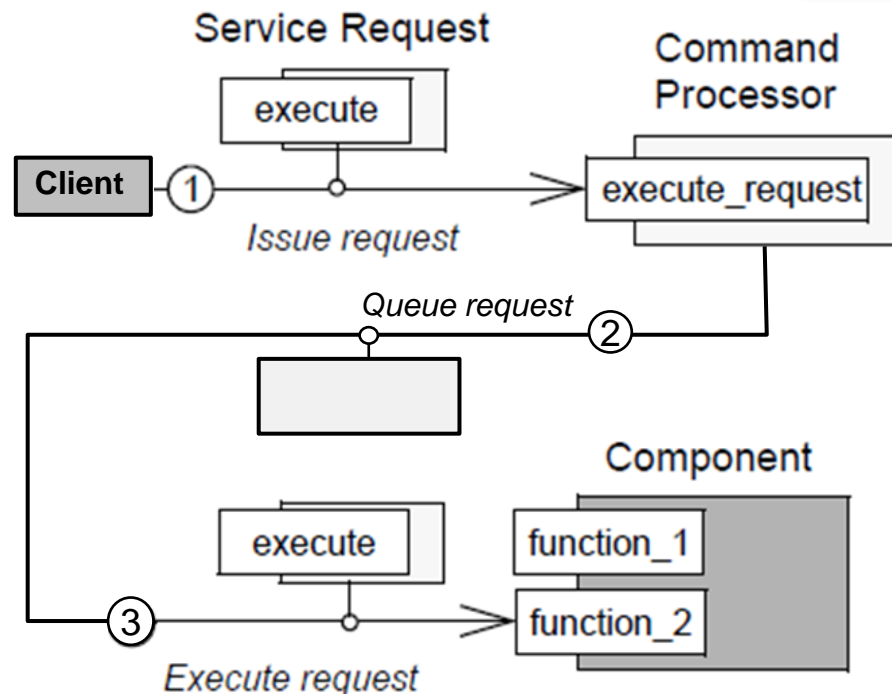
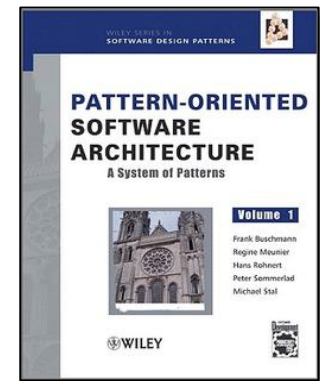
Overview of IntentService

- The most common Service subclass is IntentService
- Clients send Intents via calls to `startService()`
- A subclass of IntentService implements the hook method `onHandleIntent()`
- The IntentService does several things
- All Intents are handled in the ServiceHandler's worker thread
 - They may take as long as necessary (& will not block the app's UI Thread loop)
 - However, only one Intent will be processed at a time



Summary

- The IntentService is used to perform a certain task in the background
- The IntentService framework implements the *Command Processor* pattern



Summary

- The IntentService is used to perform a certain task in the background
- IntentService automatically stops itself when there are no more intents in its queue
- Conversely, a regular Service needs to stop itself manually via `stopSelf()` or `stopService()`



Android Services & Local IPC: Programming the IntentService Framework

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

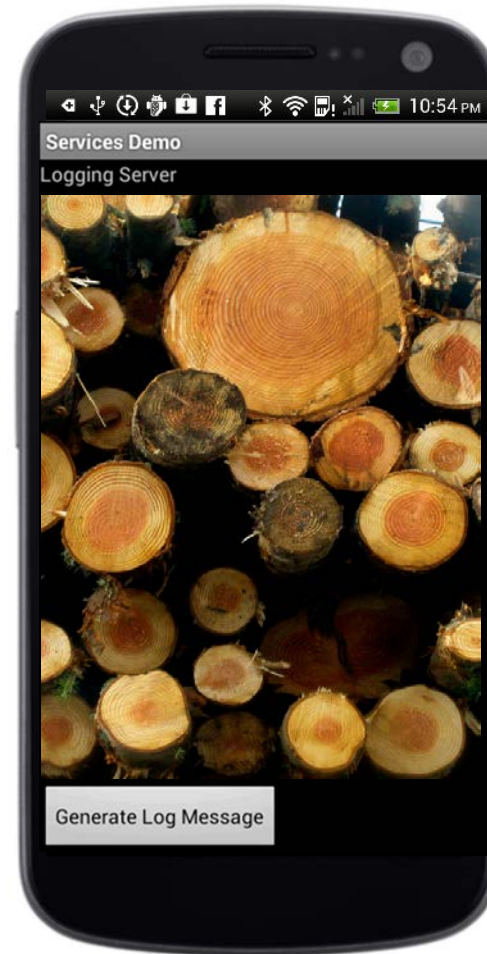
Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



Learning Objectives in this Part of the Module

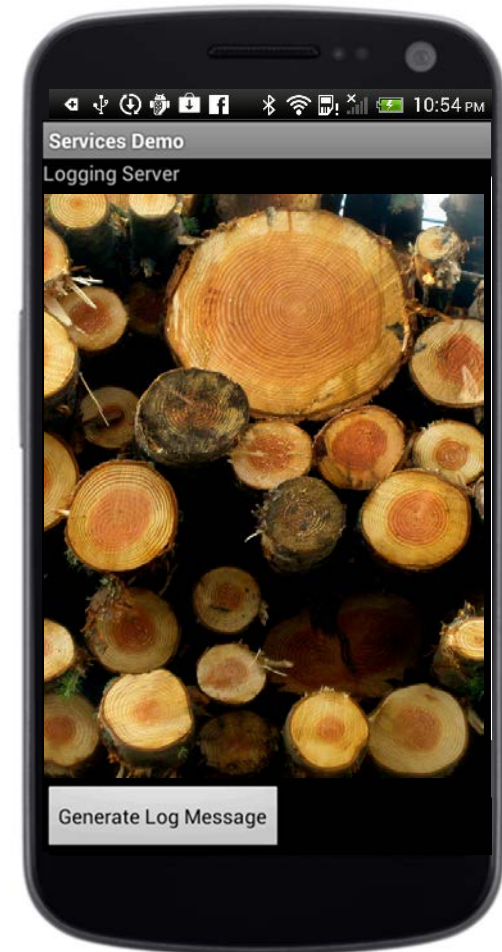
- Understand how to program the IntentService framework



Logging App Overview

- The Logging Service extends the IntentService to offload logging operations from an app's UI Thread

```
public class LoggingService
    extends IntentService {
    protected abstract void onHandleIntent
        (Intent intent);
}
```



L...	Time	PID	TID	Application	Tag	Text
I	09-19 12:...	612	612	course.examples.Ser...	Logging...	Service destroyed
I	09-19 12:...	612	631	course.examples.Ser...	Logging...	Log this message
I	09-19 12:...	612	612	course.examples.Ser...	Logging...	Service created
I	09-19 12:...	612	612	course.examples.Ser...	Logging...	Service command started

Logging App Overview

- The Logging Service extends the IntentService to offload logging operations from an app's UI Thread
- Clients send commands (expressed as Intents) via calls to `startService()`

```
Intent intent = new Intent  
    (this, LoggingService.class);  
intent.putExtra("LogMsg", "hello world");  
startService(intent);
```

**Download
Activity**



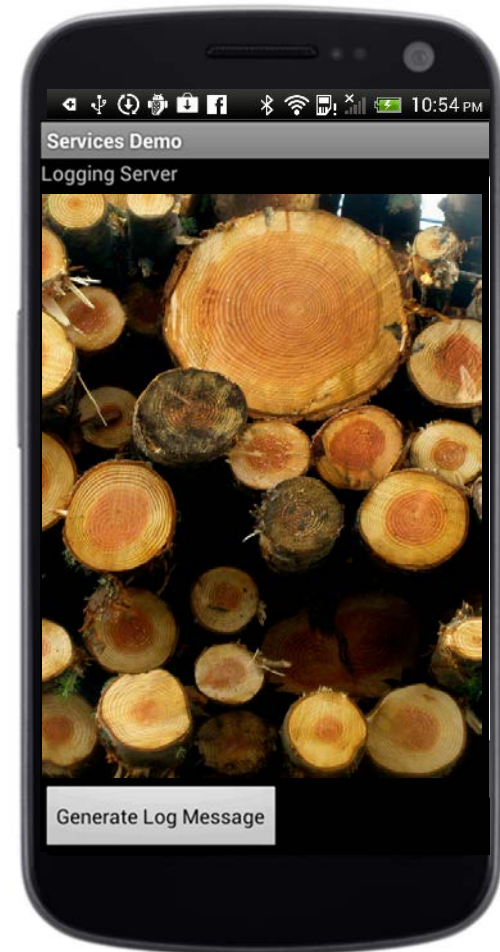
Logging App Overview

- The Logging Service extends the IntentService to offload logging operations from an app's UI Thread
- Clients send commands (expressed as Intents) via calls to startService()
- The LoggingService subclass handle intents in a worker thread asynchronously

```
public class LoggingService extends  
    IntentService {  
    void onHandleIntent(Intent intent)  
    { ... }  
}
```

Download
Activity

Download
Service



Logging App Overview

- The Logging Service extends the IntentService to offload logging operations from an app's UI Thread
- Clients send commands (expressed as Intents) via calls to startService()
- The LoggingService subclass handles intents in a worker thread asynchronously

```
public class LoggingService extends  
    IntentService {  
    void onHandleIntent(Intent intent)  
    { ... }  
}
```

Handler



Download
Activity

Download
Service

*Android starts the Service as needed,
which internally spawns a worker
thread that handles a queue of intents*



Logging App Overview

- The Logging Service extends the IntentService to offload logging operations from an app's UI Thread
- Clients send commands (expressed as Intents) via calls to startService()
- The LoggingService subclass handles intents in a worker thread asynchronously

```
public class LoggingService extends  
    IntentService {  
    void onHandleIntent(Intent intent)  
    { ... }  
}
```

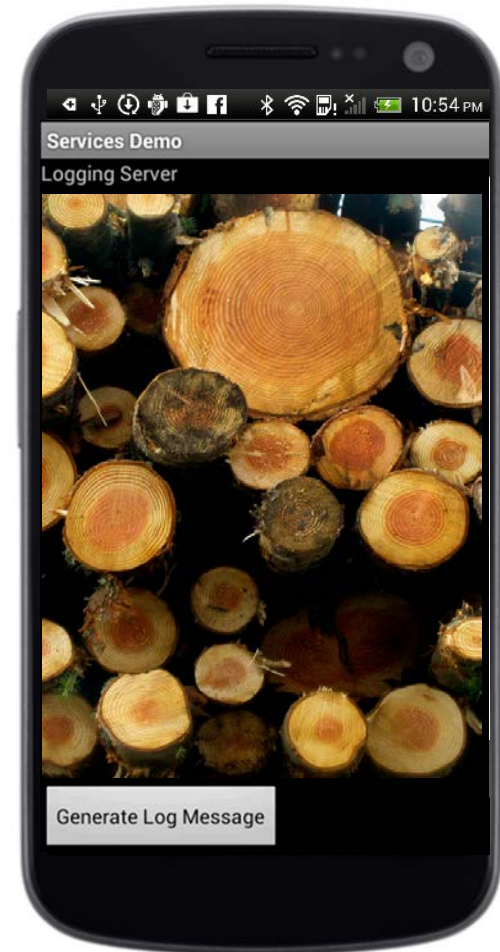
Handler



Download
Activity

Download
Service

The IntentService calls this hook method from the worker thread to handle each intent that started the Service



Logging App Overview

- The Logging Service extends the IntentService to offload logging operations from an app's UI Thread
- Clients send commands (expressed as Intents) via calls to startService()
- The LoggingService subclass handle intents in a worker thread asynchronously

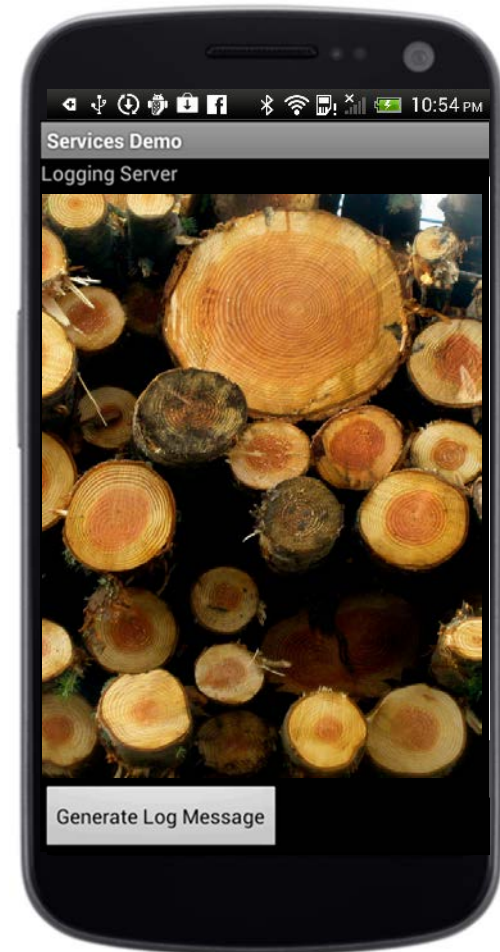
```
public class LoggingService extends  
    IntentService {  
    void onHandleIntent(Intent intent)  
    { ... }  
}
```

Handler



Download
Activity

Download
Service



*When there are no more intents to handle
the IntentService stops itself automatically*

Logging Activity Implementation

```
public class BGLoggingActivity extends Activity {  
    public void onCreate(Bundle savedInstanceState) {  
        ...  
        buttonStart.setOnClickListener(new OnClickListener() {  
  
            public void onClick(View v) {  
                Intent intent = new Intent(BGLoggingActivity.this,  
                    BGLoggingService.class);  
  
                intent.putExtra("LogMsg",  
                    "Log this message");  
  
                startService(intent);  
            }  
        });  
    }  
}
```

Create a new Intent 

Add the message to log as an "extra" 

Launch the Started Service that handles this Intent 

Logging Service Implementation

```
public class BGLoggingService extends IntentService {  
    ...
```

Inherit from IntentService class

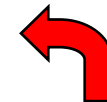


```
    public int onStartCommand(Intent intent, int flags,  
                               int startId) {  
        super.onStartCommand(intent, flags, startId);  
        return START_NOT_STICKY;  
    }
```



Don't restart this Service if it's shutdown

```
    protected void onHandleIntent(Intent intent) {  
        ...  
        Log.i(TAG, intent.getCharSequenceExtra  
              ("LogMsg").toString());  
    }  
    ...  
}
```



This hook method
runs in a worker
thread & logs the data

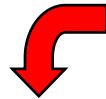
Note the "inversion of control" in the Android Service framework



AndroidManifest.xml File

```
<application ... >
```

```
    <activity android:name=".BGLoggingActivity"
              android:label="@string/app_name">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name=
                "android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
```



Service is only usable by
components in this application

```
    <service android:exported="false"
            android:name=".BGLoggingService" />
```

```
</application>
```

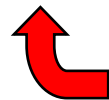
AndroidManifest.xml File

```
<application ... >
```

```
    <activity android:name=".BGLoggingActivity"
              android:label="@string/app_name">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name=
                "android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
```

```
    <service android:exported="false"
             android:name=".BGLoggingService"
             android:process=":myProcess" />
```

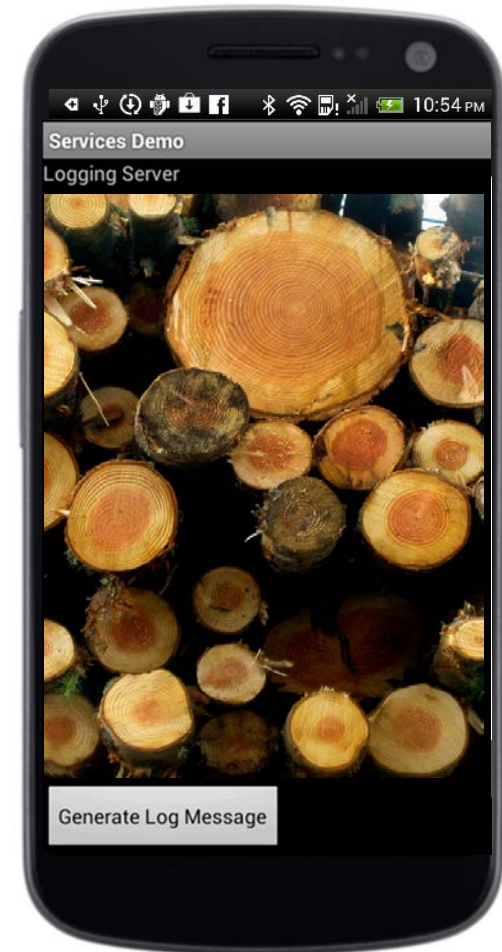
```
</application
```



Instruct Android to run the
BGLoggingService in its own process

Analysis of the Logging Service Example

- The LoggingService is an intentionally simplified example



L...	Time	PID	TID	Application	Tag	Text
I	09-19 12:...	612	612	course.examples.Ser...	Logging...	Service destroyed
I	09-19 12:...	612	631	course.examples.Ser...	Logging...	Log this message
I	09-19 12:...	612	612	course.examples.Ser...	Logging...	Service created
I	09-19 12:...	612	612	course.examples.Ser...	Logging...	Service command started

Analysis of the Logging Service Example

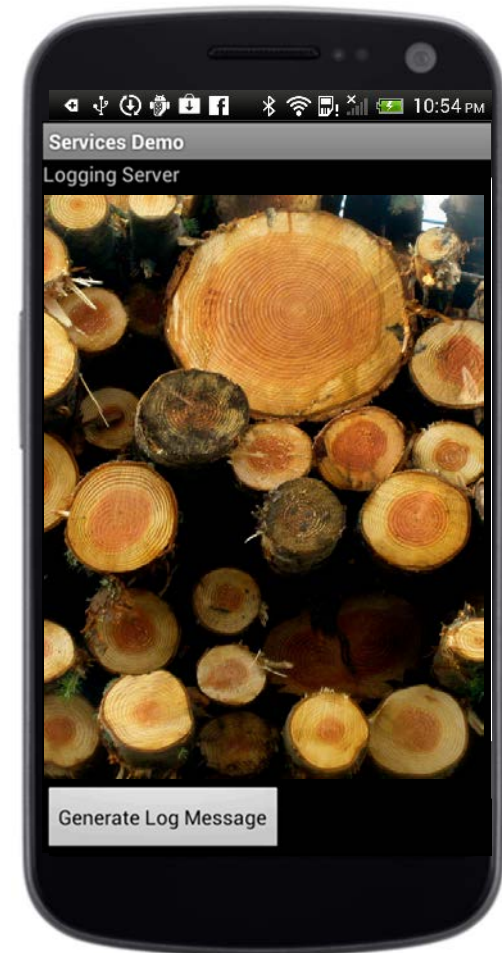
- The LoggingService is an intentionally simplified example
- You don't need to implement it as an IntentService (or even as a Service)
 - You could simply do the logging in a new Thread or ignore concurrency altogether!



L...	Time	PID	TID	Application	Tag	Text
I	09-19 12:...	612	612	course.examples.Ser...	Logging...	Service destroyed
I	09-19 12:...	612	631	course.examples.Ser...	Logging...	Log this message
I	09-19 12:...	612	612	course.examples.Ser...	Logging...	Service created
I	09-19 12:...	612	612	course.examples.Ser...	Logging...	Service command started

Analysis of the Logging Service Example

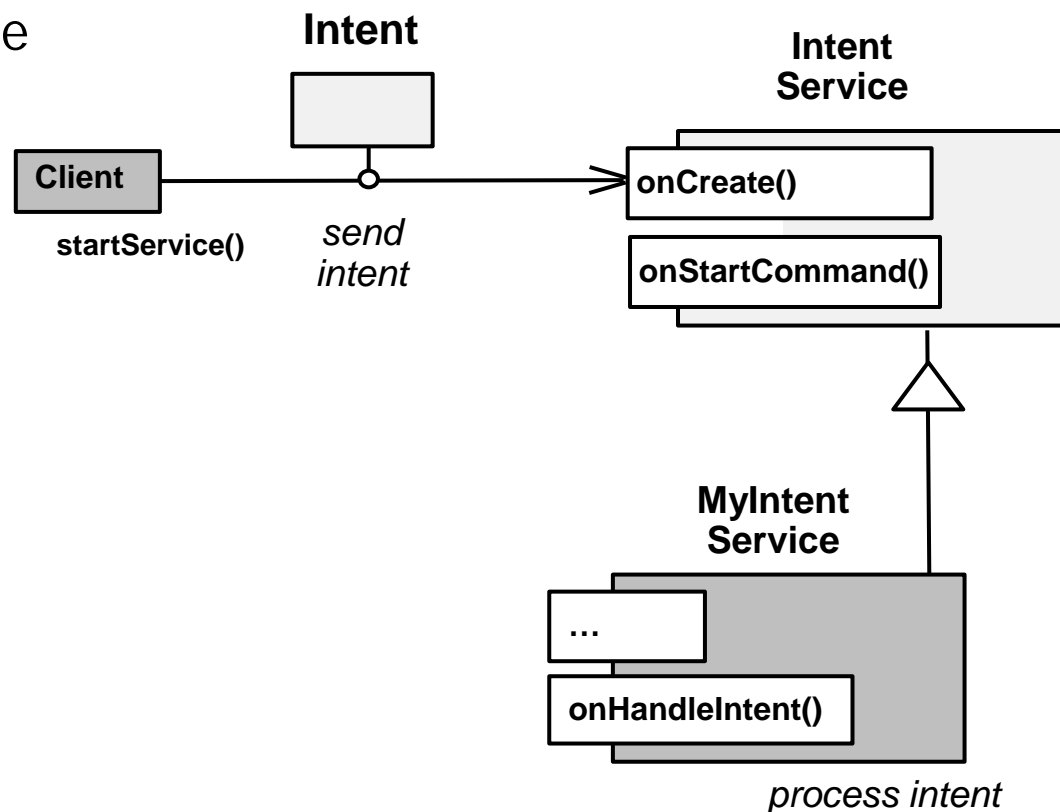
- The LoggingService is an intentionally simplified example
- You don't need to implement it as an IntentService (or even as a Service)
- In general, use a Service (or IntentService) when you want to run a component even when a user is not interacting with the app that hosts the Service



L...	Time	PID	TID	Application	Tag	Text
I	09-19 12:...	612	612	course.examples.Ser...	Logging...	Service destroyed
I	09-19 12:...	612	631	course.examples.Ser...	Logging...	Log this message
I	09-19 12:...	612	612	course.examples.Ser...	Logging...	Service created
I	09-19 12:...	612	612	course.examples.Ser...	Logging...	Service command started

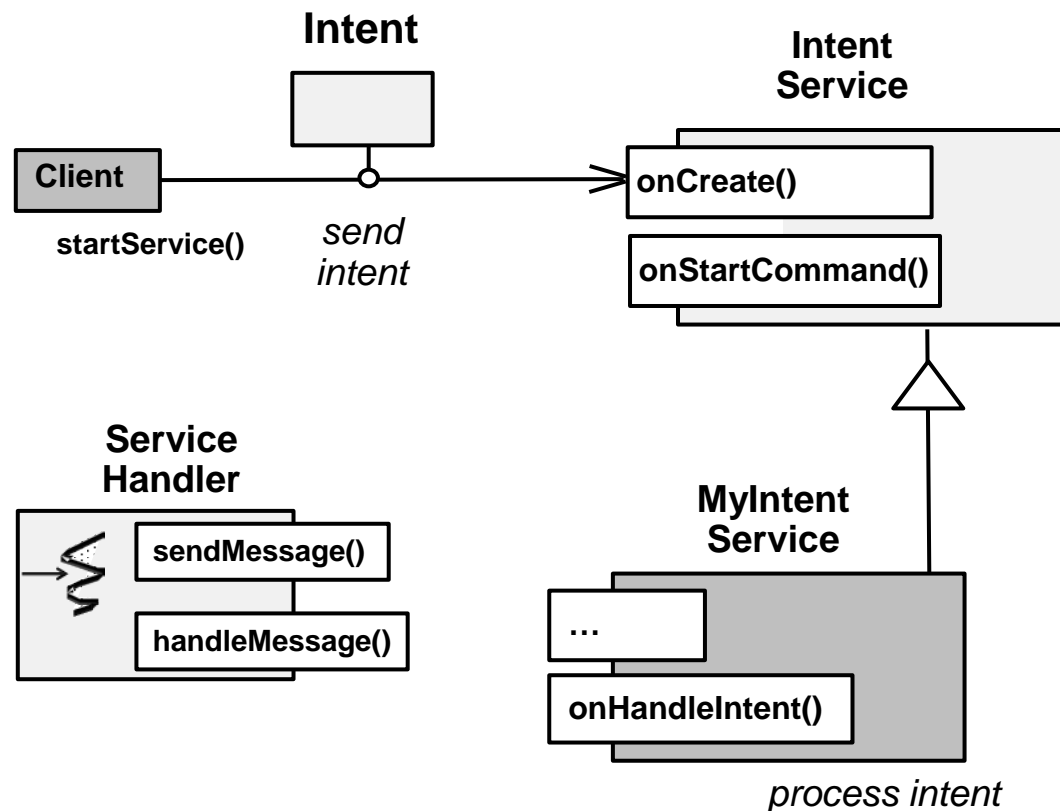
Summary

- Programming Intent Services is very straightforward
 - e.g., inherit from IntentService & override onHandleIntent()



