A safari walk-through the Android™ JNI borderline

Antonio Troina
Matr. 708267, (antonio.troina@mail.polimi.it)

Introductory report for the M.Sc. thesis in Computer Science Engineering
Supervisor: PhD. Patrick Bellasi (bellasi@elet.polimi.it)

Last update: November 11, 2012

Abstract

This article aims to briefly describe, in the form of some simple and annotated tutorials, how a developer can take advantage of some JNI capabilities under Android operating system to let the Java and Native environments communicate to each other. This operation has been made quite easy by the Android's native development kit (NDK). Special attention will be given to the callback mechanism, which is by far the most complex, nevertheless the most challenging, under the developer's point of view.

1 Introduction

The coming of Android in the smart-phones market would suggest that Java is definitely enough to rule this galaxy. Moreover, lately, the little-green-robot's operative system has approached, silently but firmly, to the embedded world which, typically, wasn't famous for being dominated by the Java language, particularly for its performance-oriented needs. That being said, who's responsible for connecting the high-level Java application layer to the native world, and the other way around? Yes, this filling is JNI, which was initially released in early 1997. With JNI, the developer can achieve two main goals: reusing his native code within a Java environment, and optimising the execution with regard to performances, so that intensive operations can run natively, instead of being interpreted, as Java pattern requires - except for the peculiar case of the *JIT-ed* code, where the bytecode is compiled *Just In Time* to run natively (this operation commonly runs at launch time, but can happen at install time, or at method invoke time).
So far, some basic examples are available (mainly on-line) which, however, are often more theoretical than practical, therefore this article is meant to be a concrete hands-on guide, to discover - some of - the secrets behind this powerful instrument which is JNI.

1.1 JNI, Android and NDK

JNI per se basically needs two components to be used: the *javah* JDK tool, which builds c-style header files from a given Java class (that will be implemented afterwards in a proper native source file, which includes the mentioned header), and the *jni.h* header file, which maps the Java types to their native counterparts. The whole flow (shown in **Figure 1**) mainly lies in four steps:

- implement a *Java class*, declare the methods you want to call on the native environment as `native`, and compile it
- generate the header file through the `javah -jni` command
- implement as native C/C++ code the function whose signatures have been generated during the step above
- compile the file above as a shared library, which will be loaded by the java class

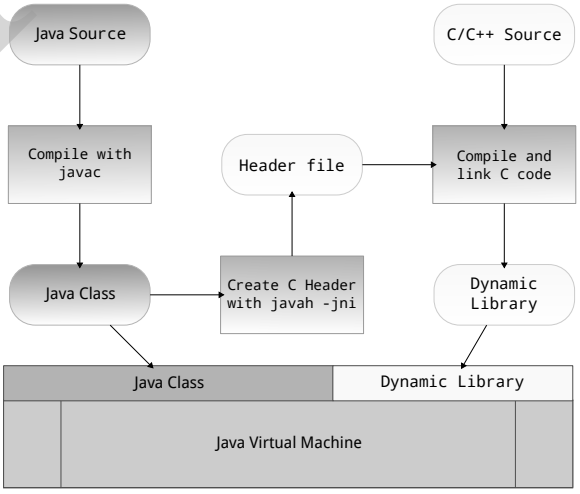

Figure 1: JNI flow

The focus of this work though points to explore JNI within the Android context, therefore our starting point will be the Android NDK, which basically consists of a ready-to-use tool-set (available on `http://developer.android.com/tools/sdk/ndk/index.html#Downloads`). The use of the NDK condenses the steps above in just one main step, which will do almost everything at once, through the

ndk-build command.

2 Getting started

Basically, to develop an "App" using the NDK, we need:

- Android SDK (http://developer.android.com/sdk/index.html)
- Android NDK (http://developer.android.com/tools/sdk/ndk/index.html)
- A source editor. Using Eclipse IDE for Java Developers automatises somehow the building process. (http://www.eclipse.org/downloads/)

From within the *"SDK/tools"* directory, you can execute the android command, to launch the Android SDK Manager: from here, you can download Android API and Extras.
The Eclipse version you've downloaded will be a plain version, not yet customised to deal with specific Android needs and options: you then need to set up Android Tools for Eclipse by clicking *"Help -> Install new software"* and putting the url https://dl-ssl.google.com/android/eclipse/.
Once the environment is set up, we are ready to start developing our fist, small, application, which will interact with a native c-written library. Instead of editing the java source code, and the native code separately, and then compiling them as shown in **Figure 1**, Eclipse will just (automatically) build the Java part, while we'll manually generate the native shared library (we'll see how to automatise this procedure at the end of this article).
The entirety of the files we'll work on (mainly, and at least two) is organised as shown below:

```
Tutorial1/
    AndroidManifest.xml
    jni/
        Android.mk
        tutorial1.c
    res/
        layout/
            main.xml
        values/
            strings.xml
    src/
        com/
            android/
                tutorial1/
                    Tutorial1Activity.java
```

Let's briefly analyse this structure, taking as example the **Tutorial1** at **Section 3**.

Manifest AndroidManifest.xml is the file where the applications are described: here they declare the presence of content providers, services, required permissions and other elements.
In this simple case, it contains the basic nodes, which are:

- manifest: here, under the package, the Java package is specified. We use com.android.tutorial1
- uses-sdk: here it's specified the minimum sdk version (we use API 16, which correspond to Android Jelly Bean)
- application: general information about the application, a human-readable application label, icons.
- activity: here we have the android:name full name of the activity (the java file), a android:label which is the name that will appear at the top of our running Activity, and the <intent-filter> node, which tells Android when this Activity should be run. Since our Tutorial1Activity will also be the one to be run after launch, we can declare this in the <action> node. When an App is called, Android looks for an activity that declares itself ready to resolve the MAIN action. In the second node, under the <intent-filter> node, we declare another attribute, which is <category>, where we specify that the category of our activity is LAUNCHER: this will allow the App to be launched from the Android desktop.

For the sake of clarity, the whole AndroidManifest.xml content is presented at **Listing 1**.

