Technical Report

Experiences on the Implementation of a Cooperative Embedded System Framework

Cláudio Maia

Luis Miguel Nogueira

Luis Miguel Pinho

HURRAY-TR-100605

Version:

Date: 06-29-2010

Experiences on the Implementation of a Cooperative Embedded System Framework

Cláudio Maia, Luis Miguel Nogueira, Luis Miguel Pinho

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: crrm@isep.ipp.pt, luis@dei.isep.ipp.pt, lmp@isep.ipp.pt

http://www.hurray.isep.ipp.pt

Abstract

As the complexity of embedded systems increases, multiple services have to compete for the limited resources of a single device. This situation is particularly critical for small embedded devices used in consumer electronics, telecommunication, industrial automation, or automotive systems. In fact, in order to satisfy a set of constraints related to weight, space, and energy consumption, these systems are typically built using microprocessors with lower processing power and limited resources.

The CooperatES framework was thus specified in this context, allowing resource constrained devices to collectively execute services with their neighbors in order to fulfil the complex QoS constraints imposed by users and applications. This paper presents a prototype implementation of the framework within the Android platform, detailing the needed extensions to the Android's architecture to handle the formation of a coalition of cooperative nodes. Building on this experience, the paper points out future perspectives for the development of real-time applications in the Android platform.

Experiences on the Implementation of a Cooperative Embedded System Framework

Cláudio Maia
CISTER Research Centre
School of Engineering of the Polytechnic Institute of Porto
Porto, Portugal
crrm@isep.ipp.pt

Luís Nogueira
CISTER Research Centre
School of Engineering of the Polytechnic Institute of Porto
Porto, Portugal
lmn@isep.ipp.pt

Luís Miguel Pinho
CISTER Research Centre
School of Engineering of the Polytechnic Institute of Porto
Porto, Portugal
lmp@isep.ipp.pt

ABSTRACT

As the complexity of embedded systems increases, multiple services have to compete for the limited resources of a single device. This situation is particularly critical for small embedded devices used in consumer electronics, telecommunication, industrial automation, or automotive systems. In fact, in order to satisfy a set of constraints related to weight, space, and energy consumption, these systems are typically built using microprocessors with lower processing power and limited resources.

The CooperatES framework was thus specified in this context, allowing resource constrained devices to collectively execute services with their neighbours in order to fulfil the complex QoS constraints imposed by users and applications. This paper presents a prototype implementation of the framework within the Android platform, detailing the needed extensions to the Android's architecture to handle the formation of a coalition of cooperative nodes. Building on this experience, the paper points out future perspectives for the development of real-time applications in the Android platform.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming — Distributed programming; D.2.11 [**Software Engineering**]: Software architectures; J.7 [**Computers in Other Systems**]: Real-Time

Keywords

Distributed Real-Time Embedded Systems, Cooperative Computing, Android

1. INTRODUCTION

During the past years the embedded device industry has faced a huge growth and the tendency is to grow even more in the next years [22]. Following this tendency, new applications, functionalities and more diverse devices are becoming available to the general audience in a fast pace, therefore bringing new technological and scientific challenges. Among these challenges is the ability to cooperatively execute resource demanding applications in such heterogeneous, open, and dynamic environments while meeting non-functional requirements that otherwise would not be met by an individual execution.

An increasing number of real-time applications need a considerable amount of computation power and are pushing the limits of traditional data processing infrastructures [23]. Consider, for example, the real-time stream processing systems described in [14, 6, 21]. The quantity of data produced by a variety of data sources and sent to end systems to further processing is growing significantly, therefore demanding more processing power. The challenges become even more critical when a coordinated content analysis of data sent from multiple sources is necessary [6]. Thus, with a potentially unbounded amount of stream data and limited resources, some of the processing tasks may not be satisfyingly answered even within the users' minimum acceptable QoS levels [21].

Several studies in computation offloading propose task partition/allocation schemes that allow the computation to be offloaded, either entirely or partially, from resource constrained devices to a more powerful neighbour [24, 8, 11]. These works conclude that the efficiency of an application execution can be improved by careful partitioning the workload between a device and a fixed neighbour. Often, the goal is to reduce the needed computation time and energy consumption [9, 18, 20, 4, 10] by monitoring different resources, predicting the cost of local execution and that of a remote one and deciding between a local or remote execution. However, most of the work in this direction is limited to the case where there is only one resource-limited device and one relatively more capable neighbour to offload computation to. Also, none of these works supports the maximisation of each user's specific Quality of Service (QoS) preferences while offloading computation among sets of heterogeneous nodes.

