A QoS-Control Architecture for Object Middleware*

L. Bergmans[1], A. van Halteren[2], L. Ferreira Pires[1], M. van Sinderen, and M. Aksit[1]

[1] CTIT, University of Twente, The Netherlands
{bergmans, pires, sinderen, aksit}@ctit.utwente.nl
[2] KPN Research, Enschede, The Netherlands
A.T.vanHalteren@kpn.com

Abstract. This paper presents an architecture for QoS-aware middleware platforms. We present a general framework for control, and specialise this framework for QoS provisioning in the middleware context. We identify different alternatives for control, and we elaborate the technical issues related to controlling the internal characteristics of object middleware. We illustrate our QoS control approach by means of a scenario based on CORBA.

1. Introduction

The original motivation for introducing middleware platforms has been to facilitate the development of distributed applications, by providing a collection of general-purpose facilities to the application designers. Currently, commercially available middleware platforms, such as those based on CORBA, are still limited to the support of best-effort Quality of Service (QoS) to applications. This constitutes an obstacle to the use of middleware systems in QoS critical applications, or in case services are offered in the scope of Service Level Agreements with strict QoS constraints. This limitation in the available middleware technology has inspired much of the research that is currently being done on QoS-aware middleware platforms.

Ideally, a middleware platform should be capable of supporting a multitude of different types of applications with (a) different QoS requirements, (b) making use of different types of communication and computing resources, and (c) adapting to changes, e.g., in the application environment and in the available resources. The architectural framework presented in this paper has been developed to be flexible and re-usable. The main benefit of our framework is that it allows us to combine and balance solutions for the control of multiple QoS characteristics.

From the research perspective, a framework-based approach supports the incremental introduction of new solutions such as control algorithms, and allows us to compare different solutions in the same setting. From a middleware developer's perspective, this approach is attractive because it supports incremental development and the construction of product families in which different family members address different sets of QoS characteristics.

* This work has been carried out within the AMIDST project (http://amidst.ctit.utwente.nl/).

H. Scholten and M. van Sinderen (Eds.): IDMS 2000, LNCS 1905, pp. 117-131, 2000.

This paper identifies the problems that have to be solved in order to elaborate our architectural framework, and discusses some techniques that can be used to solve these problems. The use of the architectural framework is illustrated by means of a scenario.

This document is further structured as follows: Section 0 introduces some basic concepts and discusses the role of a QoS-aware middleware when supporting object-based applications; Section 0 introduces our approach and its background, which stems from control theory; Section 0 discusses the technical issues that have to be addressed to realise the proposed framework, presents some requirements for each of these issues and indicates our solutions to fulfil these requirements; Section 0 illustrates the use of our architectural framework with a simple CORBA application using a naming service, and shows how QoS requirements can be enforced by the middleware platform; and Section 0 draws our conclusions.

2. Concepts of QoS-Aware Middleware

This section discusses the concepts that underlie our QoS-aware middleware architecture and identifies the role of a QoS-aware middleware platform in the support of distributed applications.

2.1 Distributed Applications

Distributed applications supported by a QoS-aware middleware consist of a collection of interacting distributed objects. Since in this paper we concentrate on the support of *operations* (invocations of methods), we assume that an object may play the role of either a client or a server on an interface. We also assume the ODP-RM computational model, in which objects may have multiple interfaces [7].

During the development of a distributed application, the interfaces of the application objects have to be specified. In principle this specification should define the attributes and operations of these interfaces. In the case of CORBA, one only specifies the server interface using IDL, and makes use of this specification for creating stubs and skeletons, or to dynamically create requests for operations. However, in general one could specify server and client interfaces, and define rules that can determine whether a server interface is capable of servicing a client interface [14, 1]. Extensions of IDL that allow the definition of both client (required) and server (offered) interfaces have already been proposed in the CORBA component model [13].

When considering QoS-aware middleware, we suppose that the interface specifications are extended with statements on QoS that can be associated with the whole interface or with individual operations and attributes. In the case of a client interface, these statements describe the required QoS, while for a server interface these statements describe the offered QoS. QML [4] and QuO [17] are languages that allow one to specify the QoS associated to interfaces.

