On the Use of Code Mobility Mechanisms in Real-Time Systems

Luis Lino Ferreira
CISTER Research Centre / School of Engineering of the Polytechnic Institute of Porto
Porto, Portugal
llf@isep.ipp.pt

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

ABSTRACT
Applications with soft real-time requirements can benefit from code mobility mechanisms, as long as those mechanisms support the timing and Quality of Service requirements of applications. In this paper, a generic model for code mobility mechanisms is presented. The proposed model gives system designers the necessary tools to perform a statistical timing analysis on the execution of the mobility mechanisms that can be used to determine the impact of code mobility in distributed real-time applications.

Keywords
Real-time systems, distributed embedded systems, mobile systems, code mobility, quality of service.

1. INTRODUCTION

Real-time systems are increasingly shifting from a set of small, local applications to powerful, resource-hungry, open distributed applications [4]. By the very nature of open real-time systems, the availability of resources is unknown beforehand and service provisioning can only be determined dynamically as new applications arrive to the system. Consequently, there is an increasing demand for supporting distributed applications with the flexibility to offload parts of their computations to neighbour or "in the cloud" nodes due to local resource scarcity, while ensuring the real-time behaviour of these applications, both during execution and during reconfiguration, after mobility of code has occurred.

Therefore, open real-time distributed systems must provide applications the support to: i) use services provided by remote components; ii) move part(s) of the application's code to remote nodes; and ii) guarantee real-time behaviour. The first requirement can be supported by a service-based infrastructure [4], to easily and transparently interconnect local and remote parts of an application. The second requirement can be supported by code mobility frameworks, allowing the installation and execution of parts of an application in remote nodes [9]. Finally, a real-time resource manager can support the third requirement. A well-established solution is to use capacity reserves. This has been proved to be successful in improving the response times of soft real-time tasks while preserving all hard real-time constraints, both for CPU [3] and network [2] scheduling.

1.1 Related work

Although not widely studied, a few solutions have already been proposed to analyse the impact of code mobility on the real-time requirements of applications.

In [11], the authors propose and experimentally characterise the behaviour of a hard real-time framework that supports the migration of tasks between nodes. However, the work does not propose a mathematical model that enables system designers to account for the impact of the mobility protocol on the overall timing behaviour of applications.

A strategy for minimising the impact of code mobility in a hierarchical preemptive fixed priority scheduling system for Real-Time Java is proposed in [10]. The authors mainly determine the points in time at which the migration process should be started, which guarantees that the deadlines of tasks are met and that the migration process is executed between consecutive evocations of a migratable task.

Statefull services require the transfer of state, whose duration depends on the length of the data being transferred. However, during this period of time no transactions can be executed on that service (known as blackout time). However, such determination is only possible in systems with a well-known and controlled timing behaviour. Therefore, in [12], the authors tackled the problem of minimising the blackout time by proposing a partial blocking and a non-blocking approach for state transfer, which are capable of providing real-time guarantees.

Nevertheless, none of these works focus on the mobility mechanism itself. Such mobility mechanisms should be supported by a mobility framework that enables the runtime relocation of services in response to reconfiguration/update events (e.g., the system might reconfigure itself due to the disappearance of a node involved in a computation).

As an example, consider running a video game on a mobile device that has offloaded parts of its computations to neighbour devices. Reconfiguration in such a distributed cooperative execution might be required if one of the nodes, currently running one of game's components, is no longer capable of outputting the required QoS. In such case, the component can be migrated to another node able to supply the required QoS. Ideally, such change should be executed seamlessly, i.e. the game delays should (preferably) be unnoticeable.

Examples of works that tackle the specific problem of selecting the new distributed configuration are [4] and [1]. The former allows the determination of a distributed configuration that maximises the satisfaction of the user's QoS preferences among a set of allowed QoS levels. The latter tries to fulfil the same goals, but each service is only allowed to specify a single QoS level. However, a mechanism to determine the impact of code mobility in distributed real-time applications is still missing.

1.2 Contributions and paper structure

Service mobility in a distributed execution environment is a complex operation that evolves through several phases, including

sending the code and state to the destination node and rebinding connections between components. Additionally, resources must be explicitly reserved on the destination node, prior to the start of the mobility process. Due to its complexity, we propose that a Mobility Management framework (represented in Figure 1 by Mx) should control the mobility of code between nodes of a distributed system. This paper focuses on the model and timing analysis for a generic code mobility mechanism for distributed soft real-time applications. The proposed model is generic enough, helping the system designer to define the most appropriate parameters for the mobility management modules and to determine the feasibility of the timing constrains imposed on applications, including mobility and reconfiguration events.

