Scientia
Interdisciplinary Studies in Computer Science
20(2): 107-118, July/December 2009
 – doi: 10.4013/sct.2009.20.2.04

Continuum software infrastructure for ubiquitous computing: A service-based approach

Cristiano Costa[1], Felipe Kellermann[1], Rodolfo Antunes[1], Jorge Barbosa[1], Adenauer Yamin[2], Cláudio Geyer[3]

[1]PIPCA, Universidade do Vale do Rio dos Sinos
Av. Unisinos 950, São Leopoldo, 93022-000, Brasil

[2]PPGINF, Universidade Católica de Pelotas
Rua Félix da Cunha 412, Pelotas, 96010-000, Brasil

[3]PPGC, Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves 9500, Porto Alegre, 91501-970, Brasil

cac@unisinos.br, {felipek, rsantunes}@gmail.com, jbarbosa@unisinos.br,
adenauer@ucpel.br, geyer@inf.ufrgs.br

Abstract
The latest technological advances, which introduced innovative and more affordable devices, have contributed to boost the practical application of research in the field of ubiquitous computing (ubicomp). For the development of applications in this area, we need an adequate software infrastructure. In order to do so, we have proposed Continuum, an infrastructure based on service-oriented architecture (SOA), making use of framework and middleware, and employing a redefinition of follow-me semantics. In this redefined vision, users can go anywhere carrying the data and application they want, which they can use in a seamlessly integrated fashion with the real world. In this article, we focus on the description of the service-based architecture proposed for Continuum. The proposal widens the web services standards to support the mobility of services, allowing them to be deployed, copied, or moved. Besides, the abstraction provided enables the adaptation of legacy applications as Continuum pluggable services. We conduct some experimental analysis, using case study methodology. Based on these assessments, we present lessons learned and draw the conclusion of our work.

KEY WORDS: software infrastructure, middleware, web services, service oriented architecture, ubiquitous computing.

Resumo
Infraestrutura de software Continuum para a computação ubíqua: uma abordagem baseada em serviço. Os mais recentes avanços tecnológicos, com a introdução de dispositivos inovadores e mais baratos, contribuem para o aumento da aplicação prática das pesquisas na área de computação ubíqua (ubicomp). Para o desenvolvimento de aplicativos nessa área, é necessária uma infraestrutura de software adequada. Para atingir esse objetivo, esse artigo propõe o Continuum, uma infraestrutura baseada na arquitetura orientada a serviços (SOA), fazendo uso de framework e middleware. Além disso, a arquitetura emprega uma visão redefinida da semântica siga-me, na qual usuários podem ir aonde quiserem carregando os dados e aplicativos desejados, utilizando-os de forma integrada com o mundo real. Nesse artigo, é dado foco para a descrição da arquitetura orientada a serviços proposta para o Continuum. A proposta amplia os padrões de serviços web para suportar a mobilidade de serviços, permitindo que eles sejam instalados, copiados ou movidos. Adicionalmente, a abstração fornecida permite a adaptação de aplicações legadas como serviços plugáveis do Continuum. Algumas análises experimentais foram conduzidas, usando a metodologia de estudo de caso. Baseada nessas avaliações, algumas lições que foram aprendidas são apresentadas e algumas conclusões do trabalho são definidas.

PALAVRAS-CHAVE: infraestrutura de software, middleware, serviços web, arquitetura orientada a serviços, computação ubíqua.

1 Introduction

Recent technological advances, along with the increasing availability of many mobile devices, highlight three main characteristics that we believe should continue to widen and consolidate those changes in the next years: (i) the emergence of new companies and products in the mobile computing market, providing new devices, technologies, and paradigms (for instance, the iPhone); (ii) the many forms of communication provided by these products, as well as the availability of ubiquitous connectivity; (iii) the growing focus on the development of applications for these new devices, as for example those offered by App Store (Apple) and Android Market (Google), and on application infrastructures, such as App Engine (Google). Together, these characteristics result in an environment where devices, with different software and architectures, are constantly connected and, hence, should benefit from the integration between the heterogeneous applications (or components) they use.

This is the beginning of the ubiquitous computing (ubicomp) reality, which allows people to use computational resources in a transparent manner, integrated with the environment (Weiser, 1991). In this vision, people access their data and applications wherever they go and however they move. To consolidate the ubicomp area, many challenges should still be addressed (Costa *et al*., 2008). The use of software infrastructures, with middleware and frameworks, has been advocated as the approach to deal with many different ubicomp-related issues (Costa *et al*., 2008; Modahl *et al*., 2006). To accomplish this goal, our work focuses on the proposal of a service-based software infrastructure for ubiquitous computing, named Continuum.

Continuum proposes a redefinition of the original follow-me semantics concept, which states that applications and data go along with users, providing a virtual environment and adapting to the current context (Augustin *et al*., 2004). In our redefined vision, users can go anywhere carrying the data and application they want, which they can use in a seamlessly integrated fashion with the real world. This notion differs from the original one in two aspects: first, there is no idea of virtual user environment but rather the idea of using the actual environment. Secondly, the user's session is not sustained for all applications and data; instead, we propose that users choose which applications and data they want to carry with them. We believe that with this new approach, we break with the idea of replicating the user desktop session in every scenario and increase the applicability of the solution to more general situations.

The Continuum software infrastructure makes use of pluggable services, built according to web services open standards (Papazoglou, 2008). To provide this support, we have developed a distributed service architecture, named CoDSA (Continuum Distributed Service Architecture), offering the possibility of deploying, migrating, or replicating services. We have also defined an abstraction of a container for applications (CoApp) wrapping all the resources, dependencies, and executable codes needed.

The focus of this article is on describing the distributed architecture model of Continuum, detailing the CoDSA and the CoApp container. We also show some services developed in Continuum to support this architecture, namely Executor and Service Manager. Continuum is a broader proposal not limited by these features. Thus, other fronts in the project are also under development, such as the context awareness subsystem, the formal representation of context using an ontology, and the communication services.

The article is organized as follows. In Section, 2 we make a brief analysis of related works. Section 3 describes the Continuum software infrastructure, showing the main layers and components. The distributed service architecture of Continuum is then described at Section 4. The next section details the modeling of this architecture, highlighting the way in which applications or components are represented and managed as services. In Section 6, we present the implementation phase and analysis of results. Finally, in Section 7, we show the conclusion and suggest some future works.

