Project 3: Android Elaboration

Out: Tuesday April 17
Due: Tuesday May 1 (11:59 PM)
Note: There is a midterm on 4/25
Turnin: Online
Teams: Pairs

There are some "unusual things about Android applications, and the Android build process. This elaboration tries to save us all a lot of time by explaining some things that could cost a lot of time but aren't central to CSE 461.

---

1. The Android App Lifecycle

We're used to a world in which users launch an app, it runs for a while, and the user then terminates it. The user may move away from our application while it's running and interact with others, but that's irrelevant to the life cycle of our app.

In Android things are a bit different. For one thing, Android has the notion of a stack of running applications. The application on top of the stack currently owns the display. The others may, or may not, be running. (Usually, they are, but it's not guaranteed.) The user adds an app to the top of the stack by launching it. Apps are popped by the user hitting the back button.

For another thing, the phone wants to act as though it's always running (never shut down completely), and that any app that has ever been launched has been running since that first launch - when you bring up an app that has, technically, stopped running since last you saw it, it may present a screen that is in exactly the state you left it in last time. Android provides support to make that easier to implement. A side effect is that you have to understand the app life cycle to understand how to implement.

The figure on this page (scroll down a bit) explains the app lifecycle. When your app comes up, it (or actually its Activity, which for Project 3 is equivalent) receives `onCreate()`, `onStart()`, and `onResume` callbacks, in that order. It then owns the screen. Powering off the display by hitting the power button, then turning it back on, results in an `onPause`/`onResume` sequence. Hitting the home button results in `onPause` / `onStop`. If your app is then relaunched (through the application menu), it sees `onStart` / `onResume`. Hitting the back button after launching your app results in `onPause`, `onStop`,`onDestroy`. You'd think that `onDestroy` was the event corresponding to normal app termination, but it isn't. If an `onStop` is received, it's possible that your app will be removed from memory, without the courtesy of an `onDestroy` callback.

This upshot of all this is that you should initialize the infrastructure (call `OS.boot()`, and the like) in the `onStart` method, and call `OS.shutdown` in the`onStop` method. You more or less have to do them there because when you get an `onStop` call you can't be sure your program's state, including TCP sockets and threads, will still be around when you get back to `onStart`. A side effect of this is that the RPC Service's port changes when the user pops your app off the stack. That's unfortunate. There's a way around this in Android ("services"), but that adds another level of complexity we don't need.

---

2. Config Files

Put the config file in the `assets` folder of your Eclipse Android project, then:

```
String configFilename = "jz.cse461.config.ini";
Properties config = new Properties();
config.load(getAssets().open(configFilename));
```

(This code must be executed inside your `Activity` object.)

---

3. JAR Files

You may need to inject JAR files into your application's `apk` file (the Android executable). You'll know if you see Android `LogCat` output indicating that some class that normally comes from a `.jar` can't be found. In my Ping, I had to inject `commons-cli-1.2.jar` (a copy of which is the `lib/` directory).

To export `.jar`'s:

- Create a folder named `libs` in your Eclipse Android project
- Create copies of the `.jar`'s in that folder

---

4. Communicating with the Emulator

The emulator is NAT'ed. (See this page if you'd like all the details.) Quoting the main assignment, "Finally, you can't connect to machines running behind NATs from machines not also behind the NAT." The emulator is the only machine behind its NAT. That means the emulator can ping itself, but no other machine can ping the emulator. The emulator's IP address for itself is both 127.0.0.1 and the string "localhost".

The emulator has no trouble pinging other machines, so long as they're not behind some NAT it isn't also behind. It can ping your development machine, or`cse461.cs.washington.edu`. It probably can't ping your neighbor's home machine, though.

