# On QoS-Aware Publish-Subscribe*

Filipe ARAÚJO
Universidade de Lisboa
filipius@di.fc.ul.pt

Luís RODRIGUES
Universidade de Lisboa
ler@di.fc.ul.pt

## Abstract

*This position paper addresses the issue of supporting quality of service (QoS) parameters in distributed publish-subscribe systems. It advocates that QoS parameters should be handled using the same constructs as other information regarding events, such as their type or content. At the same time, we claim that the use of a consistent set of mechanisms should not preclude to decouple the specification of QoS properties from the specification of type, subject or content-based constraints.*

*We also advocate that QoS parameters should not be embedded on the type or content of the events. We show that some QoS parameters can only be computed in run-time, as they depend on dynamic aspects such as the location of the participants and the system load.*

*The paper proposes a model that supports the decoupling of QoS characterization from the event characterization while, at the same time, offers an uniform treatment of both aspects.*

## 1 Introduction

The indirect communication, in particular the publish-subscribe communication model, is gaining increasing acceptance as a useful alternative to direct communication models, such as the ones based on remote invocations. The main advantage of this paradigm is that it supports a weak coupling among participants, which do not need to be aware of the location or number of its peers. This simplifies the reconfiguration of the applications and eases the re-use of the same components in different applications.

A limitation of most existing architectures that support the publish-subscribe communication is their limited support for the expression and enforcement of Quality of Service (QoS) parameters (such as required bandwidth or latency, for instance). This observation applies both to models, such as the CORBA Event

*Parts of this report will be published in the Proceedings of the International Workshop on Distributed Event-Based Systems, Vienna, Austria, July, 2002. (In conjuction with the 22nd International Conference on Distributed Computing Systems). This work has been partially supported by the project POSI/ 41473/ CHS/ 2001, INDIQoS.

Service [12], CORBA Notification Service [11], Java Message Service [14] and to systems, such as CEA (Cambridge Event Architecture) [2], Distributed Asynchronous Collections [9] or SIENA (Scalable Internet Event Notification Architectures) [6]. This is a significant drawback, since QoS features are an important component of applications, and its use and support has been widely studied in the context of direct communication [5, 4, 16, 3].

There is a fundamental reason for the current state of the art: Traditional approaches to QoS provision are based on the establishment of channels or connections that reserve the resources required to provide the desired QoS parameters. This mode of operation fits in a natural way in the direct communication model, where connections are always explicitly setup, but it has an inherent mismatch with the decoupled nature of event based systems. In the indirect communication model, the applications should not be forced to explicitly setup channels. Instead, they should remain oblivious to the number and location of the participants involved in the communication and should be concerned exclusively with the properties of the information they are able to publish or subscribe.

Therefore, a new system model has to be designed to allow the seamless integration of QoS features in indirect communication systems. This model should:

- Allow the application to indirectly determine QoS parameters, by allowing to express QoS properties as a characterization of the information being produced or subscribed.
- Delegate on the message broker the task of establishing the required low-level connections, on behalf of publishers and subscribers. These reservations need to be based on dynamic information: on the number, location and characteristics of producers and consumers and also on the QoS characteristics of the information exchanged in the system.

This position paper proposes that, in publish-subscribe systems, QoS parameters should be treated in a uniform way with regard to other event attributes. In particular, similarly to well known subject-based, content-based, or type-based subscriptions, it should be possible to make QoS-aware subscriptions. On the other hand, the paper argues that QoS-related parameters must be decoupled from the information being exchanged, as several QoS parameters are of a dynamic nature and can only be evaluated in run-time.

The rest of the paper is structured as follows. The key idea of supporting QoS-aware addressing is presented in Section 2. The sketch of a distributed QoS-aware broker architecture to support our model is given in Section 3. Section 4 outlines an instantiation of our architecture on IP networks with RSVP. Section 5 concludes the paper.

## 2 QoS-Aware Publishing and Subscribing

One of the main advantages of the publish-subscribe model is that it decouples publishers and subscribers in several dimensions. In [10] three dimensions of decoupling are introduced: *space* decoupling (that cap-

tures the fact that interacting parties do not need to known each other); *time* decoupling (that captures the fact that parties do not need to be actively participating in the interaction at the same time); and *flow* decoupling (that captures the asynchrony of the model). In this position paper we address a fourth dimension of decoupling, what we may call *QoS decoupling*, that captures the separation of QoS parameters from the type or content of events.

