ECE155: Engineering Design with Embedded Systems Winter 2015

Lecture 31 — Android IV

Jeff Zarnett

Android Networking

We have one more lecture where we're going to look a particularly advanced Android topic: networking. This is not at all necessary for your labs, which is why it did not appear at the start of term along with the rest of the Android material.

Performing Network Operations

Before we get started, note that any application that uses the network in Android will need the following permissions. Code in this section comes from [Gui14c].

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Before connecting to the network, check to make sure that the network is available. The user might have "Airplane Mode" turned on and therefore neither mobile internet nor Wi-Fi is available, for example.

```
ConnectivityManager connMgr = (ConnectivityManager)
        getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
    if (networkInfo != null && networkInfo.isConnected()) {
    ... // Do useful work
    }
```

Networks are inherently unreliable and can have unknown latency and bandwidth. If we did network operations in the main activity UI thread, we might get the "Not Responding" dialog (remember that from an earlier lecture?). To prevent this from appearing, network operations should take place in a different thread from the UI. How do we do that? In a background task (or asynchronous task), specifically the class `AsyncTask`.

AsyncTask Here is an example of a defined `AsyncTask`. Explanation follows.

```
private class DownloadWebpageTask extends AsyncTask<String, Void, String> {
        @Override
        protected String doInBackground(String... urls) {

            // params comes from the execute() call: params[0] is the url.
            try {
                return downloadUrl(urls[0]);
            } catch (IOException e) {
                return "Unable to retrieve web page. URL may be invalid.";
            }
        }
        // onPostExecute displays the results of the AsyncTask.
        @Override
        protected void onPostExecute(String result) {
            textView.setText(result);
       }
    }
```

The AsyncTask we have here is defined as a private class inside a MainActivity but it could be defined as an inner class (new AsyncTask<String, Void, String> { ...}) or in its own file. Like a collection, such as List, the AsyncTask takes several parameters inside angle brackets. The first type is the one applicable to the parameter list of doInBackground; the second for onProgressUpdate if any; the third for onPostExecute. Change them as necessary to the applicable types. If you do not need one for some reason, you can put type Void (note capital V).

According to the Android Developer guidelines on the AsyncTask [Gui14a]: When an asynchronous task is executed, the task goes through 4 steps:

1. onPreExecute(), invoked on the UI thread before the task is executed. This step is normally used to setup the task, for instance by showing a progress bar in the user interface. Optional; not shown in this example.
2. doInBackground(Params...), invoked on the background thread immediately after onPreExecute() finishes executing. This step is used to perform background computation that can take a long time. The parameters of the asynchronous task are passed to this step. The result of the computation must be returned by this step and will be passed back to the last step. This step can also use publishProgress(Progress...) to publish one or more units of progress. These values are published on the UI thread, in the onProgressUpdate(Progress...) step. Mandatory (otherwise why use a task?)
3. onProgressUpdate(Progress...), invoked on the UI thread after a call to publishProgress(Progress...). The timing of the execution is undefined. This method is used to display any form of progress in the user interface while the background computation is still executing. For instance, it can be used to animate a progress bar or show logs in a text field. Optional; not shown in this example.
4. onPostExecute(Result), invoked on the UI thread after the background computation finishes. The result of the background computation is passed to this step as a parameter.

To actually execute the Download Webpage task, use new DownloadWebpageTask().execute(url); where you fill in the parameters to execute as is appropriate to the AsyncTask. A task can be executed only once; an exception will be thrown if execute is called again. To execute the same thing, create another instance of the task with the new keyword.

A task can be cancelled while it is running by calling cancel(boolean), but this only makes isCancelled() return true; the doInBackground method should check and see if the task has been cancelled and if so, stop doing what it is doing. Also, after the task is cancelled, instead of onPostExecute, onCancelled is called (also not shown in the example).

