Technical Report

Handling Mobility on a QoS-Aware Service-based Framework for Mobile Systems

Joel Gonçalves
Luis Lino Ferreira
Luis Miguel Pinho
Guilherme Silva

HURRAY-TR-101202
Version:
Date: 12-01-2010

Handling Mobility on a QoS-Aware Service-based Framework for Mobile Systems

Joel Gonçalves, Luis Lino Ferreira, Luis Miguel Pinho, Guilherme Silva

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:

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

Abstract

Mobile applications are becoming increasingly more complex and making heavier demands on local system resources. Moreover, mobile systems are nowadays more open, allowing users to add more and more applications, including third-party developed ones. In this perspective, it is increasingly expected that users will want to execute in their devices applications which supersede currently available resources. It is therefore important to provide frameworks which allow applications to benefit from resources available on other nodes, capable of migrating some or all of its services to other nodes, depending on the user needs. These requirements are even more stringent when users want to execute Quality of Service (QoS) aware applications, such as voice or video. The required resources to guarantee the QoS levels demanded by an application can vary with time, and consequently, applications should be able to reconfigure themselves. This paper proposes a QoS-aware service-based framework able to support distributed, migration-capable, QoS-enabled applications on top of the Android Operating system.

2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing

Handling Mobility on a QoS-Aware Service-based Framework for Mobile Systems

Joel Gonçalves, Luis Lino Ferreira, Luis Miguel Pinho, Guilherme Silva
CISTER Research Center
Polytechnic Institute of Porto (ISEP/IPP)
Porto, Portugal
{vjmg, llf, lmp, grss}@isep.ipp.pt

***Abstract*—Mobile applications are becoming increasingly more complex and making heavier demands on local system resources. Moreover, mobile systems are nowadays more open, allowing users to add more and more applications, including third-party developed ones. In this perspective, it is increasingly expected that users will want to execute in their devices applications which supersede currently available resources. It is therefore important to provide frameworks which allow applications to benefit from resources available on other nodes, capable of migrating some or all of its services to other nodes, depending on the user needs. These requirements are even more stringent when users want to execute Quality of Service (QoS) aware applications, such as voice or video. The required resources to guarantee the QoS levels demanded by an application can vary with time, and consequently, applications should be able to reconfigure themselves. This paper proposes a QoS-aware service-based framework able to support distributed, migration-capable, QoS-enabled applications on top of the Android Operating system.**

Quality of Service, mobile systems, Android OS

I. INTRODUCTION

Mobile applications are increasingly more ubiquitous and more dynamic. Furthermore, mobile systems are now open to third-party developed applications, being expectable that users put more and more pressure on the locally available resources. Even considering the substantial increase in devices' capabilities, it is not expectable that they will be able to simultaneously support all applications the users may want to execute over time. This lack of support can be dynamic (lack of execution resources such as CPU or memory) or even static (lack of space for application code). The solution to this is to allow applications to scavenge resources available in other nodes by migrating some or all of its services (or of other applications) into other nodes (or from other nodes). In mobile devices it is also of paramount importance to provide applications with consistent performance parameters, i.e. application that offer adequate QoS levels to the users.

Therefore, there is a growing need to develop frameworks and applications which rely on global resources and that are able to reconfigure considering system-wide distribution of application and resources, at the same time guaranteeing the QoS levels required by the applications.

For that to succeed, applications must be able to work in a distributed manner and reconfigure themselves autonomously in response to system resource or configuration changes. Reconfiguration may occur due to resource scarcity (e.g., a node is going to shutdown), due to new nodes becoming available, or due to user-induced changes (e.g. the user may want to change location and/or device where the application is executing). Therefore, applications must support: i) the capability to connect seamlessly to remote services; ii) the capability to move some of its services to remote nodes. The first characteristic can be supported by component [1] or service-based [2] frameworks. The second characteristic must be supported by extending those frameworks with code mobility capabilities, allowing to both install and run(parts of) applications in remote nodes [3].

A mobility framework must also enable the run-time relocation of services in response to system changes, either structural or quality-driven. QoS parameters, like timeliness and bandwidth requirements, must be considered both in the regular operation of the applications, and in reconfiguration transients. Code mobility can have a large impact on the system's performance [4]. Such operations may, momentarily, require large amounts of resources such as network bandwidth (to transfer code and state), and CPU time (needed to reconfigure the system). Consequently, transient QoS impairments, such as delays, on applications must be accounted for when trying to guarantee its QoS requirements.

From the point of view of application developers, the introduction of new functionalities must require a small amount of effort. The integration of new functionalities with the currently existent concepts and architectures should be (quasi-)transparently with the existent middleware and operating system. The goal must be to provide a development framework where configuration issues are hidden from applications (except in management code). Therefore, development frameworks must use the same concepts of interaction both for local and remote services. This is the concept behind the approach proposed in this paper. It provides reconfiguration and migration capabilities, extending the main application abstractions used in the Android operating system – Activities, Services and Intents – allowing for transparent interaction in distributed configurations.

