Enabling WCET-based composition of service-based real-time applications

Iria Estévez–Ayres, Marisol García–Valls, Pablo Basanta–Val
Telematics Engineering Department
Universidad Carlos III de Madrid
Leganés, Madrid, Spain
{ayres, mvalls, pbasanta}@it.uc3m.es

Abstract

This paper presents an approach towards a framework that enables the composition of real–time applications from existing ubiquitous services. This framework allows to announce services (specifying their QoS requirements in terms of real–time parameters, such as Worst Case Execution Time, WCET), to discover services that perform a certain functionality, and to select the set of those discovered services that will be part of the application to be created. Such service set selection is based on the fulfilment of the QoS requirements of the application. The framework supports static composition, i.e., all services required to create an application have to be discovered before launching the whole application. Finally, it is described the composition model used in the framework.

1. Introduction

Ubiquitous Computing technology has introduced a great dynamism in application development and in its functionality. Instead of executing monolithic applications, this model of computation may also be used to create flexible applications dynamically from existing services, enhancing the reuse of code and decreasing the development time. In this case, the integration of the overall application will rely on the support of an underlying middleware; the middleware should guarantee that the specifications for such dynamic integration of services are met.

Specifications that the application developer needs to provide in some way are: (1) the characteristics of the desired application and (2) the different sets of services that are required to compose such application. However, to be acceptable, real–time applications (and specially hard real–time ones) must deliver a satisfactory Quality–of–Service (QoS)[7]. Then, the middleware must be aware of the QoS characteristics not only of the application itself, but also of the ubiquitous environment, the composing services, and the resources required by these services.

One of the most troublesome factors in providing QoS to real–time applications are [6]: *output timeliness requirements and distributed and concurrent computing requirements associated with the applications*. In distributed ubiquitous environments, this problem is emphasized due to the heterogeneous nature of the services to be run together. A test will have to be run previously to assure that it is possible to assign enough resources to a given set of services that execute simultaneously.

In the current work, we address the problem of achieving output timeliness in the execution of app–shared services by introducing a framework that supports appropriate composition of services. The framework enhances the capabilities of a real–time communication middleware. This way it allows to specify QoS requirements for services and applications in terms of: their Worst Case Execution Time (WCET) and the composition/selection of the appropriate service set to statically create an application from existing services in the environment.

The existence of real–time services on remote servers (acting as service proxies) introduces the necessity of using fault tolerance techniques in the middleware support. The usage of service proxies is completely out of the scope of this paper; we only consider that the appropriate versions of the services are downloaded in the client node. Therefore, no communication media is considered after service discovery and downloading.

A prototype of this framework has been implemented based on Java technology to show its feasibility. In the Java world, different technology contributions have been proposed to allow implementation, announcement, and discovery of services of interest. On one hand, the Java communications middleware, RMI[11], provides remote object communication, and it also has automatic code downloading capabilities. On the other hand, the Java distributed computing architecture, Jini[12], contains, among others, service discovery facilities. Also, there are a number of efforts also for low–level real-time support in Java technologies such as [2] and real–time RMI such as [13, 1]. They can be integrated to build real–time support into pervasive environments for composition of applications with QoS requirements, among them, timeliness.

Application composition using Java technology has been addressed by specifying service characteristics in XML technology[8]. However, such specification focuses only on functionality.

The prototype has been tested on centralised and on fully distributed environments using Jini on top of RMI. This prototype implements solely the composition of static applications with real–time capabilities. Dynamic applications, that would imply the replacement of services at run–time, are not considered.

The remaining of this paper is structured as follows: Section 2 briefly introduces the background of this work; Section 3 describes how to specify QoS for services and our task model for services and applications; Section 4 describes our proposed framework; Section 5 presents the whole process of composing an application; and, finally, Section 6 outlines the main conclusions.

2. Related Work

In the real–time world, component–based software development (CBSD) [5, 3] is an emerging development paradigm that enables the transition from monolithic to open and flexible systems. This allows systems to be assembled from a pre–defined set of components explicitly developed for multiple usages. However, none of these approaches can be applied to dynamic composition of real–time components (or more generally, applications) in a ubiquitous environment since they are focused on the design phase rather than on the execution phase.