2 Related work

Some previous works propose the replication of web services. In Juszczyk *et al*. (2006), the authors present an approach for replicating and synchronizing services to be used with ad-hoc networks, although the proposal is generic enough to be employed in other areas. Their article describes a method called hot deployment. It allows the installation of services at running servers without the need of interrupting the execution (Juszczyk *et al*., 2006). The focus of their solution is on dependability, and more specifically, on high availability. Another approach, with the same aim, is shown in Moser *et al*. (2006). In the article, some alternatives for replicating web services are presented, but not actually implemented. Besides, the article specifically considers the J2EE Application Server and the Tomcat container.

Web services migration has been also tackled before. In Hao *et al*. (2006), the authors suggest a way of dynamically migrating web services to overcome performance problems in real-time applications. Moreover, they offer a decision maker to determine when the migration should occur. Each service that can migrate from one node to another is called a weblet. The authors have also created an engine to support the migration and execution of web services (Hao *et al*., 2006).

Although there are some projects that deal with replication and migration of web services, as far as we

know, none of these works is in the context of ubicomp. Nevertheless, the use of web services, and more generally of Service Oriented Architecture (SOA), is common in this area. Here we highlight two recent articles.

In the Mobile SOA project, the researchers propose a web services extension to be used in lightweight mobile devices (Tergujeff *et al*., 2007). Their approach uses the J2ME web services specification in an MIDP platform. The project is at an initial phase, and still presents many additional challenges (Tergujeff *et al*., 2007).

COCOA (Mokhtar *et al*., 2007), on the other hand, is a more robust proposal applying web services in the area of ubicomp. The project allows the dynamic integration of available services to the execution of user tasks. Furthermore, the authors consider QoS requirements in this process.

Web services are the most common implementation for SOA because they use XML for data and employ platform-neutral communications (Howerton, 2007). The idea of obtaining functionalities as network-delivered services corresponds to a model named Software as a Service (SaaS). In Anerousis and Mohindra (2006), the authors have defended the use of SaaS for ubicomp environments, and stated that the most significant challenges in this field are how to handle periodic disconnections and how to address differences in devices. They propose the use of adaptive services to solve the latter challenge and the caching of data on devices, enabling offline operations to tackle the former problem. This vision adheres to our proposition.

3 Continuum software infrastructure

Continuum is a service-based software infrastructure for ubiquitous computing, integrating framework and middleware, as defined in Bernstein (1996), and addressing many different challenges of ubicomp. Continuum software infrastructure is an evolution of ISAM project (Augustin *et al*., 2004) based partially on the requirements offered by a comprehensive architecture model, presented in Costa *et al*. (2008), and partially on context awareness considerations. This latter aspect is not the focus of this particular article, and it will be detailed in another one.

The *Continuum framework* deals with design time abstractions needed for the implementation of ubiquitous software. It is intended to help the development of ubiquitous applications using middleware services. Moreover, the framework simplifies the use of the underling middleware services. There are three elements that constitute the framework: Application Programming Interfaces (APIs), User Interfaces (UIs), and tools. The APIs specializes the interface and simplifies the use of services offered by the middleware, providing some additional private services. UIs, on the other hand, provide a look and feel adapted to the platform being used for ubiquitous application design. Finally, the tools represent a set of generic applications to simplify the use of the framework. For instance, Execution Profiler is a tool to assist in the parameterization and deployment of services in the infrastructure.

The *Continuum middleware* hides environment complexity, isolating applications from explicit management of protocols, distributed memory access, data replication, communication faults etc. It also minimizes heterogeneity problems related to architectures, operating systems, network technologies, and even programming languages, promoting the interoperation of them. The middleware offers a set of pluggable services, which as the name implies can be loaded and used on demand, supplying the main functionalities during execution. These services are organized in subsystems, which are not an element in itself, but rather a group of related services.

Figure 1 illustrates the proposed development process using Continuum. The left side of the figure shows

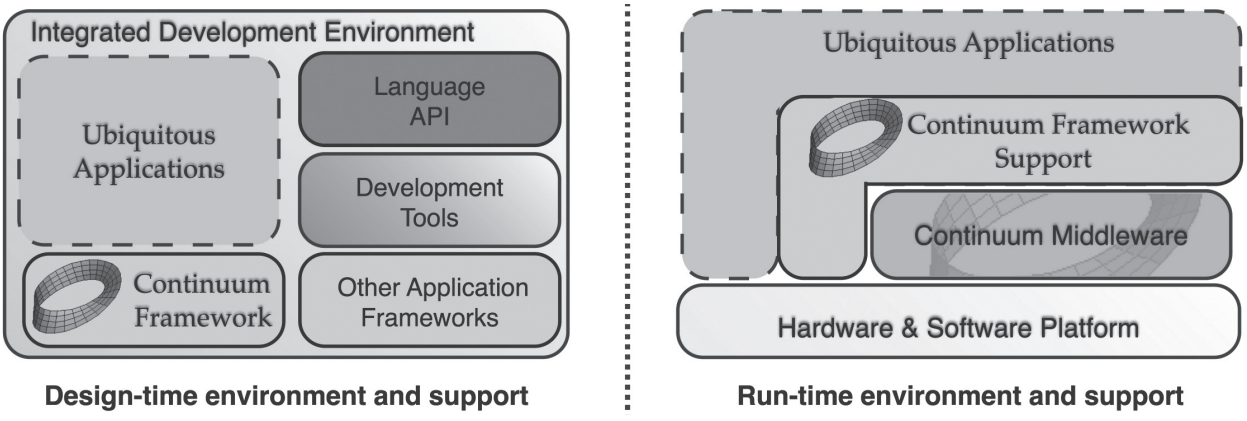

Figure 1. Continuum development process.

the environment and the support during design time. It comprises an Integrated Development Environment (IDE) for the implementation of ubiquitous applications. The IDE encompasses a language API, a set of development tools (compiler, editor, linker, debugger etc.), the Continuum framework, and other application frameworks as needed. In this environment, we can build the ubiquitous application source code. The right side of the figure presents, in general terms, the components required during execution: the application binaries, the Continuum framework and middleware, and the platform necessary to execution (network, computer, operating system and additional running support).

We identify this runtime environment as Continuum software architecture. The proposition for Continuum software architecture is presented in Figure 2. The architecture is divided in layers: foundation, middleware (subsystems and pluggable services), and user space. The foundation comprises the execution and support environment, including the network, the operating system, and the language runtime support. For instance, if the application is developed in Java, this language support includes the Java Virtual Machine (JVM). The middleware is divided in subsystems, which are further divided into services. The pluggable services constitute the core of Continuum middleware, providing support for the execution of ubiquitous applications. Finally, the user space layer contains user applications and the Continuum framework support. Applications can use the foundation layer directly and also interact with the middleware.