This is the challenge addressed by the CooperatES (Cooperative Embedded Systems) framework [16], a QoS-aware framework that facilitates the cooperation among neighbour nodes when a particular set of user-imposed QoS preferences cannot be satisfied by a single node. This paper describes

the ongoing prototype implementation of the CooperatES framework in the Android mobile platform [2], detailing the needed extensions to the Android's architecture to handle the formation of a coalition of cooperative nodes as formally described in [16]. This allows to understand the limitations of the Android's platform, and better identify the direction to follow for better support real-time applications in Android.

Android [2] was made publicly available during the fall of 2008 and is gaining strength both in the mobile industry and in other industries, with different hardware architectures (such as the ones presented in [3] and [12]). The increasing interest from the industry arises from two core aspects: its open-source nature and its architectural model. Nevertheless, there are features which have not been explored yet, as for instance the suitability of the platform to be used in Open Real-Time environments [13]. Taking into consideration works made in the past such as [19, 5], either concerning the Linux kernel or Virtual Machine environments, there is the possibility of introducing temporal guarantees allied with Quality of Service (QoS) guarantees in each of the aforementioned layers, or even in both, in a way that a possible integration may be achieved, fulfilling the temporal constraints imposed by the applications. This integration may be useful for multimedia applications or even other types of applications requiring specific machine resources that need to be guaranteed in an advanced and timely manner. Thus, taking advantage of the real-time capabilities and resource optimisation provided by the platform.

The remainder of this paper is organised as follows: Section II briefly presents the CooperatES framework. Section III describes the core features of the Android architecture. The proposed extensions to enable the formation of coalitions of cooperative heterogeneous devices with Android are detailed in Section IV. Section V points out a possible direction to include real-time behaviour on Android. Finally, Section VI concludes this paper.

2. THE COOPERATES FRAMEWORK

The CooperatES framework is primarily focused in open and dynamic environments where new services can appear while others are being executed, the processing of those services has associated real-time execution constraints, and service execution can be performed by a coalition of neighbour nodes. Nodes may cooperate either because they cannot deal alone with the resource allocation demands imposed by users and services or because they can reduce the associated cost of execution by working together.

The framework is composed by several components, as depicted in Figure 1. It is not our objective to detail the framework model presented in the figure, as it has already been done in [16]. However, a brief description helps to fully understand the extensions to the Android's architecture detailed in Section IV.

When a QoS-aware service arrives to the system, it requests execution to the framework through the *Application Interface*, thus providing explicit admission control, while abstracting the underlying middleware and operating system.

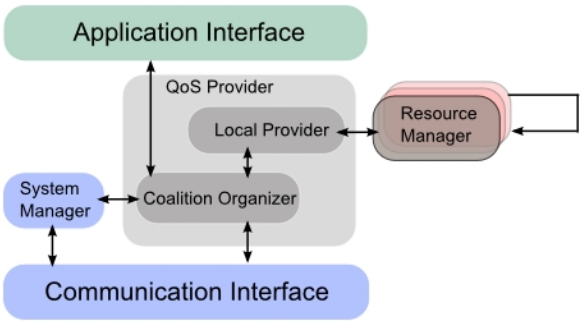

Figure 1: CooperatES Framework

Users provide a single specification of their own range of QoS preferences for a complete service, ranging from a desired QoS level to the maximum tolerable service degradation, specified by a minimum acceptable QoS level, without having to understand the individual tasks that make up the service. It is assumed that a service can be executed at varying levels of QoS to achieve an efficient resource usage that constantly adapts to the devices' specific constraints, nature of executing tasks and dynamically changing system conditions.

The service request will then be handled by the *QoS Provider*, which in turn is composed by the *Local Provider* and *Coalition Organiser* components. The *Local Provider* is responsible for determining if a local execution of the new service is possible within the user's accepted QoS range, by executing a local gradient descent QoS optimisation algorithm, quadratic in the number of tasks and resources and linear in the number of QoS levels. The goal is to maximise the satisfaction of the new service's QoS constraints while minimising the impact on the current QoS of previously accepted services [16].

Rather than reserving local resources directly, it contacts the *Resource Managers* to grant the specific resource amounts requested by the service. Each *Resource Manager* is a module that manages a particular resource, and interfaces with the actual implementation in a particular system of the resource controller, such as the network's device driver, the CPU scheduler, or even with the software that manages other types of resources (such as memory).

The CooperatES framework differs from other QoS-aware frameworks by considering, due to the increasing size and complexity of distributed embedded real-time systems, the needed trade-off between the level of optimisation and the usefulness of an optimal runtime system's adaptation behaviour. Searching for an optimal resource allocation with respect to a particular goal has always been one of the fundamental problems in QoS management. However, as the complexity of open distributed systems increases, it is also increasingly difficult to achieve an optimal resource allocation that deals with both users' and nodes' constraints within an useful and bounded time. Note that if the system adapts too late to the new resource requirements, it may not be useful and may even be disadvantageous.

