This paper was presented as part of the SOCNE workshop held in conjunction with IEEE ETFA'2011

Service Offloading in Adaptive Real-Time Systems

Luis Lino Ferreira, Guilherme Silva, Luis Miguel Pinho
CISTER/ISEP
Polytechnic Institute of Porto
R. Dr. António Bernardino de Almeida, 431
4200 – 072 Porto, Portugal
{llf, grss, lmp}@isep.ipp.pt

Abstract

Smartphones and other internet enabled devices are now common on our everyday life, thus unsurprisingly a current trend is to adapt desktop PC applications to execute on them. However, since most of these applications have quality of service (QoS) requirements, their execution on resource-constrained mobile devices presents several challenges. One solution to support more stringent applications is to offload some of the applications' services to surrogate devices nearby. Therefore, in this paper, we propose an adaptable offloading mechanism which takes into account the QoS requirements of the application being executed (particularly its real-time requirements), whilst allowing offloading services to several surrogate nodes. We also present how the proposed computing model can be implemented in an Android environment.

1. Introduction

Smartphones are nowadays an essential part of our lives, executing a multitude of applications, connecting us to social networks, online games, or internet calls. Some of these applications are monolithic, only being able to execute locally on the device, while others are distributable, being able to execute some parts locally and to request (existing) services from available nodes in the network. These systems are normally supported by high bandwidth networks.

Although the performance of mobile devices is increasing in an unprecedented way, they still do not possess the same features and resources of a common desktop or laptop PC. A potential solution is for the application to be distributed, running some parts locally and other on networked servers, which execute code that has been previously compiled and loaded.

Nevertheless, a more dynamic and flexible solution to solve the performance gap, is to allow the mobile devices to dynamically offload some of the applications' services to neighbor devices [1 - 7], taking advantage of collaborative environments, such as at home or in the car, or of infrastructures providing value-added services.

In comparison with more traditional distributed approaches, supported by "fat" network servers, the offloading solution has the following advantages: i) the code to execute is available in the client application; ii) the nodes to which computations are offloaded are nearer, consequently communications usually have less delays and better QoS; iii) the changes required on the original code are usually less significant. Consequently, several different types of solutions for code offloading have been provided, motivated by the need to obtain access to additional resources, like memory, power or more computation capabilities.

Some solutions rely on the programmer to determine which parts of the code to offload, such as Cuckoo [3], which provides an offloading environment for Android-based systems using its inter-process communication mechanisms. In MAUI [5] the programmer is responsible for the annotation of the methods which can be executed remotely, being power conserving its main objective. Other solutions adopt a more automatic approach, where the offloading framework is able, by itself, to analyze the code and determine which parts/classes can be offloaded, e.g. CloneCloud [4]. Furthermore, some of these algorithms are adaptive, i.e. they are able to dynamically, in run-time, determine an adequate application partitioning [6 - 7].

But none of these frameworks is capable of handling the application's real-time requirements; they mostly provide a best effort solution. The adaptive solutions also present the additional burden of calculating, in run-time, the application partitioning.

Code offloading also relies on libraries or frameworks that support the mobility of code or services. The work presented in [2] describes several service migration scenarios for embedded networks, based on the $\in$SOA framework. In [8] the authors propose MobFr, which supports code mobility and is also capable of providing the application with the required QoS resources, including its real-time requirements, as shown in [9].

It is in this context that in this paper we put forward a code offloading approach, allowing applications to offload some of their services to neighbor nodes. The goal is to support adaptable applications, which present variable QoS requirements, ranging from flexible sensor and control applications, multimedia streaming, or even gaming. This approach is built on top of MobFr [8]. The solution proposed in this paper addresses real-time applications which periodically run services with variable execution time.

As an example consider a physics modeling engine. These engines usually run periodically, with a period that depends on the application's specific rate, with a set of *Core Services* related with 3D/2D object movement, collisions, rendering, etc. When the size of data (number of simulated objects) is low, the mobile device handles all computations, but when the number of data items, or the computation requirements increases, only local execution may not be possible. In this case, instead of reducing the quality provided to applications (e.g., by reducing the quality of some of its computation (for instance the accuracy) or simply by reducing the rate, nearby nodes are sought to execute parts of the computation.

The offloading approach works by constantly monitoring the time required to execute the core services (t_{core}). Based on a set of past t_{core} times, the algorithm predicts the evolution of t_{core}; if it determines that the required rate cannot be achieved in the future, the offloading procedure is triggered in advance, allowing for the inexistence of timing errors. Some of the core services can be offloaded to other nodes and executed there without reducing the rate or the supported quality.