In the field of QoS, the work described in[7] (for developing a generic middleware) shows that using an application component model, it is possible to provide end–to–end application QoS via QoS–aware middleware systems. But this proposed middleware is not suitable for composing real–time applications since the QoS specified does not account for the underlying hardware and the scheduling necessities of the whole system.

Integration of QoS characteristics into component technology has also been studied in [4]. However, these approaches aim at a rather static composition environment and enterprise applications based on components and not strictly on services.

3. Modelling services

In our framework, services are entities developed by service programmers that have real–time characteristics. Each service i has n methods each one $m_j()$ with an associated *weight* $wcet^i_j$. It is responsibility of the programmer to make the association of the *weight* with all the exposed methods for the different underlying hardware platforms.

We assume that there exists a simple additive rule as included in [9] to compute the WCET bound of sequences of instructions/statements (or methods) from the WCET information about single instructions.

When the framework needs a service i for composing an application, the framework downloads the appropriate version of the service for the specific hardware platform.

Applying the approach exposed in [10] for simple instructions and extending it for methods, we can define each method invoked by an application as an edge of a T–graph $G = (V, E)$. Each edge e^i_j will be an invocation to the method $m_j()$ of service i. So, each execution of an application will be a sequence of m invocations to methods of a single service or several services,

$$P_k = ((e^i_j)^k_1, (e^i_j)^k_2, \ldots, (e^i_j)^k_m)$$

As the execution time of a path $\tau(P_k)$ is the sum of the execution times of its edges

$$\tau(P_k) = \sum_{r=1}^{m} \tau((e^i_j)^k_r)$$

it is also possible to compute the maximum of these times, which is the worst case execution time of this path:

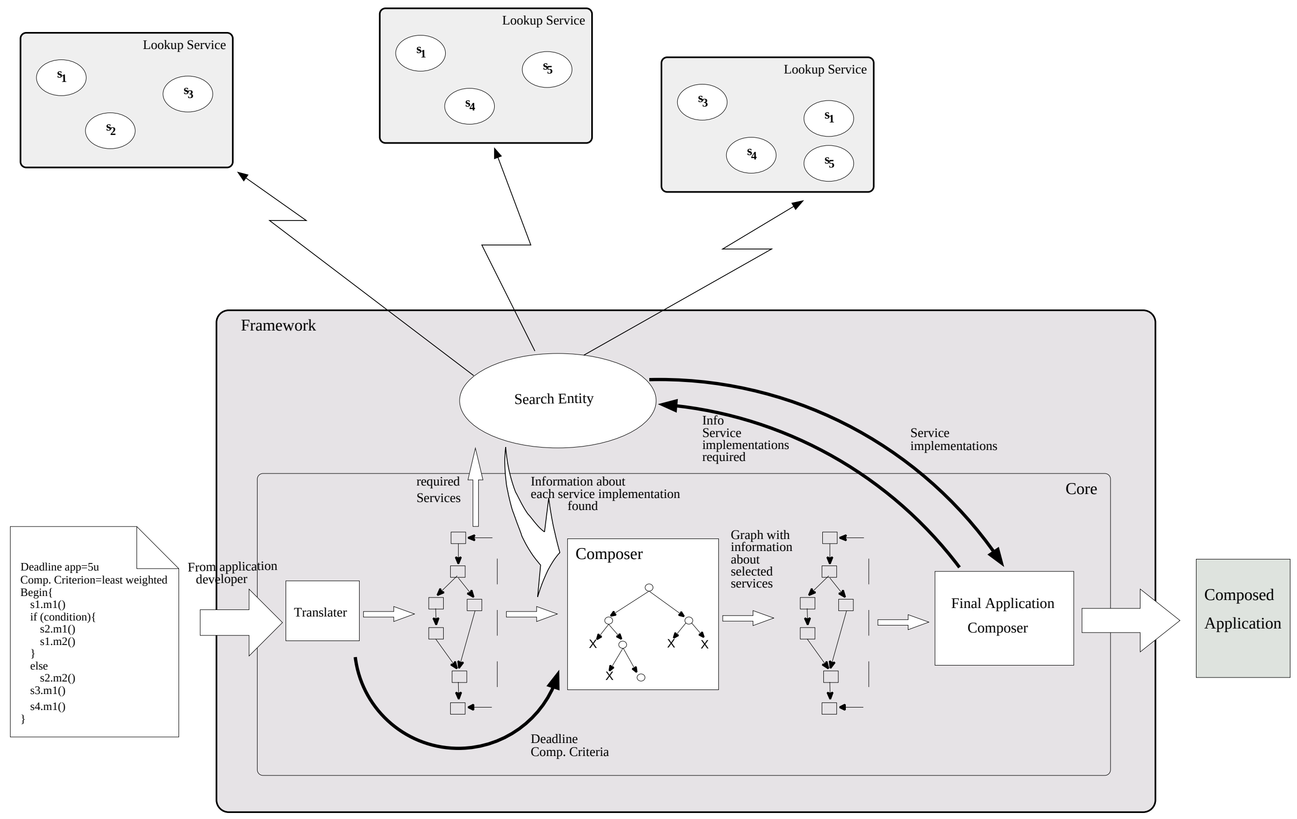
$$WCET_k = \sum_{r=1}^{m} (wcet^i_j)^k_r$$

