POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Specialistica in Ingegneria Informatica

ANDROID™ RUN-TIME RESOURCE MANAGEMENT

JNI BASED INTEGRATION OF THE BARBEQUERTRM FRAMEWORK

Autore:
Antonio Troina
matr. 708267

Relatore:
Prof. William Fornaciari
Correlatore:
Ing. Patrick Bellasi

Anno Accademico 2011-2012

Contents

Bibliography **iii**

1 Overview **3**

1.1 Typical internal flow 4

1.2 Barbeque Run-Time Resource Manager 6

2 Prior Art **7**

2.1 Barbeque Open Source Project 7

2.1.1 Abstract Execution Model (AEM) 8

2.2 OpenCL 11

2.2.1 Platform model 13

2.2.2 Execution model 14

2.2.3 Context 14

2.3 Java Native Interface (JNI) 15

2.3.1 Typical JNI use 16

2.3.2 JNI under Android: NDK 17

3 Run-Time Resource Management of Android Applications **19**

3.1 RTLib JNI bridge 23

3.2 Software design 26

3.2.1 BbqueService 26

3.2.2 Activity: common aspects 31

3.3 Testing applications 32

3.3.1 BOSP testing application 33

3.3.2 STHorm Face Detection 38

4 Experimental results **41**

4.1 Experimental setup 41

4.2 Measures and considerations 42

4.2.1 Native response time 42

4.2.2 Messaging system overhead and performances 45

4.2.3 STHormFaceDetection Application: performance profiling . . 49

5 Conclusions **51**

A Tutorials **53**

A.1 Tutorial1 53

A.2 Tutorial2 59

A.3 Tutorial3 66

A.4 Java object handling within native code 73

List of Figures

1.1 Generic stack of processing execution 4

2.1 Abstract Execution Model 9
2.2 Platform model 13
2.3 JNI flow 16

3.1 System block diagram 21
3.2 Call flow MSG_START 22
3.3 Bound service life cycle 27
3.4 Barbeque testing application screenshot 35
3.5 STHorm Face Detection activity - Sequence diagram 38

4.1 `isRegistered()` execution stack - traceview 43
4.2 `create()` execution stack - traceview 44
4.3 `start()` execution stack - traceview 46
4.4 Performance of the DecodeYUV420SP/ FDetectRun wrt `onRun` . . . 49

A.1 Tutorial1 performing a callback 58
A.2 Tutorial2 sequence diagram 63
A.3 Tutorial2 running example 65
A.4 Tutorial3 sequence diagram 67
A.5 Messenger class connected to Button0 73
A.6 Binder process handling the Message 73
A.7 IncomingHandler asked to call foo1 ()V on Tutorial3Service 74
A.8 Thread triggered by service performs _callback1 ()V 74

To my parents.

All you can do is play along at life,
and hope that sometimes you get it right.

Dexter Morgan

Chapter 1

Overview

"These aren't the droids you're looking for."

Obi-Wan Kenobi

Nowadays electronic devices and, more generally, embedded systems surround us anywhere: they can be found in mobile phones, cars, washing machines, airplanes. It is estimated that the 99% of the worldwide production of processors, today, is used in these systems. Despite they do not look like proper computers, they can hide tens or thousands of microprocessors, running tens or millions lines of program code. A common aspect of these systems lies in their time constraints: they must respond to inputs in real time, and they're known as real time systems. These systems are typically powered by ad-hoc operating systems, known as RTOS (which stands for Real-Time Operating Systems), that are characterized by a very small footprint, and performance-oriented architectures. During the last years, a young and new actor came to tread the boards and joined the *embedded* world, arising the curiosity of many developers: Android. This operating system (built by Google^{TM} within the AOSP - Android Open Source Project), which benefits from a huge and active community of developers, has succeeded in becoming one of the top mobile platforms on the market in a very short time-span. Despite not being as "lightweight and small footprint"-centric as *Real Time Operating Systems*, Android still offers a lot of opportunities to Embedded Developers: first of all, a very intuitive graphic interface. While a large number of embedded devices have little to no human interface, a substantial number of devices which would traditionally be considered "embedded" do have user interfaces. In this field, Android

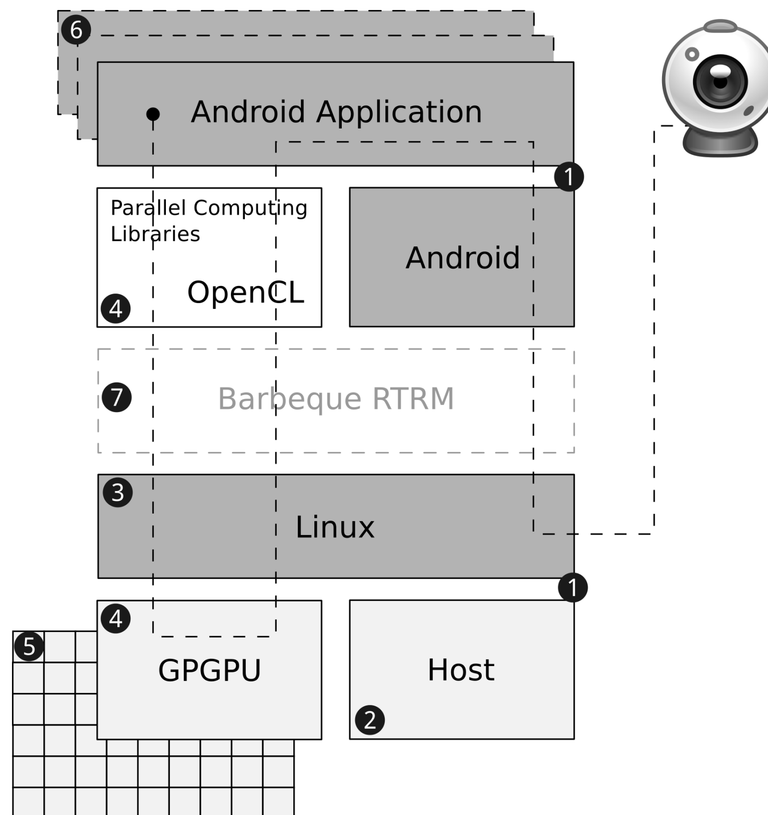


Figure 1.1: Generic stack of processing execution

comes as a bridge connecting the users that are already familiar with its interface, to embedded systems they need to interact with. Despite in the past common GUIs were window-centric and desktop-like, after iOS and Android birth the way people interact with mobile instruments and devices has changed, towards a more touch-based experience. This change, combined with Android's open source licensing, has been a terrific mix that increased embedded developers' interest in Android open source project.

1.1 Typical internal flow

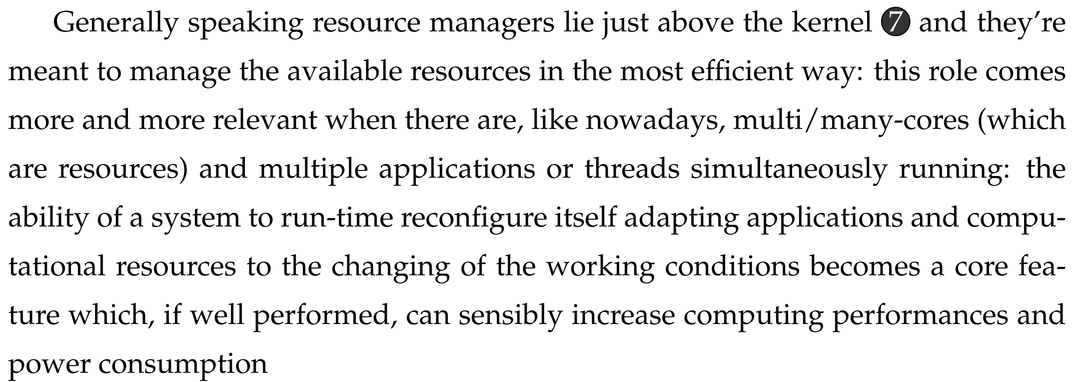
Android OS lies on a properly forked and customized Linux kernel. In Figure 1.1 a very simple idea of the stack, and a possible processing flow is shown, and explained below.

To better put this example into context, let's consider a potential surveillance application, which runs on an Android device. Android applications are written in Java programming language, and use the API offered by the operative system to ex-

ecute their tasks, such as interacting with the UI, or acquiring frames from the camera. ❶ The kernel is - among other things - responsible of the hardware control, and of the basic management of resources: it allows the native code and the Java code interacting with the device peripherals, the webcam, in our case. ❸ Android kernel is quite different from the "vanilla" kernel in fact, whereas the kernel shipped by a Linux distribution can easily be replaced by a kernel from *kernel.org* with little to no impact to the rest of the distribution's components, Android's user-space components will simply not work unless they're running on an *"Androidized"* kernel. Each computing cycle is processed by the device CPU, which is identified with the name of Host. ❷ These CPUs are typically single, dual or, more recently, quad core units (usually identified as multi-core architectures): everything processed by the device, an audio or video stream, a picture, a view or a gesture, is the result of one to thousands of CPU computing cycles. Lately CPUs have been assigned an assistant to perform some specific and performance-oriented computations: GPGPUs. ❹ The well known GPU acronym stands for Graphic Processing Unit, while the GPGPU acronym stands for General Purpose computing on Graphic Processing Unit: this means that the powerful capabilities of GPUs can be exploited to perform - usually parallelized - general purpose computing. GPGPUs find many applications in the most different fields, from high performance computing and grid computing, to physics simulations, to cryptographic computations, to audio/video DSP, FFT, and so on. It's easy to understand how versatile GPGPUs can be, and how their use can significantly improve CPUs performances. The currently dominant open GPGPU computing language is OpenCL which provides parallel computing, using task-based and data-based parallelism. More on OpenCL will be discussed in Section 2.2. As mentioned above, OpenCL can be used to give an application access to GPU for non graphical computing. Now, imagine that it's possible to use not one general purpose processing unit, but much more than that: in fact, unlike for CPUs, where the term *multi-core* is used, when we speak about GPGPUs, we're increasingly using the term *many-core*, where the number of cores is today around 64. ❺ Connecting the dots backwards, it's now trivial understanding how the surveillance application can benefit from this use of a specific processing unit: while the CPU works and handles its own tasks, each frame from the camera can be processed simultaneously by a GPGPU, performing, for instance, some face detection algorithms. In addition, what if instead of just one application, we had more applications let's say, one for each camera, simultaneously running? ❻ How can the

system properly exploit all these resources optimizing their use?

1.2 Barbeque Run-Time Resource Manager



Generally speaking resource managers lie just above the kernel ❼ and they're meant to manage the available resources in the most efficient way: this role comes more and more relevant when there are, like nowadays, multi/many-cores (which are resources) and multiple applications or threads simultaneously running: the ability of a system to run-time reconfigure itself adapting applications and computational resources to the changing of the working conditions becomes a core feature which, if well performed, can sensibly increase computing performances and power consumption
Within *Politecnico di Milano* the Barbeque Open Source Project [11] intends to give an answer to the run-time resource management matter with particular attention to identify the optimal trade-off between the Quality-of-Service (QoS) requirements of the applications and the time varying resources availability. A detailed insight about Barbeque will be given in Section 2.1.
One of the features that sets Barbeque apart from other RTRM, and which is one of the key concepts of this work, is the portability. As stated above, lately Android attracted the attention of the embedded systems developers, and Barbeque was deployed to run under this operating system as well but there hasn't been, so far, any support to make it directly accessible to Java applications. Barbeque is written in C/C++ language, while the high-level tier of Android, which includes the application one, is written in Java language: this work aims to fill the gap in the between, taking advantage of the Android Native Development Kit (NDK) and of the Java Native Interface (JNI). More information about this can be found in the next chapter.

Chapter 2

Prior Art

"The best of prophets of the future is the past."

Lord Byron

2.1 Barbeque Open Source Project

As briefly mentioned in section 1.2 the Barbeque Open Source Project (BOSP) is proposed as a portable and extensible framework for run-time resource management [7]. It supports both homogeneous and heterogeneous platforms: in homogeneous platforms, resources of the same type provide the same capabilities, and we can have, for instance, a set of elements equipped with the same architecture and performances as well. In the latter case, the heterogeneous one, platforms can provide different and specialized computing architectures: during the past few years, heterogeneous computers composed of CPUs and GPUs have revolutionized computing. By matching different parts of a workload to the most suitable processor, tremendous performance gains have been achieved.
Much of this revolution has been driven by the emergence of many-core processors such as GPUs. For example, it is now possible to buy a graphics card that can execute more than a trillion floating point operations per second (teraflops). These GPUs were designed to render beautiful images, but for the right workloads, they can also be used as high-performance computing engines for applications from scientific computing to augmented reality.
An example can be a device which embeds in addition to its main CPU (possibly

multi-core) an hardware accelerator, e.g. a GPU, to process a specific a computing intensive stream of data. In a wider vision, since we saw that GPGPUs are being used always more frequently as general purpose processing units, to perform complex computations not strictly related to graphical ones, this is the situation that is going to be the main scenario in the next years. Being able to manage resources among homogeneous as well as heterogeneous platforms is a big strength of BOSP approach, given that it's currently one of the few, if not the only one, to do so. Furthermore, another peculiarity of Barbeque is its hybrid solution between the centralized and the distributed resource allocation: initially a centralized policy is defined, and it's then refined in a distributed hierarchical manner.
To have an application which is able to adapt and reconfigure itself at run-time, it must be able to be executed according to several configurations, as resource usage levels. Each configuration is called Application Working Mode (AWM) and must be provided by the application developers. The AWM will be described by an XML file, named *Recipe*. Depending on the AWM which has been assigned to it, the application will be able to choose the best Operative Point (OP) to match the QoS for the end user. OPs are a collection of application specific parameters, bounded to each AWM. A single AWM could support multiple OPs, e.g. the same kind and amount of computational resources could accommodate multiple values of application specific parameters.
As can be easily guessed, the applications play an active role on the self-adaptiveness of the system. To support these activities a Run-Time Library (*RTLib*) is given, which supports the application by providing a rich set of features related to the interaction between the application and the framework, as well as supporting the application specific run-time management activities.