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.android.tutorial1"
      android:versionCode="1"
      android:versionName="1.0">
   <uses-sdk android:minSdkVersion="16" />
   <application android:label="Tutorial1"
                android:debuggable="true">
      <activity android:name=".Tutorial1Activity"
                android:label="Tutorial1">
         <intent-filter>
            <action
            android:name="android.intent.action.MAIN"/>
            <category
            android:name="android.intent.category.LAUNCHER"/>
         </intent-filter>
      </activity>
   </application>
</manifest>
```

Listing 1: AndroidManifest.xml for Tutorial1

Resources The res directory, contains, at least, the following subdirectories:

- layout/: This folder contains Android user interface XML files.
  Without going into detail too much, here we can find each visible element that composes what we see on the screen: strings, buttons, tables, text fields, menus. An example of a button declaration can be seen below

```
<Button android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/button0"
        android:text="@string/button0"
        android:onClick="button0"/>
```

The `@+id/button0` and the `@string/button0` are references to other resources: the former will be used by the Java code to identify this specific button element, the latter identifies the value of the variable `button0` declared into the `strings.xml` file (see below)

- `values/`: This folder contains values that the application will read during its execution, or static data an application will use for such purposes as internationalization of UI strings. In the `string.xml` file, we can declare the values corresponding to the `@string` definition, as shown below, for the `@string/button0` declared above:

```
<string name="button0">Call to native</string>
```

These xml files will then be automatically built into an `R` object, visible to the activity, which will access to it and get views, and set values on its attributes.
When editing native JNI code in an Android project using the Android NDK you may configure **Eclipse** to automatically rebuild your project when editing native code, just as it does for java. Open your project `Properties`, then click `Builder` on the left hand side, and `New` on the right hand side. From the following dialog, choose `Program` and `OK`. In the `Main` tab, we fill the fields as follows:

- Name: `NDK Builder`
- `/opt/android-ndk/ndk-build` (or wherever your ndk-build binary is).
- Working Directory:
  `${workspace_loc:/projectName}` (Press the Browse Workspace... button to select it graphically)

Then we can switch to the second tab (**Figure 2**), `Refresh`: enable the option `"Refresh resources upon completion"` and click the button `Specify Resources...`: choose the `libs` directory within the workspace tree; be sure to check this folder only.

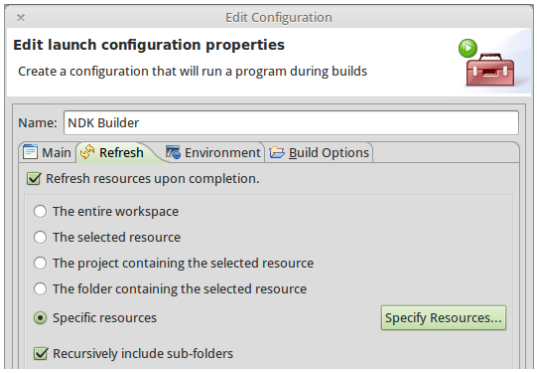

Figure 2: Refresh properties - Eclipse builder

The last tab, named `Build Options` is shown in **Figure 3**: as done for the `Refresh` tab, check the last option, and choose `Specify Resources...`. Choose the `jni` directory within your workspace tree; be sure to check this folder only.

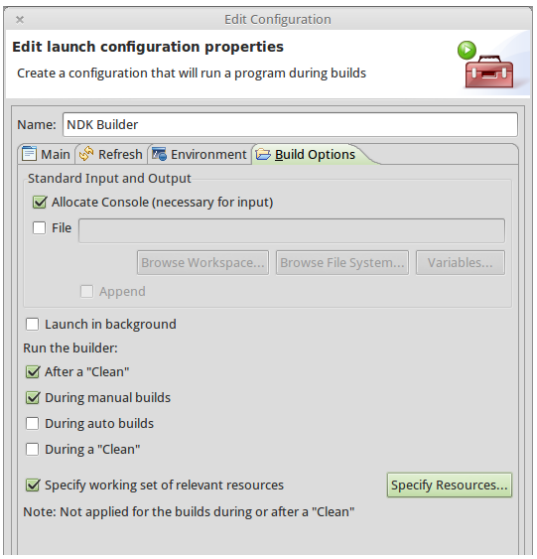

Figure 3: Build properties - Eclipse builder

This being done, the result of a project build (`ctrl+B`) is shown in **Figure 4**.

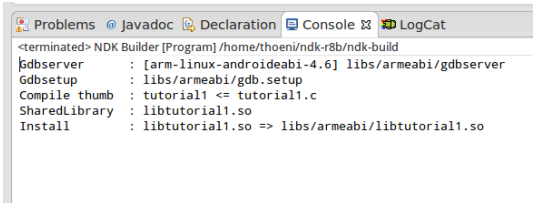

Figure 4: Native building result, into Eclipse console

3 Tutorial1

The complete code from this tutorial can be found at `https://github.com/thoeni/ndk-tutorials/tree/master/tutorial1`.
Eventually, after this brief introduction, we can get to the most interesting part of the tutorial: the Java activity, and the native code, which are respectively placed into the `src` and the `jni` directories.
The activity isn't any different from a regular Android activity except for two things: methods that correspond to native functions are declared as `native`, and a shared library is statically loaded by the `System.loadLibrary()` method.
Basically, this tutorial, illustrates how to call a native function from within the Java activity, and how to call the

native function which, in turn, calls back the Java activity: the methods that perform these operations are, respectively `foo1(String)` and `foo2`, declared within the `Tutorial1Activity`, as shown in **Listing 2**, line 1 and line 2.