Complementary, when it is not anymore advantageous to execute the offloaded services in other nodes, migration can once again take place, and these services can return to be executed on the device.

To our best knowledge, this proposal is the first which proposes a solution to integrate dynamic real-time requirements into service offloading. The solutions proposed in [1 - 2] might also be capable of guaranteeing the real-time requirements of applications but their timing performance has not, yet, been studied. Additionally, the application architecture proposed in [2] is not adaptable to other kinds of application.

The remainder of the paper is structured as follows. In Section 2 we provide the motivation and the fundamentals of offloading approaches. The proposed algorithm is detailed in Section 3, whilst Section 4 provides some of its timing issues. Section 5 then describes the architecture of the Android code offloading implementation. Finally, Section 6 discuses the results and draws some conclusions.

2. Code Offloading Motivation

In order to illustrate how much faster a desktop PC is than a smartphone, we performed some tests using a 3 GHz Intel Pentium 4 PC and two mobile devices: one HTC Magic and one Samsung Galaxy S. Our tests consisted in running the physics simulator on them, which periodically calculates the paths of 100 rectangular objects bouncing in a small screen area. The results show that on average each device needs 1.2 ms, 27.7 ms and 250.0 ms, on the PC, on the Samsung Galaxy S and on the HTC Magic, respectively, for the calculation of each simulation step.

Figure 1 illustrates the typical scenario that motivates the use of the real-time offloading algorithm proposed in this paper. The vertical axis represents the time required to run the module core services (t_{core}), the horizontal axis represents the number of objects being calculated. The dashed line at 33.3 ms characterizes the desired execution rate – 30 updates/second. Hereafter, we refer to that time as T_p, the cycle period.

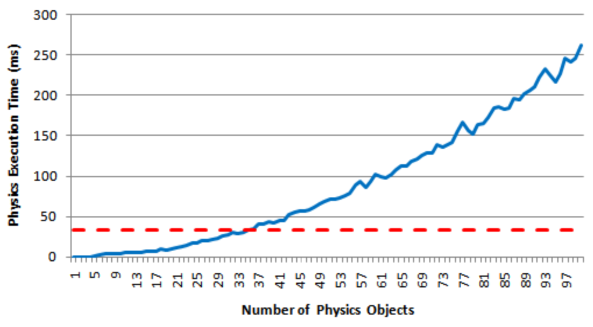

Figure 1 - Example Experience

If the execution time of the module is higher than 33.3 ms, then the application will no longer be able to give the desired frame rate to the user. Potential solutions to this problem would be to simplify the physics calculations or to reduce the details being rendered. However, this downgrade reduces the quality of the user experience leading to the impression of lower quality software or software with reduced functionalities. In Figure 1 it is noticeable that a mobile device cannot process more than 35 objects without exceeding T_p.

Generically, the application model results from the cyclic execution of specific application services which may have a variable execution time. The application is also required to maintain the periodicity on the execution of the core services and their QoS levels (e.g. the frame rate of the physics simulation).

The solution is to offload some of the calculations being performed by the module to surrogate nodes, preferably with more processing capacity and on a timely manner.

3. Real-time Offloading of Mobile Services

The algorithm being proposed relies on a code mobility framework in order to support the remote execution of code [1]; additionally, the time required to move the code to the surrogate node can be calculated based on the formulations proposed in [9], thus its timeliness can be statistically guaranteed.

The main objective of the algorithm is to dynamically adapt to the varying execution times by offloading computation to surrogate nodes in a timely way. By timely we mean that the user should not notice any disruption on the application behavior. To that purpose, the offloading algorithm tries to predict the forthcoming core execution times, based on past execution times.

Figure 2 illustrates the algorithm operation. The core services' execution time on the main and surrogate device is represented by square and triangle marks, respectively. The continuous line, without marks, shows the linear regression that best approaches the evolution of the t_{core} on the original device. This line is obtained by considering the 8 points, from 0 to 264 ms.

