Services & Interprocess Communication (IPC)

Programming the Android Platform

CS 282
Principles of Operating Systems II
Systems Programming for Android

Service Overview

- A Service is an application component that can perform long-running operations in the background & does not provide a direct user interface
  - e.g., a service might handle network transactions, play music, perform file I/O, interact with a content provider, or run periodic tasks, all from the background
- Another application component can start a service & it will continue to run in the background even if the user switches to another application/activity
- A component can also bind to a service to interact with it & perform local (or even remote) IPC

  http://developer.android.com/guide/components/services.html

Two Forms of Services

- **Started** – a service is "started" when an application component starts it by calling startService()
  - A started service often performs a single operation & doesn't return a result to the caller
    - e.g., it might download or upload a file over TCP
  - When the operation is done, the service can stop itself

- **Bound** – A service is "bound" when an application component binds to it by calling bindService()
  - A bound service offers a client-server interface that allows components to interact with the service, send requests, get results, & even do so across processes with IPC
  - A bound service runs only as long as another application component is bound to it

http://developer.android.com/guide/components/services.html

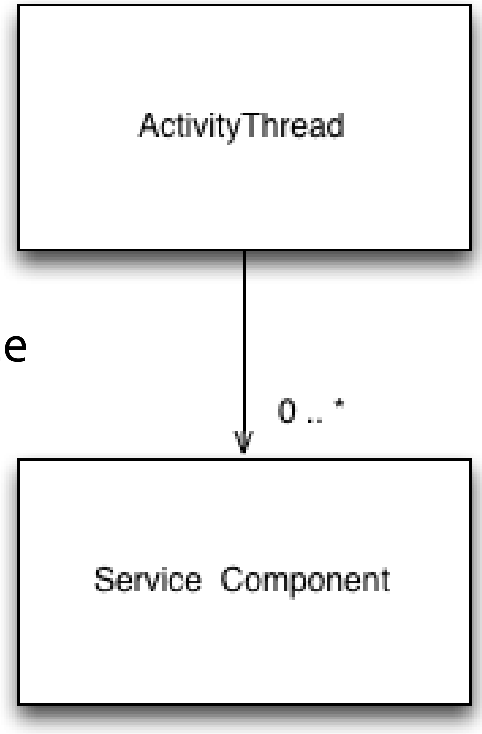
Example Services

- Logging Service
  - Client Activity sends log messages to service
  - Service writes messages to a log console
- Music Playing Service
  - Client Activity tells service to play a music file
  - Services plays music in background (even if Client Activity pauses or terminates)
- SMS, MMS, & Email Services
  - Manage messaging operations, such as sending data, text, & pdu messages
- ID Service
  - Client Activity requests system-wide unique ID
  - Service returns ID to Client

See packages/apps in Android source code for many services

Overview of a Service

- Implementing a Service is similar to implementing an Activity
  - e.g., inherit from an Android base class, override lifecycle methods, include the service in an AndroidManifest.xml file, etc.
- Services have their own lifecycle methods
  - onCreate(), which, as with activities, is called when the service process is created, by any means
  - onStartCommand(), which is called each time the service is sent a command via startService()
  - onBind(), which is called whenever a client binds to the service via bindService()
  - onDestroy() which is called as the service is being shut down



Note: Services do not automatically run in their own threads!!!

Overview of a Service (cont'd)

- You need to add the service to your AndroidManifest. xml file
- Simply add a <service> element as a child of the <application> element
- You'll also need to provide android:name to reference your service class
- Use android:process= ":my_process" to run the service in its own process

MMS Services

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

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

Music Service

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

http://developer.android.com/reference/android/app/Service.html

Programming Started Services

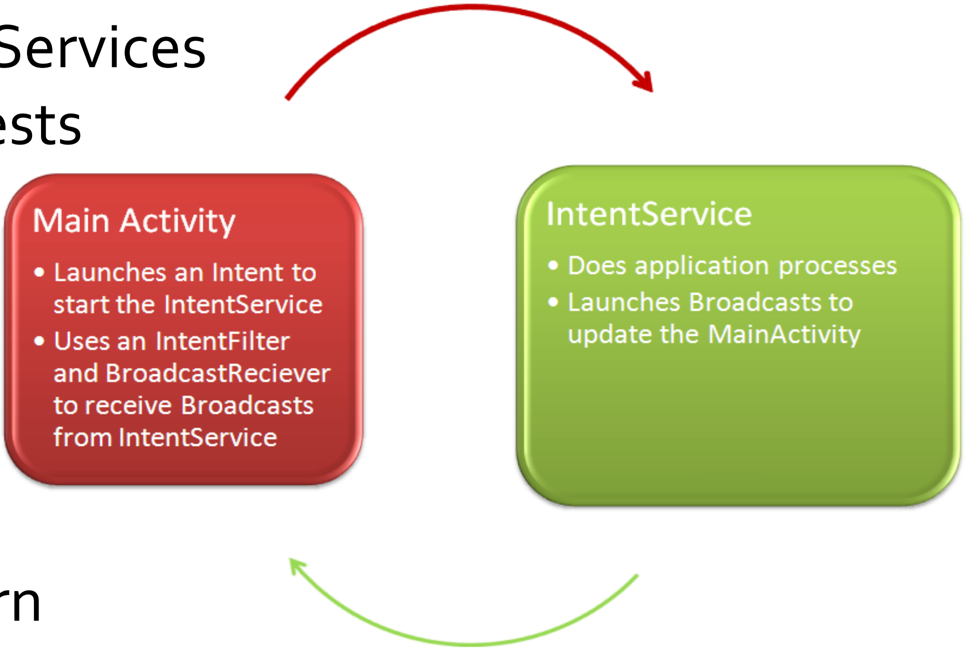
- A Started Service is activated via Context.startService()
  - The Intent identifies the service to communicate with & supplies parameters (via Intent extras) to tell the service what to do
- startService() does not block
  - If the service is not already running it will be started & will receive the Intent via onStartCommand()
  - Services often perform a single operation & terminate themselves
  - Running Services can also be halted via stopService()

- Started Services don't return results to callers, but do return values via onStartCommand():
  - START_STICKY – don't re-deliver Intent to onStartCommand()
  - START_REDELIVER_INTENT – service should be restarted via a call to onStartCommand(), supplying the same Intent as was delivered this time
  - START_NOT_STICKY – service should remain stopped until explicitly started by application code

http://android-developers.blogspot.com.au/2010/02/service-api-changes-starting-with.html

Overview of IntentService

- The most common Service subclass is IntentService
- IntentService is a base class for Services that handle asynchronous requests (expressed as Intents) on demand
- IntentService is commonly used to implement the Command Processor pattern & implements the Activator pattern
  - See http://www.dre.vanderbilt.edu/~schmidt/CommandProcessor.pdf, http://www.voelter.de/data/pub/CommandRevisited.pdf & http://www.dre.vanderbilt.edu/~schmidt/PDF/ActivatorReloaded.pdf for info on these patterns



Main Activity
• Launches an Intent to start the IntentService
• Uses an IntentFilter and BroadcastReciever to receive Broadcasts from IntentService
IntentService
• Does application processes
• Launches Broadcasts to update the MainActivity

http://developer.android.com/reference/android/app/IntentService.html

Programming an IntentService

- Clients send requests through startService(Intent) calls
  - The service is started as needed, handles each Intent in turn using a worker thread, & stops itself when it runs out of work
  - This "work queue processor" model (aka Command Processor pattern) is commonly used to offload tasks from an application's main thread
  - The IntentService class exists to simplify this pattern & take care of the mechanics
- To use an IntentService, extend the IntentService class & implement the hook method onHandleIntent(Intent)
  - The IntentService will receive the Intents, launch a worker thread, & stop the service as appropriate
- All requests are handled on a single worker thread
  - they may take as long as necessary (& will not block the application's main loop), but only one request will be processed at a time

http://www.vogella.com/articles/AndroidServices/article.html

Service vs. IntentService

- The Service class uses the application's main thread, while IntentService creates a worker thread & uses that thread to run the service
- IntentService creates a queue that passes one intent to onHandleIntent() at a time
  - Implementing a multi-threaded service should therefore often be made by extending Service class directly
- The Service class needs a manual stop using stopSelf()
  - Meanwhile, IntentService automatically stops itself when there is no intent in queue
- IntentService implements onBind() that returns null, which means the IntentService can not be bound by default
- IntentService implements onStartCommand() that places the Intent on its work queue & calls onHandleIntent()

Service vs. Thread vs. AsyncTask

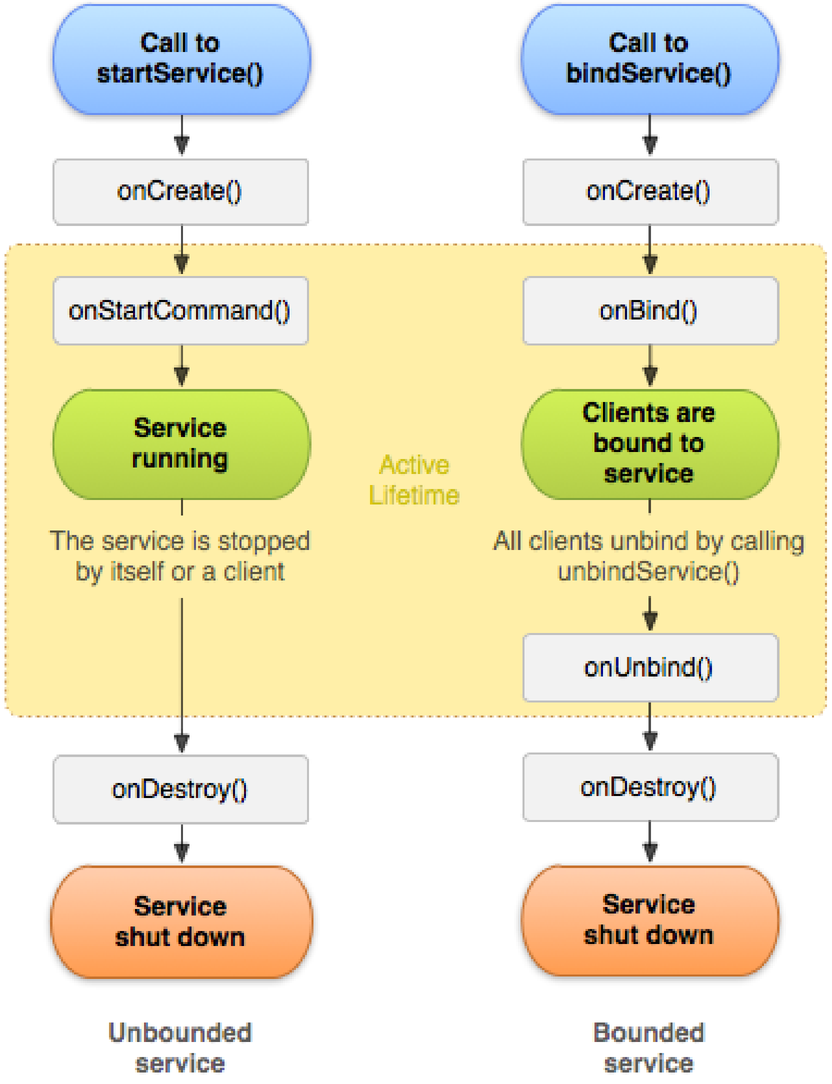
- A Service is **not** a separate process
  - The Service object itself does not imply it is running in its own process
  - Unless otherwise specified, it runs in the same process as the application it is part of
  - It keeps running until stopped by itself, stopped by the user or killed by the system if it needs memory
- A Service is **not** a thread
  - It is not a means itself to do work off of the main thread (to avoid Application Not Responding errors)
- Threads or AsyncTask perform their work in a background thread, so they don't block the main thread
- Since a Service performs its work in the main thread it might block that thread until it finishes when performing an intensive task
  - such as calling a web service
- For intensive tasks a service should run it's work in a background thread

Communicating to Services

- Activities have two ways to send requests or data to a Service
  - **Send a command via startService()**

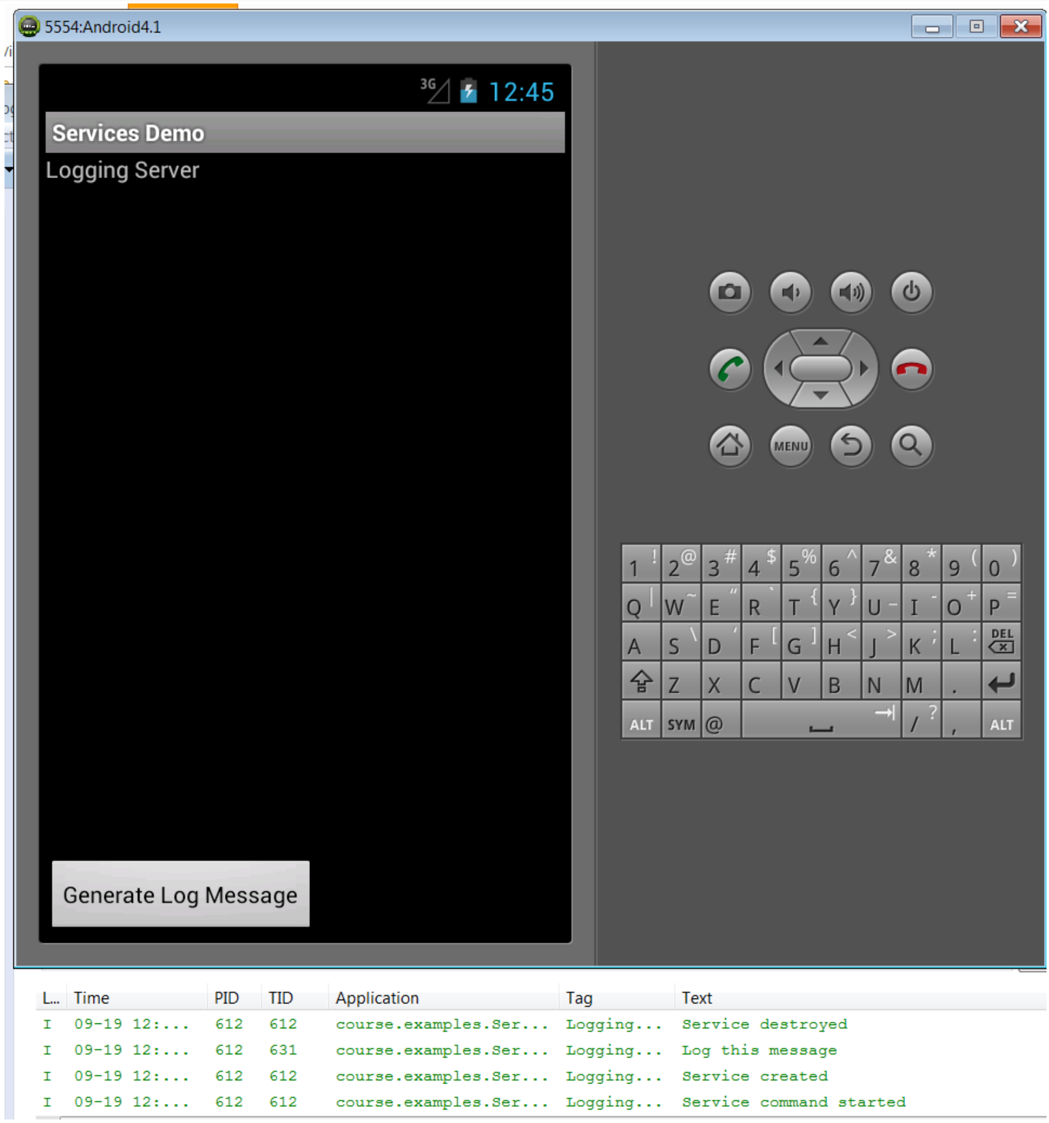
    This requires adding "extras" to the Intent used to start a Service
  - **Bind to a Service via BindService()**

