INTEGRATION OF QUALITY OF SERVICE IN DISTRIBUTED OBJECT SYSTEMS

Jérôme Daniel[1,2], Bruno Traverson[1] and Sylvie Vignes[2]

[1] EDF DER, 1 Av du Général De Gaulle, F-92140 Clamart, France
[2] ENST, 46 Rue Barrault, F-75634 Paris Cedex 13, France

Jerome.Daniel@der.edf.fr, Bruno.Traverson@der.edf.fr, Sylvie.Vignes@enst.fr

Abstract: Quality of Service (QoS) has been used in several contexts, especially in network protocols like ATM and RSVP. More recently, standardization work has started in the area of distributed object systems, like CORBA (Common Object Request Broker Architecture) and ODP (Open Distributed Processing). However, no commercial solution supports full QoS properties like: modularity, observation, guarantee, negotiation, etc. After a brief state of the art, this paper describes an integration framework of QoS in CORBA based on ODP concepts and, in a more detailed way, its QoS definition language and its engineering model.

Keywords: QoS, CORBA, ODP, distributed system, legacy system

1 INTRODUCTION

Evolution of techniques in computer science and telecom area has enabled a rapid growth of client/server applications. However, middleware environments currently available are not designed to fully take into account Quality of Service (QoS) criteria.

This lack of support may be detrimental for large-scale usage of distributed systems. Both OMG (Object Management Group) and ISO (International Organization for Standardization) are actively working on introducing QoS concepts in their respective architecture: CORBA (Common Object Request Broker Architecture) [10] and ODP (Open Distributed Processing) [3].

This paper is composed of two parts. The first part constitutes a state of the art on QoS concepts and standardization processes. The second part describes our integration framework.

L. Kutvonen et al. (eds.), *Distributed Applications and Interoperable Systems II*

2 STATE OF THE ART

System behavior may be considered into two ways: the actions it performs, which constitutes the "functional behavior" and the way it performs its actions which concerns performance, security and resource access. Quality of Service formalizes this later "non functional behavior" of systems. In practice, distinction between these two aspects is not so clear and, in particular, it depends on the point of view applied to the system.

2.1 QoS Contracts

QoS obligations and *QoS expectations* may be seen as the "non functional" part of the contract used between a system and its environment. QoS obligations express the *QoS offers* of the system, while QoS expectations indicate what the system expects from its environment.

A QoS obligation and a QoS expectation may be linked by *a QoS relationship* that can be interpreted as "the QoS obligation is satisfied while the QoS expectation is satisfied".

A *QoS criterion* is a reference point on which a user may express QoS expectations. The starting point of quality of service is to satisfy end-user requirements : These requirements must match with the current capability of the system to establish a *QoS contract.* To satisfy a specific end-user requirement, cooperation of all the components of the system involved by the requirement is required. To this respect, *final obligations* (obligations facing end-user requirements) dictate individual obligations for each of the components involved in the system and necessitate global composition relationships.

2.2 QoS Properties

Some important properties must be supported by a QoS enabled system:

- *Composition*: A QoS offer may be defined at any level of a system. A QoS offer of a set of components may be derived from QoS offers of each individual component.
- *Observation*: Quality of Service of a system must be introspected to refresh QoS offers.
- *Guarantee*: QoS may vary during the system lifecycle. Flexible contracts should be defined to maintain some invariants.
- *Negotiation*: Distinct users may have different requirements on the same system. They should be able to express and negotiate their individual requirements when binding to the system.

2.3 QoS Management

A QMF (QoS Management Function) is a function that implements several QoS mechanisms to verify one or more QoS expectations. Thus, main QMF characteristics are the following:

- Control of the QoS target,
- Establish a condition to be verified for a QoS expectation in function of a set of QoS criteria,
- Observe the values of the QoS criteria,
- Maintain the present QoS as near as the desired QoS,
- Retrieve any information related to QoS.

QoS management implies the use of multiple QMF at distinct times in the system activity. QoS expectations for a specific activity or for a set of activities can be expressed in different ways and at different times. This is why QoS management should be used at the following steps of an activity:

- In a general way, QoS expectations must be built during system configuration, during system design, based on dedicated resources and use of ad-hoc services,
- Before initialization, QoS expectations may be sent to some or all the participants in an activity before it starts,
- At the activity initialization, QoS expectations may be negotiated between users and providers of services,
- During activity processing, QoS expectations may be modified in function of a new event like, for instance, a loose of performance,
- At posteriori, after the activity processing, it is possible to go on the performance analysis, contracts analysis, etc.