Actually Using the Network So after all of that, we are ready to actually use the network!

```
// Given a URL, establishes an HttpUrlConnection and retrieves
// the web page content as a InputStream, which it returns as
// a string.
private String downloadUrl(String myurl) throws IOException {
    InputStream is = null;
    // Only display the first 500 characters of the retrieved
    // web page content.
    int len = 500;

    try {
        URL url = new URL(myurl);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setReadTimeout(10000 /* milliseconds */);
        conn.setConnectTimeout(15000 /* milliseconds */);
        conn.setRequestMethod("GET");
        conn.setDoInput(true);
        // Starts the query
        conn.connect();
        int response = conn.getResponseCode();
        Log.d(DEBUG_TAG, "The response is: " + response);
        is = conn.getInputStream();

        // Convert the InputStream into a string
        String contentAsString = readIt(is, len);
        return contentAsString;

    // Makes sure that the InputStream is closed after the app is
    // finished using it.
    } finally {
        if (is != null) {
            is.close();
        }
    }
}

// Reads an InputStream and converts it to a String.
public String readIt(InputStream stream, int len) throws IOException, UnsupportedEncodingException {
    Reader reader = null;
    reader = new InputStreamReader(stream, "UTF-8");
    char[] buffer = new char[len];
    reader.read(buffer);
    return new String(buffer);
}
```

The class `HttpURLConnection` is the key to making the connection over the web. Data can be of any type and it's not necessary to know in advance the length of the data.

Uses of this class follow a pattern [Gui14b]:

1. Obtain a new `HttpURLConnection` by calling `URL.openConnection()` and casting the result to `HttpURLConnection`.
2. Prepare the request. The primary property of a request is its URI. Request headers may also include metadata such as credentials, preferred content types, and session cookies.
3. Optionally upload a request body. Instances must be configured with `setDoOutput(true)` if they include a request body. Transmit data by writing to the stream returned by `getOutputStream()`.
4. Read the response. Response headers typically include metadata such as the response body's content type and length, modified dates and session cookies. The response body may be read from the stream returned by `getInputStream()`. If the response has no body, that method returns an empty stream.

5. Disconnect. Once the response body has been read, the `HttpURLConnection` should be closed by calling `disconnect()`. Disconnecting releases the resources held by a connection so they may be closed or reused.

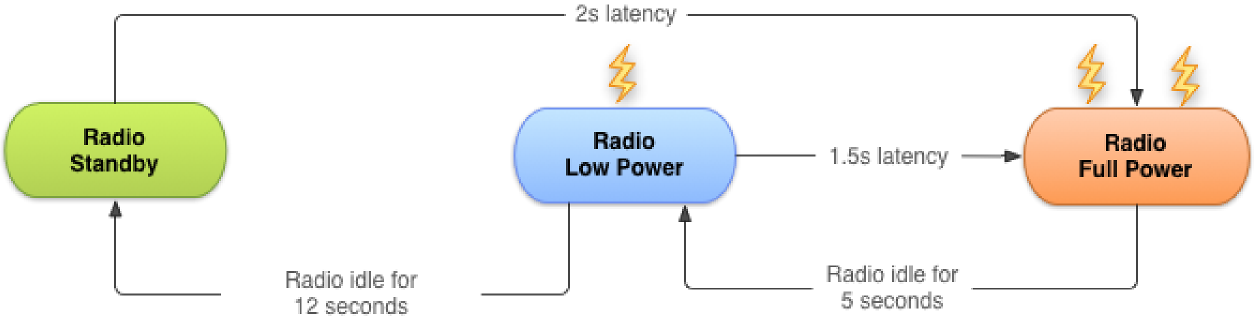
Calling `openConnection()` on a URL with the "https" (HTTP with SSL, security) scheme will return an `HttpsURLConnection`. We are not going to cover this; for more detail and including things like posting content and authentication, take a look at [Gui14b].

On Battery Use

Other than the screen, the next biggest user of battery is likely the wireless radio. The radio for a typical 3G divide has three states:

1. **Full Power**: Uses most power, transfers data at the highest rate.
2. **Low Power**: Uses half power.
3. **Standby**: Uses minimal power, when no connection is active.