The Android operating system is used both due to its open source nature and potential market, but also due to its innovative architecture. Although its use to support real-time applications is still debatable [5] it nevertheless provides a suitable architecture for quality of service-aware applications in ubiquitous, embedded systems [6].

978-0-7695-4322-2/10 $26.00 © 2010 IEEE
DOI 10.1109/EUC.2010.24

Applications like multimedia and gaming are usually resource hungry, either in terms of CPU utilization but also in relation to memory. These applications usually adapt their QoS levels to the current device, but obviously users would always like to have the highest QoS level possible. To that purpose parts of the application can be offloaded to other nodes in order to achieve a higher QoS level.

The participating nodes (coalition) might be physically near and in the same IP network. But with the advent of the Internet of Things it is expected that not only smartphones can cooperate but also fixed nodes, like game consoles, televisions, PDAs and notebooks. Another possibility is that participating nodes might also connect to cloud services. In fact, the main requirement is that these nodes should achieve the QoS level required by the application, which due to network QoS management limitations, might be more achievable on a LAN environment.

Coalition establishment depends on the availability of nodes and on the algorithms used to choose the participating nodes. We assume that the availability of the envisaged services depends mostly on the kind of enrolment, which can be mandatory (e.g. in the case of corporate users and home devices) or voluntary. If the enrolment is voluntary then there must be some kind of incentives for the users, by paying for the processing time, giving them more credit to use the system or simply by being the person which has contributed most for the system [7].

Security issues, although being important, are outside the scope of this paper, therefore, we will rather focus the remainder of the paper on architecture issues to implement our vision.

The paper is organized as follows. Section II provides the considered system model, relating to previous relevant work. Afterwards, Section III describes the Android operating system, a relevant platform for mobile applications. Then Section IV provides a description of the architecture and main functionalities of the proposed mobility framework. Section V presents and discusses some results related to the use of the framework. Finally, in Section V we draw some conclusions and discuss future work..

II. SYSTEM OVERVIEW

A. Generic System Model

We assume a system constituted by several cooperating services with Quality of Service (QoS) requirements. In such a system, an application is supported by connections between services, either local or remote.

Our model assumes that the system is formed by a set of N nodes $\Pi = \{\pi_1, \pi_2 \ldots \pi_N\}$ and a set of M services $S = \{s_1, s_2, \ldots, s_M\}$, each of these services can be connect to other services, therefore links ($l_{x,y}$) characterize connections between services S_x and S_y. An application (A_i) is represented by its services (S_i), which are a subset of S.

Figure 1 depicts an example system where an application A_1 is constituted by services $\{s_1, s_2, s_3, s_4, s_5, s_6\}$. Service s_1 uses local services s_2 and s_6. Service s_2 requires the use of two remote services, s_3 and s_4, while service s_6 only issues service requests to s_5.

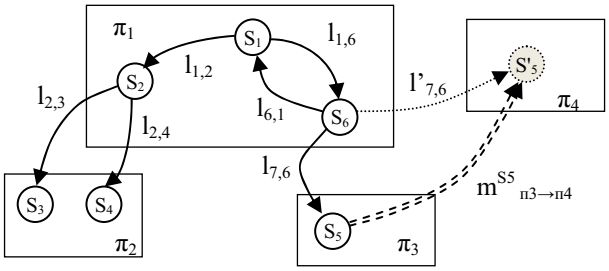

Figure 1. System model

A generic QoS model is used, where each service (S_i) or link ($l_{x,y}$) QoS requirements are defined by a tuple of o elements $R_c = \{r_{c1}, r_{c,2}, \ldots, r_{c,o}\}$, being the actual QoS model outside the scope of the paper. As a simple example these requirements can be the deadline, the execution time and the period between consecutive evocations of a service i: (R_i=$\{d_i, c_i, T_i\}$), but other more complex models may be used, like Q-RAM [8] where the QoS requirements are defined in terms of minimum and maximum values and the resource allocation tries to maximize a reward function. Another approach, the CooperatES framework [9], defines its QoS requirements as a set of possible values and it also tries to maximize a system reward function, its main strength is that this it is able to operate in a truly distributed and decentralized way.

During run-time some of the system services might migrate to a different node. This mobility requires the transfer of code and state and the rebinding of connections between services. As a consequence, there will be an interval of time during which some of the application services are in a reconfiguration mode. In a QoS-enabled system such mode might also imply the momentary reservation of extra resources (e.g. CPU or network bandwidth). Although code can be sent independently in parallel while the service is still in execution, state transfer requires the controlled stopping (also referred as passivation) of the service previously to its migration. This can be done according to the rules defined in [10,11]. After or during state transfer, connections between services have to be redirected.

The operation $m^{S5}{}_{\pi 3 \rightarrow \pi 4}$ represents the mobility operation for service S_5, between node π_3 and node π_4. In such case π_3 is denoted as the source node, in relation to the mobility mechanism and node π_4 is denoted as the destination node. Link $l'_{7,6}$ represents the connections which have to be established after the service mobility is complete, consequently, connection $l_{7,6}$ will have to be seamlessly deleted prior to the entry into operation of $l'_{7,6}$.