2.1.1 Abstract Execution Model (AEM)

To simplify the coding, the RTLib provides the Abstract Execution Model (AEM). Basically it is defined via a callback based API: the developer just needs to implement the application specific logic into the body of few methods. AEM could be represented by a generic *state machine* and specialized for each specific stream processing application, by simply defining the operations associated to each different state, as shown in Figure 2.1.

The AEM API provides the application developers an event-based programming model, where events are generated by the library, according to the previously

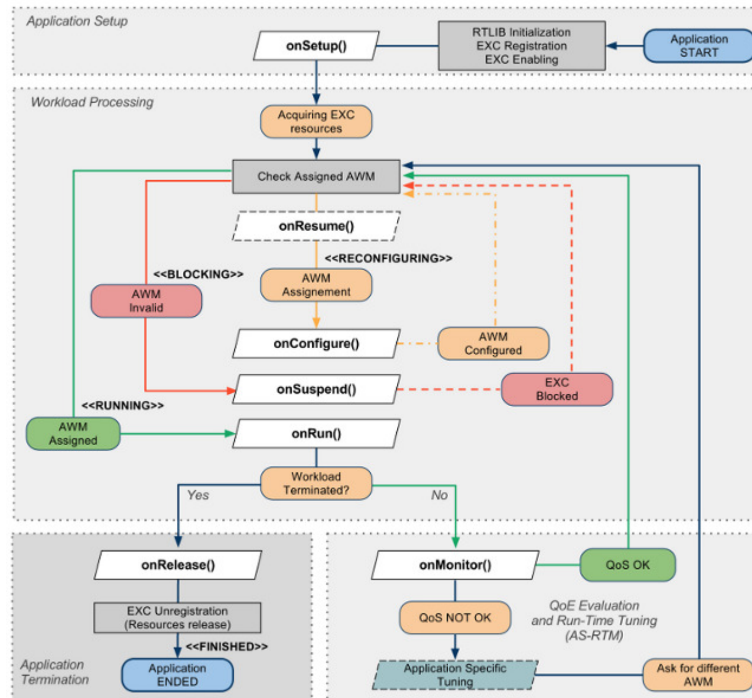


Figure 2.1: Abstract Execution Model

defined execution model, and managed by a set of application defined call-back methods. This means that the integration of a new application requires the developer just to implement the application specific logic into the body of few methods, described below.

The state machine in Figure 2.1 is implemented by the `bbque::rtlib::BbqueEXC` base class, which defines also a default implementation for all the exposed call-back methods (white parallelograms, methods with the *on* prefix). Briefly, the main call-back methods are:

- `onSetup`: This is the application defined call-back method to host all the EXC initialization code. This method will be called by the base class right after the constructor and it's the right place to host all the code to prepare the ground for the stream processing, e.g. opening input and output channels, setup internal data structures for the processing cycle to start.

- `onConfigure`: This is the application defined call-back method to host the required configuration code to switch to a new assigned AWM. The currently assigned working mode is defined by its ID with reference to the IDs defined by the Recipe specified at the EXC instantiation time. It is up to the appli-

cation developer to know the proper mapping between an AWM ID and the corresponding application parameters

- `onSuspend`: If the execution of the EXC should be suspended, for example because there are temporary no resources available for it, than this method is called back right after the completion of the current processing cycle.
  This method is intended to host all the necessary code required to keep safe all the resources allocated on the accelerator in anticipation of an imminent suspension of the EXC. Once the execution will be resumed the corresponding onResume method is called.

- `onResume`: This method is intended to host all the necessary code required to resume the execution of a previously suspended EXC.

- `onRun`: This is one of the main call-back methods which is required to be coded into an application. It is intended to host all the code required to actually process a cycle of the stream processing application, e.g. decoding a single video frame.
  The amount of data to be processed by a single call of this method it is not technically upper bounded. However, it is worth to consider its execution time and the effects on the latencies forced on the run-time management of the whole system.
  Once there are not more data to be processed, this method should return `RTLIB_EXC_WORKLOAD_NONE` to actually terminate the processing cycle. In this case the next onMonitor will not be called and the execution will continue with the onRelease. The BarbequeRTRM does its best in avoiding to interrupt an application while it is executing this method. However, in case the latency introduced by a single call of this method should not be compliant with the run-time management goals (e.g. scheduling a just started high-priority application) the RTRM has the capability to force a termination of this method, i.e. eventually also by kill a ”not responding” application. The code to monitor execution performances of a single call of this method should be better placed into the onMonitor call-back.

- `onMonitor`: This method is called right after each execution of `onRun`, thus this is the most suitable place for all the run-time monitoring code. The application developer could exploit this method to implement an application

specific run-time management policy, e.g. to tune some application specific parameters based on the behaviors obtained during the previous cycle execution.

- `onRelease`: This is the application defined call-back method to host all the EXC shutdown code. This method will be called by the base class right before the destructor and it's the right place to host all the code to clean-up everything since the stream processing application is going to be terminated, e.g. closing input and output channels, release internal data structures.

Considering this short overall view of the AEM, it should be clear that the main purpose of the AEM API is to relieve the application developer from the cumbersome code required to make an application run-time tunable. This complexity is factorized based on a generic stream processing work flow and completely masked to the actual application by a set of well defined call-back methods.
As a final remark, the run-time management solution proposed by the *BarbequeRTRM* framework is based on the exploitation of a hierarchical control. A system-wide run-time resource manager is in charge to partition the available resources, among all the running application, while each application is in charge of fine tuning itself. This application specific tuning is represented by the code which corresponds to the region in the bottom right dashed square of Figure 2.1. Here is where the Quality-of-Experience (QoE) evaluation and run-time tuning takes place.

2.2 OpenCL

Speaking of heterogeneous systems, we discussed how these platforms can perform tremendously good, if properly exploited.
A natural question is why these many-core processors are so fast compared to traditional single core CPUs.

- The fundamental driving force is innovative parallel hardware.
  Many-core processors organize their - billions of - transistors into many parallel processors, consisting of hundreds of floating point units

- Another important reason for their speed advantage is new parallel software.

Computing systems come out of the - hopefully perfect - combination of hardware and software: without a proper software which can exploit at best all the available

resources, state-of-the-art hardware can be useless. Parallel hardware delivers performance by running multiple operations at the same time. To be useful, parallel hardware needs software that executes as multiple streams of operations running at the same time; in other words, you need parallel software.
C language nicely abstracts a sequential computer, but to fully exploit heterogeneous computers, new programming models came out, which can abstract a modern parallel computer: typically techniques established so far in graphics can be used as a guide towards this path.
And here we come with *OpenCL*: few lines of story behind this programming language. Going back to some years ago, a project named *Brook for GPU* took life at Stanford University: the basic idea behind *Brook* was to treat GPU as a data-parallel processor. Brook was built as a proof of concept, nevertheless Ian Buck, a graduate student at Stanford, went on to *NVIDIA* to develop *CUDA*, an extension of Brook, which introduced many improvements to its predecessor as, for instance, the concept of cooperating thread arrays, or thread blocks.

OpenCL (Open Computing Language) - born in 2008 - provides a logical extension of the core ideas from GPU Computing, the era of ubiquitous heterogeneous parallel computing. OpenCL has been carefully designed by the Khronos Group with input from many vendors and software experts. OpenCL benefits from the experience gained using CUDA in creating a software standard that can be implemented by many vendors. OpenCL implementations run now on widely used hardware, including CPUs and GPUs from NVIDIA, AMD, and Intel, as well as platforms based on DSPs and FPGAs.
With OpenCL, you can write a single program that can run on a wide range of systems, from cell phones, to laptops, to nodes in massive super-computers. No other parallel programming standard has such a wide reach. This is one of the reasons why OpenCL is so important.

OpenCL was defined with two different programming models in mind: task parallelism and data parallelism, thus we can even think in terms of a hybrid model: tasks that contain data parallelism.

- ❏ In a data-parallel programming model, programmers think of their problems in terms of collections of data elements that can be updated concurrently. The parallelism is expressed by concurrently applying the same stream of instructions (a task) to each data element. The parallelism is in the data.

- In a task-parallel programming model, programmers directly define and manipulate concurrent tasks. Problems are decomposed into tasks that can run concurrently, which are then mapped onto processing elements (PEs) of a parallel computer for execution. This is easiest when the tasks are completely independent, but this programming model is also used with tasks that share data. The computation with a set of tasks is completed when the last task is done.

2.2.1 Platform model

The OpenCL platform model defines a high-level representation of any heterogeneous platform used with OpenCL, and it's shown in Figure 2.2

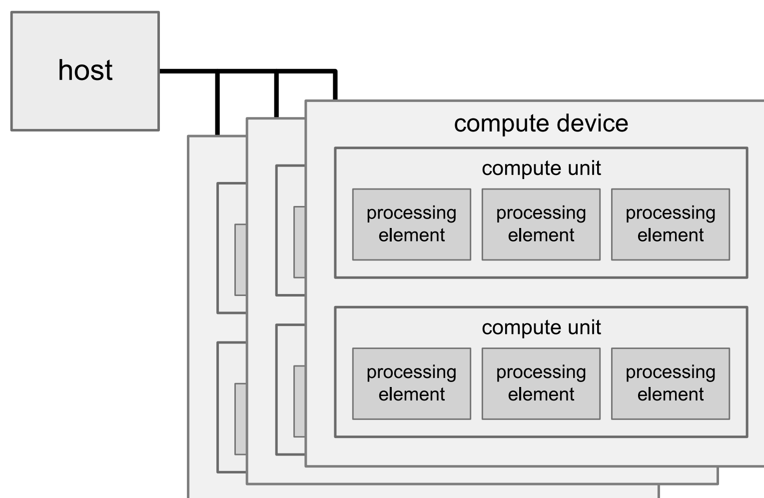


Figure 2.2: Platform model

An OpenCL platform always includes a single *Host* which interacts with the environment external to the OpenCL program, and is connected to one or more *OpenCL devices*. The device - can be a CPU, a GPU, a DSP... - is where the streams of instructions (called *kernels*) are actually executed. The OpenCL devices are further divided into *compute units* which are further divided into one or more *processing elements (PEs)*, where the actual computations really are performed.

2.2.2 Execution model

An OpenCL application consists of two distinct parts: the *host program* and a collection of one or more *kernels*. The OpenCL execution model defines how an OpenCL application maps onto processing elements, memory regions, and the host. The *host program* runs on the host, the kernels execute on the OpenCL devices. They do the real work of an OpenCL application. Kernels are typically simple functions that transform input memory objects into output memory objects. OpenCL defines two types of kernels:

- OpenCL kernels: functions written with the OpenCL C programming language and compiled with the OpenCL compiler. All OpenCL implementations must support OpenCL kernels.
- Native kernels: functions created outside of OpenCL and accessed within OpenCL through a function pointer. These functions could be, for example, functions defined in the host source code or exported from a specialized library.

2.2.3 Context

The computational work of an OpenCL application takes place on the OpenCL devices. The host, however, plays a very important role in the OpenCL application. It is on the host where the kernels are defined. The host establishes the context for the kernels. As the name implies, the context defines the environment within which the kernels are defined and execute. To be more precise, we define the context in terms of the following resources:

- Devices: the collection of OpenCL devices to be used by the host
- Kernels: the OpenCL functions that run on OpenCL devices
- Program objects: the program source code and executables that implement the kernels. Program objects are built at runtime.
- Memory objects: a set of objects in memory that are visible to OpenCL devices and contain values that can be operated on by instances of a kernel. These are explicitly defined on the host and explicitly moved between the host and the OpenCL devices.

2.3 Java Native Interface (JNI)

As mentioned in Chapter 1 the element which acts as a bridge between the Java and the Native world, is JNI, initially released in early 1997. With JNI, the developer can achieve two main goals: reusing her native code within a Java environment, and optimizing 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).
The JNI is a powerful feature that allows you to take advantage of the Java platform, but still utilize code written in other languages. As a part of the Java virtual machine implementation, the JNI is a *two-way* interface that allows Java applications to invoke native code and vice versa.
The JNI is designed to handle situations where you need to combine Java applications with native code, and it can support two types of native code: native *libraries* and native *applications*.