Transition from one state to another is not instant. When the radio is at full power, only if it is idle for five seconds will it transition to the low power state; if it is in low power state, if it is idle for 12 seconds it will go into the standby state. This prevents the situation where the radio is constantly jumping between states due to short pauses. See the diagram below for approximate transition times:

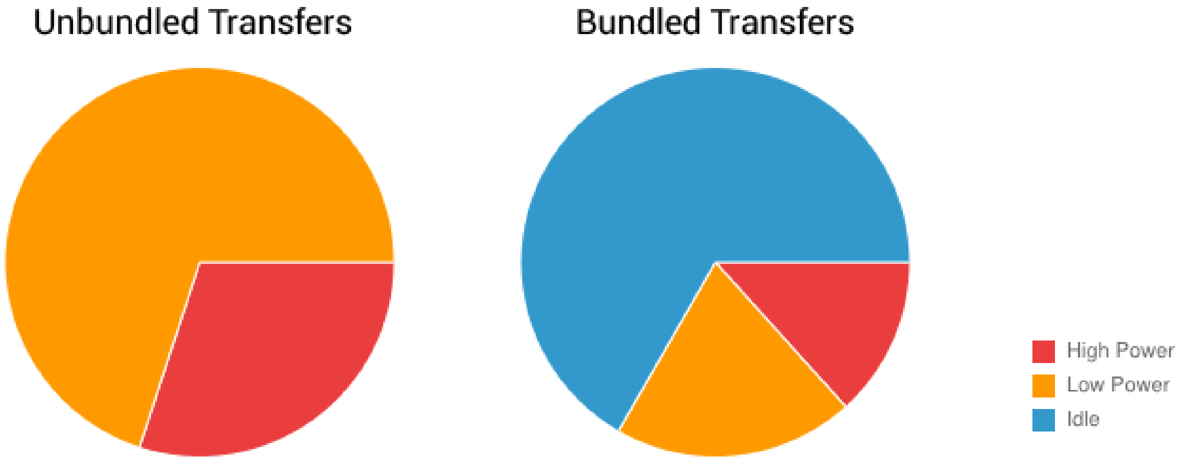


[Gui14f]

Creating a new network connection puts the radio in the full power state. If the radio is configured according to the above diagram, a one second transfer is followed by five more seconds of “tail time” in the high power state, then twelve seconds in the low power state, before the radio returns to a standby state, for a total of 18 seconds.

If an app transfers unbundled data for 1 second every 18 seconds, it will prevent the radio from ever going to standby, and this will drain a lot of battery. Out of every 60 seconds, 18 will be in the high power state, and the radio will be at the low power state for the remaining 42 seconds.

If we bundled the transfers (i.e., transfer data in bulk), we could save significantly. If we did 3 consecutive seconds of transfer, the radio would be in the high power state for only 8 seconds, and then in the low power state for an additional 12 seconds. The radio would then be idle for the remaining 40 seconds.



Unbundled Transfers

Bundled Transfers

[Gui14f]

Cloud Sync

It's possible to sync your app with the cloud so that user data is not lost even when they reinstall the app or change phones. Google provides a Backup API for small amounts of data (around 1 Megabyte) for storing the user's preferences, high scores, notes etc.

To use the backup service, register for it (https://developer.android.com/google/backup/signup.html?csw=1) and then add the XML tag to your manifest. Then implement the Backup Agent. Example code from [Gui14e]:

```
<application android:label="MyApp"
             android:backupAgent="TheBackupAgent">
    ...
    <meta-data android:name="com.google.android.backup.api_key"
    android:value="ABcDe1FGHij2KlmN3oPQRs4TUvW5xYZ" />
    ...
</application>
```

This registers a backup agent called "TheBackupAgent" (change the name as appropriate) which then we implement. Inside the onCreate() method, create a `BackupHelper`, which in this case looks like this:

```
 import android.app.backup.BackupAgentHelper;
 import android.app.backup.FileBackupHelper;


 public class TheBackupAgent extends BackupAgentHelper {
    // The name of the SharedPreferences file
    static final String HIGH_SCORES_FILENAME = "scores";

    // A key to uniquely identify the set of backup data
    static final String FILES_BACKUP_KEY = "myfiles";

    // Allocate a helper and add it to the backup agent
    @Override
    void onCreate() {
        FileBackupHelper helper = new FileBackupHelper(this, HIGH_SCORES_FILENAME);
        addHelper(FILES_BACKUP_KEY, helper);
    }
}
```