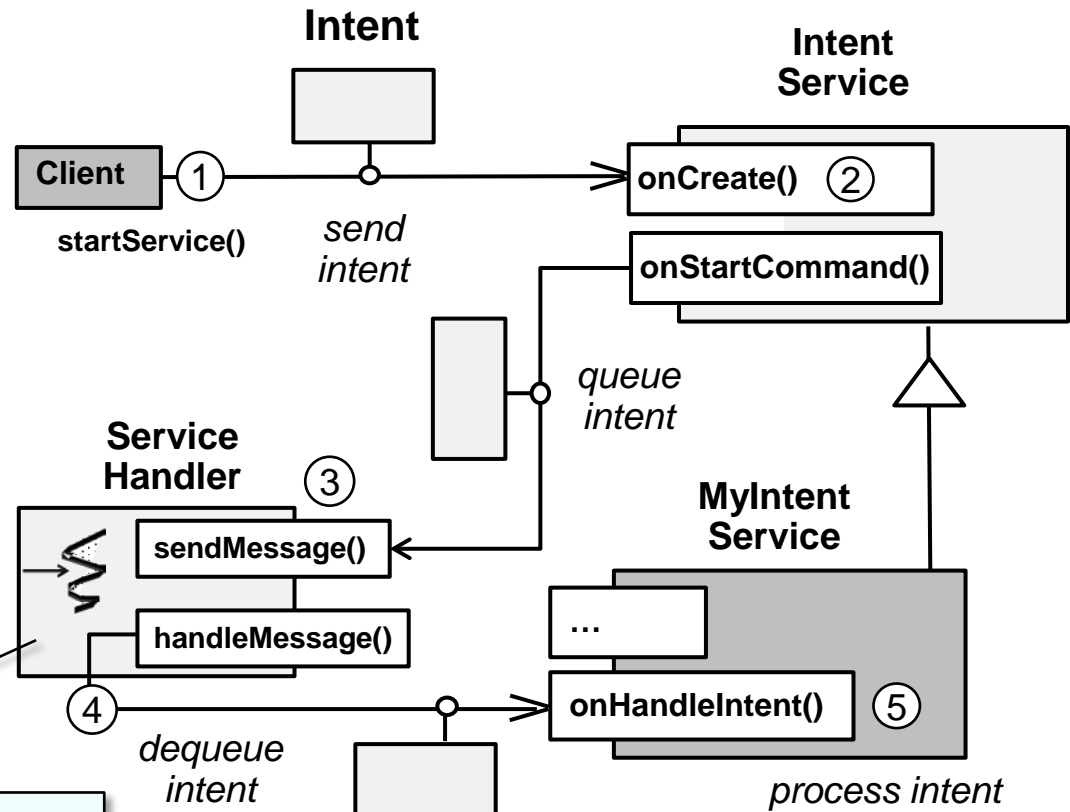
Summary

- Programming Intent Services is very straightforward
- IntentService creates a worker thread & uses that thread to run the service



Summary

- Programming Intent Services is very straightforward
- IntentService creates a worker thread & uses that thread to run the service
- IntentService also creates a queue that passes one intent at a time to onHandleIntent()



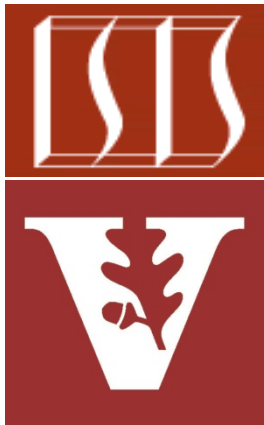
IntentService is the best option if you don't require that your service handle multiple requests simultaneously

Android Services & Local IPC: Communicating from Started Services to Activities via Messengers

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

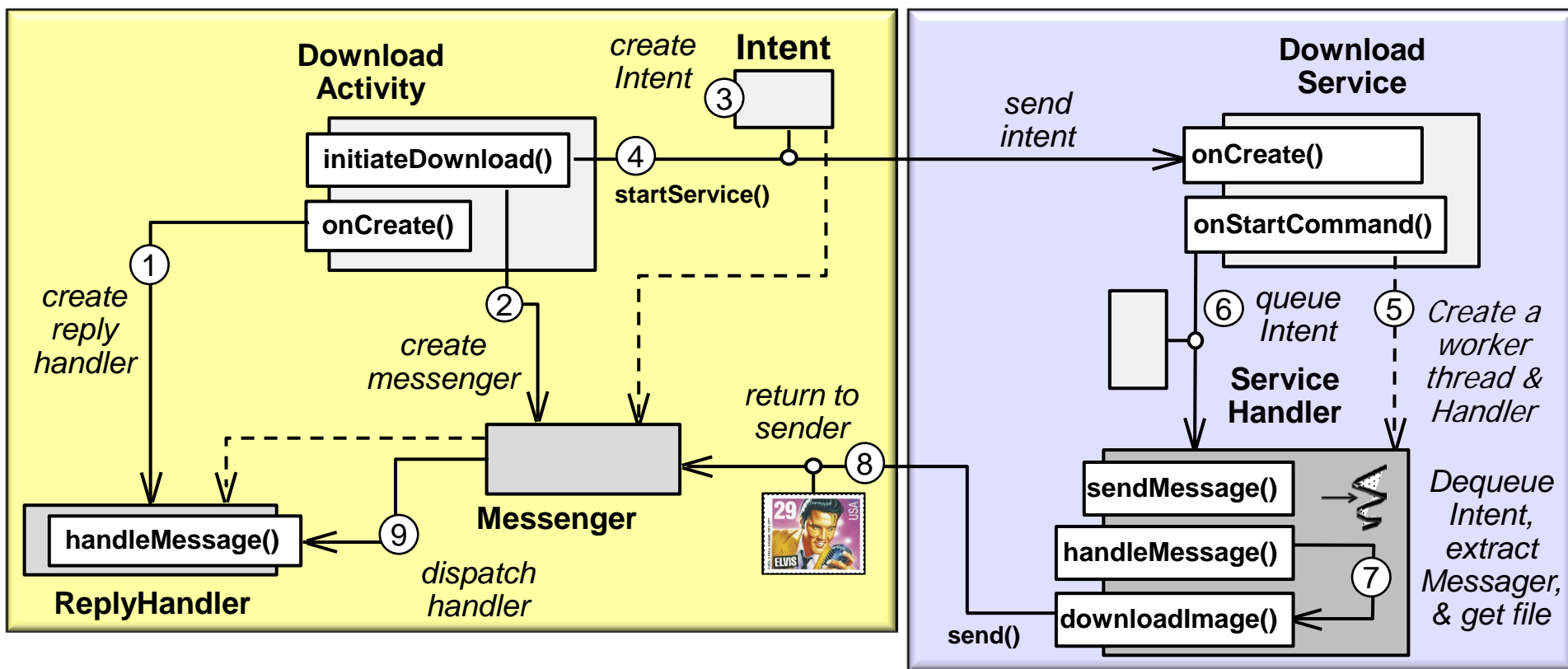
Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



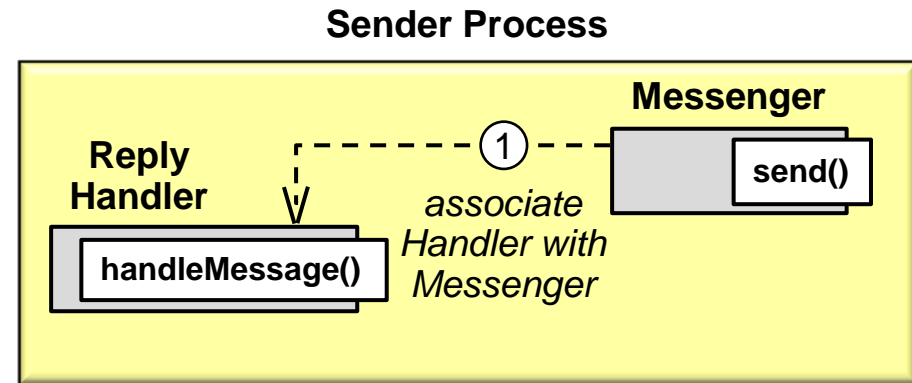
Learning Objectives in this Part of the Module

- Understand how to use Messengers to communicate from Started Services back to their invoking Activities
- Provides an interface for IPC with remote processes without using AIDL



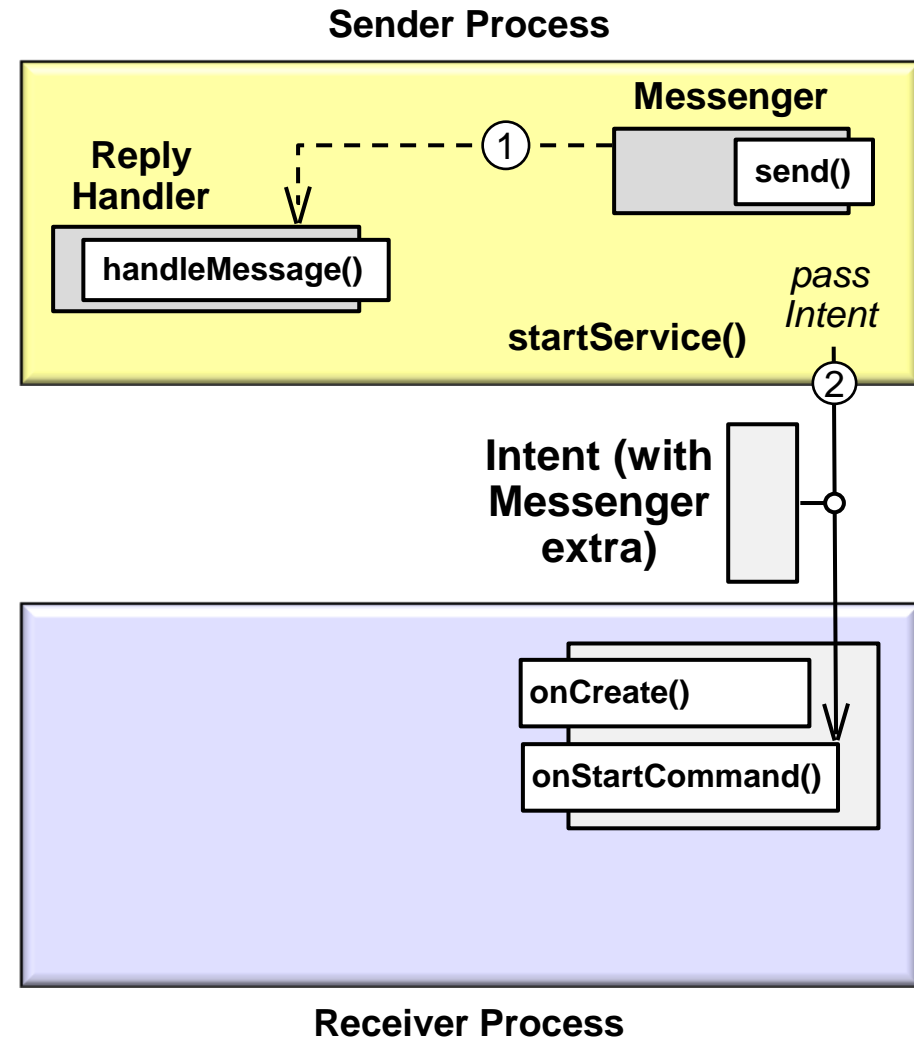
Overview of Messengers

- A Messenger provides a reference to a Handler that others can use to send messages to it



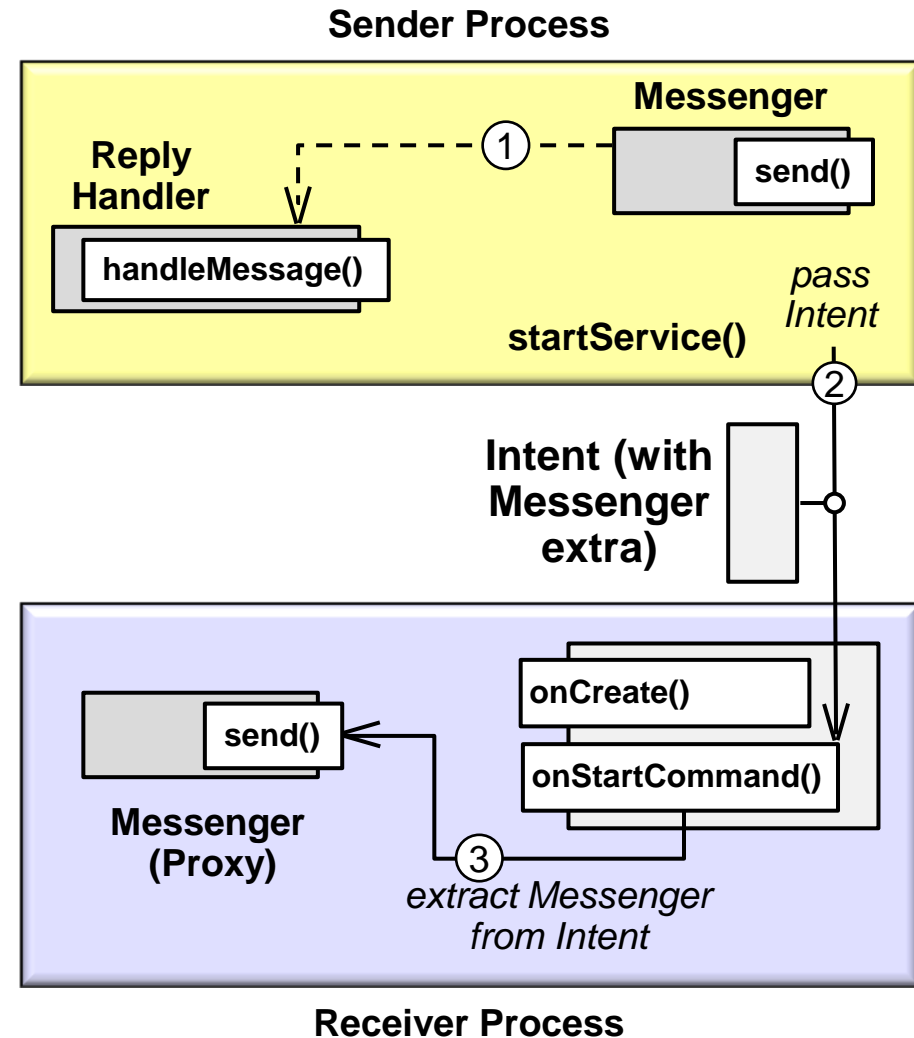
Overview of Messengers

- A Messenger provides a reference to a Handler that others can use to send messages to it
- An Activity can create a Messenger pointing to a Handler in one process & then pass that Messenger to another process



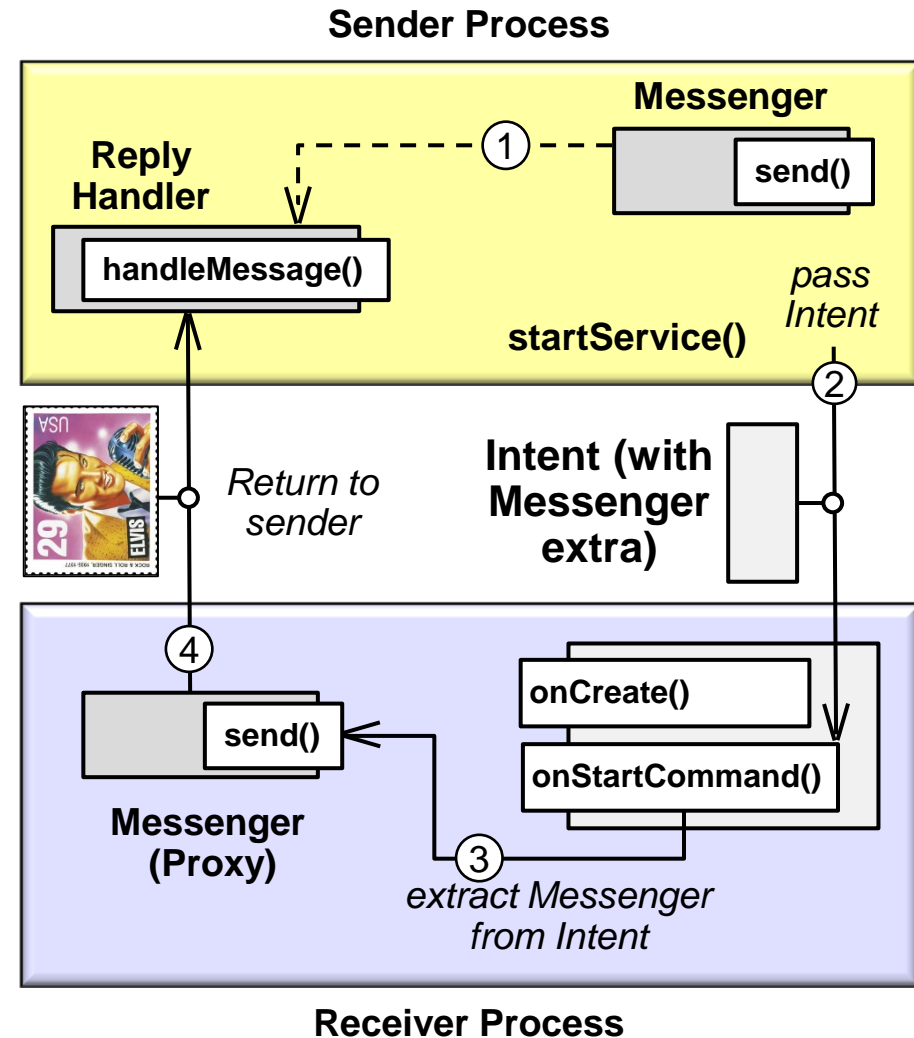
Overview of Messengers

- A Messenger provides a reference to a Handler that others can use to send messages to it
- An Activity can create a Messenger pointing to a Handler in one process & then pass that Messenger to another process
- The receiver then does several things
 - Obtains the Messenger



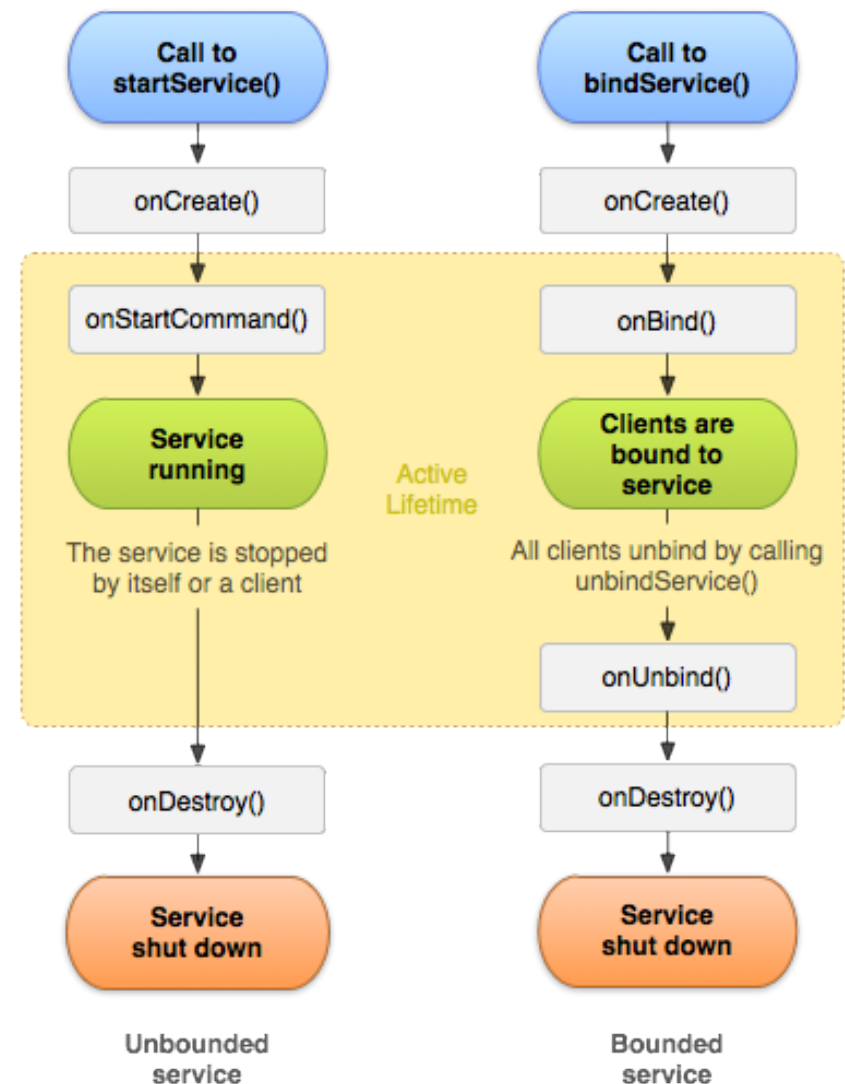
Overview of Messengers

- A Messenger provides a reference to a Handler that others can use to send messages to it
- An Activity can create a Messenger pointing to a Handler in one process & then pass that Messenger to another process
- The receiver then does several things
 - Obtains the Messenger
 - Returns the results back to the sender process



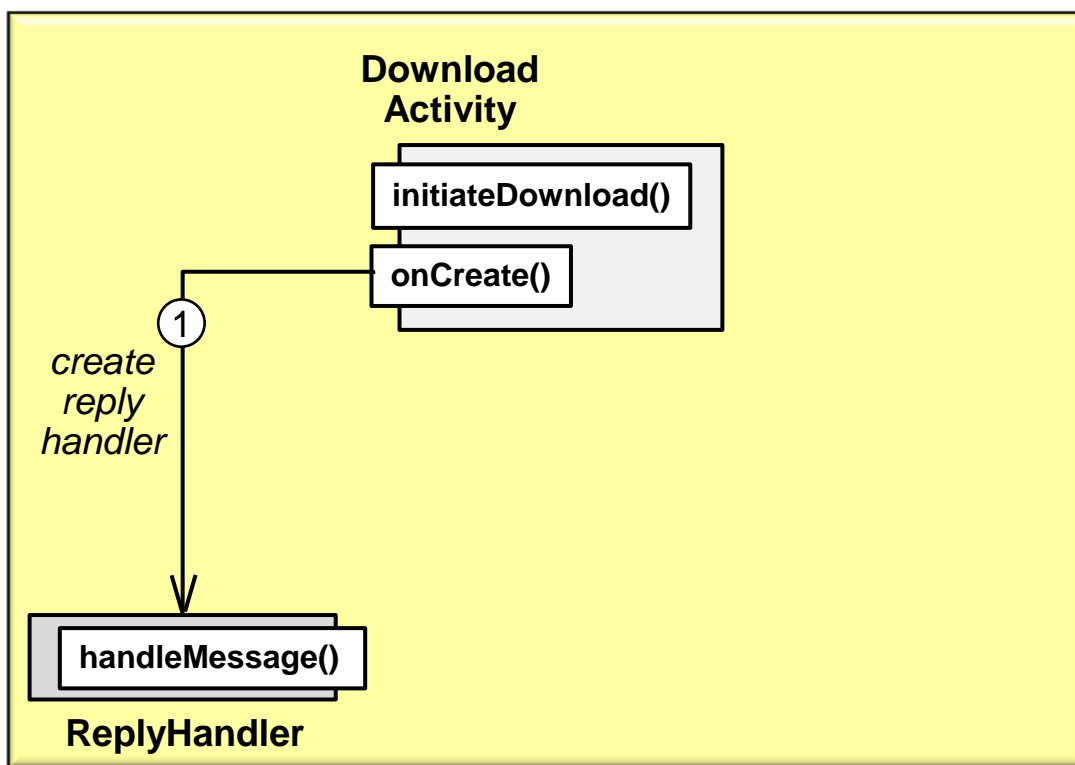
Overview of Messengers

- A Messenger provides a reference to a Handler that others can use to send messages to it
- An Activity can create a Messenger pointing to a Handler in one process & then pass that Messenger to another process
- The receiver then does several things
- You can use Messengers with both Bound & Started Services to implement the *Command Processor* pattern