Continuum services are based on SOA, and were planned as web services. Because of this characteristic, Continuum architecture inherit all the advantages of the SOA model, such as the enhanced interoperability among heterogeneous environments, the decoupling of the architecture from the hardware and low-level software infrastructure, and the independence from any type of proprietary technology, device or manufacturer.

The services in the Distributed Execution subsystem are responsible for the distributed processing support and communication in Continuum. In this component, applications are managed, services are deployed on demand, and then copied or migrated among nodes. Furthermore, this subsystem keeps the physical organization of the environment, by storing attributes related to the management of the infrastructure, i.e. resources, users, and applications.

The *Context Awareness* subsystem groups the services that deal with a variety of contextual information. It provides a formal representation for context, in an independent application manner. The subsystem also considers user preferences (requirements that vary from user to user and over time). The Context Awareness subsystem is also in charge of storing context, along with points in time at which these data have been created, and distributing/localizing them.

Another subsystem is *Adaptation Management*. Not only does it target at the adaptation process itself, but also at the management of the adaptation process, which includes agility aspects and the maintenance of system stability (Silva *et al*., 2008). On one side, we have to address the delay between the perception of a new context state and the execution of actions to adapt the system to this new environment condition; this process demands agility. On the other hand, the execution of adaptation actions has

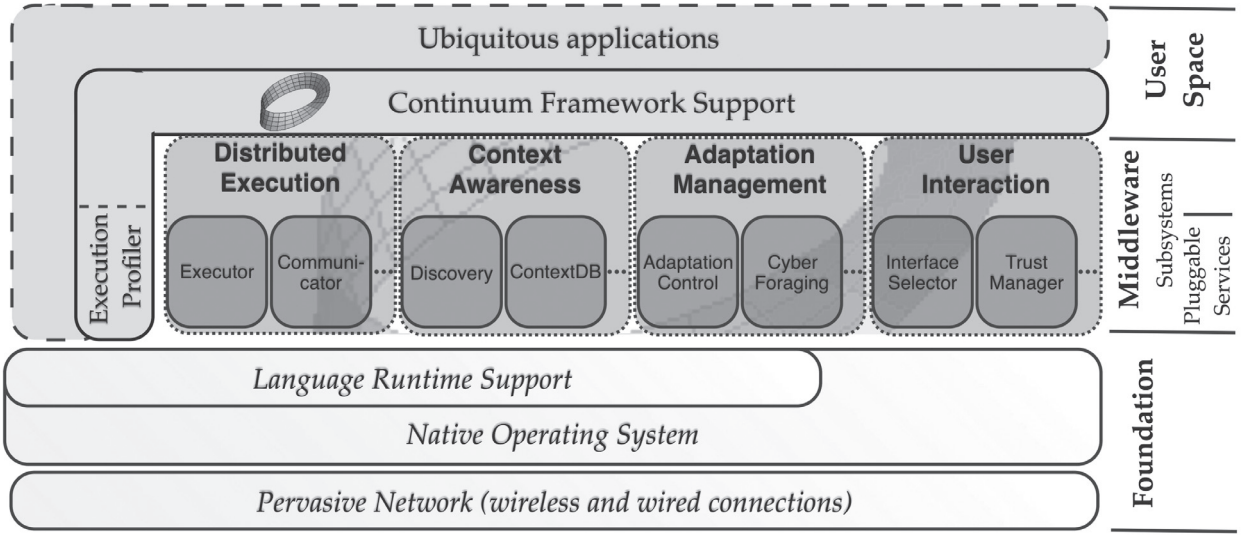

Figure 2. Continuum software architecture.

a computational cost and competes with the application itself. In an extreme case, adaptation actions can be very frequent, leading the system to a state of instability, in which the majority of resources is consumed by the execution of these adaptation actions. This requires stability maintenance in the environment.

Finally, we have the *User Interaction* subsystem. Services in the subsystem are in charge of reinforcing invisibility issues, giving special consideration to user attention and intent. The main features of this subsystem are to provide ubiquitous access to files (Frainer *et al*., 2007), to deal with trust and privacy, to supply the choice of an interface, and to help with invisibility issues, more specifically to ensure user attention, to meet user intent, and to cause minimal user intervention. In the latter functionality, interfaces suitable to each type of device or environment could be selected. To accomplish this, during design time we can define abstract user interfaces and predict different types of interaction, with the aid of the framework, so that the decision of which interface to use can be postponed to execution-time.

As already pointed out, these subsystems are only a conceptual organization; in practice, Continuum uses a service-based organization, which selects services on demand, depending on what functionalities the applications need. The framework will provide an interface that helps the selection of these services. These pluggable services add an adaptive behavior, which is important due to the high heterogeneity of the many different resources. In addition, Continuum proposes the use of Service-Oriented Computing - SOC (Papazoglou and Georgakopoulos, 2003). In SOC, the service layer follows the service-oriented architecture (SOA). The purpose of SOA is to support critical applications, which require the management and deployment of services and applications; it is also targeted at providing support for open services (Papazoglou and Georgakopoulos, 2003). The application of SOC on the web is obtained by the use of web services. SOC, SOA, and web services create a general interface, which makes interaction easier in Continuum; in a more ad hoc approach, those elements enable many applications to make effortless use of its services.

Besides being selected on demand, the services are context adaptive, i.e., the infrastructure is able to use the implementation that is better tuned to each device. Furthermore, we reduce resource consumption by selecting only services that are actually necessary. Such scheme is possible because services are defined by their semantics and interface, instead of a specific implementation. Moreover, it is easy to add other services, since we make use of SOA architecture. In the next section we describe the architecture that allows the support of these distributed services.

4 CoDSA: Distributed Service Architecture

This section describes the Continuum Distributed Service Architecture (CoDSA), which is a SOA that uses web services for communication. CoDSA manages the pluggable services in Continuum software infrastructure.

Each pluggable service in Continuum is defined as a web service. These services are reachable in the infrastructure from a node called *CoDirectory*. For scalability purposes, the physical resources, hereafter referred simply as nodes, are organized in a *cell topology*, loosely based on our previous ISAM pervasive environment (Augustin *et al*., 2004). Each cell, named *CoCell* in the infrastructure, has an associated CoDirectory and represents a (physical or abstract) place. The degree of abstraction of a cell can vary according to the application being developed. A CoCell could encompass other CoCells, benefiting from composition.