    You can then use the Binder RPC mechanism via an object defined via the Android Interface Definition Language (AIDL) or Messengers
- Depending on how Services are configured in AndroidManifest.xml the communication can be local or remote



Logging Service Example

- Service requests represented as Intents
- Uses the IntentService subclass of Service
- IntentService requests handled sequentially in a single worker thread
- IntentService started & stopped as needed



```
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 Service (cont.)

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

Logging Service (cont.)

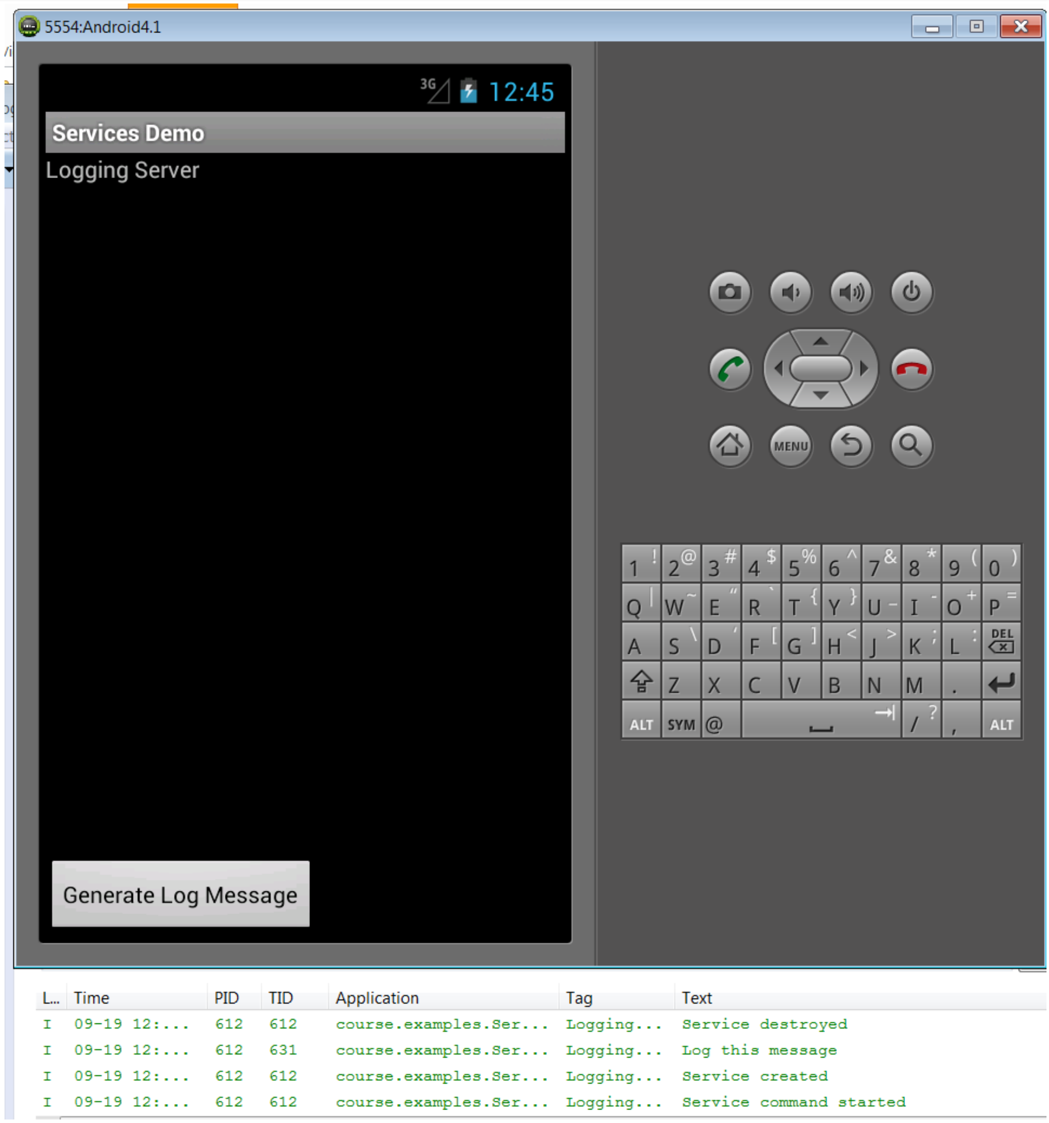
```
public class BGLoggingService extends IntentService {
…
  public int onStartCommand(Intent intent, int flags, int startId) {
    super.onStartCommand(intent, flags, startId);
    return START_NOT_STICKY;
  }
  protected void onHandleIntent(Intent intent) {
    // Optionally create & start new Thread to handle request
    …
    Log.i(TAG,arg.getCharSequenceExtra
       ("course.examples.Services.Logging").toString());
  }
…
}
```

Logging Service (cont.)

```
<application ... >
  <activity android:name=".BGLoggingDemo"
                                          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:enabled="true" android:name=".BGLoggingService" />
 </application>
```

Analysis of the Logging Service

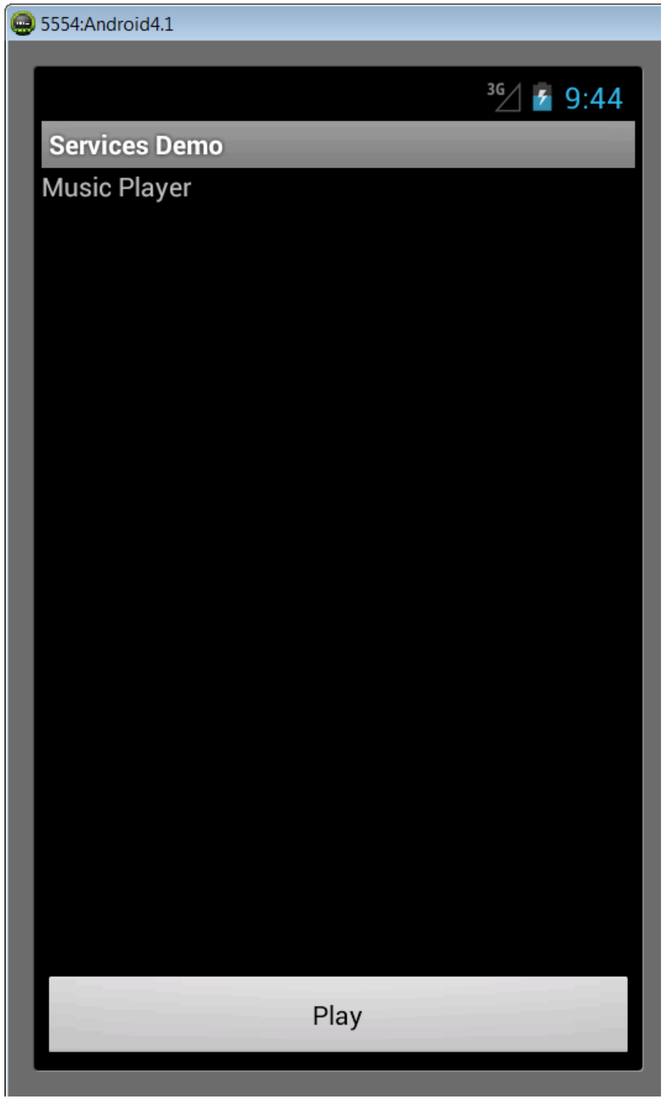
- The LoggingService is a simplified example
  - It doesn't need to be implemented as a Service
  - You could simply do the logging in a new Thread
- Use Services when you want to run a component even when a user is not interacting with the Service's hosting application



```
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
```

Music Player Service

- Client Activity can start/stop playing music via a Service
- If music is playing when client leaves the foreground, music service will continue playing



Music Player Service (cont.)

```
public class MusicService extends Service {
   MediaPlayer player;
    …
   public void onCreate() {
      player = MediaPlayer.create(this, R.raw.braincandy);
      player.setLooping(false);
   }
   public int onStartCommand (Intent intent, int flags, int startid)
    {
      player.start();
      return START_NOT_STICKY;
   }
   …
}
```

Music Player Service (cont.)

```
public class MusicServiceDemo extends Activity {
  public void onCreate(Bundle savedInstanceState) {
    ...
    button.setOnClickListener(new OnClickListener() {
      public void onClick(View src) {
        ...
        startService(
          new Intent(MusicServiceDemo.this,
                          MusicService.class));
      }
    });
  }
}
```

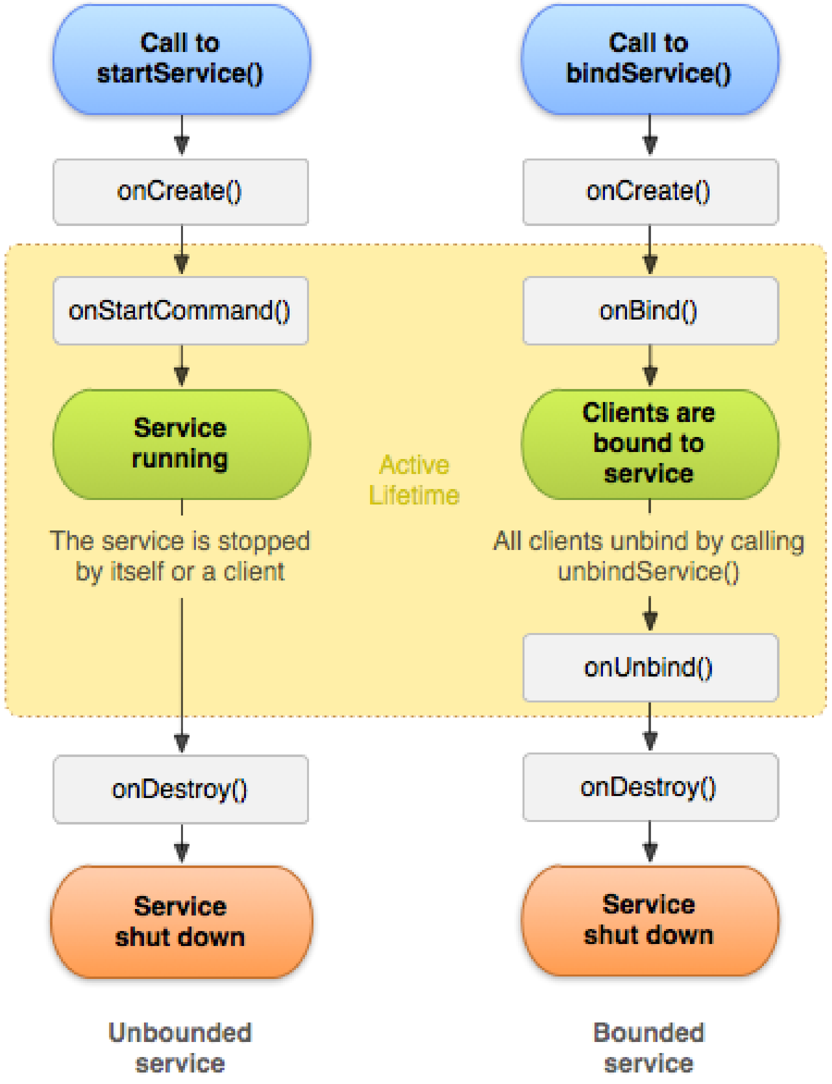
Communicating From Services

- There are several ways to get results from a Service back to an invoking Activity
  - **Use Broadcast Intents**

    This requires having the Activity register a BroadcastReceiver
  - **Use a Messenger object**

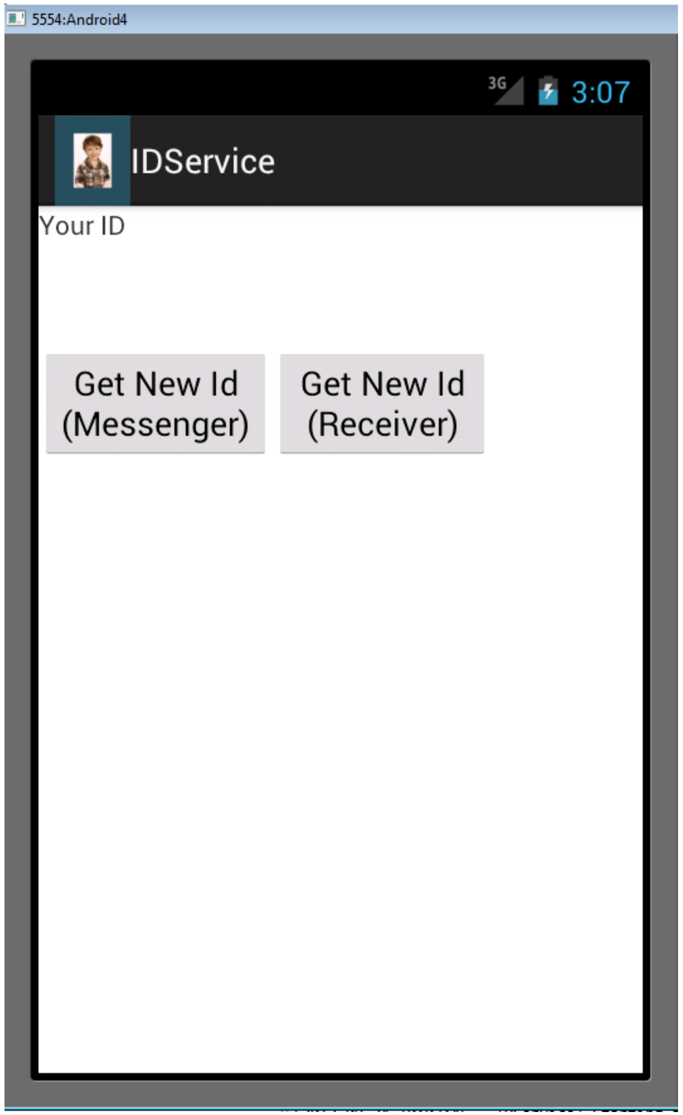
    This object can send messages to an Activity's Handler
  - **Use a Pending Intent**

    Using a PendingIntent to trigger a call to Activity's onActivityResult()



ID Service

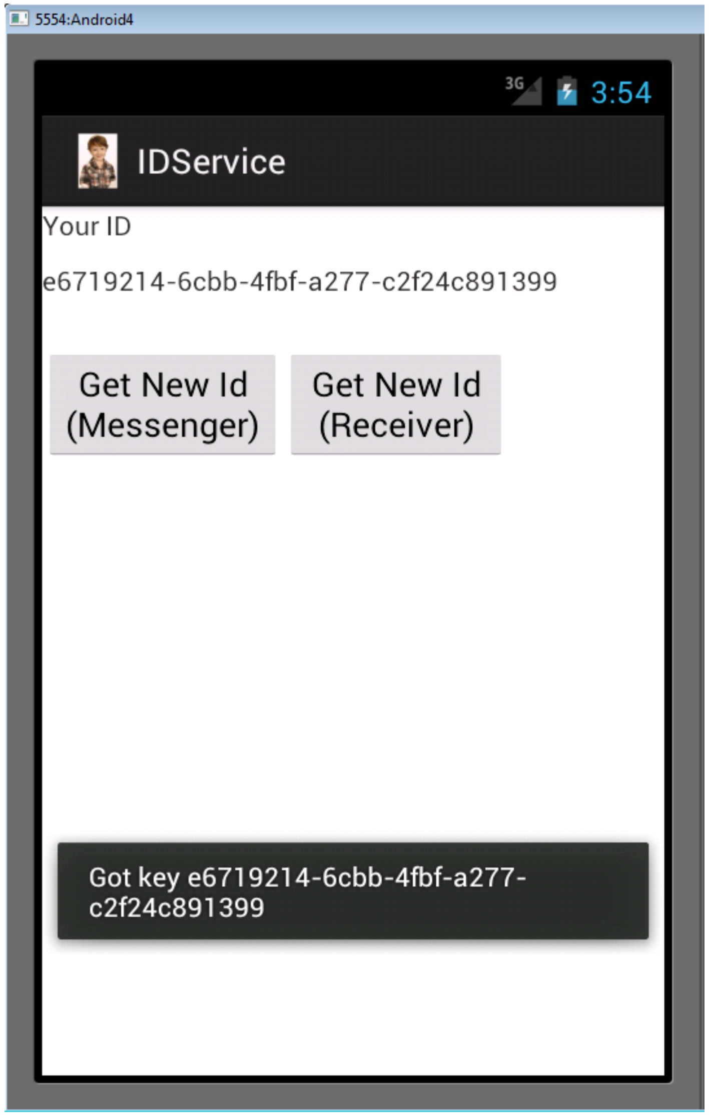
- Client Activity uses a "Started Service" hosted in another process
  - The Service is "sticky"
- Client Activity obtains a unique ID from the Service
- Requires IPC via Broadcast Intents, Messenger Object, & Pending Intent to receive the unique ID



ID Service (cont'd)

- IDService extends Service & generates a unique key