This idea has been formalised using the concepts of anytime

QoS optimisation algorithms [16], in which there are a range of acceptable solutions with varying qualities, adapting the distributed service allocation to the available deliberation time that is dynamically imposed as a result of emerging environmental conditions [15].

If the resource demand imposed by the user's QoS preferences cannot be locally satisfied, the coalition formation process is handled to *Coalition Organiser*. This component is responsible for broadcasting the service's description, the user's quality preferences, evaluating the received service proposals, and deciding which nodes will be part of the coalition.

Thus, there will be a set of independent blocks to be collectively executed, resulting from partitioning a resource intensive service. Correct decisions on service partitioning must be made at runtime when sufficient information about workload and communication requirements become available [24], since they may change with different execution instances and users' QoS preferences.

The *Coalition Organiser* interacts directly with the *System Manager* to know which nodes are able to participate in the coalition formation process. Therefore, the *System Manager* is responsible for maintaining the overall system configuration, detecting QoS-aware nodes entering and leaving the network, and managing the coalition's operation and dissolution.

3. ANDROID'S ARCHITECTURE

Android is an open-source software architecture provided by the Open Handset Alliance [1], a group of 71 technology and mobile companies whose objective is to provide a mobile software stack. It presents some positive features, such as: (i) its software licence; (ii) the target devices in which Android can be run, including mobile devices and devices based on x86 architecture [3], useful to prove the heterogeneous capabilities of the proposed framework; and (iii) its Linux-based architecture.

Android's main limitation lays on the real-time support. Hence, in order to overcome this limitation, the Capacity Sharing and Stealing (CSS) algorithm [15] is being implemented in the Linux kernel in an ongoing parallel preliminary prototype, which is included in the proposed framework implementation. The implementation of the real-time features on Android's Virtual Machine (VM) is also part of that work. The work presented in this paper is based on the standard Android architecture as it is provided by the Open Handset Alliance, and already allows to better understand the existent limitations and how Android should be augmented to better support real-time applications.

The Android stack includes an operating system, middleware and out of the box applications that can be used by the end user. It comes with a Software Development Kit (SDK) and a Native Development Kit (NDK), which provide the tools and Application Programming Interfaces (APIs) needed by the developers, in order to develop new applications for the platform. The applications can be developed using the Java programming language, if the developer uses the SDK or, on the other hand, C/C++ programming language in the case of using the NDK. In terms of features, Android incorporates the common features found nowadays in any mobile platform, such as: application framework reusing, integrated browser, optimised graphics, media support, network technologies, etc.

The Android architecture, depicted in Figure 2, is composed by the following layers: Applications, Application Framework, Libraries, Android Runtime and finally Linux kernel.

Starting by the uppermost layer, the Applications layer, it provides the core set of applications that are commonly shipped out of the box with any mobile device, i.e. Browser, Contacts, Phone, among others.

The next layer, the Application Framework, provides the framework APIs used by the applications on the uppermost layer. Besides the APIs, there is a set of services that enable the access to the Android's core features such as graphical components, information exchange managers, event managers and activity managers, as examples.

Below the Application Framework layer, there is another layer containing two important parts: Libraries and Android Runtime. The libraries provide core features to the applications. Among all the libraries provided, the most important are *libc*, the standard C system library tuned for embedded Linux-based devices; the media libraries, which support playback and recording of several audio and video formats; graphics engines; fonts; a lightweight relational database engine and 3D libraries based on OpenGL ES.

Regarding the Android Runtime, besides the internal core libraries, Android has its own virtual machine named Dalvik. This virtual machine was designed to be able to perform optimisations for minimal memory footprint as well as mapping the applications running on the uppermost layer to own processes, with its own instances of the virtual machine. This means that each application has its own address space, by running in a separate process in its own virtual machine instance. Dalvik is capable of running multiple instances of virtual machines and relies on the Linux kernel for features such as thread management and memory management.

The Linux kernel, version 2.6, is the bottommost layer and is no more than an hardware abstraction layer that enables the interaction of the upper layers with the hardware layer via device drivers. Furthermore, it also provides the most fundamental system services such as security, memory management, process management and network stack.

