ActorDroid
A distributed computing framework for mobile devices based on SCALA actors.

Pierre-André Mudry  Romain Cherix
University of Applied Sciences Western Switzerland
HES-SO Valais – Wallis
Sion, Switzerland
pierre-andre.mudry@hevs.ch / romain.cherix@hevs.ch

Abstract
With the processing power available on modern mobile devices, it becomes increasingly interesting to harvest this power for distributed computations. In this context, we developed a programming framework based on SCALA actors that runs on the Android operating system. This framework enables the distribution of an application consisting of communicating actors among multiple devices such as mobile phones or tablet PCs. Through two different experiments, we demonstrate that the flexibility of SCALA actors is a major advantage in terms of development of the framework but also in its usage thanks to various SCALA constructs such as functional programming and pattern matching.

Categories and Subject Descriptors D.1.3 [Programming techniques]: Concurrent programming

General Terms Algorithms, Performance

Keywords Actor, Scala, Android, distributed programming, stream processing

1. Introduction
The computing power of recent mobile devices is almost on par with desktop systems that were sold a few years ago. Thus, it is not uncommon today to find multi-core systems in mobile phones that run at frequencies of more than 1 GHz. A direct consequence of that processing power is that it enables the insertion of layers of abstraction between hardware and software to simplify programming and increase robustness. In this context, the Android operating system is increasingly becoming the system of choice for mobile devices, notably thanks to its openness.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ScalaDays '12 April 2012, London, UK.
Copyright © 2012 All rights reserved, HES-SO Valais 2012

1.1 Actors on Android
If the raw computing power available to mobile devices is still increasing thanks to the multiplication of cores, the question of how to use this power – i.e. how to program such systems – becomes prevalent. A possible solution to this question resides in the actors paradigm that was proposed by Hewitt in the 70s [8] and that found a new echo recently, notably thanks to the implementation of a first-class API in SCALA [6] providing a relatively simple yet powerful programming interface.

Actors can be used to write and run parallel programs that exchange information using messages. As such, they can be used (among others) in the context of mobile devices to realize systems such as sensor networks [11]. However, in and for themselves actors only propose an execution model without providing a complete infrastructure for running a complete distributed application.

1.2 Plan
In this article, we will show how was realized a SCALA programming framework based on actors which enables the programming of distributed applications on networked mobile devices.

In addition, whereas traditional embedded systems tend to rely mostly on C/C++ code, the power available in modern mobile phones and devices also enables the usage of more expressive programming languages. Thus, one major advantage of Android OS is that most applications can be written in Java and run on virtual machine – named Dalvik – which has been optimized for low-energy and memory limited devices. In the context of mobile devices, this solution ensures a good portability of applications on different hardware architectures whilst letting open the access to specific hardware features through Java native interface which is accessible using Android’s native development kit (NDK) (see for instance [12]). Finally, the fact that Android programming is based on JAVA also means that SCALA applications can also be executed on this OS.
devices running on Android. This paper is organized as follows: first we will describe the computational model that was used and the developed framework along with an example using a static network of devices. We will then demonstrate how this framework also enables to take into account dynamic topologies – where devices are added and removed during execution – to fully exploit the computing power that is available at a given time in a given location. The last part of this paper will consist of experimental results and a conclusion.

2. Context

Although the hardware in most desktop computers sold nowadays offers the possibility of executing multiple threads in parallel, a majority of the software written for these machines remains single-threaded. This situation can be explained, at least partly, by the complexity of writing multi-threaded applications: when multiple threads run in concert, the sequential model of programming breaks down. For example, the apparition of shared variables necessarily translates to concurrent accesses, which in turn requires to analyze program functions with a broader view than usual, the programmer having to think outside the scope of the function.

Of course, several programming languages propose techniques to reduce the complexity penalty of introducing parallelism: for instance, functional languages such as Haskell [9] do not include the notion of shared state nor that of sequential execution, and other languages, such as APL [10], remove the limitation of doing a single thing at a time.

However, all these languages are not widespread and they lack the acceptance and backing of the C/C++ or JAVA languages. Thus, the solution generally used to help parallel programming with more “traditional” languages relies on libraries, such as the Message Passing Interface (MPI) [16], that provide an abstraction layer between the parallel system where the application is executed and its software counterpart.