4. Composition Framework

The framework proposed allows the custom composition of applications with QoS requirements based on ubiquitous services. QoS requirements are mainly concerned with the resource requirements and timeliness of applications. The framework allows to:

- Specify time requirements and, in general, QoS requirements of services,
- Announce services with QoS requirements,
- Discover services that match a specific functionality,
- Apply WCET–aware composing algorithms for selecting service sets that match the required functionality and QoS requirements.

In the framework, we can distinguish the following main entities (as showed in figure 1):



```
Deadline app=5u
Comp. Criterion=least weighted
Begin{
  s1.m1()
  if (condition){
    s2.m1()
    s1.m2()
  }
  else
    s2.m2()
  s3.m1()
  s4.m1()
}
```

Figure 1. Overview of the framework

- *Services*. These are ubiquitous entities that perform certain functionality. They are created by service programmers. When they are announced, our framework supports the statement of their functionality and QoS requirements.

- *Search entity*. This is the entity in charge of the configuration of the discovery options and of the discovery itself based on the functionality expressed in the application developer requirements.

- *Core control flow part*. It performs all the functionality described above. It translates the information given by the application developer into a graph–based data structure; with this information and the information obtained by the *search entity*, it selects the best set of services that will compose the application; and, finally, it orders the *search entity* to download the chosen services that will be part of the final application.

The framework provides the possibility to the application developer of selecting the composing criterion between a set of composition algorithms predefined.

The current prototype of this framework, implemented based on Java technology (Jini and RMI), consists of different computational and graphic tools that contain the functionality that has been presented in the previous sections.

The computational part implements the creation, announcement, discovery, and composition of the ubiquitous services. On the other hand, a graphic tool has been developed to interact with the computational part in a friendly way. Several experiments with service creation and announcement have been carried out.

5. Composition process

In the process of composing the final application, we can distinguish four different phases:

1. *Announcement Phase*, in which services are announced on lookup services of the network.
2. *Application Development Phase*, where the application programmer specifies an application description and its QoS requirements, as well as the desired composition criteria.
3. *Discovery Phase*, where all possible implementations of each needed service are found.
4. *Composition Phase*, where the framework choose the most suitable services to create the application, follow-

ing the instructions given by the application programmer.

Announcement Phase. The services implemented by service programmers are announced on lookup services of the network. Service discovery will be based on the functionality of the application that has to be created. Therefore, the discovery phase requires that services be announced with a tag stating their functionality.

Our proposed method of composition works with the WCET of the methods that will compose an application, so each service has to be announced with the information about all the exposed methods for different underlying hardware platforms.