During this reconfiguration the QoS provided to the application may be momentarily reduced. Therefore, each service that has migration capabilities and QoS requirements must also specify the minimum QoS level, or the maximum inaccessibility time (t_{ina}) which it can support. This timing combined with the current system scheduling mechanism enables the calculation of the QoS parameters to be used for entities involved in code mobility procedures, which are described in Section IV.

B. Related Work

The area of service/component architectures and code mobility is highly studied and several software frameworks have been proposed in the last decade.

OSGi [3] is one of the most disseminated platforms for component integration in Java. It defines the means for the distributed deployment of applications through several nodes (Life Cycle management) and its reconfiguration in run-time. It also supports discovery and registry of services. This framework has been extended with component mobility services, particularly adapted for context aware applications [12]. From the original framework, they changed the key value format of the bundles manifest file to an ontology format. Services life cycle is augmented by new states related to the service migration and state dissemination. Concerning the services discovery, the extensions rely on the CoGITO framework. The OSGI framework has also been ported to the Android operating system, but no code mobility capabilities had been implemented.

Other works had focused on providing a mobility framework which not only provides a specific architecture for component interaction but also incorporates QoS parameters (which model the system) into the design [13], and uses that model to determine the most useful system configuration in reconfiguration events.

The European project Runes [14] created a middleware which can be used on the development of reconfigurable software components in embedded systems. The objective of this project was to create a homogeneous platform for heterogeneous environments, connecting sensor networks nodes, mobile devices and mobile computers. It implements weak code mobility without an explicit support for state transfer, the main services are related to the installation of new components and for the replacement of old ones.

Other projects, like FRESCOR [16] and ACTORS [17] had also provide component-based frameworks specifically for embedded systems, focusing its particularly on real-time issues.

Our framework integrates support for the development of distributed QoS-enabled applications for mobile systems. Such applications are composed of services capable of migrating between nodes. Contrarily to previous approaches, it integrates seamlessly with the Android operating system, by extending the functionalities provided by the Intent, Activity and Service abstractions used to interact between application components. Additionally, this framework also provides the necessary means to handle all kinds of QoS requirements during service mobility operations.

We also assume the use of a higher level framework like Q-RAM [8] or CooperatES [9], which are able to determine the initial allocation of the system resources between the applications. Additionally, during run-time such an framework should also be able to recalculate a new resource allocation in response to environment changes (like the departure of a node or energy scarcity in a node).

Consequently, to our best knowledge the framework being proposed in this paper is the first one that addresses both the QoS issues for mobile systems and also the mobility of services with QoS needs.

III. ANDROID PLATFORM

A. Android stack

The Android operating system is structured in 4 layers and several modules (Figure 2). It can be divided in a static unchangeable part and a changeable part. The static part is the System Image which is based on the Linux kernel, the Libraries together with the Android Runtime and the Application Framework. The changeable part is the User Applications layer, which can be freely installed and uninstalled from a device. The underlying layers and modules can only be changed by creating and installing a new system image, thus it cannot be changed by a simple application install.

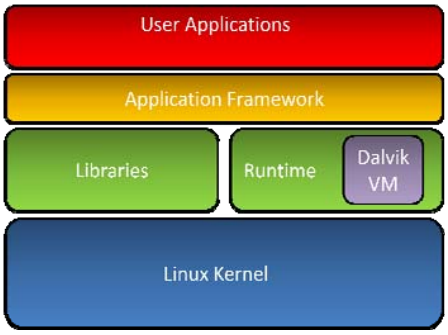

Figure 2. Android architecture (known as Android Stack)

The Linux kernel is responsible for memory and process management. The main advantage of Linux is that it has a wide community support and allows to easily add device drivers for specific hardware components. The Libraries layer provides native libraries which offer a set of functionalities to be used for application development. Since these libraries are written in native code (C code), the performance is higher than for programs running over the Dalvik Java virtual machine.

The Android Runtime is composed by several Java libraries and the Dalvik Virtual Machine (VM), capable of running applications in Dalvik bytecode format, which differs from the Java Byte code by being more adapted to systems with low memory and processor speed.

The Application Framework layer is organized to support application development. This layer is completely written in Java so that applications, also in Java, can easily interact with these modules. The abstractions provided to the applications aim to promote code reusability. For instance, the Telephony Manager provides telephone information and contact information. The Location Manager also provides location information. All this information can be made available to other applications through specific interfaces.

B. Application Development

Android uses an abstraction model to organize applications. Typically, applications are composed by several *Activities* which represent a single task that a user can

perform. The *Activities* are used for user-application interaction, so every time an *Activity* is launched a new screen is displayed to the user. The *Service* abstraction is used to perform tasks in background (without a User Interface). Communication with a *Service* is made through an interface that the service exposes, but typically an *Activity* is only able to return a value.