The model advocated in this paper has the following characteristics. The QoS of the event dissemination is established in run-time, based on the desired properties expressed by subscribers, on the shape of the sources advertised by the publishers, and on available resources. An important aspect of the model is that subscribers should be able to express QoS constraints using the same type of constructs they use to express other sort of constraints (such as content-based constraints). Publishers, on the other hand, do not tie a specific QoS with the information produced. However, they must advertise the *shape* of the information being produced, in the form of an *event QoS profile*. The event QoS profile is used in run-time by the message broker to estimate the resources demanded by a given flow and to match the QoS constraints specified by subscribers with the characteristics of the information produced by publishers. The message broker plays an important role in a QoS-aware publish-subscribe system, because it must ensure that QoS requirements are met. Besides, the message broker must cope with QoS related parameters present in advertisements, notifications and subscriptions.

To make our case we will use the following example. Consider a building where rooms are equipped with a number of temperature sensors. These sensors advertise the room temperature in an event of type *Temp*. Consider that the attributes of these events are as follows: *room*, that indicates the room where the temperature is being measured; *temperature*, that indicates the room temperature; and *precision*, that indicates the precision of the sensor.

Our case is independent of any particular language construct to be used when specifying notifications or subscriptions. In the following examples we will follow a notion that closely resembles the type-based publish-subscribe model of [8]. Using this model, typical subscriptions would be:

*Subscriber s* = **subscribe** *Temp*
**where** (room = “lab1”)

or

*Subscriber s* = **subscribe** *Temp*
**where** (temperature $>$ 60)

The first expression corresponds to a subscription of events with the temperature of room “lab1” and the second of events from any room where the temperature is greater than $60$. On the publisher side, the interface looks somehow like this:

*Publisher p* = **new** *Publisher*
**of** *Temp*
**withProfile** (room="lab1", temperature=any, precision=0.01);

*e* = **new** *Temp* (room="lab1", temperature=16, precision=0.01)

*p.publish (e)*

The *Publisher* is an auxiliary component that is used to disseminate events. Among other purposes, it allows the publisher to inform the message broker of the type of events it is going to produce. This information takes the form of advertisements. In the example above, we consider only a *content profile*, the profile that characterizes the content of the information being published. In this example, the publisher states that the events it produces may have different values in the *temperature* field but have a fixed value in the *room* and *precision* fields. This information may be used by the broker to optimize the dissemination of events [7]. We will now discuss how to advertise QoS related profile information (in addition to the *content profile*).

Consider now that each of these sensors has a different QoS parameters. Consider that *Sensor1* produces sporadic events, only when it detects a temperature change. Both *Sensor2* and *Sensor3* produce new events at a periodic pace, but with different periods.

The question is, of course, where to include the QoS characterization of the events, both at the producer and at the consumer. Since we are interested in giving the application designer a uniform interface, we would like to use mechanisms to express the QoS parameters that are similar to the ones used before to express the content of the information being produced.

One possible approach would be to code the QoS information in the event *type*. For instance, one could define two different types: *SporadicSensor* and *PeriodicSensor* and include other QoS information, such as the period, as an attribute of the *PeriodicSensor* type. However, we believe that this approach has several disadvantages. When combined with other QoS attributes, such as reliability or availability, this quickly leads to an explosion of different types for the same information being produced.

One of the main reasons to reject this sort of coupling is that some QoS attributes can only be derived at run-time. Consider for instance the case where a subscriber is interested in receiving a temperature event but wants to specify a minimum latency in the event dissemination. Clearly, the latency is not an inherent property of the information being disseminated. Furthermore, latency is a function of several run-time parameters, such as the relative location of the subscriber and the publisher and the load of the links between these participants.

