Android Services

Lecture 4

Operating Systems Practical

26 October 2016

Outline

- An Android Service is an application component without a user interface
- Designed for long-running operations in the background
- Can run even if the user is not in the hosting application
- Can be accessed by external applications directly
  - If exported by the hosting application

- By default, runs in the main UI thread of the hosting application
    - CPU intensive work and blocking operations done on a separate thread
    - A service can be configured to run in a separate process

Declaring a Service

- `<service>` tag in the AndroidManifest, under the `application` tag
- `android:name` - The class implementing the service
- `android:enabled` - Set as true or false if the system can / cannot instantiate the service
  - Default value is "true"

Declaring a Service

- `android:exported` - Whether or not other applications can access the service
  - Without intent filter - default is "false"
  - With intent filter - default is "true"
- `android:isolatedProcess` - Set to true if the service is to run in its own separate process
  - Has it own set of permissions
- `android:permission` - Permission that must be given to a component that wants to interact with the service

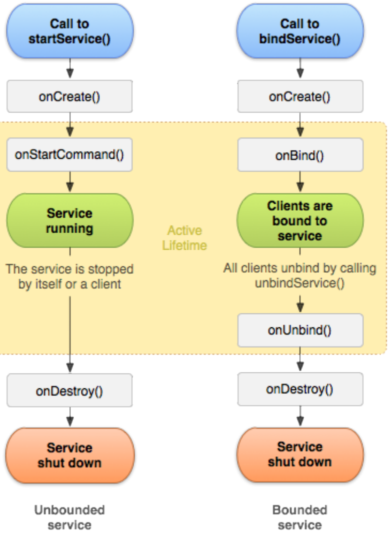
Types of Services

- Started Service
  - Performs a single operation
  - Does not return the result to the caller directly
  - Launched by an application component that calls `Context.startService()`
  - Once started, it can run indefinitely, even if the caller has terminated

Types of Services

- Bound Service
  - Can perform multiple operations
  - Offers a client-server interface, allowing interactions with it (send requests, obtain results)
  - Communication can be across processes (IPC)
  - Launched by an application component that calls `Context.bindService()`
  - Remains active as long as there is at least one component is still bound to it (has not called `Context.unbindService()`)

Service Life Cycle



Source: http://developer.android.com

Outline

Started Services

- Launched by calling `Context.startService(Intent)`
  - The `Intent` should contain relevant information for the service to do it's work
  - Information can be added to the `Intent` within its `Extras`
- After the task has been completed, the service will be kept in the running state
- A started service can be stopped in two ways:
  - By another application component which calls `Context.stopService(Intent)`
  - It can stop itself by calling `Service.stopSelf()`

Implementing a Started Service

- Extending the base `Service` class
  - Implement the `onStartCommand(Intent, flags, startId)` method
  - Need to create (and maintain) a separate thread for intensive / blocking operations within the service
  - Useful when a service needs to be both started and bound
  - START_STICKY or START_NOT_STICKY

Implementing a Started Service

- Extending the `IntentService` class
  - Uses a worker thread to handle start requests, one at a time
  - Useful when multiple requests do not need to be handled simultaneously
  - Implement the `onHandleIntent(Intent)` method and do the work without worrying about creating a new thread

Example

```
public class HelloService extends Service {
  @Override
  public int onStartCommand(Intent intent, int flags, int startId) {
      Toast.makeText(this, "service starting", Toast.LENGTH_SHORT).show();
      return START_STICKY;
  }

  @Override
  public IBinder onBind(Intent intent) {
      return null;
  }

  @Override
  public void onDestroy() {
    Toast.makeText(this, "service done", Toast.LENGTH_SHORT).show();
  }
}
```

```
Intent intent = new Intent(this, HelloService.class);
startService(intent);
```

Example

```java
public class HelloIntentService extends IntentService {
  public HelloIntentService() {
      super("HelloIntentService");
  }

  @Override
  protected void onHandleIntent(Intent intent) {
      // Normally we would do some work here, like download a file.
  }
}
```

Outline

Bound Services

- Launched by calling `Context.bindService(Intent)`
  - If another component calls `bindService()` after the service has been launched, the same service instance is given (the service is not restarted)
- Client-server paradigm
  - The server is the running service
  - The client is the application component (e.g. the Activity) bound to the service
  - The communication interface is specified by an `IBinder`
- Can receive requests from external processes / applications

Implementing a Bound Service

- Extend the `Service` class
- Implement the `onBind()` method
  - `onBind()` returns an `IBinder` object
  - The method is called only for the first component binding to the service
  - Subsequent components that bind to the service will receive the same `IBinder` object

Communicating with a Bound Service

- If the client is running in the same process
  - Extend the `Binder` class and return an instance
- For communicating with external processes you can:
  - Use a Messenger (that serializes incoming requests) and call `Messenger.getBinder()`
  - Use AIDL (especially when you need to handle multiple requests simultaneously)