4. PROPOSED EXTENSIONS

The proposed extensions to the standard Android's architecture are depicted in Figure 3. These extensions enable the formation of coalitions of heterogeneous devices for cooperative QoS-aware execution of resource intensive services. The real-time features are supported by the Linux kernel and the CSS scheduling algorithm [15]. Although one of the main limitations of Android, in the scope of our work, is the lack of real-time support, there are other important ones such as resource monitoring or even QoS support that should be handled by the framework.

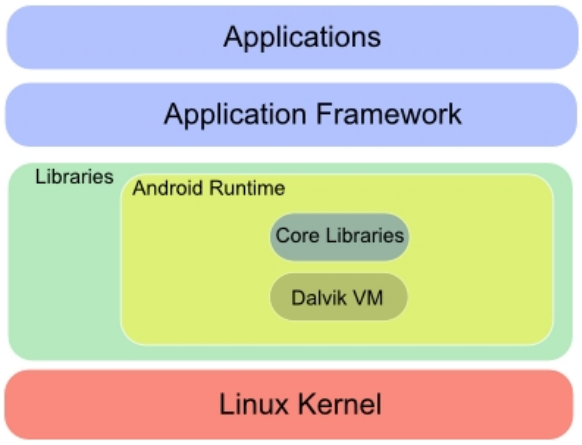

Figure 2: Android Architecture

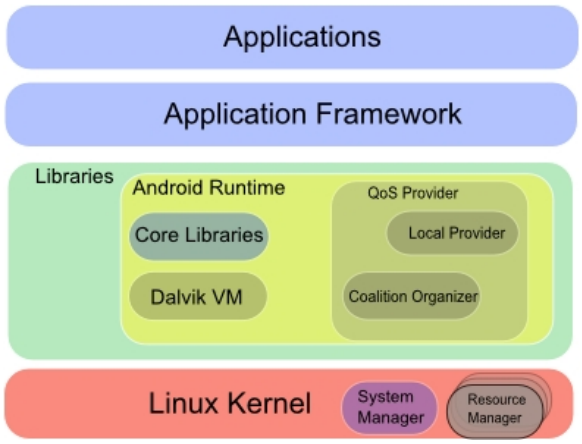

Figure 3: CooperatES Architecture

The Android architecture does not provide mechanisms to control the resource demands imposed by applications. Therefore, and to overcome this limitation, *Resource Managers* are used for handling singular resources, i.e. memory, cpu, disk, etc. Due to the dynamic nature of the framework, the amount of available resources has a direct impact on the decisions taken by the *QoS Provider*, which will decide whether to form a coalition of cooperative nodes or not, based on actual resource values.

Other limitation found in the Android's architecture is the lack of a component that is able to manage the communication schemes between different nodes. These communication schemes are important to the framework due to the fact that it permanently needs to handle all the network operations with respect to the cooperation process among nodes. This limitation was surpassed with the inclusion of the *System Manager* component.

All the components provide a socket interface, and therefore, this approach enables the communication between components located at different layers in the Android's stack. Without following this approach, it would not be possible to have a direct communication path between the *QoS Provider* and either the *Resource Managers* or *System Manager*, due to the internal design of the Android's architecture.

The *QoS Provider* runs seamlessly integrated in the Android Runtime as a natural part of its architecture. The main advantage provided by this integration is the ability of having a natural support for QoS in the architecture, which allows a decision to be taken on the best QoS for an application. There are two execution options available: (i) local execution or (ii) cooperative distributed execution. In either case, applications are processed in a transparent way for the user, as users are not aware of the exact distribution used to solve the computationally expensive applications.

During a device's boot time, the Linux kernel boots, as well as the *Resource Managers* and *System Manager*. These will run as kernel services and handle all the requests through a socket interface. It is important for the framework to be able to control resource usage through *Resource Managers* and also to be able to maintain the overall system configuration with the *System Manager*. As such, we have decided to put these components as near the device drivers as possible. The communication between the components will be socket-based, as previously mentioned.

```
if(startQoSProvider() != 0)
{
        LOGE("Unable to start QoS Provider\n");
        goto bail;
}

/* start the virtual machine */
if (startVm(&mJavaVM, &env) != 0)
        goto bail;
...
int AndroidRuntime::startQoSProvider()
{
    return createQoSProvider();
}

int createQoSProvider() {
    pthread_t threadID;

    LOGI("createQoSProvider was called\n");
    /* Create client thread */
    pthread_detach(pthread_self());
    if (pthread_create(&threadID, NULL,
         putSocketUpAndListen, NULL) != 0)
        LOGI("QoSProvider pthread_create() failed");
    return 0;
}
```

Listing 1: QoS Provider instantiation

Then, the Android Runtime is started via a process named Zygote. This process is responsible for the pre-initialisation of the Dalvik VM instance, which is used later on to spawn new VM instances with a partially initialised state. Whenever a new application is started, the main VM instance is forked via the Linux kernel and a new process is created. This process is handled by the VM itself and it is assigned to the new application. It is important to mention that Zygote is responsible for the initialisation of the core Java classes existing at the Application Framework level, which handle all the basic needs of the Android applications. The use of Zygote reduces the application's startup time, as most of the Android's core Java classes are already in a running

state when a user desires to start an application.