The Android application model is based on a service-oriented architecture, where *Intents* are abstract descriptions of an operation to be performed. It can be used to start an *Activity* or a *Service*.

All the previously mentioned abstractions are compressed and bundled in an Android Package (APK) file. Such files are very similar to the Java Archive files (*JAR*), but APK files also contain the application code (like JAR files), resources (images, sounds, etc) and a manifest file (describing the application requirements, interfaces and permissions).

Besides the Dalvik bytecode format used in a APK, another important difference is that the classes' code is not separated into several files but it is rather aggregated in a single file with extension .dex. The manifest file interface is used by the system to resolve an *Intent*; if the content of an *Intent* matches an interface entry, then the operating system launches the class associated with that entry.

The APK files are executed using the *Intent* abstraction, which represents the intention of executing a specific interface of a APK. Applications send *Intents* in order to address an *Activity* or *Service* from a specific APK. Android OS is then able to search, on the installed APKs, for the desired interface, and execute it. This message mechanism provides a simple way to use *Activities* and *Services* from the current APK or from another one installed on the device, thus promoting code reuse.

IV. RT-MobFr Framework

Android applications coarse structures are the APKs installed in the local device. During runtime, these modules can be assembled to create dynamic applications, but these functionalities are only available in the local node. In this paper, we extend this concept to a fully distributed and dynamic environment, where applications use the Activities and Services from the APKs whether they are installed locally or remote. Further, we allow APKs to migrate to other nodes, by user demand or due to system's reconfiguration. We also allow to dynamically upgrade the system devices with different service versions in run-time. Finally, the framework is also able to support applications with QoS requirements, both during its run-time operation and particularly during the migration of services.

The APKs are then executed as independent Linux processes, with distinct IDs, permissions, and, importantly, also with different QoS parameters. Therefore, our approach consists on transferring and using them as coarse grained mobile code components, the APK packages. An alternative approach, used in some Java code mobility frameworks [15], relies on the transfer of classes between nodes and on the execution of threads offering the functionalities provided by those classes. Although, this alternative requires the transfer of less data, the proposed approach has the advantage of isolating the transferred application: i) allows the setting of different QoS parameters for each service; ii) also running it in different security and resource access context. Both kinds of parameters can be defined on the APK manifest file.

Figure 3 provides an example which illustrates the concepts being proposed: the initial scenario is presented in the left part, where DEV1 is executing an application. The application is composed by an GUI activity (Activity Menu) and several services. Application code resides in three distinct APKs, spread in DEV1 (APK A) and DEV2 (APK B and APK C). This division is hidden from the application code, which can issue Intents for a service in APK B or C. These intents are mapped by the framework into the remote services in DEV2.

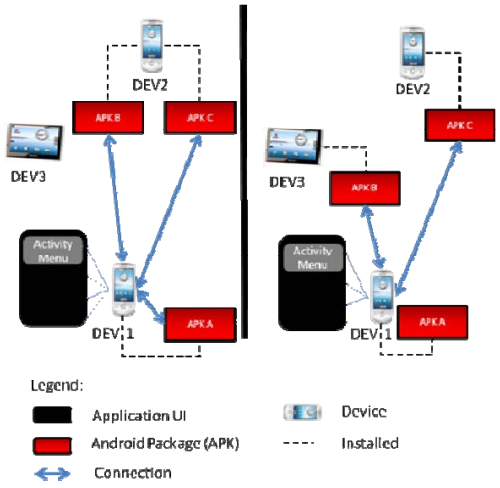

Figure 3. Example scenario

The right side of Figure 3 provides a new system configuration, which includes a new device: DEV3. After being detected by the other nodes and assuming that this device has more available resources, then an improved configuration of the system can be achieved if APK B migrates to DEV3. This new configuration might, for example, allow achieving a higher QoS level for the application or allow reducing the battery consumption in DEV2. Algorithms and frameworks as the ones proposed in CooperatES [9], FRESCOR [16] or ACTORS [17] can be used to find the new configuration for the system services, maximizing the rewards for the overall system. As a result of the evaluation, DEV3 now offers the services of APK B in order to increase the QoS of the application.

The Mobile code mechanisms of the RT-MobFr can support such approach, making possible to transfer the code and state of service B from DEV2 to DEV3, install the corresponding APK file, rebind the connections between DEV1 and the service in APK B, and continue its operation.

If the application has real-time QoS requirements then this complex set of operations must be done within a bounded time, limiting or even eliminating any QoS disruption, which can be noted by the users of the application. As a concrete example of such a scenario,

assume that the application is a 3D game which can offload some of its computations to other devices (e.g. the game intelligence, rendering or physical engines) in order to raise its quality. The player of the game should not notice any disruption on the game flow due to the system reconfiguration including the mobility of services. Usually for such a kind of interactive application a user does not notice any temporal misbehaviour if the interruption of service is less than 200 ms.