```
1  public native String foo1(String message);
2  public native void foo2();
3
4  public void foo3Callback() {
5     String message = "foo3Callback called back by foo2";
6     output.setText(message);
7  }
8
9  static {
10    System.loadLibrary("tutorial1");
11 }
```

Listing 2: Part of Tutorial1Activity.java

In the same piece of code, we can see - line 4 - a regular `public void` method, named `foo3Callback()` which will be called by the native code, and - line 9 - the static declaration to load the shared library that will be generated by the `ndk-build` command.
For this first tutorial, we analyse entirely the native code. These are the basics of JNI, and later on, we won't need to specify everything this much: the C code can be seen at **Listing 3**

```
1  #include <jni.h>
2  #include <android/log.h>
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7
8  #define LOG_TAG "tutorial1"
9  #define LOGI(...) __android_log_print(ANDROID_LOG_INFO,
10                                LOG_TAG,__VA_ARGS__)
11 #define LOGE(...) __android_log_print(ANDROID_LOG_ERROR,
12                                LOG_TAG,__VA_ARGS__)
13
14 jstring Java_com_android_tutorial1_Tutorial1Activity_foo1(
15        JNIEnv* env, jobject thiz, jstring message) {
16    //To print out a char* we have to convert
17    //the jstring to char*
18    const char *nativeString = (*env)->GetStringUTFChars(env,
19                                message, 0);
20    LOGI("foo1 called! Input parameter: %s", nativeString);
21    //Then we have to release the memory allocated for
22    //the string
23    (*env)->ReleaseStringUTFChars(env, message, nativeString);
24    return (*env)->NewStringUTF(env, "JNI call J2C performed!");
25 }
26
27 void Java_com_android_tutorial1_Tutorial1Activity_foo2(
28        JNIEnv* env, jobject thiz) {
29    LOGI("foo2 called!");
30    //Get class from the calling object
31    jclass clazz = (*env)->GetObjectClass(env, thiz);
32    if (!clazz) {
33       LOGE("callback_handler: failed to get object Class");
34       goto failure;
35    }
36    //Get the methodID from the class which the calling
37    //object belongs
38    jmethodID method = (*env)->GetMethodID(env, clazz,
39                       "foo3Callback", "()V");
40    if (!method) {
41       LOGE("callback_handler: failed to get method ID");
42       goto failure;
43    }
44    //Call the method on the calling object, defined by the
45    //methodID
46    (*env)->CallVoidMethod(env, thiz, method);
47
48    failure: return;
49 }
```

Listing 3: tutorial1.c

The first thing that we can see is the `#include <jni.h>` at line 1: this is the header file - included with the JDK - that maps Java types to their native counterparts (**Table 1**).
There are two functions (line 14 and line 27): their names have a specific format which maps the full name of the "caller" Java class - where the *underline* character replaces the typical Java *dot notation* - and, at the end, there's the name of the function, which corresponds to the `native` method declared into our class.
Let's take `foo1` as example: the method declaration in **Listing 2:1** is natively implemented in **Listing 3:14**. As it's done for the name, the input parameters and the return type have to match as well, therefore we have that the java `String` type is mapped to a native `jstring` type (thanks to the jni.h library that we included some lines above); the same goes for the input (`String/jstring` again), with a peculiarity: native functions mapped to Java methods through JNI paradigm always have two more parameters, a `JNIEnv*` which is a pointer to the current Java environment, and a `jobject` which is a reference to the caller object. Given this, depending on the variety of input(s) and of the output type, we're able to properly create the correct JNI function declaration: from now on, with some attentions, native code can freely run.
In this tutorial I decided to pass a `String` as input to have the opportunity to show how the developer should proceed when she needs to handle variable types that are not primary ones. In this case, we have to explicitly convert the string from `jstring` to a *"native-friendly"* `char*`: if you don't, the error is sometimes not as verbose as you would expect it to be, and it's quite hard to debug this situation, therefore pay attention to the Java/native types mismatch. As the reader can easily understand, we declare a native string, which will save the result of the right-hand side function (**Listing 3:18**) `GetStringUTFChars` which obtains string characters represented in the Unicode format. The third parameter indicates whether we want a copy, or the pointer to the actual java string: in the latter case, the developer must pay attention not to modify the contents of the returned string. This parameter is typically set to NULL, and the JVM will autonomously determine the choice to pick.
We can finally log our parameter as *information*. When the native code finishes using the UTF-8 string obtained through GetStringUTFChars, it calls ReleaseStringUTFChars, thus the memory taken by the UTF-8 string can be freed.
To satisfy the return type declared for this `foo1`, we return a new string. The procedure shown at **Listing 3:24** is specific for generating a jstring object: we'll see further in this article that from the native code we are able to find java classes, instantiate them as jobjects, and call methods on them.

foo2 function represents a very simple example of a JNI callback: typically, whenever we need to callback a Java method implemented into the class that made the call, we go through three main steps:

- get the class (jclass), starting from the object (jobject) which made the call, through the GetObjectClass **Listing 3:31**
- get the method identifier, through the GetMethodID, which performs a lookup for the method in the given class. The lookup is based on the name and type descriptor of the method. If the method does not exist, GetMethodID returns NULL **Listing 3:38**
- call the method, on the caller object, passing the methodID **Listing 3:46**

The code in **Listing 3:27-49** is easy to read, despite for a detail that we are going to analyse: at line 38, the GetMethodID function, takes as input 4 parameters:

- JNIEnv*, the pointer to the Java environment
- jobject, the caller object, that we passed to the native function as input parameter
- const char*, the string which identifies the name of the method to call back
- const char*, the "signature" of the method, called the method descriptor: despite it could seem bizarre, it's very easy to understand. The first part, between the round brackets, represents the input parameters, and their types; the last identifier, after the closed round bracket, represents the return type. In our case, ()V means void foo3Callback(). If our function was, let's say, float foo3Callback(int i) we would write, as its descriptor: (I)F. And, again, with float foo3Callback(int i, float f, int j) we would write (IFI)F. A guide table is given at **Table 1**. We will see how to declare complex descriptors, with strings, objects and arrays further.

Signature	Java Type	Native Type
Z	boolean	jboolean
B	byte	jbyte
C	char	jchar
S	short	jshort
I	int	jint
L	long	jlong
F	float	jfloat
D	double	jdouble
-	void	void

Table 1: Types correspondence Java/JNI and signatures

Signature for object and arrays are:

Signature	Java Type
Lcom/qualified/class;	com-qualified-class
[type	type[]

We still miss the last step, which allows the Java class to load the shared native library: the native code needs to be compiled by the ndk-build command. Before doing this, we have to create the Android.mk into the jni directory. The content of this file, for this tutorial, is shown at **Listing 4**: after defining the name of the module that will be compiled, its source file, and local libs that we want to use, through the include $(BUILD_SHARED_LIBRARY) instruction, we tell the compiler to create the shared object. Doing so, this library will be available to be loaded by Java classes. The use of ndk-build command is highly suggested, since it will generate all the files and folders that our environment needs.

