

Quality of Service Semantics for Component-Based Systems

Richard Staehli¹

Frank Eliassen¹

Jan Øyvind Aagedal²

Gordon Blair³

¹*Simula Research Laboratory*, ²*SINTEF*, ³*University of Tromsø*

{richard, frank}@simula.no, jan.aagedal@sintef.no, gordon@comp.lancs.ac.uk

Abstract

QoS management in component architectures requires that we understand how to specify QoS for a composition of components. Other researchers have described domain specific models for QoS specification, but with imprecise semantics that make it difficult to reason about a mapping to lower layers of system implementation. This paper describes a general model for defining the semantics of a QoS specification that can be applied at any level. The key features of the model are a definition of a service based on a composition of logical component behavior and a definition of quality dimensions based on a metric space of service behavior. The model supports mapping between QoS behavior of a service and the QoS behavior of the component services that implement it. We give examples to show how to define the quality dimensions used in practical applications. The precise semantics of our model allow us to move forward with research into the use of such quality dimensions in component-based QoS management.

1 Introduction

The need for QoS management support in component architectures is well known [1] [2]. While many aspects of QoS management have been investigated in the context of distributed systems, component architectures present new challenges. One of those new challenges is how to specify QoS for a composition of components.

Component architectures such as the CORBA Component Model (CCM) guarantee that applications *assembled* from prebuilt components will function correctly when deployed on any sufficiently provisioned implementation of the component architecture platform. We refer to this as the *safe deployment property*. Although current component architecture standards have had good success in some domains, such as e-business, an application that performs well in one deployment may be unusable as load scales up or when connections are redistributed across low-bandwidth connections. We use the term *QoS-sensitive application* to refer to an application that will commonly perform unacceptably if platform resources are scarce or if the deployment is not carefully configured and tuned for the anticipated load. State-of-the-art middleware and component technologies provide QoS management APIs that force application developers to

code deployment-specific knowledge into the application [3][4][5]. This approach is unable to preserve the safe deployment property.

The QuA project is investigating how a component architecture can preserve the safe-deployment property for QoS-sensitive applications [6]. We believe that *platform-managed* QoS is the only general solution that preserves the safe deployment property. This means that applications and application components should be written without dependencies on physical resources or knowledge of platform service implementations. Application requirements for accuracy and timeliness of outputs must refer only to the logical properties of the component interfaces. It also means the platform must be able to reason about how QoS behavior of a service depends on QoS of the services from which it is composed.

The problem we address in this paper is how to define a standard representation and semantics for QoS requirements that a component architecture platform can use for QoS management. In the next section, we propose a general model and semantics for QoS specification. In Section 3 we discuss how this model can be applied to a component architecture.

2 A General QoS Model

Fundamental to our approach is that we believe component architectures should be service oriented. If "everything is a service", then we can analyze the behavior of any complex system as a composition of simpler services.

Definition 1 A *service* is a subset of input messages to some composition of objects and their causally related output messages.

In this definition, we assume the most basic model of object-oriented computation where objects communicate by sending messages. The sending of a message is both an input event for the receiver and an output event for the sender. We make no assumptions about the implementation of the objects and therefore we must allow that objects may receive input messages concurrently from multiple senders. This suits a general model of distributed computation.

One important feature of this definition is that two services may share the same object and even the same interface of that object, but each message send is associated with exactly one service.

It is useful to view a service as an abstract object composed of the network of concrete objects: both the objects that directly receive service inputs and emit outputs and the other objects that participate in the causality.

A service specification can be as simple as an interface specification with causally related outputs defined by the semantics of interface operations. For example, a thermometer object might offer a `get` operation that returns the current temperature in a `return(temp)` message and a `set(temp)` message used by the environment to communicate with the thermometer. The input events are the `set(temp)` and `get` messages and the output events are the `return(temp)` messages. The semantics of the `get` operation dictate that the return value should be the same as the value in the most recent `set(temp)` message, so a `return(temp)` message is causally related to both its associated `get` message and some prior `set(temp)` message.