The remainder of the paper is organised as follows. Section II defines the generic model for the distributed applications and for the mobility mechanism. Section III discusses and analyses the code mobility phases and their timings. The main consequence of the mobility mechanism is the introduction of a bounded inaccessibility period during which one of the application's services is not available. The proposed analysis allows computing the adequate resources required by the mobility framework to guarantee the timeliness of the application. Finally, Section IV discusses the model provided in the paper and presents some conclusions

2. System Model

2.1 Module components

This work applies to soft real-time applications composed by a set of interconnected components, each supplying some service, either in the same local node, but particularly when components are distributed among several nodes. The model considers the system to be composed of a set of N nodes $\{H_1, ..., H_N\}$ and a set of M services $\{S_1, \ldots, S_M\}$. Services are interconnected through links. $l_{x,y}$ characterises a connection between services S_x and S_y, (Figure 1). Each service and each link has a set of real-time requirements that are out of the scope of this paper (a detailed discussion can be found in [4]).

Each node runs a *Mobility Management* module M_x, where x is the index of the node. Each module M_x can be connected to other Mobility Management modules M_y through a network connection, $l_{mx,my}$.

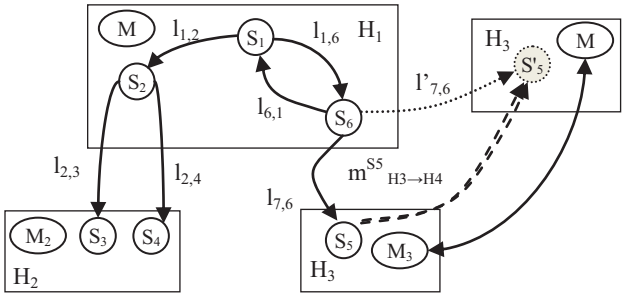

Figure 1. System Model

As depicted in Figure 1, an operation $m^{S5}_{H3\rightarrow H4}$ represents the mobility of service S_5 between nodes H_3 and H_4. In such case, H_3 is denoted as the *source* node and H_4 is denoted as the *destination* node. Link $l'_{7,6}$ represents the connection that has to be established after the mobility operation is completed (rebinding). Consequently, connection $l_{7,6}$ will have to be safely deleted prior to $l'_{7,6}$ becomes operational. By safely, we mean that no messages should be lost or delivered to wrong nodes. This operation implies offloading the code of S_5, its data state, and rebinding its connections, all within timing constraints.

2.2 Resource management

It is assumed that access to the system's resources can be modelled as a set of isolated servers, either related to CPU [3] or network [2] scheduling. Each of these servers is characterised by its maximum reserved capacity (Q_i) that can be used during a period (T_i); at the end of this period the capacity is replenished. Other CPU schedulers can also be used, like the Capacity Sharing and Stealing scheduler (CSS) proposed in [13]. For the network scheduling, any scheduling algorithm with similar characteristics can also be used, like the ones based on the Flexible Time-Triggered approach [14].

Based on this guarantees, it is possible to determine services' average response time using the formulations proposed in [3]:

$$R_i^{avg} = C_i^{avg} + (T_i - Q_i)\sum_{k=0}^{+\infty}\left(1 - F_C(k \times Q_i)\right) \qquad (1)$$

where C^{avg}_i represents the average execution time of task Ti and $F_C(x)$ is the cumulative distribution function (c.d.f.) of the task's execution time. In the remainder of the paper, we will use the notation $R(Q_i, T_i, F_C^i())$ to represent Equation (1).

2.3 Mobility Management Framework

We assume the existence of a modular Mobility Management framework in each node, similar to the one proposed in [9].

These mobility management modules have CPU and networking servers assigned to them, guaranteeing the timing requirements of its operations. Servers associated with the CPU offer a capacity of C_F over a period T_F. Network resources are split between two channels, one for bulky data transfer and another for the exchange of short control messages. The first has a capacity of B_{data} and a period of T_{data} while the second has a capacity of B_{ctrl} and a period of T_{ctrl}. The main advantage of using these two channels is that we can guarantee small response time for control messages, but for larger data transfer we are able to make the transfer with small overhead.