A. *Architecture Overview*

In order to implement this approach we propose the architecture depicted in Figure 4. The architecture assumes the existence of a QoS Manager on the Linux kernel, the addition of Application Layer features to support the core functionalities of a mobile code framework and the use of supporting libraries (the RT-MobFr libraries) which allow programmers to use the full set of capabilities offered by the framework.

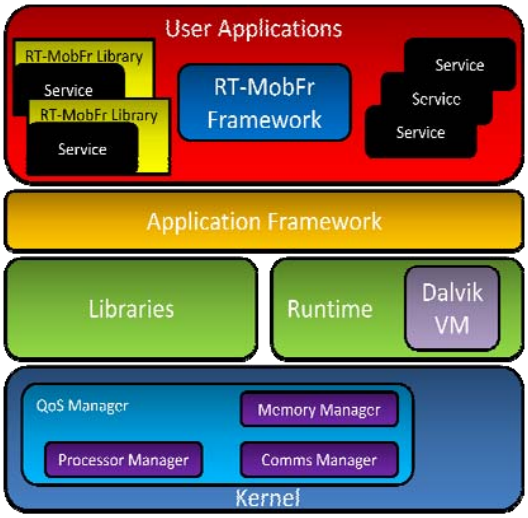

Figure 4. Implementation Overview

At the operating system level, we assume the existence of a QoS Manager module, which interacts with several distributed resource management modules, for the acceptance of new, and adaptation of existent, QoS-aware services. This can be based in implementations currently being provided (e.g. [6] or [18]), and use available solutions particularly adapted to wireless networks, like IEEE 802.11e or IEEE 802.15.1. Our framework can also use the Resource Reservation Protocol (RSVP) [19], which already provides a set of well defined QoS-related functionalities in WANs. This module can also encapsulate the functionalities of high level QoS control frameworks, like the one defined in [9]. Although, for convenience purposes we had represented this module at the kernel level it can also have some other components at the User Applications level, to control the RT-MobFr framework sending to it orders including QoS changes and orders to migrate services to other nodes.

The *Framework Core* functionalities (Figure 5) constitute a separate module implemented as 4 Android services, which takes care of service migration, to and from a node, interacting with the QoS Manager.

Any Android application can use the services of the framework in order to support some simple mobility operations, like transferring, installing and running an Android APK, without dependencies from other services, in another node. More advanced services, which require the rebinding of connections between components, are only supported if applications use the RT-MobFr library.

The RT-MobFr library offers a set of helper classes that extend the core functionalities of Android with real-time and distributed capabilities. The library allows to transparently extend the *Activity* and *Service* abstractions for distributed systems. Consequently, application code is only required to send an *Intent*, after which the framework is responsible for determining if it refers to a local or remote component. The library also implements the services required for communication establishment and automatic rebinding of component connections when a component migrates.

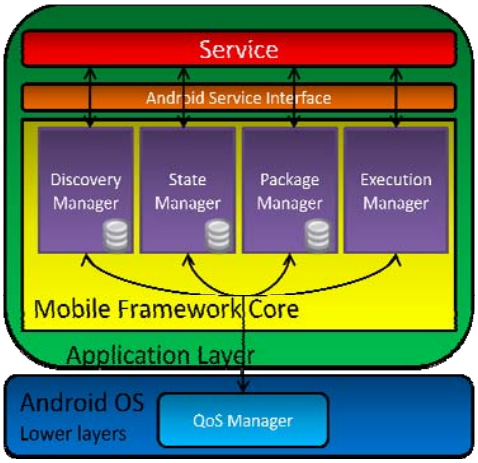

Figure 5. Framework core modules

The proposed framework is implemented at the Application layer level since this layer is more appropriate to deploy new software components, as it does not require any changes to the Android Operating System image. The installation of the framework is equivalent to an ordinary application installation, and since it is implemented as a standard Android Service its functionalities are accessible to all applications installed on the device.

B. *Framework managers*

The core services provided by the framework are: *Discovery Manager*, *Package Manager*, *State Manager* and *Execution Manager*.

1) *Discovery Manager*

The *Discovery Manager* module is designed to discover neighbour devices on a local network and advertise the host device capabilities. To that purpose, every node in the network periodically broadcasts information regarding its status and installed services, such as the APKs installed, their associated Intents interfaces and QoS capabilities.

These messages allow the QoS Manager to determine the state of the system and consequently, when required, determine a new system wide configuration. In this work we assume that this entity is responsible to send orders to the RT-MobFr framework which triggers the reconfiguration of the QoS parameters of applications and also the mobility of its services between nodes.

2) Package Manager

The *Package Manager* is used to install, uninstall and transfer the code of APKs between Android devices. Applications can start executing when the node receives an Intent request from a remote *Execution Manager* (see below). During the installation process the code is stored in a specific directory, which, for security reasons, can only be accessed by the *Discovery Manager* to detect the new APKs. It is very common to use these kinds of frameworks to support the evolution of services; therefore, this module is also able to manage multiple versions of the APKs.

The *Package Manager* is also responsible for the interaction with the *QoS Manager* in order to request specific QoS levels for the service being handled. It is the responsibility of the *QoS Manager* to accept or reject service installations if the QoS required level cannot be guaranteed.

