A Qinna Experiment, a Component-Based QoS Architecture for Real-Time Systems

Jean-Charles Tournier
France Telecom R&D
Chemin du Vieux Chene, BP98
38243 Meylan, France
jeancharles.tournier@francetelecom.com

Jean-Philippe Babau
CITI/INSA Lyon
Bat. Leonard de Vinci
69621 Villeurbanne Cedex, France
jean-philippe.babau@insa-lyon.fr

Vincent Olive
France Telecom R&D
Chemin du Vieux Chne, BP98
38243 Meylan, France
vincent.olive@francetelecom.com

***Abstract*— Component-based software engineering (CBSE) is quickly becoming a mainstream approach to software development. At the same time, there is a massive shift from desktop applications to handheld systems: it is especially the case for multimedia applications such as video player, games, etc. Moreover, these applications have several Quality of Service (QoS) constraints which must be reached. A key issue of CBSE in embedded systems is its ability to integrate QoS management.**

In this paper, we present a component-based QoS architecture, called Qinna, and its implementation to a real-time case study. The Qinna architecture allows to integrate dynamic and heterogenous QoS management, and favours reusability thanks to its identified components.

I. Introduction

Nowadays, softwares become more and more complex with need for advanced functionnalities (multimedia, internet, communications in heterogenous world i.e. Bluetooth, GSM, GPRS, etc.) which can be added or removed dynamically. Moreover, such systems have numerous Quality of Service (QoS) constraints such as real-time, reliability, fault tolerance or security.

To implement this kind of systems component-based software engineering (CBSE) appears as a promising solution. One of the claims of CBSE is to build better software in less time [6]. CBSE may be one of the most efficient ways to reduce time development because it is intrinsically oriented to reuse existing parts, component in this case, of a system. Moreover, as a system architecture results from the assembly of components, CBSE increases maintainability of systems allowing to replace parts of systems when required [13].

Several component models are available or developing. Industrial models include the Microsoft® component family (COM, DCOM and more recently .NET [7]), the solution from SUN Microsystem® (JB, EJB [11]) or standardized model such as the proposal from OMG (CORBA Component Model [8] which is close to the EJB model). These models are designed for traditional workstation (such as PC) software and are mainly seen as *business models*. In the area of embedded systems, component models result mainly from the research domain. For instance, Think [3], PURE [1] or OSKit [4] allow to build entire operating systems as an assembly of components.

The main lack of this component models is that the functionnal point of view is well achieved whereas the QoS one is not. Some premilinary works such as [12], [9] or [10] tend to integrate QoS management to components, but heterogenity of the QoS, in terms of declaration, management or hardware support lead to produce heavy and complex models with a huge cost.

In this paper, we present a case study to show how QoS can be dynamically managed in component-based real-time systems. The case study is based on Qinna, a component-based QoS architecture compatible with the Fractal [2] component model.

The rest of the paper is organized as follow. The first part presents the Fractal component model, while the second part defines the Qinna architecture. Then the third part presents an implementation of Qinna for real-time systems. The fourth part concludes the paper.

II. Fractal

In this section we pesent the main concepts of the Fractal [2] component model.

A Fractal component is formed out of two parts: a controller and a content. The content of a component is composed of (a finite number of) other components, which are under the control of the controller of the enclosing component.

A component can interact with its environment through operations at identified access points, called interfaces. Operations provide the basic interaction primitives in the Fractal model. They can be either one-way or two-way. A one-way operation consists only in an operation invocation. A two-way operation consists in an operation invocation, followed by the return of a result. Operation invocations and operation returns carry arguments which can be names (esp. interface references), values or passivated forms of components.

Interfaces are either client interfaces or server interfaces. A server interface can receive operation invocations (and return

operation results of two-way operations). A client interface can emit operation invocations (and receive operation results of two-way operations).

Usually a Fractal component type is characterized by its interfaces, while a Fractal component class is characterized by its interfaces and its implementation.

Figure 1 represents a component X which provides an x interface and requires an y interface. In this example, interface y is provided by component Y. Component X can provide its

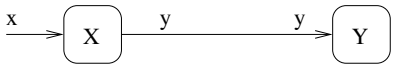

Fig. 1. Simple component configuration.

x interface only if its required y interface is filled: there is a contract between components X and Y. On a QoS point of view, the QoS level on the provided interface is a function of the QoS level seen on the required interfaces and of its own implementation.