2.1 The stream processing model

The stream programming paradigm is a form of parallel processing that simplifies the development of certain types of applications, at the expense of some limitations. This model – on which the presented framework is based – makes use of kernel functions, represented with a directed acyclic graph (DAG), that are applied to a set of data called the data stream. In this DAG, the vertices represent the kernel functions and the edges account for the precedence relations taken by the data stream.

Multiple languages were developed with a form of this paradigm as a cornerstone, such as the StreamIT language at MIT [5, 17], the Brook language at Stanford [2] or Intel’s Ct (which is a proprietary language). These languages were mainly developed for specific so-called stream processors, such as the Stanford Imagine [11] and Merrimac [3], the MIT RAW or to a certain extent the Sony-Toshiba-IBM Cell [15]. However, the scope of these languages can be extended to model a whole range of execution platforms, such as the graphic processing units used in graphic cards, which also use a similar programming model in their shader language (for instance, Cg [4]).

The wide and increasing acceptance of the model comes from a flexibility that enables it to be used in a whole range of situations. Another advantage in its favor is that the kernel functions work locally on independent data. These two facts greatly simplify the development of parallel programs because they allow to bring back a more serial model, which helps maintaining a good overview of the problem tackled by the application. Thus, application writing becomes a two-fold process:

• During the first step, when the kernel graph is defined, the programmer can focus on splitting the application in different, parallel tasks. Because this separation helps keeping a linear and sequential view of the application, the development is simpler, closely resembling non-parallel code programming;

• The second step consists in programming the kernels present in the graph. Thanks to the explicit parallelism of the graph, each task separated from the others receives a clear, distinct input and has to produce an output with similar characteristics. The advantage of this clear role of the kernel is that the programming becomes completely sequential.

Finally, another advantage of the kernel model is that synchronization, allocation and communication do not need to be explicitly defined in the application code. These details can be hidden from the end-user, a property which can be quite useful, as we will see in the remainder of this paper.

Of course, all these advantages come at a cost and some limitations are inherent to the model, which cannot be applied to every kind of parallel program:

1. The communication cost as well as the execution time are not constant in every program and may greatly vary during execution. If this information is used, for example by scheduling algorithms, the user must ensure to model it correctly;

2. Not every application can be described as a kernel graph, especially without cycles;

3. The approach requires determining a certain granularity for parallelism to achieve the appropriate balance between computation and communication.

Despite these limitations, the model is sufficient for our requirements and provides enough advantages to reduce the impact of these drawbacks, as we will see.
2.2 Parallelism using actors

As we briefly discussed in the introduction, an actor is a computational model for parallel computing that encapsulates code, data and an execution environment (such as a thread for example). Developed from Hewitt’s model [8], the default SCALA actor library (see [7]) provide its own domain-specific language (DSL) and enable parallel execution of code in a message loop. Actors communicate asynchronously by passing immutable messages and, when such a message is received, the actor can act and then optionally reply to the sender with a message.

As such, SCALA actors closely match the stream processing paradigm. In addition, actors also support to be executed remotely, i.e. it is possible to have actors running in different virtual machines, possibly located on different processors or devices. Such remote actors transmit their messages using TCP–IP messaging.

In and for themselves, actors do not require a particular type of networking topology and the execution model of actors is relatively independent from its physical implementation. This both means that actors can be executed using different implementations (a feature which is used by the actor library with the thread pool for example) and that remote transmission of data between actors can be achieved using different techniques. In addition, parallelism is clearly exposed in the model (in the act or react methods) without exposing too much the end-user to delicate parallel programming structures such as locks or mailboxes. In that sense, the model is very interesting and SCALA greatly helps exposing its strengths, notably thanks to the DSL approach that is proposed. With it, implementing a complete actor is a matter of a few lines (see [6] for instance).

3. The ActorDroid framework

Atop the actors infrastructure and API, we built a distributed application framework for stream processing primarily focused on Android, although it is not limited to it. This framework, named ActorDroid, enables the execution and distribution of SCALA and JAVA programs among a dynamic network of mobile devices. Thanks to it, it is possible to create a dynamic and adaptive network of communicating nodes sharing their resources (computation time, special hardware such as a camera . . . ) as it is shown in Figure 1.

![Figure 1](image)

Figure 1. A sample ActorDroid application for image processing.

In such a DAG, each node represents an actor and the small stack represents the fact that messages can be buffered in a mailbox. The role of the framework is to allow the programmer to maintain such an abstract view of this structure, focused on the computation itself, but at the same time to execute all those actors on a single device or distribute them among multiple mobile devices, depending on the execution context chosen by the programmer. In the depicted example, the first device runs a single actor for taking photographs and another device executes two actors, one that processes the image and another that displays it within a GUI on the mobile device.