To address these issues we propose an architecture where publication and subscription operations are augmented with QoS attributes that can be used to define filtering conditions in a similar way to that of

content-based filtering. In order to do so, publisher must advertise a *profile* of the event publishing pattern. In our example above, sensors should characterize the nature of the event pattern, declaring if it follows a sporadic or periodic profile. For instance, the sporadic sensor would declare the shape of the information produced as a *QoS profile* that can be provided in addition to the content profile:

// *Sensor1*
*Publisher p* = **new** *Publisher*
**of** *Temp*
**withProfile** (room="lab1", temperature=any, precision=0.005)
**withQoSProfile** *Sporadic*

While periodic sensors would have also to specify the period in order to fully characterize the shape of the source:

// *Sensor2*
*Publisher p* = **new** *Publisher*
**of** *Temp*
**withProfile** (room="lab1", temperature=any, precision=0.01)
**withQoSProfile** *Periodic (period = 1)*

// *Sensor3*
*Publisher p* = **new** *Publisher*
**of** *Temp*
**withProfile** (room="lab1", temperature=any, precision=0.01)
**withQoSProfile** *Periodic (period = 10)*

Note that while advertisements are not mandatory in non-QoS-aware publish-subscribe systems, they are of utmost importance in a QoS-aware system. In fact, some QoS related information, such as the period, is not a characteristic of each individual event but of the *shape* of the traffic produced by the publisher. Given the type of decoupling aimed in the model proposed here, the *profile* of the source must be advertised independently of each individual publish operation.

On the subscriber side, the desired QoS attributes could be expressed using a filtering condition similar to the one used for the information contents. For instance:

Subscription s = **subscribe** *Temp*
**where** (temperature $>$ 60)
**withQoS** ((*Periodic(period<1)*) **and** (*latency<10*))

There are a number of issues regarding this model that need to be emphasized. First, some of the QoS attributes specified in the subscription, such as the latency attribute, have no match in the information being advertised, and must be interpreted by the message broker itself. Other examples include a QoS specification including a reliability attribute, that depends of the available transport protocols. Additionally, a subscription may be refused due to lack of system resources. For instance, it may be impossible to satisfy the latency constraint specified in the subscription.

## 3 QoS-Aware Distributed Message Brokers

Some QoS parameters are already supported in some publish-subscribe models or systems, such as CORBA Notification Service [11], Java Message Service [14] or Distributed Asynchronous Collections [9]. This is the case of message reliability, message priority, message earliest delivery time, message expire time, duplicate message detection or message ordering, for instance. Depending on the architecture, these QoS parameters may be supported or not.

As far as we know, QoS parameters such as latency, bandwidth, availability, jitter or loss ratio, that have been widely studied in the direct communication paradigm, are not adequately addressed in publish-subscribe systems. Hence, we envision a message broker that also copes with this type of QoS parameters. This is a difficult task that is considerably different from ensuring existing QoS parameters such as message reliability or message ordering, for instance. To ensure this sort of QoS parameters it is necessary to do reservation of resources along the path(s) connecting publishers and subscribers. In a publish-subscribe system, to preserve the decoupling among the participants, reservations should be done by the message broker on behalf of the applications. This clearly prompts for the development of QoS-aware distributed message brokers.

A QoS-aware message broker is a distributed component that manages the following entities:

- Publishers' advertisements, including the *QoS profiles* of the information being advertised.
- Subscriptions, including desired *QoS conditions*.
- System resources.

The system resources represent the networking, memory and processing resources available to support the exchange of events. They encapsulate low-level QoS protocols, such as RSVP or other similar mechanisms widely used in direct communication systems [5, 4, 16, 3]. To ensure QoS, the access to these resources must be restricted. A QoS-aware message broker must implement a resource accounting module and an admission control module. The former should be responsible for the bookkeeping part, while the latter is responsible for admitting or rejecting new subscriptions (to do that it must use the accounting module facilities).

Consider for instance the network of Figure 1 and subscriptions of the form:

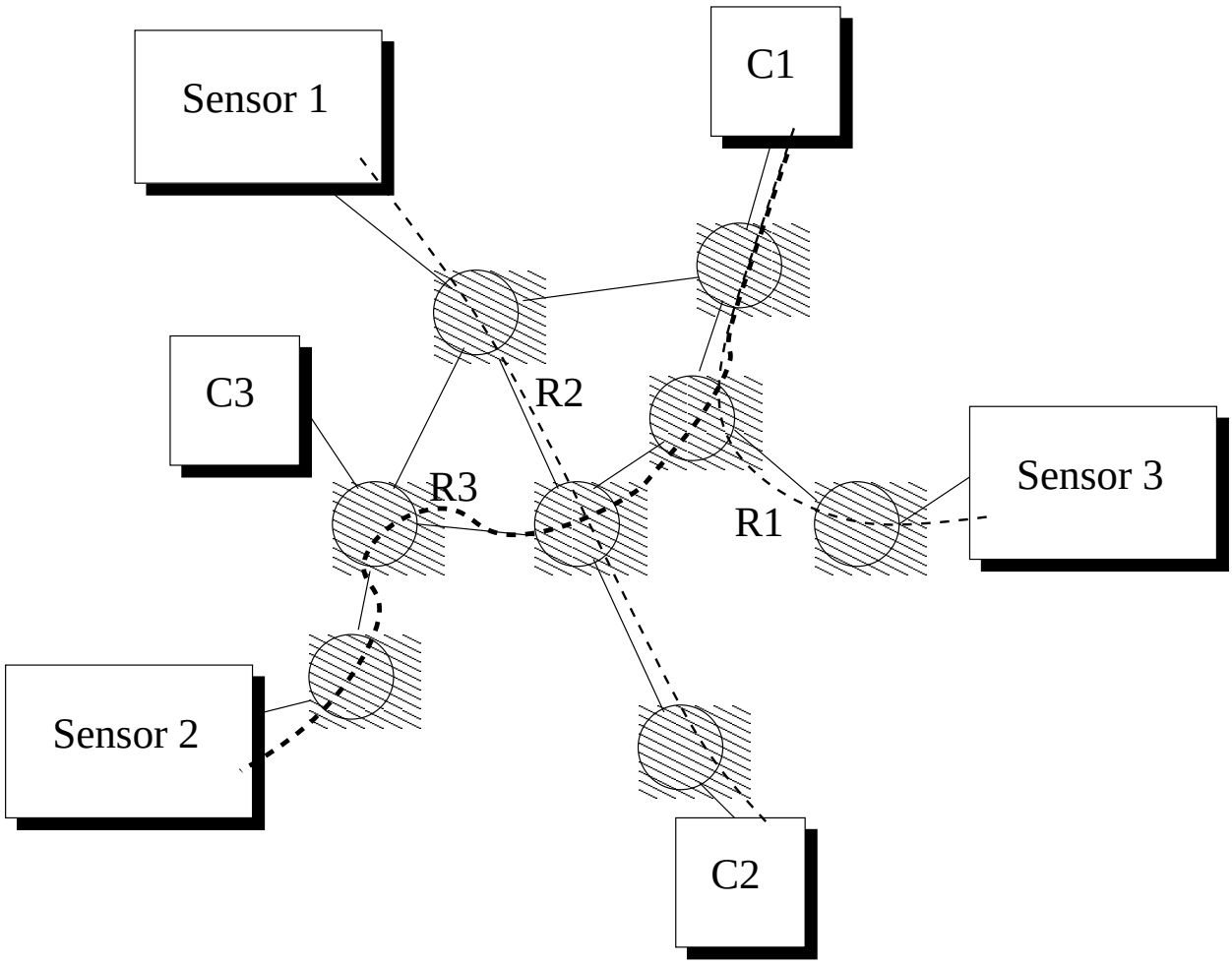


**Figure 1. Automatic reservations**

*//C1*

*Subscriber s* = **subscribe** *Temp*

**where** (temperature $>$ 60)

**withQoS** (*Periodic(period*$\leq$*1)* **and** *latency*$<$*10*)

*//C2*

*Subscriber s* = **subscribe** *Temp*

**where** (temperature $>$ 60, precision $=$ 0.005)

**withQoS** (*Sporadic*)

In response to such subscription, the message broker would have to make reservations $R1$ and $R3$ to satisfy the request of Client $C1$ and reservation $R2$ to satisfy the subscription of client $C2$. The message broker is also responsible for optimization of resources. To save resources it should try to merge subscrip-

tions as close to subscribers as possible. For instance, suppose that another client performs the following subscription:

*//C3*
*Subscriber s* = **subscribe** *Temp*
**where** (temperature $>$ 60)
**withQoS** (*Periodic(period*$\leq$*1)* **and** *latency*$<$*20*)

This third subscription can be satisfied using the reservation $R3$, made to satisfy the subscription of client $C1$. A QoS-aware broker must be able to implement this type of optimizations to save valuable resources. This type of problem, often known as the merging problem, has been studied for content-based addressing [7] and must now be extended to cover also QoS considerations.