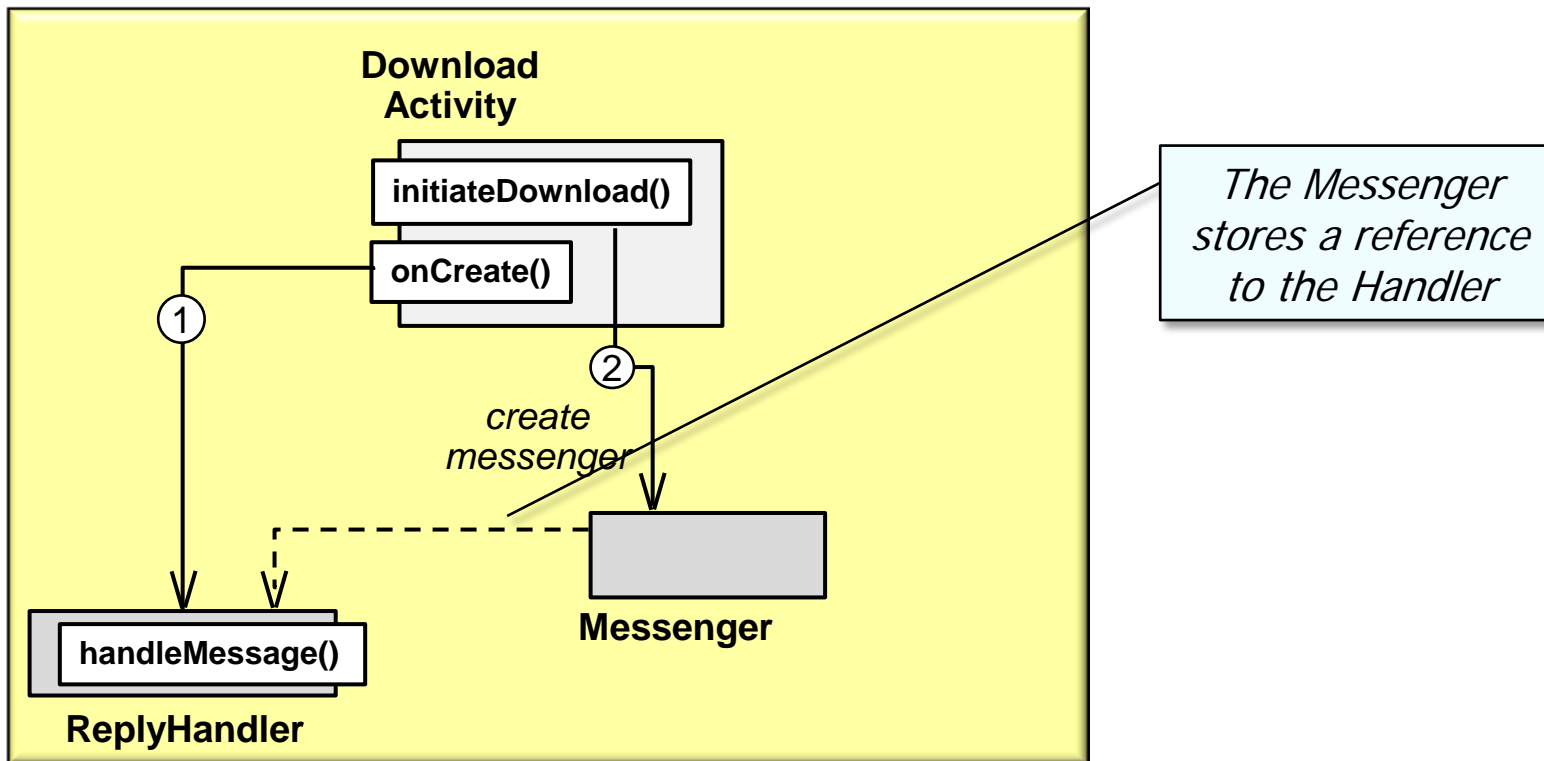
Using Messenger in Download App

- DownloadActivity passes Messenger as an “extra” to the Intent used to activate the DownloadService
- DownloadService uses the Messenger to reply back to the Activity



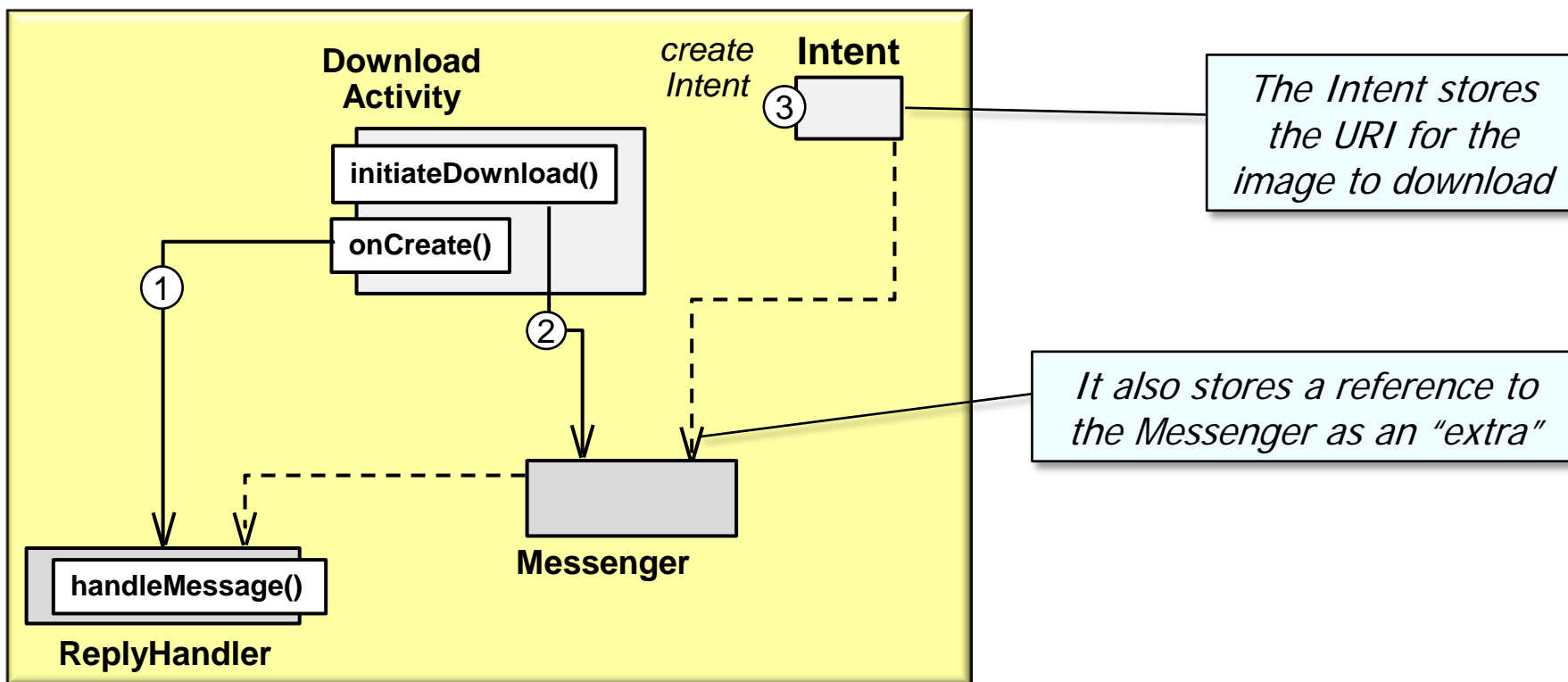
Using Messenger in Download App

- DownloadActivity passes Messenger as an “extra” to the Intent used to activate the DownloadService
- DownloadService uses the Messenger to reply back to the Activity



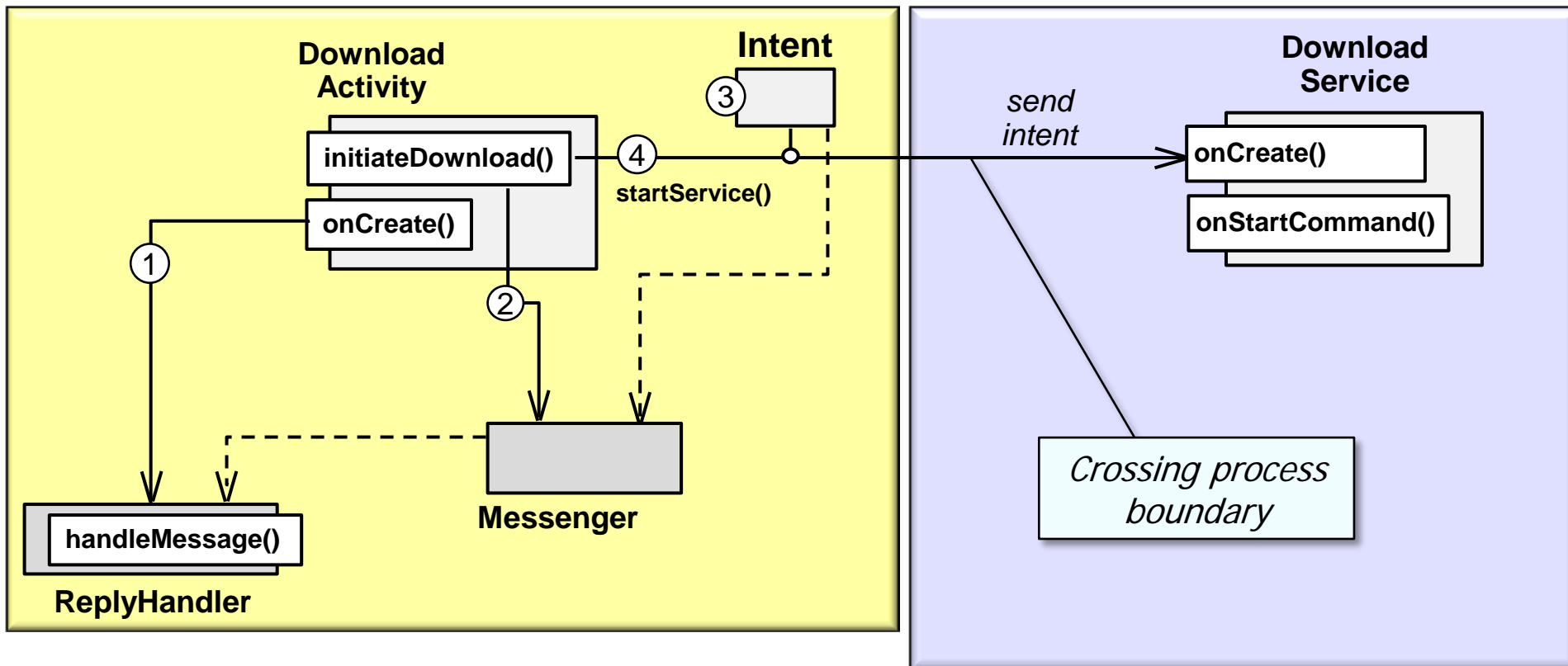
Using Messenger in Download App

- DownloadActivity passes Messenger as an “extra” to the Intent used to activate the DownloadService
- DownloadService uses the Messenger to reply back to the Activity



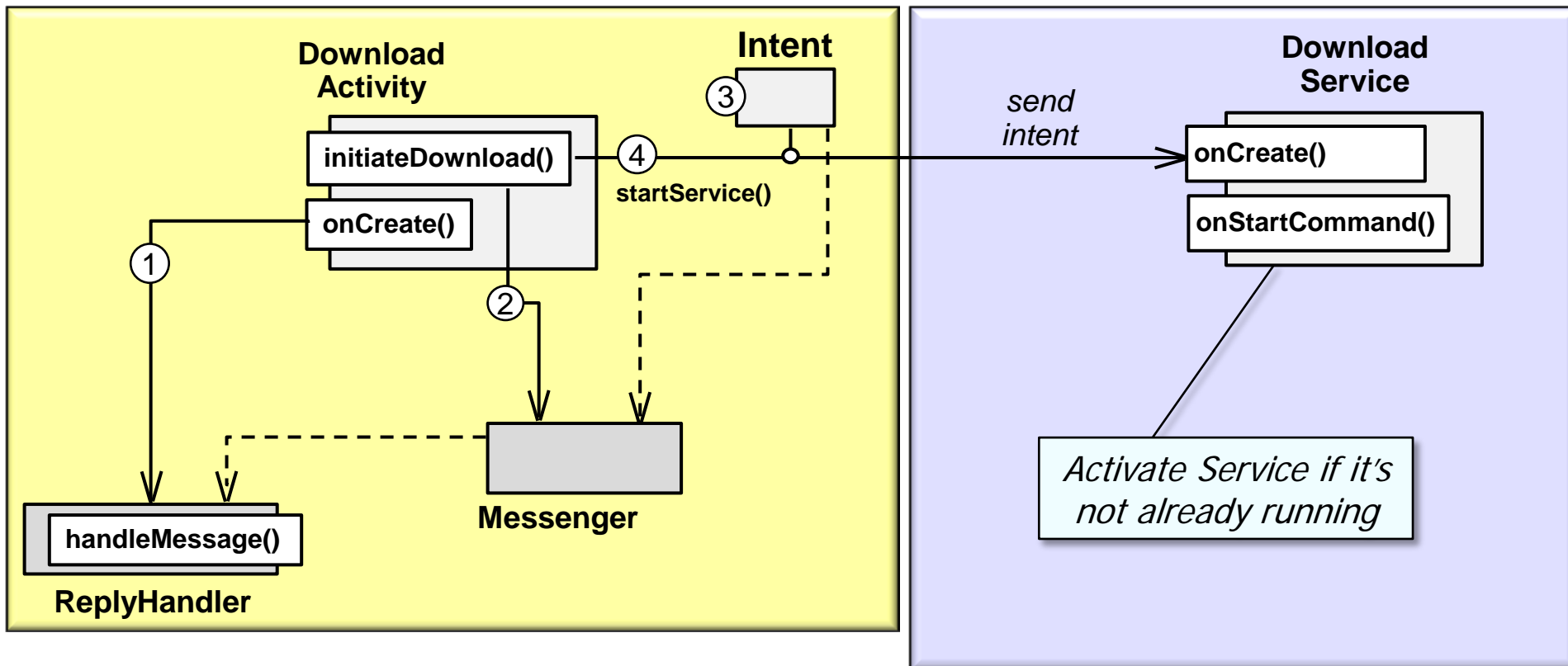
Using Messenger in Download App

- DownloadActivity passes Messenger as an “extra” to the Intent used to activate the DownloadService
- DownloadService uses the Messenger to reply back to the Activity



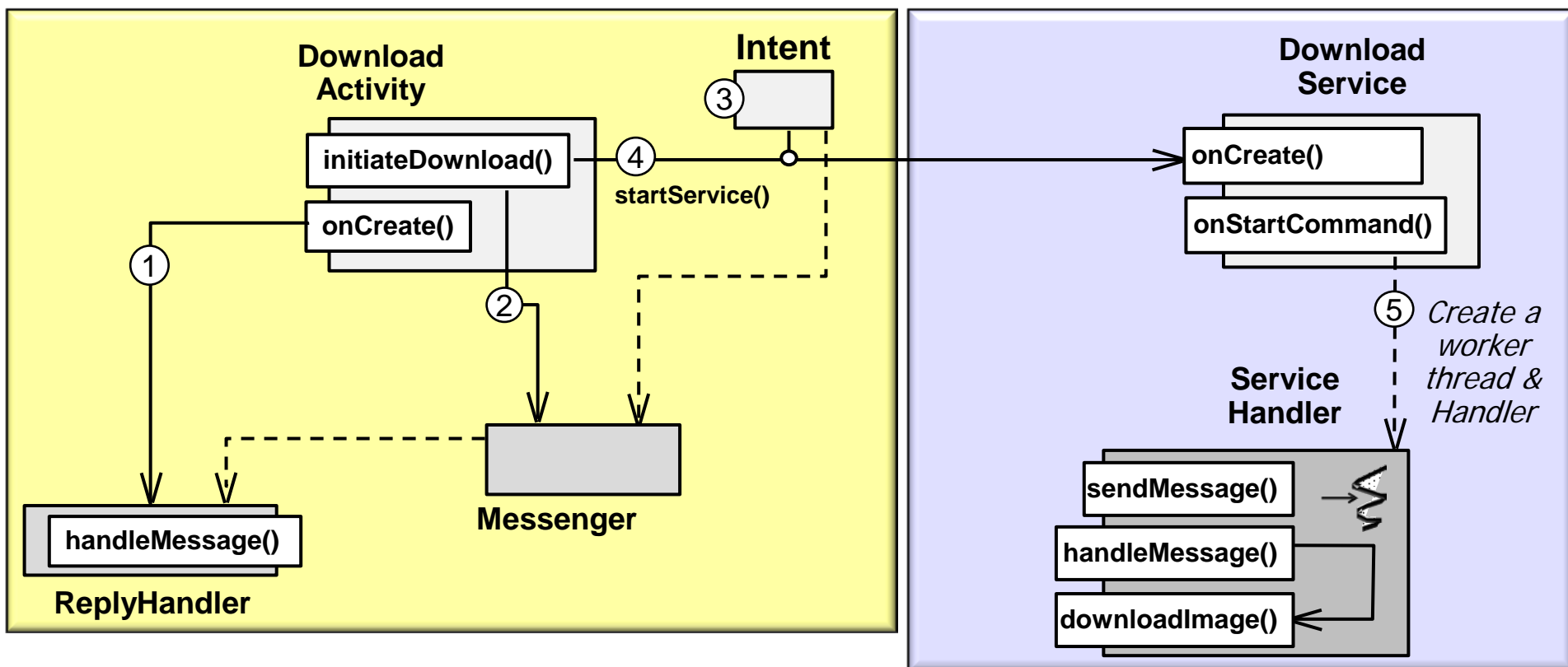
Using Messenger in Download App

- DownloadActivity passes Messenger as an “extra” to the Intent used to activate the DownloadService
- DownloadService uses the Messenger to reply back to the Activity



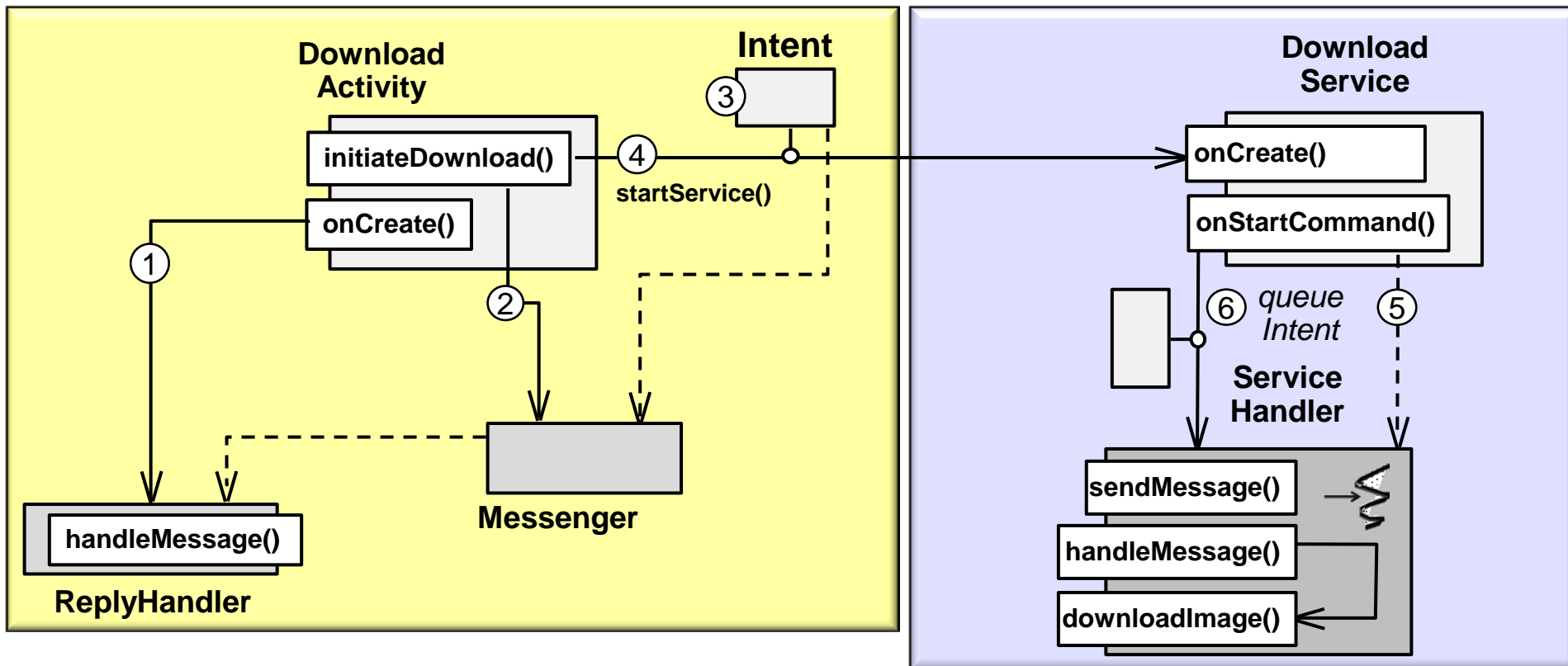
Using Messenger in Download App

- DownloadActivity passes Messenger as an “extra” to the Intent used to activate the DownloadService
- DownloadService uses the Messenger to reply back to the Activity



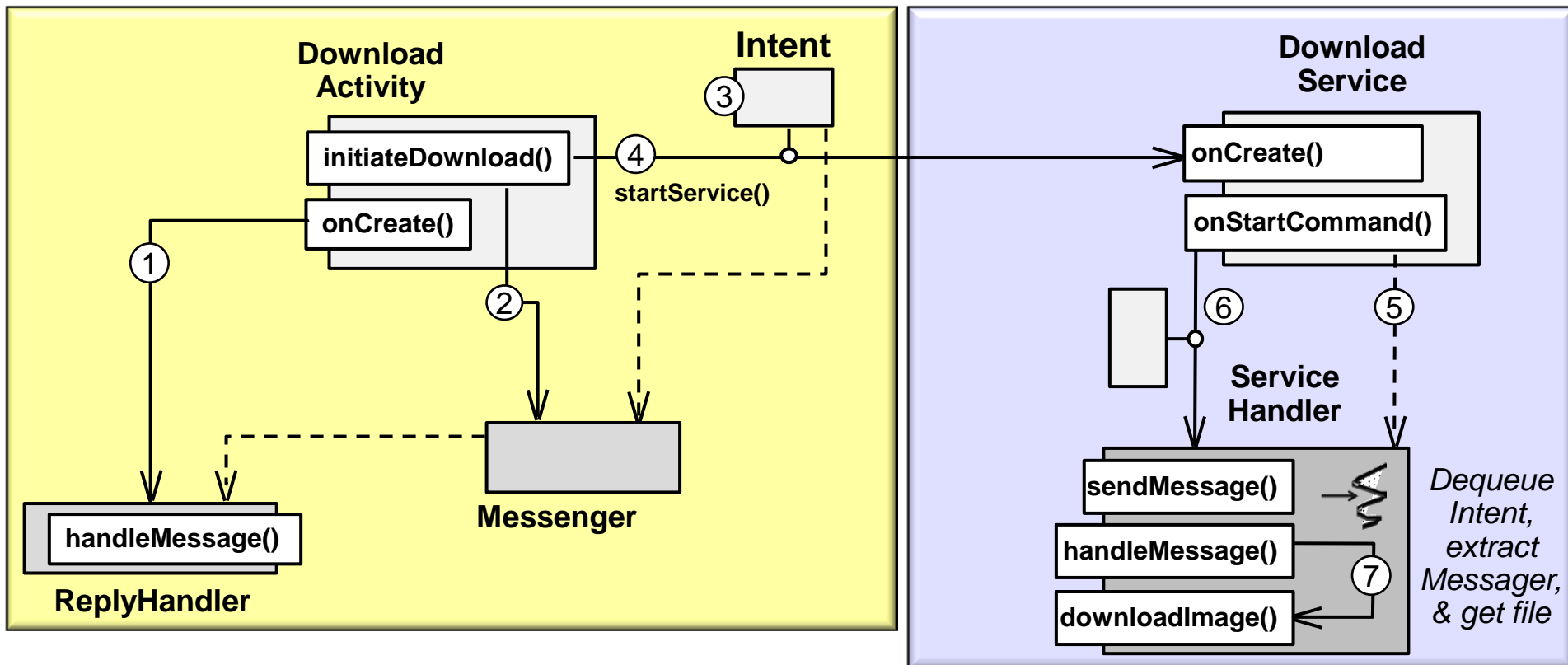
Using Messenger in Download App

- DownloadActivity passes Messenger as an “extra” to the Intent used to activate the DownloadService
- DownloadService uses the Messenger to reply back to the Activity



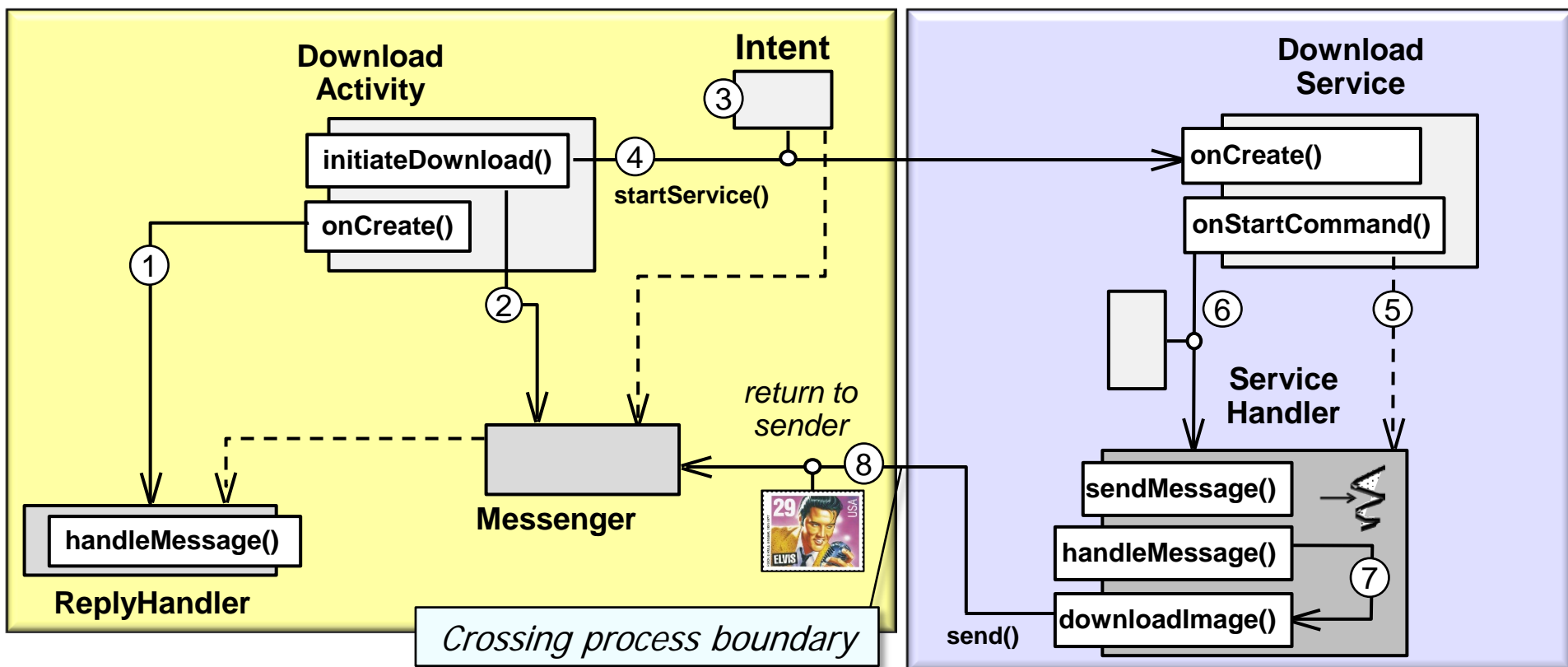
Using Messenger in Download App

- DownloadActivity passes Messenger as an “extra” to the Intent used to activate the DownloadService
- DownloadService uses the Messenger to reply back to the Activity