Each cell has at least one CoDirectory, which helps in finding the available services. To avoid bottleneck issues, it is possible to have more than one CoDirectory in each cell. Furthermore, it is possible to have CoCells without a CoDirectory physically present in it. In this particular situation, it is available in the next outer cell in the hierarchy that services this cell, i.e. it has a CoDirectory node. Additionally to CoDirectory, the CoDSA specifies other types of nodes in the infrastructure: *CoProvider* and *CoConsumer*. The former represents a node in Continuum that offers services, called *CoServices*, while the latter is the name given to each node that uses a CoService in the infrastructure. Figure 3 illustrates the CoDSA, with the entities described above, and also presents two possibilities in the architecture: CoService migration and replication.

Migration allows a CoService to change its location from a CoProvider to another. This occurs in the scope of a CoCell, and it is used to improve the system performance, reducing communication costs and delays. Currently, web services architecture does not support this feature. A related concern is how to decide when a CoService should migrate and to which location. This decision must consider hysteresis and the costs involved. In our current proposal, these aspects are not tackled, but intended as the subject of future work.

Another option in CoDSA is replication. CoServices can be replicated among CoProviders in the scope of a CoCell. This can improve system reliability of CoServices. Whenever a node changes its location or disappears, which is common in mobile environments, the CoService it provides becomes unavailable. If there is a replica, it can be discovered from the CoDirectory. Besides this discovery and registration feature, we also need a mechanism for the replication and synchronization

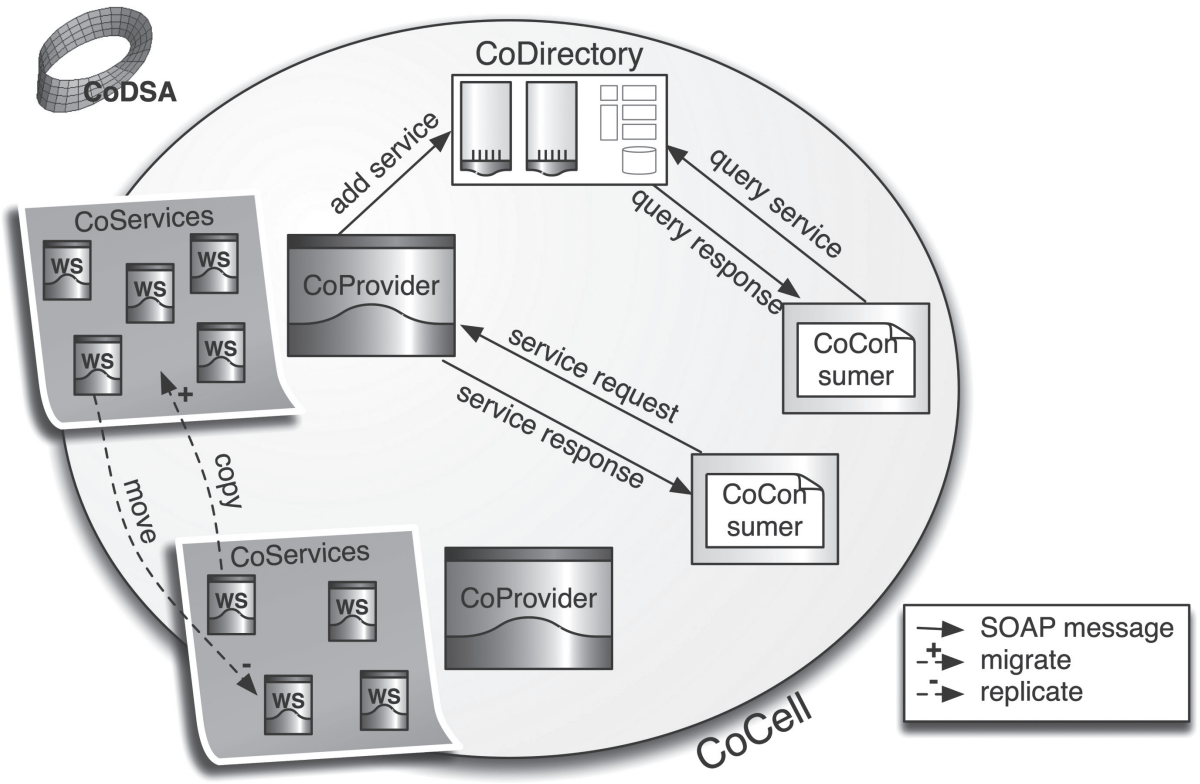

Figure 3. Continuum distributed service architecture.

of web services. The two main problems with replication are the synchronization of CoService copies and the decision whether a replica of a CoService should be made. Moreover, communication costs of the dynamic copy must be considered. Similarly to what happens during migration, it is the Distributed Execution subsystem that is in charge of the main operation, as well as of dealing with the other problems pointed out here.

CoDSA allows the dynamic selection of CoServices, according to the functionalities needed for each node of the system. We can obtain an adaptive behavior in Continuum by replacing or reconfiguring the CoServices that a CoConsumer employs. During software design, the Execution Profiler provides support for the selection of services needed by each node in the system. During execution, it is the Adaptation Management that takes this decision.

The interaction among nodes in the CoDSA, as illustrated in Figure 3, uses SOAP messages. SOAP[1], an acronym for Simple Object Access Protocol, is a lightweight communication protocol based on XML that allows the accessing of web services. The protocol is platform and language independent. As the name implies, it is very simple and also extensible. SOAP enables asynchronous client-sever communications and can make use of a wide range of protocols, including HTTP.

The next section presents the services modeled to support the CoDSA and the additional abstractions needed to represent and manage applications as services.

5 Architecture model and application support

The proposed model is divided in two main concepts, one related to the services necessary to the support of CoDSA and another related with an abstraction provided to the representation of applications in Continuum. Regarding the first concept, two services of the Distributed Execution subsystem are directly related to CoDSA.

Executor is a service that should be present in all Continuum nodes, called *CoNodes*. The aim of Executor is to offer a minimum support for the existence and execution of applications in Continuum infrastructure. For instance, common activities carried out by this service include deployment, initialization, and finalization of

[1]Specification available at http://www.w3.org/TR/soap12-part1/.

applications. The Executor acts as a thin layer between the foundation layer and the middleware, and consequently, other pluggable services.

The other pluggable service related to CoDSA is *Service Manager*. This service should be present in each CoDirectory and acts as a directory service. Additionally, this service includes the desired coordination characteristics in Continuum, i.e. migration and replication of services.