The main difficulty of implementing a QoS-aware distributed message broker is that one must be able to deal with complex optimization problems. The definition of scalable and efficient heuristics to deal with allocation and sharing of resources in face of dynamic subscription and advertisement patterns is a challenging research area.

## 4 An Instantiation Using RSVP

We have started to build a first prototype of our QoS event-architecture (IndiQoS) on IP networks with RSVP [4] with Integrated Services [5, 16]. This work, reported in [1], addresses the following issues:

- *selection of meaningful QoS parameters for publishers and subscribers*: QoS parameters must be chosen in a way that allows translation to the underlying network architecture;
- *mapping problem*: how to setup network resources to optimally route events from publishers to subscribers. Usually, this problem reduces to distribute (map) available IP multicast addresses to subscribers of events.

For the sake of simplicity, in this first prototype, we chose token bucket parameters as QoS parameters for applications (as in [5], we also require the peak data rate value to be provided). Therefore, publishers and subscribers should specify values for bucket size and rate. Additionally, subscribers may also include a latency constraint.

Depending on the specified QoS parameters, different services are required from the network protocols. A controlled-load service [15] is required when both publisher and subscriber request token-bucket parameters. A guaranteed service [13] is required when the subscriber also requests latency. If the subscriber does not specify any QoS parameter, only best-effort service is required.

In the same paper [1], we also discuss why QoS parameters must be taken into account when solving the mapping problem. In this setting, an efficient solution to the mapping problem should try to merge related

subscriptions such that a same connection at the network level can be used to support several subscribers with compatible QoS requirements.

## 5 Conclusion

This paper discusses the issue of supporting QoS attributes in publish-subscribe systems. We advocated that QoS attributes should be managed in a uniform way with regard to other attributes such as type or content. In particular, we presented a model for applications, with QoS-aware publications and subscriptions that preserves the decoupling that makes the publish-subscribe model so appealing. Using QoS based subscription, consumers of information may specify in a declarative manner both the type, content and QoS attributes such as latency, reliability, etc, of the information they are interested. To support such model, new QoS-aware distributed message brokers must be built. These brokers must be able to match subscription with run-time parameters such as the location of participants and the available resources. Additionally, these brokers must be able to promote resource sharing when subscriptions are compatible.

## References


[1] Filipe Araújo and Luís Rodrigues. The IndiQoS message broker: an instantiation using RSVP. DI/FCUL TR 02–3, Department of Computer Science, University of Lisbon, March 2002.

[2] Jean Bacon, Ken Moody, John Bates, Richard Hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel, and Mark Spiteri. Generic support for distributed applications. *IEEE Computer*, March 2000.

[3] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differenciated services, December 1998. RFC 2475.

[4] Ed.R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (RSVP) — version 1 functional specification, September 1997. RFC 2205.

[5] R. Braden, D. Clark, and S. Shenker. Integrated services in the internet architecture: an overview, June 1994. RFC 1633.

[6] Antonio Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, December 1998.

[7] Antonio Carzaniga, David S. Rosenblum, and Alexander L Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[8] P. Th. Eugster, R. Guerraoui, and Christian H. Damm. Linguistic support for large-scale distributed programming. In *In 16th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001)*, pages 131–146, October 2001.

[9] P. Th. Eugster, R. Guerraoui, and J. Sventek. Distributed asynchronous collections: Abstractions for publish/subscribe interaction. In *In 14th European Conference on Object Oriented Programming (ECOOP 2000)*, pages 252–276, June 2000.

[10] Th. Eugster Felber. The many faces of publish/subscribe. Technical report, Swiss Federal Institute of Technology in Lausanne (EPFL), 2001.

[11] Object Management Group, OMG Headquarters, 250 First Avenue, Suite 201, Needham, MA 02494, USA. *Notification Service Specification*, June 2000.

[12] Object Management Group, OMG Headquarters, 250 First Avenue, Suite 201, Needham, MA 02494, USA. *Event Service Specification*, March 2001.

[13] S. Shenker, C. Partridge, and R. Guerin. Specification of guaranteed quality of service, September 1997. RFC 2212.

[14] Sun Microsystems, 901 San Antonio Road, Palo Alto, CA 94303, USA. *Java Message Service*, November 1999.

[15] J. Wroclawski. Specification of the controlled-load network element service, September 1997. RFC 2211.

[16] J. Wroclawski. The use of RSVP with IETF integrated services, September 1997. RFC 2210.