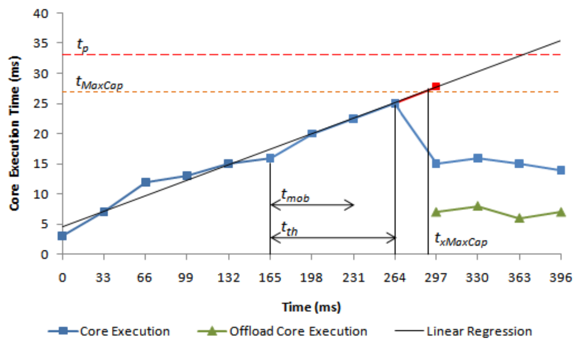

Figure 2 – Offloading algorithm example

Based on the linear regression parameters, at 165 ms, it is possible to estimate the time ($t_{xMaxCap}$) at which the core execution time will exceed the maximum capacity of the original device – t_{MaxCap}, which for the case of Figure 2 occurs in the interval between 264 ms and 297 ms. Note that t_{MaxCap} is the maximum reserved CPU capacity per cycle period (T_p), reserved by the underlying operating system resource manager. Obviously, to fulfill the operating objective of the application, no QoS disruption should occur, consequently the code mobility operation, which precludes the offloading procedure, should be completed prior to 264 ms.

If the time required for code mobility is equal to t_{mob} then we can express the offloading decision as:

$$t_{th} = \left\lfloor \frac{t_{xMaxCap}}{T_p} \right\rfloor \times T_p \leq \left\lceil \frac{t_{mob}}{T_p} \right\rceil \times T_p \qquad (1)$$

If this expression is true then the system should start the offloading procedure of its selected services in parallel with its current operations at 165 ms (as in previous works [3, 5] it is up to the programmer to determine the services to offload). The floor and ceiling functions, used in Eq. (1), normalize all results to multiples of T_p.

Figure 2 shows that prior to 297 ms the necessary code is transferred to a surrogate device, consequently at 297 ms there is a noticeable reduction on the core execution time on the main device, since the surrogate device enters into operation. In this example, there is only one surrogate device, but, if required, the middleware can handle offloading to a group of devices.

Note that it is up to the programmer to determine how to parallelize the application code, but the framework operates as follows: i) each of the surrogate devices receives new data from the main device; ii) executes the calculation over that data and returns the results; and finally, iii) after receiving all responses the coordinating device aggregates the responses from the surrogate devices with its local calculations.

3.1. Code Offloading Algorithm

The offloading algorithm pseudo-code is shown in Listing 1. This algorithm assumes that the application must make available a set of interface methods which can be used by the underlying offloading framework and that some are periodically called by the application itself. One of those methods is the `update` function, which is periodically executed by the application with a periodicity of T_p.

The `update` function starts by determining if the services had already been offloaded to other surrogate devices (line 2). If the condition is true then it tests to determine if an additional surrogate node is needed (lines 23– 31). That is done by testing if the execution time on each surrogate device will reach the maximum execution capacity in that node using Eq. (1). Note that this test also includes the main device. The `requiresNewSurrogate` function also calls `tryRebalance` function, which is an application dependent function that tries to determine if by rebalancing the load between nodes it is possible not to add a new surrogate device or not.

If an additional surrogate is needed then the `addAdditionalDevice` thread is started (lines 33 – 40). This thread might require a few cycles to complete due to the time consuming sequence of operations that it must execute when using the underlying code mobility framework [1]. This thread starts by determining if there is a node, with available resources in the neighbourhood (line 34), after it offloads the required code to it (line 37), otherwise an error is signalled to the application. It is important to note that once started the offloading process is not stopped, although the main node may choose not to execute any offloaded service.

Listing 1

```
1.  Function update() {
2.   If isOffloading()
3.      If requiresNewSurrogate()
4.         new Thread(addAdicionalDevices())
5.         runOffloaded()
6.      else {
7.         if needsToStopOffloading() {
8.            runLocally()
9.         } else
10.           runOffloaded()
11.        }
12.    }
13. else //if is not offloaded
14.    If requiresNewSurrogate() {
15.       new Thread(addAdicionalDevices())
16.       runLocally()
17.    } else
18.       runLocally()
19.    }
20. }
21. }
22.
23. Function requiresNewSurrogate() {
24. Output result:bool - determines if a new
surrogate node is required.
25.
26. ForEach(dev in devices) {
27.    If eq (1) is true
28.       If !tryRebalance() Return true
29.    Return False
30. }
31. }
32.
33. Thread addAdicionalDevice() {
34. newDev = DiscoveryManager.getDevice()
35. If (newDevice != null)
36.    Devices.add(newDev)
37.    OffloadCode(newDev)
38. Else
39.    signalError()
40. }
```

The algorithm then runs the core services in offloaded mode (Listing 2). Basically, it starts by partitioning the data to be computed by surrogate devices. This operation is done by an application specific function and it can be adjusted on every cycle, e.g. for load balancing proposes. After, the data is sent to every surrogate device, computed and the results are returned, using the function `sendData&Execute`. The last step is related to the aggregation of results on the main node in order to compute the final results.