Definition 2 A *message event trace* is a set of mes-

sage values associated with the sender, the receiver, and the time the message was sent.

A message value represents all content of the message, including its type or signature. Where the distinction is important, we adopt the convention that the a message is always sent at the sender's location so that time is relative to the sender. In order for a remote object to receive such a message, it must be represented by a local proxy, which receives the message at the same time it is sent via a local procedure call.

We use the term *input trace* to refer to the message event trace with all input events for a service, and *output trace* to refer to the message event trace with all output events for a service. The behavior of a service implementation is determined not only by the semantics of its declared interfaces, but also by the availability and scheduling of resources in the underlying platform. To define quality of service, we need to ask what is the ideal behavior of a service.

Definition 3 For a given input trace, the *ideal output trace* is generated when the service executes completely and correctly on an infinitely fast platform with unlimited resources.

That is, a computation takes no time and results are obtained at the same time they are requested. For deterministic computations, the semantics of a service's interfaces defines the ideal output trace as a function of an input trace.

In a real implementation of a service, the actual output trace will differ from the ideal in both the timing and value of message events. The causes for this deviation from ideal include finite CPU speed, queueing delays, and bandwidth reduction strategies. This is the stuff of QoS management.

We can view the possible output traces for a service as points in a behavioral space and consider distance from the ideal output trace as an error measurement, but to use this concept in QoS specifications, we must first define the dimensions of this behavioral space. It helps to begin with a small example.

Consider again, the thermometer object and a service whose inputs are the `set(temp)` messages and a single `get` message, and outputs are the single causally related return value message. For an

input trace where the `get` message occurs at time t , the ideal output trace reports the return value message also occurring at time t with the exact value from the most recent `set(temp)` event.

The only possible ways in which an actual output trace can differ are in the time of the output event and in the value returned. Each independent way in which a trace can differ from the ideal represents another dimension of the behavioral space. Let \vec{X} be the vector of values that may differ in a trace X , ordered by event order in the ideal trace and by the order they occur in the message value. Then the difference between an actual output trace A and the ideal trace I is the difference vector $\vec{\delta} = \vec{A} - \vec{I}$.

As the complexity of an ideal trace increases, through additional messages and more complex message structure, the number of ways in which actual traces may differ explodes. Fortunately, we frequently are concerned only with an overall measure of distance from the ideal. For example, we can ignore the individual values for event delay and instead monitor aggregate statistics such as maximum and median.

Definition 4 An *error model* is a set of n functions $\epsilon_1(\vec{\delta}), \dots, \epsilon_n(\vec{\delta})$ that each map from a difference vector $\vec{\delta}$ to a real number such that each is zero when $\|\vec{\delta}\| = 0$ and each is monotonically increasing with every component of $\vec{\delta}$.

One such error model is the set of projection functions $\pi_i(\vec{\delta}) = \delta_i$; these are zero when $\|\vec{\delta}\| = 0$ and increase or remain constant when any component of $\vec{\delta}$ increases. But the value of this definition is that it allows us to construct a much simpler error space corresponding to the type of QoS characteristics commonly discussed in the literature for QoS management. A point in this error space represents an equivalence class of output traces that are the same distance from the ideal.

To continue the thermometer example, if we define a service consisting of all `get` messages from a single client and the associated return values, then both the input trace and its associated ideal output trace may contain an unbounded number of events. We can construct an error model with the four functions *maxDelay*, *medianDelay*, *maxError* and *medianError*, where delay is difference between the time the temperature is returned and the ideal time

and error is the absolute value of the difference between the temperature returned and the ideal value. A point in this error space; say *maxDelay* = 1 second, *medianDelay* = 0.5 second, *maxError* = 1.0 C, and *medianError* = 0.1 C; corresponds to all output traces in which the maximum delay is one second, and so forth. This set of output traces forms a surface in the behavioral space that surrounds a neighborhood of the ideal output trace. Inside this neighborhood, all output traces are closer to the ideal and can be considered *better* according to this error model.

