Android Development

Lecture 8
Android Background Services

Lecture Summary

- Android Service
- Service Life Cycle
- Unbound Service
- Bound Service
- Intent & Intent Filter
- Broadcast Receiver

Background Tasks in Android

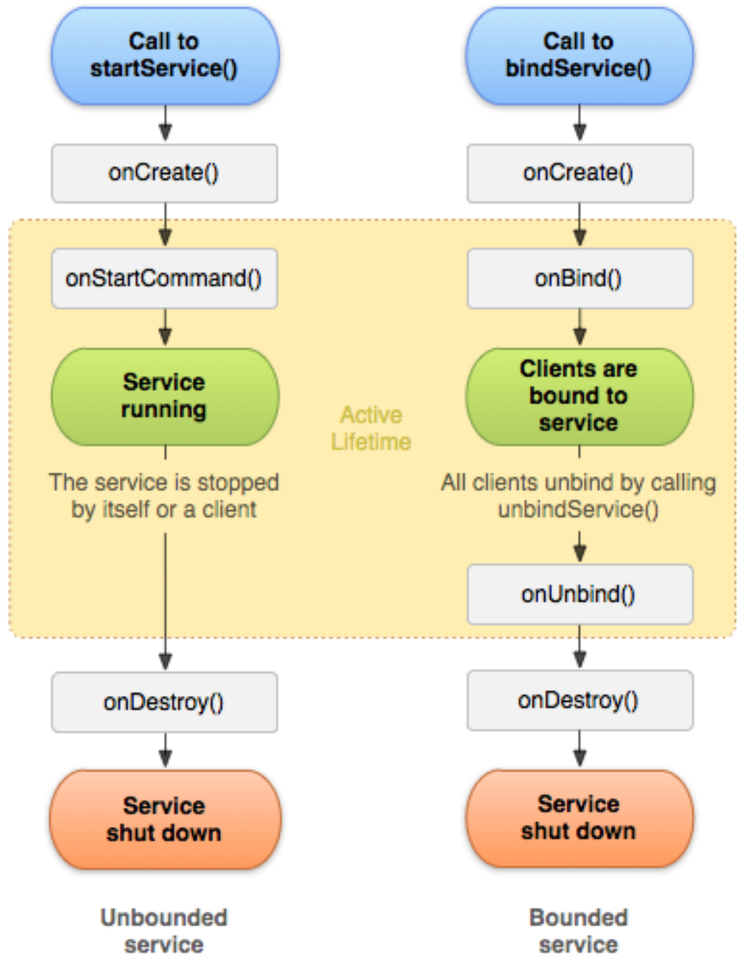
- The Android Platform supports Background Processing in 4 different ways:
  - **Threads**: Android supports the usage of the Threads class to perform asynchronous processing. Android also supplies the java.util.concurrent package to perform something in the background, e.g. using the ThreadPools and Executor classes. **Only the user interface face is allow to update the user interface. If you need to update the user interface from another Thread, you need to synchronize with this user interface Threads or you can use the "android.os.Handler" or "AsyncTasks" classes.**
  - **Handler**: The Handler class can update the user interface. A Handler provides methods for receiving instances of the Message or Runnable class.
  - **AsyncTask**: Is a special class for Android development that encapsulate background processing and facilitates the communication and updating of the application's UI.
  - **Service**: A Service allows an application to implement longer-running background operations. An application controls when its service runs by explicitly starting and stopping the service.

Android - Service

- An important application component in the Android platform is a service.
- A Service is an application component that can perform long-running operations in the background and does not provide a user interface. Another application component can start a service and it will continue to run in the background even if the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC). For example, a service might handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.
- A service can essentially take two forms:
- **Started:** A service is "started" when an application component (such as an activity) starts it by calling startService(). Once started, a service can run in the background indefinitely, even if the component that started it is destroyed. Usually, a started service performs a single operation and does not return a result to the caller. For example, it might download or upload a file over the network. When the operation is done, the service should 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, and even do so across processes with interprocess communication (IPC). A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.