Connecting to a Bound Service

- Implement the `ServiceConnection` interface
  - `onServiceConnected()` callback gives the `IBinder` used to call remote methods
  - `onServiceDisconnected()` callback gets called when the connection to the service has died
- Call `bindService()` and give it an instance of your `ServiceConnection` implementation
  - `bindService()` returns immediately
  - The framework will call `onServiceConnected()` when connection to the service has been established

Disconnecting to a Bound Service

- Call `unbindService()` to end service connection
  - If the current component unbinding is the only one who had been still bound, the service should be destroyed
  - The service is kept alive only if it is also a Started Service (another component has called `startService()` on it)

Extending the Binder class

- In the `Service` class, create a member variable of a class extending `Binder` that defines communication with the service in either of the following manners:
  - The `Binder` instance has public methods that can be called from the outside
  - It can return a reference to the Service class, which itself has public methods
  - It can return a reference to another class, hosted within the service, which has public methods
- From the `Service`'s `onBind()` method return the member variable

Using a reference to the Service

```
public class LocalService extends Service {
    private final IBinder mBinder = new LocalBinder();
    private final Random mGenerator = new Random();

    public class LocalBinder extends Binder {
        LocalService getService() {
            return LocalService.this;
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }

    public int getRandomNumber() {
      return mGenerator.nextInt(100);
    }
}
```

Using a reference to the Service

```
public class BindingActivity extends Activity {
    LocalService mService;
    boolean mBound = false;
    @Override
    protected void onStart() {
        super.onStart();
        Intent intent = new Intent(this, LocalService.class);
        bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
    }
    @Override
    protected void onStop() {
        super.onStop();
        if (mBound) {
            unbindService(mConnection);
            mBound = false;
        }
    }
    private ServiceConnection mConnection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName className, IBinder service) {
            LocalBinder binder = (LocalBinder) service;
            mService = binder.getService();
            mBound = true;
        }
        @Override
        public void onServiceDisconnected(ComponentName arg0) {
            mBound = false;
        }
    };
}
```

Outline

Communicating through a Messenger

- In the `Service` class, create a member variable of a class extending `Handler`
  - Implement the `handleMessage(Message)` method
  - Communication with the service is done by how different `Message` types are handled
- Create a `Messenger` member variable passing it's constructor an instance of your `Handler` class

Communicating through a Messenger

- In the `onBind()` method return `Messenger.getBinder()`
- The client's `ServiceConnection` instance creates a `Messenger` object based on the `IBinder` object passed as a parameter to the `onServiceConnected()` method

Example

```
public class MessengerService extends Service {
    static final int MSG_SAY_HELLO = 1;

    class IncomingHandler extends Handler {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case MSG_SAY_HELLO:
                    Toast.makeText(getApplicationContext(), "hello!",
                                        Toast.LENGTH_SHORT).show();
                    break;
                default:
                    super.handleMessage(msg);
            }
        }
    }

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

    @Override
    public IBinder onBind(Intent intent) {
        return mMessenger.getBinder();
    }
}
```

Example

```
public class ActivityMessenger extends Activity {
    Messenger mService = null;
    boolean mBound;

    private ServiceConnection mConnection = new ServiceConnection() {
        public void onServiceConnected(ComponentName className, IBinder service) {
            mService = new Messenger(service);
            mBound = true;
        }

        public void onServiceDisconnected(ComponentName className) {
            mService = null;
            mBound = false;
        }
    };

    public void sayHello(View v) {
        if (!mBound) return;
        Message msg = Message.obtain(null, MessengerService.MSG_SAY_HELLO, 0, 0);
        try {
            mService.send(msg);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

}
```

Communicating through a Messenger

- The `handleMessage()` method returns `void`
  - The service has no readily-available means to respond to the client
- To have two-way communication you need to implement a similar `Messenger` mechanism in the client
  - Set the client's `Messenger` as the `replyTo` parameter of the `Message`
  - The service receives a reference to the client's `Messenger` that can be used to send it's responses

Outline

Interface Definition Languages

- Specification languages used to describe a software component's interface
- Are commonly used in Remote Procedure Calls (RPC)
- An external entity (usually called a broker) is responsible for enabling communication between components exposing their respective interfaces

Interface Definition Languages

- Examples of IDLs include:
  - AIDL - Android IDL
  - OMG IDL (Object Management Group IDL) - implemented in CORBA for RPC services
  - Protocol Buffers - Google's method of serializing structured data
  - WSDL - Web Services Description Language

The Need for AIDL

- Android provides security through sandboxing
  - An app's process cannot normally access the memory of another app's process
- For two processes to communicate they need to be able to decompose objects into primitives that can be marshalled across the system
  - The `Binder` system handles these operations
- Writing the code to marshall / unmarshall objects and call the framework's `Binder` services is considered tedious
  - The system does this automatically when using AIDL