3) State Manager

The *State Manager* handles the transfer of state for statefull services. Services are responsible for explicitly defining their state as different state items. By separating the service state into several pieces it is possible to apply different techniques for the transfer of the service complete state. We define a State Item (SI_i) as the tuple:

$$SI_i = \{ID, \psi, version\}$$

where *ID* is a string which univocally indentifies this State Item, ψ is a byte array which contains the state item and *version* is an integer that corresponds to the version number of this state item. By using this constructor it is possible to separate the state of a service into different variables, different objects or even combinations of several objects and variables. It is up to the service using this functionality to define how to use state items.

The *State Manager* is responsible for transferring either the full state or specific state items. This allows implementing strategies, as the ones proposed in [12] (which use the publish/subscribe model for the migration of context-aware applications) and [20] (which propagates only the operations performed over the data instead of the data itself), or any other strategy more appealing to the application being implemented. The flexibility on the implementation of the state migration policies can be of paramount importance for the use of code mobility techniques in real-time systems since it can allow the reduction on the unavailability time of a service.

In [4] we propose the division of state transfer in two phases, where during the first phase we assume that the service will not be stopped, e.g. it can be used to transfer the initialization parameters of a service. Only, on the second phase the service will be stopped (passivated) and this phase is used to transfer items which reflect the internal state of the service, e.g. variable values. This simple strategy reduces the unavailability time of the service.

It is also important to note that the framework can (optionally) store different version of a state item, thus "browsing" through different state items of a service can be useful for error recovery or for debugging purposes.

4) Execution Manager

The *Execution Manager* allows launching services on a host device or on a remote node. Android *Intents* are exchanged between devices and used locally to start up a service, which can be a standard Android *Activity* or *Service. Intents* that address *Activities* or *Services* in remote nodes are parsed to extract their parameters and sent to the remote *Execution Manager*, which then reconstructs the intent and launches it locally (on the remote node).

This module can also be used to start up *Activities* and *Services* that were developed without considering mobility issues – legacy APK. Since Android abstractions (*Intents*, *Activities* and *Services*) operate locally, the framework assumes the existence of a proxy mechanism which allows the interaction between local and remote legacy services. Services which are based on the framework use the new versions of *Service* and *Activity* classes; therefore, when calling for a remote service they connect directly using the functionalities provided by the RT-MobFr Library.

C. RT-MobFr Library

The RT-MobFr Framework core functionalities are implemented as normal Android services, but such interface only offers a basic set of functionalities, mainly for service mobility and discovery. Full transparency is only achieved by using the extended functionalities provided in the RT-MobFr Library. Based on this library, developers can easily create services with mobile and QoS requirements. Additionally, it is also possible to extend the functionalities in order to adapt to the application being developed.

The `MobileServiceAbstraction` is the core class on the development of a service. It is mainly an aggregation of five classes (Fig. 6), which are capable of accessing the core functionalities of the framework.

Our framework extends the `MobileActivity` and `MobileService` classes, which should be used, instead of the standard Activity and Service classes since the new versions provide the necessary means to abstract the developer from the inherent distribution of services. Additionally, this technique also significantly reduces the effort involved on the porting of standard applications to the new paradigm.

The `IMobileService` is a common interface, implemented by the `MobileActivity` and `MobileService` classes, which enables the handling of both types of abstractions in a common way by the other classes of the framework. As example the `Synchronize ServiceAbstraction` can use that interface to stop a service or activity.

The `SynchronizeServiceAbstraction` class has been designed to handle any kind of events between services, either local or remote. It should also be used to

control the mobility of a service, when commanded by an external entity (e.g. by the *QoS Manager*) or to signal changes on the QoS level of a service. Other kinds of events can also trigger actions by this class, like the handing of stopping and resuming commands for a service. As an example if the QoS manager determines that it can increase the QoS level of an application, then sends an event to this class, which, by its turn, adapts the application to the new QoS availability.

The `SynchronizeServiceAbstraction` class also takes care of handling the installation of local or remote services and of its execution on a destination node, by using the classes `PackageManagerConnection` and `Execution ManagerConnection`, respectively.

The `ConnectionServiceAbstration` allows communications between services, either in the same node or on a remote node. Local communications are supported by inter-process communication mechanisms. Communications between services in different nodes, are also supported by this class. Specifically, it insures the closing of old connections and the establishment of new ones, in a controlled fashion (rebinding) when a service migrates.