We can use an error model to both quantify the *loss of quality* and to constrain it. Let $\epsilon(\vec{A} - \vec{I})$ be the tuple of error values for an error model ϵ , an actual trace A and the ideal trace for a service I . Let *limits* be a tuple of positive real numbers representing an upper bound for each of the functions in ϵ . Then we say a service with output trace A is acceptable if for each i , $\epsilon_i(\vec{A} - \vec{I}) < \text{limits}_i$.

It is easy to see that many error models exist for a given service [7]. This prompts us to ask which error models are better than others. The formal definition we have proposed allows us to formally define desirable properties of a good error model. For brevity, we suggest only informal definitions here. We say an error model is *sound* if any set of non-zero error limits will define a non-trivial set of acceptable output traces. An error model is *complete* if, for any output trace that is different from the ideal, we can find a set of non-zero error limits that would exclude this trace. An error model is *minimal* if no function can be removed without losing the ability to distinguish between some output traces. We say an error model M is more *expressive* than an error model N if it can define exactly the same sets of acceptable output traces as N , and then some more.

3 Application to Component-Based Systems

In the QuA project, we are designing a QoS-aware component architecture that allows clients to express QoS requirements without specifying how those requirements are to be met [6]. The QuA component architecture applies the service abstraction at all levels of system implementation to model the behavior of a composition of objects. This allows the QuA platform to reason about how application level QoS results from a composition of application

and platform sub-services.

Application clients ask the QuA Platform to instantiate and bind application objects by providing a service specification together with QoS requirements. A service specification is a composition of sub-service types, each with potentially many implementations. QoS requirements are specified by an object that provides a utility function. A QuA *utility function* is a function that returns 1.0 when error does not reduce the value of the service and 0.0 when error renders the service of no value. The input to the utility function is the measurement (or prediction) of error according to some error model. The QuA Platform is responsible for discovering configuration alternatives and for implementing a configuration with the requested QoS. Service implementation brokers may provide service offers that are input into the utility function to support QoS-driven planning. Alternatively, service implementations may provide dynamic QoS management components that use the utility function for service monitoring and adaptation.

Continuing the thermometer example from the previous section, a client wanting to read the temperature on a remote machine would request a thermometer service whose `get` interface is bound to the client's thermometer capability and whose `set(temp)` interface is bound to the remote machine environment. The platform is free to discover a software component implementing the thermometer service type and to deploy this component wherever it can obtain computing resources. The platform is also free to implement the bindings between client, thermometer, and environment with any compatible binding types and protocols.

Because the thermometer client requested a service bound to a remote environment, the implementation of the thermometer service must include a remote messaging sub-service. If a remote messaging service offers predictable delay, the platform service planner may be able to negotiate a service contract to ensure that end-to-end delay requirements are met. If a remote messaging service offers dynamic QoS management, the platform service planner can allocate a portion of the acceptable end-to-end delay as a QoS goal for the remote messaging service component.

It is important to note that a remote messaging service may not only introduce delay between a `get` event and the associated `return(temp)` event, but

also between a `set(temp)` event and the time that the thermometer stores this value. This can cause the temperature returned to lag behind the ideal temperature independent of the response time for a `get` operation. Further, the accuracy of the temperature returned may be degraded by round-off error and bandwidth reduction, e.g., dropping some of the `set(temp)` messages. If we want the middleware to independently manage the level of tolerance for each of these implementation variables, then we need a more expressive error model with error components like *temperatureLag* and *precision*.

The QoS of the composition of remote messaging sub-services with the implementation of the thermometer itself may be analyzed just as we would analyze the composition of the thermometer service with other application level services. A remote messaging service that communicates `get` messages has only `get` events as inputs and only `get` events as outputs. For this service, we would want an error model that described only delay and loss of messages. For a remote messaging service that communicates `set(temp)` messages we would want an error model to also include loss of precision.