```
public class IDService extends Service {
  private Set<UUID> keys = new HashSet<UUID>();

  public static final String ACTION_COMPLETE =
    "examples.IDService.action.COMPLETE";

  public String getKey() {
    UUID id;
    synchronized (keys) {
      do {
        id = UUID.randomUUID();
      } while (keys.contains(id));
      keys.add(id);
    }
    result = Activity.RESULT_OK;
    return id.toString();
  }
```



Your ID

e6719214-6cbb-4fbf-a277-c2f24c891399

Got key e6719214-6cbb-4fbf-a277-c2f24c891399

ID Service (cont'd)

- Client Activity sends Intents via startService()

```
public void getIdReceiver (View view) {
     Intent intent = new Intent(this,
                                  IDService.class);
     startService(intent);
}

public void getIdMessenger (View view)  {
     Intent intent = new Intent(this,
                                    IDService.class);
     Messenger messenger =
        new Messenger(handler);
     intent.putExtra("MESSENGER", messenger);
     startService(intent);
}
```

```
public void getIdPendingIntent(View view) {
     Intent intent =
        new Intent(this, IDService.class);

     pendingIntent = createPendingResult
           (IDService.KEY_ID,
            new Intent(), 0);

     intent.putExtra("PENDING_INTENT",
                          pendingIntent);

     startService(intent);
  }
```

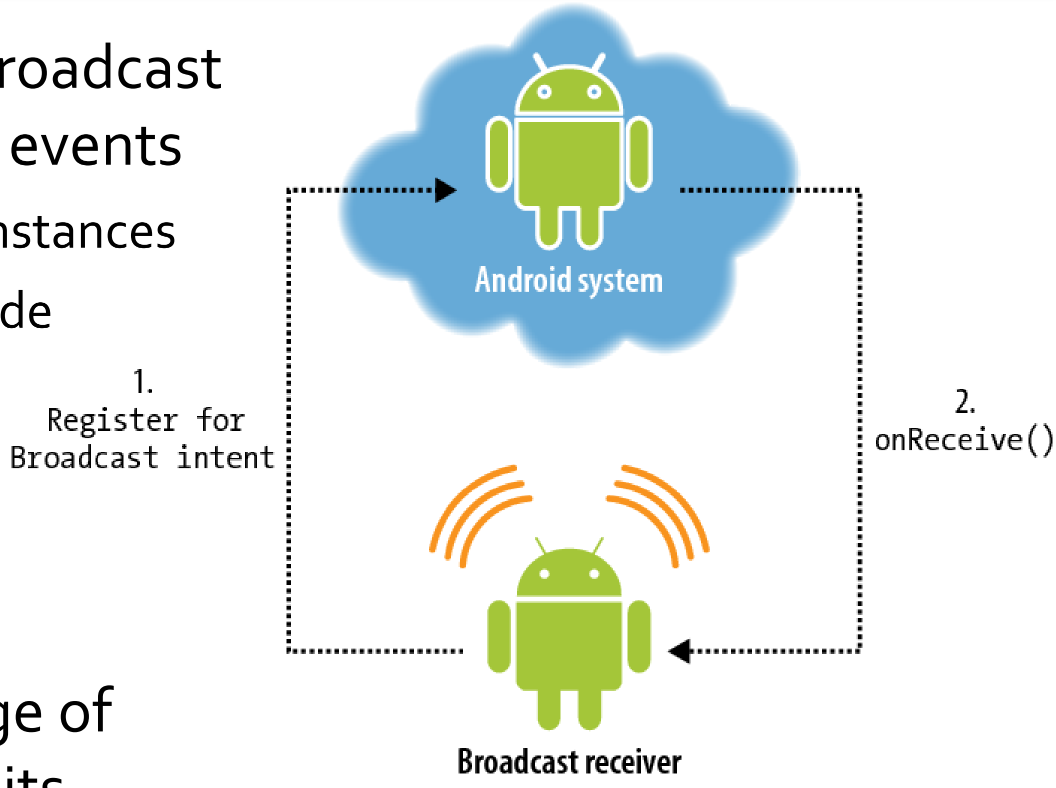
ID Service (cont'd)

- Service receives Intents via onStartCommand()

```
public int onStartCommand(Intent intent, int flags, int startId) {
    Bundle extras = intent.getExtras();
    if (extras != null) {
      if (extras.get("MESSENGER") != null)
        getKeyMessenger (intent);
      else if (extras.get("PENDING_INTENT") != null)
        getKeyPendingIntent (intent);
      else
        Log.e(getClass().getName(), "Unknown extras");
    }
    else
      getKeyReceiver (intent);

    return Service.START_STICKY;
  }
```

Overview of Broadcast Receivers

- Components that listen for broadcast events & receive/react to the events
  - Events implemented as Intent instances
  - Events are broadcast system-wide
  - Interested BroadcastReceivers receive Intent via onReceive()
- BroadcastReceivers have no user interface
- Android supports a wide range of notification mechanisms via its Intents Framework



http://developer.android.com/reference/android/content/BroadcastReceiver.html

Using Broadcast Intents

- Service replies to Activity via the method sendBroadcast()

```
private void getKeyReceiver
   (Intent intent) {
      Intent replyIntent =
          new Intent
            (ACTION_COMPLETE);
      replyIntent.putExtra
         (RESULT_KEY,
          getKey());
      sendBroadcast(replyIntent);
   }
```

- Client Activity is a BroadcastReceiver

```
private BroadcastReceiver onEvent = new BroadcastReceiver()  {
          public void onReceive(Context ctxt, Intent intent) {
             String key =
                 intent.getStringExtra(IDService.RESULT_KEY);
             output.setText(key);
          }
};

public void onResume() {
      super.onResume();
      IntentFilter filter =
          new IntentFilter(IDService.ACTION_COMPLETE);
      registerReceiver(onEvent, filter);
}

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

Overview of Messenger Objects

- A Messenger provides a reference to a Handler that others can use to send messages to it
- This class allows message passing IPC across processes via the Command Processor pattern
- Client Activity creates a Messenger pointing to a Handler in one process & handing that Messenger to another process
- If you need your service to communicate with remote processes, then you can use a Messenger to provide the interface for your service
- This technique allows you to perform IPC without the need to use AIDL
- You can use Messengers with both Bound & Started Services

http://developer.android.com/reference/android/os/Messenger.html

Using a Messenger Object

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

```
private void sendPath (String key,
   Messenger messenger) {
    Message msg = Message.obtain();
    msg.arg1 = result;
    Bundle bundle = new Bundle();
    bundle.putString(RESULT_KEY, key);
    msg.setData(bundle);
    messenger.send(msg);
   …
private void getKeyMessenger (Intent intent) {
    String key = getKey();
    sendPath(key, (Messenger)
          intent.getExtras().get("MESSENGER"));
}
```

- Client Activity receives Message via its Handler event looper

```
Handler handler = new Handler() {
public void handleMessage
       (Message msg) {
            Bundle data = msg.getData();
            String key = data.getString
                   (IDService.RESULT_KEY);
            …
           output.setText(key);
        }
};
```

Overview of Pending Intents

- A description of an Intent & target action to perform with it
- Instances of this class are created with getActivity(), getBroadcast(), getService(), & createPendingResult()
- The returned object can be handed to other applications so that they can perform the action you described on your behalf at a later time
- By giving a PendingIntent to another application, you are granting it the right to perform the operation you have specified as if the other application was yourself
- A PendingIntent itself is simply a reference to a token maintained by the system describing the original data used to retrieve it

http://developer.android.com/reference/android/app/PendingIntent.html

Using a Pending Intents

- Service replies to Activity via PendingIntent's send() method

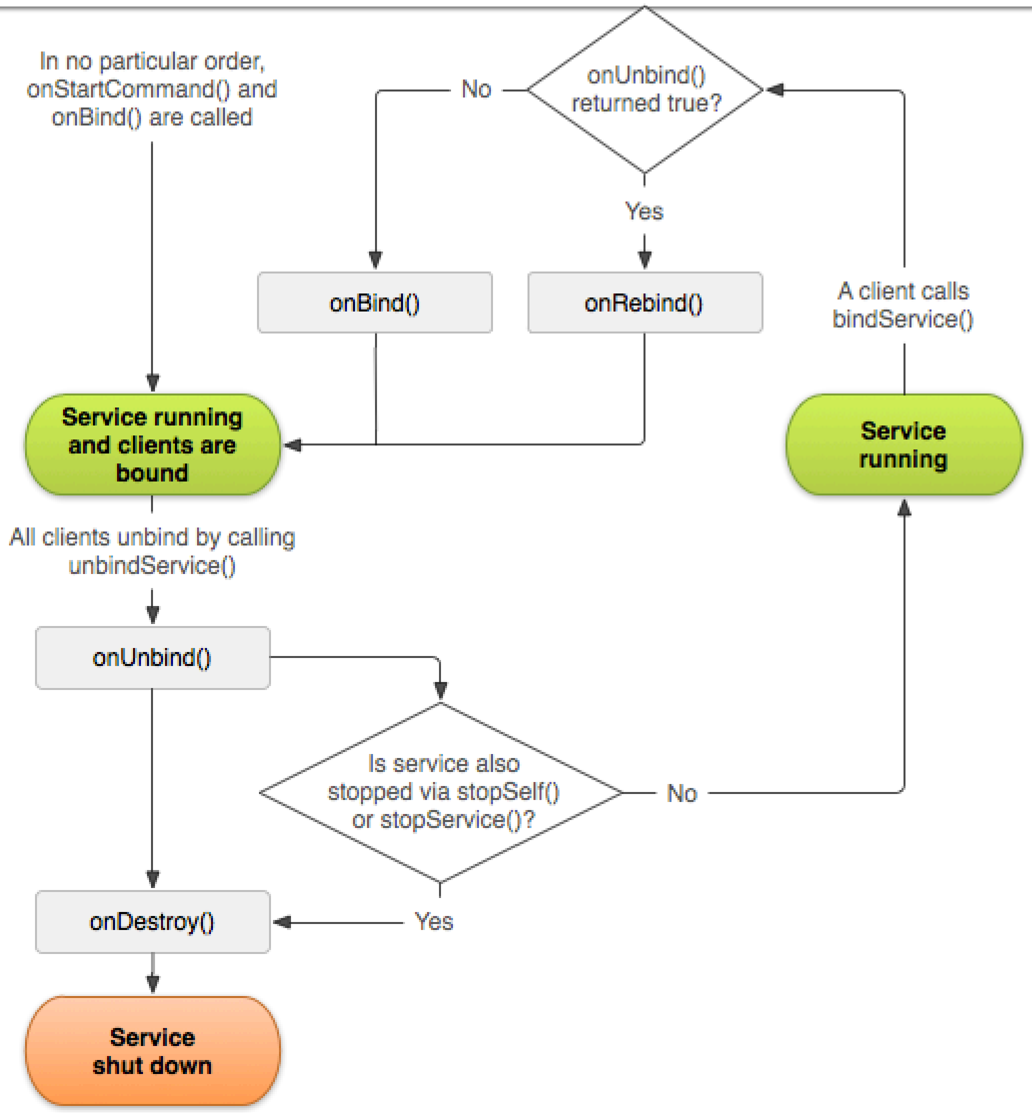
```
private void getKeyPendingIntent
  (Intent intent) {
    PendingIntent pendingIntent = (PendingIntent)
        intent.getExtras().get
         ("PENDING_INTENT");
    Intent replyIntent = new Intent();
    replyIntent.putExtra(RESULT_KEY,
                                 getKey());
    try {
      pendingIntent.send(this, KEY_ID, replyIntent);
    } catch (PendingIntent.CanceledException e1) {
      ...
    }
}
```

