An Evaluation of Qinna, a Component-Based QoS Architecture for Embedded Systems

Jean-Charles Tournier
France Télécom R&D Division
Chemin du Vieux Chêne
38243 Meylan, France
jeancharles.tournier@francetelecom.com

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

Vincent Olive
France Télécom R&D Division
Chemin du Vieux Chêne
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 demonstrate the feasibility of integrating QoS concepts to CBSE. The demonstration is based on Qinna, a component-based QoS architecture integrating the main QoS concepts. Moreover, Qinna respects separation of concerns and can be easily reused thanks to its identified components.

Keywords

Component Based Software Engineering, Quality of Service Architecture, Embedded Systems

1. INTRODUCTION

Nowadays, there is a massive shift from desktop applications to embedded systems: it is especially the case for PDAs, smart-phones, web-tabs, etc. Software for those systems becomes more and more complex with need for advanced functionalities (multimedia, Internet, communications in a heterogeneous 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 reliability, fault tolerance, real-time 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 [14]. 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 [22].

Several component models are available or developing. Industrial models include the Microsoft® component family (COM, DCOM and more recently .NET [15]), the solution from SUN Microsystem® (JB, EJB [19]) or standardized model such as the proposal from OMG (CORBA Component Model [16] 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 [10], PURE [4] or OSKit [12] allow to build entire operating systems as an assembly of components.

The main lack of these component models is that the functional point of view is well achieved, whereas the QoS one is not. Heterogeneity of the QoS in term of declaration, management or even hardware support tends to produce heavy and complex models with a huge cost.

In this paper, we demonstrate the feasibility of the integration of QoS concepts to CBSE. The demonstration is based on Qinna, a QoS architecture integrating main QoS concepts from [7] and compatible with the Fractal [6] component model.

The rest of the paper is organized as follows. We first present the Fractal component model and the main concepts of QoS architectures. We then define the Qinna architecture and present an implementation for H3800 iPaq® of a multimedia system using Qinna. We finally review some related work and conclude the paper.

2. SETTING THE CONTEXT

This work combines ideas from two clearly identified research areas. The first one is about about Component-Based Software Engineering (CBSE) and more precisely about the Fractal [6] component model, while the second one is about QoS architecture. In the two following paragraphs, we present and define the main concepts of each research area.

2.1 The Fractal 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).

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

### 2.2 QoS architecture

Systems may manage QoS in different ways. At one end, QoS requirements can be met statically during design and implementation by proper design and configuration choices (such as scheduling rules, network bandwidth allocation, etc.). This will give a well-defined behavior, but without any flexibility. At the other end is the dynamic approach that lets systems negotiate at run-time the restrictions of the QoS characteristics they need for their activities. This approach often involves an adaptive aspect by having monitors and corrective steps to be taken when the QoS level drops below a certain threshold. This approach is very flexible as QoS policies can be changed at run-time, but the behavior is not well defined and performance may be degraded if costly adaptation schemes are used.

To be able to dynamically manage QoS, a QoS architecture should integrate at least the following abilities [7]:

- **QoS specification.** QoS specification is concerned with capturing application-level quality of service requirements and management policy. QoS specification is declarative in nature; users specify what is required rather than how this is to be achieved by underlying QoS mechanisms. A QoS specification is composed of several aspects [3] [2]:
  - *Performance* which characterizes user’s performance requirements.
  - *Level of service* which specifies the degree of commitment required (e.g. deterministic [11], predictive [9], adaptive [8] and best effort ).
  - *QoS management policy* which allows to specify adaptation to the required performance.