An abstraction for representing mobile and distributed services instantiation is necessary to support deployment, migration, and replication. We need this abstraction because the current web services standard, which is modeled using WSDL (Web Service Description Language), does not cover all the requirements of a pluggable service in Continuum. To draw a parallel, we consider the affirmation in Papazoglou (2008), which states that an XML scheme alone could not define a web service, requiring an additional standard, i.e. WSDL. The abstraction for representing applications in Continuum is called *CoApp*.

Conceptually, a CoApp is an application or a component in the form of a web service that is ready to be used in the Continuum infrastructure. It comprises the unit that can be installed, removed, updated or registered as a CoService, in which case it could also be replicated, migrated or synchronized. The outer interface of a CoApp is always a web service.

Technically, a CoApp consists of a data format that represents a group of related files compressed and archived, in which there is a sufficient amount of information to infer the type of application, different versions, and requirements, such as the execution runtime needed, the location of resources, and the external service interface. It is our aim that the CoApp format be as simple and as small as possible.

In Figure 4 we present the simplified conceptual vision of a CoApp. The figure presents the CoApp container, divided in three sections (CAC, CAR, and CAI). The lines at the lower part of the representation correspond to the services that the CoApp exports.

Three types of information are stored in the specific sections: metadata, resources, and implementation. The *CoApp Configuration* (CAC) section stores the metadata, such as versions, localizations, types of codifications, and references. This is a mandatory section in every CoApp, represented by an XML file and an XSD specification. Resources are referred in the *CoApp Resources* (CAR) section, which may include databases, images, internalization code etc. CAR is an optional section. The final section is named *CoApp Implementation* (CAI), which contains one or more implementation of the application. At the user's convenience, each implementation may have a different runtime.

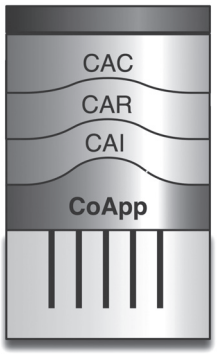

Figure 4. CoApp conceptual vision.

The transformation process of a preexisting application in a CoApp is straightforward and involves two steps. The first step is to describe the application, possibly with a WSDL interface, in a standard CoApp XML definition, containing the three sections described before. An XSD is also provided to validate the XML document created. This process is very simple, but we intend to automate it in the future. The second step is to create a single archive that has the binary codes, XML definitions, and resources, among others files. We have used the standard ZIP format in this process with *base64* codification.

A sample CoApp description is presented in Figure 5, an application named Calendar. In the CAC section, we define the elements *Name*, *Version*, *Author*, and *Description*. Additional elements are supported, such as *License* or *Copyright*. The element *Service* defines the service interface, using WSDL. The last element present in the CAC section of the sample CoApp is *Requirements*. Requirements of the "Runtime" type apply during execution. "Component", a generic type of requirement used for stating CoApp dependencies, is not shown in the example. In the CAR section, all the resources employed by the CoApp are listed. According to the runtime requirements fulfilled in the CAC section, the associated runtime in the CAI section is chosen. In the example, there are two possibilities in terms of runtime: Python and Lua.

```
<CoApp>
   <Name>com.continuumproject.Calendar</Name>                    CAC
   <Version>1.0</Version>                                        Configuration
   <Author>Continuum Project</Author>
   <Description>Calendar Provider</Description>
   <Service type="WSDL">
      <Path>Service/Calendar.wsdl</Path>
   </Service>
   <Requirements>
      <Require type="Runtime" name="Python"/>
      <Require type="Runtime" name="Lua" version="5.0"/>
   </Requirements>
   <Resources>                                                   CAR
      <Database type="sqlite">Calendar.sql</Database>            Resources
   <Resources>
   <Runtime Name="Python">                                       CAI
      <Path>Runtime/Python</Path>                                Implementation
   </Runtime>
   <Runtime Type="Lua">
      <Path>Runtime/Lua</Path>
   </Runtime>
</CoApp>
```

Figure 5. A sample CoApp description.

5.1 Applications in the infrastructure

The execution of applications in Continuum is accomplished by the use of the Executor service. Every CoNode should run this service. The aim of Executor is to support the management of applications in the infrastructure (the WSDL interface is presented in Figure 6).

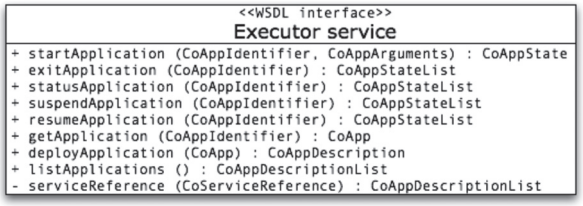

Figure 6. Pluggable service Executor.

For the execution of applications in Continuum, the first step is to use *deployApplication* from any node in the infrastructure. Then, the application is deployed and registered as belonging to Continuum. Eventually, this CoApp may be registered as a pluggable service using Service Manager. After this operation, we consider that a CoApp is a CoService. Before the registration, a CoApp is considered as a standalone application, which can use the services offered by the Executor (with the exception of the *serviceReference* private operation). To start the execution of a CoApp, we use *startApplication*. The complementary *exitApplication* may finalize either one or all instances of a CoApp.

To obtain information about a CoApp, such as application state, we use *statusApplication*. One way of changing the state of an application is using *suspendApplication* and *resumeApplication*, which, as the name implies, respectively, stop and continue its execution. A CoApp can be serialized using *getApplication*. Service Manager typically uses this operation to move or copy services. Another operation (*listApplications*) obtains the list of all applications in a certain node. Finally, *serviceReference* is a private method used during Service Manager coordination. This function does not have a straight purpose in the node itself.

5.2 Applications as services

To instantiate a CoApp as a Continuum pluggable service, we need to register it in the Service Manager. This service is present only in special nodes named CoDirectory, as already pointed out. Therefore, the first step is to find these nodes. We have employed Multicast-DNS (mDNS) as the discovery protocol (Giordano, 2005).

After discovering a Service Manager, we can use its functions (the WSDL interface is shown in Figure 7). There are only two methods that deal with CoApps: *registerService* and *updateService*. The former transforms a CoApp into a CoService, making it possible to remotely call its methods (along with copying and moving capabilities), while the latter only updates a previously registered CoApp, generating a new CoService version.

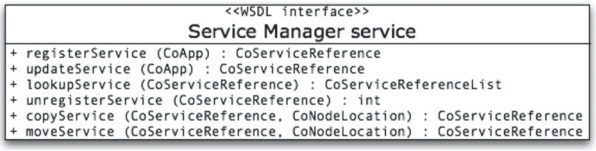

Figure 7. Pluggable service Service Manager.