Fractal is a model and has several implementations. Think [3] is an implementation applied to operating systems. Think provides several components in order to build operating system as an assembly of components.

III. QINNA

In this section we define the Qinna architecture. Qinna is made of four kind of component: *QoSComponent*, *QoSComponentBroker*, *QoSComponentManager* and *QoSDomain*.

A. QoSComponent

A QoSComponent is a component which provides, at least, one interface and may requires others. To provide a QoS level on its interface, a QoSComponent needs to control QoS on its required interfaces and must be configured via a *local constraint* typed by `T_CL`.

Additionally to its original interfaces, this kind of component must provides the **iLocalConstraint** interface. This interface provides two services:

- ***int set(T_CL tcl);***
- ***T_CL get();***

B. QoSComponentBroker

A QoSComponentBroker relies on a *global constraint* to accept, or not, to set the local constraints of QoSComponent and give a reference to it. QoSComponentBroker is responsible for QoS admission testing and QoSComponent reservation. To each class of QoSComponent is associated a QoSComponentBroker.

QoSComponentBroker requires the **iLocalConstaint** interface and provides the **iBroker** interface with the following services:

- ***QoSComponentId reserve(T_CL tcl);***
- ***int free(QoSComponentId cid);***
- ***int modify(QoSComponentId cid, T_CL tcl);***

It also provides the **iBrokerCfg** interface to configure the global constraint typed by `T_CG`. Services are:

- ***int set(T_CG cg);***
- ***T_CG get();***

C. QoSComponentManager

A QoSComponentManager is responsible for the QoS level provided by QoSComponents. Its goals is to first initialize, from a QoS point of view, the execution of a QoSComponent. From a specification which includes a required QoS level, the QoSComponentManager translates it to the needed QoS level on required interfaces and the needed local constraint on the related QoSComponent. This is responsibility of QoS mapping operation. The mapping is based on a map table, which is a property of the QoSComponentManager. Then the QoSComponentManager is in charge to follow the contract and to adapt it in case of context change. To each QoSComponentBroker is associated a QoSComponentManager.

A QoSComponentManager relies on a QoSComponentBroker to get a QoSComponent configured with the desired local constraint, and on other QoSComponentManager to get the QoSComponent which is able to provided the required interface with desired QoS level.

A QoSComponentManager requires the **iBroker** interface and provides two interfaces:

- **iQoSManager**
  - ***QoSComponentId reserve(T_QoS_U qu);*** asks for a reference of QoSComponent which is able to provided a level of QoS equal to `qu`. Returns `null` if the operation fails.
  - ***int free(QoSComponentId cid);***
  - ***int modify(QoSComponentId cid, T_QoS_U qu);***
- **iQoSAdapter**
  - ***int degrade(QoSComponentId cid);***
  - ***int upgrade(QoSComponentId cid);***

Moreover, QoSComponentManager provides the **iManagerCfg** interface to configure the map table:

- ***int set(T_MT mt);***
- ***T_MT get();***

D. QoSDomain

A QoSDomain is the highest level of the architecture. All other components are encapsulated in a QoSDomain component.

QoSDomain is responsible for the adaptation of QoS level. Adaptation is based on priorities specified by the user.

Moreover the QoS Domain component forms a boundary: each component is under the control of only one QoSDomain.

QoSDomain requires two kind of interfaces: **iQoSManager** and **iQoSAdapter**. It provides the **iQoSDomain** interface with the following services:

- ***QoSComponentId reserve(T_QoS_U qu, T_Prio prio);*** asks for a reference of a QoSComponent which is able to provide a level of QoS equal to `qu` and with a prioirity

equal to `prio`. Returns `null` if the operation fails. QoSDomain transmits it to the appropriate QoS Manager and stores the priority to be able to decide which QoS component to adapt.

- ***int free(ComponentIdenity cid);***
- ***int modify(ComponentIdenity cid, T_QoS_U qu, T_Prio prio);***

E. Type constraints

The Qinna architecture defines six different abstract types. Each type has its own constraints and are defined as follow. Syntax $c : T$ means that c is typed by T.