```
1  LOCAL_PATH:= $(call my-dir)
2
3  include $(CLEAR_VARS)
4
5  LOCAL_MODULE := tutorial1
6  LOCAL_CFLAGS := -Werror
7  LOCAL_SRC_FILES := tutorial1.c
8  LOCAL_LDLIBS := -L$(SYSROOT)/usr/lib -llog
9
10 include $(BUILD_SHARED_LIBRARY)
```

Listing 4: Android.mk for tutorial1.c

An example of the activity performing a callback can be found at **Figure 5**: the output visible at the top of the screen comes from the Java method at **Listing 2:5**

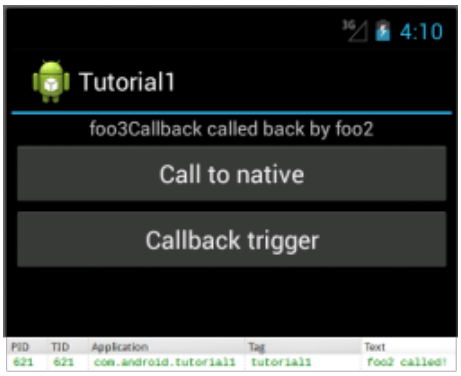

Figure 5: Tutorial1 performing a callback

So far we introduced the main components needed to develop within the Android NDK, and we saw a simple example where an activity calls - and is called back by - a native function.
In the next section, we'll add some features to this first example.

4 Tutorial2

The complete code from this tutorial can be found at `https://github.com/thoeni/ndk-tutorials/tree/master/tutorial2`.
Often we deal with threads, and we want to implement asynchronous calls from native to Java environment. Asynchronous calls are easy to do when the direction is Java to native, while the other way around - implemented through the callback mechanism - isn't that easy to figure out.
This example will consist of basic activity (almost identical to the one we used for *Tutorial1*), that will be able to "run" and "stop" a routine. This routine will consist of a loop of four different callbacks, that will run into a thread: the loop exit condition is based on a flag that will be set and unset by the activity.
In addition to the basic blocks we saw for *Tutorial1* at **Listing 2**, we can find at **Listing 5** the lines relevant to understand how *Tutorial2* performs.

```
1  int int0;
2  float float0;
3  String string0;
4
5  public void onCreate(Bundle savedInstanceState) {
6     // [...]
7     init();
8     handler = new Handler();
9  }
10
11 // [...]
12
13 public void callback1() {
14    System.out.println("callback1 called");
15    handler.post(callback1Thread);
16 }
17
18 Runnable callback1Thread = new Runnable() {
19    @Override
20    public void run() {
21       output.setText("callback 1, no params");
22    }
23 };
24
25 public int callback2(int param0, float param1,
26                      String param2) {
27    System.out.println("callback2 called, params are: "
28       + param0 + " " + param1 + " " + param2);
29    int0 = param0;
30    float0 = param1;
31    string0 = param2;
32    handler.post(callback2Thread);
33    return 0;
34 }
35
36 Runnable callback2Thread = new Runnable() {
37    @Override
38    public void run() {
39       output.setText("callback 2, params are: "
40          + int0 + ", " + float0 + ", " + string0);
41    }
42 };
43
44 public void callback3(String param0) {
45    System.out.println("callback 3, param is: "
46       + param0);
47    string0 = param0;
48    handler.post(callback3Thread);
49 }
50
51 Runnable callback3Thread = new Runnable() {
52    @Override
53    public void run() {
54       output.setText("callback 3, param is: " + string0);
55    }
56 };
57
58 public float callback4(float param0) {
59    System.out.println("callback 4, param is: " + param0);
60    float0 = param0;
61    handler.post(callback4Thread);
62    return param0;
63 }
64
65 Runnable callback4Thread = new Runnable() {
66    @Override
67    public void run() {
68       output.setText("callback 4, param is: " + float0);
69    }
70 };
71
72 // [...]
73
74 public native void init();
75 public native void foo1();
76 public native void foo2();
77
78 static {
79    System.loadLibrary("tutorial2");
80 }
```

Listing 5: Part of Tutorial2Activity.java source code

At **Listing 5:1-3** we declare three global variables where some callback values will be stored, and that will be accessed during some of the callbacks methods execution.
Within the `onCreate` method we call the first out of three native methods, named `init()`: the corresponding native function initialises some parameters and retrieves the various `methodIDs`.
Then callback methods declarations follow, at **Listing 5:13-70**: for each callback method, there's a corresponding `Runnable` object which represents a command that can be executed, and is often used to run code in a different thread. This class declares an `abstract void` method, which therefore is mandatory to implement, and is the `run()` method: within this method we can put the active part of the code that must be executed and, typically, we use this to change the output view and display some values on the screen.
Below (**Listing 5:74-76**) there are the declarations of three `native` functions.
Compared to the example we considered for the *Tutorial1*, this native code is slightly more complex, and uses a thread and some JNI specific types we haven't had the chance to introduce before. To start, here we use the `JNI_OnLoad` function (**Listing 6**), which performs initialization operations for a given native library and returns the JNI version required by the native library. The virtual machine implementation calls `JNI_OnLoad` when the native library is loaded, therefore we can use it to save a reference to the current Virtual Machine which - unlike the case of the `JNIEnv*` that is local to each call - lives along the whole life of the application.

```
1  static JavaVM *gJavaVM;
2
3  jint JNI_OnLoad(JavaVM* vm, void* reserved)
4  {
5    JNIEnv *env;
6    gJavaVM = vm;
7    LOGI("JNI_OnLoad called");
8    if ( (*vm)->GetEnv(vm, (void**) &env, JNI_VERSION_1_4)
```

```
9               != JNI_OK) {
10      LOGE("Failed to get the environment using GetEnv()");
11      return -1;
12    }
13    return JNI_VERSION_1_4;
14  }
```

Listing 6: Part of tutorial2.c - JNI_OnLoad

The `init()` function saves global references to the caller object and, hence, to the caller class the objects belongs to. Besides it saves the references to the `methodIDs` statically declared into the `callback_t` structure (specifically defined for this example).
In **Listing 7** an extract of the structure declaration (**lines:1-15**), the global variables to save the global references to (**lines:17-18**), the same `GetMethodID` implementation we already saw (**line:35**) and the `NewGlobalRef()` function (**lines:26-28**), which creates a new global reference to the object referred to by the second argument. Global references must be explicitly disposed of by calling `DeleteGlobalRef`.
Since `init()` native method is called within the `onCreate`, this whole procedure will be executed the first time we run the activity, and when we launch it again after having destroyed it.
Now we can see what happens when the user calls `foo1`, and starts the testing routine.