Within the framework, each node makes available its different resources through one or multiple services, each service corresponding to a computation or an action (like displaying an information on the screen or take a photograph). From an implementation perspective, each service corresponds to a SCALA actor, a solution which provides two distinct advantages:

1. Each service runs independently in its own execution environment (depending on the implementation: thread pool, lightweight process . . . ) ;
2. Each service can communicate with other services through messages. Those messages can either be transmitted to other actors that reside on the same device or on remote devices through TCP–IP.

Thanks to a discovery algorithm that we developed and that will be explained in the next section, the framework allows the creation of a dynamic network where devices containing one or multiple services can be added and removed dynamically, providing an infrastructure where computation nodes can be added on the go.

3.1 Motivation

To understand the interest of this framework, consider the following example: a company wishes to set up a security system to protect a working site. The system has to be smart enough to be able to detect intrusions using image processing but also has to be robust enough in case one or several components of the systems are malfunctioning or are destroyed. For this purpose, smart cameras running Android and networked together are installed. At regular interval, the cameras take a picture and send it to a central computer that is able to apply complex image filtering techniques to detect intrusion but also to backup safely those data. Simultaneously, the cameras exchange messages to determine the status of the system, possibly collaboratively deciding to send an alarm using a different network system (such as SMS messaging) when problematic situations occur.

Our framework would support such a scenario by providing a complete API to distribute the messages, dynamically detect device removal or addition, access the camera pictures and send messages using SMS.
3.2 Topology discovery algorithm

Because in this work we are dealing with mobile devices, which by definition, are not bound to a given location, the organization of the network can change dynamically. Thus, its topology (which nodes are present, how they are connected logically at software level) must be able to adapt to the insertion of new nodes or to their removal.

To achieve this behavior, we opted for semi-decentralized approach for distributing the computation and creating the list of the available nodes in the network. In this approach, a single node acts as a master node which is responsible for creating and maintaining a coherent view of the network. However, in order to maintain a scalable solution, we designed an algorithm for dynamically determining the role (master or slave) of each node present at a given time in the network. Strictly speaking, a master node is not required but it helps reducing the traffic between the nodes because the network information is managed by a single node.

The whole discovery protocol is contained in a SCALA remote actor that exchange information through the use of case classes and case objects. The output produced by the algorithm is a NetworkInformation object that is updated with relevant information during the discovery process (present nodes, their IP addresses, the services they provide).

With this algorithm, the master node is collaboratively chosen by the nodes among all the existing nodes and it is possible to elect a new master node if the existing one is removed. The master node selection and network discovery algorithm we implemented works as follows:

Slave insertion As soon as the framework is started on a mobile device, it listens to a specific UDP port for a multicast message sent by another node. If a message is received, this means that a master node already exists in the network. In that case, the new node sends a message containing all the pertinent information about it such as the provided services. The master then inform the members of the network of the changes that occurred.

Master selection If no message from the master is received after a random time \( t \), which is in the order of a few seconds, the new node entering the network emits a broadcast message containing its IP address and becomes the master node. Additional steps have been introduced to prevent the creation of multiple masters which is inherent to this method (as we proposed in [14]).

Master removal The master node also periodically broadcasts a message (every few seconds). If this message is not received by slaves in a given time lapse, the master selection mechanism starts again.

Slave removal Slave nodes answer the master broadcast messages with an alive message. When they quit the network, they also send an appropriate message to the master which then updates its network information. Improper node removal (nodes that quit without registering) is detected and if the master node does not receive messages from a node in a certain time lapse. In every case, removal is notified to all the members of the network.

With this dynamic topology algorithm, nodes can be added dynamically and the global computation can then adapt. For instance, if new worker nodes (i.e. nodes that provide computation services) arrive, the computation can adapt automatically by being also distributed to those new nodes. This approach is partially similar to the self-scaling stream processing approach that we proposed in [13]. It will be demonstrated in section 5.

3.3 Actor-based services

Apart from the distributed network discovery algorithm, another important feature of the framework is the creation of services by the nodes. As we mentioned earlier, a node can host one or multiple services that correspond either to a computation or to an action (e.g. taking a photograph, display a message on the screen). In addition to their behavior, services also require a non negligible part of logics in order to advertise their capabilities to other nodes (which is achieved in conjunction with the network manager actor of the node).