2.4 Service's internal state

In the proposed model, services are able to split their internal state into different *State Items*, representing different variables, different objects or combinations of both. It is up to the service to define how state items are configured. The state of a service is thus a set of state items defined exclusively by the service, where a State Item (SI^{Si}_p) is only associated to a service S_i, is defined as a tuple:

$$SI_p^{Si} = \{ID_p^{Si}, B_p^{Si}\}$$

ID^{Si}_p univocally identifies this State Item and B^{Si}_p is the bandwidth required for the transfer of this state item. Some state items are created during the service initialisation and are not changed subsequently, while others are updated regularly when service calls are executed. Therefore, state items are divided in two groups: one

that can be migrated during the normal operation of the service (S*tatic Sate Items*) and another that can only be migrated if there are no ongoing service calls (*Dynamic State Items*).

Based on the model exposed in this section, Section III shows how it is possible to devise a timing model for a generic mobility mechanism.

3. Code Mobility Timing model

Service mobility can be split in two main phases: *Preparatory* and *Blackout* phases. During the *Preparatory* phase, the migrating service continues operational in the source node. This phase is further divided into three subphases: *mobility decision*, *code shipping*, and *initial state transfer*. During the *Blackout* phase, the service is totally inaccessible to others. It includes the subphases: *state transfer*, *connections rebinding*, and *service restart*. Some of the subphases are executed serially while others can be executed in parallel. Figure 2 depicts a timeline containing an example of a mobility procedure. A detailed description and analysis of each step is given in the following subsections. In this analysis, for the sake of simplicity, we assume that no other service mobility operation occurs during the complete procedure and that a higher-level resource control framework assures such control.

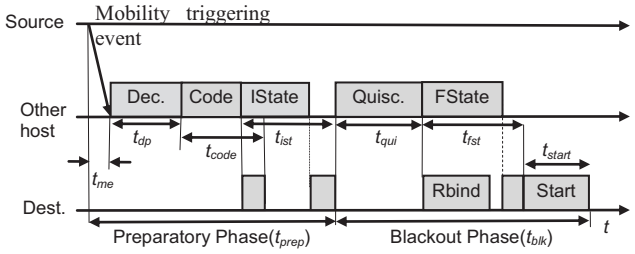

Figure 2 – Mobility-related timings

3.1 Preparatory Phase

3.1.1 Decision process

The start of service mobility (*mobility triggering event*) results from a decision by the application (currently using the service) or by request from an external entity (the user, another application or specific framework). As an example, in Figure 2, the triggering event is received from another node.

State changes can also trigger service mobility whenever a user requests the execution of an application that can only be admitted into the system if the system is reconfigured by migrating some services of previously admitted applications to neighbour nodes. As an example, consider that a user decides to play an mp3 file in its mobile device, having to migrate part of a local application to a neighbour notebook.

Note that migration can only be allowed if there is a feasible system configuration that allows the service to continue operating within its required QoS levels. Algorithms such as those provided by the Prism [1] or CooperatES [4] frameworks take a high-level approach, finding a solution for the distribution of the application services between nodes in a way that maximises a global utility function and, simultaneously, guarantees enough resources (CPU, network, memory, etc) for every admitted service. While the former assumes just one possible QoS level to the application, the latter assumes that each service can work with multiple QoS levels, each one with a different utility value to the overall system. Additionally, the algorithms proposed in [4] are capable of generating a system configuration in a bounded amount of time. These algorithms are able to use a global view of the system state or can simply use a partial view of the system, e.g. if the node computing a decision only has access to a limited number of nodes.

We should point out that the algorithms proposed in [1] and [2] do not take into account the cost introduced by systems reconfiguration and particularly code mobility.

3.1.2 Code shipping

After finding a new distributed solution, the source node informs the destination node of the QoS requirements for the service being migrated. The destination node can then make all necessary local confirmations on the feasibility of receiving the service.

Then, the service code is coded (*e.g.* for data serialisation) for transmission on the source node, shipped through the network, and decoded on the destination node (e.g. by using a deserialisation method).

The bandwidth required to transfer the code is equal to β_{code}, a constant, since the code size is not expected to vary during transit. Therefore, the average time required for the transmission of code (t_{code}) can be calculated by:

$$t_{code} = R\left(C_F^S, T_F^S, F_C^{code,S}()\right) + R(B_{data}, T_{data}, \beta_{code}) + R\left(C_F^D, T_F^D, F_C^{code,D}()\right) \quad (2)$$

$F_C^{code,S}()$ and $F_C^{code,D}()$ are the c.d.f. of the execution time required by the framework, on the source and destination nodes, respectively.