```
1   callback_t cb[] = {
2     // cb[0]
3     {
4       "callback1",
5       "()V",
6       JNI_WRAPPER_rVOID,
7     },
8     // cb[1]
9     {
10      "callback2",
11      "(IFLjava/lang/String;)I",
12      JNI_WRAPPER_rINT_p,
13    },
14    //[...]
15  };
16
17  static jobject gObject;
18  static jclass gClass;
19
20  void
21  Java_com_android_tutorial2_Tutorial2Activity_init(
22                              JNIEnv* env, jobject thiz)
23
24    LOGI("init native function called");
25    //[...]
26    gObject = (jobject)(*env)->NewGlobalRef(env, thiz);
27    jclass clazz = (*env)->GetObjectClass(env, thiz);
28    gClass = (jclass)(*env)->NewGlobalRef(env, clazz);
29    //[...]
30    int i = sizeof cb / sizeof cb[0];
31    //[...]
32    while(i--) {
33        LOGI("Method %d is %s with signature %s", i,
34                        cb[i].cbName, cb[i].cbSignature);
35        cb[i].cbMethod = (*env)->GetMethodID(env, clazz,
36                        cb[i].cbName, cb[i].cbSignature);
37    }
38  }
```

Listing 7: Part of tutorial2.c - init()

`foo1` has no parameters, and its return value is void: its main code consists of a call to the function `daemonStart()`, therefore it's useless to show its code, though can be easily found on-line at the repository address. `daemonStart()`, in turn, creates a pthread and exits, and this is traditional C code, so far.
The function that is launched as thread is called `randomCaller()`, and uses the `gJavaVM` variable to retrieve the current `JNIEnv` and send it as input parameter to another function, which is a wrapper that, depending on the input parameters, chooses which method has to be called. A sequence diagram to clarify its working is shown in **Figure 6**. Unlike *Tutorial1*, this example uses a thread, and there's a proper procedure to *attach* the current thread to the JVM, as shown in **Listing 8**.

```
1   void *randomCaller() {
2     flag = 1;
3     JNIEnv *env;
4     int isAttached = 0;
5     int status = (*gJavaVM)->GetEnv(gJavaVM, (void **) &env,
6               JNI_VERSION_1_4);
7     if(status < 0) {
8       LOGE("callback_handler: failed to get JNI environment,
9               assuming native thread");
10      status = (*gJavaVM)->AttachCurrentThread(gJavaVM, &env,
11                                                  NULL);
12      if(status < 0) {
13        LOGE("callback_handler: failed to attach current
14              thread");
15      }
16    isAttached = 1;
17
18    /*
19    * callMethodWrapper(i++)
20    * sleep(2);
21    */
22
23    if(isAttached)
24      (*gJavaVM)->DetachCurrentThread(gJavaVM);
25  }
```

Listing 8: tutorial2.c - thread attachment in randomCaller()

At **Listing 8:5** is called the `GetEnv` function on the global reference of the JVM: this function sets `*env` to `NULL` if the current thread is not attached to the given virtual machine instance, and returns `JNI_EDETACHED` which corresponds to `-2`; if the specified interface is not supported, it sets `*env` to `NULL`, and returns `JNI_EVERSION` which corresponds to `-3`. Otherwise, sets `*env` to the appropriate interface, and returns `JNI_OK`, which corresponds to `0`. Since we know we are running this code within a thread, the status value will be `-2`, and the thread attachment will be attempted (**Listing 8:10**): the `AttachCurrentThread` function sets up the current native thread to run as part of a virtual machine instance. Once a thread is attached to the virtual machine instance, it can then make JNI function calls to perform such tasks as accessing objects and invoking methods. The `DetachCurrentThread` function (**Listing 8:24**) informs a virtual machine instance that the current thread no longer needs to issue JNI function calls, allowing the virtual machine implementation to perform clean-ups and free resources.
How the `switch` condition works inside the `randomCaller` is easy to understand (**Listing 8:18-21**), hence the full code isn't presented in this paper, nonetheless it's important to

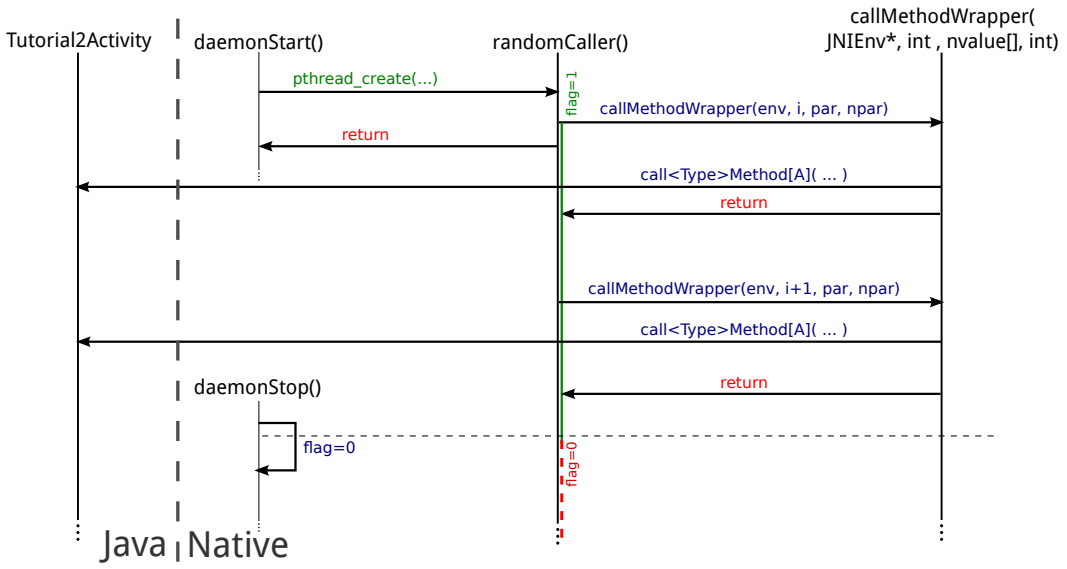

Figure 6: Tutorial2 sequence diagram

stress the peculiarity of a calling method, which is the `call<Type>MethodA()` **Listing 10:24**: to test this cycle, the `randomCaller` defines some variables, and calls the wrapper passing four arguments to it. An example about this can be seen in **Listing 9**: at **line:11** the developer sets as input parameters the env variable, the ordinal number of the chosen callback method (**Listing 7:9-13**), the array of native params (which is a `struct` we defined) and the size of this array.