In order to let the program focus on the objectives of its application and attain a high level of abstraction with respect to the service implementation itself, every service inherits from this class which contains the necessary code to integrate new services to the computing framework. This class, named SuperService, notably informs the framework of the existence or removal of the service. Every implemented service from the end-user should then inherit from this class and define its own serializable objects (which are case classes or case objects) containing all the salient information required. Behavior of the service upon reception of messages is then defined in a method that is inherited from the abstract method theAction(any : Any).

As an example, a service displaying the text received embedded in a StringMessage class can be implemented as follows:

```
// The information token to transmit between services
case class StringMessage(val msg : String)

class Print(
  name : String, port : Int, symbol : Symbol,
  display : (AnyRef) ⇒ Unit)
  extends SuperService(name, port, symbol){

  startActor

  // Behaviour of Print service for incoming messages
  override def theAction(any : Any) ⇒ any match {
    case StringMessage(msg : String) ⇒ display("Received: " + msg)
    case _ ⇒ display("Received: an unknown message")
  }
}
```

That code is sufficient to create and register a Print service on the network. Instantiating this class will start the corresponding actor and make it available to the whole network. Following code shows how the created services instances are
accessible via the NetworkInformation object (here the service receives the message locally, but remote functionality is similar):

```scala
// Create service instance on the nexus1 device
new Print(printService, 9010, 'printService, println)

// Send a StringMessage to Nexus1's "printService"
NetworkNode.friends("nexus1").service(printService) !
StringMessage("This is a message to display")
```

In the next section, we will demonstrate how this framework can be used, first in the context of mobile image processing and then in the domain of distributed computing.

4. Static topology

This first application of our framework focuses on the distribution of an image processing among a static network of mobile devices, i.e. the number of nodes is known a priori and do not change during execution.

In this application, the task devoted to each device is determined before execution as in the following example: a mobile device takes pictures of the surroundings of a building and sends them to processing devices that filter the images to detect unauthorized personnel. If an intrusion is found, an alarm is displayed on a screen of a mobile device located in a control room. In this scenario, the number and role of each device is static and the order of the operations to be performed is serial.

Implementing such a scenario with our framework is relatively straightforward: thanks to the master discovery algorithm, a node is chosen to dispatch the topology of the network to every node. This done, we define a DeviceID class that describes the devices (a method relatively similar to defining an enumeration):

```scala
trait DeviceID{
  case object one extends DeviceID { override def toString = "Nex1" }
  case object two extends DeviceID { override def toString = "Nex2" }
  case object three extends DeviceID { override def toString = "Nex3" }
  case object four extends DeviceID { override def toString = "Nex4" }
}
```

The sequence of the operations as well as the relation for the couple service–device is determined in the following class:

```scala
case class InstructionList(
  deviceSeq: Seq[DeviceID],
  actionSeq: Seq[ActionProcess],
  index: Int,
  imageData: Array[Array[Int]]
)
```

The first attribute contains a sequence of the devices that are implied in the image processing. The second attribute contains the operations that each device has to perform. As an object of this class is carried along the computation, the index reflects the step of the computation that is currently performed and is used to determine the next destination of the payload, which in this case is an image (the last attribute of the class).

Finally, a class ActionProcess contains the various possible actions that are implemented by the different services:

```scala
trait ActionProcess {
  case object ActionProcess {
    case object BlackAndWhiteFilter extends ActionProcess
    case object SobelFilter extends ActionProcess
    case object ThresholdFilter extends ActionProcess
    case object DisplayImage extends ActionProcess
  }
}
```

4.1 Experimental results

Figure 2 depicts how the aforementioned scenario can possibly be modeled with a DAG.

![Figure 2. Static image processing application.](image)

For the experimental results, our testbed consisted of five Samsung Nexus S phones (running Android OS 2.3.4). These devices were connected to a 802.11g WiFi network with WPA2 PSK cryptography. For the development of SCALA programs under Android, we used the 2.9.1 version of SCALA and a standard ANT-based tool chain with the notable difference that the generated bytecode is shrank with ProGuard. This last step is required to reduce the size of the SCALA library, which is relatively big in the context of mobile devices, and also to remove all unused code. The result is then converted into the Dalvik VM format before being copied to the different mobile devices. Concerning the size of the image, for the sake of simplicity we did not capture a picture with the camera for every trial but we used a stored 512x512, 24-bit image.