3.1.3 Initial state transfer

We assume that a set of *Static State Items* (*e.g.* configuration data) can be transferred prior to the quiescence of the service on the source node. After the transfer of the state items, the destination node acknowledges its reception. Consequently, the delay associated with the initial state transfer is given by:

$$t_{ist} = R\left(C_F^S, T_F^S, F_C^{ist,S}()\right) + R\left(B_{data}, T_{data}, F_C^{ist,N}()\right) + R\left(C_F^D, T_F^D, F_C^{ist,D}()\right) \quad (3)$$

where $F_C^{ist,N}()$ is the c.p.f. for the required bandwidth, $F_C^{ist,S}()$ and $F_C^{ist,D}()$ are the c.p.f. of the CPU processing time, on the source and destination node, respectively.

3.1.4 Total delay of the Preparatory phase

The time required for the *Preparatory* phase is given by:

$$t_{prep} = t_{me} + t_{dp} + t_{code} + t_{ist} - R\left(C_F^D, T_F^D, F_C^{code,D}()\right) \quad (4)$$

where t_{me} is the time that elapses from the event that triggered the mobility of a service until being received by the node responsible to determine a new system configuration. It is assumed that the new system configuration is computed in a bounded time t_{dp} [4].

It is important to note that, depending on the scenario, some of these timings can be equal to zero. As an example, assume the case where it is the user that decides to migrate its application from its mobile device to its TV.

Most importantly, during this phase the service continues totally operational, but the characterisation of this delay is required in order determine the dynamic of the mobility procedure.

3.2 Blackout Phase

3.2.1 Quiescence achieving

Usually, in reconfiguration operations, the service to be updated has to be in a safe state called quiescence [7]. In this state, the service being migrated:

i) is not currently engaged in a transaction;
ii) will not initiate a new transaction;
iii) is not servicing a transaction; and
iv) no transaction has or will be initiated by other services that require service from this service. At the same time, all services connected with the migrating service must go into a passive state, which requires the fulfilling of condition i) and ii).

One initial solution to achieve quiescence has been proposed in [7], while a less demanding solution, called tranquillity was later proposed in [8]. Achieving quiescence requires the completion of pending requests by a service and the knowledge of all other services that might issue new requests. These other services must evolve into a passive state in which they cannot evoke the service being migrated, although they can evoke other services available in the system. The time needed to achieve quiescence can be determined through a timing analysis of the mechanisms proposed in [7] or [8]. This calculation, out of the scope of this paper, is assumed to be known and equal to t_q.

We argue that achieving quiescence is not a necessary condition for the mobility of services in a distributed system, as shown by the implementation described in [9], if the service calls are stored by the mobility management and delivered to the destination node only after the completion of the mobility procedure.

3.2.2 Final state transfer

Several different approaches can be considered for state transfer:

i) transfer all state in a single bundle [10];
ii) propagate only the operations done on state items [5];
iii) separate the state space into several groups of items, each transferred with its own periodicity [6] or;

retransmit the state whenever it changes [12]. The mobility model here considered adapts to these approaches.

The final state transfer is the subphase that most influences the latencies of a service migration, due to its duration and due to the service being in a quiescent state (it involves the transfer of *Dynamic State Items* which can only maintain consistency if the service is not operational).

The set of state items that can only be transferred after achieving quiescence require a bandwidth of $F_C^{fst,N}()$ and CPU processing requirements of $F_C^{fst,S}()$ and $F_C^{fst,D}()$, respectively on the source and destination nodes.

CPU processing time is required for the preparation of the data to be sent and the required processing time to decode the data on the destination node. Therefore, the final state transfer duration (t_{fst}) can be calculated, similarly to the case of t_{ist}, as follows:

$$t_{fst} = R\left(C_F^S, T_F^S, F_C^{fst,S}()\right) + R\left(B_{data}, T_{data}, F_C^{fst,N}()\right) + R\left(C_F^D, T_F^D, F_C^{fst,D}()\right) \quad (5)$$

3.2.3 Connections rebinding

In the migration process, connections between services need to be changed according to the new location of the migrating service.

This procedure can be performed in parallel with the *final state transfer* and it involves the exchange of messages between 2 or more nodes: the source, destination and, if any, other nodes whose services connect to the service being migrated. It mainly requires the exchange of messages containing the location of the new end points, which requires a bandwidth of β^{Si}_{reb}. Therefore, assuming that service S_i has $ncon^{Si}$ connections with other nodes, the total bandwidth required to rebind all connections (β_{reb}) is $ncon^{Si} \times \beta^{Si}_{reb}$. The time required internally by each service to change the connection end point addresses is considered negligible.