This `BackupAgentHelper` takes backups of the user's high scores file. If you use `SharedPreferences` instead, the backup agent helper implementation would look for like this:

```
import android.app.backup.BackupAgentHelper;
import android.app.backup.SharedPreferencesBackupHelper;

public class TheBackupAgent extends BackupAgentHelper {
    // The names of the SharedPreferences groups that the application maintains.  These
    // are the same strings that are passed to getSharedPreferences(String, int).
    static final String PREFS_DISPLAY = "displayprefs";
    static final String PREFS_SCORES = "highscores";

    // An arbitrary string used within the BackupAgentHelper implementation to
    // identify the SharedPreferencesBackupHelper's data.
    static final String MY_PREFS_BACKUP_KEY = "myprefs";

    // Simply allocate a helper and install it
    void onCreate() {
        SharedPreferencesBackupHelper helper =
                new SharedPreferencesBackupHelper(this, PREFS_DISPLAY, PREFS_SCORES);
        addHelper(MY_PREFS_BACKUP_KEY, helper);
    }
}
```

To request a backup, just create an instance of the `BackupManager`, and call its `dataChanged()` method. This tells the system that data has been changed. If you call `dataChanged()` more than once before the backup actually takes place, the backup will occur only once (so no extra network traffic will be generated). Restoring from backup happens automatically when the user reinstalls the application, but you can force it with `requestRestore()`.

Resolving Sync Conflicts It's possible that when you save data to the cloud you end up with conflicts: the user has two or more devices and more than one of them try to save data to the cloud. There are some simple solutions we could follow [Gui14d]:

- **Strategy 1: Newer is better**. A more recent choice should override an older choice. In this case, you would probably choose to store the timestamp in the cloud save data. When resolving the conflict, pick the data set with the most recent timestamp.
- **Strategy 2: Value Judgement**. Choose based on some data that can be defined as "best". For example, if the data represents the player's best time in a racing game, keep the best (smallest) time.
- **Strategy 3: Merge**. Merge by union. For example, if your data represents the set of levels that player has unlocked, then the resolved data is simply the union of the two conflicting sets. This way, players won't lose any levels they have unlocked.

These strategies work if the conflict and data are simple, but we might also have some more complex situations. If we are tracking something important like money, choosing the higher of the two values, for example, is an incorrect solution. Consider the following scenario where we just store the total:

1. Starting condition: the user has 0 coins on Device A, 0 on Device B.
2. Player collects 10 coins on A.
3. Player collects 15 coins on B.
4. Device B saves.
5. Device A saves - conflict detected.
6. Conflict resolution: choose the largest of the two.

Error occurred: player collected 25 coins but the value of 15 was chosen, so the user has “lost” 10 coins.

Idea: Why not send the deltas instead of the values? What if in this situation instead of setting the total, we just send “+10” and “+15”? This works to resolve the scenario here, but Android will send only the most recent update if network connectivity is not available. Imagine that the user collected 5 coins on A while on an airplane (network off) and then in another session, collected another 5 coins. When the synchronization occurs, only the second update will be sent, so only 5 coins will be added to the user’s total. Still incorrect.

Solution: store sub-totals per device. Have a separate “account” for each device. When the user collects 10 coins on device A, write it into a value for coins collected on A. Then when the time comes to show the overall total, simply sum up the coins collected on A and B.

References

[Gui14a] Android Developer Guide. AsyncTask. `http://developer.android.com/reference/android/os/AsyncTask.html`, 2014. Online; accessed 17-April-2014.

[Gui14b] Android Developer Guide. HttpURLConnection. `https://developer.android.com/reference/java/net/HttpURLConnection.html`, 2014. Online; accessed 17-April-2014.

[Gui14c] Android Developer Guide. Performing network operations. `https://developer.android.com/training/basics/network-ops/index.html`, 2014. Online; accessed 17-April-2014.

[Gui14d] Android Developer Guide. Resolving cloud save conflicts. `https://developer.android.com/training/cloudsave/conflict-res.html`, 2014. Online; accessed 17-April-2014.

[Gui14e] Android Developer Guide. Syncing to the cloud. `https://developer.android.com/training/cloudsync/index.html`, 2014. Online; accessed 17-April-2014.

[Gui14f] Android Developer Guide. Transferring data without draining the battery. `https://developer.android.com/training/efficient-downloads/index.html`, 2014. Online; accessed 17-April-2014.