Service - Life Cycle

- Like an activity, a service has lifecycle callback methods that you can implement to monitor changes in the service's state and perform work at the appropriate times.
- Life Cycle methods are:
  - onCreate()
  - onStartCommand() / onStart()
  - onBind()
  - onUnbind()
  - onRebid()
  - onDestroy()



Service - Life Cycle

```
public class ExampleService extends Service {
    @Override
    public void onCreate() {
        ...
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        ...
    }
    @Override
    public IBinder onBind(Intent intent) {
        ...
    }
    @Override
    public boolean onUnbind(Intent intent) {
        ...
    }
    @Override
    public void onRebind(Intent intent) {
        ...
    }
    @Override
    public void onDestroy() {
        ...
    }
}
```

Service - Life Cycle

- The entire lifetime of a service happens between the time onCreate() is called and the time onDestroy() returns.
- Like an activity, a service does its initial setup in onCreate() and releases all remaining resources in onDestroy(). For example, a music playback service could create the thread where the music will be played in onCreate(), then stop the thread in onDestroy().
- The onCreate() and onDestroy() methods are called for all services, whether they're created by startService() or bindService().
- The active lifetime of a service begins with a call to either onStartCommand() or onBind(). Each method is handed the Intent that was passed to either startService() or bindService(), respectively.
- If the service is started, the active lifetime ends the same time that the entire lifetime ends (the service is still active even after onStartCommand() returns). If the service is bound, the active lifetime ends when onUnbind() returns.

Creating a Service (unBounded)

- Creating a Service in Android involves extending the **Service** class and adding a service tag to the AndroidManifest. Then you must override the method onCreate(), onStart(), onStartCommand(), and onDestroy().
- If you are implement a bounded Service you should implement some additional methods that we will analyze later in this lecture.
- onStart() and onStartCommand() methods are essentially the same. The only difference is that onStart() has been deprecated in API Level 5 and above.
- If the Service is created by the system Context.startService(...) the method onCreate() is called just before the onStart() or onStartCommand().
- If the Service is bound with a call to Context.bindService() method, the onCreate() method is called just before the onBind() method. onStart() and onStartCommand() are not called in this case.

Creating a Service

Returns a null object because it is not a Bound Service

```
public class LocationTrackingService extends Service{

    @Override
    public IBinder onBind(Intent arg0) {
        return null;
    }

    @Override
    public void onCreate() {
        super.onCreate();
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
    }

    @Override
    public void onStart(Intent intent, int startId) {
        super.onStart(intent, startId);
        startServiceTask();
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        startServiceTask();
        return super.onStartCommand(intent, flags, startId);
    }

    @Override
    public boolean onUnbind(Intent intent) {
        return super.onUnbind(intent);
    }
...

}
```

Both methods are implemented to support old Android versions.

startServiceTask() is an example to show how to implement a common method to start your Service activities.

The implementation of these method define and start the background activities.

Service and Android Manifest

- The system does not know about a Service unless it has been defined in the AndroidManifest using the <service> tag.
- In the XML block you can specify the Service name and that the service is enabled.
- If you want you can additionally add an Intent filter to start, control and send specific command to the Service.
- Intent Filter are used to inform the system which implicit intents a Service/Activity (multiple IntentFilter are allowed). Each filter describes a capability of the component, a set of intents that the component is willing to receive. It, in effect, filters in intents of a desired type, while filtering out unwanted intents — but only unwanted implicit intents (those that don't name a target class).An explicit intent is always delivered to its target, no matter what it contains; the filter is not consulted. But an implicit intent is delivered to a component only if it can pass through one of the component's filters.

```
<service android:enabled="true" android:name="LocationTrackingService">
     <intent-filter>
         <action android:name="it.unipr.dsg.tracker.LocationTrackingService.SERVICE" ></action>
     </intent-filter>
</service>
```

Start a Service

- Starting a service is a easy procedure based on the creation of a Intent. You can use a Implicit or Explicit Intent.
- With an Implicit Intent you should define an IntentFilter for the Service and use the one of the defined action to start the Service.
- With an Explicit Intent you directly use the Context and the Service class name as already done to start an Activity.
- The code to stop the Service is essentially the same as starting the Service but with a call to the **stopService()** method.

Implicit Intent →

```
Intent service = new Intent("it.unipr.dsg.tracker.LocationTrackingService.SERVICE");
startService(service); / stopService(service);
```

Explicit Intent →

```
Intent service = new Intent(mContext,LocationTrackingService.class);
startService(service); / stopService(service);
```

Service and User Notification

- Once running, a service can notify the user of events using Toast Notifications or Status Bar Notifications.
- A toast notification is a message that appears on the surface of the current window for a moment then disappears, while a status bar notification provides an icon in the status bar with a message, which the user can select in order to take an action (such as start an activity).
- Usually, a status bar notification is the best technique when some background work has completed (such as a file completed downloading) and the user can now act on it. When the user selects the notification from the expanded view, the notification can start an activity (such as to view the downloaded file).