Creating an AIDL File

- Within the hosting app's `src/` folder, create a `.aidl` file
- In the file, declare a single Java interface containing only method signatures
- AIDL allows using the following data types as return values and method parameters:
  - Primitive Java types (`int`, `float`, `boolean`, etc.)
  - `String`
  - `CharSequence`
  - `List` (the system will use `ArrayList`)
  - `Map` (the system will use `HashMap`)
- All `Collections` can only have elements from the other supported data types

Implementing the AIDL interface

- Building the application will generate a `YourInterface.java` file within the project's `gen/` folder
- The generated interface also contains a `YourInterface.Stub` subclass which contains all methods declared in the `.aidl` file
- Within your Service, instantiate the `YourInterface.Stub` and implement its methods
- Return the `Stub` from the Service's `onBind()` method

Connecting to a Service using AIDL

- Make sure that the application from which `bindService()` will be called has a copy of the `.aidl` file in the `src/` folder
- Create a `ServiceConnection` instance within the component from which binding to the service will be performed
- Within the `onServiceConnected()` method use the `IBinder` parameter to get a reference to the AIDL interface by calling `YourInterface.Stub.asInterface(IBinder)`
- It is recommended to guard calls to service methods in a `try{...} catch` block
  - `DeadObjectException` should be caught - occurs when the connection has broken

Example

```
public class RemoteService extends Service {
    @Override
    public void onCreate() {
        super.onCreate();
    }

    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }

    private final IRemoteService.Stub mBinder = new IRemoteService.Stub() {
        public int getPid(){
            return Process.myPid();
        }
        public void basicTypes(int anInt, long aLong, boolean aBoolean,
            float aFloat, double aDouble, String aString) {
        }
    };
}
```

```
IRemoteService mIRemoteService;
private ServiceConnection mConnection = new ServiceConnection() {
    public void onServiceConnected(ComponentName className, IBinder service) {
        mIRemoteService = IRemoteService.Stub.asInterface(service);
    }

    public void onServiceDisconnected(ComponentName className) {
        mIRemoteService = null;
    }
};
```

Sending Custom Objects over IPC

- Using custom classes in the context of IPC can be done if we implement the `Parcelable` interface
- The method to be implemented is `writeToParcel()`
- The class must contain a `public static final Parcelable.Creator<YourClass>` member variable named `CREATOR`
  - Implement `createFromParcel()` and `newArray()` interface methods
- Create a `YourClass.aidl` file in which you declare the class as parcelable
  - Besides the package declaration, the `.aidl` file should only contain a `parcelable YourClass;` line

Example

```
public class MyParcelable implements Parcelable {
     private int mData;

     public void writeToParcel(Parcel out, int flags) {
         out.writeInt(mData);
     }

     public static final Parcelable.Creator<MyParcelable> CREATOR
             = new Parcelable.Creator<MyParcelable>() {
         public MyParcelable createFromParcel(Parcel in) {
             return new MyParcelable(in);
         }

         public MyParcelable[] newArray(int size) {
             return new MyParcelable[size];
         }
     };

     private MyParcelable(Parcel in) {
         mData = in.readInt();
     }
 }
```

Outline

Foreground Services

- A foreground service is a service that the user is aware of in some manner (E.g. - a music app playing music even when the user is within another app)
- Due to it considered as relevant to the user, it will not be killed as fast by the system in low-memory situations
- A foreground service needs to present an *on-going* notification (that cannot be dismissed) while it is running

Foreground Services

- Started by calling `startForeground(notificationId, Notification)`
  - Called from within the service itself
  - Specifying which component (Activity) to start is done in the creation of the `Notification`
- Stopped by calling `stopForeground()`

Example

```
Notification notification = new Notification(R.drawable.icon,
                    getText(R.string.ticker_text), System.currentTimeMillis());

Intent notificationIntent = new Intent(this, ExampleActivity.class);
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0,
                                        notificationIntent, 0);
notification.setLatestEventInfo(this, getText(R.string.notification_title),
                    getText(R.string.notification_message), pendingIntent);

startForeground(ONGOING_NOTIFICATION_ID, notification);
```

- http://developer.android.com/guide/components/services.html
- http://developer.android.com/guide/components/bound-services.html
- http://developer.android.com/guide/components/aidl.html
- http://developer.android.com/reference/android/app/Service.html
- http://developer.android.com/reference/android/app/IntentService.html
- http://developer.android.com/reference/android/content/ServiceConnection.html
- http://developer.android.com/reference/android/os/Messenger.html
- http://developer.android.com/reference/android/os/Message.html

Keywords

- Android Services
- Started Services
- Foreground Services
- IntentService
- Bound Services
- IBinder
- ServiceConnection
- Handler
- Messenger
- Message
- AIDL
- Parcelable