The exchanged messages can also be used to withdraw all connected services from the passive state. Therefore, the rebinding time (t_{rbind}) is given by:

$$t_{rbind} = R\left(B_{ctrl}, T_{ctrl}, \beta_{reb}\right) \quad (6)$$

Since the number of exchanged message can be high, but with a small payload, its transmission is performed by the communication server assigned for control messages.

3.2.4 Service restart

The final subphase, which starts at the end of both the connection rebinding and final state transfer subphases, is responsible for the restart of the service on the destination node. All code and state must already be on the destination node and all necessary operations for the installation of the service (if required) have been completed. After being started the service re-establishes its internal state using the state items previously transferred and enters full operation. This operation is performed by the service using its scheduling budget (C^D_{Si}, T^D_{Si}), and therefore the time required for service restart is given by:

$$t_{rstart} = R\left(C_{Si}^D, T_{si}^D, F_C^{rstart,D}()\right) \quad (7)$$

where $F_C^{fst,S}()$ is the p.d.f. of the CPU requirements for service restart.

3.2.5 Total delay of the Blackout phase

During this phase, all transactions involving the migrating service are stopped, thus leading to a blackout period (t_{blk}). On a real-time system this time is particularly important since it influences the timeliness of the distributed application. Therefore, the total duration of the *Blackout* phase is given by:

$$t_{blk} = t_q + max\{t_{fst}, t_{rbind}\} + t_{rstart} \quad (8)$$

Since the final state transfer and the rebinding of connections can be executed in parallel, then we use the function $\max\{t_{fst}, t_{rbind}\}$ to determine the maximum of both subphases.

As discussed previously, the Quiescence Achieving subphase might be eliminated if the system is supported by adequate mobility management facilities. The rebinding process is based on a simple exchange of messages and on the reconfiguration of transmission and receptions ports. The service restart is an operation with a small overhead. But, the final transfer subphase delay varies with the size of the data being transferred. Particularly, when the state size is high, strategies like the ones proposed in [12] can be used in order to reduce t_{fst}. Such strategies enable the implementation of partial blocking and non-blocking approach on service calls for a migrating service.

4. Mobility framework Architecture

A Mobility Framework, which enables the mobility of services on the Android Operating system, has been developed [9]. The framework will used to demonstrate the use of the proposed model on real scenarios.

The framework is implemented as an Android service, which takes care of service migration to and from another node, at the same time it interacts with the operating system Resource Manager in order to determine if the QoS requirements of the service can be supported.

The Android operating system is used both due to its open source nature to its innovative architecture. Although its use to support real-time applications is still debatable [15] it nevertheless provides a suitable architecture for quality of service-aware applications in ubiquitous, embedded systems [16].

The core services provided by the framework are the: *Discovery Manager*, *Package Manager*, *State Manager* and *Execution Manager*. Additionally, the framework also relies on a *QoS Manager* module that is responsible for assuring that QoS requirements of each service can be meet. Figure 3 depicts the main modules of the framework.

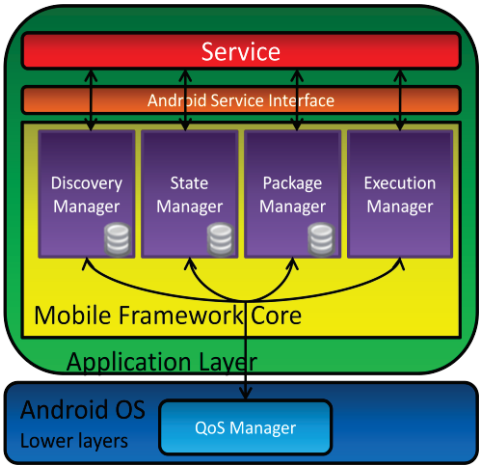

Figure 3 – Mobility framework

The *Discovery Manager* module is designed to discover neighbour devices on a local network and advertise the host device capabilities. Advertise messages contain information about the applications and services installed, their associated Intents interfaces and QoS requirements. Originally, Android intents provide the means for the reutilization of functionalities implemented by other application installed in the same device. Therefore, the *Discovery Manager* provides a standard mechanism, for each node, to obtain information about installed services and about the availability of resources in neighbour devices. It also keeps track of node and service disconnections from the network.

The *Package Manager* is used to install, uninstall and transfer the code of Android services, which are contained in *APKs* files. This module is also responsible for the interaction with the *QoS Manager* in order to request specific QoS levels for the service being handled. Therefore, it is the responsibility of the *QoS Manager* to accept or reject service installations, particularly if the QoS required level cannot be guaranteed.