Just before starting the VM, the runtime starts the *QoS Provider*, through the *startQoSProvider()* method, with its internal components attached – the *Local Provider* and the *Coalition Organiser*. The code listing for this step in presented in Listing 1. As *QoS Provider* runs as if it natively belongs to the runtime itself, this step is strategic. Our intention is to provide QoS management to applications, where each application is mapped into an instance of the Dalvik VM and, at the same time, to a Linux process. This way, we are able to provide the application with an instance of a *QoS Provider* that is capable of handling each application's QoS constraints.

Zygote continues and starts up all the remaining components, beginning with Dalvik VM, then the core classes on the Application Framework and finally, the phone applications.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
  <application ...>
    <activity ...></activity>
  </application>
<qos-parameters>
  <dimensions>
    <dimension>
      <dimension-name>Video
          Quality</dimension-name>
        <attributes>
          <attribute>
            <attribute-name>color
                depth</attribute-name>
              <attribute-values
                domain="discrete"
                datatype="integer">1,3,8,16,24
              </attribute-values>
              </attribute>
            <attribute-name>sampling
                rate</attribute-name>
                <attribute-values
                  domain="discrete"
                  datatype="string">SQCIF,QCIF,CIF
                </attribute-values>
            </attribute>
           <attribute>...</attribute>
        </attributes>
     </dimension>
     <dimension>...</dimension>
  </dimensions>
</qos-parameters>
</manifest>
```

Listing 2: AndroidManifest.xml

Note that Android applications are dynamically able to use the regular Android stack, if no QoS management is required, or use the proposed extensions to satisfy their QoS requirements, which can be added to the *AndroidManifest.xml* file. This file contains the definition of an application, e.g. the declaration of the libraries, components, and specific permissions required or handled by the application. Listing 2 shows an example of this file, considering the specification of the application's QoS requirements.

Basically, each application can specify its QoS requirements through a set of quality dimensions, as proposed in [16]. Each quality dimension must specify a name, and a set of attributes associated to the quality dimension being specified. Each attribute also has an identifier, and a set of quality values. These values have a data type associated- *float*, *integer* or *string*; a domain - discrete or continuous - and finally, the attribute values that will be used as a reference by the *QoS Provider*, in order to perform the maximisation process with the application of the anytime algorithms. Each application may define an infinite set of dimensions according to its QoS needs.

```
else if (tagName.equals("qos-parameters"))
{
        if (!parseQoS(pkg, res, parser, attrs, flags,
            outError))
        {
                return null;
        }
        else
        {
                pkg.usesQoS = true;
                pkg.applicationInfo.setQoS(pkg.
                    qosDimensions);
                pkg.applicationInfo.usesQoS = true;
        }
...
}

private boolean parseQoS(...)
{
        if (tagName.equals("dimensions"))
        {
                ArrayList <QoSDimension> dimArray =
                    parseDimension(...);
        }
}
...
private ArrayList <QoSDimension> parseDimension(...)
{
        ArrayList <QoSDimension> dimensionArray =
            new ArrayList<QoSDimension>();
        QoSDimension tmpDimension;

        ...

        // each TAG is parsed and if everything is looks
            good
        // add the dimension to the array
        dimensionArray.add(tmpDimension);
...
}
```

Listing 3: QoS parameter parsing in PackageParser.java

By allowing the possibility to define the QoS requirements in an XML format, quality information consistency among heterogeneous nodes is achieved and therefore, a common understanding of the QoS requirements by all the nodes involved in the cooperative execution exist. This is a major advantage offered by the Android architecture, and was also taken into consideration in the framework implementation. With this XML format, it is also possible to map and form hierarchies between quality dimensions and its attributes.

The application QoS parameters are parsed by the Package Parser, as presented in Listing 3. Here, the *parseQoS()*

method is called to load the parameters into memory by creating Java objects which will then be used by Android to handle the application's workflow. Each dimension is also parsed and added to a list where all the dimensions and its attributes are mapped. Note that all the application's properties are also loaded into memory during boot time, since the Android platform does not persist them.

Then, the parameters are sent to the *QoS Provider* component via a Java Native Interface (JNI) call to the Zygote. Listing 4 presents this step. One should note that there is a decision path based on the existence of QoS parameters. If the application presents QoS constraints, the method *Zygote.forkAndSpecializeQoS()* is called, otherwise, the standard Android method for forking the applications will be called, namely *Zygote.forkAndSpecialize()*. The listing is related to the *ZygoteConnection.java* file.