- Client Activity receives the reply intent via its getActivityResult() hook method

```
protected void onActivityResult
                 (int requestCode,
                  int resultCode,
                  Intent data) {
    if (requestCode ==
              IDService.KEY_ID) {
        String key =
           data.getStringExtra
              (IDService.RESULT_KEY);
      output.setText(key);
    }
}
```

Overview of Bound Services

- A Bound Service is the server in a client/server interface
- A Bound Service allows components (such as Activities) to
  - interact with the service
  - send requests
  - get results &
  - converse across processes via IPC
- A Bound Service typically lives only while it serves other application component(s)


In no particular order, onStartCommand() and onBind() are called
No
onUnbind() returned true?
Yes
onBind()
onRebind()
A client calls bindService()
Service running and clients are bound
Service running
All clients unbind by calling unbindService()
onUnbind()
Is service also stopped via stopSelf() or stopService()?
No
Yes
onDestroy()
Service shut down

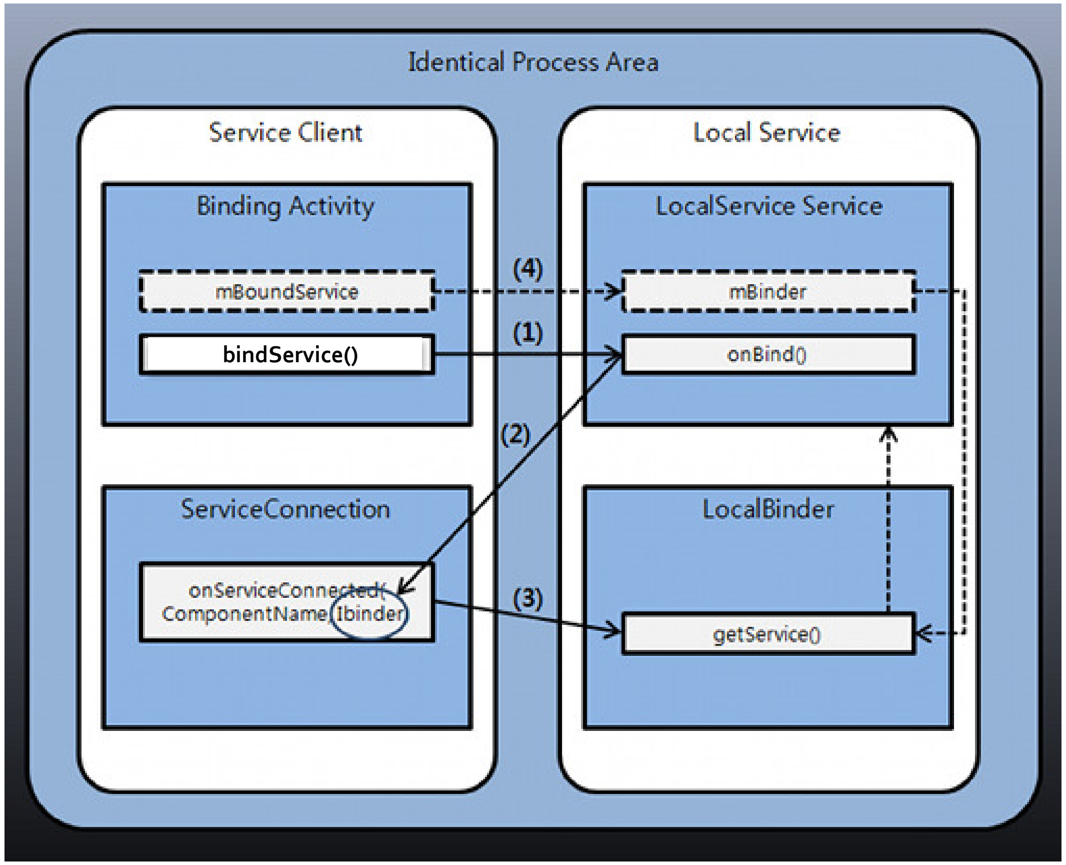
Programming Bound Services

- A Bound Service extends the Service class
- The Bound Service must implement the onBind() hook method
  - This method returns an Ibinder that defines a programming interface clients can use to interact with the service
- A bound service runs only as long as another application component is bound to it
  - Multiple components can bind to service at once, but the service is destroyed when all of them unbind
- Client components can bind to a Service when they want to interact with it by calling Context.bindService ()
- The client must provide an implementation of a ServiceConnection
  - This object monitors the connection with the Service
- The Service will be started if necessary

http://developer.android.com/guide/components/bound-services.html

Bound Service Interactions

- Application components (clients) calls bindService() to bind to a service
- The Android system then calls the service's onBind() method, which returns an IBinder for interacting with the service
  - The binding is asynchronous—bindService() returns immediately & does not return the IBinder to the client
- To receive the IBinder, the client must create an instance of ServiceConnection & pass it to bindService()
  - The ServiceConnection includes a callback method that the system calls to deliver the IBinder

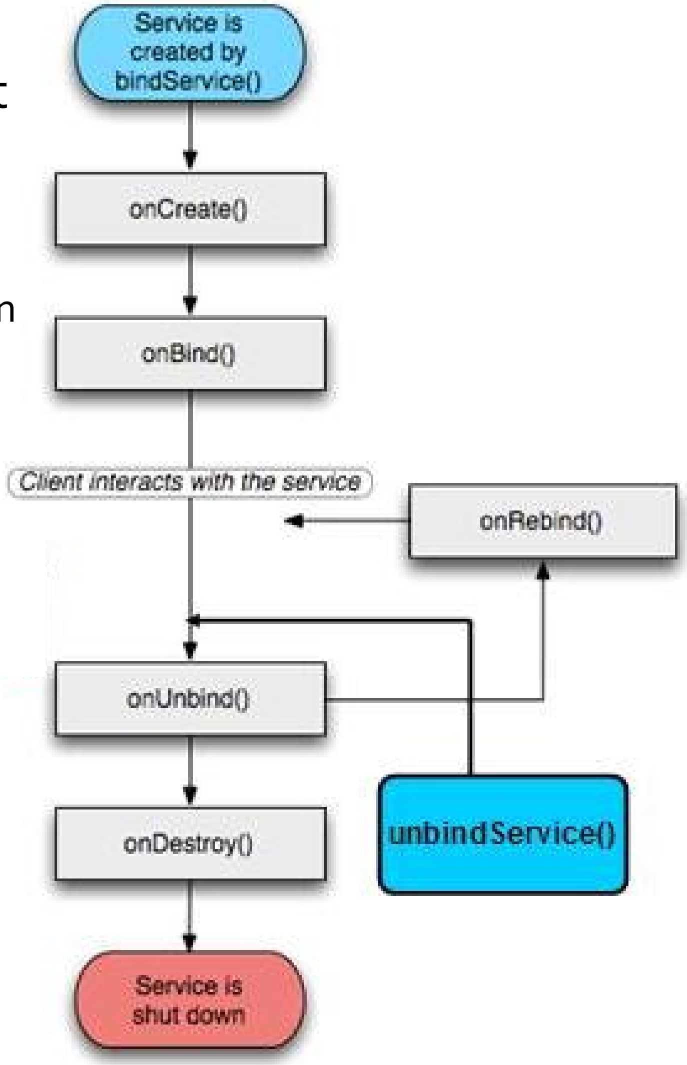


Bound Service Interactions (cont'd)

- To bind to a service from your client, you must:
  1. Implement ServiceConnection & must override two callback methods:
     - **onServiceConnected()** – Android calls this to deliver the IBinder returned by the service's onBind() method
     - **onServiceDisconnected()** – Android calls this when the connection to the service is unexpectedly lost, such as when the service has crashed or has been killed (not called with client calls unbindService())
  2. Call **bindService()**, passing the ServiceConnection implementation
  3. When the system calls your **onServiceConnected()** callback method, you can begin making calls to the service, using the methods defined by the interface
  4. To disconnect from the service, call **unbindService()**
- When your client is destroyed, it will unbind from the service
  - Always unbind when you're done interacting with the service or when your activity pauses so that the service can shutdown while its not being used

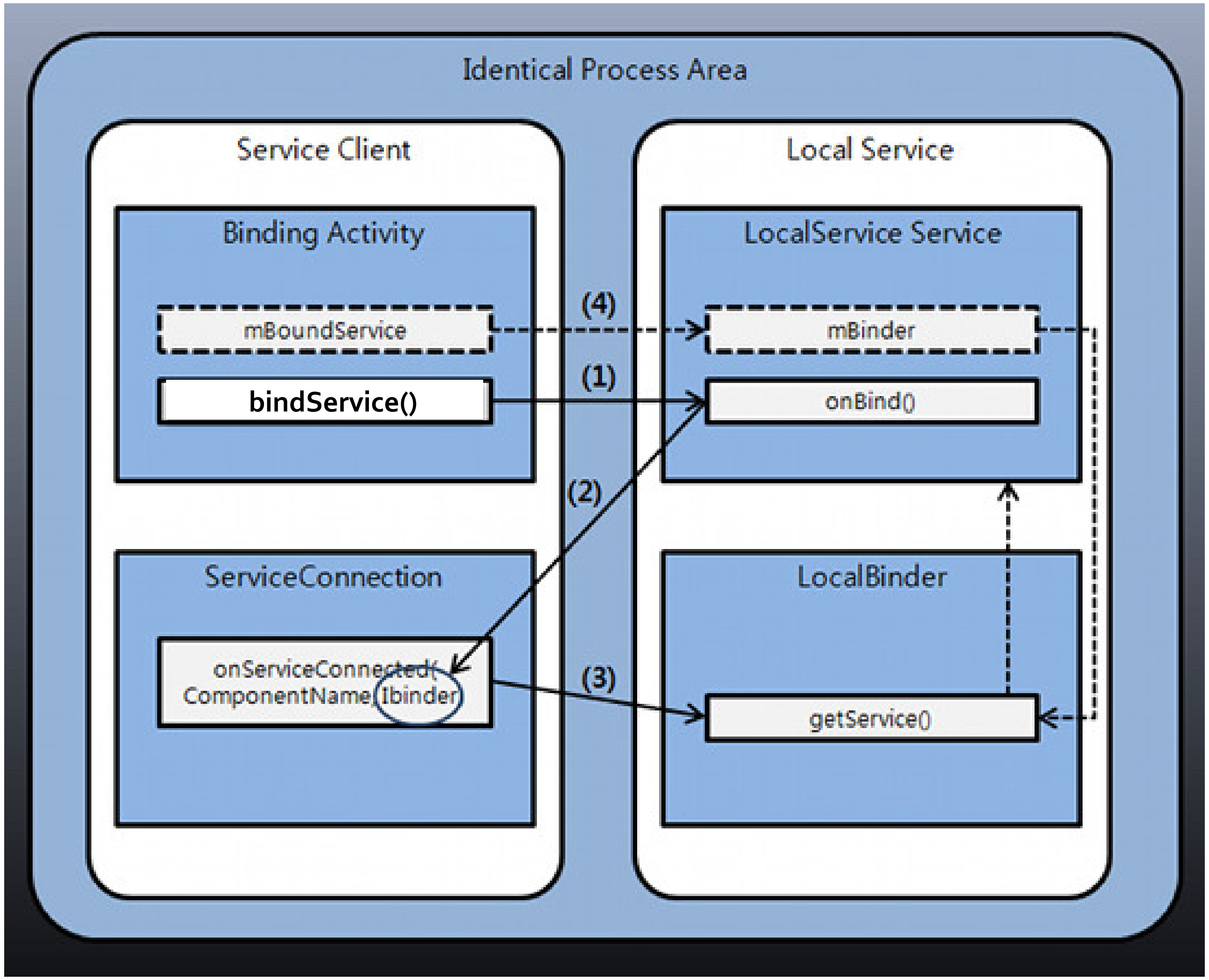
Communicating w/Bound Services

- When creating a Bound Service, you must provide an IBinder via an interface clients can use to interact with the Service via the following
  - **Extending the Binder class**
    - If your service runs in the same process as the client you can extend the Binder class & return an instance from onBind()
  - **Using a Messenger**
    - Create an interface for the service with a Messenger that allows the client to send commands to the service across processes using Message objects
    - Doesn't require thread-safe components
  - **Using Android Interface Definition Language (AIDL)**
    - AIDL performs all the work to decompose objects into primitives that the operating system can understand & marshal them across processes to perform IPC
    - Does require thread-safe components