```
1  nvalue v1, v2, v3, npar[3];
2  v1.type = INT;
3  v1.i = 12;
4  v2.type = FLOAT;
5  v2.f = 2.3;
6  v3.type = STRING;
7  v3.s = "string";
8  npar[0] = v1;
9  npar[1] = v2;
10 npar[2] = v3;
11 callMethodWrapper(env, 1, npar, 3);
```

Listing 9: Part of tutorial2.c - method calls in randomCaller()

The `callMethodWrapper` function is partially shown in **Listing 10**: the `for` at **line:7** maps all the native values to `jvalue[]`, which is a `struct` already defined for the JNI environment. It comes quite useful in this case because we can take advantage of the `call<Type>MethodA` function, where the `A` at the end of its name means that it takes as fourth parameter an `jvalue` array.

```
1  int callMethodWrapper(JNIEnv* env, int mid, nvalue npar[],
2                        int parSize) {
3     jvalue jpar[parSize];
4     //[ ... ]
5     if (parSize > 0) {
6     int i;
7     for (i=0; i<parSize; i++) {
8        switch (npar[i].type) {
9           case INT:
10             jpar[i].i = npar[i].i;
11          break;
12          case FLOAT:
13             jpar[i].f = npar[i].f;
14          break;
15          case STRING:
16             jpar[i].l = (*env)->NewStringUTF(env, npar[i].s);
17          break;
18       }
19    }
20 }
21 switch (cb[mid].jniWrapper) {
22    //[ ... ]
23    case JNI_WRAPPER_rINT_p:
24       (*env)->CallIntMethodA(env, gObject, cb[mid].cbMethod,
25                              jpar);
26    break;
27    //[ ... ]
28 }
```

Listing 10: Part of tutorial2.c - callingMethodWrapper

A screenshot of *Tutorial2* activity and logcat during the execution can be found at **Figure 7**

5 Tutorial3

For this tutorial, we apply all we've seen so far, adding some concepts that can let us taking advantage of the native code through some mechanisms which are typical of Android: we'll introduce a Service, and two different messaging systems, to enable a communication channel between the Activity and the Service.
Doing so, we'll decouple the `View` (the activity) and the `Controller` (the service): the only interface to the Native environment will be the Service, and the Activity will be used only as a "trigger" for Java calls to Native, and as a "display" of calls coming from the Service (and, to it , from the Native). A sample sequence diagram of this tu-


```
dalvikvm          Trying to load lib /data/data/com.android.tutorial2/lib/libtutorial2.so 0x411e29e8
dalvikvm          Added shared lib /data/data/com.android.tutorial2/lib/libtutorial2.so 0x411e29e8
tutorial2         JNI_OnLoad called
tutorial2         init native function called
tutorial2         getting methodIDs of 4 configured callback methods
tutorial2         Method 3 is callback4 with signature (F)F
tutorial2         Method 2 is callback3 with signature (Ljava/lang/String;)V
tutorial2         Method 1 is callback2 with signature (IFLjava/lang/String;)I
tutorial2         Method 0 is callback1 with signature ()V
Choreographer     Skipped 41 frames!  The application may be doing too much work on its main thread.
gralloc_goldfish  Emulator without GPU emulation detected.
tutorial2         daemonStart called!
tutorial2         callback_handler: failed to get JNI environment, assuming native thread
tutorial2         callStaticMethodWrapper called for cb[0] method callback1,signature ()V,switch case 0
System.out        callback1 called
tutorial2         callStaticMethodWrapper called for cb[1] method callback2,signature (IFLjava/lang/String;)
System.out        callback2 called, params are: 12 2.3 string
tutorial2         daemonStop called!
```

Figure 7: Tutorial2 running example

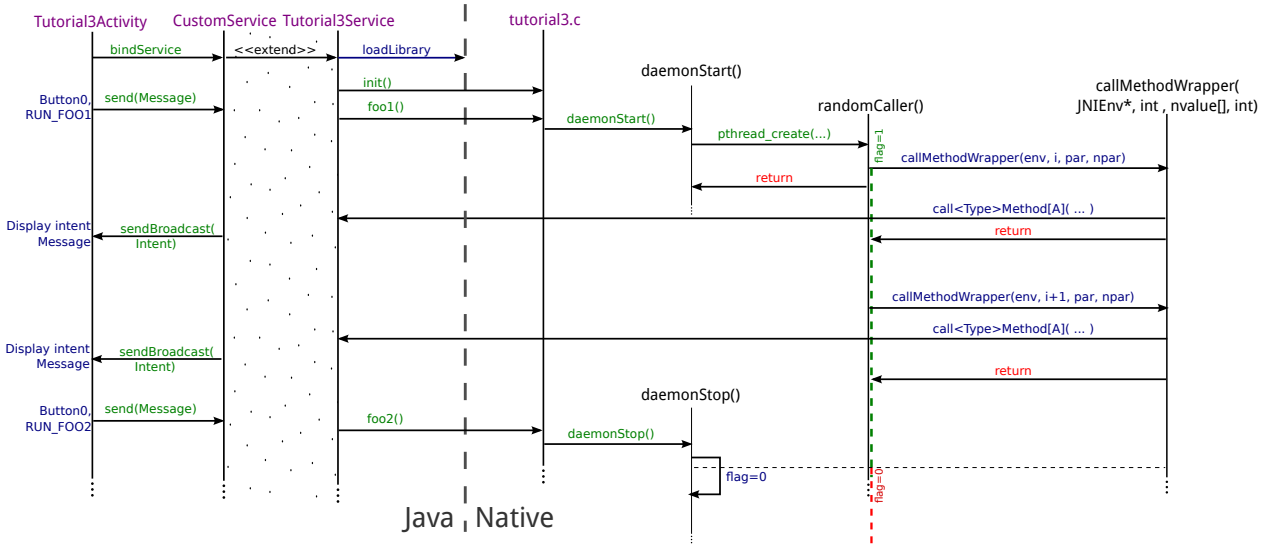

Figure 8: Tutorial3 sequence diagram

torial can be found at **Figure 8**: we'll refer to this figure often, to understand the structure of this application.
The activity, as one of its first tasks, will bind the service: this will also implicitly invoke the startService method, therefore this will be executed into the "onStart()" method.
The service we are going to bind, is the "custom" one, which extends our own service, called *Tutorial3Service* that loads the shared native library (in the same say we say for tutorial1 and tutorial2).
When the service is created, typically, it calls an `"init()"` function, which initialises the shared library performing some actions, that we consider will be described into the Service class.
Generally speaking, this tutorial, except for the service and the activity-service communication channel, it's exactly the same as the previous one. The native code is the same. This being said, let's analyse the high level part, how to implement a service, and hot to enable the communication between the activity and the service.
Within the `AndroidManifest.xml` file, we have to explicitly declare that the application we are going to build will interface itself with a service , and we do this by adding the **Listing 11:8**:

```
1  package com.android.tutorial3;
2
3  <application android:label="Tutorial3"
4                  android:debuggable="true">
5          [...]
6          <activity ... >
7          [...]
8          </activity>
9          <service android:name=".CustomService"></service>
10     </application>
```

Listing 11: AndroidManifest.xml

Typically a Service is implemented to run in background

mode, and to perform operations that don't need to appear on the foreground; a Service is run into the same thread as the Application one. After binding an Activity to a Service, the `onCreate()` and the `onStartCommand()` are called, the service is launched, and we're able to communicate with it. Let's see how to set the communication channel between the activity and the service up. Among the possible available methods, we chose two of them:

- the `Messenger` class to handle the Activity to Service communication
- the `Intent` class, and `sendBroadcast()` method to support the Service to Activity communication

In **Listing 12** the code to be added to implement the server side `Messenger` receiver and handler is shown.

```
1  //** Available messages to the Service ***
2  static final int MSG_RUN_FOO1= 1;
3  static final int MSG_RUN_FOO2 = 2;
4  //****************************************
5
6  /* *
7   * Instantiate the target - to be sent to clients - to
8   * communicate with this instance of Service
9   */
10 final Messenger mMessenger = new Messenger(
11                       new IncomingHandler());
12
13 @Override
14 public IBinder onBind(Intent intent) {
15        return mMessenger.getBinder();
16 }
17
18 /**
19  * Handler of incoming messages from clients.
20  */
21 class IncomingHandler extends Handler {
22        @Override
23        public void handleMessage(Message msg) {
24                switch (msg.what) {
25                case MSG_RUN_FOO1:
26                        Toast.makeText(getApplicationContext(),
27                "Calling foo1()!", Toast.LENGTH_SHORT).show();
28                        foo1();
29                        break;
30                case MSG_RUN_FOO2:
31                        Toast.makeText(getApplicationContext(),
32                "Calling foo2()!", Toast.LENGTH_SHORT).show();
33                        foo2();
34                        break;
35                default:
36                        super.handleMessage(msg);
37                }
38        }
39 }
```

Listing 12: Tutorial3Service.java - message handling

The `nMessenger` is the object the service will share with the activity. In **Listing 13** the correspondent code from the `Activity` class, which allows it to interact with the service thanks to the `nMessenger` object.

```
1  /**
2   * Messenger for communicating with the service.
3   */
4  Messenger mService = null;
5
6  /**
7   * Flag indicating whether we have called bind on
8   * the service.
9   */
10 boolean mBound;
11
12 /**
13  * Class for interacting with the main interface
14  * of the service.
15  */
16 private final ServiceConnection mConnection =
17               new ServiceConnection() {
18        @Override
19        public void onServiceConnected(
20                        ComponentName className,
21                        IBinder service) {
22                mService = new Messenger(service);
23                mBound = true;
24        }
25        @Override
26        public void onServiceDisconnected(
27                        ComponentName className) {
28                mService = null;
29                mBound = false;
30        }
31 };
32
33 /* *
34  * Buttons interacts with the Service through
35  * Messenger paradigm.
36  */
37
38 public void button0(View v) {
39        if (!mBound) return;
40        Message msg = Message.obtain(
41        null, CustomService.MSG_RUN_FOO1, 0, 0);
42        try {
43                mService.send(msg);
44        } catch (RemoteException e) {
45                e.printStackTrace();
46        }
47 }
48
49 public void button1(View v) {
50        if (!mBound) return;
51        Message msg = Message.obtain(
52        null, CustomService.MSG_RUN_FOO2, 0, 0);
53        try {
54                mService.send(msg);
55        } catch (RemoteException e) {
56                e.printStackTrace();
57        }
58 }
```

Listing 13: Tutorial3Activity.java - message handling

As it's easy to infer, when the service is connected - within the `mConnection` - a new `Messenger` object is created (giving it as input the `IBinder service` interface (**Listing 13:22**). On this object, the activity will be able to call the `send(msg)` method, to send simple messages to the service beneath. Anytime the activity will send a message, the `handleMessage()` method will process the `what` attribute, and execute the proper branch of the `switch`. The main difference from the previous example lays in the service, which acts as a controller between activity and native code, while in **Section 4** we had the activity loading and calling straight to the native library. From now on, the activity is ready to communicate with the Service

So far we've seen how the activity can communicate with the service; the vice versa has been implemented - in this tutorial - taking advantage of the possibility into Android platform to broadcast some messages, which, in this case, are called *Intents*. As done before, let's see the pieces of code which implement this functionality into the service class, and into the activity class, which are shown, respectively, in **Listing 14** and **Listing 15**.