- **T_CL** has a comparison and sum operators:

$$\frac{c_1 : T\_CL, c_2 : T\_CL}{c_1 + c_2 : T\_CL}; \frac{c_1 : T\_CL, c_2 : T\_CL}{c_1 < c_2 : \{true, false\}}$$

  Moreover, `T_CL` has a `default` element.
- **T_CG** has a comparison and sum operator with `T_CL` type:

$$\frac{c_1 : T\_CG, c_2 : T\_CL}{c_1 + c_2 : T\_CG}; \frac{c_1 : T\_CG, c_2 : T\_CL}{c_1 < c_2 : \{true, false\}}$$

- **T_QoS_U** has a comparison operator:

$$\frac{c_1 : T\_QoS\_U, c_2 : T\_QoS\_U}{c_1 < c_2 : \{true, false\}}$$

  `T_QoS_U` has also a `default` element.
- **T_MT** is composed of one `T_CL`, one `T_CL` and one `T_QoS_U` types. Moreover, `T_MT` has a comparison operator:

$$\frac{c_1 : T\_MT, c_2 : T\_MT}{c_1 < c_2 : \{true, false\}}$$

  `T_MT` has a `default` element.
- **T_Prio** has a comparison operator:

$$\frac{c_1 : T\_Prio, c_2 : T\_Prio}{c_1 < c_2 : \{true, false\}}$$

  `T_Prio` has a `default` element.
- **QoSComponentId** has no constraint.

Qinna is a QoS architecture which obeys to the following principles: *(a)* it respects the separation of concerns between the functional and QoS views and *(b)* it is component-based. Therefore Qinna can be easily reused thanks to its identified components and takes into account the heterogeneous aspect of QoS management. In the following part, Qinna is applied to a real-time case study.

IV. Real-Time Case Study

The goal of this section is to show how Qinna may be implemented in real-time systems in order to dynamically manage QoS.

A. System Description

We consider a system with two components, `A` and `B`, which produce one interface, `a` and `b`. These components need thread components in order to produce their interfaces. Thread components are scheduled thanks to an EDF scheduler. Figure 2 represents components of the system.

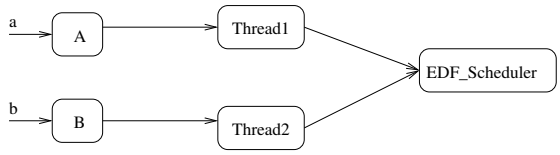

Fig. 2. Components representation.

The QoS level provided by component `A` and `B` depends on the thread attached to it. In the system, threads are characterized by an execution delay and a deadline. Tables I and II give the relation between required QoS levels on `A` and `B` components and thread characteristics. Moreover, we

A QoS Map Table	
QoS Level	Thread Charac.
GOOD	(4,10)
BAD	(4,20)

TABLE I
A MAPPING.

B QoS Map Table	
QoS Level	Thread Charac.
GOOD	(7,10)
AVERAGE	(7,15)
BAD	(7,20)

TABLE II
B MAPPING.

define two priority level: `HIGH` and `LOW`.

B. Qinna Integration

To make the system QoS aware, we need to integrate the Qinna architecture. First of all, we identify four component classes, where only three need QoS management. The QoSComponents are `A`, `B` and `Thread`.As described in section III, we associate one QoSComponentBroker and one QoSComponentManager to each class of QoSComponent. Moreover, one QoSDomain is added in order to encapsulate a real-time QoS domain.

The next step is to configure the QoSComponentBrokers, QoSComponentManagers and QoSDomain. The QoS mapping of `A` and `B` are given by tables I and II respectively. QoS mapping of threads are empty since they do not need QoS management on their required interfaces. Global constraint of threads QoSComponentBroker is set to 100%: it represents the maximum processor load. Local constraint of thread components is the pair $(Execution\_Delay, Deadline)$. Global constraint of `A` and `B` QoSComponentBrokers allows to give reference of only one `GOOD` or `BAD` QoSComponent. Local constraint of `A` and `B` QoSComponent represents their current QoS level, i-e `GOOD` or `BAD`. Finally, QoSDomain integrates the priority levels and knows that a `HIGH` request is more prioritary than a `LOW` one.

The overall system architecture is shown in figure 3.