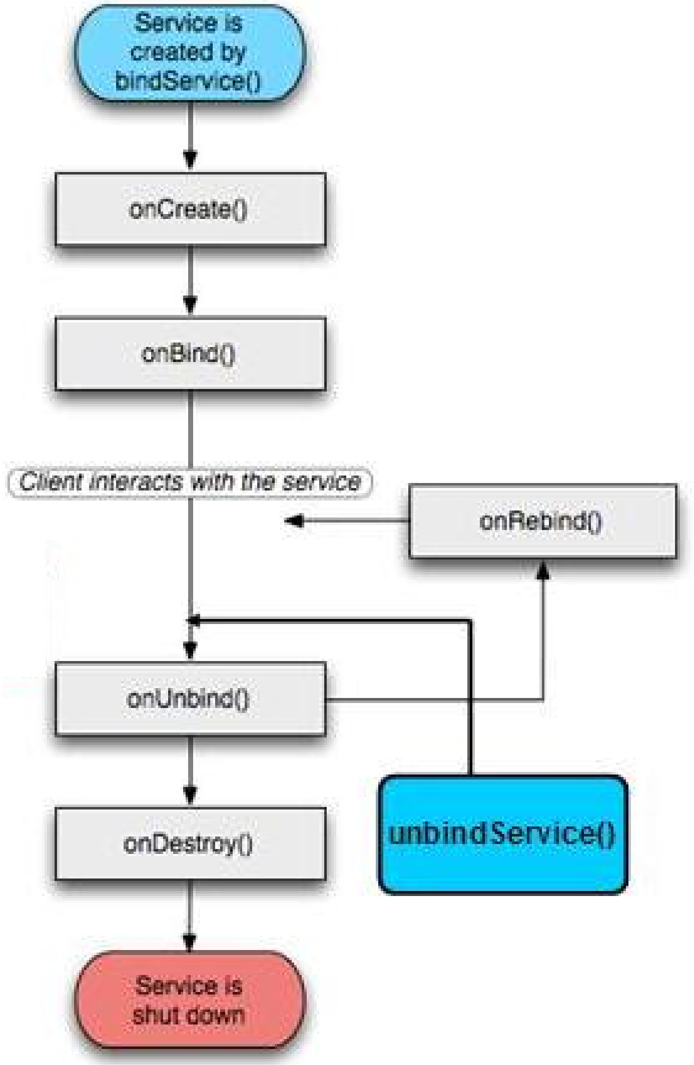
Extending the Binder Class

- Sometimes a service is used only by the local application & need not work across processes
- In this case, you can implement your own Binder subclass that provides your client direct access to public methods in a service



How to Extend the Binder Class

- Here's how to use locally Bound Service:
  - In your service, create a Binder object that either:
    - contains public methods the client can call
    - returns the current Service instance, which has public methods the client can call, or
    - returns an instance of another class hosted by the service with public methods the client can call
  - Return this instance of Binder from the onBind() callback method
  - In the client, receive the Binder from the onServiceConnected() callback method & make calls to the Bound Service using the provided methods



Example of Extending Binder

```
public class LocalService extends Service {
  // Binder given to clients
  private final IBinder mBinder = new
              LocalBinder();
  // Random number generator
  private final Random mGenerator =
                              new Random();

 // Class used for the client Binder
 public class LocalBinder extends Binder {
    LocalService getService() {
      // Return instance so clients can
      // call public methods
      return LocalService.this;
    }
 }

  // Hook method called by Android
  // when client calls bind_service()
 public IBinder onBind(Intent intent) {
     return mBinder;
  }

  // method for clients
  public int getRand() {
     return mGenerator.nextInt(100);
  }
}
```

Example of Extending Binder (cont'd)

```
public class BindingActivity extends
                                        Activity {
   // Instance of the service returned
   // from onBind()
   LocalService mService;

   // Keep track of whether we're bound
   boolean mBound = false;

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

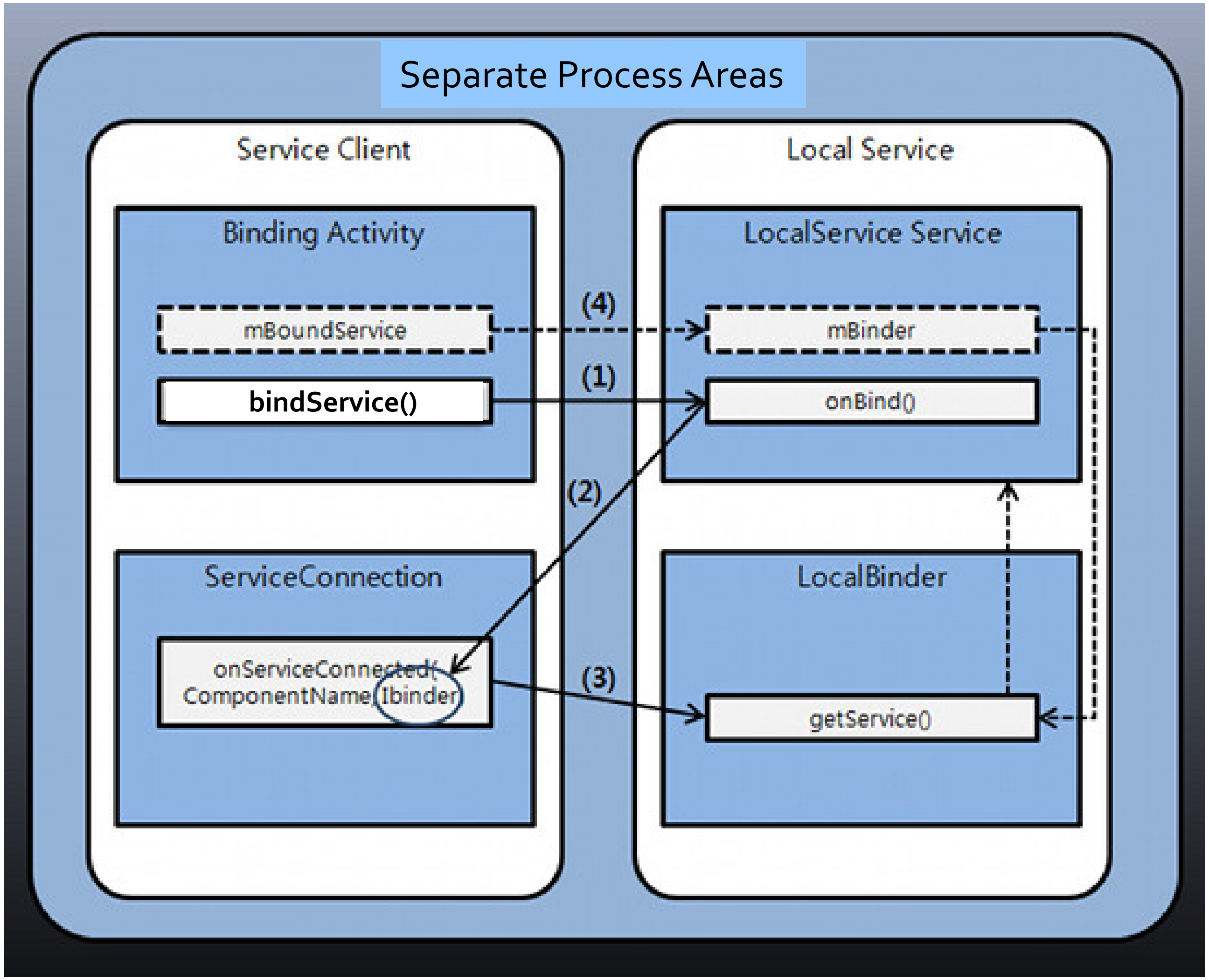
```
protected void onStart() {
    super.onStart();
    Intent intent =
         new Intent(this,
              LocalService.class);
    // Bind to LocalService
    bindService(intent, mConnection,
         Context.BIND_AUTO_CREATE);
}

protected void onStop() {
    super.onStop();
    if (mBound) {
      unbindService(mConnection);
      mBound = false;
    }
 }
```

Extending the Binder Class (cont't)

```
    public void onButtonClick(View v) {
        if (mBound) {
          Toast.makeText(this, mService.getRand(), Toast.LENGTH_SHORT).show();
        }
     }

    // Defines callbacks for service binding, passed to bindService()
    private ServiceConnection mConnection = new ServiceConnection() {
        public void onServiceConnected(ComponentName className, IBinder service) {
            // We're bound to LocalService, cast the IBinder & get LocalService instance
            LocalBinder binder = (LocalBinder) service;
           mService = binder.getService();
           mBound = true;
        }
        public void onServiceDisconnected(ComponentName argo) { mBound = false; }
    };
}
```

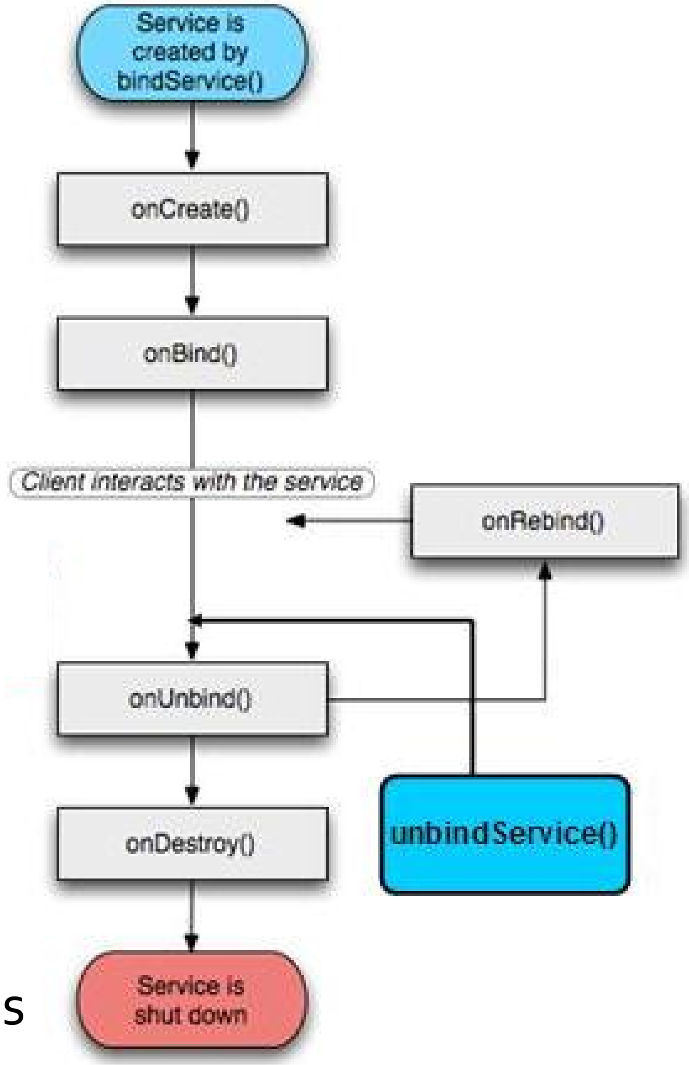
Using a Messenger

- You can use a Messenger to communicate with a Bound Service in another process
- This technique allows you to perform IPC between Activities & Services without the need to use AIDL (which is more complicated)



How to Use a Messenger

- Here's how to use a Messenger:
  - The service implements a Handler that receives a callback for each call from a client
  - The Handler is used to create a Messenger object (which is a reference to the Handler)
  - The Messenger creates an IBinder that the service returns to clients from onBind()
  - Clients use the IBinder to instantiate the Messenger (that references the service's Handler), which the client uses to send Message objects to the service
  - The service receives each Message in its Handler—specifically, in the handleMessage() method
- Two-way messaging is a slight variation on this



Example of Using Messenger

```
public class MessengerService extends
                                   Service {
  // Command to service to display msg
  static final int MSG_SAY_HELLO = 1;

  // Handler of incoming client msgs
  class IncomeHandler extends Handler {
    public void handleMessage(Message
      msg) {
      switch (msg.what) {
        case MSG_SAY_HELLO:
          displayMsg(msg); break;
        default:
          super.handleMessage(msg);
      }
    }
  }
  // Target we publish for clients to
  // send messages to IncomeHandler
  final Messenger mMessenger =
      new Messenger
          (new IncomeHandler());

  // When binding to the service, we
  // return an interface to our messenger
  // for sending messages to the service
  public IBinder onBind(Intent intent) {
    return mMessenger.getBinder();
  }
}
```

Example of Using Messenger (cont'd)

```
public class ActivityMessenger extends Activity {
  // Messenger for communicating with the service.
  Messenger mService = null;
  // Flag indicating whether we have called bind on the service
  boolean mBound;

  private ServiceConnection mConnection = new ServiceConnection() {
    public void onServiceConnected(ComponentName className, IBinder service) {
      // This is called when the connection with the service has been established,
      // giving us the object we can use to interact with the service.
      mService = new Messenger(service); mBound = true;
    }
    public void onServiceDisconnected(ComponentName className) {
      // Called when the connection with the service is unexpectedly disconnected
      mService = null;  mBound = false;
    }
  };
```

Example of Using Messenger (cont'd)

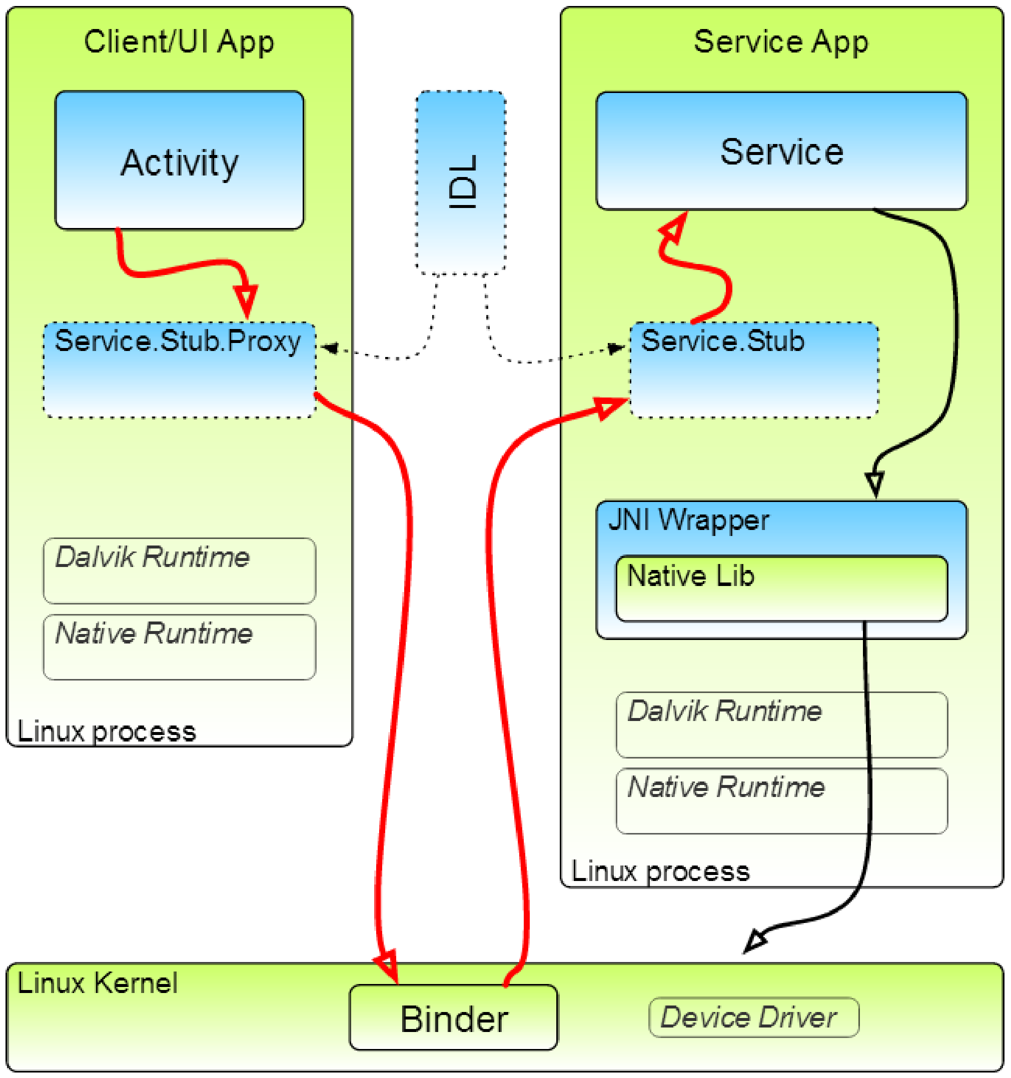
```
public void onButtonClick(View v) {
   if (!mBound) return;
   // Create & send a message to service,
   // using a supported 'what' value
   Message msg = Message.obtain(null,
         MessengerService.MSG_SAY_HELLO,
         0, 0);
   try { mService.send(msg); } catch
     (RemoteException e) {
       e.printStackTrace();
   }
}
protected void onCreate
   (Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.main);
}
```