For a given composition of services, we can reason about how error derives from the component services. This reasoning requires knowledge of the error model semantics for both the components and the composition. For example, loss of a `set(temp)` message contributes to *temperatureLag* in the composition since a subsequent `return(temp)` will have to return an older value. We believe that the well defined semantics of our error models will facilitate similar reasoning about arbitrary QoS characteristics and service compositions.

4 Related Work

There has been little work published specifically addressing QoS models for component architectures. Researchers at BBN developed QuO (Quality of Service for Objects) [8] as a framework for management of QoS properties in CORBA applications. A recent paper introduced *goskets* as a means for reusing adaptive QoS behaviors, but they have not yet adapted this work to a component architecture [9].

Dynamic approaches to QoS management often include QoS negotiation that seeks to maximize a

worth function [10] [11] [12]. We use a general model of a multi-dimensional worth function and extend this by outlining a way to relate its definition to components in the application design. As part of a comprehensive initiative from OMG to support real-time systems, OMG is pursuing a UML profile for QoS specification [13]. We are actively involved in this initiative and the approach outlined in this paper is one way of using the UML QoS specifications in a QoS management architecture.

5 Conclusions

In this paper we have extended earlier work on QoS for multimedia presentations to create a general model and semantics for QoS specification. The key features of the model are a definition of a service that is based on a composition of logical component behavior and a definition of quality dimensions based on a metric space of service behavior. The model supports mapping between QoS behavior of a service and the QoS behavior of the component services that implement it. The model allows quality specifications to use domain-specific error models to characterize quality dimensions.

References

- [1] Qos enabled distributed objects. <http://qedo.berlios.de>.
- [2] I. Foster, C. Kesselman, J. Nick, S. Tuecke. Grid Services for Distributed System Integration. *Computer*, 35(6), 2002.
- [3] G. Coulson, G. S. Blair, M. Clarke, N. Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15:109–126, 2001.
- [4] J. P. Loyall, R. E. Schantz, J. A. Zinky, D. E. Bakken. Specifying and Measuring Quality of Service in Distributed Object Systems. In *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98)*, pages 20–22, Kyoto, Japan, 1998.
- [5] I. Pyrali, D. Schmidt, R. Cytron. Achieving End-to-End Predictability of the TAO Real-time CORBA ORB. In *Proceedings of the 8th IEEE Real-Time Technology and Applications Symposium*, San Jose, CA, 2002.
- [6] Richard Staehli, Frank Eliassen. QuA: A QoS-Aware Component Architecture. Technical Report Simula 2002-13, Simula Research Laboratory, 2002.
- [7] R. Staehli. *Quality of Service Specification for Resource Management in Multimedia Systems*. PhD thesis, Oregon Graduate Institute of Science & Technology, 1996.
- [8] J. A. Zinky, D. E. Bakken, R. E. Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3, 1997.
- [9] R. Schantz, J. Loyall, M. Atighetchi, P. Pal. Packaging Quality of Service Control Behaviors for Reuse. In *Proceedings of ISORC 2002, The 5th IEEE International Symposium on Object-Oriented Real-time distributed Computing*, Washington, DC, 2002.
- [10] Koistinen, J. and Seetharaman, A. Worth-Based Multi-Category Quality-of-Service Negotiation in Distributed Object Infrastructures. In *Proceedings of Second International Enterprise Distributed Object Computing Workshop (EDOC '98)*, pages 239–249, San Diego, CA, USA, 1998.
- [11] Chatterjee, S., Sabata, B. and Sydir, J. J. ERDoS QoS Architecture. Technical Report ITAD-1667-TR-98-075, SRI International, Manlo Park, CA, 1998.
- [12] J. Walpole, C. Krasic, L. Liu, D. Maier, C. Pu, D. McNamee, D. Steere. Quality of Service Semantics for Multimedia Database Systems. In *Proceedings Data Semantics (DS.8): "Semantic Issues in Multimedia Systems"*, Rotorua, NZ, 1999.
- [13] UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms, Request for Proposal, OMG Document: ad/2002-01-07, 2002.