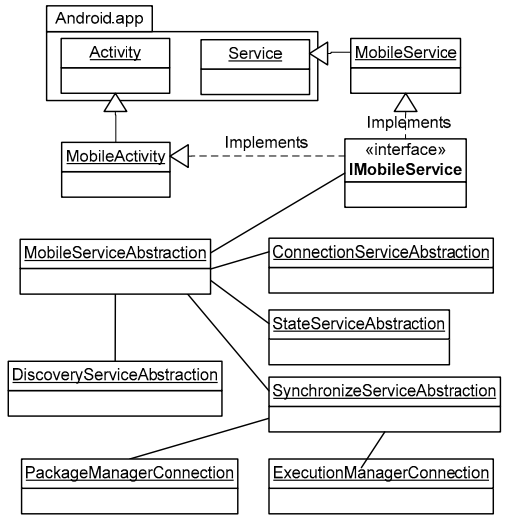

Figure 6. Mobile Library abstraction model

The library also offers classes to perform queries based on the information retrieved by the framework core *Discovery Manager* module, to perform state saving and transfer operations, implemented by the `Discovery ServiceAbstraction` and by the `StateService Abstraction`.

The complete framework is available online (see [21]), it also offers some standard implementation of the classes for demo purposes. The library still lacks integration with more advanced QoS management services, which are being developed [5].

V. RESULTS AND DISCUSSION

To test the framework and evaluate its timing performance we performed several experiments which aimed at determining the delays involved on the service mobility phases. Our tests were run on a conventional Android OS without any specific QoS manager. To the purpose of these tests we had only evaluated how changes on process priority influence the timing performance on the mobility of a service belonging to an application.

Our test setup is constituted by two HTC Magic and an Android emulator, all running Android OS version 2.0. In this case the emulator simulates a fixed Android device, like a television or a netbook. The scenario chosen is equal to the one depicted in Fig. 3 in which a service, represented by APK B, migrates between DEV2 and DEV3. The main focus has been on the evaluation of the time required to migrate APK B between the two nodes.

Our tests started by evaluating the time required to gather information by the Discovery Manager about neighbouring devices in the same broadcast domain. This phase occurs in parallel to the application execution prior to the order to move APK B, consequently, it does not influence the application and additionally the Discovery Manager runs on a low priority level (nice value of 10). Note that a nice value of -20 represents the highest priority and 19 the lowest priority for a thread. We obtained an average value of 95 ms, but the results the varied between 6 ms and 221 ms. It is important to note that just the execution of the garbage collector, during this phase can require 200 ms or more.

After that, a decision must be taken in relation to the new configuration of the system, in this test we are ignoring that phase, but the algorithm proposed in [9] is capable of reaching a solution in a bounded time.

Our mobility logic is implemented on the `SynchronizeServiceAbstraction` class which triggers the mobility of APK B. It all starts by shipping the code (an APK with a size of 116 kB) and state followed by its installation on the destination device. It is important to note that during the execution of these operations the service can continue active. Additionally, if the APK already exists on the destination device it is not necessary to install it (that is the case used in these tests).

The next three phases require the stopping of the services provided by APK B. These phases involve the transfer of remaining state items, the sending of an Intent to start executing the services in APK B and the rebinding of connections (i.e. the connection between APK A and APK B must be redirected according to the address of DEV 3).

Our experiments varied the priority levels of the framework and of the application in order to determine its influence on the overall delays of the mobility operation. Fig. 7 shows a graphic in which the framework and application priorities are varied between -15 and 15. It shows that there is a high level of variability on the mobility timings but the priority increase allows reducing its average values and most importantly its variability.

It is important to note that Fig. 7 only focus represents the overall delays from the start of the mobility procedure until the connections between APK B and APK A are reconfigured. In fact the only during a fraction of that time the service is totally inaccessible. In average the

inaccessibility time is equal to 224 ms (about 50% of the total mobility time).

VI. CONCLUSIONS

In this paper we proposed a framework for the development of distributed QoS-aware applications with self-reconfiguration capabilities. The framework is particularly targeted for the Android Operating System and its implementation extends the main abstractions used in Android – *Activities*, *Services* and *Intents*, allowing for transparent interactions of application code in both local and distributed settings. QoS requirements of both applications and reconfiguration services can be supported by the underlying operating system QoS Manager module.

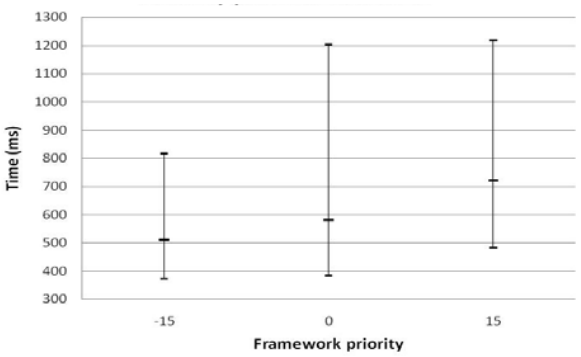

Figure 7. Delay of the total code mobility mechanism

This framework will be used to enhance the real-time capabilities of the Android OS and particularly to developed adequate strategies for multiple parameter QoS applications (considering both tasks and communication streams). We also plan to investigate further on how to better manage dynamic adaptations required on the system QoS levels during reconfiguration phases. An implementation of the framework proposed in this paper is available at [21].

Acknowledgment. This work is partially funded by the Portuguese Science and Technology Foundation (Fundação para a Ciência e a Tecnologia - FCT) and project SENODs (CMU-PT/SAI/0045/2009).