For any specific activity, the selection of the step when it is more appropriate to apply QoS management depends on the kind of QoS expectations and the activity lifetime.

2.4 QoS Notation

QoS notation is the mean, for the user, to express its requirements in term of quality of service. In the area of distributed application development, this may be done together with the functional description of the objects, i.e. with an IDL (Interface Definition Language).

2.5 Formal Expression of QoS

Formal expression of quality of service should permit the analysis and the composition of QoS expectations. The TLA (Temporal Logic of Actions) formalism has been selected by ISO [5]. This formalism, already used in telecommunication applications for temporal design [6], fits well to time-related QoS criteria (performance, for instance) but does not seem so well adapted to the other categories of QoS Criteria (for instance, reliability). However, one very important feature of

TLA is that it includes automata for verification, composition and validation of formula. Thus, the use of such automata at the QMF level seems to us interesting.

2.6 QoS in CORBA

In its more recent publication [10], OMG describes one possible integration of QoS in CORBA. The approach is quite different compared to that taken by ISO for ODP. It simply consists of defining a set of policies to express QoS.

Three levels for meeting QoS expectations are distinguished. The lowest level corresponds to the ORB level. Some expectations may be expressed by default for the ORB. An upper level is the "thread" level, where thread means client-server relationship. Some expectations at this level may override those expressed at the ORB level. Lastly, the upper level is the object level, where expectations are processed in priority and override, if necessary, those expressed at the thread or at the ORB level.

Thus, when a server is developed, a set of policies is specified. These policies are related to QoS criteria. When a server becomes accessible by exporting its reference, policy names are added to this later. By importing the reference, the client provides to the ORB a way to verify the QoS expectations of the client with regards to the criteria exported by the server.

This solution seems to us simple but inadequate because it does not cover all the needs for QoS. It is limited to criteria expression and verification of validity with what expects the user without enabling any dynamic and adaptive behavior.

2.7 QoS in ODP

To support QoS, a system must implement mechanisms that analyze requirements and then guaranty them. These mechanisms are called "QoS management" in ODP terms [4]. Among expected mechanisms, the main ones are the following:

- Contract refinement: decomposition of a global QoS contract into a set of individual QoS contracts, each applying to one computational object.
- Validation: verification of a QoS contract.
- Measurement: observation of the state of a QoS relationship.

These functions may be realized at distinct moments. For instance, validation and measurement may apply during the application execution while contract refinement may apply during application design.

The validation process is a basic mechanism, which supports QoS guaranties in ODP. The validation takes place in a negotiation zone, which is a collection of isolated objects. The term "isolated" means that QoS expectations only apply to objects belonging to the collection.

The validation process applies to the objects of the negotiation zone during a negotiation period. During this period, the validation process collects the QoS contracts derived from the QoS offers of the objects belonging to the negotiation zone.

2.8 Summary

In summary, the viewpoints taken in ODP and in CORBA on QoS are very different. ISO has chosen a generic approach, which is more flexible while OMG has taken a pragmatic solution which minimizes the impact on its current specification. This analysis has led us to design and to develop a generic QoS Framework that could support ODP concepts on CORBA platforms.

3 QUALITY OF SERVICE MANAGEMENT

This second part introduces a QoS Framework that could be applied to any distributed environments. We are going to discover how to express QoS relationships and how to build a QoS Framework that is able to guarantee QoS properties [2].

3.1 QoS Framework overview

As shown in the first part of this paper, quality of service is a concept that relates to the behavior of a system or an application. Quality of service is expressed by QoS relationships that link QoS criteria. Indeed, a QoS relationship establishes a link between expectations (for an object) and obligations (for the same object):

```
Expectations ( Object ) → Obligations ( Object )
```

Each object of a distributed system, that expresses quality of service is concerned by a QoS relationship and this, even if expectations or obligations are always true.

```
True → Obligations ( Object )          Pure obligation
Expectations ( Object ) → True         Pure expectation
```

In complex cases, QoS relationships could be composed to form a global QoS relationship. The composition mechanisms are hardly expressible because we have no generic notation that covers all QoS points of view and that could help us to simply express composition. Our framework will have to solve as much as possible this problem.