```
protected void onStart() {
    super.onStart();
    // Bind to the service
    bindService(new Intent
        (this, MessengerService.class),
         mConnection,
         Context.BIND_AUTO_CREATE);
}
protected void onStop() {
    super.onStop();
    // Unbind from the service
    if (mBound) {
       unbindService(mConnection);
       mBound = false;
    }
}
}
```

Using AIDL

- AIDL uses the Binder RPC mechanism to implement the Broker pattern
- There are two main steps:
  - Define a remote interface via AIDL
    - The AIDL compiler generates tedious (de)marshaling code via its stub methods
  - You implement Service- & Client-specific methods

AIDL is the most powerful & complex solution



Overview of AIDL

- AIDL is a Android-specific language for defining Binder-based service interfaces
- AIDL follows Java-like interface syntax & allows application developers to declare their "business" logic methods
- Each Binder-based service is defined in a separate .aidl file & saved in a src directory
- The Android aidl build tool extracts a real Java interface (along with a Stub providing Android's android.os.IBinder) from each .aidl file & places it into a gen directory
  - Eclipse ADT automatically calls aidl for each .aidl file it finds in a src directory

```
package
course.examples.Services.
                    KeyCommon;

interface KeyGenerator {
   String getKey();
}
```

http://developer.android.com/guide/components/aidl.html

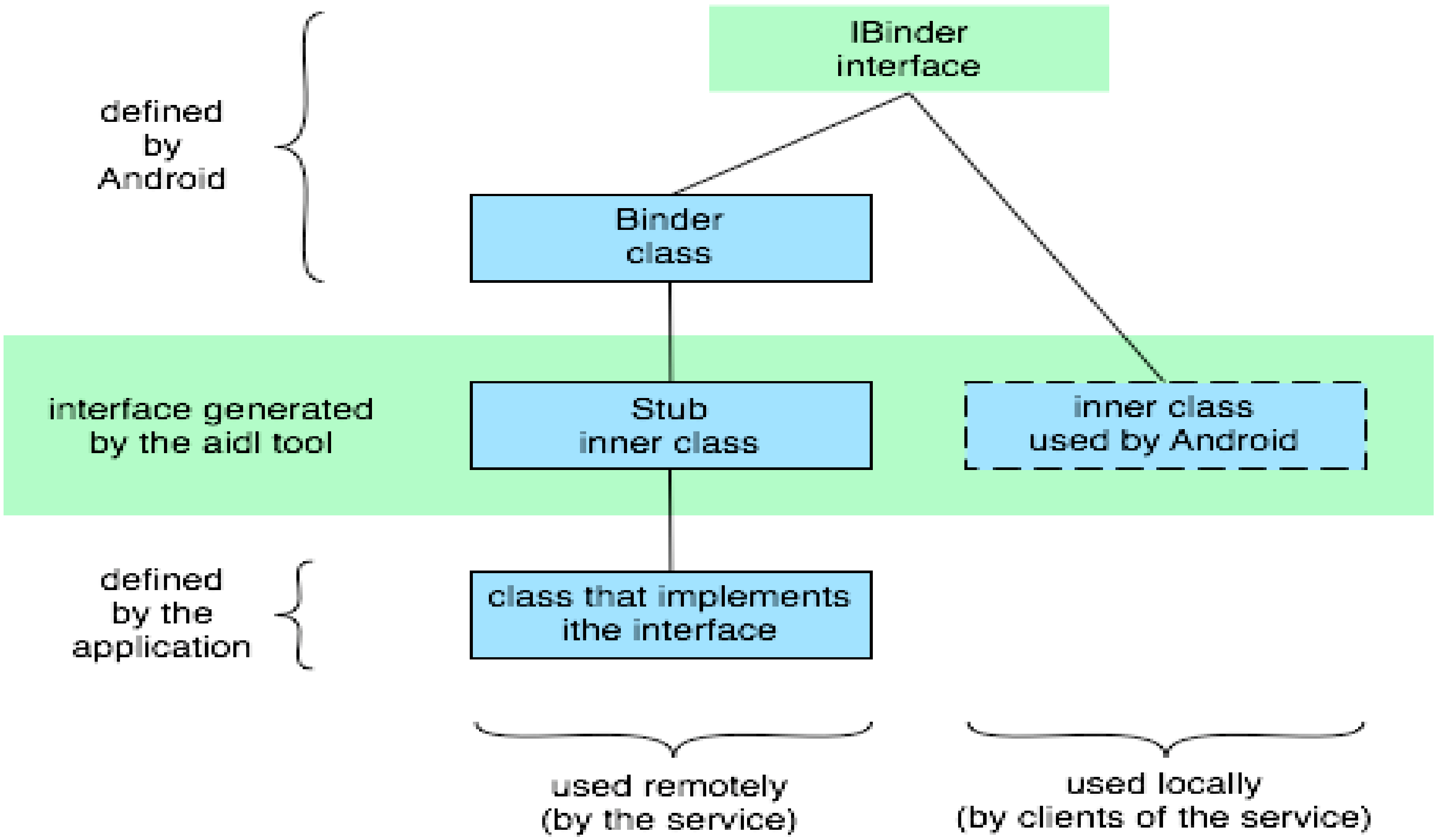
AIDL Syntax & Data Types

- Similar to Java interface syntax
  - Can declare methods but not
  - static fields
- Remote method parameters can be labeled
  - in: (default) xfer'd to remote method
  - out: returned to the caller
  - inout: both in & out

- Java primitive types
  - boolean, boolean[], byte, byte[], char[], int, int[], long, long[], float, float[], double, double[]
  - java.lang.CharSequence, java.lang.String
- java.util.List (uses java.util.ArrayList internally)
  - List elements must be valid AIDL data types
  - Generic lists supported
- Java.util.Map (uses java.util.HashMap internally)
  - Map elements must be valid AIDL data types
  - Generic maps not supported
- Other AIDL-generated interfaces
- Classes implementing the Parcelable protocol

AIDL interface methods can ***not*** throw any exceptions

Structure of AIDL-based Solutions


IBinder interface
defined by Android
Binder class
interface generated by the aidl tool
Stub inner class
inner class used by Android
defined by the application
class that implements ithe interface
used remotely (by the service)
used locally (by clients of the service)

http://developer.android.com/reference/android/os/Binder.html

Implementing an AIDL Interface

- Given an auto-generated AIDL server stub, you need to implement the service
  - You can either do this directly in the stub or by routing the stub implementation to other methods you already wrote

- The mechanics of implementing a service are straightforward:
  1. Create a private instance of the AIDL-generated .Stub class (e.g., IScript.Stub)
  2. Implement methods matching up with each of the methods you placed in the AIDL
  3. Return this private instance from your onBind() method in the Service subclass

```
public class KeyGeneratorImpl extends Service {
    private final KeyGenerator.Stub binder =
                                  new KeyGenerator.Stub()
    {
      public String getKey()  {
           // generate unique ID in a thread-safe
           // manner & return it
      }
    };

   public IBinder onBind(Intent intent) {
     return this.binder;
   }
}
```

Accessing an AIDL Interface

- To use an AIDL-defined service, you first need to create an instance of your own ServiceConnection class
- Your ServiceConnection subclass needs to implement two methods:
  - onServiceConnected(), which is called once your activity is bound to the Service to obtain a proxy to the Binder implementation
  - onServiceDisconnected(), which is called if your connection ends abnormally, such as the Service crashing