- ❏ JNI can be used to write *native methods* that allow Java applications to call functions implemented in native libraries: Java applications call native methods in the same way that they call methods implemented in the Java programming language. Behind the scenes, however, native methods are implemented in another language and reside in native libraries.
- ❏ JNI supports an invocation interface that allows you to embed a Java virtual machine implementation into native applications. Native applications can link with a native library that implements the Java virtual machine, and then use the invocation interface to execute software components written in the Java programming language. Part of this feature can be seen as the use of the Java methods callbacks that will be illustrated in the next chapters.

JNI is an interface that can be supported by all Java virtual machine implementations on a wide variety of host environments. With the JNI:

- ❏ Each virtual machine implementor can support a larger body of native code.
- ❏ Development tool vendors do not have to deal with different kinds of native method interfaces.

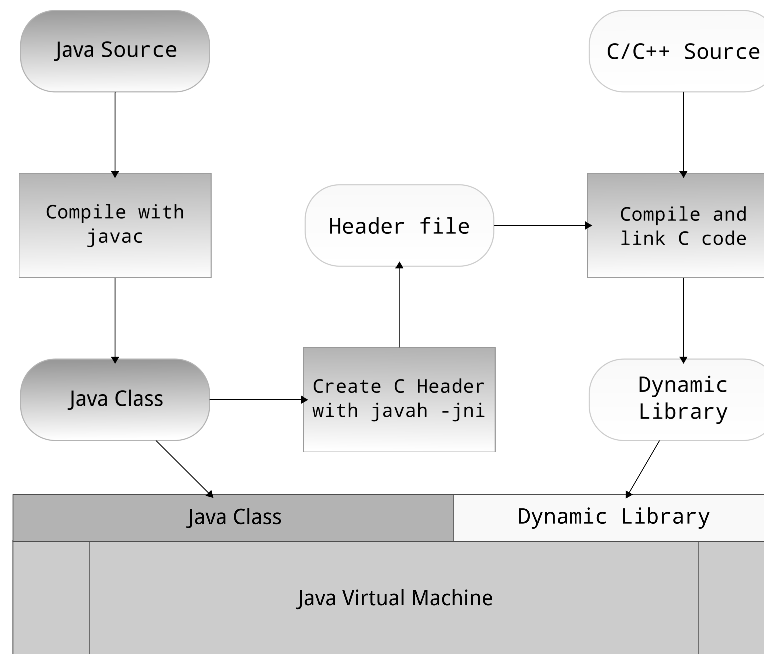


Figure 2.3: JNI flow

- Most importantly, application programmers are able to write one version of their native code and this version will run on different implementations of the Java virtual machine.

2.3.1 Typical JNI use

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 2.3) 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

2.3.2 JNI under Android: NDK

It is frequent, for developers who work on the Android framework - especially when their device introduces some new hardware components - the need for accessing to native functions, or drivers, from the Java side. Hence Android can be seen as a typical example of a situation where Java and Native worlds have to strictly cooperate: this is where using JNI can be useful (although each developer must carefully decide whether to use it, or to choose any possible alternative instead of embedding portions of native code, such as using already implemented Java libraries or interfaces). This being said, there's a toolset that lets developers embed components that make use of native code into Android applications very easily, with no need to necessarily follow the typical JNI steps mentioned above: this toolkit is named *"Android Native Development Kit"* (NDK), which can be found at: `http://developer.android.com/tools/sdk/ndk/index.html`). 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.
Under Android it becomes very easy to embed native code and, basically, the procedure can be resumed as follows:

- ❏ Download and install the Android NDK (one shot action)
- ❏ Develop your Android application, declaring the functions you want to access to as `native` methods, and use them as if they were (as actually are) Java methods
- ❏ Write your native code within a `jni` folder, and write an appropriate makefile (`Android.mk`) to export the compiled code as shared library
- ❏ Launch the `ndk-build` command within the application folder
- ❏ Launch your application, and that's all.

A detailed description and some examples of JNI implementations can be found in Appendix A

Chapter 3

Run-Time Resource Management of Android Applications

"My favorite things in life don't cost any money.
It's really clear that the most precious resource
we all have is time."

Steve Jobs

In the previous chapter, was described what the *BOSP* project is, and how it is structured, and some elements about JNI were given as well.
As previously mentioned, Barbeque is developed in native (C/C++) code and, as a consequence of its characteristic of portability, an Android-ready build can be deployed directly from its menu (through the `make menuconfig` command). Once Barbeque has been built, the aforementioned package is automatically deployed to the first device reachable by the *Android Debug Bridge (adb)*, a versatile command line tool that lets you communicate with an emulator instance or connected Android-powered device. Once Barbeque is deployed to an Android system, it is possible to run it, and to write native applications that exploit its functionalities, and implement its callback methods.
But Android applications, as it is known, are developed in Java language: *so, how can we let Android developers building their own Java applications which interact with Barbeque?*
The answer will be described in this chapter, and sometimes documented with short code listings and clarifying figures.

To have a general idea of how the whole stack is structured, and where this work gives its contribute, we can analyse Figure 3.1.

Three main sections can be identified within the aforementioned diagram: the Java, the native, and the hardware one. The system shown is clearly an heterogeneous one, as we can see in the hardware section, where there's a *GPGPU* platform (typically it will be identified, within this work, as a specific many-core accelerator platform by STMicroelectronics), and a general *Host* platform, which can be a multi-core processor.
In the first one, which is *Java-coded* and *Android-specific*, an *activity* and a *service* can be found: the former corresponds to a user screen, and it's the class which directly interacts with the user, the latter corresponds to a generic android service which extends our `BarbequeService`. This service class statically loads, when created, the shared object which maps the native library (Listing 3.6). The activity and the service communicate to each other thanks to a messaging paradigm, already implemented in *Android* and largely discussed in 3.2.1. Moving from here to the second section, the native one, the first block we encounter is what we called *JNI Bridge*, a portion of code that represents a sort of midland between the pure Java code, and the pure native code: this intermediate code is, precisely, the *Java-Native Interface (JNI)*. This block, along with the block which lays below it, is built as *shared library*, as indicated with the dashed line. The native libraries can be found right below the JNI bridges: in the figure we put the *RTLib* (Sections 2.1, 3.1), and a generic *GPGPU* library. The latter is inserted into a *GPGPU SDK*, while the former is part of the *BarbequeRTRM*, and lets any application interacting to it, through the already described native command and callbacks (Section 2.1). Both the communication streams between the *GPGPU SDK* and the *BarbequeRTRM* go through the *Linux Kernel* and the respective *drivers* to the last section, the *hardware* one. Here the hardware platforms are placed: the *GPGPU* with its own (reconfigurable) firmware, the Host cpu, the memory the two share, and the possible peripherals (eg. a webcam, or a keyboard). *BarbequeRTRM* interacts with both the *GPGPU* and the *Host* processors, controlling the resources availability and applying the best possible both hardware and software configuration to provide the best *run-time* experience and resources allocation.

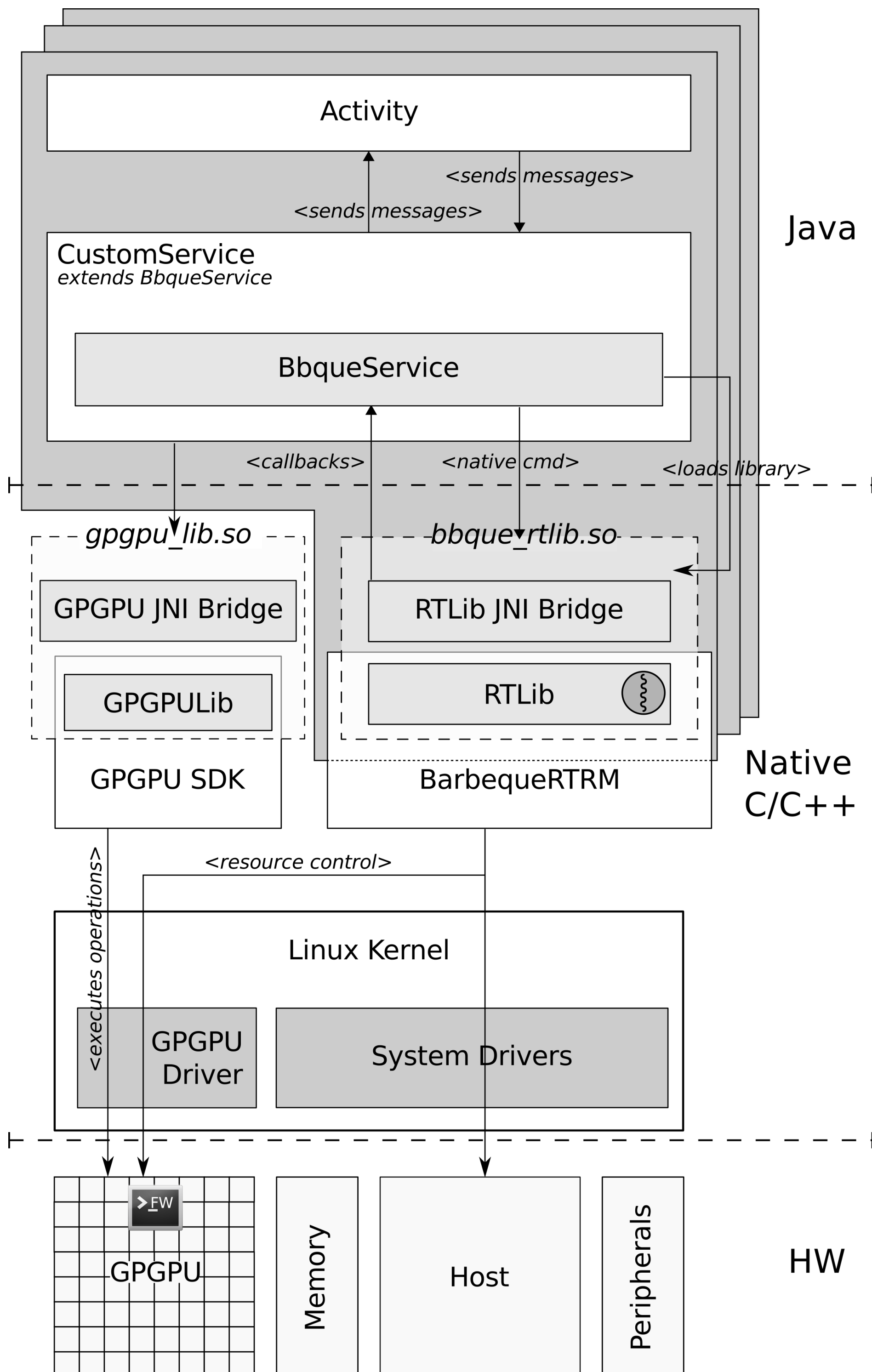


Figure 3.1: System block diagram

Stream example: From within the *Activity*, two messages are dispatched, and delivered to the `BbqueService` class:

- `MSG_CREATE`: to create an execution context. This command is executed on the *shared library* which calls the native function in charge of creating an *RTLib* instance and register it to the Barbeque manager. If everything goes smoothly, a specific value is returned.
- `MSG_START`: to kick start the Barbeque *state machine* (see Figure 2.1): from this very moment on, Barbeque typically executes its cycle of `onRun` and `onMonitor` callbacks along with potential others, depending on the resource availability and needs (e.g. it may call the `onConfigure`, when the application reconfiguration is requested)

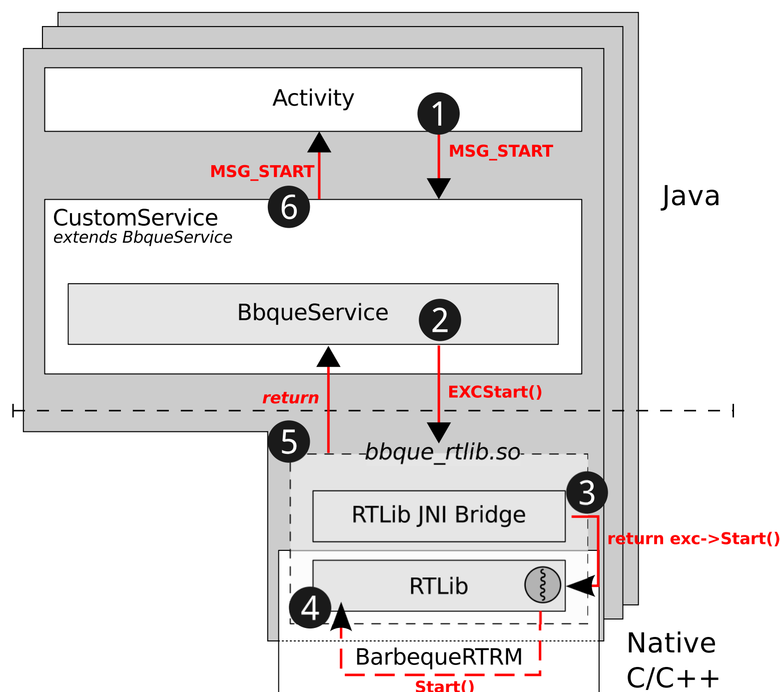


Figure 3.2: Call flow for MSG_START

What happens during the aforementioned calls is shown in Figure 3.2. The activity (in this specific case the call is triggered by a button, from the user interface) obtains a *message* and sends it to the correspondent instance of the *service* ❶: the latter executes the native call ❷ on the shared library, which embeds an instance of the RTLib that, in turn, interacts with Barbeque ❸. Barbeque performs its own checks and routines ❹ and its response - which is returned as a native value by the

RTLib to the JNI bridge ❺ - is then sent back by the *service* to the *activity* through the *message* with the correspondent label ❻.