In order to respect the main QoS concepts in ODP, our framework will also have to include some functions that support the QoS properties : negotiation, modularity, observation and guarantee.

Thus, the QoS framework will have to include:

- a QoS notation to express QoS relationships (and if possible to allow compositions), and
- a QoS manager that supports QoS mechanisms.

3.2 QoS Notation

In first, we have to select a QoS notation, which is difficult to define, because this notation must be able to describe any QoS point of view. That is why we introduce a new QoS term called "QoS Object" which is a formal view of a QoS relationship. In

a similar way as in oriented object language, we use the term "QoS Instance" to specify an instance of a QoS Object.

A QoS Object is a description that includes two sections, which express expectations and obligations. These sections are called: "Require" and "Provide". To describe a QoS Object, we use a language called QDL (QoS Definition Language). Some examples in this part illustrate QDL and the complete grammar of the language is given in an appendix. The following example shows a very simple QoS object described with QDL. To keep this first example simple, the Require and Provide sections are empty.

```
QoS an_empty_qos_object
{
Require :
                    // etc.
Provide :
                    // etc.
}
```

To be used by any developer, QDL must be simple but it must also be complete enough to describe any QoS notation needs. Obligations (listed in the Provide section) consist of QoS offers that can be expressed by properties. Expectations (listed in the Require section) are constraints on properties of other objects (including system resources which can be designed as objects). To describe a constraint, QDL uses the OCL (Object Constraint Language) which is defined in UML (Unified Modeling Language) [8]. OCL has been designed to express any kind of constraints. In QDL, OCL expressions use QoS Objects and their properties as constrained objects.

To describe a "Require" section, QDL uses the following production rules (BNF notation):

```
< Require > ::= "Require" ":" < constraint > +
< constraint > ::= "{" < OCL constraint > "}"
```

In a "Provide" section of a QoS object description, we distinguish two kinds of properties:

- simple property, and
- complex property.

A simple property is just a name / value pair which can be constant (the same value during all the lifecycle of the QoS Instance) or variable. The property value is typed and QDL uses OCL types for property types. The production rule to describe a simple property is (the property value will be assigned at the object instantiation):

```
[ const ] "property" < OCL type > < Identifier > ";"
```

"const" keyword is only used for constant properties.

A complex property is a constrained property. This means that the property value can not be evaluated directly but needs to use other QoS object definitions (this kind of property implies composition). The production rules to describe a complex property are:

```
constrained property < OCL type > < Identifier >
                "=" "{" < OCL constraint > "}" ";"
```

To illustrate QDL, we are going to describe some QoS Objects. The first QoS Object, is a pure obligation that describes a QoS offer of a basic object.

```
QoS Base_QoS
{
 Provide :
        // Identify the QoS Object Provider
        const property string provider;
}
```

The second example also describes a pure obligation but shows, at the same time, the QDL capabilities to support inheritance. This QoS Object illustrates a possible QoS relationship for an ORB.

```
QoS ORB : Base_QoS
{
 Provide :
        // ORB functionalities
        const property boolean real_time;

        // Common services
        property set initial_services;

        // Objects available
        property set objects_available;
}
```

The next example describes a pure expectation that expresses a requirement from the ORB (it requires that ORB provides a Naming Service into its common object services set):

```
QoS Object_Using_NamingService : Base_QoS
{
 Require :
     { ORB system;
      system.initial_services.exists
        ( string s | s = "NamingService" );
    }
}
```

The following example shows a QoS object that contains Require and Provide sections. The main interest of this example is to illustrate the constrained property concept.

```
QoS Complete_Object : Base_QoS
{
 Require :
        { ORB system;
         system.objects_available.exists
                ( string s | s = "ObjectA" ); }
 Provide :
        constrained property real availability_rate =
                { ObjectA O;
                 availability_rate = O.availability_rate };
}
```

3.3 QoS Object compositions

A constrained property implies a composition between several QoS objects. We distinguish two kinds of compositions:

3.3.1 A cooperative composition

Such a composition implies the use of another QoS object to provide a QoS offer. In this example, QoS Object A cooperates with QoS object B to resolve one or several constrained property values (from QoS object A Provide section). To implement this composition we need to look for all cooperative QoS objects, to provide global QoS offers. This consensus mechanism consists in an enumeration of all cooperative QoS objects that are linked together. These links will be called "cooperative link".