Therefore the service will be announced with a tag stating its functionally and a set of parameters corresponding to its methods for different hardware platforms. So, each implementation of service i for a specific platform will contain the information showed in figure 2

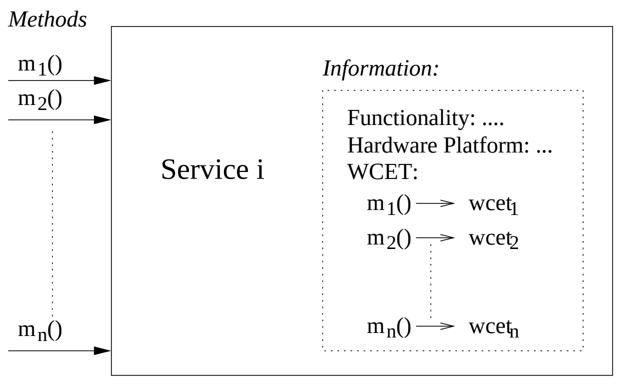
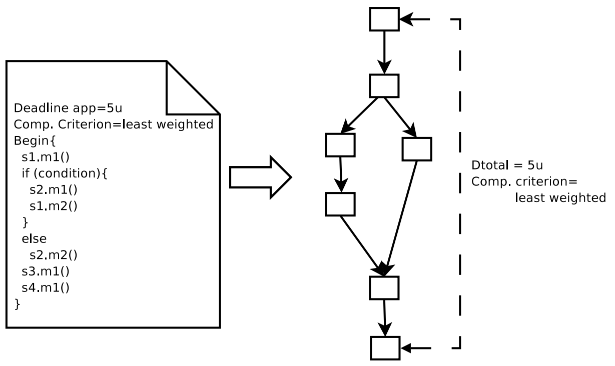


Figure 2. Information contained in service i

During the *Application Development Phase*, the application developer provides *QoS specification* of the desired application. She must specify an application description, i.e. the set of ordered services (or application components) that will compose the application, the methods invoked in each one and the deadline that has to be met by the whole application. Additionally, she can specify the composition criterion that has to be followed (for instance, to select the first set of services implementations that meets the requirements or to select the least weighted path attending to the WCET). Otherwise, the composition criterion will be to select the least weighted path. Our framework translates this information into a graph, without service intra–information as it can be seen in figure 3. This data structure will be used to store the information about the different implementations of the services that has to be managed in the composition phase.

Composition phase. The framework must select the most suitable services to create the application. For each application component there will be several implementations of the service with different parameters. It must download only the information (the previous set of parameters) about all the possible implementations and construct the whole application graph. To select the most suitable one, it has to explore the graph adding the WCET of each method until it reaches the total deadline of the application. In this case, this path of the graph will be discarded.



```
Deadline app=5u
Comp. Criterion=least weighted
Begin{
  s1.m1()
  if (condition){
    s2.m1()
    s1.m2()
  }
  else
    s2.m2()
  s3.m1()
  s4.m1()
}
```

Figure 3. Translating application information into a graph

An example of the composition phase is shown in figure 4. Each edge, e_j is a method invocation with a weight equal to the WCET of the method, $wcet_j$. In this phase, the framework finds out all the possible combinations of the found services (in the example, it found two implementations of each service, so the combinations will be (s_1^A, s_2^A), (s_1^A, s_2^B), (s_1^B, s_2^A) and (s_1^B, s_2^B)). And it constructs the graph as explained above. In the example, as the figure shows, only the tuple of services (s_1^A, s_2^B) meet the requirements of the application. In the example, we make the whole exploration in the graph because the composition criteria was to select the least weighted path, but it is also possible to specify that the criterion must be to select the first path that meets the requirements.

If a set of implementations meets the requirements, the *Search entity* will download them to compose the final application that will be offered to the *application developer*.

6. Conclusions and Future Work

This paper has presented a framework to compose static real–time applications from existing services in an ubiquitous environment. The approach presented here allows reusability of code and represents, as far as we know, the first approach to compose applications with real–time requirements in an ubiquitous environment, setting the basis of a framework that allows application composition. The next step in our research, is to extend the framework to