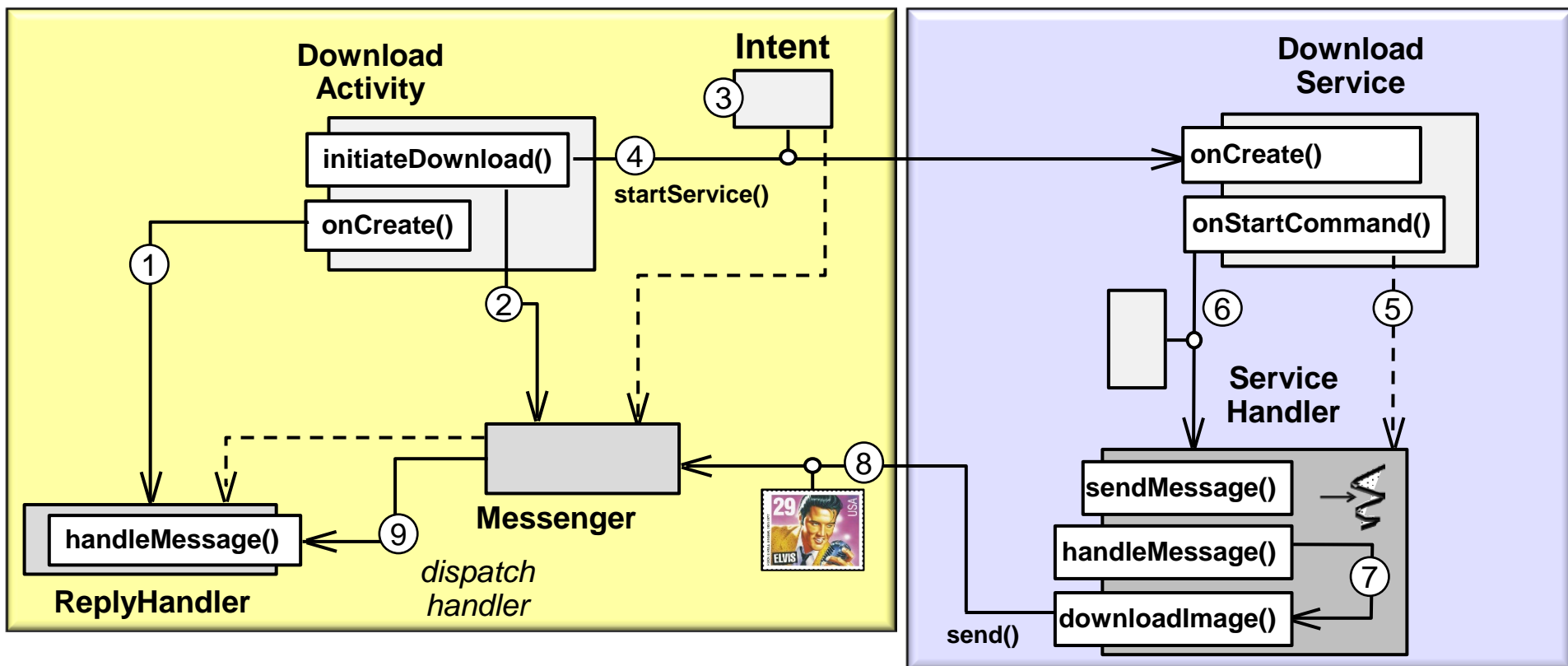
Using Messenger in Download App

- DownloadActivity passes Messenger as an “extra” to the Intent used to activate the DownloadService
- DownloadService uses the Messenger to reply back to the Activity



Using Messenger in Download App

- DownloadActivity passes Messenger as an “extra” to the Intent used to activate the DownloadService
- DownloadService uses the Messenger to reply back to the Activity



Programming a Messenger in Download Activity

- DownloadActivity passes a Messenger to the DownloadService

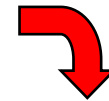
```
public class DownloadActivity extends Activity {  
    ...  
    Handler downloadHandler = new Handler() { /* ... */ }
```



Create a Handler to process reply from DownloadService

```
public void initiateDownload(View v) {  
    Intent intent = new Intent(DownloadActivity.this,  
                             DownloadService.class);  
    ...
```

Pass a Messenger as an "extra" in the Intent
used to start the DownloadService



```
intent.putExtra("MESSENGER",  
               new Messenger (downloadHandler));  
startService(intent);
```



```
}  
...
```



Start the service

Programming a Messenger in Download Service

- DownloadService replies to Activity via Messenger's send() method

```
public class DownloadService extends Service {  
    ...  
    private final class ServiceHandler extends Handler {  
        ...  
        public void downloadImage(Intent intent) {  
            // ...  Code to downloading image to pathname goes here  
  
            Message msg = Message.obtain();  
            msg.arg1 = result;  
            Bundle bundle = new Bundle();  
            bundle.putString("PATHNAME", pathname);  
            msg.setData(bundle);  
            Messenger messenger = (Messenger)  
                intent.getExtras().get("MESSENGER");  
            messenger.send(msg);   
        }  
        ...  
    }  
    ...  
}
```

Extract Messenger & return pathname to the client

Programming a Messenger in Download Activity

- DownloadActivity receives Message via its Handler in the UI Thread

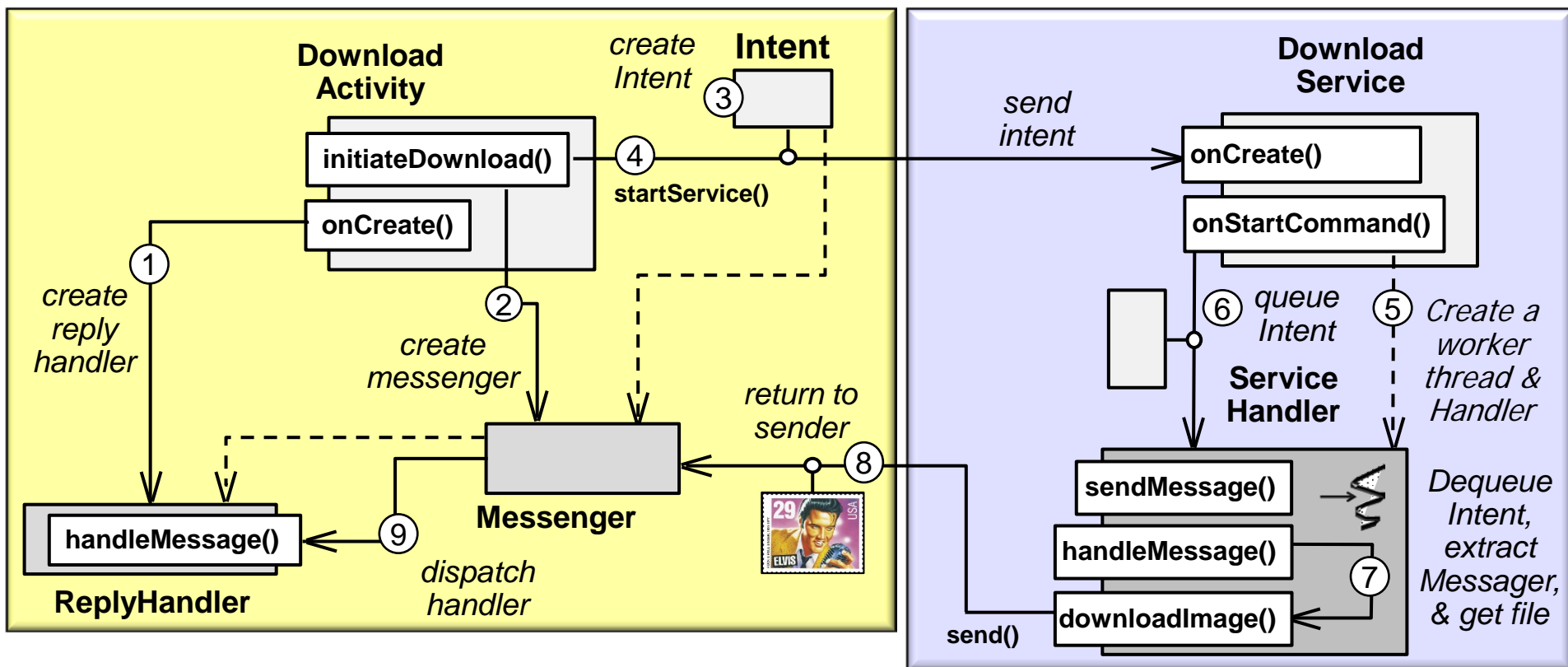
```
public class DownloadActivity extends Activity {  
    ...  
    Handler downloadHandler = new Handler() {  
        public void handleMessage(Message msg) {  
            Bundle data = msg.getData();  
            String pathname = data.getString ("PATHNAME");  
  
            if (msg.arg1 != RESULT_OK || path == null) {  
                Toast.makeText(DownloadActivity.this,"failed download",  
                    Toast.LENGTH_LONG).show();  
            }  
            displayImage(path);  
        }  
    };  
    ...
```

Get pathname
from Download
Service

Display the image

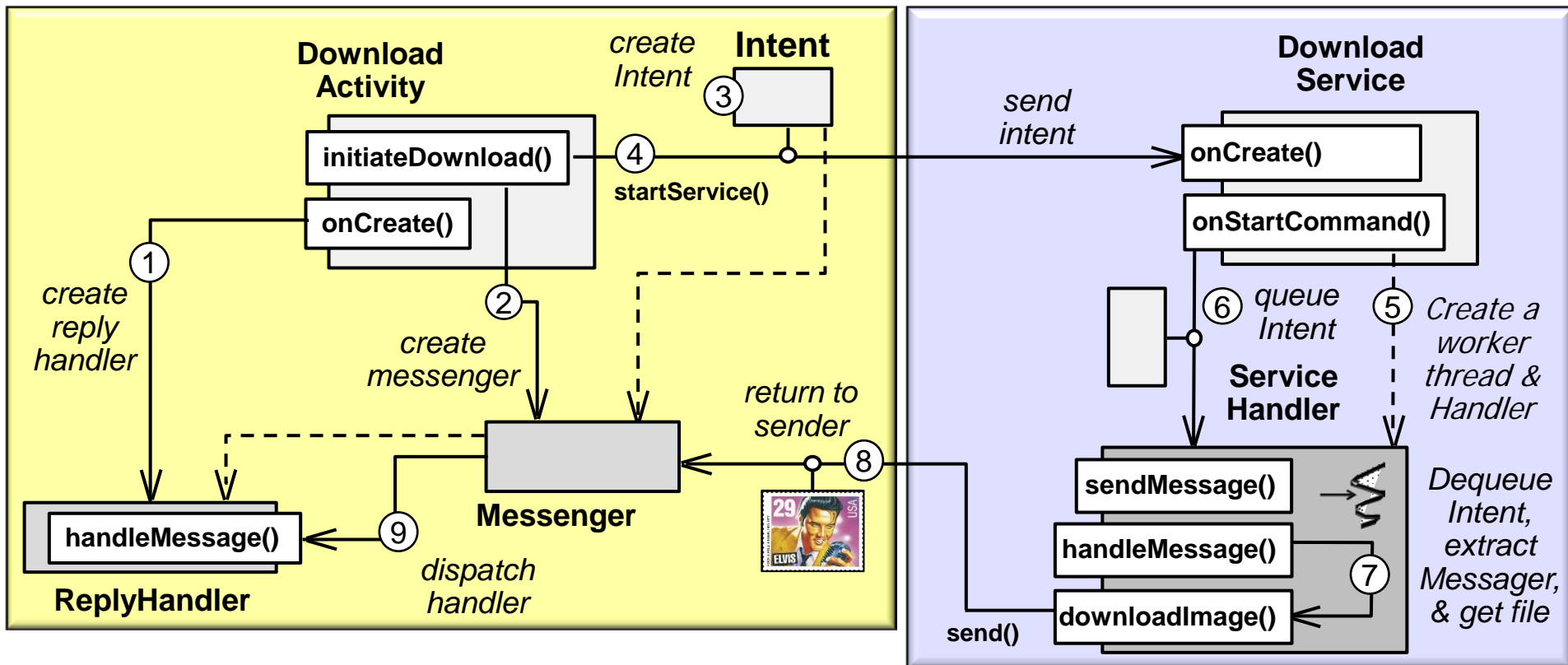
Summary

- Messengers provide a flexible framework for communicating between processes in Android



Summary

- Messengers provide a flexible framework for communicating between processes in Android
- Messengers make asynchrony easy, though non-trivial use-cases can be hard



Android Services & Local IPC: Communicate from Started Services to Activities via Broadcast Receivers

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

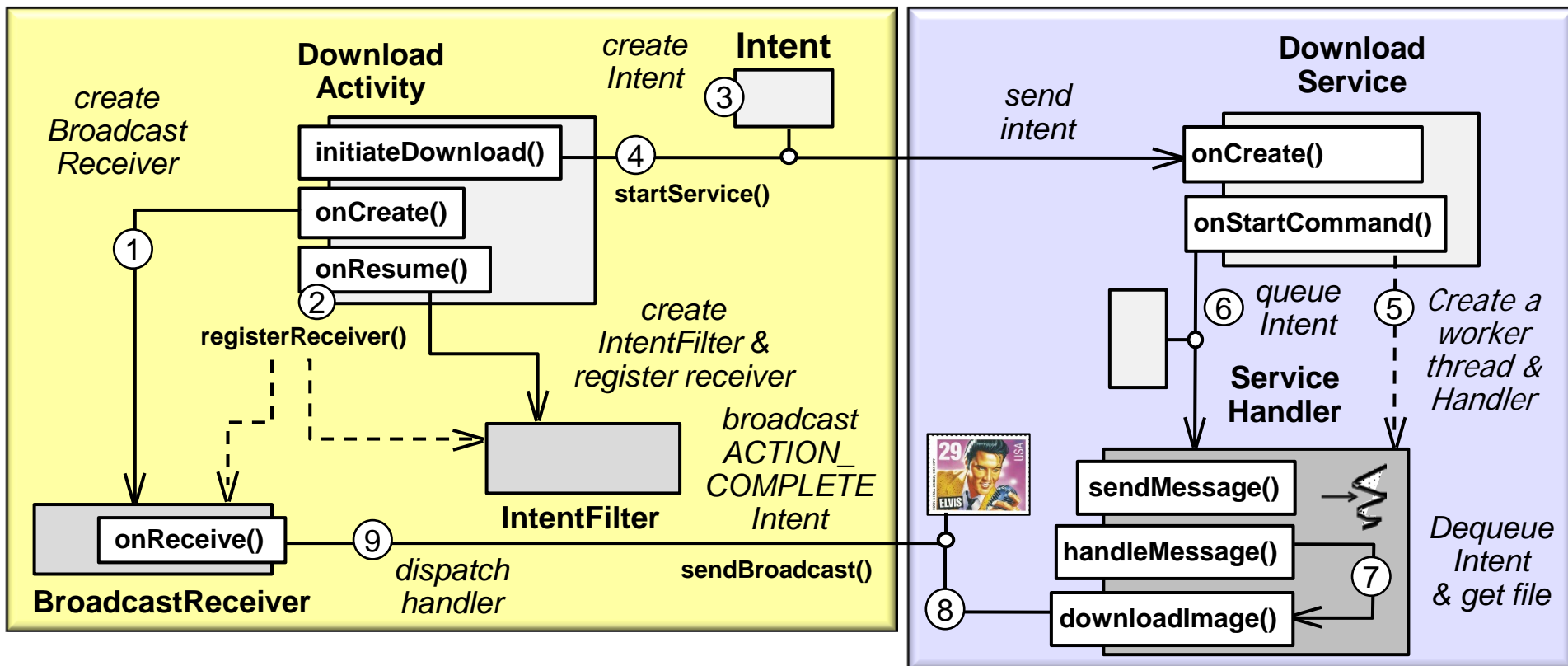
Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



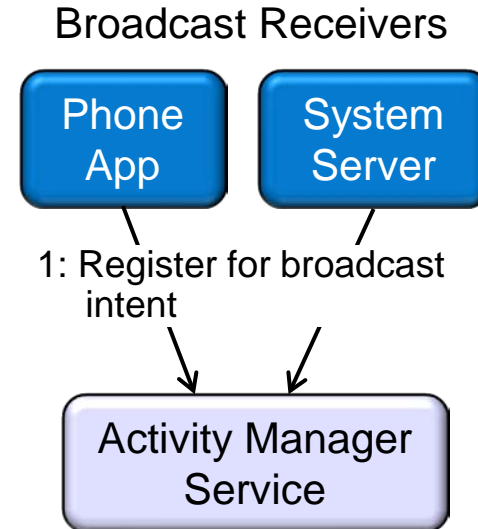
Learning Objectives in this Part of the Module

- Understand how to use Broadcast Receivers to communicate from Started Services back to their invoking Activities
- Provides IPC with (multiple) remote processes without using AIDL



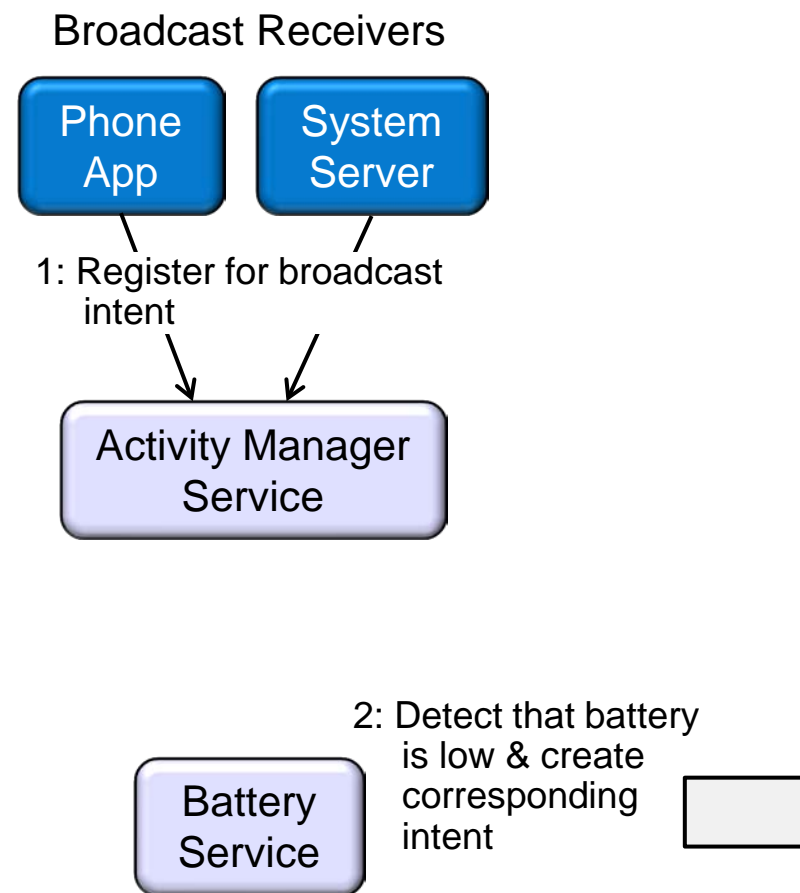
Overview of Broadcast Receivers

- BroadcastReceivers are components (*receivers*) that register for broadcast events & receive/react to the events



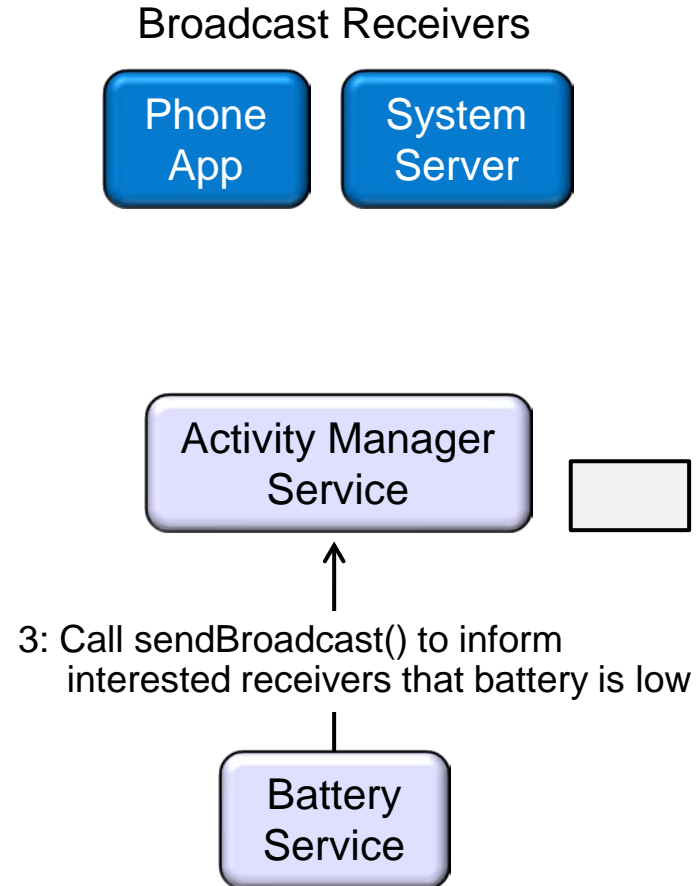
Overview of Broadcast Receivers

- BroadcastReceiver are components (*receivers*) that register for broadcast events & receive/react to the events
- Events implemented as Intents



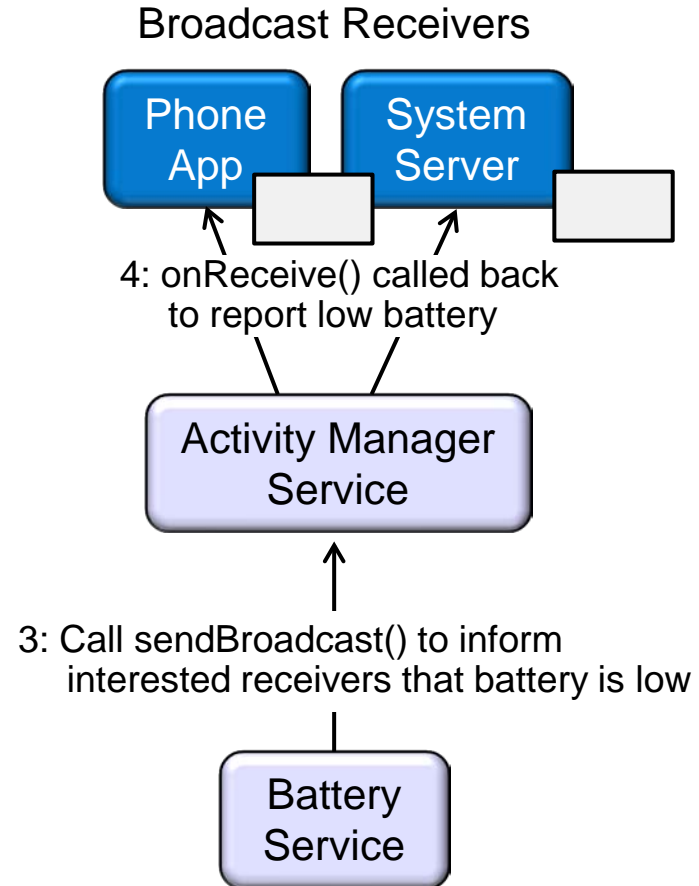
Overview of Broadcast Receivers

- BroadcastReceiver are components (*receivers*) that register for broadcast events & receive/react to the events
 - Events implemented as Intents
 - Events are broadcast system-wide