Creating a Bound Service

- Sometimes is useful to have more control over a service than just system calls to start and stop its activities.
- A bound service is one that allows application components to bind to it by calling **bindService()** in order to create a long-standing connection (and generally does not allow components to start it by calling startService()).
- You should create a bound service when you want to interact with the service from activities and other components in your application or to expose some of your application's functionality to other applications, through interprocess communication (IPC).
- To create a bound service, you must implement the onBind() callback method to return an **IBinder** that defines the interface for communication with the service. Other application components can then call bindService() to retrieve the interface and begin calling methods on the service.
- The service lives only to serve the application component that is bound to it, so when there are no components bound to the service, the system destroys it (you do not need to stop a bound service in the way you must when the service is started through onStartCommand()).

Creating a Bound Service

- There are three ways you can define the IBinder interface:
  - **Extending the Binder class:** If your service is private to your own application and runs in the same process as the client (which is common), you should create your interface by extending the Binder class and returning an instance of it from onBind(). The client receives the Binder and can use it to directly access public methods available in either the Binder implementation or even the Service. This is the preferred technique when your service is merely a background worker for your own application. The only reason you would not create your interface this way is because your service is used by other applications or across separate processes.
  - **Using a Messenger:** If you need your interface to work across different processes, you can create an interface for the service with a Messenger. In this manner, the service defines a Handler that responds to different types of Message objects. This Handler is the basis for a Messenger that can then share an IBinder with the client, allowing the client to send commands to the service using Message objects. Additionally, the client can define a Messenger of its own so the service can send messages back. This is the simplest way to perform interprocess communication (IPC), because the Messenger queues all requests into a single thread so that you don't have to design your service to be thread-safe.

Creating a Bound Service

- **Using AIDL:** AIDL (Android Interface Definition Language) performs all the work to decompose objects into primitives that the operating system can understand and marshall them across processes to perform IPC. The previous technique, using a Messenger, is actually based on AIDL as its underlying structure. As mentioned above, the Messenger creates a queue of all the client requests in a single thread, so the service receives requests one at a time. If, however, you want your service to handle multiple requests simultaneously, then you can use AIDL directly. In this case, your service must be capable of multi-threading and be built thread-safe. To use AIDL directly, you must create an **.aidl** file that defines the programming interface. The Android SDK tools use this file to generate an abstract class that implements the interface and handles IPC, which you can then extend within your service.

```
// IRemoteService.aidl
package com.example.android;

// Declare any non-default types here with import statements

/** Example service interface */
interface IRemoteService {
    /** Request the process ID of this service, to do evil things with it. */
    int getPid();

    /** Demonstrates some basic types that you can use as parameters
     * and return values in AIDL.
     */
    void basicTypes(int anInt, long aLong, boolean aBoolean, float aFloat,
            double aDouble, String aString);
}
```

Extending the Binder class

- If your service is used only by the local application and does not need to work across processes, then you can implement your own Binder class that provides your client direct access to public methods in the service.
- Note: **This works only if the client and service are in the same application and process, which is most common.** For example, this would work well for a music application that needs to bind an activity to its own service that's playing music in the background.
- To set up a Binder you should:
  - Create an instance of Binder that:
    - contains public methods that 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 and make calls to the bound service using the methods provided.

Extending the Binder class

```
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.  Because we know this service always
     * runs in the same process as its clients, we don't need to deal with IPC.
     */
    public class LocalBinder extends Binder {
        LocalService getService() {
            // Return this instance of LocalService so clients can call public methods
            return LocalService.this;
        }
    }

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

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

The LocalBinder provides the **getService()** method for clients to retrieve the current instance of LocalService.

This allows clients to call public methods in the service. For example, clients can call **getRandomNumber()** from the service.

Public Method/s for the client

Binding to a Service

- Application components (clients) can bind to a service by calling bindService(). 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 and does not return the IBinder to the client.
- To receive the IBinder, the client must create an instance of ServiceConnection and pass it to bindService(). The ServiceConnection includes a callback method that the system calls to deliver the IBinder.
- **Note**: Only activities, services, and content providers can bind to a service—you cannot bind to a service from a broadcast receiver.

Binding to a Service

- To bind to a service from your client, you must:
  - Implement ServiceConnection.Your implementation must override two callback methods:
    - onServiceConnected() The system calls this to deliver the IBinder returned by the service's onBind() method.
    - onServiceDisconnected()
      The Android system calls this when the connection to the service is unexpectedly lost, such as when the service has crashed or has been killed. This is not called when the client unbinds.
- Call bindService(), passing the ServiceConnection implementation.
- When the system calls your onServiceConnected() callback method, you can begin making calls to the service, using the methods defined by the interface.
- To disconnect from the service, call unbindService(). When your client is destroyed, it will unbind from the service, but you should 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.

Binding to a Service

Save the Service reference to in a dedicated variable in order to access public method provided by the Service.