- **QoS provision mechanism** is composed of the following aspects:
  - *QoS mapping* performs the function of automatic translation between representations of QoS at different system levels. For example a **GOOD** user QoS specification is translated to 25 frames/sec at application level and to 60% CPU at resource level.
  - *Admission testing* is responsible for comparing resource requirements arising from the requested QoS against available resources in the system
  - *Resource reservation.*

- **QoS management.** To maintain agreed levels of QoS it is often not sufficient just to commit resources. QoS management is frequently required to ensure that the contracted QoS is still valid. Fundamentals of management include:
  - *QoS monitoring* allows each level of the system to track the ongoing QoS levels achieved by the lower layer.
  - *QoS maintenance* compares the monitored quality of service against the expected performance and then exerts a tuning operation.
  - *QoS degradation* QoS degradation alerts indicate to the users when it determines that the lower layers have failed to maintain the QoS of the flows and nothing further can be done by the QoS maintenance mechanism.

In the following section Qinna is presented. Qinna is a QoS architecture suitable for component-based software engineering. The architecture integrates the previous requirements for QoS architecture and is based on Fractal components.

### 3. Qinna Architecture

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

#### 3.1 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**.

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();`

#### 3.2 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 admission testing and reservation. To each class of QoSComponent is associated a QoSComponentBroker.

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

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

3.3 QoSComponentManager

A QoSComponentManager is responsible for the QoS level provided on an interface of a QoSComponent. Its goals is to initialize, from a QoS point of view, the execution of a QoSComponent: from a specification which includes the desired the QoS level, the QoSComponentManager translates it to the needed QoS level on required interfaces and the needed local constraint on its managed QoSComponent.

The QoSComponentManager is responsible for QoS mapping. To each QoSComponentBroker is associated a QoSComponentManager.

A QoSComponentManager relies on a QoSBrokerComponent to get a QoSComponent configured with the desired interface with desired QoS level.

A QoSComponentManager requires the iBroker interface and provides two interfaces:

• iQoSManager
  – ComponentIdentity 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( ComponentIdentity cid);
  – int modify( ComponentIdentity cid, T_QoS_U qu);

• iQoSAdapter
  – int degrade( ComponentIdentity cid);
  – int upgrade( ComponentIdentity cid);

3.4 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:

• ComponentIdentity 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 priority 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( ComponentIdentity cid);
• int modify( ComponentIdentity cid, T_QoS_U qu, T_Prio prio);

3.5 Type constraints

The Qinna architecture uses four different 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_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_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.

• ComponentIdentity 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; (b) it takes the main concepts of QoS architecture described in [7]; and (c) 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 typical case study.

4. EXPERIMENT

The goal of this section is to show how Qinna may be implemented. The experiment aims for the construction of a multimedia system with QoS ability on a iPaq platform. For this experiment, we choose a H3800 iPaq® with an Intel® Strongarm SA1100 processor at 206 MHz, 60 Mo of DRAM memory and 32 Mo of flash memory. From a software point of view, we choose the Think framework [10] which is an implementation of the Fractal [6] component model applied to operating systems. The choice of the Think framework is motivated by the fact that it allows a fine-grain control of the system and is available for ARM processor.

4.1 Functional description

To simplify the experiment, we limit the system to two multimedia applications. The first one is a video player, while the second one is playing the Doom [1] game. These applications consume a lot of resources and need QoS management. Each application is driven by a graphical user interface and needs resources to be executed (i.e. CPU time and memory) as shown in figure 1. The QoS level provided by the video and Doom game is a function of the amount of allocated resources.

4.2 QoS description

In this system, we implemented a simple round-robin scheduler which is able to ponderate thread execution. For example, if thread A has a ponderation equal to 4 and thread B has a ponderation equal to 2, the scheduling is:

ABABAA|ABABAA|ABAB...
Figure 1: Multimedia system components representation.

To be realistic, the scheduler context switching is set to 10 MHz.

We then define a QoS mapping for each application as shown in tables 1 and 2. These mappings give the needed resources in function of the QoS level. The CPU column represents the thread ponderation while the memory one represents memory usage expressed in kilobytes.