```
public class KeyUser extends Activity {
 private KeyGenerator service;
 private boolean bound;
 private ServiceConnection connection =
        new ServiceConnection() {
   public void onServiceConnected(
                    ComponentName className,
                    IBinder iservice) {
      service =
        KeyGenerator.Stub.asInterface(iservice);
      bound = true;
   }
...
output.setText(service.getKey());
```

AIDL Call Semantics

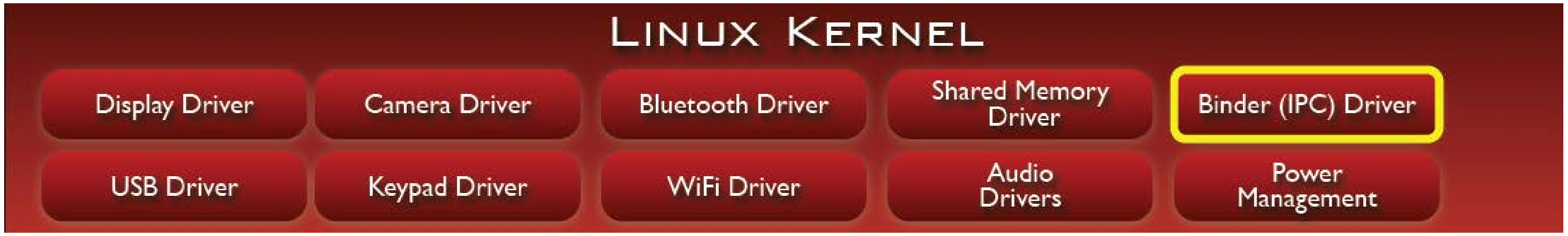
- Calls made from the local process are executed in the same thread that is making the call
  - If this is your main UI thread, that thread continues to execute in the AIDL interface
  - If it is another thread, that is the one that executes your code in the service
- Calls from a remote process are dispatched from a thread pool the platform maintains inside of your own process
  - an implementation of an AIDL interface must therefore be completely thread-safe
- The oneway keyword modifies the behavior of remote calls
  - When used, a remote call does not block; it simply sends the transaction data & returns immediately
  - If oneway is used with a local call, there is no impact & the call is still synchronous

http://developer.android.com/guide/components/aidl.html

Overview of Binder RPC

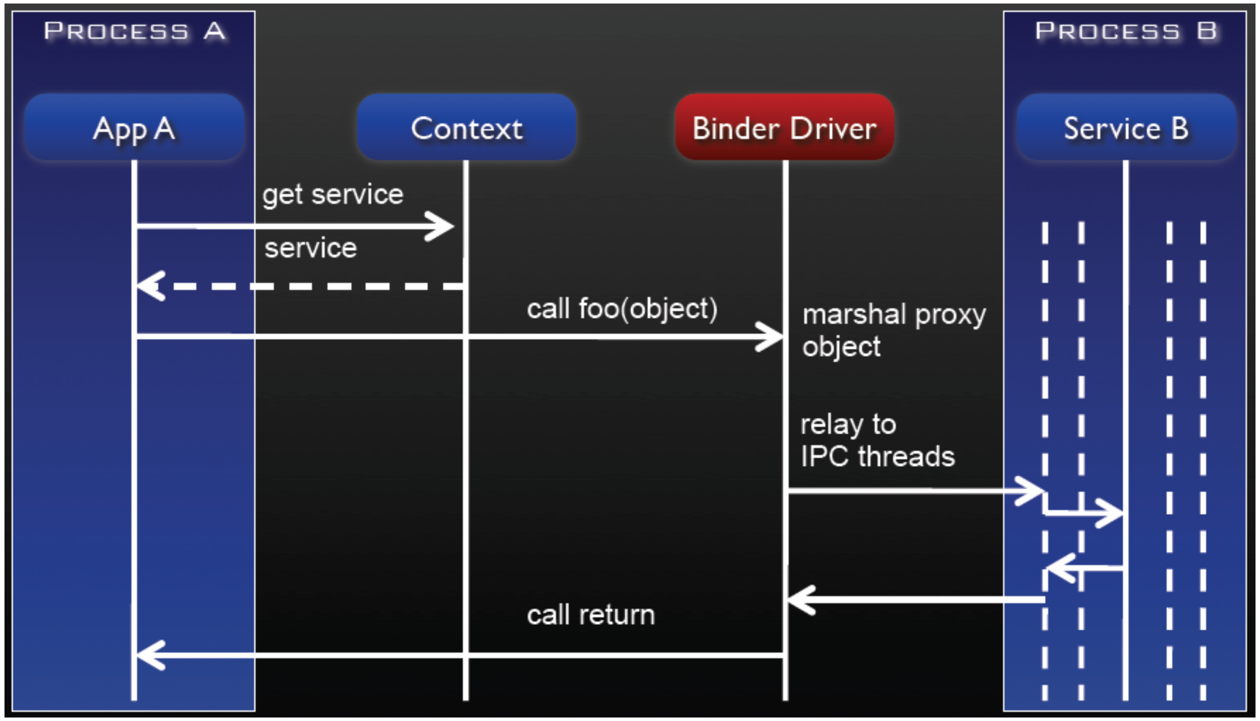
- Binder provides a high performance RPC mechanism for in-process & cross-process calls
- Binder-capable services are described in AIDL, which is similar to other IDL languages
- Since Binder is provided as a Linux driver, the services can be written in both C/C++ as well as Java
- Most Android services written in Java

- The Binder Driver is installed in the Linux kernel to accelerate IPC
- High performance through shared memory
- Per-process thread pool for processing requests
- Reference counting & mapping of object references across processes
- Supports synchronous calls between processes
  - Asynchrony is supported via callbacks



https://sites.google.com/site/io/anatomy--physiology-of-an-android

AIDL & Binder RPC

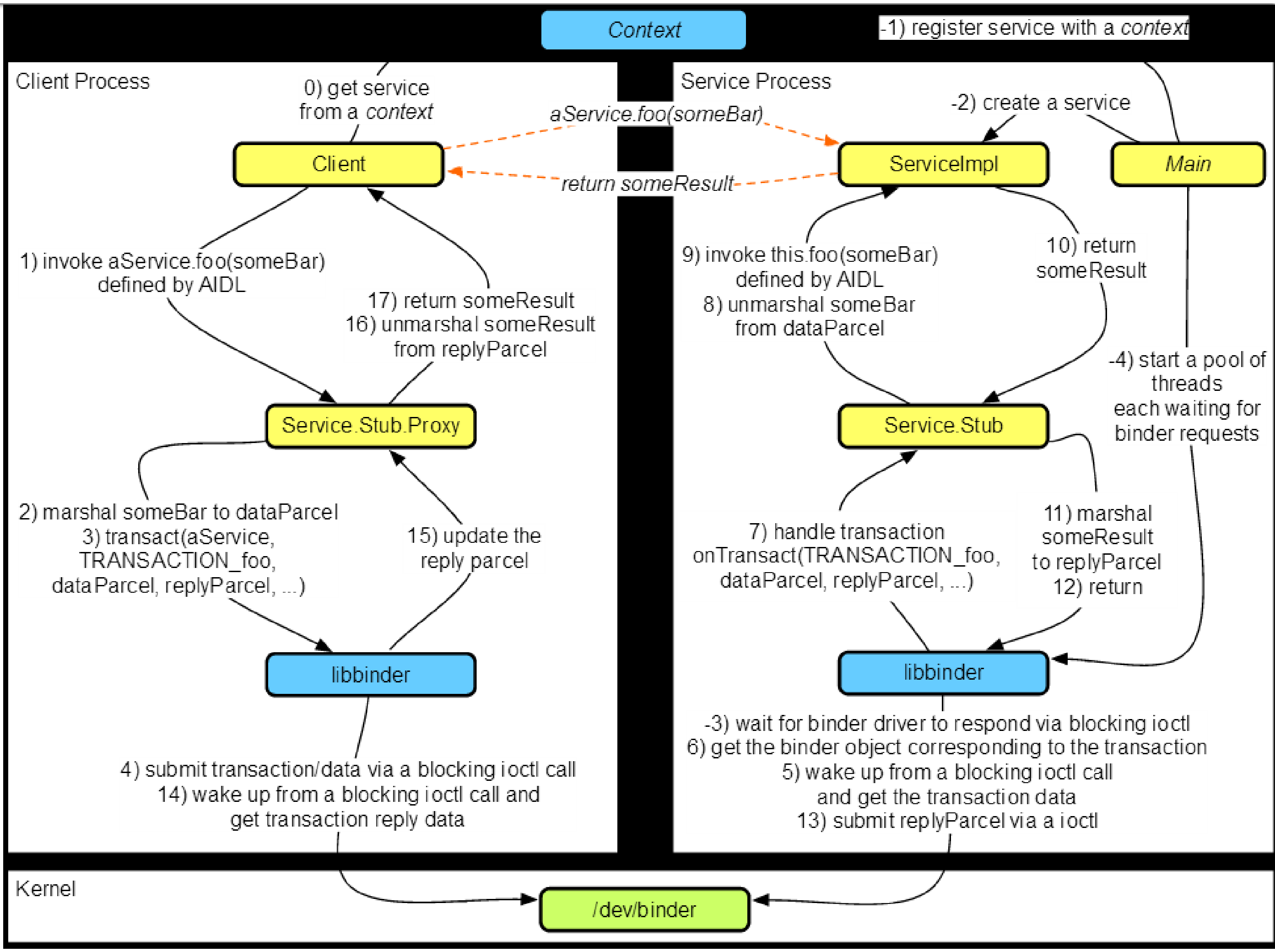


- To cross process boundaries, AIDL uses an RPC transport mechanism that handles passing of data from one process (caller) to another (callee)
- Caller's data is marshaled into tokens that Binder RPC understands, copied to callee's process, & finally demarshaled into what callee expects
- Callee's response is also marshaled by Binder RPC, copied to caller's process where it is demarshaled into what caller expects
- (de)marshaling is automatically provided by the Binder RPC mechanism

https://sites.google.com/site/io/anatomy--physiology-of-an-android

AIDL & Binder RPC (cont'd)

- All caller calls go thru the transact() method, which automatically marshals arguments & return values via Parcels
  - Parcel is a generic data structure that maintains meta-data about contents
- Caller calls to transact() are by default synchronous
  - i.e., provide same semantics as a local method call

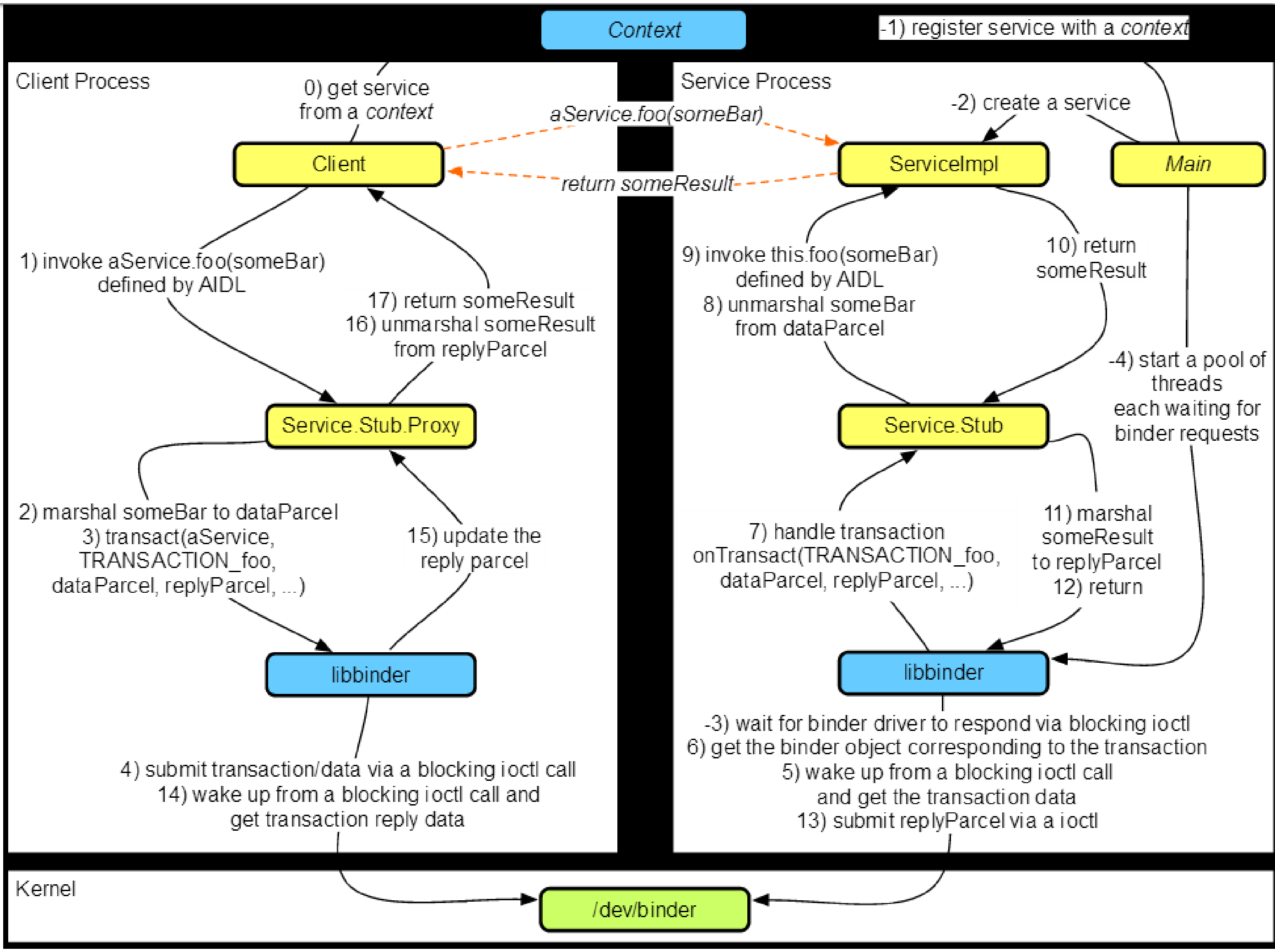


Context
-1) register service with a context
Client Process
0) get service from a context
aService.foo(someBar)
Service Process
-2) create a service
Client
ServiceImpl
Main
return someResult
1) invoke aService.foo(someBar) defined by AIDL
17) return someResult
16) unmarshal someResult from replyParcel
9) invoke this.foo(someBar) defined by AIDL
8) unmarshal someBar from dataParcel
10) return someResult
-4) start a pool of threads each waiting for binder requests
Service.Stub.Proxy
Service.Stub
2) marshal someBar to dataParcel
3) transact(aService, TRANSACTION_foo, dataParcel, replyParcel, ...)
15) update the reply parcel
7) handle transaction onTransact(TRANSACTION_foo, dataParcel, replyParcel, ...)
11) marshal someResult to replyParcel
12) return
libbinder
libbinder
4) submit transaction/data via a blocking ioctl call
14) wake up from a blocking ioctl call and get transaction reply data
-3) wait for binder driver to respond via blocking ioctl
6) get the binder object corresponding to the transaction
5) wake up from a blocking ioctl call and get the transaction data
13) submit replyParcel via a ioctl
Kernel
/dev/binder

http://marakana.com/static/courseware/android/internals/index.html

AIDL & Binder RPC (cont'd)

- On callee side, the Binder maintains a pool of threads to handle incoming RPC requests
  - unless call is local, in which case same thread is used
- Callee methods can be oneway, in which case caller calls return immediately
- Callee's mutable state must be thread-safe to handle concurrent requests from multiple callers

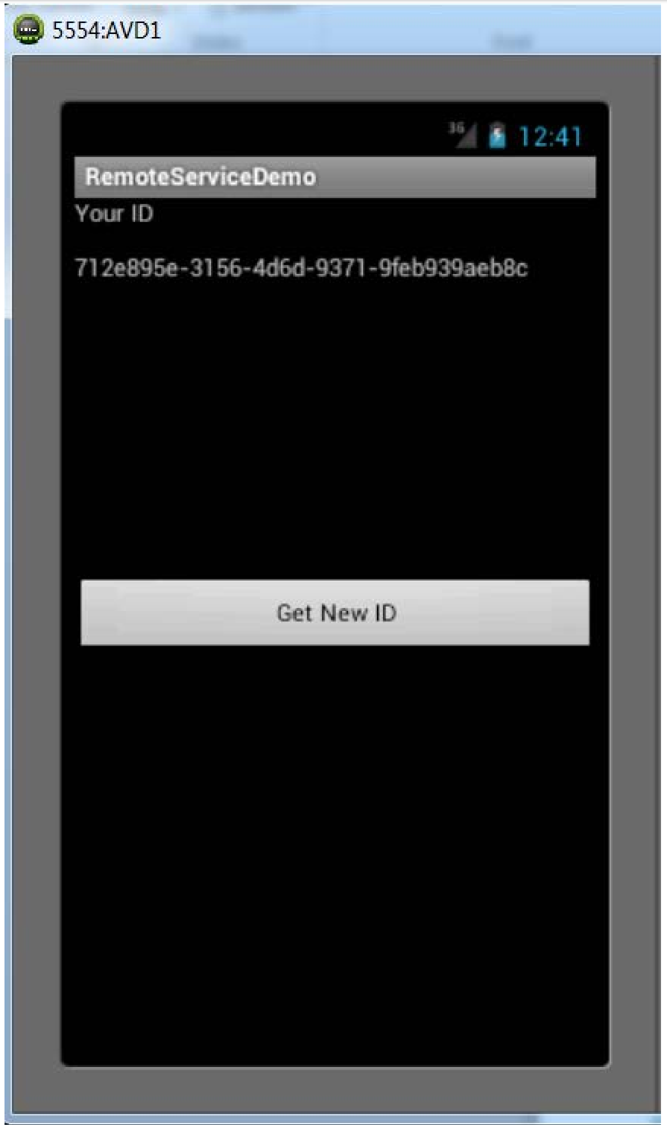


Context
-1) register service with a context
Client Process
0) get service from a context
aService.foo(someBar)
Service Process
-2) create a service
Client
return someResult
ServiceImpl
Main
1) invoke aService.foo(someBar) defined by AIDL
17) return someResult
16) unmarshal someResult from replyParcel
9) invoke this.foo(someBar) defined by AIDL
8) unmarshal someBar from dataParcel
10) return someResult
-4) start a pool of threads each waiting for binder requests
Service.Stub.Proxy
Service.Stub
2) marshal someBar to dataParcel
3) transact(aService, TRANSACTION_foo, dataParcel, replyParcel, ...)
15) update the reply parcel
7) handle transaction onTransact(TRANSACTION_foo, dataParcel, replyParcel, ...)
11) marshal someResult to replyParcel
12) return
libbinder
libbinder
4) submit transaction/data via a blocking ioctl call
14) wake up from a blocking ioctl call and get transaction reply data
-3) wait for binder driver to respond via blocking ioctl
6) get the binder object corresponding to the transaction
5) wake up from a blocking ioctl call and get the transaction data
13) submit replyParcel via a ioctl
Kernel
/dev/binder

http://marakana.com/static/courseware/android/internals/index.html

ID Service via Synchronous AIDL

- Client uses a "Bound Service" hosted in another application
- Client needs an ID from service
- Requires IPC via the Android Interface Definition Language (AIDL)
  - AIDL uses the Android Binder RPC mechanism
- Overall structure of this solution is similar to the Messenger solution presented earlier
  - Main difference is that the AIDL calls are synchronous, whereas the Messenger calls are asynchronous



Your ID

712e895e-3156-4d6d-9371-9feb939aeb8c

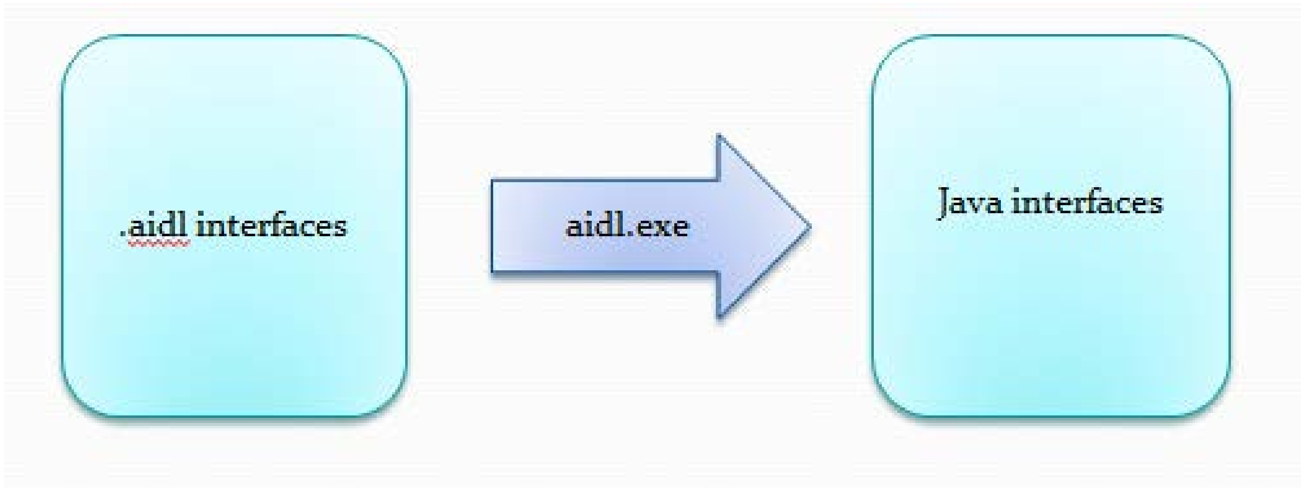
Define Remote Interface

- Declare interface in an .aidl file