Listing 2

```
1. Function runOffLoadded(offloadService)
2. Input offloadService: the code that can be
executed in offloading.
3. {
4.   parts = offloadService.dataPartitioner()
5.   sendData&Execute(devices, parts)
6.   Result[0] =
offloadService.runLocally(parts[0])
7.   receiveData(devices, results)
8.   offloadService.aggregateResults(results).
9. }
```

Listing 1, also accommodates the case when the main node is not offloading any computations (lines 15 – 20). In this case, it determines if a surrogate is needed, and, if needed, it releases a thread that runs the function `addAdditionalDevice` to prepare the offloading of code. Meanwhile, the code is executed locally.

Another situation occurs when the load no longer justifies the offloading procedure, this condition is tested in line 7, but in this paper we do not elaborate any further in this subject.

4. Timing Issues

In Section 3, some timing parameters were not detailed, in this section we give details on how to determine the time when the linear regression line, which is used to predict the evolution of the core execution time, reaches the maximum capacity. We also explain how the core execution time is measured and how the mobility timings can be obtained.

4.1. Determining $t_{xMaxCap}$

To determine when to start the offloading procedure, Eq. (1) requires the knowledge of $t_{xMaxCap}$ time, the time at which an estimated value for the core execution time (t_{core}) reaches the maximum capacity (t_{maxCap}).

The solution we propose is to use linear regression to determine the line which best approaches the evolution of the core execution times and determine when that line crosses the maximum capacity line.

Each point *i* of the estimation line is expressed by the formula $t'_{core,i} = m.t + b$, where *m*, the line slope, is calculated by solving the following equation:

$$m = \frac{n\sum_{i=0}^{n}(t_i \times t_{core,i}) - \sum_{i=0}^{n} t_i \times \sum_{i=0}^{n} t_{core,i}}{n\sum_{i=0}^{n}(t_i)^2 - (\sum_{i=0}^{n} t_i)^2} \qquad (2)$$

In this equation *n* is the number of past core execution times being considered. Parameter t_i is the time at which the core execution time ($t_{core,i}$) had been measured.

The setting of *n* has a big impact on the behaviour of the algorithm. If *n* is set to a small value then the algorithm becomes more sensitive to rapid changes on the t_{core} value, otherwise the algorithm is slower to react.

Parameter *b* is calculated by solving the following equation:

$$b = \frac{\sum_{i=0}^{n} t_{core,i} - m\sum_{i=0}^{n} t_i}{n} \qquad (3)$$

After having calculated m and b it is possible to determine $t_{xMaxCap}$ as follows:

$$t_{xMaxCap} = \frac{t_{maxCap} - b}{m} \qquad (4)$$

Obviously, other regression algorithms could be used, like a polynomial regression, but the number of calculations to be performed would be much higher, although the results could potentially also be more precise, particularly, when the variation of the core execution time does not follow a linear rule. The main advantage is that the proposed algorithms can be executed with minimum overhead in devices with limited computation capabilities.

Nevertheless, it is possible to increase the performance of the linear regression parameters calculation by: i) if *m* is negative then calculating *b* is not necessary since the line will not cross the maximum capacity line in the future; ii) the summations which are required to be calculated in Eq. (2) and (3) can use previously calculated values. As an example, the calculation of $a = \sum_{i=0}^{n} t_i$, can be done using the following recurring formulation:

$$a_x = a_{x-1} - a_{x-n} + a_x \qquad (5)$$

Where, a_0 is the summation of the first *n* values.

4.2. Code Mobility timings

Another value required to determine when to start offload is the interval of time that elapses from the time when the offloading decision is taken until the new device is ready to start executing the offloaded code – the code mobility time (t_{mob}).

We support this calculation on the formulations proposed in [9], which are adapted to this specific case. Therefore, t_{mob} can be calculated by:

$$t_{mob} = t_{conf} + t_{code} + t_{ist} + t_{start} \qquad (6)$$

Where t_{conf} is the time required to find a feasible system configuration, i.e. a surrogate node where to offload the code. Time t_{code} is the time required to transmit the offloaded code. Some configuration data can also be sent along with the code, which requires a time of t_{ist} to be transmitted. Finally, the code must be installed on the surrogate node and started, prior to be ready to start processing items sent from the source node, thus requiring a time of t_{start}. It is important to note that these timings are not worst-case timings, but they represent just average or any other kind of statistic value.