This being done, while *Barbeque* keeps calling the `onRun`, which executes the elementary piece of code needed to process an atomic element of the stream (e.g. a frame of a video stream), the need for the application reconfiguration could arise: if so, *Barbeque* receives a signal from the hardware layer, and notifies this situation to the *Service*, by executing the `onConfigure` callback; depending on the implementation of this callback method, the application reconfigures its settings to perform at best according to the available resources.

3.1 RTLib JNI bridge

As described in Section 2.1, the *Run-Time Library (RTLib)* supports the application which needs to interact with Barbeque Framework, by providing a rich set of features as well as supporting the application-specific run-time management activities. When some *JNI* mentions were made in Section 2.3, and deeply described in Appendix A, whenever it is needed to expose some native functions to the Java side, it is required to express their signatures following a very specific pattern, let's analyse a brief example: if we need to call a native function `foo` from within a Java class `ExampleClass`, which belongs to the package `com.examples.my`, there are two main steps we have to take:

1. Declare within the Java class the method
   `public native void foo();`
   without any implementation, so that it's visible to the class
2. Implement a native function with the specific signature:
   `Java_com_example_my_foo(JNIEnv *env, jobject thiz)`

This being said, to expose *RTLib* native functions, we had to create a native wrapper class, which makes all the already working functions accessible to Java through the appropriate signature, wrapping each call within its correspondent JNI name. An example of a native signature from Barbeque's RTLib can be seen at Listing 3.1, its wrapping function to expose it through JNI can be seen at Listing 3.2, and the Java declaration and usage is shown at 3.3.

```
1 RTLIB_ExitCode_t RTLIB_Init(const char* name, RTLIB_Services_t **services);
```

Listing 3.1: RTLib native function signature from rtlib.h

```
//Global reference to the RTLib instance
RTLIB_Services_t *rtlib = NULL;

// RTLIB_ExitCode_t RTLIB_Init(const char *name, RTLIB_Services_t **rtlib)
JNIEXPORT jint
Java_it_polimi_dei_bosp_BbqueService_RTLIBInit(
            JNIEnv *_env, jobject _thiz,
            jstring _name) {
      const char *name = _env->GetStringUTFChars(_name, 0);
      RTLIB_ExitCode_t result;

      obj = (jobject)_env->NewGlobalRef(_thiz);

      result = RTLIB_Init(name, &rtlib);
      if (result != RTLIB_OK) {
            LOGE("RTLIB initialization failed");
            return (-result);
      }

      LOGI("RTLIB initialization done");
      return RTLIB_OK;
}
```

Listing 3.2: RTLib wrapping example of Listing 3.1

```
public native int RTLIBInit(String mode);

public void onCreate() {
      Log.d(TAG, "onCreated");
      creationTime = System.currentTimeMillis();
      super.onCreate();
      int response;
      response = RTLIBInit("test");
      Log.d(TAG, "Response from RTLIBInit is:"+response);
}
```

Listing 3.3: Java declaration and usage of the JNI function declared at Listing 3.2 within the Java class `it.polimi.dei.bosp.BbqueService`

The aforementioned wrapping concerns the Java-to-Native call. Barbeque Framework strongly relies on callbacks, which were wrapped as well as straight calls.
In Listing 3.4, an example of the callback wrapping for the `onRun` function is shown: the framework will call the *onRun* function as usual, but the Java callback will be performed, thanks to the
`env->CallIntMethod(obj, cb[ON_RUN].method)`
The way we chose to proceed is mutuated from the *Tutorial3* shown into Appendix

A.3. During the initialization of the library, a call to fill a *callbacks array* is called, where the structure `callback_t` of the array is shown in Listing 3.5

```
1 RTLIB_ExitCode_t
2 BbqueAndroid::onRun() {
3       LOGD("Callback onRun(), %d", Cycles());
4       if (env->CallIntMethod(obj, cb[ON_RUN].method))
5             return RTLIB_EXC_WORKLOAD_NONE;
6       return RTLIB_OK;
7 }
```

Listing 3.4: RTLib wrapping example for the onRun callback

```
1 typedef struct {
2       const char* name;
3       const char* signature;
4       jmethodID method;
5 } callback_t;
6
7 typedef enum {
8       ON_SETUP = 0,
9       ON_CONFIGURE,
10      ON_SUSPEND,
11      ON_RESUME,
12      ON_RUN,
13      ON_MONITOR,
14      ON_RELEASE,
15      CB_COUNT // This must be the last entry
16 } cbid_t;
17
18 static callback_t cb[CB_COUNT];
```

Listing 3.5: RTLib wrapping example for the onRun callback

The makefile `Android.mk` will let this library being built as a *shared library*: how it is done, is shown in Appendix A.
To sum up, this first part of the work consisted in creating some native code to wrap the pre-existing functions into *JNI-compliant* signatures, and the calls to native callbacks into proper *JNI-based* Java method invocations.

This being done, all that remains will be coded within the Java environment: it's easy to understand how writing this small portion of native code helped us in adapting the *RTLib* instead of writing all from scratch in Java language.
Now, we can analyse how the integration into Android has been implemented, and a full-working Android Java API to Barbeque has been developed.

3.2 Software design

To make *RTLib* functions available to the Android Apps developer an Android Service has been developed. The basic idea is to provide the developer with a main `Service` class that can be extended by a customized service, that will actually implement the callback methods, so that they will be executed whenever Barbeque will call them.

3.2.1 BbqueService

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, which is exactly what we needed: our Service "maps" the RTLib, and it's a kind of interface to that, always running and handling the interaction with Barbeque. Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC): we are going to analyse this communication aspect in 3.2.1.3.
A service is *"bound"* when an application component (like an *Activity*) binds to it by calling `bindService()`. 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. To properly bind a service, you need to implement a couple of callback methods: `onStartCommand()` to allow components to start it and `onBind()` to allow binding. Our Service is called `BbqueService`, and obviously extends the Android `Service` class, and is going to be a *bound* service, since we will need to open a communication channel between it and the Activity.
A schema of the bound service life cycle is depicted at Figure 3.3.
The first operation that the service class will perform when instantiated will be loading the shared library, as shown in Listing 3.6.

```
1 static {
2       System.loadLibrary("bbque_rtlib");
3 }
```

Listing 3.6: Service loads shared library

The `BbqueService` mainly consists of three sections:

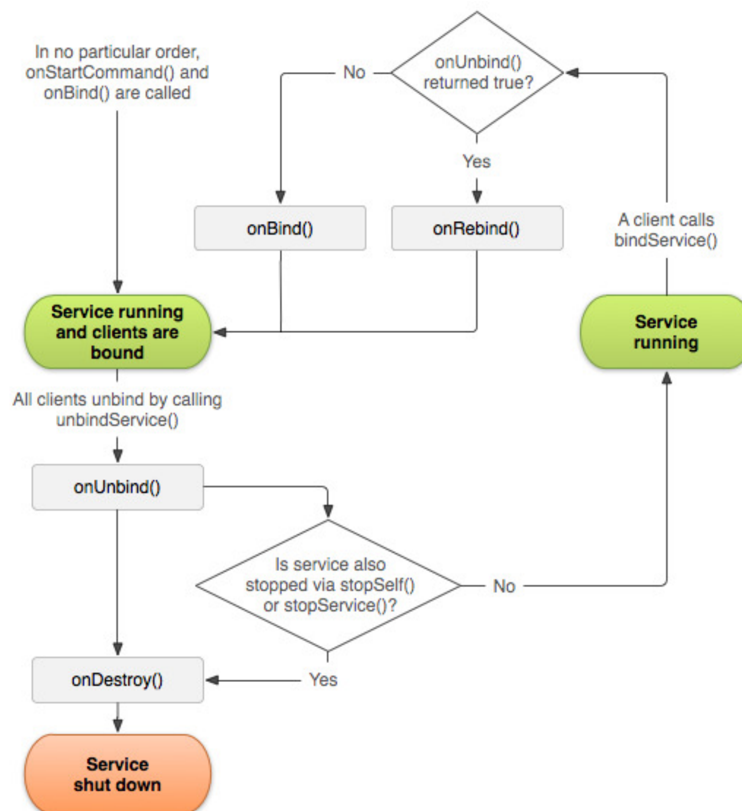


Figure 3.3: Bound service life cycle

- Native methods declaration
- Callbacks implementation
- Messaging handling

3.2.1.1 Native methods declaration

The `BbqueService` declares all the signatures corresponding to the native functions he wants to access as "`public native`", as we saw for example at line 1 of Listing 3.3. Those methods are called whenever needed, within the service class, through proper public methods, that are visible to the *custom service* that will extend it: doing so, the developer is able to create her own Android service which extends our `BarbequeService` and accesses to RTLib native methods.

The main RTLib functions were exposed as native calls, which can be performed within the *service* (and therefore any activity which is bound to it), by sending the proper message: in Table 3.1 the message code and the native method correspon-

dence is shown.

Message code	Native method
MSG_ISREGISTERED	EXCisRegistered()
MSG_CREATE	EXCCreate(name,recipe)
MSG_START	EXCStart()
MSG_WAIT_COMPLETION	EXCWaitCompletion()
MSG_TERMINATE	EXCTerminate()
MSG_ENABLE	EXCEnable()
MSG_DISABLE	EXCDisable()
MSG_GET_CH_UID	EXCGetChUid()
MSG_GET_UID	EXCGetUid()
MSG_SET_CPS	EXCSetCPS()
MSG_SET_CTIME_US	EXCSetCTimeUs()
MSG_CYCLES	EXCCycles()
MSG_DONE	EXCDone()
MSG_CURRENT_AWM	EXCCurrentAWM()

Table 3.1: The correspondence between message code (left) and native method called (right) is shown. Each message code can be used to execute calls from the Activity to the Service, and vice versa: this means that for each task to accomplish, the activity and the service have a unique code which identifies the task, and sets up the communication

By extending the Android `Service` class, our `BbqueService` overrides the `onCreate()` method, which is called whenever a service is instantiated and, in our specific case, also *bound*.

3.2.1.2 Callbacks implementation

Basically the *service* declares, in addition to native methods, all the callback methods needed by the native *RTLib* wrapper. They just send a broadcast intent, to notify they've been called. They can then be overridden by the *customized service* that extends the `BbqueService`, so that the callback from the native library will be then diverted to the service-to-be. In Listing 3.4 the native JNI callback for the `onRun` method is shown: its Java counterpart can be found at Listing 3.7.

```
1 public int onRun() {
2       Log.d(TAG,"onRun called");
```

```
3         intent.putExtra("BBQ_DEBUG", "onRun called");
4         intent.putExtra("INTENT_TIMESTAMP",
5                   System.currentTimeMillis()-creationTime);
6         intent.putExtra("APP_NAME", name);
7         sendBroadcast(intent);
8         return 0;
9  }
```

Listing 3.7: Example for the onRun callback - Service side

In addition to the `onCreate`, the callbacks the developer can customise by overriding can be found in Table 3.2;

Callback
onSetup()
onConfigure(int)
onSuspend()
onResume()
onRun()
onMonitor()
onRelease()

Table 3.2: Apart from the onRun() method overriding, which is mandatory to let the application execute its processing block when Barbeque calls back, the methods above listed can be (and should be) implemented as well, to properly exploit the run-time auto-configuration of the application.

3.2.1.3 Message handling

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). Consequently to our approach of implementing a service to act as an interface to *RTLib*, we needed a lightweight and well structured messaging protocol to allow the *activity* communicating with the *service*, and by this, to the *native library*. This protocol was identified into the `Messenger` class. Using `Messenger` is the simplest way to perform interprocess communication (IPC), because it queues all requests into a single thread so that you don't have to design your service to be thread-safe. The service defines a Handler (Listing 3.8, line 1) 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 (Listing 3.8 line 20-21), allowing the client to send commands to the service using *message objects* (Table 3.1). Additionally, the client can define a `Messenger` of its own so the service can send messages back: a double definition of a proper Handler within the two classes (client and server) lets the two communicate bidirectionally. As an implementing choice, we decided that the two directions of the same task (e.g. starting the processing from the activity, and receiving the response by the service) share the same message code.

```
1  protected class BbqueMessageHandler extends Handler {
2        @Override
3        public void handleMessage(Message msg) {
4              switch (msg.what) {
5                    case MSG_ISREGISTERED:
6                          isRegistered(msg.replyTo);
7                          break;
8                    case MSG_CREATE:
9                          create(msg.replyTo, msg.obj);
10                         break;
11                   default:
12                         super.handleMessage(msg);
13             }
14       }
15 }
16
17 final Messenger mMessenger = new Messenger(new BbqueMessageHandler());
18
19 @Override
20 public IBinder onBind(Intent intent) {
21       return mMessenger.getBinder();
22 }
```

Listing 3.8: Messenger protocol implementation - Service side

What is shown in Listing 3.8 is a portion of the service code needed to properly handle the receiving of messages sent by another component, typically the caller activity. As can be guessed, the parameter `replyTo` at lines 6 and 9, contains the reference to the dual `Messenger` object belonging to the caller: hence through this reference the service is able to reply with its own response message to the caller, as shown in Listing 3.9, line 12.

```
1  protected void create(Messenger dest, Object obj) {
2        String messageString = obj.toString();
3        String params[] = messageString.split("#");
4        name = params[0];
```

```
5           recipe = params[1];
6           Log.d(TAG, "create, app: "+name+" with recipe "+recipe);
7           int response = EXCCreate(name, recipe);
8           Message msg = Message.obtain(null, MSG_CREATE,
9                                         response,
10                                        0);
11          try {
12                  dest.send(msg);
13                  replyTo = dest;
14          } catch (RemoteException e) {
15                  e.printStackTrace();
16          }
17  }
```

Listing 3.9: Messenger protocol implementation - Service side

The `Message.obtain(...)` can be used with different signatures, and Android designers provided the developers with some standard parameters, to be used in basic cases, and typically there are two `int` called `arg1` and `arg2` as well as a generic `Object` called `obj` which, as a only constraint, has to implement the `Parcelable` interface. The overhead added by a message, in terms of time, is around 170˜200 microseconds per message.