```
package course.examples.Services.KeyCommon;

interface KeyGenerator {
   String getKey();
}
```

Compile .aidl File

- Generate a Java interface with same name as .aidl file
  - Eclipse does this automatically
- Generated interface contains:
  - Abstract inner class called Stub
  - Interface & helper methods



Implement Remote Methods

```
public class KeyGeneratorImpl extends Service {
  …
  private final KeyGenerator.Stub binder =
                                    new KeyGenerator.Stub() {
    public String getKey() {
      // generate unique ID **in a thread-safe manner** & return it
    }
  };
…
```

Implement Service Methods

```
...
  public IBinder onBind(Intent intent) {
    return this.binder;
  }
}
```

Implement the ID Client

```
public class KeyUser extends Activity {
 private KeyGenerator service; // handle to Remote Service
 private boolean bound;
// Remote Service callback methods
 private ServiceConnection connection =
                                                    new ServiceConnection() {
   public void onServiceConnected(
                        ComponentName className, IBinder iservice) {
      service = KeyGenerator.Stub.asInterface(iservice);
      bound = true;
   }
…
```

Implement the ID Client (cont.)

```
…
public void onServiceDisconnected(
                                        ComponentName className) {
      service = null; bound = false;
   }
 };
…
```

Implement the ID Client (cont.)

```
protected void onStart() {
    super.onStart();
    Intent intent = new Intent(KeyGenerator.class.getName());
    // bind to Service
    bindService(intent,this.connection,
                    Context.BIND_AUTO_CREATE);
 }
 protected void onStop() {
   // unbind from Service
   if (bound) unbindService(this.connection);
   super.onStop();
 }
}
```

Implement the ID Client (cont.)

```
…
public void onCreate(Bundle icicle) {
    …
    goButton.setOnClickListener(new OnClickListener() {
      public void onClick(View v) {
        try {
          // call remote method
          output.setText(service.getKey());
        } catch (RemoteException e) {}
      }
    });
…
}
```

Client AndroidManifest.xml

```
<manifest … package="course.examples.Services.KeyClient">
   <application …">
      <activity android:name=".KeyUser" …>
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name=
                                     "android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>
   </application>
</manifest>
```

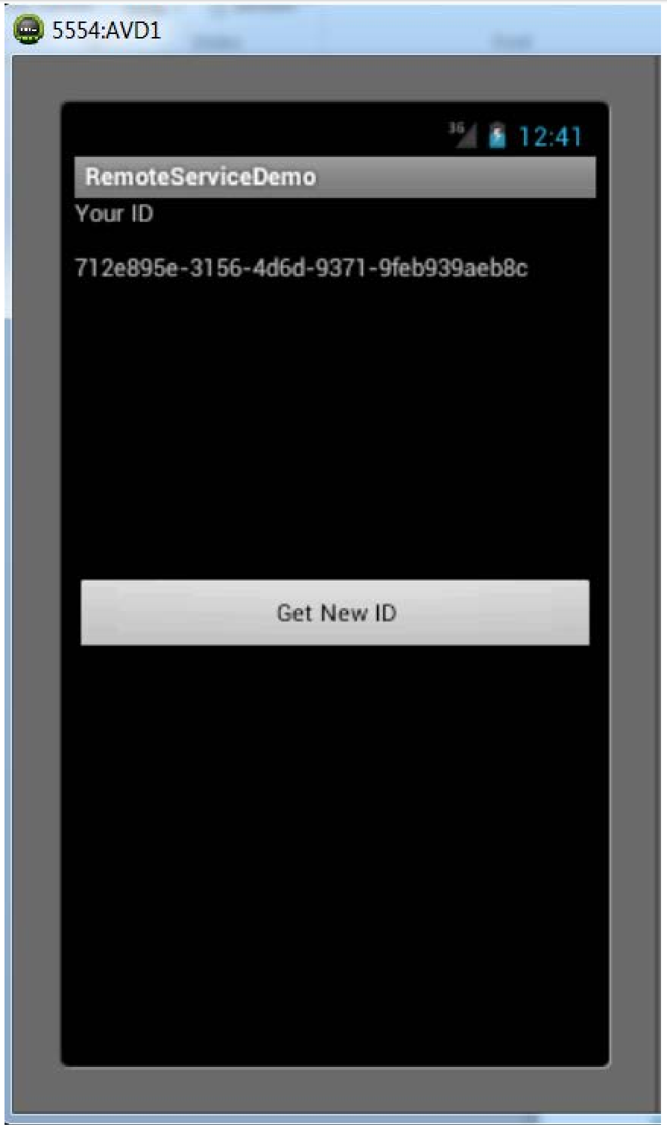
Service AndroidManifest.xml

```
<manifest ...package="course.examples.Services.KeyService">
   <application ...">
     <service android:name=".KeyGeneratorImpl"
                                           android:exported="true">
        <intent-filter>
          <action android:name=
             "course.examples.Services.KeyCommon.KeyGenerator"/>
        </intent-filter>
      </service>
   </application>
</manifest>
```

- The Service AndroidManifest.xml file must be registered with Android before the client tries to run

ID Service via Asynchronous AIDL

- Client uses a "Bound Service" hosted in another application
- Client needs an ID from service
- Requires IPC via the Android Interface Definition Language (AIDL)
  - AIDL uses the Android Binder RPC mechanism
- Overall structure of this solution is similar to the AIDL solution presented earlier
  - Main difference is that the AIDL calls in this example are oneway asynchronous, whereas the previous AIDL example calls are twoway synchronous



Your ID

712e895e-3156-4d6d-9371-9feb939aeb8c

Define Remote Interfaces

- Declare interface in two .aidl files
  - You can't define more than one interface in each *.aidl file

KeyGeneratorCallback.idl

```
package course.examples.Services.
KeyCommon;

interface KeyGeneratorCallback {
  oneway void sendKey
                        (in String key);
}
```

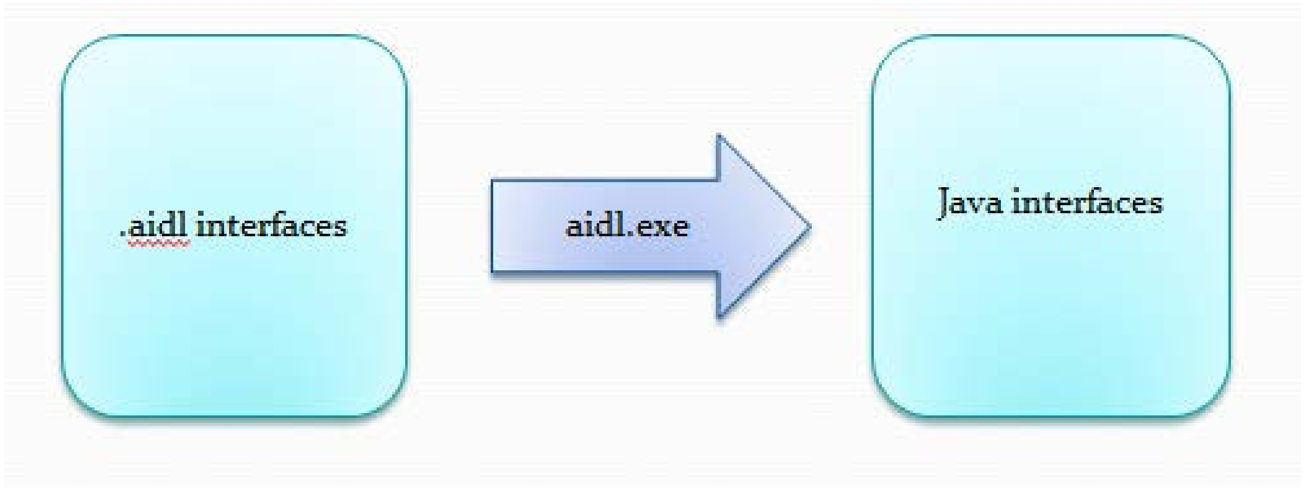
KeyGenerator.idl

```
package course.examples.Services.
KeyCommon;

import course.examples.Services.
KeyCommon.KeyGeneratorCallback;

interface KeyGenerator {
   oneway void setCallback
        (in KeyGeneratorCallback  callback);
}
```

- When oneway is used on a remote call it does not block; it simply sends the transaction data & immediately returns

Compile .aidl File

- Generate Java interfaces with same name as .aidl files
  - Eclipse does this automatically
- Generated interfaces contain:
  - Abstract inner class called Stub
  - Interface & helper methods



Implement Remote Methods

```
public class KeyGeneratorImpl extends Service {
  ...
  private final KeyGenerator.Stub binder =
                                                  new KeyGenerator.Stub() {
    public void setCallback(final keyGeneratorCallback callback) {
      // generate unique ID in a thread-safe manner & return it
      // by invoking callback.sendKey(key)
    }
  };
...
```

- The setCallback() method runs in a thread from the Android Binder framework's thread pool
- Since setCallback() is a oneway it doesn't block the remote caller

Implement Service Methods

```
…
  public IBinder onBind(Intent intent) {
    return this.binder;
  }
}
```

Implement the ID Client

```
public class KeyUser extends Activity {
  private KeyGenerator service; // handle to Remote Service
  private boolean bound;
 // Remote Service callback methods
  private ServiceConnection connection =
                                        new ServiceConnection() {
    public void onServiceConnected(
                    ComponentName className, IBinder iservice) {
      service = KeyGenerator.Stub.asInterface(iservice);
      bound = true;
    }
...
```

Implement the ID Client (cont.)

```
...
public void onServiceDisconnected(
                                        ComponentName className) {
      service = null; bound = false;
   }
 };
...
```

- Note that this connection object enables both AIDL calls from the client to the service & from the service back to the client

Implement the ID Client (cont.)

```
protected void onStart() {
    super.onStart();
    Intent intent = new Intent(KeyGenerator.class.getName());
    // bind to Service
    bindService(intent,this.connection,
                    Context.BIND_AUTO_CREATE);
 }
 protected void onStop() {
   // unbind from Service
   if (bound) unbindService(this.connection);
   super.onStop();
 }
}
```

Implement the ID Client (cont.)

```
...
private final KeyGeneratorCallback.Stub callback = new
    KeyGeneratorCallback.Stub() {
    public void sendKey (final String key) {
      runOnUiThread (new Runnable() {
        public void run() {
           output.setText(key); // Output the key to the user's display
        }
      });
    }
...
```

- sendKey() runs in a thread from the Android Binder framework's thread pool, so it needs to queue the output to run in main thread
- Since sendKey() is oneway it doesn't block the remote caller

Implement the ID Client (cont.)

```
…
public void onCreate(Bundle icicle) {
    …
    goButton.setOnClickListener(new OnClickListener() {
      public void onClick(View v) {
        try {
          // call remote method to register callback
          if (bound) service.setCallback(callback);
        } catch (RemoteException e) {}
      }
    });
…
}
```

ID Service AndroidManifest.xml

```
<manifest ... package="course.examples.Services.KeyDemo">
   <application ...">
      <activity android:name=".KeyUser" ...>
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name=
                                 "android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>
      <service android:name=".KeyGeneratorImp"
        android:exported= "true" android:process=":remote" >
      </service>
  </application>
</manifest>
```

Comparing AIDL with Messengers

- AIDL implements the Broker pattern, whereas Messengers implements the Command Processor pattern
- When you need to perform IPC, using a Messenger for your interface is simpler than implementing it with AIDL
  - The Messenger queues all calls to the service, which handles them *asynchronously*
  - In contrast, a pure AIDL interface sends simultaneous requests to the service *synchronously*, which must then handle multi-threading
- If you need asynchronous behavior for AIDL interfaces you'll need to implement callback objects, as shown in the last example
- For many practical apps the Service doesn't need to perform multi-threading
  - Using a Messenger allows the service to handle one call at a time
  - If it's important that your service be multi-threaded, then you should use AIDL to define your interface

Source Code Examples

- LoggingServiceExample
- MusicPlayingServiceExample
- ServiceWithIPCExampleClient
- ServiceWithIPCExampleService