To remove a CoService from Service Manager, we can employ *unregisterService*. Moreover, the *lookupService* method is used to search for a service, either using the service name (white pages) or descriptive semantics (yellow pages). The other two methods, *copyService* and *moveService*, are used to respectively copy and move services from one node to another. They are detailed in the next subsection.

5.3 Service Replication and Migration

Two fundamental characteristics of our proposal are the support of copying and migrating services. The Service Manager and the Executor coordinately handle these characteristics. In this subsection, we briefly detail the functional semantics of each operation.

Replication consists in copying one CoService from one CoNode to another in the Continuum infrastructure. The operations offered by this service continue to be accessible in the origin, even during the copying process, in addition to the new availability at the destination. In the developed model, this operation involves some steps. First of all, a CoConsumer asks for replication calling *copyService*. This CoConsumer can be any node, probably (but not necessarily) the node that contains the service being replicated. After that, Service Manager verifies if it has the CoApp content; if not, the service calls the Executor's *getApplication* method in the origin node, thus obtaining the desired content. The next step is to issue *deployApplication* in the Executor of the destination CoNode, sending the CoApp content to it. Subsequently, when the destination node accepts the CoApp, the Service Manager must be updated. This is accomplished by calling *registerService* in the CoDirectory that is associated with the destination CoCell. When a node searches for a specific service, it receives a *CoNodeLocation*, which contains the reference of both CoProviders (origin and destination).

Migration is very similar to replication, with a few additional steps. The difference in this operation is that we move the service from origin to destination. During

the moving process, all requests are still fulfilled from the origin CoNode. After the conclusion of this operation, the service is no longer available from the origin and requests must be redirected to the destination node. If a node tries to access this service in its former location, it receives a message that indicates that the service is not present. As a consequence, it must call *lookupService* from the Service Manager to obtain the new address. To minimize the occurrence of this situation, whenever a service is migrated, the Service Manager notifies the new location to the CoConsumers that are currently using the moved service, sending its new *CoServiceReference*. To perform this notification, the Service Manager has to call the Executor's *serviceReference* method. Another additional step is the unregistering of the CoService in the CoDirectory associated with the origin CoCell, which is done by the *unregisterService* operation of the Service Manager.

Activities in web services tend to be coordinated, using standards such as BPEL (Business Process Execution Language), sometimes referred as web services orchestration in the literature (Papazoglou, 2008). This coordination reinforces the fact that services and components should be reutilized. As a consequence, the most complex operations proposed (replication and migration) are defined as low-level, in a coordinated approach. We believe that in this way, new features could be added to the model without any adaptation in the provided infrastructure.

6 Implementation and analysis of results

In this section, we present the developed prototype and assess the distributed service architecture proposed for the Continuum project. The method used for validation is based on experimental evaluation, i.e., we propose case studies to assess the basic ideas of the model. The idea behind this approach is to choose a single instance, also called an event or a case, in which an in-depth and over-time examination is prepared (Flyvbjerg, 2006). The cases should be selective, focusing on the main issues that are important to the subject being analyzed; besides, the choice of the case to be studied must increase the extent of what can be learned, in the time interval available for the completion of the work.

We decided to employ this methodology in order to abridge the time in obtaining results for our research. Furthermore, besides being considered a scientific method, case studies are deemed as acceptable, in terms of perception of the facts involving the object of study. They also fulfill the three main ideas of the qualitative method: describing, understanding, and explaining (Flyvbjerg, 2006).

In the case study, we discuss the proposition of CoDSA. To accomplish this goal, we have modeled and implemented some services proposed in the distributed execution subsystem, more specifically the Executor and the Service Manager. The services were first defined using WSDL. In doing so, heterogeneity has been assured and the language employed for the implementation was not significant for interoperability purposes. We chose Python as the language for implementing our services. Our choice was based on various characteristics of the language, such as: platform independence, open source license, and ease of programming. Perhaps the main motive to employ Python was the fact that it is considered one of the best languages for quick development and initial prototyping, since the code is usually shorter and faster to write, due to its high expressiveness and set of libraries.

The use of web services in Python is provided by various libraries, which support SOAP, WSDL, and other related protocols. Among those, we have chosen SOAPpy. Another library employed in our implementation was ElementTree. This toolkit helps the management of XML files in Python, providing a container object to store hierarchical data structures in memory.

As the communication protocol, we used the standard TCP/IP stack with HTTP in the transport layer. For discovery purposes, we employed the Multicast-DNS protocol. By using this protocol, it is possible to apply the standard Domain Name Service (DNS) management in small networks without the need of a DNS server. CoNodes utilized this protocol for finding their associated CoDirectory.

Using the developed prototype, we generated some cases. In this article, we present four of those cases: application deployment, register of a CoApp as a CoService, replication of a CoService, and migration of a CoService. All cases were executed in a real environment using commercial off-the-shelf (COTS) PCs and HP Ipaq PDAs.

In Figure 8 we present the deployment process of a CoApp in the infrastructure, containing one desktop and two mobile devices, highlighting four possible steps. The *deployApplication* method is employed, which generates the message *deployApplicationRequest* (step 1). This message contains a whole CoApp. As a return (step 2), the node produces a CoApp descriptor.

An alternative way of deployment is presented as the next steps of Figure 8. A CoNode calls the *getApplication* operation of the Executor service (step 3), using a *getApplicationRequest* message and passing a CoApp descriptor. The response is a *getApplicationResponse* message (step 4), which contains the application in its CoApp format. This received application is then installed in the local CoNode.

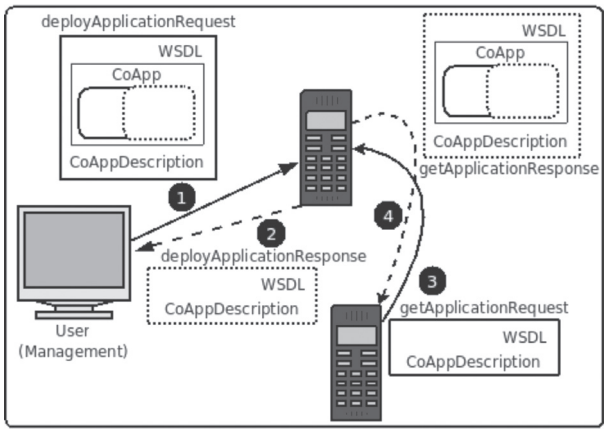

Figure 8. Deploying applications in Continuum.