Being clear how the `BbqueService` was implemented, and how the messaging protocol works, we can briefly see how a developer can set up her own Activity.

3.2.2 Activity: common aspects

In Section 3.3 two sample activities that were realised within this work are deeply analysed and commented, but here some common aspects, and best practices to follow when developing applications that need to interact with Barbeque framework are described.
The *bound service* concept was broadly described above: the *activity* has to **bind** itself to the service, and this is typically done withing the `onStart()` callback method, which is overridden from the parent main `Activity` class, as shown in Listing 3.10, where the `CustomService.class` which appears at line 5 is a service which extends our `BbqueService`.

```
1   @Override
2   protected void onStart() {
3           super.onStart();
4           // Binding to the service.
5       bindService(new Intent(this, CustomService.class),
6                           mConnection, Context.BIND_AUTO_CREATE);
```

```
7 }
```

Listing 3.10: Activity: bind to a Service

The `mconnection` object that we pass to the `bindService` method is an object of type `ServiceConnection`, and its construction is shown in Listing 3.11. The `mService` object which appears at lines 5 and 10 is the one we use to send messages to the `CustomService`: obviously, given that the latter extends the `BbqueService`, each message sent through the `mService` will be sent to this one as well.

```
1  private final ServiceConnection mConnection = new ServiceConnection() {
2        @Override
3        public void onServiceConnected(ComponentName className,
4                    IBinder service) {
5              mService = new Messenger(service);
6              mBound = true;
7        }
8        @Override
9        public void onServiceDisconnected(ComponentName className) {
10             mService = null;
11             mBound = false;
12       }
13 };
```

Listing 3.11: Activity: creation of the `ServiceConnection` object

The *message handling* into the activity is absolutely dual with respect to the one described at 3.2.1.3.

3.3 Testing applications

With the *JNI wrapper*, properly compiled, and the Java class which implements the *Barbeque service*, we are ready to test the Barbeque framework by analysing two Java applications specifically developed for this purpose. The first one just tests the calls to Barbeque, and the callbacks from Barbeque, while the second one consists of a *face detection* application which uses a library by *ST Microelectronics* developed to work on the many-core *STHorm* platform, codename *P2012*. Such a platform will be the ideal application field for a RTRM as Barbeque.

3.3.1 BOSP testing application

With respect to what has been described in the previous sections, to build a customised application the developer has to create a *service* - which extends the `BbqueService` and one (or more than one) *activity*.

Custom service This very compact `CustomService` basically does three main things:

- Declares a `CustomeMessageHandler`, which extends the default barbeque message handler `BbqueMessageHandler` (Listing 3.8, line 1) to handle new custom messages others than the default ones (in which cases it will call the `super` implementation of the handler. An example can be seen at Listing 3.12

```
1  class CustomMessageHandler extends BbqueMessageHandler {
2        @Override
3        public void handleMessage(Message msg) {
4              switch (msg.what) {
5              case MSG_CYCLES:
6                    Log.d(TAG, "Message cycles setting: " + msg.arg1);
7                    cycle_n = msg.arg1;
8                    break;
9                    default:
10                   super.handleMessage(msg);
11             }
12       }
13 }
```

Listing 3.12: `CustomService`: message handler overriding

  In our implementation, the custom message added to the default ones will be used to set, from the activity, the number of *onRun* cycles to be performed.

- Overrides the `onBind` method, to return an `IBinder` interface to the *activity* that will execute the binding, which now links to its own instance of `Messenger` object, as shown in Listing 3.13

```
1  final Messenger cMessenger = new Messenger(new CustomMessageHandler());
2
3  @Override
4  public IBinder onBind(Intent intent) {
5        return cMessenger.getBinder();
```

```
6 }
```

Listing 3.13: `CustomService`: onBind overriding

- Overrides Barbeque callbacks methods or, at least, the `onRun`, which is mandatory to have a working application. The code related to this overriding is shown at Listing 3.14, and a screenshot of the execusion can be found at Figure 3.4, where the displayed information in the `TextView` ❶ comes from lines 5-6 of the aforementioned listing.

```
1  @Override
2  public int onRun() {
3      int cycles = EXCCycles();
4      Log.d(TAG, "onRun called, cycle: " + cycles);
5      intent.putExtra("BBQ_DEBUG", "onRun called, cycle: " + cycles);
6      sendBroadcast(intent);
7      try {
8          Thread.sleep(1000);
9          if (cycles >= cycle_n)
10             return 1;
11     } catch (InterruptedException e) {
12         }
13     return 0;
14 }
```

Listing 3.14: `CustomService`: onRun overriding

This being done, the `CustomService` is ready to be bound to an application component, typically our activity, and to interact with Barbeque native code as well. The binding configuration must be explicitly declared in the configuration file named `AndroidManifest.xml`, which is local to any Android application. Within the `<application>` node, the following line must be added:
`<service android:name=".CustomService" ></service>`

Testing activity The activity developed to test the integration of the Barbeque basic features is shown in Figure 3.4. Below each part, with its related behind-code is analysed.

❶ `TextView` shows messages which come as Barbeque *responses* to methods invocations through the messaging protocol

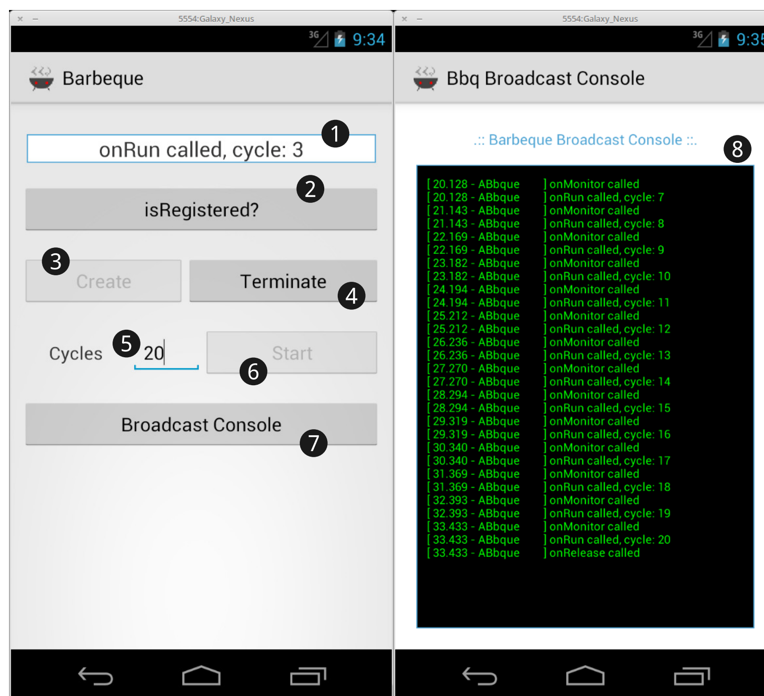


Figure 3.4: Barbeque testing application screenshot

❷ *isRegistered* button triggers the dispatch of the `MSG_IS_REGISTERED` message, which is directly handled by the `BbqueService` as shown at Listing 3.8 line 5

❸ *Create* button triggers the dispatch of the `MSG_CREATE` message, which is directly handled by the `BbqueService`. For the sake of clarity, the code behind this button is shown at Listing 3.15, to be considered ad part of the flow received by the *service* as shown at Listing 3.8 line 8 and, from there, forwarded to Listing 3.9.

```
1  public void btnCreate(View v) {
2        Log.d(TAG, "Create button pressed...");
3        if (!mBound) return;
4        Message msg = Message.obtain(null, CustomService.MSG_CREATE, 0, 0);
5        msg.obj = APP_NAME+"#"+APP_RECIPE;
6        try {
7              msg.replyTo = mMessenger;
8              mService.send(msg);
9        } catch (RemoteException e) {
10             e.printStackTrace();
11       }
```

```
12 }
```

Listing 3.15: `BbqueActivity`: Create button code

❹ *Terminate* button triggers the dispatch of the message `MSG_TERMINATE`, which at the end, in the *service* calls the `EXCTerminate()`

❺ `EditText` field is used to set the number of *onRun* cycles we want the application to execute. When the current cycles number exceeds the upper bound, the execution is terminated, and Barbeque calls the *onRelease* callback.

❻ *Start* button triggers the dispatch of the `MSG_CYCLES` which is handled by the `CustomService` to set a local variable, and consequently the `MSG_START` message, is sent, and handled by the `BbqueService`

❼ *Broadcast Console* button opens the *Bbque Broadcast Console* ❽ which basically outputs all the intents broadcasted by the *service*. More on this will be discussed further.

To receive *intents* sent by the *service*, and output them into the `TextView` ❶, a listener to them has been enabled. At first a `BroadcastReceiver` is created, which overrides the `onReceive` methods in which body the code to be executed whenever an intent it's receiver of is contained. The code is shown at Listing 3.16 How these *intents* are broadcast by the *service* is described within the next paragraph.

```
1  IntentFilter receiverFilter = new IntentFilter ();
2  BroadcastReceiver receiver = new BroadcastReceiver() {
3        @Override
4        public void onReceive(Context context, Intent intent) {
5              String bbqDebugIntent = intent.getStringExtra("BBQ_DEBUG");
6              output.setText(bbqDebugIntent);
7        }
8  };
9  @Override
10 public void onCreate(Bundle savedInstanceState) {
11       //...
12       receiverFilter.addAction("it.polimi.dei.bosp.BBQUE_INTENT");
13       registerReceiver(receiver, receiverFilter);
14       //...
15 }
```

Listing 3.16: `BbqueActivity`: Intent handling

Barbeque Broadcast Console *Barbeque Broadcast Console (BBC)* is an activity, called by the main Barbeque activity we've seen in the previous paragraph, which explores the possibilities offered by the *intents mechanism* provided by Android.
Intents are asynchronous messages which allow Android components to request functionalities from other components of the Android system. For example an Activity can send an Intent to the Android system which starts another Activity. Therefore Intents allow to combine loosely coupled components to perform certain tasks. An Intent can also contain data.
The *BBC* implements a `BroadcastReceiver` as seen for the previous activity (Listing 3.16), but is able to retrieve much more information, from the intent it has registered a filter for with the method:
`receiverFilter.addAction("it.polimi.dei.bosp.BBQUE_INTENT");`
as show in Listing 3.17

```
1  BroadcastReceiver receiver = new BroadcastReceiver() {
2        @Override
3        public void onReceive(Context context, Intent intent) {
4              String entry = String.format("[%7.3f - %-15s] %s \n",
5                    ((float)(intent.getLongExtra("INTENT_TIMESTAMP",0)))/1000,
6                    intent.getStringExtra("APP_NAME"),
7                    intent.getStringExtra("BBQ_DEBUG"));
8              console.append(entry);
9              final int scrollAmount = console.getLayout().getLineTop(
10                         console.getLineCount())-console.getHeight()+10;
11          // if there is no need to scroll, scrollAmount will be <=0
12          if(scrollAmount>0)
13             console.scrollTo(0, scrollAmount);
14          else
15             console.scrollTo(0,0);
16       }
17 };
```

Listing 3.17: `BBCActivity`: Intent handling

Whenever an intent from the *service* is received, the *action* retrieves all the attributes associated to that very specific intent.
In Listing 3.7 the intent dispatching from within the *service* is shown, with regard to the *onRun* callback. A partial log of the output is shown at ❽ in Figure 3.4.