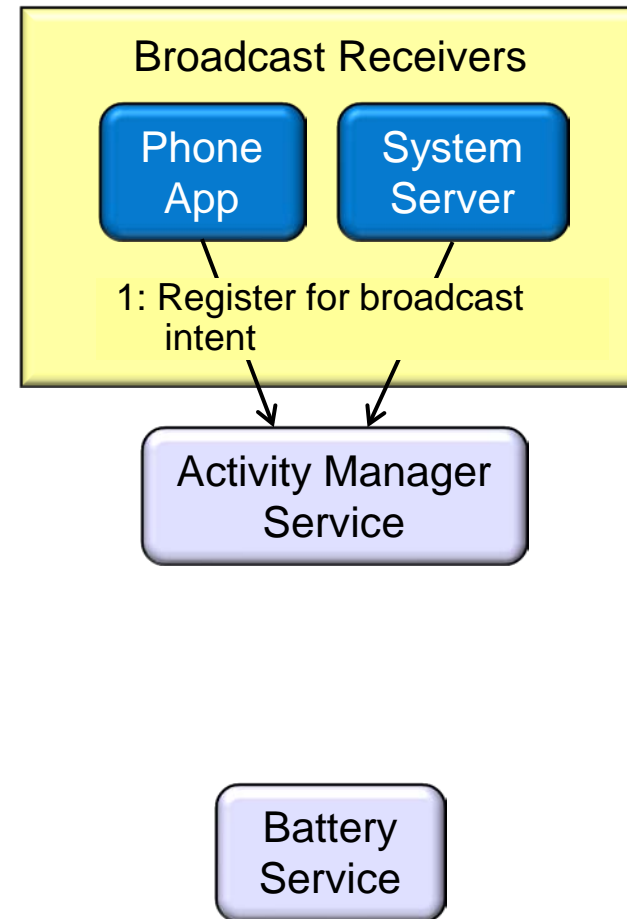
Overview of Broadcast Receivers

- BroadcastReceiver are components (*receivers*) that register for broadcast events & receive/react to the events
 - Events implemented as Intents
 - Events are broadcast system-wide
- When an event occurs the Intents are disseminated to all matching receivers via their `onReceive()` hook methods



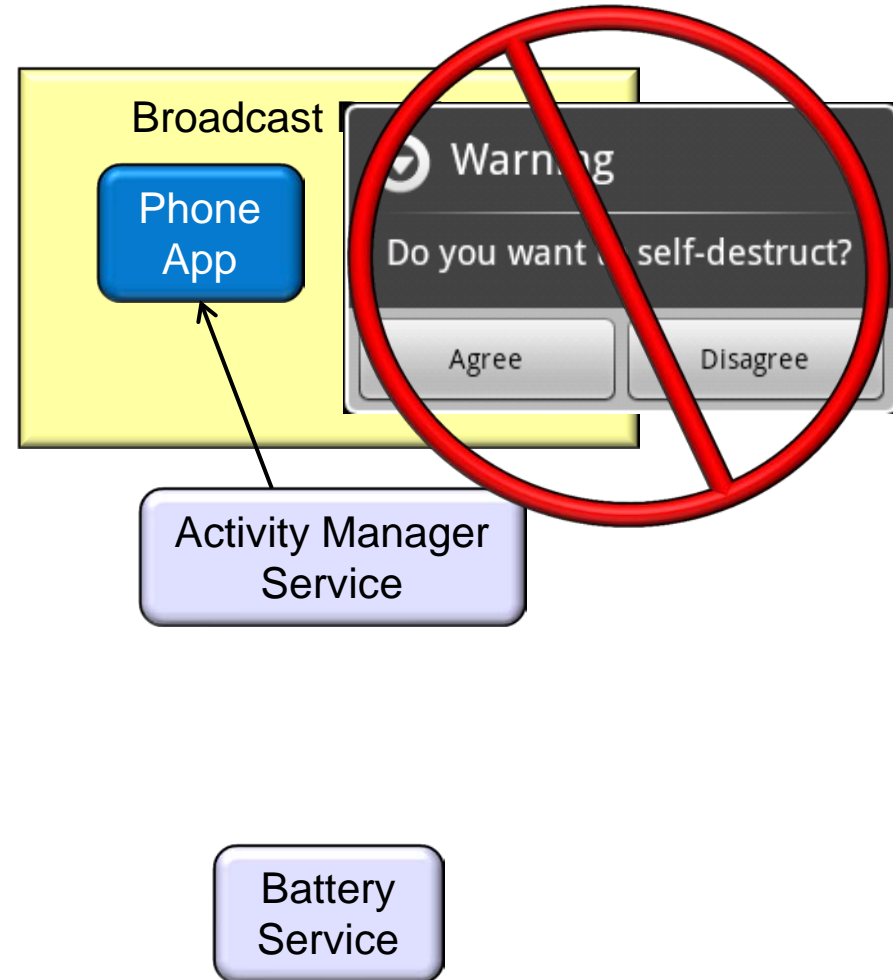
Overview of Broadcast Receivers

- BroadcastReceivers are components (*receivers*) that register for broadcast events & receive/react to the events
- Activities can create receivers that register for system or app events



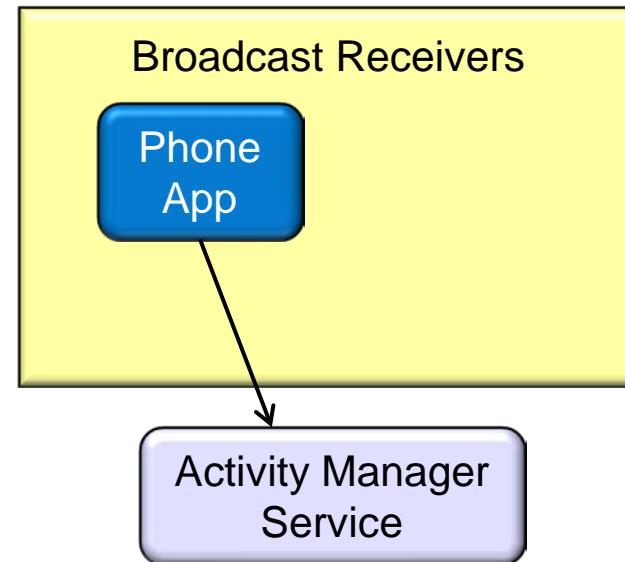
Overview of Broadcast Receivers

- BroadcastReceiver are components (*receivers*) that register for broadcast events & receive/react to the events
- Activities can create receivers that register for system or app events
- A receiver is restricted on what it can do when it handles an Intent
 - e.g., it may *not* show a dialog or bind to a service



Overview of Broadcast Receivers

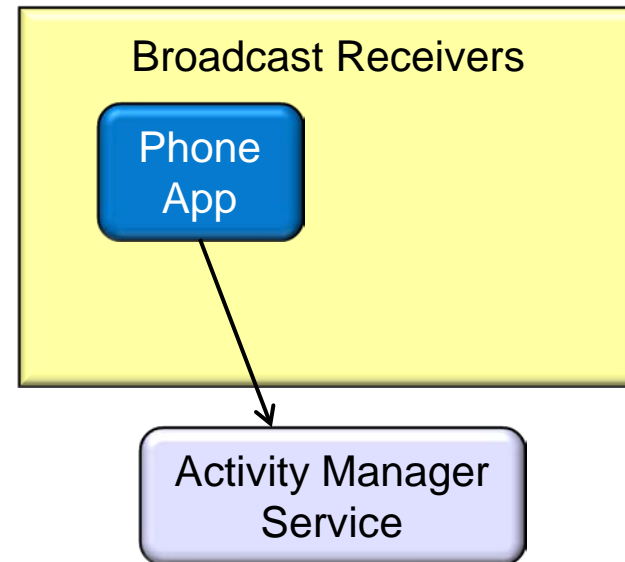
- BroadcastReceivers are components (*receivers*) that register for broadcast events & receive/react to the events
- Activities can create receivers that register for system or app events
- A receiver is restricted on what it can do when it handles an Intent
- Two ways to register a receiver:
 - Statically publish it via the `<receiver>` tag in the `AndroidManifest.xml` file



```
<receiver android:name="PhoneApp$NotificationBroadcastReceiver"
          exported="false">
  <intent-filter>
    <action android:name="
      com.android.phone.ACTION_HANG_UP_ONGOING_CALL" />
    <action android:name="
      com.android.phone.ACTION_SEND_SMS_FROM_NOTIFICATION"/>
  </intent-filter>
</receiver>
```


Overview of Broadcast Receivers

- BroadcastReceivers are components (*receivers*) that register for broadcast events & receive/react to the events
- Activities can create receivers that register for system or app events
- A receiver is restricted on what it can do when it handles an Intent
- Two ways to register a receiver:
 - Statically publish it via the <receiver> tag in the AndroidManifest.xml file

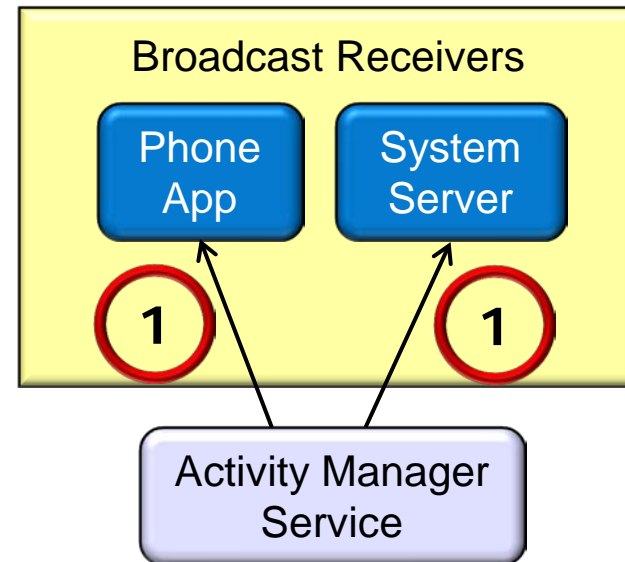


- Dynamically register it with Context.registerReceiver()

```
final BroadcastReceiver mReceiver =  
    new PhoneAppBroadcastReceiver();  
...  
IntentFilter intentFilter =  
    new IntentFilter(Intent.ACTION_AIRPLANE_MODE_CHANGED);  
...  
registerReceiver(mReceiver, intentFilter);
```

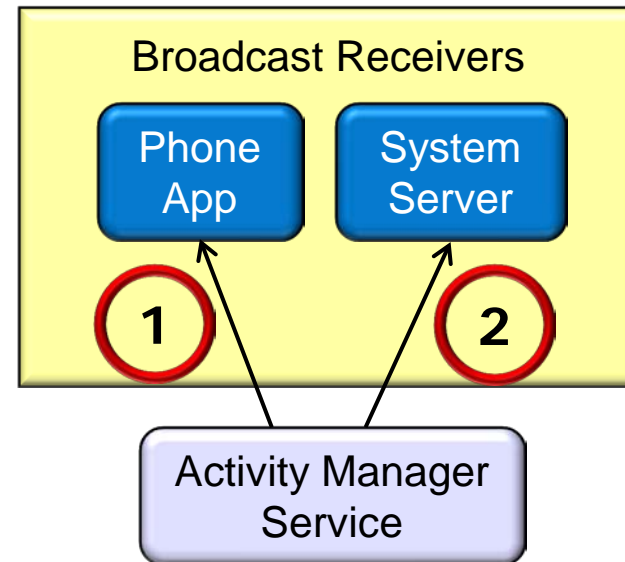
Overview of Broadcast Receivers

- BroadcastReceivers are components (*receivers*) that register for broadcast events & receive/react to the events
- Activities can create receivers that register for system or app events
- A receiver is restricted on what it can do when it handles an Intent
- Two ways to register a receiver
- Android supports several broadcast mechanisms
 - *Normal* – Sent with `Context.sendBroadcast()`, which is completely asynchronous



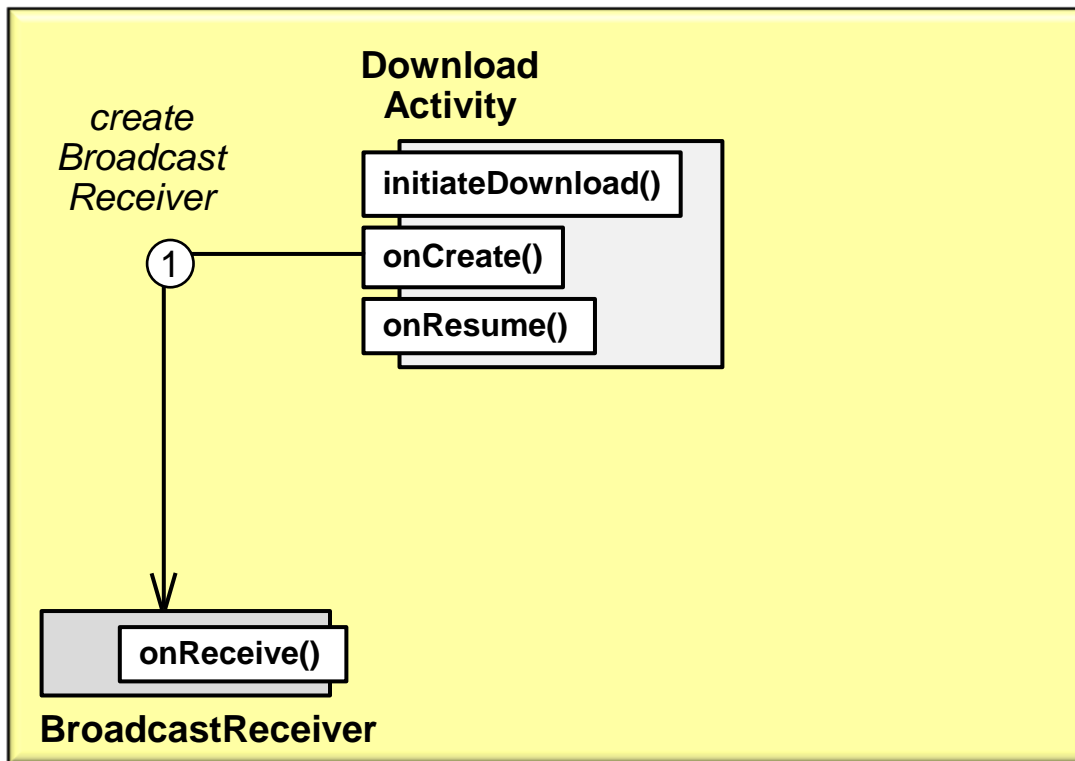
Overview of Broadcast Receivers

- BroadcastReceivers are components (*receivers*) that register for broadcast events & receive/react to the events
- Activities can create receivers that register for system or app events
- A receiver is restricted on what it can do when it handles an Intent
- Two ways to register a receiver
- Android supports several broadcast mechanisms
 - *Normal* – Sent with `Context.sendBroadcast()`, which is completely asynchronous
 - *Ordered* – Sent with `Context.sendOrderedBroadcast()`, which is delivered to one receiver at a time



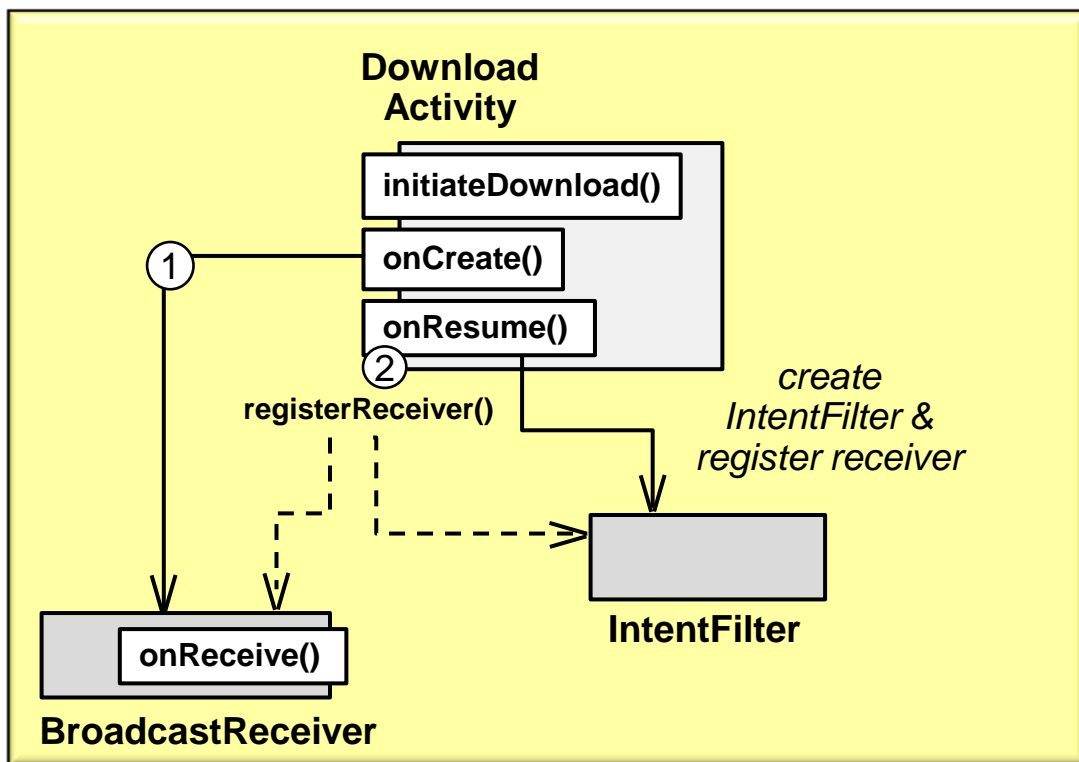
Using a Broadcast Receiver in the Download App

- DownloadActivity creates & registers a BroadcastReceiver with an IntentFilter configured with the ACTION_COMPLETE action
- DownloadService broadcasts an ACTION_COMPLETE back to the Activity



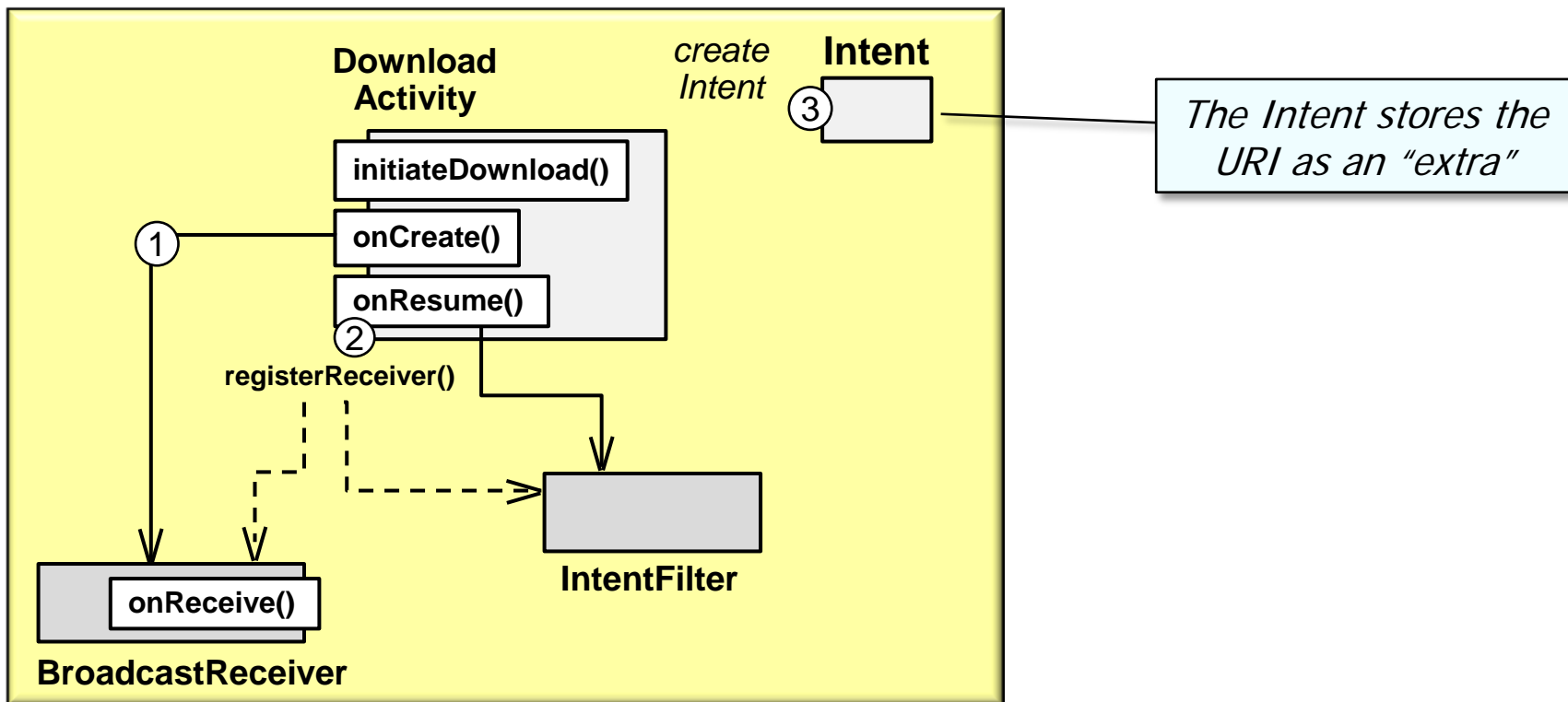
Using a Broadcast Receiver in the Download App

- DownloadActivity creates & registers a BroadcastReceiver with an IntentFilter configured with the ACTION_COMPLETE action
- DownloadService broadcasts an ACTION_COMPLETE back to the Activity



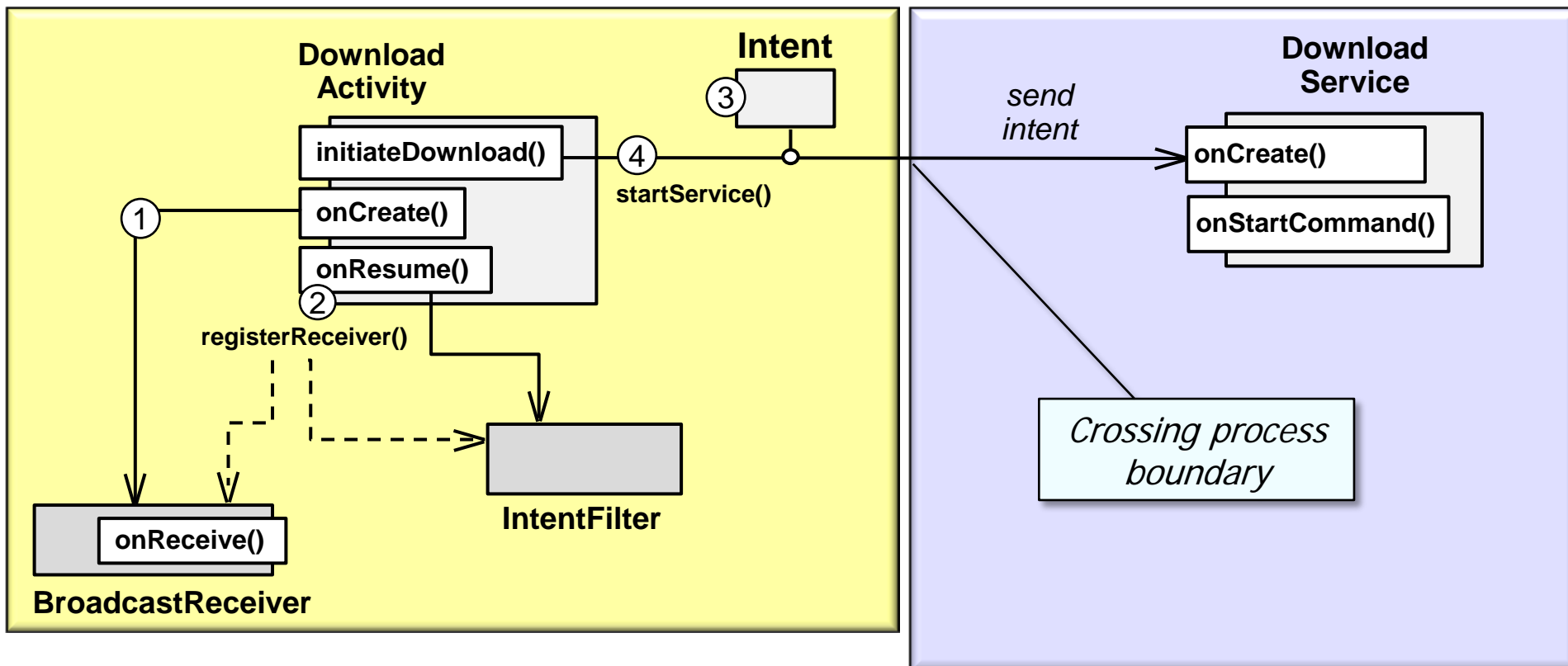
Using a Broadcast Receiver in the Download App

- DownloadActivity creates & registers a BroadcastReceiver with an IntentFilter configured with the ACTION_COMPLETE action
- DownloadService broadcasts an ACTION_COMPLETE back to the Activity