```
1  public void _callback1() {
2          callback1();
3          Intent intent = new Intent(
4                  "com.android.tutorial3.TUTORIAL_3_INTENT");
5          intent.putExtra("CALLBACK_EXEC",1);
6          sendBroadcast(intent);
7  }
```

Listing 14: Tutorial3Service.java - intent broadcasting

The Intent is created passing to the constructor a `String` which will be used to identify it, and listen to this specific intent, when broadcast; an extra parameter is put (in our example we use it to identify the callback method integer identifier), and the intent is then sent. **Listing 14:4-6**

```
1  /**
2   * IntentFilter used to receive broadcast intents launched
3   * by service
4   */
5  IntentFilter receiverFilter = new IntentFilter ();
6
7  @Override
8  public void onCreate(Bundle savedInstanceState) {
9          //[...]
10         receiverFilter.addAction(
11                 "com.android.tutorial3.TUTORIAL_3_INTENT");
12         registerReceiver(receiver, receiverFilter);
13 }
14
15 /**
16  * Broadcast receiver: catches messages sent by the
17  * Tutorial3Service
18  */
19 BroadcastReceiver receiver = new BroadcastReceiver() {
20         @Override
21         public void onReceive(
22           Context context, Intent intent) {
23           int running = intent.getIntExtra("CALLBACK_EXEC", 0);
24             switch (running) {
25               case 1:
26                 Toast.makeText(
27                   getApplicationContext(),"Exec. callback1",
28                   Toast.LENGTH_SHORT).show();
29               break;
30               //[...]
31             }
32         }
33 };
```

Listing 15: Tutorial3Activity.java - intent catching

An intent filter can be declared into the AndroidManifest or, at runtime, into the Activity. In our example, we chose the second option. At **Listing 15:10-12** the `IntentFilter` is initialised and it's registered to the `BroadcastReceiver` which is declared some lines beneath, at **Listing 15:19**. By overriding the `onReceive` method, we put the code which is run when the specific intent is caught. In this example we just throw a toast message on the screen.
The native code is the same we already discussed in **Section 4**.

6 Java object handling within native code

It's possible that you would need to pass to the Java world a reference to an object that doesn't exist yet.
We'll analyse how it's possible to find a class, create an instance of this class, and call some methods on it.

Let's suppose we created a class, called Param, within the `tutorial2` package, which looks like the one in **Listing 16**

```
1  package com.android.tutorial2;
2
3  public class Param {
4          private int[] iParams;
5          private float[] fParams;
6          private String[] sParams;
7
8          public Param(int[] iP, float[] fP, String[] sP) {
9                  setiParams(iP);
10                 setfParams(fP);
11                 setsParams(sP);
12         }
13
14         public int[] getiParams() {
15                 return iParams;
16         }
17         public void setiParams(int[] iParams) {
18                 this.iParams = iParams;
19         }
20         public String[] getsParams() {
21                 return sParams;
22         }
23         public void setsParams(String[] sParams) {
24                 this.sParams = sParams;
25         }
26
27         public float[] getfParams() {
28                 return fParams;
29         }
30         public void setfParams(float[] fParams) {
31                 this.fParams = fParams;
32         }
33 }
```

Listing 16: Java class Param

Suppose that we need to create a Param object, assign some values to its fields, and pass its reference to a Java method through the callback mechanism. Let's see the main steps to achieve this aim.
To begin, we need to get a reference to the `Param` class, and we do this as follow:

```
1  //Find the "Param" Class
2  cls = (*env)->FindClass(env, "com/android/tutorial2/Param");
```

We can then create java arrays and, depending on the type of array we are going to create, we have to use a different constructor function, for instance, to create a `jint` array:

```
1  int isize = 3;
2  jint iparams[isize] = {0, 1, 2};
3  //###Create a new Array of integers###
4  iarr = (*env)->NewIntArray(env, isize);
5  //Fill the array with the integer input parameter
6  (*env)->SetIntArrayRegion(env, iarr, 0, isize, iparams);
```

Now the object `iarr` is a Java array of integers.
The same thing can be done for floats, as follows:

```
1  int fsize = 2;
2  jfloat fparams[fsize] = {1.2, 3.2};
3  //###Create a new Array of floats###
4  farr = (*env)->NewIntArray(env, fsize);
5  //Fill the array with the float input parameter
6  (*env)->SetFloatArrayRegion(env, farr, 0, fsize, fparams);
```

Now, for the `String` we don't have any ready-to-use function to create an array of `String` objects, therefore we can use the more general `NewObjectArray` function, as fol-

lows:

```
int ssize = 2;
char* sparams[ssize] = {"ab","cd"};
sarr = (*env)->NewObjectArray(
               env, ssize, (*env)->FindClass(
                              env, "java/lang/String"), NULL);
for (i=0; i<ssize; i++)
   (*env)->SetObjectArrayElement(
               env, sarr, i, (*env)->NewStringUTF(
                              env, sparams[i]));
```

Now, we have three Java arrays correctly initialised: we can create an object of the `Param` class, and use its constructor (**Listing 16:8**) to initialise it with these newly generated arrays:

```
jmethodID constructor;
//Find the constructor of the Param object, which takes as
//parameter an int array and a float array
constructor = (*env)->GetMethodID(env, cls, "<init>",
               "([I[F[Ljava/lang/String;)V");
//Create the Object with its constructor, and the arrays as
//parameters
obj = (*env)->NewObject(
               env, cls, constructor, iarr, farr, sarr);
//Call the callback method passing the object as input
//parameter
(*env)->CallStaticVoidMethod(
               env, gClass, cb[id].cbMethod, obj);
```

As we saw in **Section 3**, the following signature `"([I[F[Ljava/lang/String;)V"` indicates a method - in this case `<init>` identifies the constructor method - which has a `void` return, and has input structured like:

- `[I` -> array of integers: int[]
- `[F` -> array of floats: float[]
- `[Ljava/lang/String;` -> array of objects `L` indicates object, of type `java/lang/String`: String[]

Therefore this signature means `void <init> (int[], float[], String[])`, which is exactly how the `Param` constructor is defined at **Listing 16:8**.

7 Conclusions

This report and all the source code are publicly available through the git repository at `https://github.com/thoeni/ndk-tutorials` or at `https://bitbucket.org/atroina/ndk-tutorials`.

References

[1] Liang, S.: The Java Native Interface. Addison-Wesley (1999)

[2] Gargenta, A.: Jni reference example (Oct. 2012)

[3] Gargenta, M.: Learning Android. O'Reilly (2011)

[4] Mednieks, Dornin, Meike, Nakamura: Programming Android. O'Reilly (2011)