The second case, illustrating the transformation of a CoApp into a CoService, is presented in Figure 9. It considers the same infrastructure employed in the former event. Whenever a CoApp is registered as a service, its interface is disclosed, so that other CoNodes can make use of its operations as a pluggable service. The first call is to the *registerService* operation of the Service Manager, which uses the *registerApplicationRequest* message (step 1), containing a CoApp. As a return (step 2), we obtain a *CoServiceReference* composite message. Not only does this message contain the functional and semantic description of a service, but also its physical location (represented by *CoNodeLocation*).

In Figure 9 we also present the process of finding a service, accomplished by calling the *lookupService* method of the Service Manager. A *lookupServiceRequest* message is generated (step 3), containing *CoServiceReference*. As a result, a *lookupServiceResponse* message is produced (step 4), which also contains the same *CoServiceReference*. The difference between both messages is that in the first one only the location of the service is relevant whereas in the second a list of references could be possibly obtained.

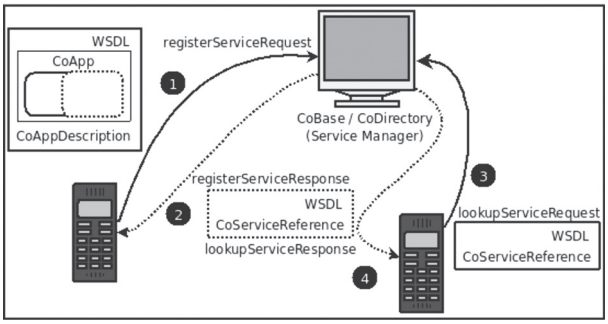

Figure 9. Applications becoming Services in Continuum.

Figure 10 presents the event of replicating a service. Differently from the previous cases, this diagram does not present the WSDL messages, but rather only some symbolic names. This is a simplification of the real process, which also involves additional steps. The process starts when a node calls *copyService* (step 1), passing a *CoServiceReference* (*myService* in the figure) and the *CoNodeLocation* to which the node will be copied (*destination* in the figure).

In the case study developed, the CoDirectory did not have a copy of the CoApp that implements the desired service (*myService*). Therefore, the CoApp must obtain the application from its origin using the *getApplication* method (steps 2 and 3). If the CoApp is available in the CoDirectory, these steps could be omitted. After that, the service must be deployed in the destination CoNode (step 4). The destination nodes accept the installation of the CoApp as a trust relationship has been previously created between this CoNode and the CoDirectory (the aspects of security and trust are out of the scope of the present article). The Node returns a *deployApplicationResponse* message (step 5), which contains the descriptor of the CoApp. An additional step, omitted in the diagram, is the registering of the new location in the CoDirectory itself. The final step (6) is the message returned to the CoApp that started the operation, containing the reference of the newly instantiated service.

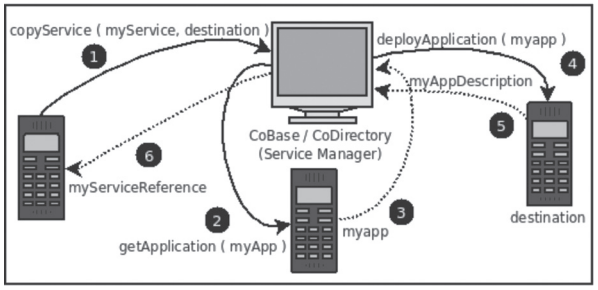

Figure 10. Service replication in Continuum.

Finally, in Figure 11 we show the last case, which illustrates the service migration. This case has also been simplified and has similar steps (from 1 to 6). The only difference is the called method, which is *moveService* instead of *copyService*. The additional steps are related to the movement of the service from origin to destination (step 7) and the update of a CoNode that is using the service being migrated (step 8). This last step consists of updating the references, in nodes that are using the service, with those of the new location, so that the subsequent accesses employ the new address (illustrated in step 9).

From the case studies developed and shown in this article, we have learned some lessons. The main advantage of the proposed model is that it does not create a completely new technology, but rather, while developing the distributed service architecture for

Continuum, we are extending the current web services standards adding a new data abstraction and introducing a set of service orchestrations. By doing so, not only are we inheriting technological aspects of the SOC, but we are also inheriting an existing knowledge, terminology, and understanding on the field. Another important point is that we are reusing existing applications with as little change as possible, making them available in the Continuum infrastructure.

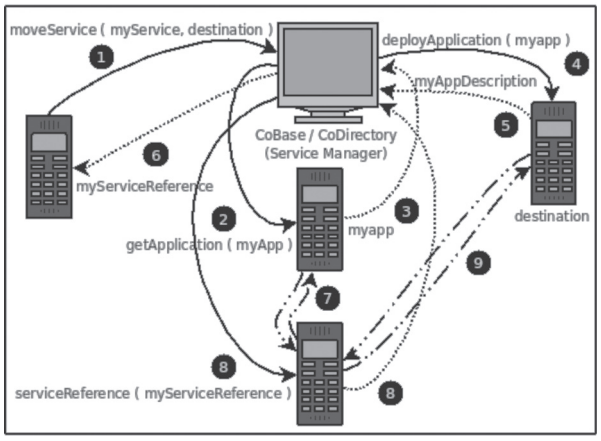

Figure 11. Service migration in Continuum.

In the cases developed we also detected some limitations. First, all applications should have a CoApp description. Currently this is a manual process, which introduces some overhead in porting legacy code. Second, the execution of existent applications is only offered for software that uses runtimes, such as those developed for virtual machines. Programs compiled to a specific platform, are currently not supported. We tested our prototype with applications implemented in Python and in Java. We do not see this as a very strong limitation, since the use of virtual machines has been defended as a solution to improve heterogeneity in ubicomp (Costa *et al*., 2008).

Another limitation concerning our current implementation is that we had limited the discovery and availability of services only to one CoDirectory. This occurs since we have not developed a mechanism for CoDirectory orchestration yet. Furthermore, the programmer is entirely responsible for all decisions regarding when and where to migrate (or to copy) a service. In the future, we can add a specific algorithm to help in this decision.

Although we observed some limitations, we believe that the use of SOA in ubicomp is a promising opportunity. It clearly fosters heterogeneity and integration among software implemented in different languages. Also, concerning our model, we believe that it helps the deployment and availability of code due to its pluggable feature. Another strength is related to the possibility of specifying alternative codes in the CoApp, possibly using distinctive runtimes. This feature adds a certain degree of adaptability, since code can be select according to the runtime available in the destination node.