4.3. Measuring t_{core}

An essential part of the algorithm is to be able to measure the core execution time of all surrogate nodes (t^{s}_{core}, $s \in \{1, 2, ..., nSurr\}$) and on the local node (t^{0}_{core}).

On the local node this time is simply the execution time of function `runLocally()`.

The measurement of the core execution time on the surrogate nodes is performed at the source node, since it must also take into account the communication delays, consequently:

$$t^{s}_{core} = t^{main\to s}_{req} + t^{s}_{exec} + t^{s\to main}_{res} \qquad (6)$$

Where $t^{main\to s}_{req}$ represents the time required for the data to be sent from the source to the destination node. Time t^{s}_{exec} is the execution at surrogate node *s* and $t^{s\to main}_{res}$ is the time required to transmit the response from the surrogate node back to the source node.

5. Implementation in Android

The proposed architecture is based on the code mobility framework (MobFr) for the Android operating system, which has been proposed in [8]. The MobFr is a service-based QoS-aware framework capable of handling code mobility in a cooperative environment.

Among other characteristics, the MobFr is designed to: i) detect neighbour devices; ii) determine the best candidate where to run the offloaded code, according to the QoS requirements of the application and the available resources on the surrogate nodes; iii) migrate the code and initial state; iv) remotely control the code execution; and finally, v) handle the transfer of data between nodes.

The core modules provided by the framework are the: *Discovery Manager*, *Package Manager*, and *Execution Manager*. Additionally, the framework also relies on a *QoS Manager* module (not shown in Figure 3) that is responsible for assuring that the QoS requirements of each module can be met.

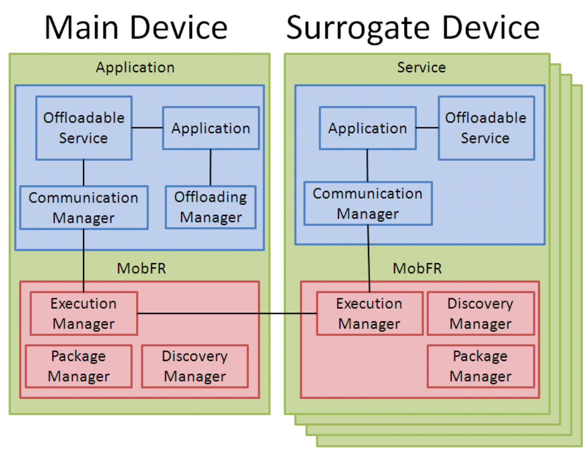

Figure 3 - Framework Structure

The *Discovery Manager* module is designed to discover neighbor devices on a local network, advertise the host's resource availability and gather information about the resource availability on neighbour devices. The *Package Manager* is used to install, uninstall and transfer services. This module is also responsible for the interaction with the *QoS Manager* in order to request specific QoS levels for the service being transferred. The *Execution Manager* allows executing services on a surrogate node through the exchange of Android intents, thus allowing the development of transparent applications (in relation to its distribution). The *QoS Manager* administers the system resources, either locally, on a node, or in a distributed environment. It also encapsulates the functionalities of high level QoS control frameworks, like the one used in [8]. Consequently, this module can interact with remote code offloading framework modules in order to choose the most appropriate nodes where to run the offloaded services. The Code offloading framework and its modules is depicted in Figure 3.

In this framework the *Communication Manager* takes care of communications between the main and surrogate devices. Basically, this module implements the functionalities described in Listing 2: sending data, receiving and aggregating the results. It is the responsibility of the *Offloading Manager* to take care of the initial configuration, monitoring the core execution times and control the creation of new surrogates.

The *Offloadable Service* represents the application modules which can be offload to other nodes.

Figure 4 presents the framework's UML model, which illustrates the dependencies between the framework, the underlying MobFr framework and the application.

Any class which can be offloaded must extend the abstract class `OffloadableServiceAbstraction`, which defines some methods and defines the interface for the implementation of application specific methods. The `Update` method is one of those. This method should be periodically called by the application. The `runOffloaded` method is another, it takes care of running the offloaded services and aggregating the results. The Offloadable Service must also implement the abstract methods required for data partitioning (`dataPartitioner`) and the `runLocally` method which runs the service on the main device.