3.3.2 STHorm Face Detection

STHorm (codename Platform2012) is a many-core embedded architecture that has been designed by STMicroelectronics and by CEA as a scalable and customizable acceleration device. [9] A test chip in 28nm is being sampled now, featuring 69 processors in less than 20mm2. Our testing application currently runs on an emulated platform, but is meant to show one of the many possible applications that can be coded directly in Java, and supported by the BarbequeRTRM, which is being deployed to be tested on the aforementioned platform.
This application has been largely reimplemented, and integrated to exploit our new `BbqueService`, which is the core of this work.
A sequence diagram to clarify the whole process is provided at Figure 3.5

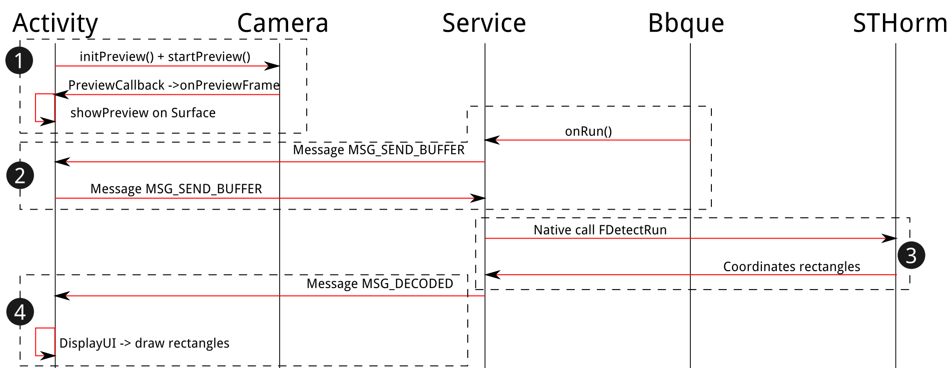


Figure 3.5: STHorm Face Detection activity - Sequence diagram

We mainly have five actors, and the communication between the *activity* and the *service* is implemented exploiting the messaging protocol we mentioned before. In Figure 3.5, four main steps have been identified.

❶ The *activity* initialises the `Camera` object. The preview feature of the *Android Camera* has been chosen: it already outputs camera frames to a `Surface` object, and its frame rate is high enough to give to the user a good user experience. On top of this *surface* we put an `ImageView` object, where we print, if and whenever available, the rectangles indicating the faces within the picture.
Whenever a new frame is available from the camera, it is shown on the surface, and a callback is executed by default within Android: within the method `onPreviewFrame`, it's possible to save (for example into a double buffer, as we did) frames for future use.

➋ The *Service* mentioned is an abstraction of both the basic `BbqueService` and a customized `STHormFDService`, which extends the former, and implements its own processing code, including the `onRun` overriding. Thus, whenever an *onRun* callback is performed by *Barbeque*, a *message* is sent to the *activity*, asking for a new frame. The activity replies sending back the most recent frame, taken from the aforementioned double buffer.

➌ The *Service* has received the frame, and calls some native code to process it: this is possible, again, through a JNI call. The *Service* gets back an array with the number of faces found into the frame and, if any, the coordinates of the rectangles corners that enclose each face.

➍ The *Service* dispatches the retrieved information to the *activity*, through a *message*, and rectangles are drawn upon the camera preview surface.

The previous enumeration is repeated for any frame the *onRun* is able to process, at the speed set by the user, from the slider element (`SeekBar`) available on the interface.

Chapter 4

Experimental results

"No amount of experimentation
can ever prove me right;
a single experiment can prove me wrong."
Albert Einstein

It's a very challenging aspect to analyse how a customized and innovative run-time resource manager performs, when integrated into the Android system, and its JVM. In this work some basic measures were done, mostly time related, and every now and then some improvement suggestion are given.

4.1 Experimental setup

Android ships with a debugging tool called the Dalvik Debug Monitor Server (DDMS), which provides port-forwarding services, screen capture on the device, thread and heap information on the device, logcat, process, and radio state information, incoming call and SMS spoofing, location data spoofing, and more. This page provides a modest discussion of DDMS features; it is not an exhaustive exploration of all the features and capabilities. To perform our analysis, we mainly used two instruments:

- *Traceview*: a graphical viewer for execution logs created by using the Debug class to log tracing information in the code. Traceview can help debugging your application and profiling its performance.

- *dmtracedump*: a tool that provides an alternate way of generating graphical call-stack diagrams from trace log files. The tool uses the Graphviz Dot utility to create the graphical output, so you need to install Graphviz before running dmtracedump. The dmtracedump tool generates the call stack data as a tree diagram, with each call represented as a node. It shows call flow (from parent node to child nodes) using arrows. *dmtracedump* provides the possibility to export the .trace binary output file into a human readable text file, an html file, or a png diagram.

Some bash scripting has been used as well, to post-process trace logs.

4.2 Measures and considerations

4.2.1 Native response time

The main goal of this work was to integrate the *native code* into the *Java* environment, which is the default one for Android applications.
Some tests were performed, with respect to some native commands, as:

- `EXCisRegistered`
- `EXCCreate`
- `EXCStart`

Two main situations can be considered, to analyse some performances and, in particular, the overhead added to the native code: we sampled the execution time for the *EXC* methods (native calls through JNI), and the execution time for the Java methods within the service that, in turn, call the *EXC* methods.
In Table 4.1 we can see the added time needed to perform the native call through the JNI mechanism from the Java environment, which was calculated in 227.033±23 μs.

# samples	min time [μs]	max time [μs]	avg time[μs]	CI 99% [μs]
30	186	417	227.033	±23.004

Table 4.1: `EXCisRegistered` timings on the sample application BbqueActivity on 30 samples

This native call is performed within a Java call, which runs within the Bbque-Service: it's interesting to analyse how much time overhead is added by the whole

Java procedure, from the actual Java call, to the response, and an example can be seen in Table 4.2, where are shown the statistic data regarding 30 samples. In addition to data, the correspondent graphical representation of the table is shown in Figure 4.1, where the *traceview* output can be found.

call	avg time[μs]	σ	CI 99% [μs]
`isRegistered`	1051.233	244.352	$\pm$115.100
Messenger.send	347.700	144.251	$\pm$67.948
EXCisRegistered	227.033	48.836	$\pm$23.004
Log.d	225.567	29.376	$\pm$13.837
Message.obtain	69.900	10.512	$\pm$4.952

Table 4.2: `BbqueService.isRegistered()` timings on the sample application BbqueActivity, with detail of inner children methods on 30 samples

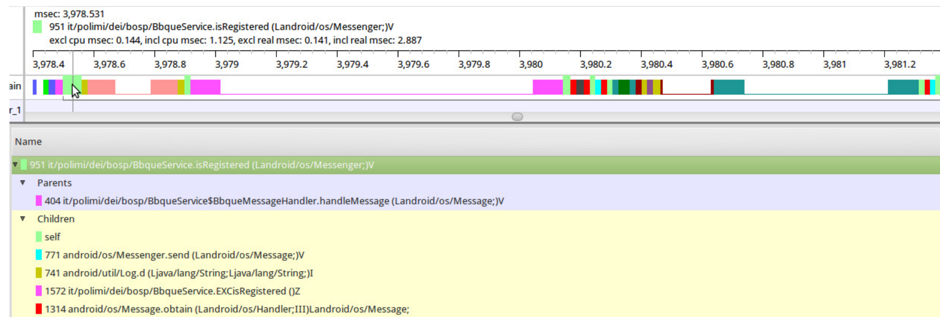


Figure 4.1: `isRegistered()` execution stack - traceview

The same considerations will be done for the `EXCCreate` function, which is the one that creates the execution context. Data on 60 samples are given in Table 4.3. The native call executes in an average time of 1.463$\pm$ 0.187ms.

# samples	min time [μs]	max time [μs]	avg time[μs]	CI 99% [μs]
60	1042	3807	1463.317	$\pm$187.423

Table 4.3: `EXCCreate` timings on the sample application BbqueActivity on 60 samples.

The aforementioned results can be contextualised within the "parent" Java call, which is typically executed by the `BbqueService.create()` method. The results shown in Table 4.4 refer to the native result *BbqueActivity* of Table 4.3.

call	avg time[μs]	σ	CI 99% [μs]
`create`	4925.017	672.085	±223.856
EXCCreate	1463.317	562.702	±187.423
String.split	1383.267	632.963	±210.825
StringBuilder.append	243.061	72.662	±13.973
Messenger.send	417.550	243.547	±81.120
Log.d	229.350	28.768	±9.582
Message.obtain	106.517	196.555	±65.468

Table 4.4: `BbqueService.create()` timings on the sample application *BbqueActivity* on 60 samples

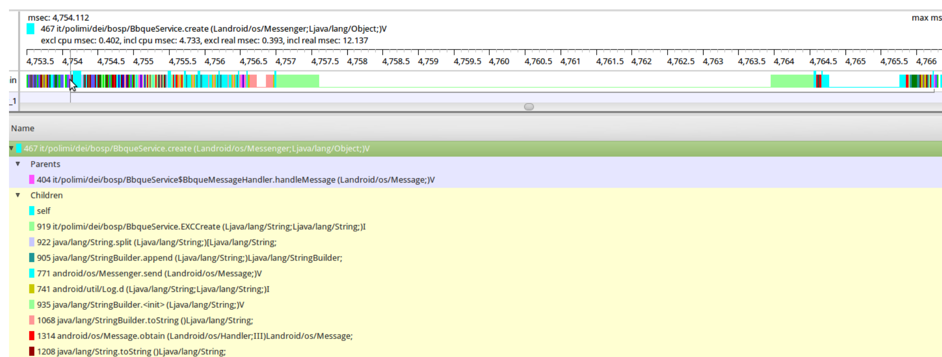


Figure 4.2: `create()` execution stack - traceview

In addition to the basic information we can get from data, we can notice that a simple split operation on a string costs a lot, if compared to the main operation run by this method, almost the same amount of time.
Generally the `EXCCreate` takes much more time than the `isRegistered` above, because of its JNI implementation, where the native function operates on strings, to retrieve the application name and the recipe.

The last native call that was mentioned at the beginning of this chapter, is the `EXCStart`. As done for the previous functions its time profiling is shown at Table 4.5. The traceview process execution stack of the Java `start` method (which, in turn, calls the `EXCStart`) is shown at Figure 4.3, and statistics out of 30 *start* samples is given at Table 4.6.

Some considerations about the aforementioned methods: being those calls embedded within the *BbqueService*, they generally won't be overridden, therefore their performances can vary only depending on the running architecture, not on the im-

# samples	min time [μs]	max time [μs]	avg time[μs]	CI 99% [μs]
30	554	1737	793.200	±126.380

Table 4.5: `EXCStart` timings on the sample application BbqueActivity on 30 samples.

call	avg time[μs]	σ	CI 99% [μs]
`start`	1615.233	350.317	±165.014
EXCStart	793.200	268.299	±126.380
Messenger.send	379.700	157.081	±73.992
Log.d	234.917	55.385	±18.447
Message.obtain	72,933	15.929	±7.503

Table 4.6: `BbqueService.start()` timings on the sample application *BbqueActivity* on 30 samples

plementation. They will also likely be called not frequently, therefore the small overhead added by the Java/JNI code can be considered absolutely reasonable.

4.2.2 Messaging system overhead and performances

As described in 3.2.1.3 the Messenger class was used to pass messages among processes, in particular between the activity and the service, and vice versa.
It's interesting to analyse how this mechanism can influence performances and completion time. Data were sampled, again, from the *BbqueActivity* performing basic operations, like the `isRegistered`, the `create` and the `start` methods.
The first thing that pops up, among data, is that the delivery time of messages can vary between one or even two orders of magnitude: this is because messages are continuously used by *Android* as an IPC method, and they don't have priorities nor any parallelism. By the tests here conducted, for example, some of the messages were enqueued behind system ones, therefore they were affected by the execution time of system functionalities, as UI interactions. This appears to be a huge limitation of this messaging system, but still we have to consider the delay it introduces within the context of processing big streams of data which, once the computation is started, is an atomic activity.
In the tables that follow, calls stack with in/out methods timings is shown.
As an example, if we consider the `isRegistered` call, which starts from a UI but-

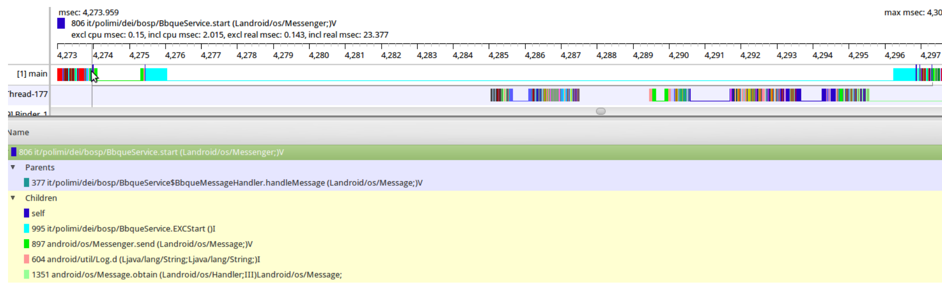


Figure 4.3: `start()` execution stack - traceview

ton, and we measure the time which elapses between the message sending, to the correspondent actual method execution within the service class, we see that a message, to be delivered and its action triggered, takes a time which is one order of magnitude bigger than the time needed for the execution of the method itself. As will be discussed later, however, this impact is almost negligible, especially when compared to the main computation that typically is performed within core methods of the service, as the *onRun*.

`btnIsRegistered`	IN [ms]	OUT [ms]	δ(i-1)	δ
Messenger.send	1434.054	1434.376	-	0.322
Other operations executed for UI rendering, through message sending				
Message.dispatch	1446.117	1447.881	11.741	1.764
CustomService.CustomMessageHandler	1446.135	1447.863	0.018	1.728
BbqueService.BbqueMessageHandler	1446.155	1447.847	0.020	1.692
BbqueService.isRegistered	1446.175	1447.829	0.020	1.654
Time from "Messenger.send" end to "isRegistered" start			**11.799**	

Table 4.7: Button *isRegistered?* press: timings on the sample application. IN = instant method enters, OUT = instant method exits, δ(i-1) = IN(i)-IN(i-1), δ = OUT(i)-IN(i)

Before considering more data samples, it's worth noticing that the *message handler* implementation within the `BbqueService` described in 3.2.1.3, when extended by the `CustomService` it brings an additional cost in terms of computation time, since the `switch` structure will be repeated twice: this comes clear at Table 4.7, where there's a double 18-20 microseconds time.