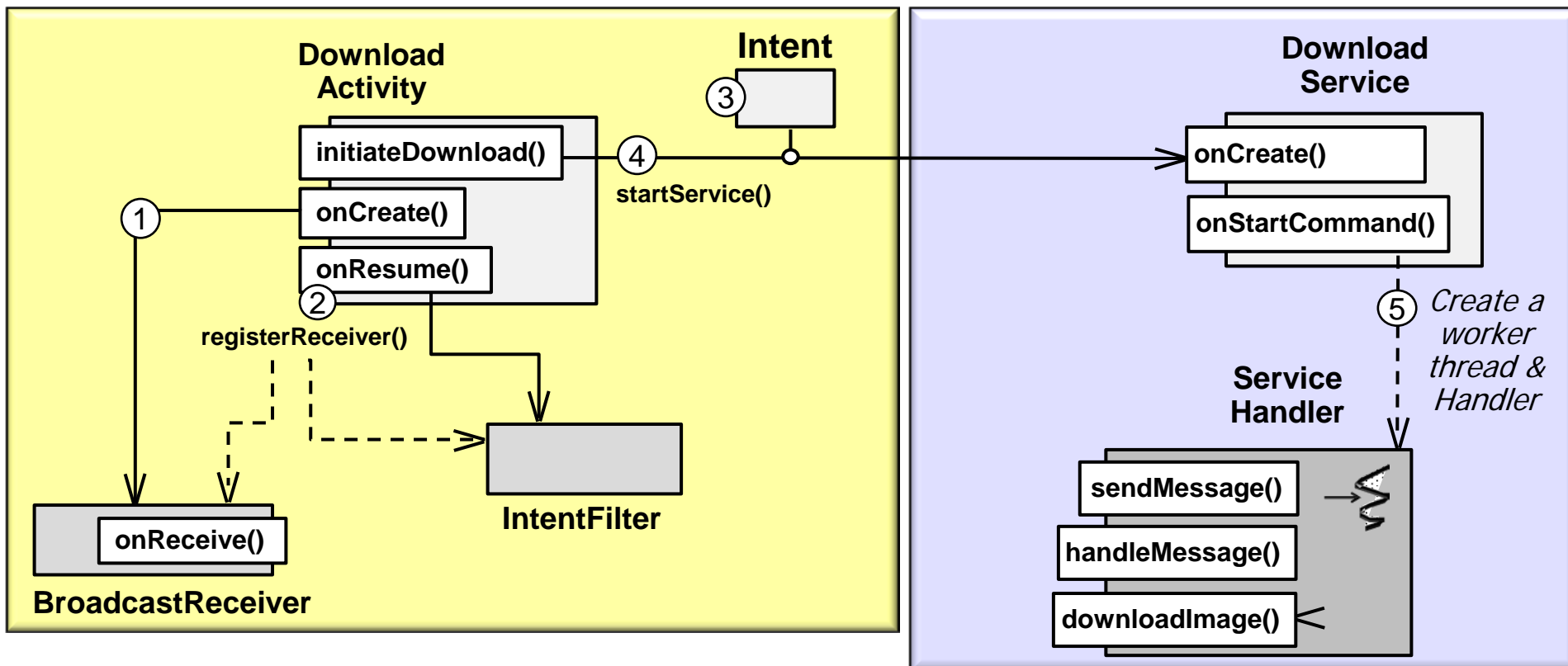
Using a Broadcast Receiver in the Download App

- DownloadActivity creates & registers a BroadcastReceiver with an IntentFilter configured with the ACTION_COMPLETE action
- DownloadService broadcasts an ACTION_COMPLETE back to the Activity



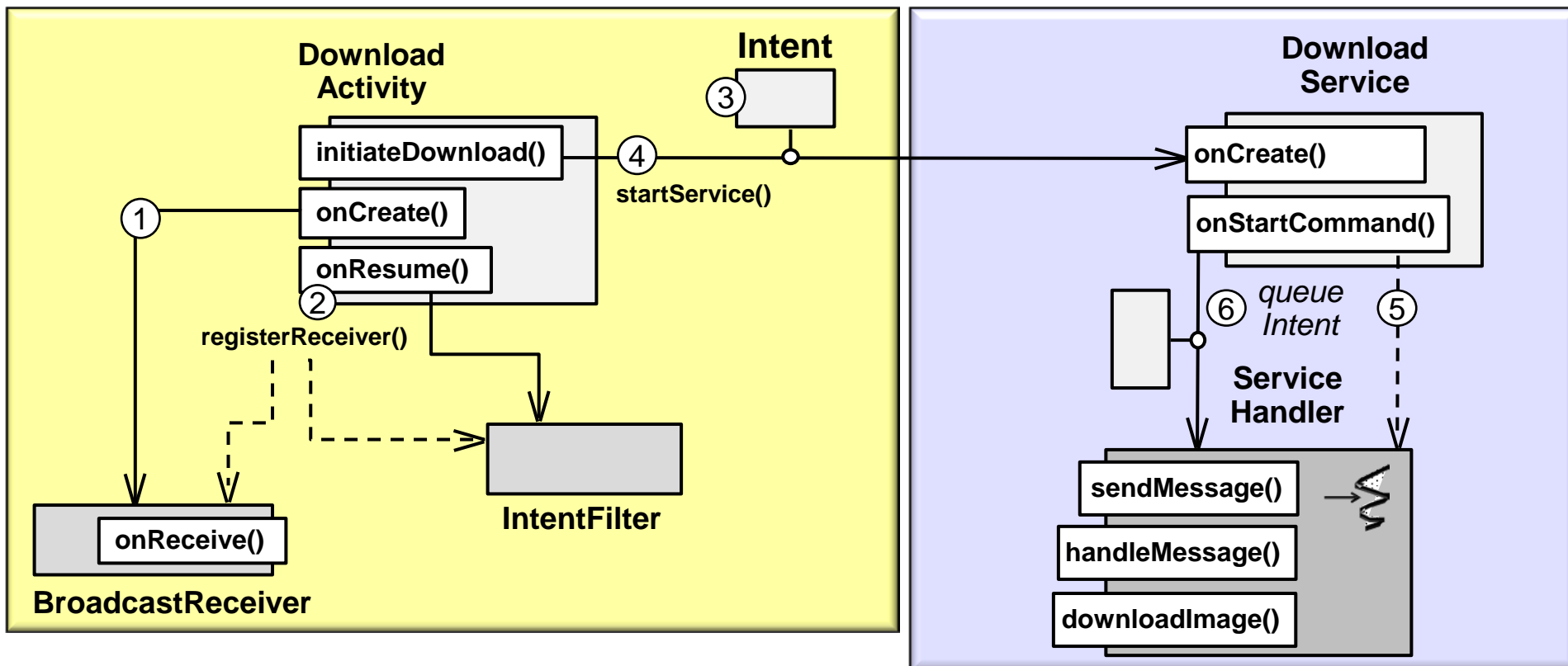
Using a Broadcast Receiver in the Download App

- DownloadActivity creates & registers a BroadcastReceiver with an IntentFilter configured with the ACTION_COMPLETE action
- DownloadService broadcasts an ACTION_COMPLETE back to the Activity



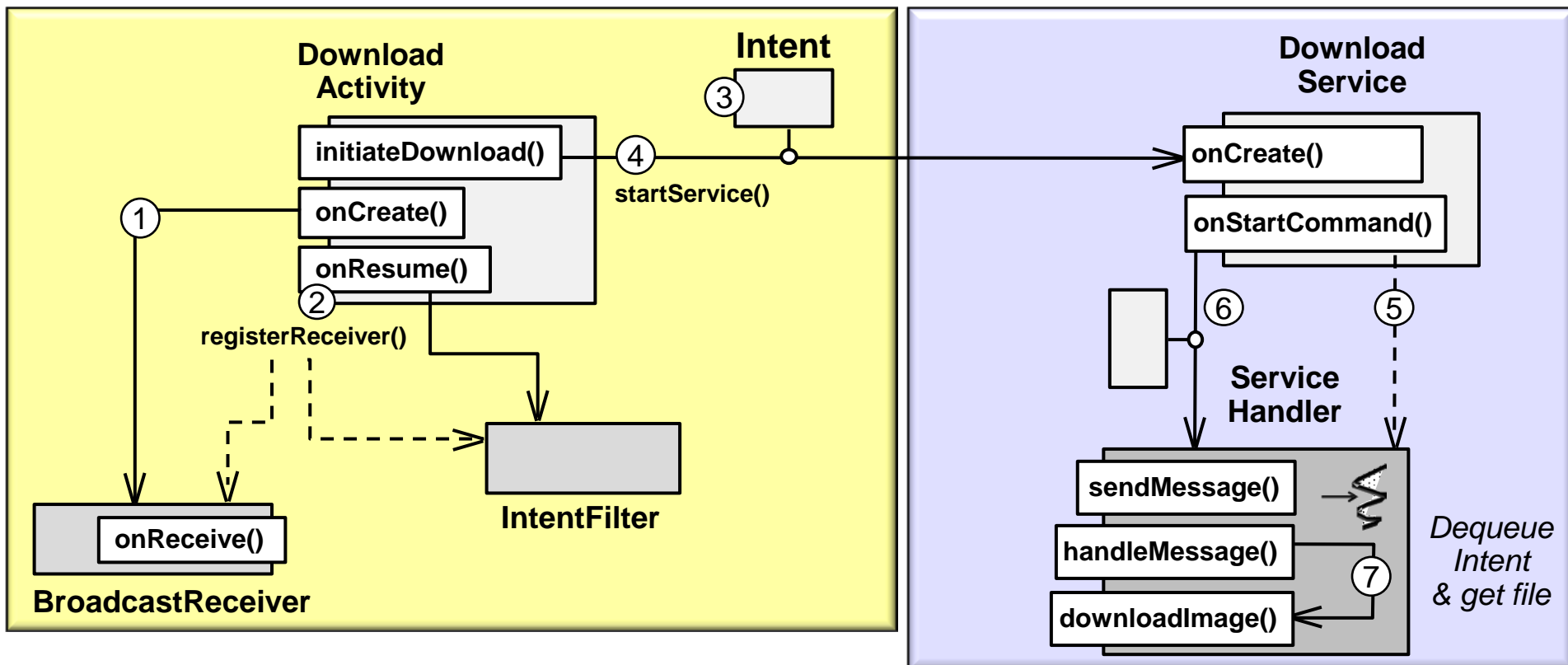
Using a Broadcast Receiver in the Download App

- DownloadActivity creates & registers a BroadcastReceiver with an IntentFilter configured with the ACTION_COMPLETE action
- DownloadService broadcasts an ACTION_COMPLETE back to the Activity



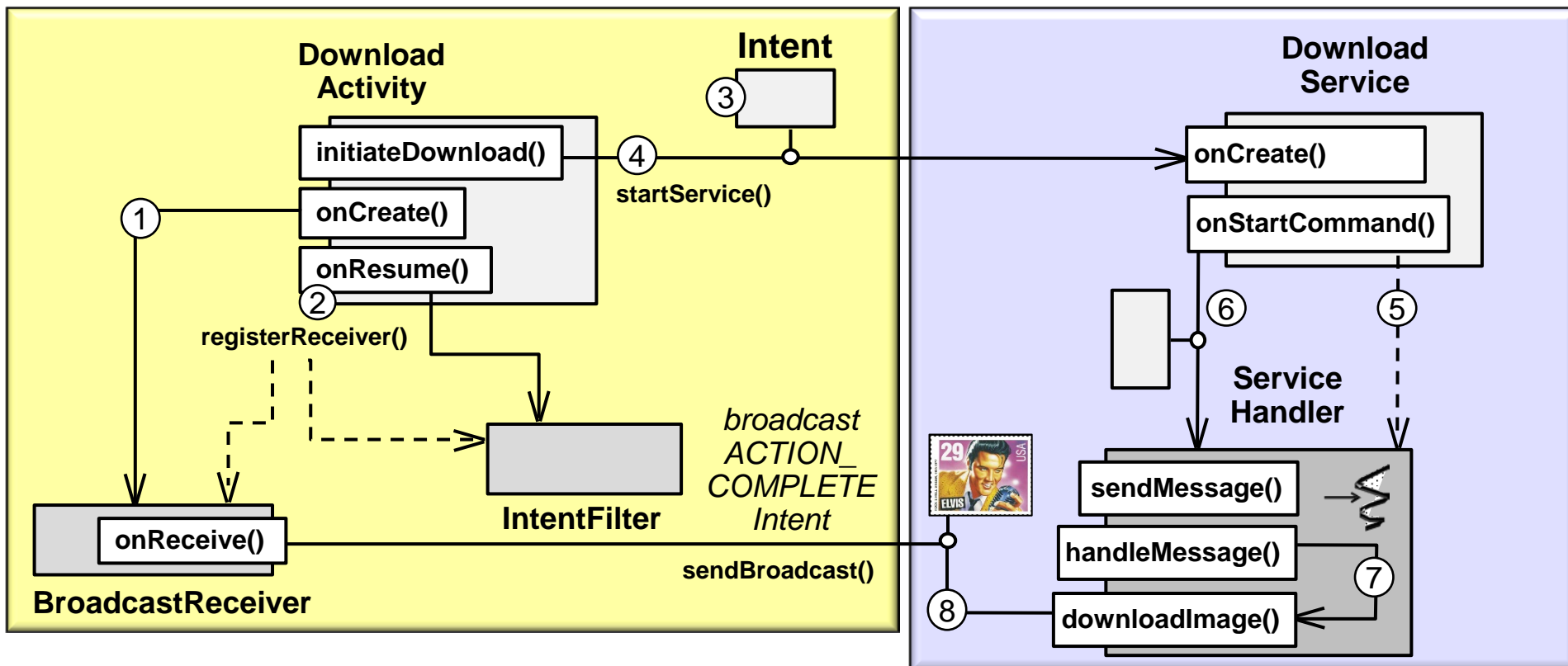
Using a Broadcast Receiver in the Download App

- DownloadActivity creates & registers a BroadcastReceiver with an IntentFilter configured with the ACTION_COMPLETE action
- DownloadService broadcasts an ACTION_COMPLETE back to the Activity



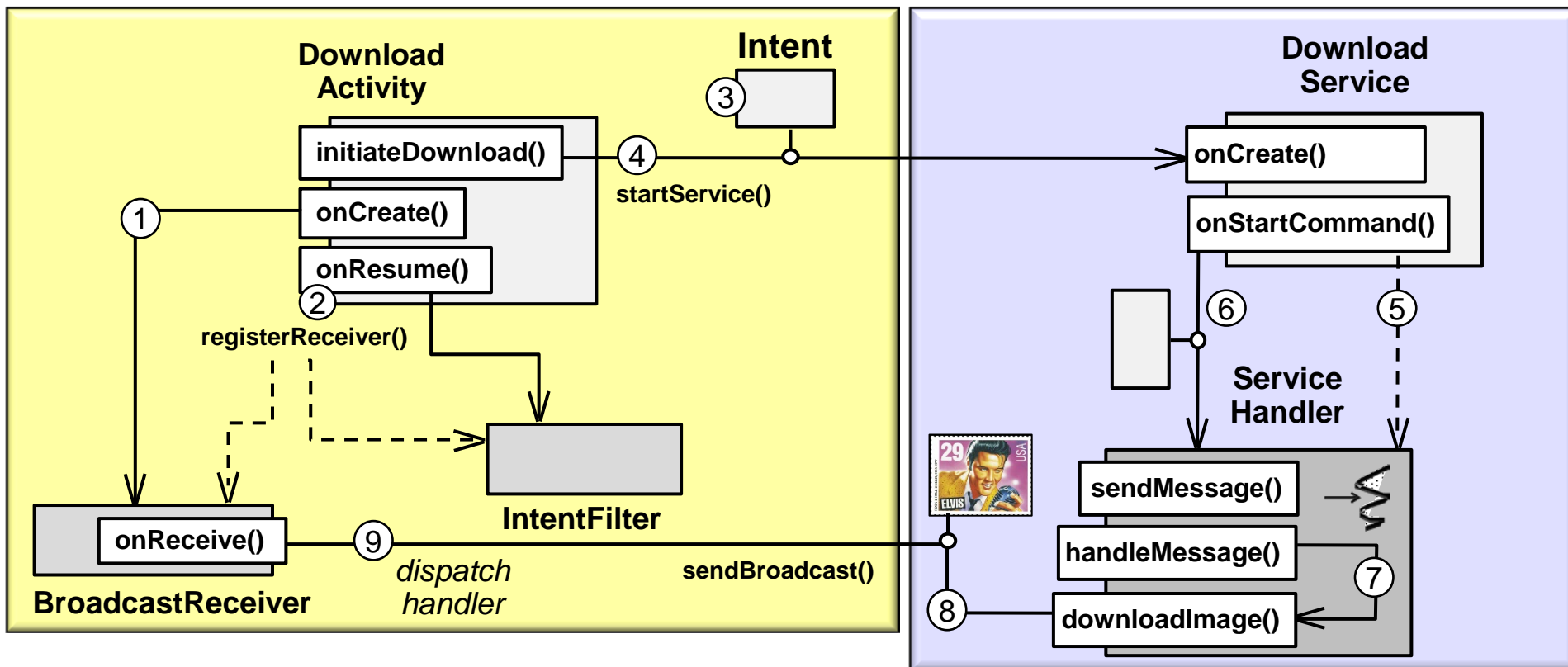
Using a Broadcast Receiver in the Download App

- DownloadActivity creates & registers a BroadcastReceiver with an IntentFilter configured with the ACTION_COMPLETE action
- DownloadService broadcasts an ACTION_COMPLETE back to the Activity






Using a Broadcast Receiver in the Download App

- DownloadActivity creates & registers a BroadcastReceiver with an IntentFilter configured with the ACTION_COMPLETE action
- DownloadService broadcasts an ACTION_COMPLETE back to the Activity



Programming a Broadcast Receiver in Activity

- DownloadActivity contains a BroadcastReceiver instance with hook method

```
public class DownloadActivity extends Activity {  
    private BroadcastReceiver onEvent = new BroadcastReceiver() {  
        public void onReceive(Context context, Intent intent) {  
             Receive Intent sent by sendBroadcast()  
  
            String path = intent.getStringExtra(RESULT_PATH);  
             Extract the path using "extra" within the Intent  
  
            if (path == null) {  
                Toast.makeText(DownloadActivity.this,  
                    "Download failed.",  
                    Toast.LENGTH_LONG).show();  
            }  
            displayImage(path);  
             Display the image  
        }  
    };  
    ...  
}
```

Programming a Broadcast Receiver in Activity

- DownloadActivity's lifecycle methods register & unregister the receiver

```
public class DownloadActivity extends Activity {  
    ...  
    public void onResume() {  
        super.onResume();  
        IntentFilter filter =  
            new IntentFilter(ACTION_COMPLETE);  
        registerReceiver(onEvent, filter);  
    }
```



Register BroadcastReceiver when Activity resumes

```
    public void onPause() {  
        super.onPause();  
        unregisterReceiver(onEvent);  
    }  
    ...
```





Unregister BroadcastReceiver before Activity pauses

Programming a Broadcast Receiver in Activity

- DownloadActivity passes the package name to the DownloadService



```
public class DownloadActivity extends Activity {  
    ...  
  
    public void initiateDownload(View v) {  
        Intent intent = new Intent(DownloadActivity.this,  
                                   DownloadService.class);  
  
        ...  
        Pass a package name as an "extra" in the  
        Intent used to start the DownloadService  
  
        intent.putExtra(PACKAGE_NAME, getPackageName());  
        startService(intent);  
    }  
    ...  
}
```



Start the service

Programming a Broadcast Receiver in Service

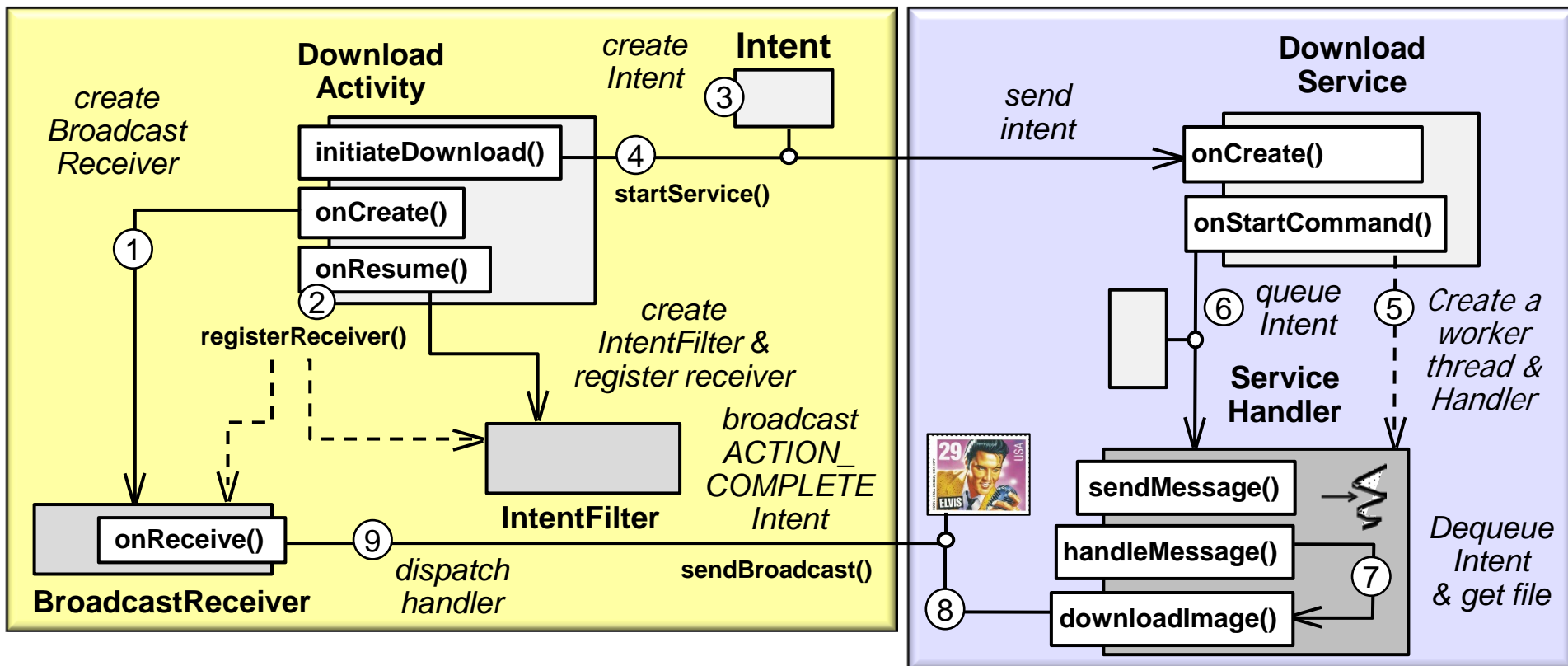
- DownloadService replies to DownloadActivity via `sendBroadcast()`

```
public class DownloadService extends Service {  
    ...  
    private final class ServiceHandler extends Handler {  
        ...  
        public void downloadImage(Intent intent) {  
            // ...  Code to downloading image to pathname goes here  
  
            Intent replyIntent = new Intent(ACTION_COMPLETE);  
            replyIntent.putExtra(RESULT_PATH, pathname);  
            String packageName = intent.getStringExtra(PACKAGE_NAME);  
            intent.setPackage(packageName);  
             Restrict the target of the broadcast  
            sendBroadcast(replyIntent);  
        }  
    }  
    ...  
}
```

 Broadcast pathname to Activity

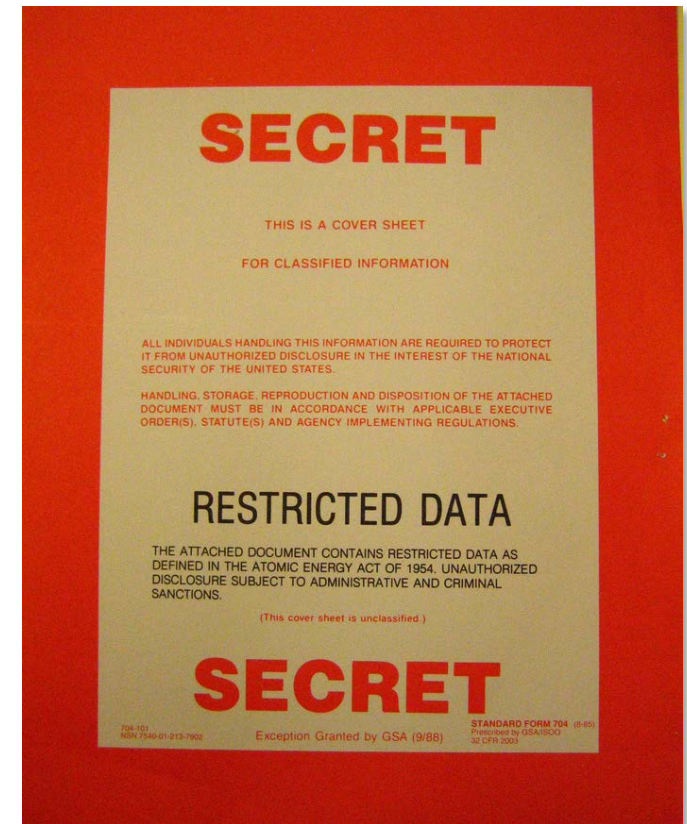
Summary

- Broadcast Receivers provide a scalable framework for communicating between (potentially multiple) processes in Android
- Broadcast Receivers are generally used for more interesting use-cases...