The `CommunicationManager` class handles all interactions with the service mobility framework. This class also has access to a list of devices in the network, which is used to find an adequate set of surrogate nodes.

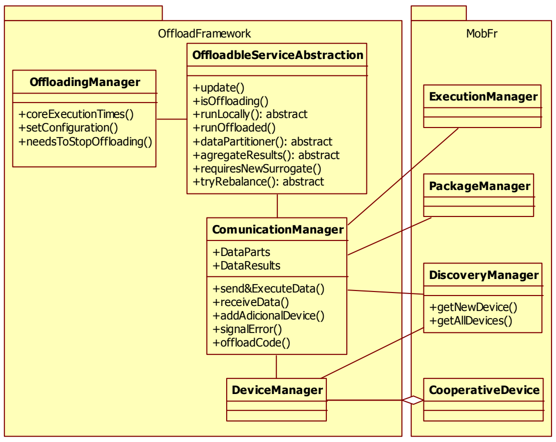

Figure 4 – Framework's UML model

6. CONCLUSIONS

The use of smartphones and other internet enabled devices is changing the habits of users, which more and more require that their desktop applications are seamlessly supported in these resource-constrained devices. One solution to support these requirements is to offload some of the applications' services to devices nearby, taking advantage of high-capacity local networks. Code offloading techniques have proven to be useful in increasing the performance or the battery life of mobile devices.

In this paper, we put forward an offloading mechanism that considers the QoS of the applications, offloading services to neighbor nodes and, at the same time, adapting to changing real-time execution parameters of the application. The implementation of this approach in an Android environment is also outlined.

Acknowledgments. This work is partially funded by the CMU-Portugal Program, funded by National Funds (PT), through the FCT - Portuguese Foundation for Science and Technology, under the Carnegie Mellon Portugal Program, with Grant ref. FCT-CMU-PT/0012/2006 and the CISTER Research Unit (608FCT) funded by FEDER funds through COMPETE (POFC - Operational Programme 'Thematic Factors of Competitiveness) and by National Funds (PT), through the FCT - Portuguese Foundation for Science and Technology.

References

[1] Nimmagadda, Y., Kumar, K., Lu, Y., Lee, C.S.G., "Real-time moving object recognition and tracking using computation offloading", IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS), Oct. 2010, pp. 2449 – 2455.

[2] Sommer, S., Schola, A., Gapanova, I. , Knoll, A., Kemper, A., Buckl, C., Kainz, G, Heuer, J., Schmitt, A., "Service Migration Scenarios for Embedded Networks", 2010 IEEE 24th Intl. Conf. on Advanced Information Networking and Applications Workshops, 2010, pp. 502 – 507.

[3] Kemp, R., Palmer, N., Kielmann, T., Bal, H., "Cuckoo: a Computation Offloading Framework for Smartphones", Proc. of the 2nd Intl. Conf. on Mobile Computing, Applications, and Services, (MobiCASE '10), 2010.

[4] Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., Patti, A., "Clonecloud: Elastic execution between mobile device and cloud", In EuroSys 2011, April 2011.

[5] Cuervo, E., Balasubramanian, A., Cho, D., Wolman, A., Saroiu, S., Chandra, R., Bahl, P., "MAUI: making smartphones last longer with code offload", Proc. of the 8th Intl. Conf. on Mobile systems, applications, and services (MobiSys '10). ACM, Jun. 2010, pp. 49 – 62.

[6] Gu, X., Nahrstedt, K., Messer, A., Greenberg, I., Milojicic, D., "Adaptive Offloading Inference for Delivering Applications in Pervasive Computing Environments", First IEEE International Conference on Pervasive Computing and Communications (PerCom'03), 2003, pp. 107 – 114.

[7]Xian, C., Lu, Y., Li, Z., "Adaptive computation offloading for energy conservation on battery-powered systems", International Conference on Parallel and Distributed Systems, 2007, vol. 2, Dec., 2007, pp. 1 – 8.

[8] Gonçalves, J. Ferreira, L. Pinho, N., Silva, G., "Handling Mobility on a QoS-Aware Service-based Framework for Mobile Systems", Intl. Conf. on Embedded and Ubiquitous Computing (EUC 2010), Dec. 2010, pp.97 – 104.

[9] Ferreira, L., Nogueira, L., "On the Use of Code Mobility Mechanisms in Real-time Systems", accepted for publication at the 10th Intl. Workshop on Real-Time Networks, Jul., 2011.