```
LocalService mService;
private ServiceConnection mConnection = new ServiceConnection() {
    // Called when the connection with the service is established
    public void onServiceConnected(ComponentName className, IBinder service) {
        // Because we have bound to an explicit
        // service that is running in our own process, we can
        // cast its IBinder to a concrete class and directly access it.
        LocalBinder binder = (LocalBinder) service;
        mService = binder.getService();
    }

    // Called when the connection with the service disconnects unexpectedly
    public void onServiceDisconnected(ComponentName className) {
        Log.e(TAG, "onServiceDisconnected");
        mService = NULL;
    }
};
```

Binding to a Service

- With this ServiceConnection, the client can bind to a service by passing this it to bindService(). For example:
- The first parameter of bindService() is an Intent that explicitly names the service to bind (thought the intent could be implicit).
- The second parameter is the ServiceConnection object.
- The third parameter is a flag indicating options for the binding. It should usually be BIND_AUTO_CREATE in order to create the service if its not already alive.

```
Intent intent = new Intent(this, LocalService.class);
bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
```

Bound Service Notes

- You should usually pair the binding and unbinding during matching bring-up and tear-down moments of the client's lifecycle. For example:
  - If you only need to interact with the service while your activity is visible, you should bind during onStart() and unbind during onStop().
  - If you want your activity to receive responses even while it is stopped in the background, then you can bind during onCreate() and unbind during onDestroy(). Beware that this implies that your activity needs to use the service the entire time it's running (even in the background), so if the service is in another process, then you increase the weight of the process and it becomes more likely that the system will kill it.
  - **Note**: You should usually not bind and unbind during your activity's onResume() and onPause(), because these callbacks occur at every lifecycle transition and you should keep the processing that occurs at these transitions to a minimum. Also, if multiple activities in your application bind to the same service and there is a transition between two of those activities, the service may be destroyed and recreated as the current activity unbinds (during pause) before the next one binds (during resume). (This activity transition for how activities coordinate their lifecycles is described in the Activities document.)

Running a Service in the Foreground

- A foreground service
  - is a service associated to something that the user is aware of and thus not a candidate for the system to kill when low on memory.
  - must provide a notification for the status bar, which is placed under the "Ongoing" heading, which means that the notification cannot be dismissed unless the service is either stopped or removed from the foreground.
- A music service that plays media files should be set to run in the foreground, because the user is explicitly aware of its operation. The notification in the status bar might indicate the current song and allow the user to launch an activity to interact with the music player.
- The method to start a service in the foreground is **startForeground()**. This method takes two parameters: an integer that uniquely identifies the notification and the Notification for the status bar.

http://developer.android.com/guide/topics/fundamentals/services.html#Foreground

Running a Service in the Foreground

- Inside your Service you should create the Notification and call the startForeground() method as illustrated in the following code:

```
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_messa
ge), pendingIntent);