Summary

- Broadcast Receivers provide a scalable framework for communicating between (potentially multiple) processes in Android
- However, there are subtle issues with security



Summary

- Broadcast Receivers provide a scalable framework for communicating between (potentially multiple) processes in Android
- However, there are subtle issues with security
 - The Intent namespace is global
 - This may cause subtle conflicts



Summary

- Broadcast Receivers provide a scalable framework for communicating between (potentially multiple) processes in Android
- However, there are subtle issues with security
 - The Intent namespace is global
 - registerReceiver() allows any app to send broadcasts to that registered receiver
 - Use permissions to address this



Summary

- Broadcast Receivers provide a scalable framework for communicating between (potentially multiple) processes in Android
- However, there are subtle issues with security
 - The Intent namespace is global
 - `registerReceiver(BroadcastReceiver, IntentFilter)` allows any app to send broadcasts to that registered receiver
 - When a receiver is published in an app's manifest & specifies intent-filters for it, any other app can send broadcasts to it regardless of the specified filters
 - To prevent others from sending to it, make it unavailable to them with `android:exported="false"`

```
<receiver
  android:enabled=
    ["true" | "false"]
  android:exported=
    ["true" | "false"]
  android:icon="drawable resource"
  android:label="string resource"
  android:name="string"
  android:permission="string"
  android:process="string" >
  ...
</receiver>
```

Summary

- Broadcast Receivers provide a scalable framework for communicating between (potentially multiple) processes in Android
- However, there are subtle issues with security
 - The Intent namespace is global
 - `registerReceiver(BroadcastReceiver, IntentFilter)` allows any app to send broadcasts to that registered receiver
 - When a receiver is published in an app's manifest & specifies intent-filters for it, any other app can send broadcasts to it regardless of the filters that are specified
 - `sendBroadcast()` et al allow any other app to receive broadcasts
 - Broadcasts can be restricted to a single app with `Intent.setPackage()`

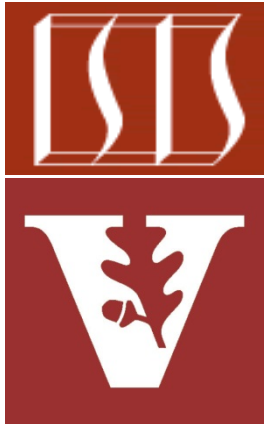


Android Services & Local IPC: Communicating via Pending Intents

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

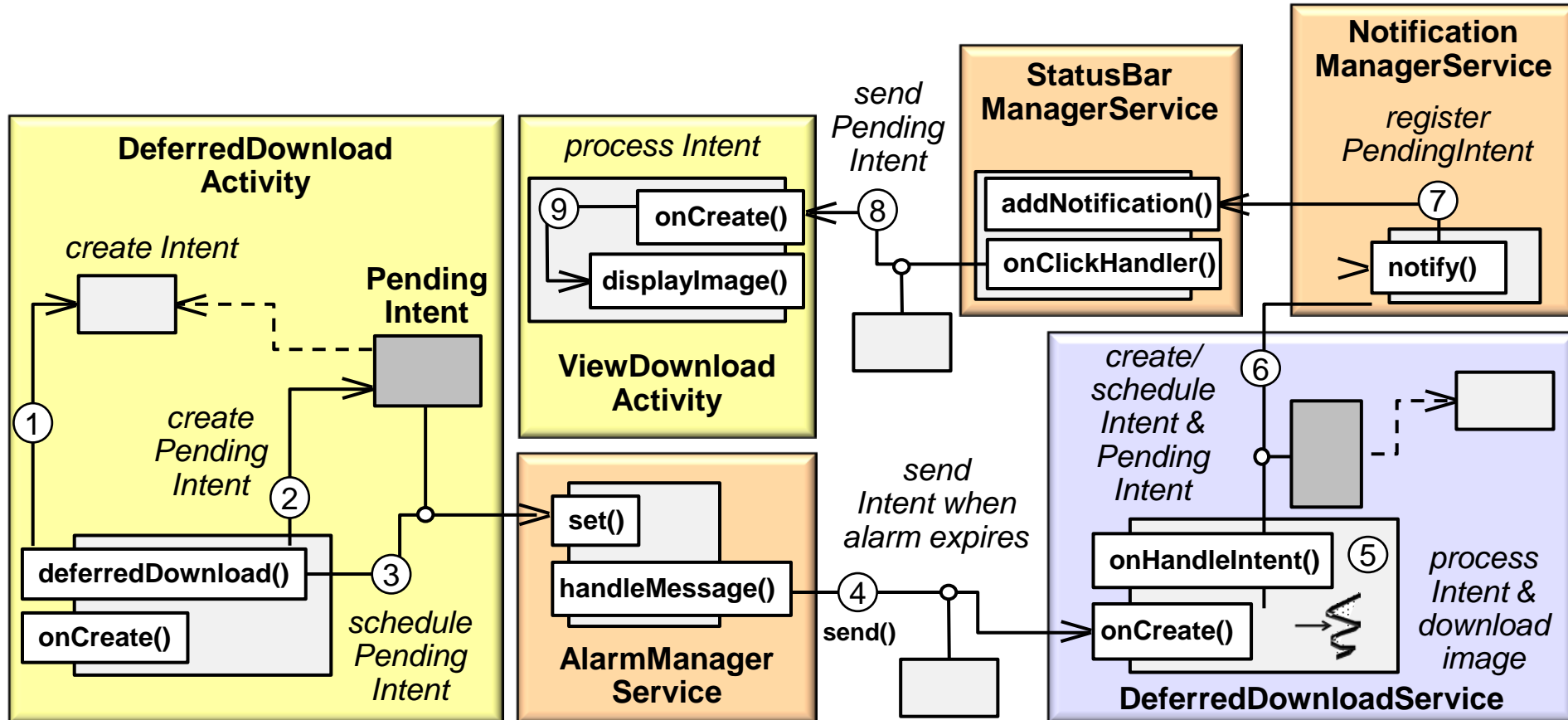
Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



Learning Objectives in this Part of the Module

- Understand how to use Pending Intents to communicate from (Started) Services back to other components (e.g., Activities, Broadcast Receivers, etc.)
- A PendingIntent is a token given to an App to perform an action on your Apps' behalf *irrespective* of whether your App's process is alive



Overview of Pending Intents

- A PendingIntent is a token given by an App to another component that allows it to use the permissions of the App to execute a piece of code
- e.g., Notification Manager, Alarm Manager, or other 3rd party apps

*Notifications in the
notification area*

*Notifications in the
notification drawer*



Overview of Pending Intents

- A PendingIntent is a token given by an App to another component that allows it to use the permissions of the App to execute a piece of code
 - e.g., Notification Manager, Alarm Manager, or other 3rd party apps
- The token maintained by the system represents an Intent & the action to perform on that Intent later
- Can be configured to work irrespective of whether the original App process is alive or not

*Start an Activity
to read email*



Overview of Pending Intents

- A PendingIntent is a token given by an App to another component that allows it to use the permissions of the App to execute a piece of code
- PendingIntents can be created via various methods, e.g.:
 - getActivity() on PendingIntent
 - The PendingIntent returned by this method starts a new Activity when send() is called on it

```
public static PendingIntent getActivity (Context  
context, int requestCode, Intent intent, int flags) Added in API level 1
```

Retrieve a PendingIntent that will start a new activity, like calling `Context.startActivity(Intent)`. Note that the activity will be started outside of the context of an existing activity, so you must use the `Intent.FLAG_ACTIVITY_NEW_TASK` launch flag in the Intent.

For security reasons, the `Intent` you supply here should almost always be an *explicit intent*, that is specify an explicit component to be delivered to through `setClass(android.content.Context, Class) Intent.setClass`

Parameters

<i>context</i>	The Context in which this PendingIntent should start the activity.
<i>requestCode</i>	Private request code for the sender (currently not used).
<i>intent</i>	Intent of the activity to be launched.
<i>flags</i>	May be <code>FLAG_ONE_SHOT</code> , <code>FLAG_NO_CREATE</code> , <code>FLAG_CANCEL_CURRENT</code> , <code>FLAG_UPDATE_CURRENT</code> , or any of the flags as supported by <code>Intent.fillIn()</code> to control which unspecified parts of the intent that can be supplied when the actual send happens.

Returns

Returns an existing or new PendingIntent matching the given parameters. May return null only if `FLAG_NO_CREATE` has been supplied.



Overview of Pending Intents

- A PendingIntent is a token given by an App to another component that allows it to use the permissions of the App to execute a piece of code
- PendingIntents can be created via various methods, e.g.:
 - getActivity() on PendingIntent
 - getBroadcast() on PendingIntent
 - The PendingIntent returned by this method sends a broadcast to a Receiver when send() is called on it

```
public static PendingIntent getBroadcast (Context  
context, int requestCode, Intent intent, int flags) Added in API level 1
```

Retrieve a PendingIntent that will perform a broadcast, like calling `Context.sendBroadcast()`.

For security reasons, the `Intent` you supply here should almost always be an *explicit intent*, that is specify an explicit component to be delivered to through `setClass(android.content.Context, Class)` `Intent.setClass`

Parameters

<i>context</i>	The Context in which this PendingIntent should perform the broadcast.
<i>requestCode</i>	Private request code for the sender (currently not used).
<i>intent</i>	The Intent to be broadcast.
<i>flags</i>	May be <code>FLAG_ONE_SHOT</code> , <code>FLAG_NO_CREATE</code> , <code>FLAG_CANCEL_CURRENT</code> , <code>FLAG_UPDATE_CURRENT</code> , or any of the flags as supported by <code>Intent.fillIn()</code> to control which unspecified parts of the intent that can be supplied when the actual send happens.

Returns

Returns an existing or new PendingIntent matching the given parameters. May return null only if `FLAG_NO_CREATE` has been supplied.

Overview of Pending Intents

- A PendingIntent is a token given by an App to another component that allows it to use the permissions of the App to execute a piece of code
- PendingIntents can be created via various methods, e.g.:
 - getActivity() on PendingIntent
 - getBroadcast() on PendingIntent
 - getService() on PendingIntent
 - The PendingIntent returned by this method starts a new Service when send() is called on it

```
public static PendingIntent getService (Context  
context, int requestCode, Intent intent, int flags) Added in API level 1
```

Retrieve a PendingIntent that will start a service, like calling `Context.startService()`. The start arguments given to the service will come from the extras of the Intent.

For security reasons, the `Intent` you supply here should almost always be an *explicit intent*, that is specify an explicit component to be delivered to through `setClass(android.content.Context, Class)` `Intent.setClass`

Parameters

<i>context</i>	The Context in which this PendingIntent should start the service.
<i>requestCode</i>	Private request code for the sender (currently not used).
<i>intent</i>	An Intent describing the service to be started.
<i>flags</i>	May be <code>FLAG_ONE_SHOT</code> , <code>FLAG_NO_CREATE</code> , <code>FLAG_CANCEL_CURRENT</code> , <code>FLAG_UPDATE_CURRENT</code> , or any of the flags as supported by <code>Intent.fillIn()</code> to control which unspecified parts of the intent that can be supplied when the actual send happens.

Returns

Returns an existing or new PendingIntent matching the given parameters. May return null only if `FLAG_NO_CREATE` has been supplied.

Overview of Pending Intents

- A `PendingIntent` is a token given by an App to another component that allows it to use the permissions of the App to execute a piece of code
- `PendingIntents` can be created via various methods, e.g.:
 - `getActivity()` on `PendingIntent`
 - `getBroadcast()` on `PendingIntent`
 - `getService()` on `PendingIntent`
 - `createPendingResult()` on `Activity`
 - The `PendingIntent` returned by this method sends data back to the Activity via its method `onActivityResult()`

```
public PendingIntent createPendingResult (int  
requestCode, Intent data, int flags)
```

Added in API level 1

Create a new `PendingIntent` object which you can hand to others for them to use to send result data back to your `onActivityResult(int, int, Intent)` callback. The created object will be either one-shot (becoming invalid after a result is sent back) or multiple (allowing any number of results to be sent through it).

Parameters

- | | |
|--------------------|--|
| <i>requestCode</i> | Private request code for the sender that will be associated with the result data when it is returned. The sender can not modify this value, allowing you to identify incoming results. |
| <i>data</i> | Default data to supply in the result, which may be modified by the sender. |
| <i>flags</i> | May be <code>PendingIntent.FLAG_ONE_SHOT</code> , <code>PendingIntent.FLAG_NO_CREATE</code> , <code>PendingIntent.FLAG_CANCEL_CURRENT</code> , <code>PendingIntent.FLAG_UPDATE_CURRENT</code> , or any of the flags as supported by <code>Intent.fillIn()</code> to control which unspecified parts of the intent that can be supplied when the actual send happens. |

Returns

Returns an existing or new `PendingIntent` matching the given parameters. May return null only if `PendingIntent.FLAG_NO_CREATE` has been supplied.

Using PendingIntent w/AlarmManager Service

- PendingIntents are often used with alarms
- Activity creates & schedules a PendingIntent with the Alarm Service

Cause the alarm to restart the Activity when it expires

```
PendingIntent pi;  
AlarmManager mgr;  
  
void onCreate(Bundle b) {  
    AlarmManager mgr =(AlarmManager)  
        getSystemService  
            (ALARM_SERVICE);  
    Intent replyIntent = new Intent();  
    ... // Set "extras" in replyIntent  
    pi = createPendingResult  
        (ALARM_ID, replyIntent, 0);  
    mgr.setRepeating  
        (AlarmManager.  
            ELAPSED_REALTIME_WAKEUP,  
            SystemClock.elapsedRealtime()  
            + PERIOD, PERIOD, pi);  
    finish();  
}
```

Activity

Alarm
Manager

Using PendingIntent w/AlarmManager Service

- PendingIntents are often used with alarms
- Activity creates & schedules a PendingIntent with the Alarm Service

```
void setRepeating(int type,
    long triggerAtTime,
    long interval,
    PendingIntent operation) {
    Alarm alarm = new Alarm();
    ...
    alarm.when = triggerAtTime;
    alarm.repeatInterval = interval;
    alarm.operation = operation;

    Message msg = Message.obtain();
    msg.what = ALARM_EVENT;
    ...
    mHandler.sendMessageAtTime
        (msg, alarm.when);
}
```

AlarmManager maintains its schedule outside of an App's process, so it can give the App control, even if it has to start up a new process along the way

**Alarm
Manager**

Using PendingIntent w/AlarmManager Service

- PendingIntents are often used with alarms
 - Activity creates & schedules a PendingIntent with the Alarm Service
- When the timer expires the Alarm Service sends a reply back to the Activity

```
class AlarmHandler extends
    Handler {
    void handleMessage(Message m) {
        ...
        alarm.operation.send();
        ...
    }
}
```

**Alarm
Manager**

Using PendingIntent w/AlarmManager Service

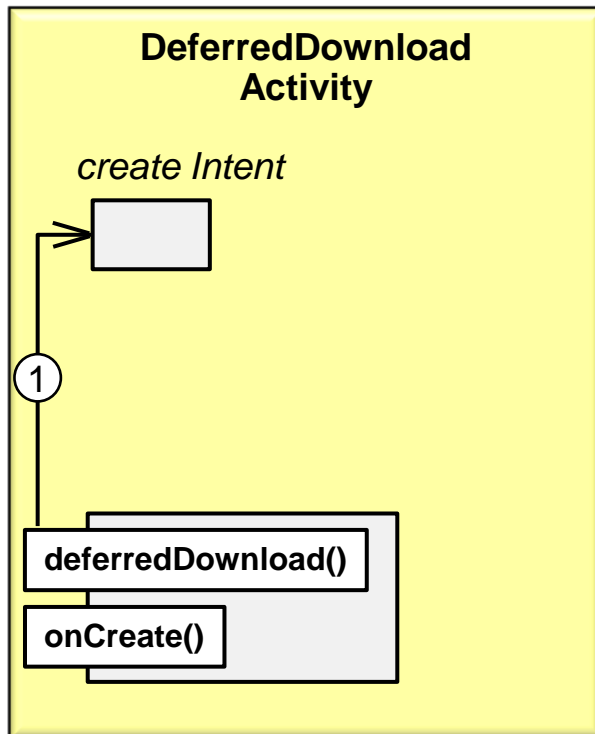
- PendingIntents are often used with alarms
 - Activity creates & schedules a PendingIntent with the Alarm Service
 - When the timer expires the Alarm Service sends a reply back to the Activity
 - The Activity is restarted & its onActivityResult() method handles the reply

```
void onActivityResult  
(int requestCode,  
 int resultCode,  
 Intent data) {  
    if (requestCode == ALARM_ID)  
    {  
        // Do something with  
        // data in the Intent  
    }  
}
```



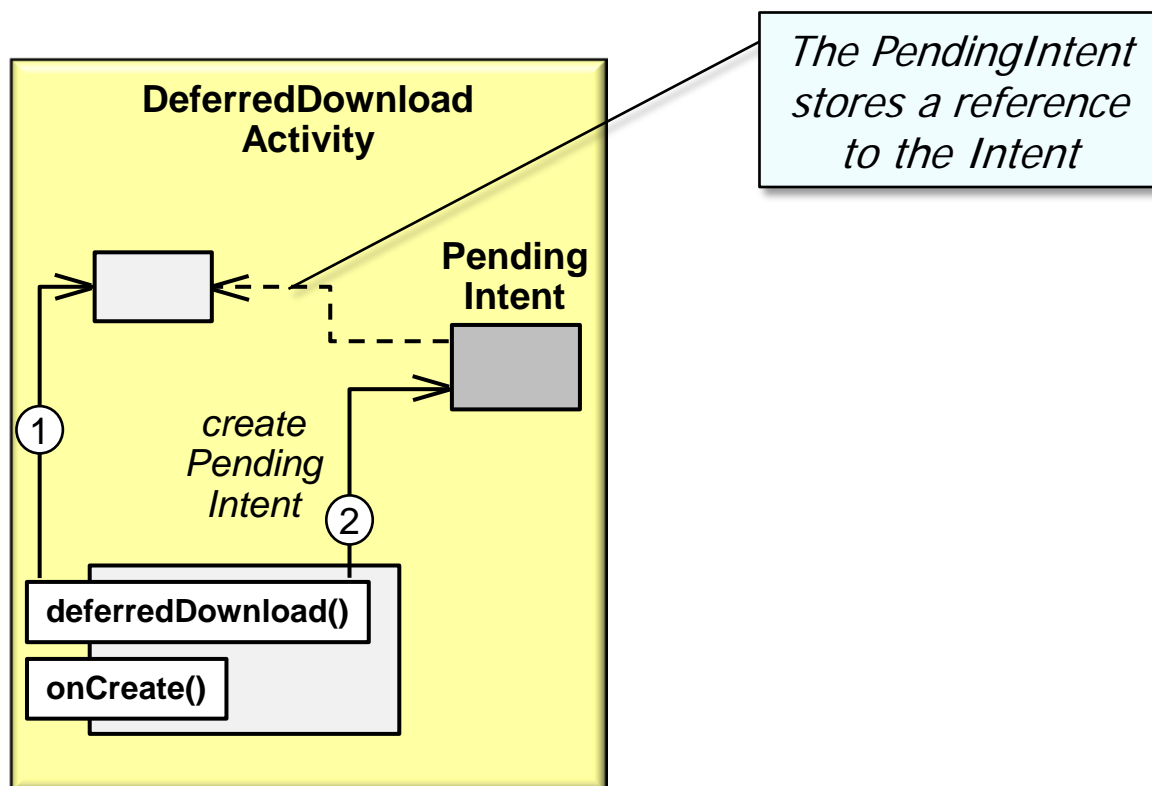
Using Pending Intents in Deferred Download App

- DownloadActivity creates a PendingIntent that's registered with the Alarm Service to start DeferredDownloadService to download an image in the future
- DeferredDownloadService uses Notification Service to inform user when the image has been downloaded



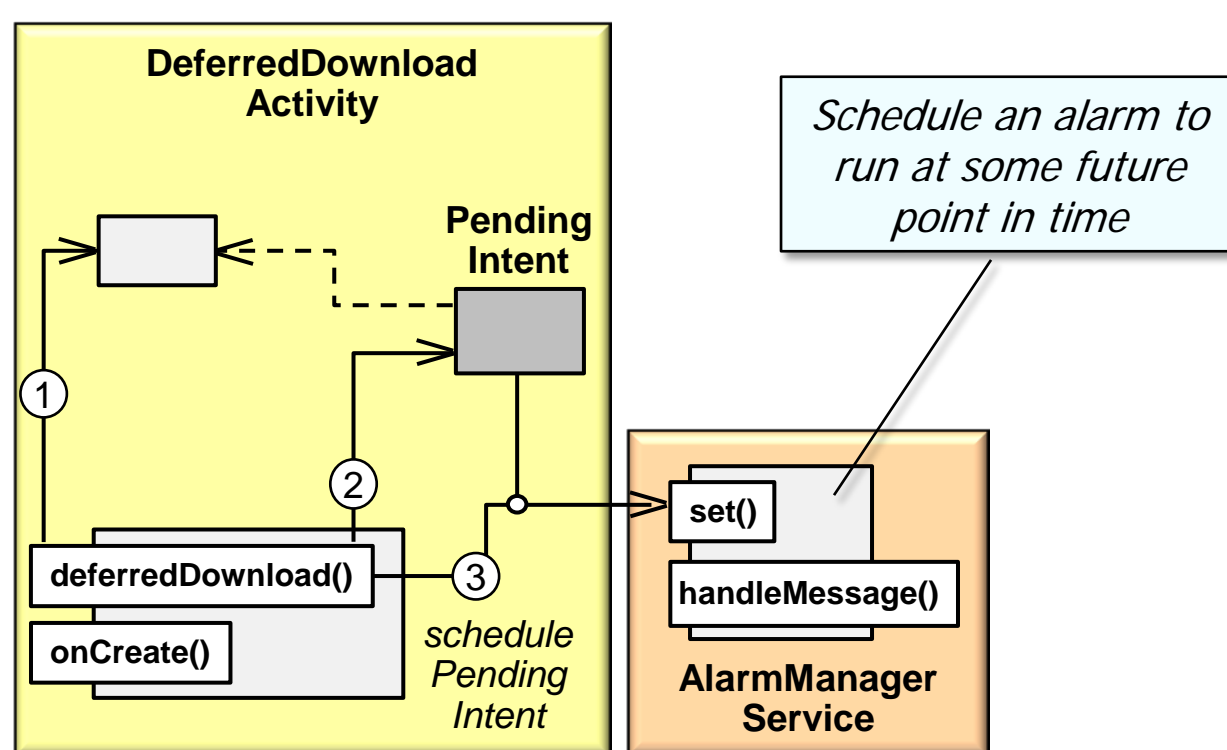
Using Pending Intents in Deferred Download App

- DownloadActivity creates a PendingIntent that's registered with the Alarm Service to start DeferredDownloadService to download an image in the future
- DeferredDownloadService uses Notification Service to inform user when the image has been downloaded



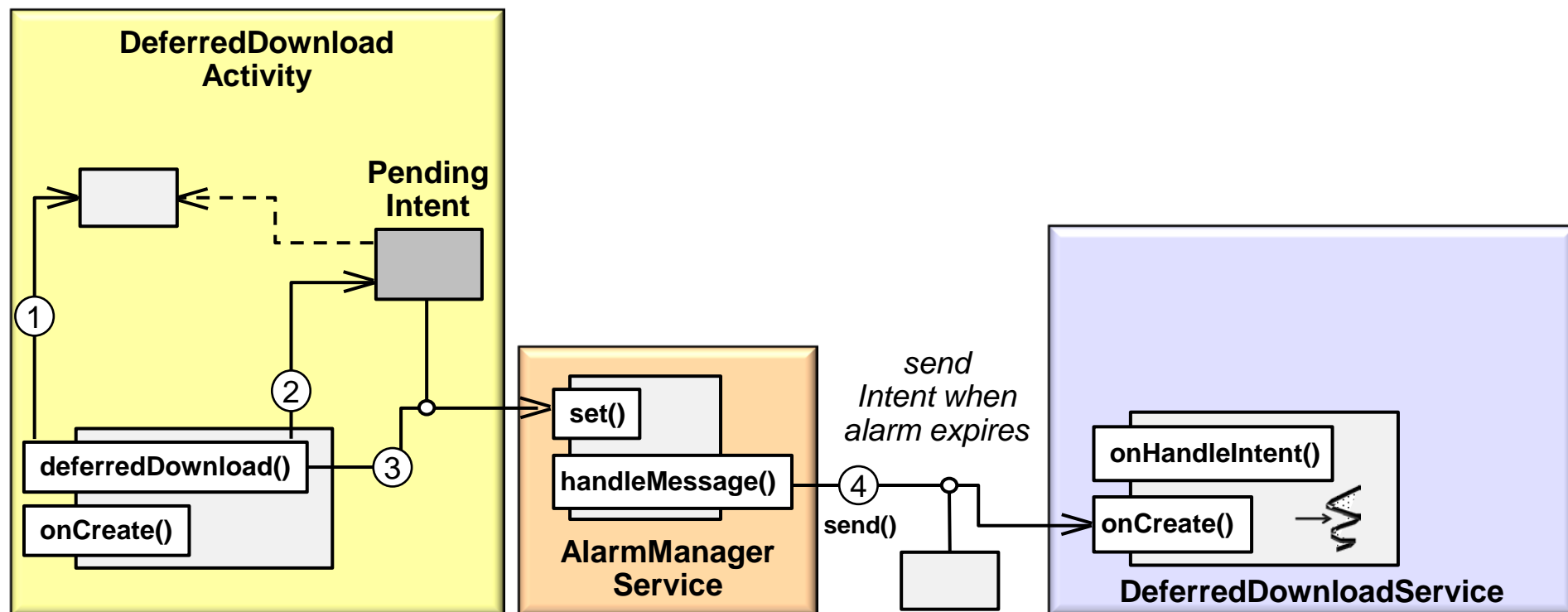
Using Pending Intents in Deferred Download App