```
...
if(parsedArgs.qosArgs != null)
{
        qosArgs =
            parsedArgs.qosArgs.toArray(stringArray2d);

        pid =
            Zygote.forkAndSpecializeQoS(parsedArgs.uid,
            parsedArgs.gid,
        parsedArgs.gids, parsedArgs.debugFlags, rlimits,
            qosArgs);
}
else
{
        pid = Zygote.forkAndSpecialize(parsedArgs.uid,
            parsedArgs.gid,
          parsedArgs.gids, parsedArgs.debugFlags,
              rlimits);
}
...
```

Listing 4: JNI Call to Dalvik VM

This JNI call is handled by *dalvik_system_Zygote.c*, which belongs to the VM itself. The *Zygote.forkAndSpecializeQoS()* method presented in Listing 5 is responsible for handling the application QoS parameters and sending them via a socket communication to the *QoS Provider* which is listening for requests.

Each request is delivered to the *Local Provider* which based on the resource allocation levels sent by each *Resource Manager*, determines if there are enough resources to locally run the application. If this is the case, the VM will be forked and a new process is assigned to the application, following the typical flow of Android. On the other hand, the *Coalition Organiser* is invoked whenever the imposed QoS constraints cannot be locally satisfied.

```
static void
    Dalvik_dalvik_system_Zygote_forkAndSpecializeQoS(
        const u4* args,
    JValue* pResult)
{
pid_t pid;
ArrayObject *qosArgs = (ArrayObject *) args[5];

if (qosArgs == NULL)
{
        LOGI("Zygote Args are NULL\n");
}
else
{
  // socket code was removed

  u4 i, j;

  ArrayObject** tuples = (ArrayObject **)
      (qosArgs->contents);

  for (i = 0; i < qosArgs->length; i++)
  {
    u4 count = tuples[i]->length;
        StringObject** qos_tuple = (StringObject**)
            tuples[i]->contents;

        for (j = 0; j < count; j++)
        {
                StringObject* contents =
                    (StringObject*) qos_tuple[j];
                char* tmpString =
                    dvmCreateCstrFromString(contents);

                LOGI("Sendind QoS parms to
                    QoSProvider -> %s \n",
                    tmpString);

                // send params to QoS Provider
                n = write(sockfd, tmpString,
                    strlen(tmpString));

                memset(&buffer, 0, sizeof (buffer));
        }
}

// call standard function for fork
pid = forkAndSpecializeCommon(args);