Even though the results are correctly computed, the performance of the implemented solution using five phones (each one running a service) is not faster than when using a single phone running the five services in parallel as shown in Table 1. Please note that in this Table, the npictures parameter corresponds to the number of pictures that are sent

in series for showing the influence of the communication layer of the framework on the results. In other words, the

time taken to transmit the information is relatively too high compared to the time taken for the computation. This can be
totally explained by the fact that WiFi transmission adds a significant latency time (a round-trip with an image without
any processing takes roughly one second). In addition, the maximum throughput of the devices used is relatively low
(16 MBit per second measured in standard Android benchmarks), which also plays a non-negligible role in data-driven
applications such as the one described here.

Even though the raw performance of the application does not show a decrease of total time, the static topology is still
valuable, notably in situations where services leverage the special hardware capabilities of the devices (camera, hard-
drive, database…).

5. Dynamic topology and routing

In this section, we present how the addition of a dynamic task scheduler to the framework enables the distribution of com-
putations that dynamically takes advantage of the resources present in the network. The challenges of the dynamic task
scheduler we faced during its implementation are two-fold:

• First, new nodes sharing their resources that are added during the computation can be used automatically to dis-
  tribute a running computation on them. The computation scheduler takes advantage of all the present nodes us-
ing a round-robin work allocation policy to distribute the workload among the non-busy node. As soon as a node
provides the application computation service and is free, the scheduler sends it a new chunk of work;

• Second, the framework allows the dynamic removal of computing nodes. This means that the scheduler is in-
  formed of removal of nodes and that it must be able to resume proper functionality and insure correctness of the
result (resubmit chunk lost for example). Depending on the computation algorithm taking place, this can be com-
plicated. However, the stream processing model we use simplifies this task.

An example of a DAG for distributed computing is shown in Figure 3.

<table>
<thead>
<tr>
<th>Number of pictures sent</th>
<th>Time [ms]</th>
<th>Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 device</td>
<td>5 devices</td>
</tr>
<tr>
<td>1</td>
<td>2891</td>
<td>9552</td>
</tr>
<tr>
<td>2</td>
<td>5151</td>
<td>12264</td>
</tr>
<tr>
<td>3</td>
<td>8140</td>
<td>13716</td>
</tr>
<tr>
<td>4</td>
<td>10371</td>
<td>17805</td>
</tr>
<tr>
<td>5</td>
<td>13214</td>
<td>18980</td>
</tr>
<tr>
<td>6</td>
<td>15408</td>
<td>21821</td>
</tr>
</tbody>
</table>

Table 1. Time measurements, static image processing.

Here, the dispatcher nodes distributes the workload among different devices that all implement the same worker
service. Among the possible work allocation policies that are possible, we implemented the following two variants.
Depending on the distributed algorithm that should be run, the work can either be divided in as many chunks as there
are nodes available (mode 1) or the work can be divided into an arbitrary number of chunks that are then distributed
to every non-busy nodes present in the network (mode 2).

5.1 Experimental results

To demonstrate the usage of our framework in the context of distributed computing, we implemented an algorithm to
estimate the value of π. The algorithm we used is based on the computation of the following Gregory-Leibniz series:

$$\pi = 4 \cdot \sum_{k=0}^{\infty} \frac{(-1)^k}{2k + 1}$$

Even though this series converges slowly and that more efficient ways to approximate π are available, this algorithm
has the important characteristic of being very easy to be sliced into many parts that can then be distributed to multiple
workers. As such, it is a good candidate for demonstrating the correctness and robustness of the presented framework.

For this algorithm, we used mode 1 in which the dispatcher node divides the total amount of work into equal parts for
each present device. In this experiment, each worker nodes receives the information of the range that should be computed for the estimation of π. For each worker, the result is then summed by the dispatcher node. The frame-
work performance is depicted in Figure 4.

This figure shows that the performance of the various measurements (represented by a cross) closely follows
$$f(x) = k / x$$, with k a constant, for an approximation of π using 36’000’000 iterations for the sum. In comparison the
image processing experiment where data is the bottleneck, this computation requires far less communication between
the nodes and shows the validity of the chosen approach.

6. Conclusion

In this article, we discussed how a SCALA based programming framework for distributed computing could be used
in different contexts. Of course, the described framework does not encompass all the possible applications for dis-
tributed systems and is especially tailored for independent tasks. Nevertheless, we demonstrated that actors are a robust