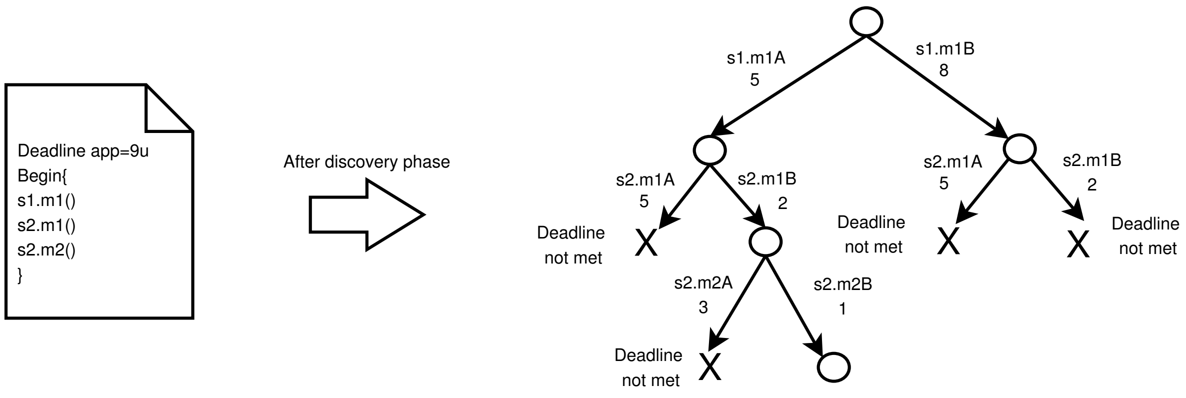


Figure 4. Applying the algorithm to select the most suitable services

be aware of the execution requirements, i.e. scheduling and resources such as memory, to allow dynamic composition of applications. To achieve this goal, we are enhancing the middleware support based on real–time–Java technology.

References

[1] P. Basanta-Val, M. García-Valls, and I. Estévez-Ayres. No heap remote objects: Leaving out garbage collection at the server side. In R. Meersman, Z. Tari, and A. Corsaro, editors, *OTM Workshops*, volume 3292 of *Lecture Notes in Computer Science*, pages 359–370. Springer, 2004.

[2] G. Bollela et al. *The Real-Time Specification for Java, version 1.1*, 2004. Avaliable on http://www.rtsj.org.

[3] I. Crnkovic and M. Larsson. A case study: Demands on Component–based Development. In *Proc. of 22nd Int. Conf. of Software Engineering, Limerick (Ireland)*, June 2000.

[4] M. A. de Miguel, J. Ruiz, and M. García-Valls. QoS–Aware Component Frameworks. In *Proc. of the International Workshop on Quality of Service*, May 2002.

[5] D. Isovic and C. . Norström. Components in Real–time Systems. In *Proc. of the 8th Conf. on Real–Time Computing Systems and Applications, Tokyo*, 2002.

[6] K. H. Kim. Toward QoS Certification of Real–Time Distributed Computing Systems. In *Proc. 7th IEEE International Symposium on High–Assurance Systems Engineering (HASE 2002), 23-25 October 2002, Tokyo, Japan*, pages 177–188. IEEE Computer Society, 2002.

[7] K. . Nahrstedt, D. Xu, D. Wichadakul, and B. Li. QoS–Aware Middleware for Ubiquitous and Heterogeneous Environments. *IEEE Communications Magazine*, 39(2):140–148, Nov. 2001.

[8] S. Paal, R. Kammüller, and B. Freisleben. Customizable Deployment, Composition, and Hosting of Distributed Java Applications. In *Proc. of the Fourth International Simposium on Distributed Objects and Application*, Newport Beach (California), May 2002.

[9] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real–Time Programs. *Real–Time Systems*, 1(2):159–176, Sep. 1989.

[10] P. Puschner and A. Schedl. Computing Maximum Task Execution Times – A Graph–Based Approach. *Real–Time Systems*, 13(1):67–91, 1997.

[11] Sun Microsystems. *Java RMI Remote Method Invocation.* Available on http://java.sun.com.

[12] Sun Microsystems. *Jini Specification, version 2.0.* Available on http://sun.com.

[13] A. Wellings, R. Clark, D. Jensen, and D. Wells. A Framework for Integrating the Real–Time Specification for Java and Java's Remote Method Invocation. In *Proc. Fifth IEEE International Symposium on Object Oriented RealTime Distributed Computing*, 2002.