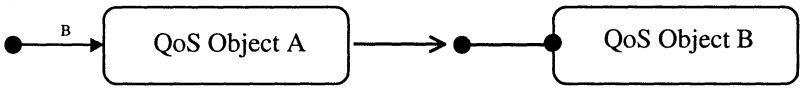

Figure 1. Example of cooperation composition.

3.3.2 A concurrent composition

A concurrent composition means that a QoS Object uses several other QoS objects to provide a QoS offer. So, this kind of composition implies a multiparty composition. In the example, QoS object A cooperates with QoS objects B and C. This composition is concurrent because a QoS offer of B may be incompatible with a QoS expectation of C. For example, a QoS property of B could have a value that violates a QoS expectation of C. To implement a concurrent composition, we need another kind of relation that link potential concurrence: concurrent links. In this way, we first solve each cooperative branch and check if there is no concurrent link between selected QoS objects.

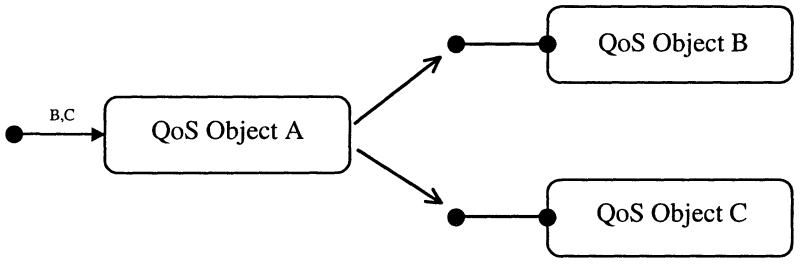

Figure 2. Example of concurrent composition.

3.3.3 A compatibility composition

The cooperative composition needs a checking on "Provide" sections, whereas a concurrent composition implies a checking on "Provide" and "Require" sections. There is a third case, where a checking between "Require" sections is needed. Such a checking must be completed even if QoS Objects are not composed. It implies a special link between QoS Object "Require" sections that we called "compatibility link". The following table resumes all possibilities:

	Require	Provide
Require	compatibility	concurrent
Provide	concurrent	cooperation

Table 1. Links nature depending on "Require" and "Provide" sections.

3.3.4 An example

The following example illustrates the three kinds of composition.

```
QoS A
{
 Require :
        {s:securityService, s.level=3; }
 Provide :
        property long width;
        constrained property long height =
                {b:B ; this.height=b.size; }
}
```

The first QoS Object (A) provides a constrained property (height) that uses another QoS object (B) property (size). This implies a cooperation link.

```
QoS B
{
 Require :
        {s:securityService, s.level=5; }
 Provide :
        property long size;
}
```

The second QoS Object (B) express into its "Require section" the needs of a "security service" with a "level" property value equals to 5. As QoS objects A and B express a constraint on the same QoS offer, they are linked by a compatibility link.

Into this example, A and B are denoted incompatible (the compatibility link is impossible to establish because they require a different level of security).

```
QoS C
{
 Require :
        {s:securityService, s.level=5; }
        {b:B, b.size !=50; }

}
```

At last, in a similar way QoS object C have two compatibility links between A and B. Moreover, C expresses a second constraint on B from its Require section. This last link is a concurrent link.

3.4 QoS Manager

To identify the QoS Manager components, we are going to analyze the successive steps of QoS management. First of all, QDL descriptions are parsed. This parsing generates two components:

- a QoS Proxy, and
- a QoS Unit.

A QoS Proxy is a proxy file that links an object implementation to a QoS object. In this way, several different objects can share the same QoS object. This concept is very important because it is a natural way to think that different objets share a same need for QoS. In order to report QoS modification from an object implementation, we need a connection between this object and the QoS manager. That is why the proxy is linked to the QoS Manager by a QoS channel:

Figure 3. Role of QoS Channel between QoS Manager and QoS Proxy.

The QoS channel allows QoS data exchanges between object implementation and QoS Manager. The QoS Proxy is automatically generated and permits an easy introduction of QoS in an object implementation. The independence between implementation and QoS also allows the management of QoS for legacy objects. In this later case, it is possible to affiliate QoS expectations and QoS offers to legacy objects with a generic QoS Proxy. These functionalities could be applied by administrative tools.