---

5. (A) Obtaining The Phone's IP Address and (B) Eclipse

Part A
The usual Java way of obtaining your own IP address (the one that works on CSE machines, and your machine) doesn't work on Android. The following seems to work on physical phones, but not the emulator:

```
  WifiManager wifiManager =
(WifiManager)getSystemService(WIFI_SERVICE);
  WifiInfo wifiInfo =
wifiManager.getConnectionInfo();
  int wifiIPInt = wifiInfo.getIpAddress();
  String wifiIP =
Formatter.formatIpAddress(wifiIPInt);
```

On the emulator and the Android 2.3 phones, this seems to work:

```
   Enumeration interfaces =
NetworkInterface.getNetworkInterfaces();
    NetworkInterface iface;
    while ( interfaces.hasMoreElements() ) {
       iface = interfaces.nextElement();
       if ( iface.isLoopback() ) continue;
       EnumerationipVec =  iface.getInetAddresses();
       while ( ipVec.hasMoreElements() ) {
         InetAddress addr = ipVec.nextElement();
       }
    }
```

(On Android 4.0, it seems to produce an IPv6 address, but not an IPv4 one.)

I don't know of any single technique that works everywhere. Of course, the IP of the emulator is nearly worthless, as it can't be contacted from outside anyway, so the first technique probably suffices for your code.

Part B
The RPCService class has a public method, localIP() that returns the local IP address as a String. The issue is making that code portable: the Java/Linux implementation doesn't work on Java/Android, and the Java/Android implementation doesn't even compile on Java/Linux.

In theory, this problem isn't so bad. Create two implementations of some class, say IPFinder, one for Android and one for everything else. Include only one of the two in your application, depending on whether the target is Android or not-Android. Finally, in RPCService.localIP invoke a static method ofIPFinder to get the local IP address.

The problem now is Eclipse. You need to compile one of the two in an Android project, and the other in a Java Application project. Additionally, Eclipse is picky about what's in folders and various other things.

The easiest thing to do is to create two new projects, utilConsoleOnly and utilAndroidOnly. Put the Android version of IPFinder in the latter, and the other version in the former. Now (we're almost done...) in your Android Ping project, link the folder containing the Android version (by using Build Path... then Configure Buid Path then the Source tab then Link Folder; select the src folder of the version you want). Do the corresponding thing for non-Android projects.

Confused? It'll make more sense when you try it, but we're also available for help.

---

6. Manifest Permissions

For security reasons, Android requires applications to declare the protected resources they'd like to use. You do this in the AndroidManifest.xml file. I added the first three lines shown here. (The fourth is for context, so you know where to put them.)

```
    <uses-permission
android:name="android.permission.INTERNET" />
    <uses-permission
android:name="android.permission.ACCESS_WIFI_STATE"
/>
    <uses-permission
android:name="android.permission.ACCESS_NETWORK_STAT
E" />

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

7. Android 4.0 Woes / Android 2.3 Tips

Starting with level 3.0, Android includes an improvement that strictly prohibits trying to talk over the network using the main (UI) thread. That means that if you have a phone running Android 4.0, you *must* create a thread to boot the OS and do other initialization. (Below Android level 3.0, it's *recommended* that you not use the UI thread for networking, but it's allowed.) Also, when it's time to perform ping (the user has hit the ping button, say), you must create a thread to do the pinging. It's not too bad, although admittedly another complication. See this page, and the example code below.

The final complication is that only the UI thread can update the display. (Geesh...) So, when the thread you spawned to collect a ping result gets that result, it needs to do something special to display the result. The easiest thing is for it to use runOnUiThread() to execute a code fragment to update the display.

As an example of both mechanisms, here's some code in my Ping. (I've omitted `try...catch` for clarity.) `initialize()` calls `OS.boot()` and does other initialization. I then display the phone's IP and port on the screen. An instance variable, `mMyIP`, is used to communicate the IP address string from the non-UI thread that figured it out to the UI thread that can actually display it. (This code is in the main Activity's `onStart()` method.)

```
   // main thread is running here, in onStart()
   ...
   new Thread(){
     public void run() {
       // worker thread starts running asynchronously
here
       initialize();
       mMyIP =
((RPCService)OS.getService("rpc")).localIP();

       runOnUiThread(new Runnable() {
         public void run() {
           // main thread runs this code
           String msg = mMyIP + ":" +
Integer.toString(((RPCService)OS.getService("rpc")).
localPort());
```

```
          TextView myIpText =
(TextView)findViewById(R.id.myiptext);
          if ( myIpText != null )
myIpText.setText(msg);
        }
      });
    }
  }.start();
```