RETURN_INT(pid);
}
```

Listing 5: Zygote before forking

The *Coalition Organiser* is responsible for partitioning the service in a set of blocks; requesting service proposals from neighbour nodes (detected by the *System Manager*) for each of those blocks; deciding which nodes will form the coalition based on their service proposals and user's QoS requirements; and disseminating blocks among the chosen nodes. For now, only services that can be divided in sets of independent blocks are supported, although algorithms for coordinating the execution of interdependent blocks were already developed [17]. A directed graph, describing the inputs and outputs of each block is dynamically formed when the service is partitioned.

Whenever coalition partners receive an application's block sent by the requesting node, a new VM instance is spawned by forking the main Dalvik VM instance, dealing with blocks as if they were locally started by an user.

Figure 4 presents the framework's workflow in the form of Unified Modelling Language (UML) activity diagram. The diagram presents a high level overview of the framework and its objective is to provide a clear view of the concepts presented in this section.

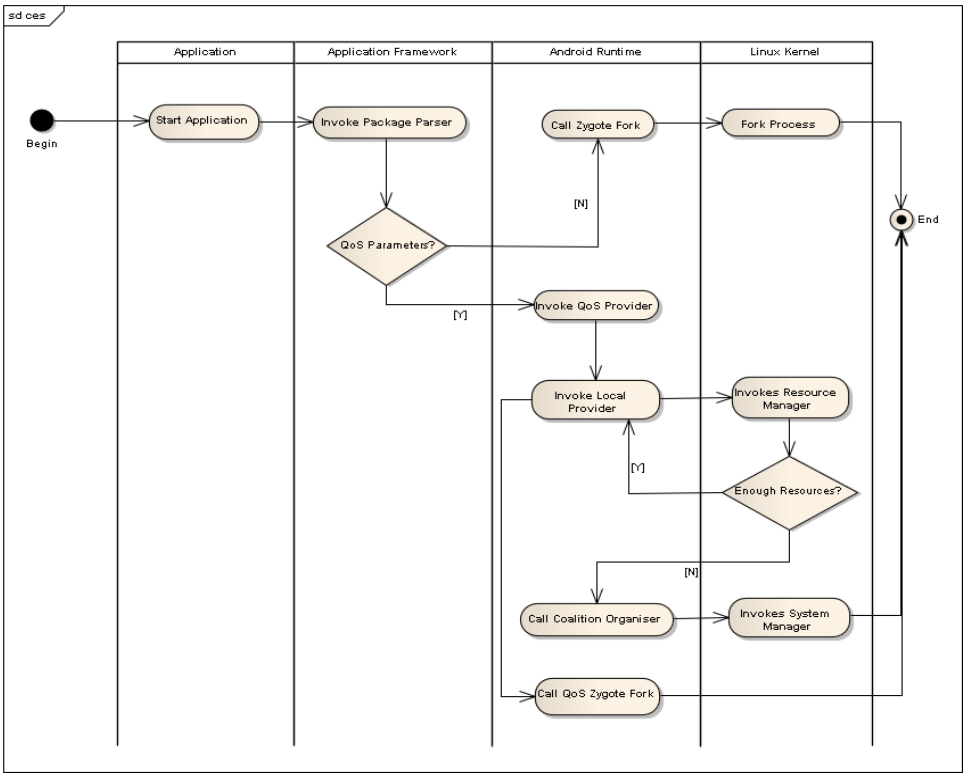

Figure 4: CooperatES Framework Workflow

5. EXTENDING ANDROID FOR REAL-TIME EMBEDDED SYSTEMS

In [13], we discuss the suitability of Android for open embedded real-time systems, analyse its architecture internals, point out its current limitations, and propose four possible directions to incorporate real-time behaviour into the Android platform:

- Inclusion of a real-time VM, besides Dalvik, along with the inclusion of a real-time operating system;
- Extension of Dalvik with real-time features based on the Real-Time Specification for Java [7], as well as the inclusion of a real-time operating system;
- Inclusion of a real-time operating system in order to allow only native applications to have the desired real-time behaviour;
- Inclusion of a real-time hypervisor that parallelises the execution of Android and real-time applications. Both run as guests over the hypervisor which is responsible for handling the scheduling and memory management operations.

The acquired experiences with the directions proposed in [13] allows us to conclude that the first direction is the one that causes less impact in the system as a whole. This direction allows the possibility of having Dalvik serving the needs of any Android application, while at the same time, the real-time VM can handle the specific requests made by any QoS-aware application that presents temporal constraints.

The inclusion of this second VM brings the desired real-time behaviour to the Android platform. Nevertheless, its inclusion also brings important challenges that should be considered. One may think of how the scheduling operations between both VMs are mapped into the operating system; how the memory management operations will be managed in order to take advantage from the system's resources and finally, how to handle thread synchronisation and asynchronous events in this dual VM environment.

Regarding scheduling, it must be assured by the operating system that all the real-time tasks have higher priority than the normal Android tasks. This can be achieved by having a mechanism that maps each of the real-time tasks to a higher priority operating system task. Then, the operating system scheduler is responsible for assuring that these tasks are dispatched earlier than the remaining tasks. Thus, at a lower end limit a simple mapping mechanism must exist to perform this operation.

As for the memory management, one possible solution to consider would be to have a memory management abstraction layer that handles all the memory operations requested by both VMs, i.e. allocation and deallocation through the use of a smart garbage collector. The main benefit from this layer would come from the fact that it would be possible to have a single heap where all the objects would be managed and thus, the system's resources to deal with the dual VM environment would be optimized. The disadvantage from this approach lies in the way that Dalvik performs. Each Android application runs on its own Linux process with its own VM and garbage collector instances. Also, there is a

part of the heap that is shared among all the processes. This *modus operandi* entails the need to, at least, integrate Dalvik with the abstraction layer and at the same time to modify its behaviour related to the per-process garbage collector instances.

Regarding thread synchronisation, as long as the real-time threads do not have the need to communicate with Dalvik threads, it is assured that this will not pose any kind of problems. However, if this communication is desired, a protection mechanism must be implemented in order to assure that a real-time thread will not block on a Dalvik thread or even that priority inversion does not happen. In terms of asynchronous events, a mapping mechanism must be sufficient to assure that the task that is waiting for the event will receive it in a bounded time interval. This mechanism must be implemented at the operating system level in order to forward the events to the correct VM. Both VMs just need to implement the handlers for the events.