startForeground(ONGOING_NOTIFICATION, notification);
```

- Call stopForeground() in order to remove the service from the foreground. The boolean parameter indicates whether to remove the status bar notification as well. This method does not stop the service. However, if you stop the service while it's still running in the foreground, then the notification is also removed.

Intent

- Three of the core components of an application are:
  - Activities
  - Services
  - Broadcast Receivers
- They are activated through messages, called intents. Intent messaging is a facility for late run-time binding between components in the same or different applications.
- An Intent object, is a passive data structure holding an abstract description of an operation to be performed — or, often in the case of broadcasts, a description of something that has happened and is being announced.

Intent

- There are separate mechanisms for delivering intents to each type of component:
  - An Intent object is passed to **Context.startActivity()** or **Activity.startActivityForResult()** to launch an activity or get an existing activity to do something new. (It can also be passed to **Activity.setResult()** to return information to the activity that called **startActivityForResult()**.)
  - An Intent object is passed to **Context.startService()** to initiate a service or deliver new instructions to an ongoing service. Similarly, an intent can be passed to **Context.bindService()** to establish a connection between the calling component and a target service. It can optionally initiate the service if it's not already running.
  - Intent objects passed to any of the broadcast methods (such as **Context.sendBroadcast()**, **Context.sendOrderedBroadcast()**, or **Context.sendStickyBroadcast()**) are delivered to all interested broadcast receivers. Many kinds of broadcasts originate in system code.
  - **Intent are really useful with Android Service to exchange information and notification between the calling object and the Service.**

Intent Filter

- To inform the system which implicit intents they can handle, **activities, services, and broadcast receivers can have one or more intent filters.**
- Each filter describes a capability of the component, a set of intents that the component is willing to receive. It, in effect, filters in intents of a desired type, while filtering out unwanted intents — but only unwanted implicit intents (those that don't name a target class).
- **An explicit intent is always delivered to its target, no matter what it contains; the filter is not consulted.**
- **An implicit intent is delivered to a component only if it can pass through one of the component's filters.**
- An intent filter is an instance of the IntentFilter class. However, since the Android system must know about the capabilities of a component before it can launch that component, intent filters are generally not set up in Java code, but in the application's manifest file (AndroidManifest.xml) as <intent-filter> elements. (The one exception would be filters for broadcast receivers that are registered dynamically by calling Context.registerReceiver(); they are directly created as IntentFilter objects.)

Intent Filter

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

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
    <intent-filter>
        <action android:name="it.unipr.dsg.tracker.NEW_LOCATION" ></action>
    </intent-filter>
</activity>

<service android:enabled="true" android:name="LocationTrackingService">
    <intent-filter>
        <action android:name="it.unipr.dsg.tracker.LocationTrackingService.SERVICE" ></action>
    </intent-filter>
</service>

<receiver android:name="it.unipr.dsg.tracker.LocationTrackingReceiver">
    <intent-filter>
        <action android:name="it.unipr.dsg.tracker.NEW_LOCATION" ></action>
    </intent-filter>
</receiver>
```

Broadcast Receiver

- A broadcast receiver is a class which extends BroadcastReceiver and which is registered as a receiver in an Android Application via the AndroidManifest.xml file (or via code).
- Alternatively to the this static registration, you can also register a BroadcastReceiver dynamically via the Context.registerReceiver() method.
- This class will be able to receive intents. Intents can be generated via the Context.sendBroadcast() method.
- The class BroadcastReceiver defines the onReceive() method. Only during this method yourBroadcastReceiver object will be valid, afterwards the Android system can recycle theBroadcastReceiver. Therefore you cannot perform any asynchronous operation in the onReceive() method.
- The sendBroadcast() method allows to send Broadcast Intents. Broadcasted messages are intercepted by Activity, Service and stand alone BroadCast Receiver with that declare the appropriate

Broadcast Receiver Example 1

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="package.name.test"
    android:versionCode="1"
    android:versionName="1.0" >

     <uses-sdk android:minSdkVersion="10" />

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

        <application
            android:icon="@drawable/icon"
            android:label="@string/app_name" >
            <receiver android:name="TestPhoneReceiver" >
                <intent-filter>
                    <action android:name="android.intent.action.PHONE_STATE" >
                    </action>
                </intent-filter>
            </receiver>
        </application>
</manifest>
```

Broadcast Receiver Example 1

```
package package.name.test;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.telephony.TelephonyManager;
import android.util.Log;

public class MyPhoneReceiver extends BroadcastReceiver {
   @Override
   public void onReceive(Context context, Intent intent) {
      Bundle extras = intent.getExtras();
      if (extras != null) {
         String state = extras.getString(TelephonyManager.EXTRA_STATE);
         Log.d("TEST", state);
         if (state.equals(TelephonyManager.EXTRA_STATE_RINGING)) {
            String phoneNumber = extras
                  .getString(TelephonyManager.EXTRA_INCOMING_NUMBER);
            Log.d("TEST", phoneNumber);
         }
      }
   }
}
```

Broadcast Receiver Example 2

```
<activity
      android:name=".MainActivity"
      android:label="@string/app_name" >
      <intent-filter>
       <action android:name="android.intent.action.MAIN" />
          <category android:name="android.intent.category.LAUNCHER" />
       </intent-filter>
       <intent-filter>
           <action android:name="it.unipr.dsg.tracker.NEW_LOCATION" ></action>
       </intent-filter>
  </activity>
```