In addition to the single example given so far, we provide a statistic which is shown in Table 4.8, where times corresponding to the last line of Table 4.7 were sampled

on 30 runs.

# samples	min [ms]	max [ms]	avg [ms]	σ	CI 99% [ms]
30	11.054	17.104	12.552	1.334	$\pm$0.618

Table 4.8: Time elapsed from "Messenger.send" exit to "isRegistered" enter, on the sample application BbqueActivity on 30 samples.

Given these data, we can consider the Messenger protocol very fast and reliable, with the only limitation of being sequential and queue-based without any possibility to set a priority, being used by the operative system as well.

The same considerations can be done for the remaining methods we took into consideration previously, the *create* and the *start*. Details on time needed to a message to trigger the call are shown in Tables 4.9 *(create)*, 4.11 *(start)* and the statistics on message delivery timings are shown in Tables 4.10 *(create)*, 4.12 *(start)*.

`btnCreate`	IN [ms]	OUT [ms]	δ(i-1)	δ
Messenger.send	1380.490	1380.787	-	0.297
Other operations executed for UI rendering, through message sending				
Message.dispatch	1393.090	1397.626	12.303	4.536
CustomService.CustomMessageHandler	1393.108	1397.607	0.018	4.499
BbqueService.BbqueMessageHandler	1393.126	1397.588	0.018	4.443
BbqueService.create	1393.145	1397.569	0.019	4.424
Time from "Messenger.send" end to "create" start			**12.358**	

Table 4.9: Button *Create* press: timings on the sample application. IN = instant method enters, OUT = instant method exits, δ(i-1) = IN(i)-IN(i-1), δ = OUT(i)-IN(i)

# samples	min [ms]	max [ms]	avg [ms]	σ	CI 99% [ms]
60	11.591	42.381	13.501	4.006	$\pm$1.334

Table 4.10: Time elapsed from "Messenger.send" exit to "create" enter, on the sample application BbqueActivity on 60 samples.

The data above confirm the generally small time overhead of the integration implementing choice: we will see later that the time needed to the activity-service paradigm to communicate is acceptable in the overall performance context.
Before continuing, at Table 4.13 the statistic data about the onRun callback maxi-

`btnStart`	IN [ms]	OUT [ms]	δ(i-1)	δ
Messenger.send	1822.847	1827.347	-	4.500
Other operations executed for UI rendering, through message sending				
Message.dispatch	1840.771	1842.979	13.424	2.208
CustomService.CustomMessageHandler	1840.794	1842.961	0.023	2.167
BbqueService.BbqueMessageHandler	1840.814	1842.943	0.020	2.129
BbqueService.start	1840.833	1842.925	0.019	2.092
Time from "Messenger.send" end to "start" start			**13.486**	

Table 4.11: Button *Start* press: timings on the sample application. IN = instant method enters, OUT = instant method exits, δ(i-1) = IN(i)-IN(i-1), δ = OUT(i)-IN(i)

# samples	min [ms]	max [ms]	avg [ms]	σ	CI 99% [ms]
30	12.586	53.410	15.861	7.261	$\pm$3.420

Table 4.12: Time elapsed from "Messenger.send" exit to "start" enter, on the sample application BbqueActivity on 30 samples.

mum rate are given: the elapsed time between two *onRun* callbacks was measured on 100 samples. Notice that testing performances are likely highly affected by the Android tracing system (which sometimes dramatically slows down performances) and by the emulated environment. All tests should be run on the real target platform, to have a more reliable statistic.

# samples	min [ms]	max [ms]	avg [ms]	σ	CI 99% [ms]
100	8.432	12.268	9.755	0.789	$\pm$0.204

Table 4.13: Time elapsed between two onRun executions on the sample application BbqueActivity on 100 samples.

With these data, we could consider as a limit case where the *onRun* method computing tends to zero, the processing rate as high as 100 operations per second. This is obviously a theoretical value (based on an emulation).

4.2.3 STHormFaceDetection Application: performance profiling

As mentioned in Section 3.3.2, the application that was used to test the integration of *Barbeque* is a *Face Detection* application, developed by *STMicroelectronics*.
We had the opportunity to run the Java application on the system simulator provided by STMicroelectronics [8], and some performance data were collected, and are shown in the following tables. The data we present here concern the mere method execution, but the actual time perceived by the user is bigger, because of other operation that are executed in the meanwhile.
We let the application run and process 30 frames, with and without the face detection filter, and results can be seen at Tables 4.14 and 4.15. In Figure 4.4 a graphical representation of the 30 samples is shown, with focus on the main processing methods and their execution elapsed time with respect to each *onRun*.

Function	min [ms]	max [ms]	avg [ms]	σ	CI 99% [ms]
`onRun`	166.254	209.550	189.052	14.706	±6.927
`decodeYUV`	132.859	158.004	142.547	5.650	±2.662
`FDetectRun`	1.271	4.766	1.6282	0.630	±0.297

Table 4.14: BbqueFDApplication - 30 frames processing with FD filter ON

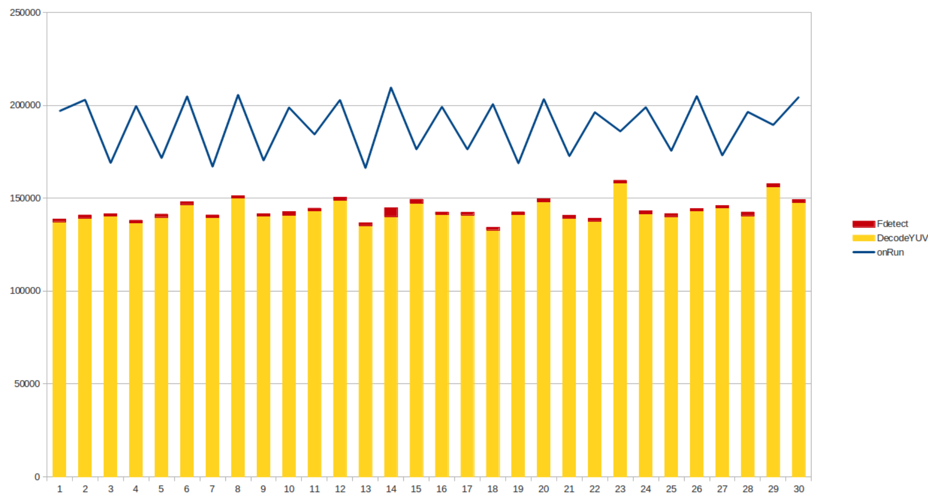


Figure 4.4: Performance of the DecodeYUV420SP and FDetectRun with respect to the `onRun`

As mentioned before, perceived execution time appears being much higher, and it empirically is around 3 to 4 seconds: this aspect is still under investigation.

Function	min [ms]	max [ms]	avg [ms]	σ	CI 99% [ms]
`onRun`	158.898	200.605	176.262	10.959	$\pm$5.162
`decodeYUV`	139.322	167.840	151.780	6.029	$\pm$2.840

Table 4.15: BbqueFDApplication - 30 frames processing with FD filter OFF

In conclusion, it's worth considering that all the *tracing* mechanism slew down the whole execution, therefore the data provided so far should be considered as a worst case and, in general, correct with regard to the proportions and not necessarily to the absolute values. The `onRun`, for instance, from the logcat, goes almost twice faster. More precise measures can be collected when the application will be running on the real platform.

Chapter 5

Conclusions

"Imagination is more important than knowledge."
Albert Einstein

During this work we analysed how it's possible to integrate the native *BarbequeRTRM* framework within Android by exposing its functionalities to the typical Android Java application environment. This goal was reached mainly by using the *JNI* interface, creating a sort of bridge between the two worlds. The main functions were mapped by an android *service* specifically designed to be extended by application-specific services that developers can design ad-hoc, embedding their core tasks.
Beside this, two applications were developed, to test this integration, a *BarbequeRTRM* testing application, and a *face detection* application, which was derived by an STMicroelectronics application, and properly customised to exploit our own custom service.
During the experimental session, we gathered some data to analyse the impact of the Java-code overhead, and to profile some design choices, like the Android messaging system, which we decide to adopt as communication channel between the application activity and the service: we determined a downside, which is due to the Android system messaging use, that queues all messages without priority. However, still this issue doesn't affect the computing performances executed within the service, being limited to the user-interaction code path.
This work, so far, provides the developers with a Java-based API that lets her ac-

cessing to the *BarbequeRTRM* framework, and some measures that can be used to consider possible future improvements, as well the overall performances of the system integration.

As future work, there's a wide range of possibilities as, for instance:

- *Real STHORM device testing*: our tests were run on an emulator, therefore the most significant step would be testing this solution on the real device, which will be actually available for January 2013. Measures on the real platform, which consists of 64 cores, will give the real feedback on the use of a resource manager like *BarbequeRTRM*.
- *Eclipse integration*: to have a better integration with the development framework (Eclipse IDE is used by default to develop Android applications) it's possible to implement an Eclipse plugin to make easier to developers the creation of their own *BarbequeRTRM compliant* services, for example with a *wizard* approach, which already prepares the main service code to be completed with the application specific functionalities.
- *Monitor application*: a Java application that uses the provided API to monitor and profile the system performances, as well as to control *BarbequeRTRM* whenever necessary.
- *Improve the STHORM Face Detection application*: implementing different messaging solutions to compare their performances, still considering that, in some cases, messaging impact can be a minor issue, especially if compared to the real computing time.

Appendix A

Tutorials

A.1 Tutorial1

The complete code from this tutorial can be found at the Github repository: 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 A.1, 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 A.1: 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 A.2

```
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 A.2: 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 A.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 A.1:**1** is natively implemented in Listing A.2:**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 A.2:**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 A.2:**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 A.2:**31**
- get the method identifier, through the `GetMethodID`, which performs a search 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 A.2:**38**
- call the method, on the caller object, passing the `methodID` Listing A.2:**46**

The code in Listing A.2:**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

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 A.1: Types correspondence Java/JNI and signatures

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

Table A.2: Signatures for objects and arrays

- `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 A.1. We will see how to declare complex descriptors, with strings, objects and arrays further.

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 A.3: 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

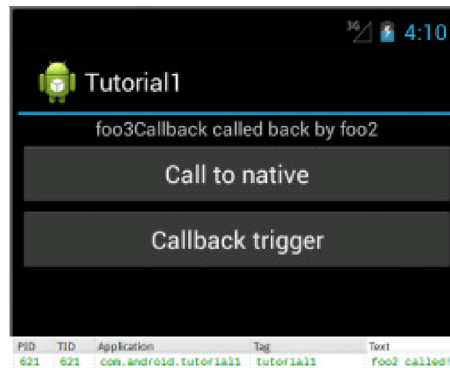


Figure A.1: Tutorial1 performing a callback

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 A.3: Android.mk for tutorial1.c

An example of the activity performing a callback can be found at Figure A.1: the output visible at the top of the screen comes from the Java method at Listing A.1:**5**

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.

A.2 Tutorial2

The complete code from this tutorial can be found at the Github repository: `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 A.1, we can find at Listing A.4 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 A.4: Part of Tutorial2Activity.java source code

At Listing A.4:**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 A.4:**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 A.4:**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 A.5), 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 A.5: 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 A.6 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 the proper function: `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   //[...]
```

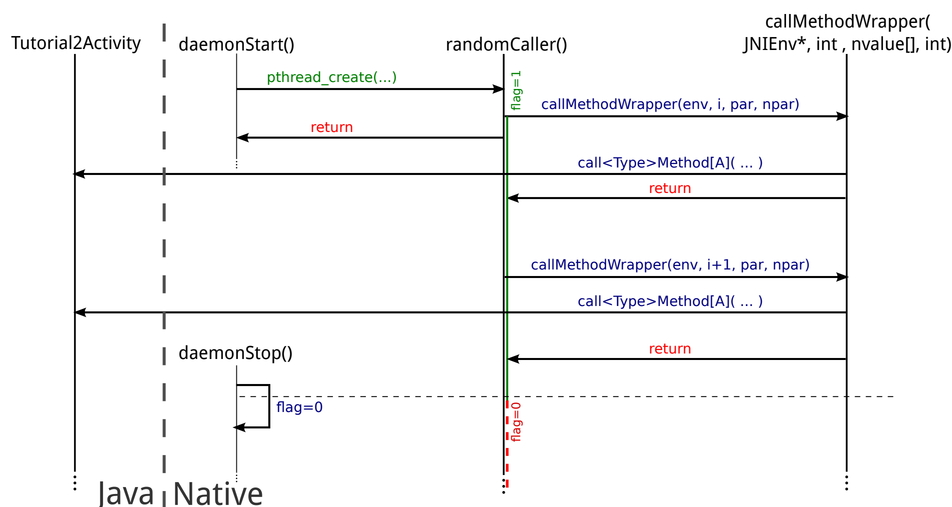


Figure A.2: Tutorial2 sequence diagram

```
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 A.6: 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 A.2. 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 A.7.