References

[1] Zachariadis, S., Mascolo, C., Emmerich, W, "The SATIN Component System-A Metamodel for Engineering Adaptable Mobile Systems," IEEE Transactions on Software Engineering, Nov. 2006, pp. 910-927.

[2] Erl, T., "Service-Oriented Architecture: Concepts, Technology, and Design", Prentice Hall PTR, 2005.

[3] G. O. Young, "OSGi Service Platform Release 4 Version 4.0", OSGi Fondation, 2010.

[4] Ferreira, L., "On the use of Code Mobility Mechanisms in Real-time Systems", HURRAY-TR-100508, http://www.hurray.isep.ipp.pt/, May, 2010.

[5] Maia, C., Nogueira, L., Pinho, L., "Evaluating Android OS for Embedded Real-Time Systems", Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2010), Brussels, Belgium, 2010, pp. 63-70.

[6] Maia, C., Noqueira, L, Pinho, L., "Cooperative embedded application in Android Environments", Submitted for publication on the 8th International Workshop on Java Technologies for Real-time and Embedded Systems - JTRES 2010 .

[7] Elespuru, P. R., Shakya, S., Mishra, S., "MapReduce System over Heterogeneous Mobile Devices", Proceedings of the 7th IFIP WG 10.2 international Workshop on Software Technologies For Embedded and Ubiquitous Systems (Newport Beach, CA, November, 2009), Lecture Notes In Computer Science, vol. 5860. Springer-Verlag, Berlin, Heidelberg, 168-179.

[8] Rajkumar, R., Lee, C., Lehoczky, J., Siewiorek, D., "A resource allocation model for QoS management", Proceedings of the 18th IEEE Real-Time Systems Symposium, San Francisco, USA, 1997, pp. 298-307.

[9] Nogueira, L., Pinho, L., "Time-bounded Distributed QoS-Aware Service Configuration in Heterogeneous Cooperative Environments", Journal of Parallel and Distributed Computing, Vol. 69, Issue 6, 2009, pp. 491-507.

[10] Kramer, J., Magee, J., "The Evolving Philosophers Problem: Dynamic Change Management", IEEE Transactions on Software Engineering, Vol. 16, Issue 11, Nov. 1990, pp. 1293-1306.

[11] Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T., "An alternative to Quiescence: Tranquility", Proceedings of the 22nd IEEE international Conference on Software Maintenance - ICSM, Washington, DC, Sep. 2006, pp. 73-82.

[12] Preuveneers, D. and Berbers, Y. (2010) "Context-driven migration and diffusion of pervasive services on the OSGi framework", International Journal of Autonomous and Adaptive Communications Systems, Vol. 3, No. 1, 2010, pp. 33-22

[13] Malek, S., Edwards, G., Brun, Y., Tajalli, H., Garcia, J., Krka, I., Medvidovic, N., Mikic-Rakic, M., Sukhatme, G., "An Architecture-Driven Software Mobility Framework", Journal of Systems and Software, special issue on Software Architecture and Mobility, Vol. 83, Issue 6, Jun. 2010, pp. 972-989.

[14] Costa, P., Coulson, G., Mascolo, C., Mottola, L., Picco, G., Zachariadis, S., "Reconfigurable Component-based Middleware for Networked Embedded Systems", International Journal of Wireless Information Networks, Vol, 14, Nº 2, Jun 2007, pp. 149-162

[15] Calder, B., Krintz C., Hölzle U., "Reducing transfer delay using Java class file splitting and prefetching" ACM SIGPLAN Notices, International, 1999 ISSN:0362-1340.

[16] Cucinotta, T., Palopoli, L., Lipari, G., "Control Algorithms for Coordinated Resource-Level and Application-Level Adaptation I", Deliverable D-AQ2v1 from the FP6/2005/IST/5-034026 Framework for Real-time Embedded Systems based on COntRacts (FRESCOR), 2007.

[17] Adaptivity and Control of Resources in Embedded Systems (ACTORS), European Project, ref. 216586, http://www.actors-project.eu/, (May 2010).

[18] Faggioli, D., Trimarchi, M., and Checconi, F., "An implementation of the earliest deadline first algorithm in Linux", Proceedings of the 2009 ACM Symposium on Applied Computing (SAC '09), Honolulu, Hawaii, 2009, pp. 1984-1989.

[19] Resource ReSerVation Protocol (RSVP): RFC2205, Sep. 1997, http://tools.ietf.org/html/rfc2205.

[20] Bourges-Waldegg, D., Duponchel, Y., Graf, M., Moser, M., "The fluid computing middleware: bringing application fluidity to the mobile Internet", Proceedings of the 2005 Symposium on Applications and the Internet, Trento, Italy, 2005, pp. 54-63.

[21] Distributed and Mobile Framework for Real-Time Systems (DiseRTS), http://www.hurray.isep.ipp.pt/activities/RTSoft/distFrameworkOverview.ashx/, May 2010.