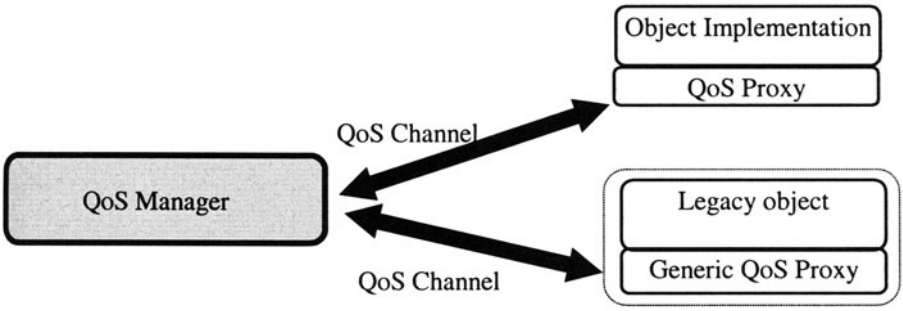

Figure 4. Example of legacy object QoS management architecture.

The second component generated at QDL compilation time is a QoS Unit, which is simply a binary form of a QoS Object. Each QoS Unit is put into a repository called "QoS Unit Repository (QUR)".

An object implementation exports its QoS property values to create a QoS Instance, which is added to a "QoS Instance Base (QIB)". A special component called "QoS Manager Function" is responsible of the insertion of the QoS Instance into the QIB. The following figure illustrates the QoS Manager architecture.

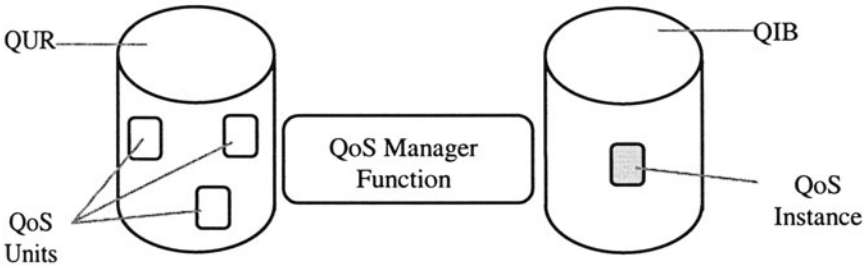

Figure 5. QoS Manager components.

Each time a QoS Instance must be added to the QIB, the QoS Manager function firstly checks the "Require" section of the new instance. This mechanism consists in the creation of "compatibility links" and "concurrent links". If the checking is successful, the second step is to compose "Provide" sections to create "cooperation links".

Finally, there is into the QIB a graph that represents all the composition links between QoS instances. When a QoS Offer is selected by a client application, a path is marked into the graph that represents a QoS contract.

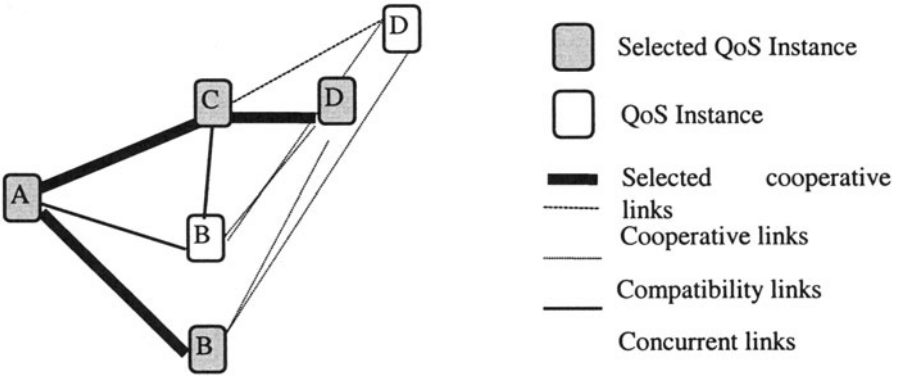

Figure 6. Example of QoS contract.

When a property value changes, the QoS proxy sends the new value to QoS Manager. The value is modified into the QoS Instance and the QoS contract is checked. If the QoS contract is no more valid, a new QoS contract is created if any is possible. If a new contract is impossible, an exception is thrown to the concerned QoS instance (or client application). On the other hand, if a new contract is created, the new implementation objects receive the communications from the old implementation objects (which are no more valid with their QoS relationships).