6. CONCLUSION

The CooperatES framework is a QoS-aware framework addressing the increasing demands on resources and performance by allowing services to be executed by temporary coalitions of nodes. Users encode their own relative importance of the different QoS parameters for each service and the framework uses this information to determine the distributed resource allocation that maximises the satisfaction of those constraints.

Android mobile platform was made publicly available during the fall of 2008 and is gaining strength both in the mobile industry and in other industries, as a suitable platform for distributed embedded systems. The increasing interest from the industry arises from its open-source nature and its architectural model.

This paper proposes the needed extensions to the Android architecture to integrate the cooperative execution concept allied with real-time capabilities in a seamless way and as a natural ability of the architecture to satisfy the needs of the industry or the most demanding users. This implementation allows to analyse the limitations of the Android platform for real-time embedded applications, for which the paper proposes a suitable solution.

7. REFERENCES

[1] O. H. Alliance. Home page, June 2010.

[2] Android. Home page, Jan. 2010.

[3] Android-x86. Android-x86 project, Jan. 2010.

[4] G. Chen, B.-T. Kang, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and R. Chandramouli. Studying energy trade offs in offloading computation/compilation in java-enabled mobile devices. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):795–809, 2004.

[5] A. Corsaro. jrate home page, Mar. 2010.

[6] V. S. W. Eide, F. Eliassen, O.-C. Granmo, and O. Lysne. Supporting timeliness and accuracy in distributed real-time content-based video analysis. In *Proceedings of the 11th ACM international conference on Multimedia*, pages 21–32. ACM Press, 2003.

[7] R.-T. S. for Java. Rtsj 1.0.2, Jan. 2010.

[8] X. Gu, A. Messer, I. Greenberg, D. Milojicic, and K. Nahrstedt. Adaptive offloading for pervasive computing. *IEEE Pervasive Computing Magazine*, 3(3):66–73, 2004.

[9] U. Kermer, J. Hicks, and J. Rehg. A compilation framework for power and energy management on mobile computers. In *14th International Workshop on Parallel Computing*, pages 115–131, 2001.

[10] Z. Li, C. Wang, and R. Xu. Computation offloading to save energy on handheld devices: a partition scheme. In *Proceedings of the 2001 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 238–246. ACM Press, 2001.

[11] Z. Li, C. Wang, and R. Xu. Task allocation for distributed multimedia processing on wirelessly networked handheld devices. In *Proceedings of the 16th International Symposium on Parallel and Distributed Processing*, page 79. IEE Computer Society, 2002.

[12] G. Macario, M. Torchiano, and M. Violante. An in-vehicle infotainment software architecture based on google android. pages 257 –260, July 2009.

[13] C. Maia, L. Nogueira, and L. M. Pinho. Evaluating android os for embedded real-time systems. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, Brussels, Belgium, July 2010.

[14] L. Marcenaro, F. Oberti, G. L. Foresti, and C. S. Regazzoni. Distributed architectures and logical-task decomposition in multimedia surveillance systems. *Proceedings of the IEEE*, 89(10):1419–1440, October 2001.

[15] L. Nogueira and L. M. Pinho. Capacity sharing and stealing in dynamic server-based real-time systems. In *Proceedings of the 21th IEEE International Parallel and Distributed Processing Symposium*, page 153, Long Beach,CA,USA, March 2007.

[16] L. Nogueira and L. M. Pinho. Time-bounded distributed qos-aware service configuration in heterogeneous cooperative environments. *Journal of Parallel and Distributed Computing*, 69(6):491–507, June 2009.

[17] L. Nogueira, L. M. Pinho, and J. Coelho. Coordinated runtime adaptations in cooperative open real-time systems. In *Proceedings of the 7th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, Vancouver, Canada, August 2009.

[18] M. Othman and S. Hailes. Power conservation strategy for mobile computers using load sharing. *SIGMOBILE Mobile Computing Communications Review*, 2(1):44–51, 1998.

[19] RTMACH. Linux/rk, Mar. 2010.

[20] A. Rudenko, P. Reiher, G. J. Popek, and G. H. Kuenning. Saving portable computer battery power through remote process execution. *Mobile Computing and Communications Review*, 2(1):19–26, 1998.

[21] S. Schmidt, T. Legler, D. Schaller, and W. Lehner. Real-time scheduling for data stream management systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 167–176, 2005.

[22] M. Stanley. The mobile internet report, Jan. 2010.

[23] M. Stonebraker, U. Cetintemel, and S. Zdonik. The 8

requirements of real-time stream processing. *SIGMOD Record*, 34(4):42–47, 2005.

[24] C. Wang and Z. Li. Parametric analysis for adaptive computation offloading. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 119–130. ACM Press, 2004.