```
public class ActivityLocationTrackingReceiver extends BroadcastReceiver {

   @Override
   public void onReceive(Context context, Intent intent) {
     Bundle extras = intent.getExtras();
     Log.d(MainActivity.TAG,"ActivityLocationTrackingReceiver ---> Received Intent: " + intent.getAction());
   }
}
```

Background Tasks & Sleep Mode

- Android sleep mode primarily shuts down the CPU. Along the way, non-essential radios (WiFi, GPS) will have been shut down as well.
- It does not stop the GSM or CDMA radio (for incoming calls, SMS, and IP packets) and AlarmManager.
- In order to keep a background task active when the device goes to SleepMode you can use:
  - The PowerManager class to set a WakeLock (keeps the CPU and/or the screen active) and/or a WifiManager.WifiLock to keep the Wi-Fi radio awake.
  - AlarmManager and AlarmReceiver to periodically receive an Intent and restart a background task.

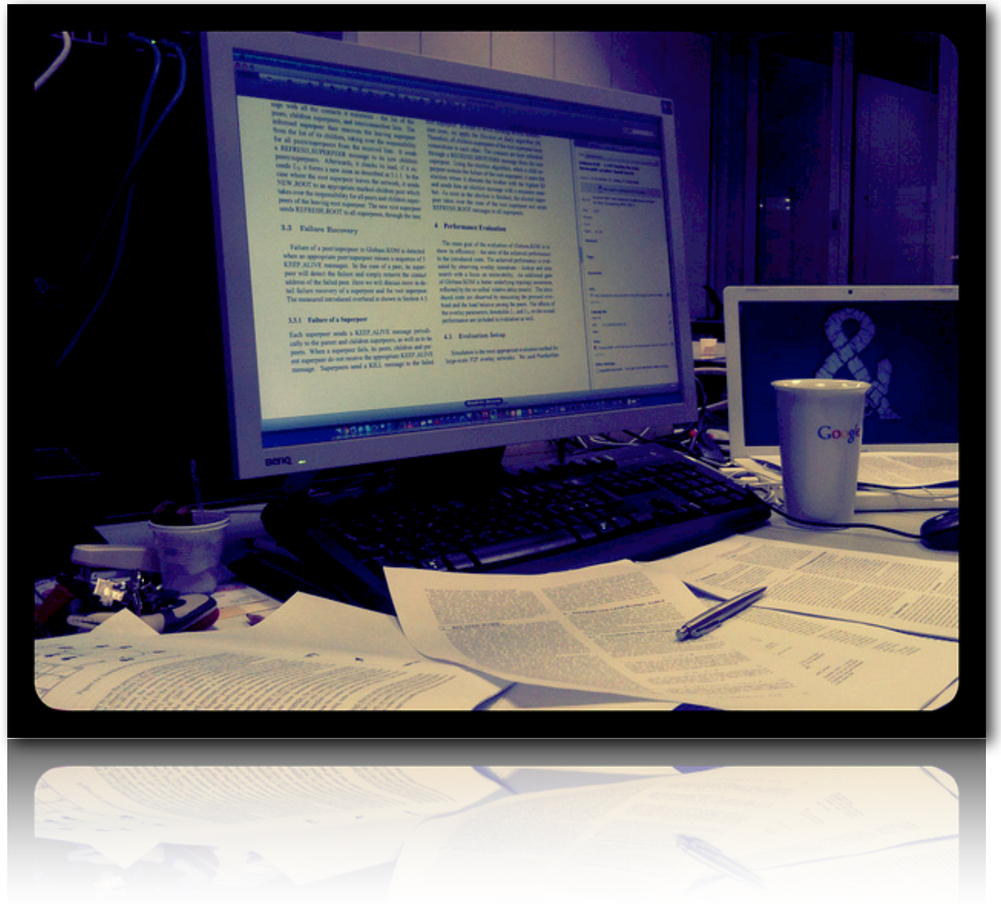
http://developer.android.com/reference/android/app/AlarmManager.html

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

http://developer.android.com/reference/android/net/wifi/WifiManager.WifiLock.html

Coming Up

- Next Lecture
  - Networking
- Homework
  - Review unBound and Bound Service Applications.
  - Update one of the presented application adding a MapView showing user location.



Android Development

Lecture 8
Android Background Services