As QoS instances are representations of QoS offers and QoS expectations, and QoS contracts are relationship between QoS instances, the QIB is a QoS mirror of the distributed system.

At this point, we have defined a QoS manager that is able to manage and to compose QoS relationship. Moreover, we have defined a QoS notation that seems to be able to describe any QoS point of view (including system QoS: To dispose of a complete model, it is necessary to completely describe the system offers). Lastly, we have been able to affect QoS relationship and to manage legacy objects with administrative tools.

4 CONCLUSION

Client/server architecture is mature enough to enable the support of Quality of Service, which is becoming vital in mission-critical applications. Standard organizations are in the process of introducing QoS mechanisms in their current specifications. However, there is no implementation, as far as we are aware, which solve problems of notation, composition and guaranty of QoS properties. This paper describes a model that answers to these problems and permits the introduction of these capabilities into legacy systems.

We can expect convergence between ISO and OMG works in a near future. An OMG green paper [9] proposes to use ODP concepts for CORBA. There are multiple QoS related working groups at OMG. Other works describe similar notation as QDL to express QoS notation [12].

We are currently implementing the QoS Manager on our CORBA prototype [1]. This will permit to validate as well the notation and the composition mechanisms as the integration of legacy systems by administrative tools.

References

[1] DANIEL J., TRAVERSON B., ZAKARIA M., *Prototyping activities on distributed object platforms*, EDF technical overview, May 1997.

[2] DANIEL J., TRAVERSON B., *Study of QoS mechanisms in distributed object systems* (french), EDF paper, September 1998.

[3] *Open Distributed Processing Reference Model, parts 1, 2, 3, 4*, ISO/IEC IS 10746-1.. 4 or ITU-T X901..4, 1995.

[4] *Open Distributed Processing Reference Model, Quality of Service*, ISO/IEC WD, January 1998.

[5] LAMPORT L., The temporal logic of actions, *ACM Transactions on Programming Languages and Systems*, pages 872-923, May 1994.

[6] LEBOUCHER L., NAJM E., *A Framework for real-time QoS in distributed systems*, January 1998.

[7] MAFFEIS S., SCHMIDT D., *Constructing reliable distributed communication systems with CORBA*, 1997.

[8] Object Constraint Language Specification, Version 1.1, *Rational Software*, September 1997.

[9] *Quality of Service (QoS)*, OMG Green Paper, June 1997.

[10] *The Common Object Request Broker Architecture and Specification, Revision 2.2*, OMG, February 1998.

[11] *Quality of Service,* Draft Paper, OMG, February 1998.

[12] RAKOTONIRAINY A., Adaptable transaction consistency for mobile environments, *DEXA'98,* pages 440-445, Ed. Roland R. Wagner.

Appendix: QDL production rules

This appendix completely defines the QDL grammar using a BNF notation.

```
1 : < QDL description > ::=   [ < import clause > ]
                              < QoS Object description >*

2 : < import clause > ::= {   'import' ' " '
                              < path and file name >
                              ' " ' ' ; ' }+

3 : < QoS Object description > ::=    'QoS'
                                      < identifier >
                                      [ < inheritance > ]
                                      < description body >

4 : < inheritance > ::=       ' : '
                              < identifier >
                              { ' , ' < identifier > }*

5 : < description body > ::= ' { '
                              [ < expectations > ]
                              [ < obligations > ]
                              ' } '

6 : < expectations > ::= ' Require ' ' : ' < constraint >*

7 : < obligations > ::= ' Provide ' ' : ' < property >+

8 : < constraint > ::= ' { ' < OCL expression > ' } '

9 : < property > ::= < simple property > | < complex property > ' ; '

10 : < simple property > ::=  [ ' const ' ]
                              ' property ' < OCL type >
                              < identifier >

11 : < complex property > ::= ' constrained ' ' property '
                              < OCL type > < identifier >
                              ' = ' < constraint > ' ; '

12 : < OCL type > ::= see OCL specification [OCL 97]

13 : < OCL expression > ::= see OCL specification [OCL 97]

14 : < identifier > ::= see OCL specification [OCL 97]
```