7 Conclusion and future works

In this work, we have presented Continuum, a software infrastructure employing middleware and framework in ubicomp. In the detailing of Continuum, we described a distributed architecture for service support based on web services and SOA. Our proposal allows the deployment, copy, and migration of services.

We conducted some experimental evaluations to assess Continuum distributed service architecture. We proposed some experiments, based on the case study methodology. By doing so, we demonstrate that its implementation is possible, and also the main strengths and limitations that it could present.

As future work we plan the addition of exception mechanisms and fault handling to the model proposed. Furthermore, we intend to do a thoroughly investigation on communication overhead and also on strategies that help deciding when migrating or replicating a CoService. As previously stated, one of the future improvements is on supporting or automating the task of describing an existing application as a CoApp. Also, currently the proposed model is focused only on supporting services described by the WSDL standard, therefore making the existence of a WSDL description an implicit dependency for exposing an existing application as a Continuum pluggable service. The proposed model is also focused on the standard web services, using SOAP, though we are aware of increasing efforts toward creating less resource consuming approaches, such as the RESTful Web services (Pautasso *et al*., 2008).

We believe that the use of web services is an appropriate solution for ubicomp, as it improves the standardization of formats and protocols for describing services and their communication mechanisms. Moreover, we trust that the development of an architecture independent of runtime, platform or language incorporate advantages already obtained by SOA and web services standards. Among those advantages, we highlight the interoperability among heterogeneous environments, the decoupling of the architecture from the hardware and low-level software infrastructure, and the independence from any type of proprietary technology, device or manufacturer.

References

ANEROUSIS, N.; MOHINDRA, A. 2006. The software-as-a-service model for mobile and ubiquitous computing environments. *In*: INTERNATIONAL CONFERENCE ON MOBILE AND UBIQUITOUS SYSTEMS: NETWORKING & SERVICES, III, San Jose, 2006. *Proceedings*... New York, IEEE, p. 1-6.

AUGUSTIN, I.; YAMIN, A.; BARBOSA, J.; SILVA, L.; REAL, R.; FRAINER, G.; CAVALHEIRO, G.; GEYER, G. 2004. ISAM - Joining context-awareness and mobility to building pervasive applications *In:* M. ILYAS; I. MAHGOUB (eds.), *Mobile Computing Handbook*. Portland, CRC, p. 73-94.

BERNSTEIN, P. 1996. Middleware: A model for distributed system services. *Comm. of the ACM*, **39**(2):86-98.

COSTA, C.; YAMIN, A.; GEYER, C. 2008. Towards a general software infrastructure for ubiquitous computing. *IEEE Pervasive Comput*, **7**(1):64-73.

FLYVBJERG, B. 2006. Five misunderstandings about case-study. *Qualitative Inquiry,* **12**(2):219-245.

FRAINER, G.; SILVA, L.C.; GEYER, C.; AUGUSTIN, I.; YAMIN, A. 2007. Flexible application and context aware adaptation in a pervasive file system. *In*: INTERNATIONAL CONFERENCE ON SELF-ORGANIZATION AND AUTONOMOUS SYSTEMS IN COMPUTING AND COMMUNICATIONS. III, Leipzig, 2007. *Proceedings*... New York, IEEE, p. 118-121.

GIORDANO, M. 2006. DNS-based discovery system in service oriented programming. *In*: EUROPEAN GRID CONFERENCE, Amsterdan, 2005. *Proceedings*... Amsterdan, Springer, p. 840-850.

HAO, W.; GAO, T.; YEN, I.; CHEN, Y.; PAUL, R. 2006. An infrastructure for web services migration for real-time applications. *In*: INTERNATIONAL WORKSHOP ON SERVICE-ORIENTED SYSTEM ENGINEERING, II, Shanghai, 2006. *Proceedings*... New York, IEEE, p. 41-48.

HOWERTON, J. 2007. Service-oriented architecture and web 2.0. *IT Professional,* **9**(3):62-64.

JUSZCZYK, L.; LAZOWSKI, J.; DUSTDA, S. 2006. Web service discovery, replication, and synchronization in ad-hoc networks. *In*: INTERNATIONAL CONFERENCE ON AVAILABILITY, RELIABILITY AND SECURITY, I, Vienna, 2006. *Proceedings*... New York, IEEE, p. 847-854.

MODAHL, M.; AGARWALLA, B.; SAPONAS, T.; ABOWD, G.; RAMACHANDRAN, U. 2006. UbiqStack: A taxonomy for a ubiquitous computing software stack. *Pers. and Ubiquit. Comput*., **10**(1):21-27.

MOKHTAR, S.; GEORGANTAS, N.; ISSARNY, V. 2007. COCOA: Conversation-based service composition in pervasive computing environments with QoS support. *J. Syst. Softw.,* **80**(12):1941-1955.

MOSER, L.; MELLIAR-SMITH, P.; ZHAO, W. 2006. Making web services dependable. *In*: INTERNATIONAL CONFERENCE ON AVAILABILITY, RELIABILITY AND SECURITY, I, Vienna, 2006. *Proceedings*... New York, IEEE, p. 440-448.

PAPAZOGLOU, M. 2008. *Web Services: principles and technology.* Harlow, Pearson, 752 p.

PAPAZOGLOU, M.; GEORGAKOPOULOS, D. 2003. Introduction: Service-oriented computing. *Comm. of the ACM*, **46**(10):24-28.

PAUTASSO, C.; ZIMMERMANN, O.; LEYMANN, F. 2008. Restful web services vs. 'big' web services: Making the right architectural decision. *In*: INTERNATIONAL CONFERENCE ON THE WORLD WIDE WEB, XVII, Beijing, 2008. *Proceedings*... New York, ACM, p. 805-814.

SILVA, L.C.; COSTA, C.A.; GEYER, C.; AUGUSTIN, I.; YAMIN, A. 2008. On the control of adaptation in ubiquitous computing. *In*: ANNUAL ACM SYMPOSIUM ON APPLIED COMPUTING, XXIII, Fortaleza, 2008. *Proceedings*... New York, ACM, p. 2228-2229.

TERGUJEFF, R.; HAAJANEN, J.; LEPPANEN, J.; TOIVONEN, S. 2007. Mobile SOA: Service orientation on lightweight mobile devices. *In*: INTERNATIONAL CONFERENCE ON WEB SERVICES, V, Salt Lake City, 2007. *Proceedings*... New York, IEEE, p. 1224-1225.

WEISER , M. 1991. The computer for the twenty-first century. *Scientific American*, **265**(3):94-101.

Submitted on November 5, 2009.
Accepted on December 3, 2009.