The *State Manager* handles both the initial and final state transfer operations in a flexible way, based on the state items paradigm.

The *Execution Manager* allows launching services on a host device or on a remote node through the exchange of Android Intents that allow the programming of transparent applications (in relation to the distribution). In this implementation an Intent resolution procedure, based on the data collected by the Discovery Manager, determines if the Intent can be run locally or if it must be redirected to the node, where the service is running.

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 defined in [4]. Consequently, this module can interact with our framework conveying orders for the deployment of services in the distributed system.

5. Conclusions and Future Work

In recent years, many real-time systems have become open to unpredictable operating environments where both system workload and platform may vary significantly at run time. As such, the set of applications to be executed and their aggregate resource and timing requirements are unknown until runtime but, still, a timely answer to events must be provided in order to guarantee a desired level of performance.

In this context, a distributed execution of resource intensive applications among neighbour nodes seems a promising solution to address the increasingly complex demands on resources and desirable performance.

This paper proposed a generic model for code mobility in soft real-time systems, where applications are constituted by interconnected distributed services.

The main consequence of mobility to the running application is that it might result on a temporary degradation on the provided quality of service, due to the consequent blackout period. We state that it is up to the application programmer to determine the amount of degradation that can be supported by the application.

As such, this work gives the system designer the necessary tools to perform a statistical timing analysis on the execution of the mobility mechanisms and to determine the most appropriate parameters of the mobility framework components, either in relation to the local (CPU) or to network resources.

The proposed model divides the mobility mechanism in two phases, thus allowing a reduction on the time during which a service is inaccessible (the *Preparatory* phase is not considered).

This work can leverage future research in the field of code mobility and service update in distributed real-time systems. The proposed analysis can support the development and evaluation of suitable mobility mechanisms. Future work will focus on the use of the state items paradigm to propose new state transfer algorithms.

REFERENCES

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

[2] T. Nolte and K. Lin, "Distributed Real-time System Design using CBS-based End-to-end Scheduling," in Proc. of the 9th International conference on Parallel and Distributed Systems, pp. 355 – 360, 2002.

[3] L. Abeni, G. Buttazzo, "Integrating multimedia applications in hard realtime systems", in Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, 1998, p. 4.

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

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

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

[7] J. Kramer and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management", in IEEE Trans. on Software Engineering, Vol. 16, Issue 11 (Nov. 1990), pp. 1293-1306.

[8] Y. Vandewoude, P. Ebraert, Y. Berbers and T. D'Hondt, "An alternative to Quiescence: Tranquility", in Proc. of the 22nd IEEE Int. Conf. on Software Maintenance, Washington, DC, (Sep. , 2006), pp. 73-82.

[9] J. Gonçalves, L. Ferreira, L. Pinho and G. Silva, "Handling Mobility on a QoS-Aware Service-based Framework for Mobile Systems", in Proc. of the 8th IEEE International Conference on Embedded and Ubiquitous Computing (EUC 2010), Hong Kong, December 2010, to be published.

[10] M. ALRahmawy, A. Wellings, "A model for real time mobility based on the RTSJ," in Proc. of the 5th international Workshop on Java Technologies For Real-Time and Embedded Systems (Vienna, Austria, Sep. 2007), vol. 231. ACM, New York, NY, pp. 155-164.

[11] B. K. Choi, S. Rho, R. Bettati, "Fast software component migration for applications survivability in distributed real-time systems," in Proc. of the 7th Object-Oriented Real-Time Distributed Computing, Vienna, Austria, May 2004, pp.269-276.

[12] E. Schneider, "A Middleware Approach for Dynamic Real-Time Software Reconfiguration on Distributed Embedded Systems", PhD Thesis, Université Louis Pasteur – Strasbourg, 2004.

[13] Nogueira, L., Pinho, L., "A Capacity Sharing and Stealing Strategy for Open Real-time Systems", Published in Journal of Systems Architecture, Volume 56, Issues 4-6, April-June 2010, pp. 163-179.

[14] P. Pedreiras, P. Gai, L. Almeida, G. Buttazzo, "FTT-ethernet: A flexible real-time communication protocol that supports dynamic QoS management on ethernet-based systems", IEEE Transactions on Industrial Informatics, vol. 1, no. 3, p. 162-172, August 2005.

[15] 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.

[16] Maia, C., Nogueira, 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.