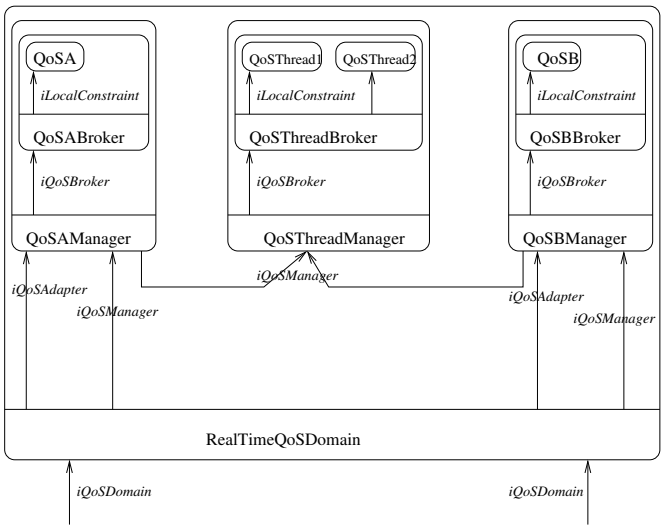

Fig. 3. Global system architecture.

C. Analysis

In this section, we analyze how the architecture works in terms of dynamicity, reusability, efficiency and heterogeneity for this case study.

- **Dynamicity.** Using Qinna architecture, the system is able to dynamically manage QoS. The QoSDomain is responsible for priority evaluation in order to ask for an adaption (thanks to iQoSAdapter interface), while the QoSComponentManager has the adaptation mechanism.
- **Reusability.** Qinna architecture is component-based and has well identified components. It helps to reuse part of the system or to modify it. For example, if the EDF scheduler is changed by a RM one, only the QoSThread-Broker must be changed. Another example is that if A component is reused in another system, QoSABroker and QoSAManager can be also reused.
- **Efficiency.** As Qinna is an architecture, its efficiency depends on its implementation. For example, to be more efficient, a period parameter can be added to thread characteristic in order to avoid repetitive thread reservation.
- **Heterogeneity.** Heterogeneity is taken into account in the Qinna architecture. For example, real-time and non realtime threads can be managed. If there is a non real-time request for a thread, that is to say with no specified deadline, the QoSThreadBroker will set the deadline using the `default` element of `T_CL`. In this case, `default` is equal to the greatest one of existing threads. The `default` element is dynamically updated.

V. Conclusion and future work

In this paper, we presented a generic component-based QoS architecture called Qinna. We demonstrate that Qinna can be used for real-time component-based system. Using the Qinna architecture, the system respects the separation of concerns between the functionnal and the QoS one; integrates heterogenous QoS; is easily reusable thanks to its identified components; and can dynamically managed QoS.

In future work, we will implement more complex and representative case studies such as real-time communicating systems. We will also integrate complex QoS management policies such as hierarchical CPU scheduler [5].

References

[1] Beuche and al. The PURE Family of Object-Oriented Operating Systems for deeply Embedded Systems. In IEEE Press, editor, *"Proc 2nd IEEE Int'l Symp. ObjectOriented Real-Time Distributed Computing"*, Piscataway, N.J., 1999.

[2] Bruneton, Coupaye, and Stefani. *Recursive and Dynamic Software Composition with sharing*. 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02), 2002.

[3] Fassino, Stefani, Lawall, and Muller. *Think: A Software for Component-based Operating System kernels*. Usenix Annual Technical Conference, 2002.

[4] Ford and al. The Flux OSKit: A Substrate for kernel and Language Research. In ACM Press, editor, *"Proc 16th ACM Symp. Operating Systems Principles"*, pages 38–51, New York, 1997.

[5] Goyal, Guo, and Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 107–121, 1996.

[6] Meyer and Mingis. *Component-Based Development: from Buzz to Spark*, volume 32. IEEE Computer, 1999.

[7] Microsoft. *.NET Framework*. http://www.microsoft.com/net/.

[8] OMG. *Corba Component Model, V3.0 - Specification*. http://www.omg.org/technology/documents/formal/components.htm, 2002.

[9] Le Sommer. *Contractualisation des ressources pour les composants logiciels: une approche rflexive*. PhD thesis, University of Bretagne Sud, France, 2003.

[10] Staehli and Eliassen. *QuA: a QoS-Aware Component Architecture*. Simula Research Laboratory, Research Report Simula, 2002.

[11] Sun Microsystems. *EJB*. http://java.sun.com/products/ejb/.

[12] Wang, Kircher, and Schmidt. *Towards a Reflective Middleware Framework for QOS-enabled CORBA Component Model Applications*. IEEE Distributed Systems Online special issue on Reflective Middleware, 2001.

[13] Xia, Lyu, Wong, and Fu. *Component-Based Software Engineering: Technologies, Quality Assurance Schemes and Risk Analysis Tools*. International Journal of Software Engineering and Knowledge Engineering, 2000.