<table>
<thead>
<tr>
<th>Video QoS Mapping</th>
<th>Doom QoS Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>QoS Level</td>
<td>CPU</td>
</tr>
<tr>
<td>GOOD</td>
<td>14</td>
</tr>
<tr>
<td>BAD</td>
<td>4</td>
</tr>
<tr>
<td>QoS Level</td>
<td>CPU</td>
</tr>
<tr>
<td>GOOD</td>
<td>15</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>10</td>
</tr>
<tr>
<td>BAD</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 1: Video mapping.
Table 2: Doom mapping.

Moreover we define two priority levels: HIGH and LOW.

4.3 QoS architecture description

To make the system QoS aware, we need to integrate the Qinna architecture. First of all, we identify five component classes, where only four need QoS management. The QoSComponents are video, Doom, CPU and memory. As described in section 3, we associate one QoSComponent-Broker and one QoSComponentManager to each class of QoSComponent. Moreover, a QoSDomain is added in order to delimit a multimedia QoS domain.

The next step is to configure the QoSComponentBrokers and the QoSComponentManagers. The QoS mapping of video and Doom QoSComponentManagers are given by tables 1 and table 2 respectively. QoS mapping of CPU and memory QoSComponentManagers are empty since they do not have any required interface.

Global constraint of CPU and Memory QoSComponent-Brokers are set to 20 and 1000Ko respectively. By this way, we can limit resources consumption of applications executed in the QoS multimedia domain.

The overall system architecture is shown in figure 2.

4.4 QoS management description

Due to space restrictions, we detail only the most meaningful scenario which shows how QoS adaption is managed by the Qinna architecture.

We suppose that the system is already executing a video. This video has a QoS level equal to GOOD and a priority equal to LOW. We now suppose that the user requests for a Doom game with a QoS level equal to HIGH and priority equal to HIGH. The video will be degraded to BAD QoS to let enough resources to the Doom game. QoS adaptation is explained in the figures 3 and 4.

4.5 Performances
In this section, we give the impact of the Qinna architecture over the system previously presented.

- Memory footprint overhead: 13.6 Kb. This is the size of Qinna components. It represents 2% of the whole system.
- Duration of QoS initialization: 4.08 ms. It represents the delay needed by Qinna to setup the QoS level of the video component.
- Duration of QoS degradation: 4.16 ms. It represents the delay to realize the scenario of figures 3 and 4.
- Duration of QoS upgrading: 2.26 ms. It represents the delay to stop the Doom component and to upgrade the QoS level of the video component to GOOD.

5. RELATED WORK

In the area of business models, works like [21] tend to incorporate QoS management in CCM [16]. In the same kind of area, [17] defines a Java platform to manage resources of its specific Java components. QuA [18] has similar goals to our work: it proposes a QoS-aware component architecture thanks to a global platform which is responsible for the whole QoS management. In comparison with this work, QuA brakes component encapsulation because the QoS management of a component is not associated to it. Moreover, QuA and [17] are not component-based.

In embedded systems, SEESCOA [20] integrate QoS aspects to their component model using contracts taken from [5]. In comparison with our work, SEESCOA mainly focuses on temporal QoS aspect thanks to timed MSC associated to each interface. Meanwhile, this work does not take dynamic QoS management into consideration.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented a generic component-based QoS architecture called Qinna. We illustrated its feasibility through a typical multimedia handheld system including dynamic and heterogeneous QoS management. The handheld system is made of an H3800 IPaq and of the Think component framework. Using the Qinna architecture, it is able to dynamically manage QoS of a video player and of the Doom game.

Using Qinna, the system (a) respects the separation of concerns, (b) can be easily reused thanks to its identified components and (c) integrates heterogeneous aspects of QoS management.

In future work, we will implement more complex and representative case studies such as real-time or communicating systems. We will also integrate complex QoS management policies such as a hierarchical CPU scheduler [13]. Finally, we are working for the integration of the monitoring concept.

7. REFERENCES