- DownloadActivity creates a PendingIntent that's registered with the Alarm Service to start DeferredDownloadService to download an image in the future
- DeferredDownloadService uses Notification Service to inform user when the image has been downloaded



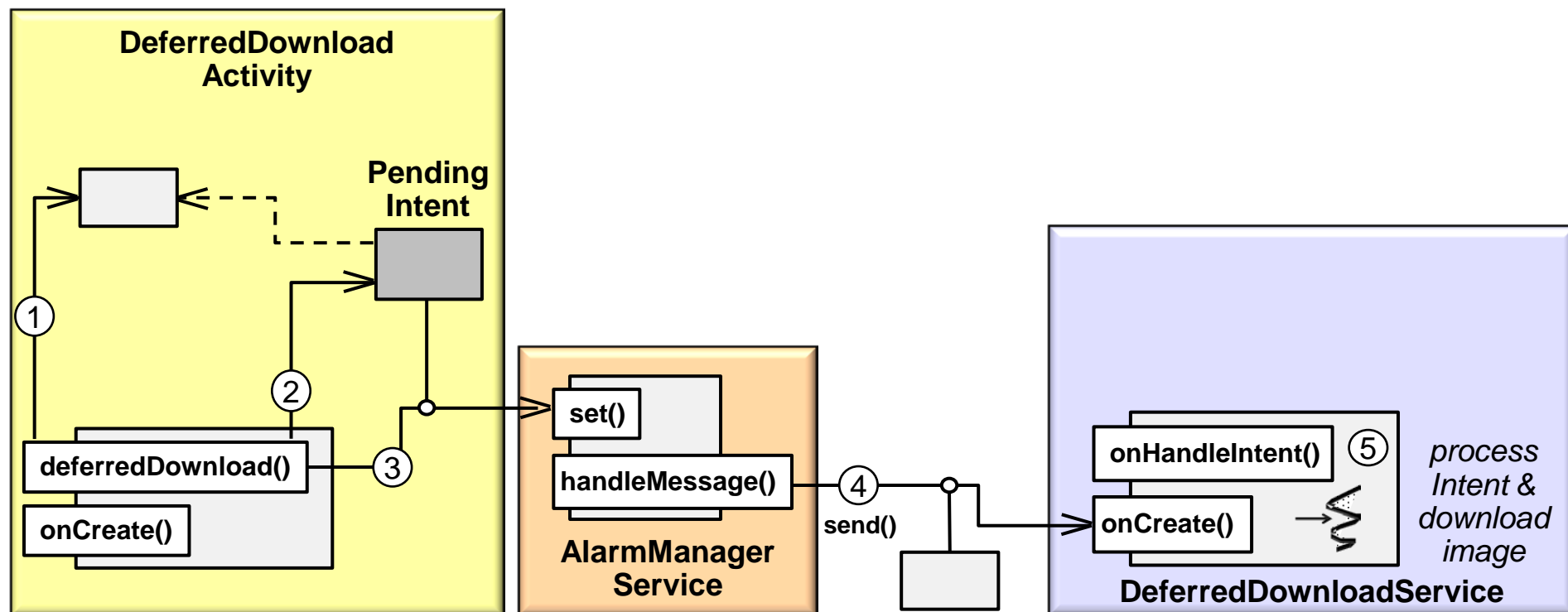
Using Pending Intents in Deferred Download App

- DownloadActivity creates a PendingIntent that's registered with the Alarm Service to start DeferredDownloadService to download an image in the future
- DeferredDownloadService uses Notification Service to inform user when the image has been downloaded



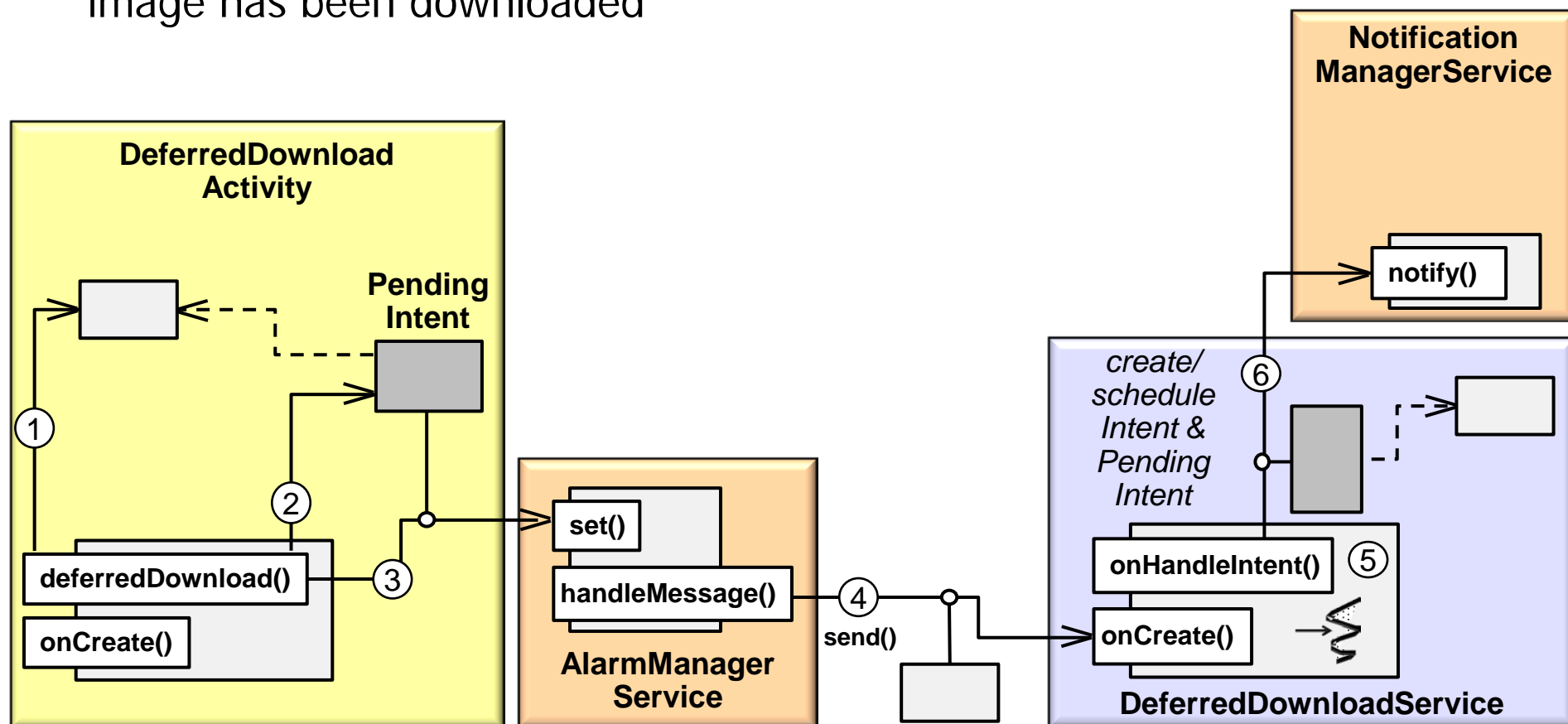
Using Pending Intents in Deferred Download App

- DownloadActivity creates a PendingIntent that's registered with the Alarm Service to start DeferredDownloadService to download an image in the future
- DeferredDownloadService uses Notification Service to inform user when the image has been downloaded



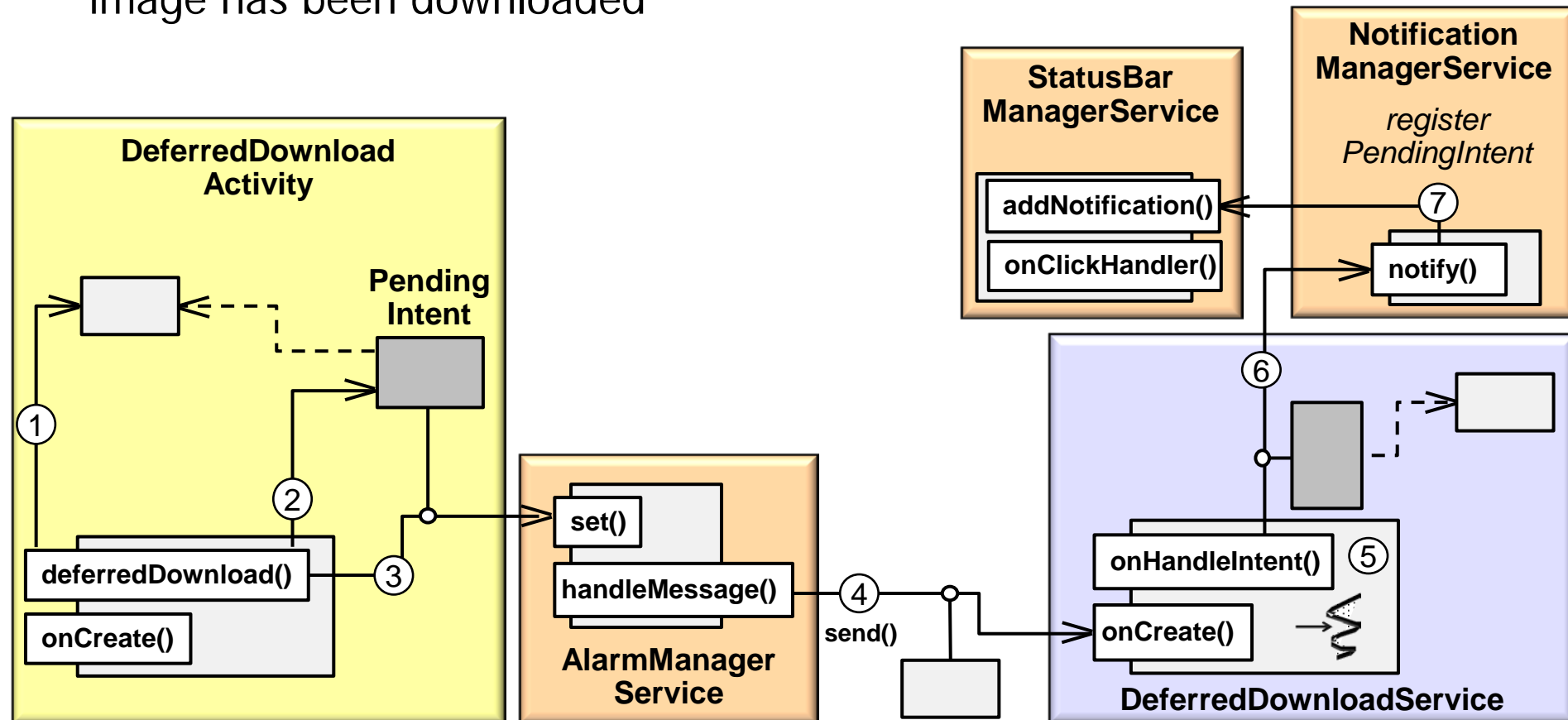
Using Pending Intents in Deferred Download App

- DownloadActivity creates a PendingIntent that's registered with the Alarm Service to start DeferredDownloadService to download an image in the future
- DeferredDownloadService uses Notification Service to inform user when the image has been downloaded



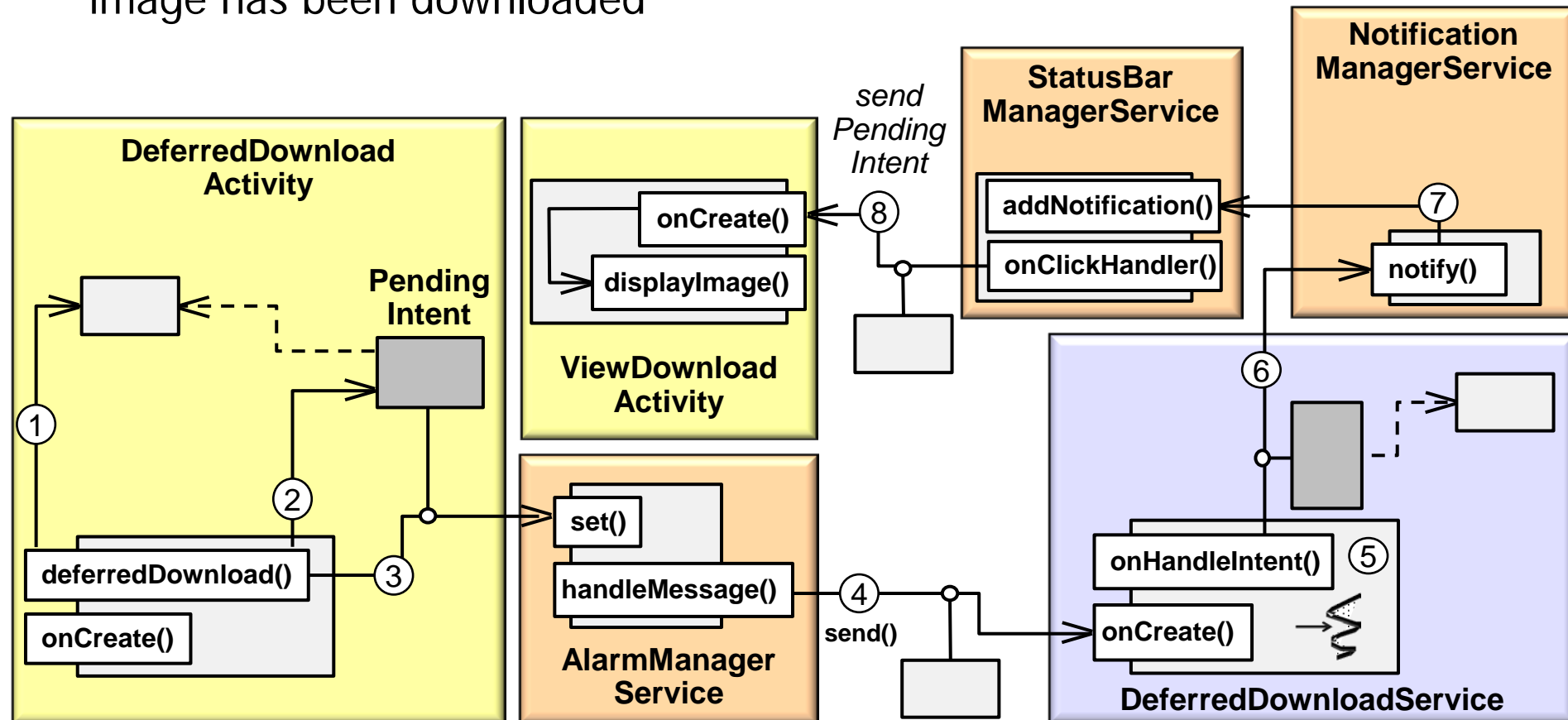
Using Pending Intents in Deferred Download App

- DownloadActivity creates a PendingIntent that's registered with the Alarm Service to start DeferredDownloadService to download an image in the future
- DeferredDownloadService uses Notification Service to inform user when the image has been downloaded



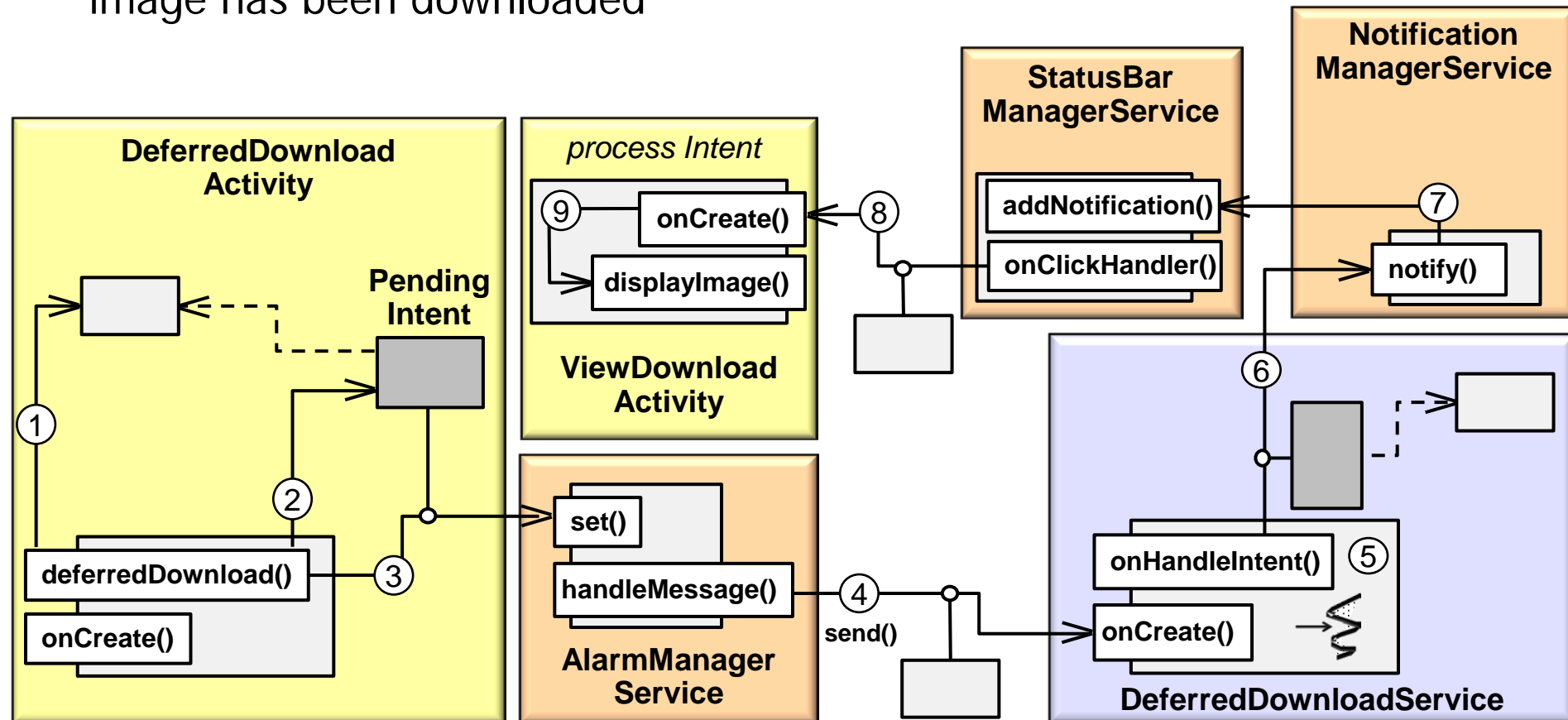
Using Pending Intents in Deferred Download App

- DownloadActivity creates a PendingIntent that's registered with the Alarm Service to start DeferredDownloadService to download an image in the future
- DeferredDownloadService uses Notification Service to inform user when the image has been downloaded





Using Pending Intents in Deferred Download App

- DownloadActivity creates a PendingIntent that's registered with the Alarm Service to start DeferredDownloadService to download an image in the future
- DeferredDownloadService uses Notification Service to inform user when the image has been downloaded



Programming DeferredDownloadActivity

- This Activity creates a PendingIntent & schedules it with Alarm Service

```
public class DeferredDownloadActivity extends Activity {  
    ...  
    public void initiateDeferredDownload(View v) {  
        Intent intent = new Intent(DownloadActivity.this,  
                                   DeferredDownloadService.class);  
        PendingIntent sender = PendingIntent.getService(  
                                   DownloadActivity.this, 0,  
                                   intent, 0);  
        Create PendingIntent that starts a Service to download the image   
  
        AlarmManager am =  
            (AlarmManager) getSystemService(ALARM_SERVICE);  
  
        am.set(AlarmManager.ELAPSED_REALTIME_WAKEUP,  
              downloadTime,  
              sender);  
        Schedule an alarm to trigger the PendingIntent at the desired time   
    }  
}
```

Programming DeferredDownloadService

- DeferredDownloadService uses the Android Notification Service to alert user when a requested image has been downloaded

```
public class DeferredDownloadService extends IntentService {  
    ...
```

```
    protected void onHandleIntent(Intent intent) {  
        String pathname = downloadImage(intent);
```

Code to downloading image to pathname goes here

```
        Intent viewDownloadIntent =  
            new Intent(this, ViewDownloadActivity.class);  
        intent.setData(pathname);
```

Prepare Intent to trigger if notification is selected

```
        PendingIntent pendingIntent =  
            PendingIntent.getActivity(this, 0, viewDownloadIntent,  
                                    0);
```

... Create PendingIntent to register with Notification Service

Programming DeferredDownloadService

- DeferredDownloadService uses Notification Service to alert user when a requested image has been downloaded

```
public class DeferredDownloadService extends IntentService {  
    ...  
    protected void onHandleIntent(Intent intent) {  
        ...  
        Notification notification = new Notification.Builder(this)  
            .setContentTitle("Image download complete")  
            .setContentText(pathname).setSmallIcon(R.drawable.icon)  
            .setContentIntent(contentIntent).build();
```

Build notification



```
        NotificationManager nm = (NotificationManager)  
            getSystemService(NOTIFICATION_SERVICE);  
        notification.flags |= Notification.FLAG_AUTO_CANCEL;  
        notificationManager.notify(0, notification);  
        ...
```



Register with the Notification Service

Programming ViewDownloadActivity

- This Activity is called when the user selects a notification that indicates the download has succeeded

```
public class ViewDownloadActivity extends Activity {  
    ...  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        Intent callersIntent = getIntent();  
        String pathname = callersIntent.getData().toString();
```

Get the pathname from the Intent
that started this Activity



```
        displayImage(pathname);
```

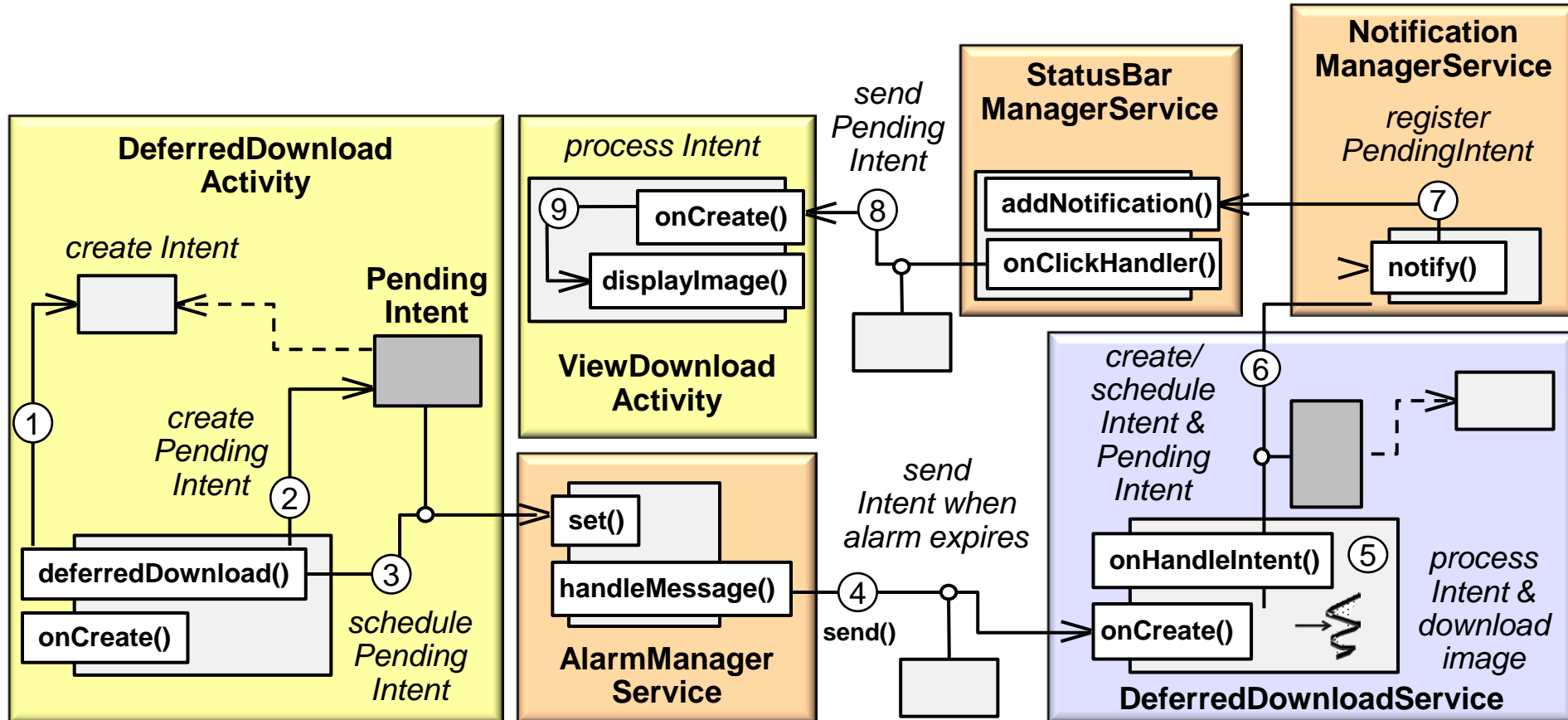


Display the image

```
    }
```

Summary

- Pending Intents provide a powerful framework for an App to delegate some processing to another App at some future time or in some other context



Summary

- Pending Intents provide a powerful framework for an App to delegate some processing to another App at some future time or in some other context
- Pending Intents can also be used to communicate from a (Started) Service back to some other Android component
- They are a bit complicated to use...