```
void *randomCaller() {
  flag = 1;
  JNIEnv *env;
  int isAttached = 0;
  int status = (*gJavaVM)->GetEnv(gJavaVM, (void **) &env,
          JNI_VERSION_1_4);
  if(status < 0) {
    LOGE("callback_handler: failed to get JNI environment,
          assuming native thread");
    status = (*gJavaVM)->AttachCurrentThread(gJavaVM, &env,
                                             NULL);
    if(status < 0) {
      LOGE("callback_handler: failed to attach current
          thread");
    }
  isAttached = 1;

  /*
  * callMethodWrapper(i++)
  * sleep(2);
  */

  if(isAttached)
    (*gJavaVM)->DetachCurrentThread(gJavaVM);
}
```

Listing A.7: tutorial2.c - thread attachment in randomCaller()

At Listing A.7:**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 the value `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 A.7:**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 A.7:**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

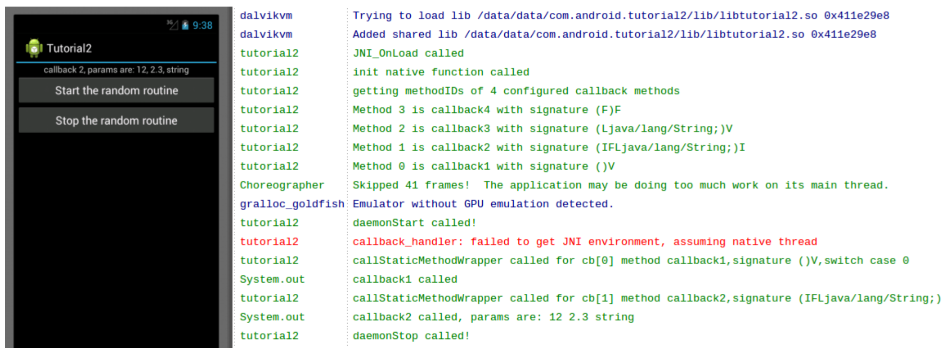


Figure A.3: Tutorial2 running example

(Listing A.7:**18-21**), hence the full code isn't presented in this paper, nonetheless it's important to stress the peculiarity of a calling method, `call<Type>MethodA()` Listing A.9:**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 A.8: at **line:11** the developer sets as input parameters the env variable, the ordinal number of the chosen callback method (Listing A.6:**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 A.8: Part of tutorial2.c - method calls in randomCaller()

The `callMethodWrapper` function is partially shown in Listing A.9: 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 A.9: Part of tutorial2.c - callingMethodWrapper

A screenshot of *Tutorial2* activity and logcat during the execution can be found at Figure A.3

A.3 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 tutorial can be found at Figure A.4: 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

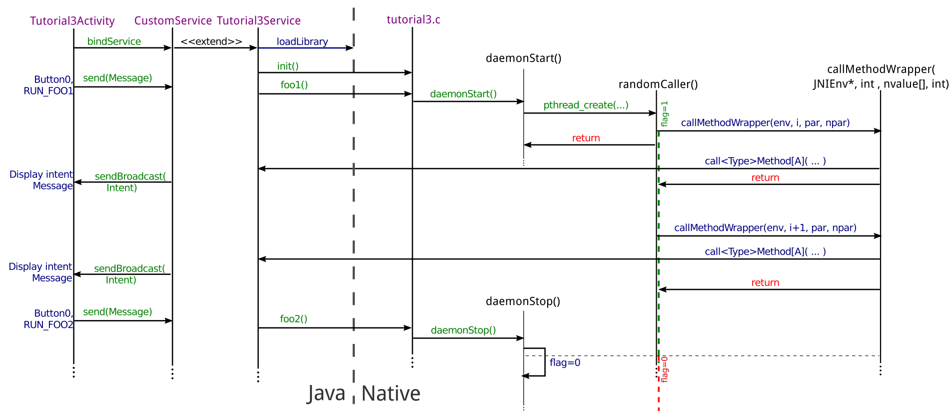


Figure A.4: Tutorial3 sequence diagram

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 A.10:**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 A.10: 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 A.11 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 A.11: Tutorial3Service.java - message handling

The `nMessenger` is the object the service will share with the activity. In Listing A.12 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 A.12: 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 A.12:**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 A.2 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 A.13 and Listing A.14.

```
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 A.13: 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 A.13:**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 A.14: 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 A.14**:10-12** the `IntentFilter` is initialised and it's registered to the `BroadcastReceiver` which is declared some lines beneath, at Listing A.14**: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 A.2.
In **Figures from A.5 to A.8** is shown the *traceview* log of the first call to `foo1()` from the `button0`, which sends a message of the class `Messenger` to the class `Tutorial3Service` which, in turn, calls the corresponding native function to spawn a thread, that then performs the callbacks.

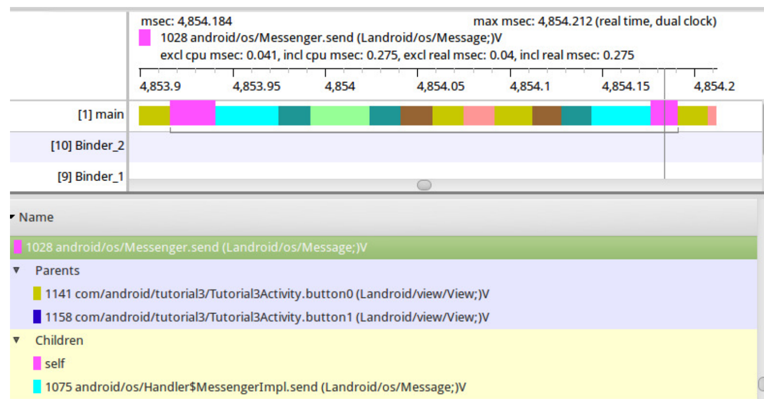


Figure A.5: Messenger class connected to Button0

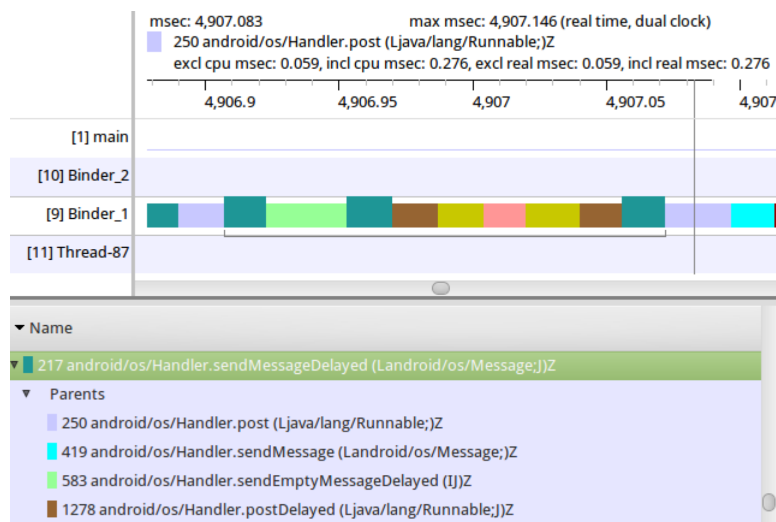


Figure A.6: Binder process handling the Message

A.4 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 A.15

```
1 package com.android.tutorial2;
2
```

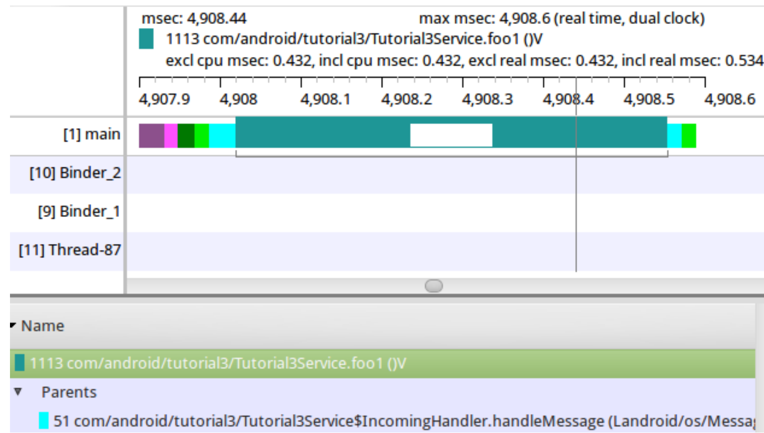


Figure A.7: IncomingHandler asked to call foo1 ()V on Tutorial3Service

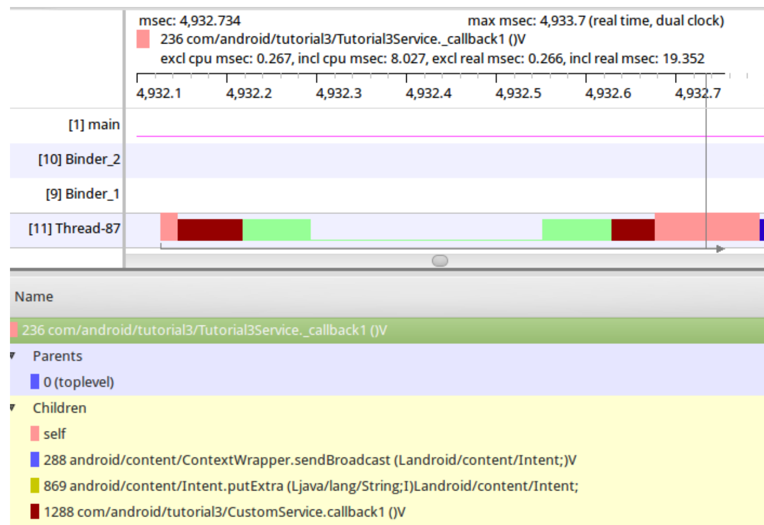


Figure A.8: Thread triggered by service performs _callback1 ()V

```
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 A.15: 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 follows:

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

Now, we have three Java arrays correctly initialised: we can create an object of the `Param` class, and use its constructor (A.15:8) to initialise it with these newly generated arrays:

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

As we saw in A.1, 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 A.15:**8**.

Bibliography

[1] M. GARGENTA, *Learning Android*, Oreilly Series, O'Reilly Media, Incorporated, 2011.

[2] S. LIANG, *The Java Native Interface: Programmer's Guide and Specification*, Addison-Wesley Java series, Prentice Hall, 1999.

[3] Z. MEDNIEKS, L. DORNIN, B. MEIKE, and M. NAKAMURA, *Programming Android: Java Programming for the New Generation of Mobile Devices*, Oreilly Series, O'Reilly Media, Incorporated, 2011.

[4] A. MUNSHI, B. GASTER, T. MATTSON, J. FUNG, and D. GINSBURG, *OpenCL Programming Guide*, OpenGL Series, Prentice Hall, 2011.

[5] K. YAGHMOUR, *Embedded Android*, Oreilly Series, O'Reilly Media, Incorporated, 2011.

[6] I. DARWIN, *Android Cookbook*, Cookbook Series, O'Reilly Media, Incorporated, 2012.

[7] P. BELLASI, G. MASSARI, and W. FORNACIARI, A RTRM proposal for multi/many-core platforms and reconfigurable applications, in *7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip, York, UK, 07/2012*, pp. 1–8, IEEE, 2012.

[8] L. BENINI, E. FLAMAND, D. FUIN, and D. MELPIGNANO, P2012: Building an Ecosystem for a Scalable, Modular and High-Efficiency Embedded Computing Accelerator, in *Design Automation Conference in Europe. Proceedings., 2*, 2012.

[9] D. MELPIGNANO, L. BENINI, E. FLAMAND, B. JEGO, T. LEPLEY, G. HAUGOU, F. CLERMIDY, and D. DUTOIT, Platform 2012, a many-core computing accelerator for embedded SoCs: Performance evaluation of visual analytics applications, in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, 2012.

[10] P. BELLASI, G. MASSARI, and W. FORNACIARI, Exploiting Linux Control Groups for Effective Run-time Resource Management, in *PARMA 2013 Workshop HiPEAC 2013, Jan 2013, Berlin, Germany*, 2013.

Webography

[11] Barbeque Open Source Project, http://bosp.dei.polimi.it.

[12] Android Open Source Project, http://source.android.com/.

[13] JNI Reference Example, http://marakana.com/s/jni_reference,1292.

[14] Remixing Android, http://marakana.com/s/post/1044/remixing_android.

[15] Using Camera API, http://marakana.com/forums/android/examples/39.html.

[16] Android API Guides, http://developer.android.com/guide.

[17] Android Training, http://developer.android.com/training.

[18] Java Native Interface, http://docs.oracle.com/javase/1.5.0/docs/guide/jni/.

[19] Vogella - Android and Eclipse dev, http://www.vogella.com/.

[20] JNI Examples for Android, http://android.wooyd.org/JNIExample/.

[21] Java Programming with JNI, http://www.ibm.com/developerworks/java/tutorials/j-jni/.

[22] Using JNI from a native activity, http://blog.tewdew.com/post/6852907694/using-jni-from-a-native-activity.

[23] About face detection on Android, http://www.smallscreendesign.com/2011/01/25/about-face-detection-on-android-part-1/.

[24] MyANdroid Book, http://home.comcast.net/ tomhorsley/book/.