2.2 Objects Life-Cycle

After the objects of a distributed application have been implemented, the application is deployed. We consider the general case in which persistent objects and late binding can be supported by the middleware platform. In this case, an object has the following life cycle:

1. Object creation, in which interface references for the server interfaces of an object are created and can be referred to by other objects;
2. Object activation, in which an object starts execution, which implies that all local resources necessary for the object to execute should be properly allocated;
3. Object deactivation, in which local resources allocated to an object may be released, although the interface references may still be valid in case persistent objects are supported;
4. Object destruction, in which the object is deactivated (if it is still active) and its interface references are destroyed.

A QoS-aware middleware platform can use object activation to refine the offered QoS, by restricting the ranges originally described for the offered QoS at design and implementation time. The run-time status of the middleware platform and the communication and computing resources should make it possible to determine this offered QoS more precisely.

2.3 Explicit Binding

Object interfaces have to be bound to each other in order to allow these objects to interact through the middleware. In CORBA, this binding happens implicitly when the client object issues a request (implicit binding).

For QoS-aware middleware platforms, however, implicit binding is not desirable, since the QoS requirements may demand that resource allocation procedures are performed before the request is executed. Unfortunately, we can not predict the speed and reliability of these procedures. In the worst case, we may still have to activate the server object. This means that we can not always guarantee the QoS requirements by using implicit binding. Therefore,in QoS-aware architectures *explicit binding* is necessary, which consists of taking explicit actions at the computational level in order to establish the binding before interacting [7]. Our case for explicit binding for QoS-aware operation support is somewhat similar to the reasoning in [1] for stream interface bindings.

The client object requests the establishment of the binding, giving to the middleware a reference to a server interface. This request also contains the required QoS, which can be retrieved from a QoS specification repository. The middleware platform then searches for the server object. In case the server object has not been activated, the middleware platform activates this object and continues the establishment procedure. Otherwise, the middleware platform compares the offered QoS with the required QoS and uses its internal information to determine an agreed QoS. This process is called *QoS negotiation*. In case the binding establishment has been successful, the client and server objects are informed that a binding has been built. From this moment on these objects can interact through the binding. Figure 1 shows the establishment of a binding using a QoS-aware middleware.

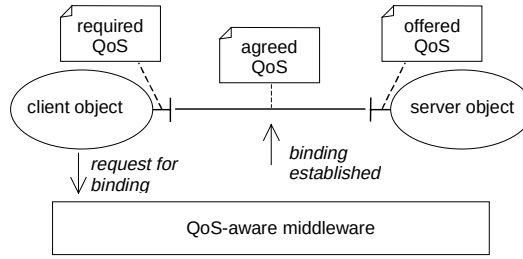

Fig. 1. Binding establishment using a QoS-aware middleware platform

The agreed QoS is determined by considering the required QoS on one hand, and the composite QoS capabilities of the server object (the offered QoS) and the middleware platform on the other hand. The agreed QoS serves as a *contract* between the application objects and the middleware platform, which should be respected during the operational phase when the objects interact through the binding.

The binding establishment may also result in the creation of a binding object. This object binds the client object and the server object, and offers a control interface that allows, for example, the inspection and modification of the agreed QoS. In this paper we ignore the adjustment of the agreed QoS through such an interface, but this is an interesting topic for further work.

Our QoS control approach considers that a binding has been successfully established and that the agreed QoS has to be maintained. The middleware is responsible for that, and is constantly adjusting its internal characteristics and the usage of computing and communication resources in order to achieve it.

3. Control Framework for QoS Provisioning

This section introduces our approach and subsequently discusses a specialisation of a generic control system model for the purpose of controlling QoS in a middleware context.

3.1 Approach

The design of our QoS-aware middleware architecture is constrained by two conflicting requirements: a) the architecture has to be flexible enough such that it enables us to experiment with different QoS strategies and cope with different kinds of application demands; and b) certain aspects of the architecture have to be fixed so that the robustness and portability of the architecture can be guaranteed.

For this reason we start off with a generic control system model, which we specialise, such that it applies to QoS-control in a middleware context. This specialised model forms our architectural framework, i.e. the fixed part of our architecture. Although some decisions are made with respect to the scope of control, the architectural framework is independent of any specific QoS-control strategy or algorithm. Therefore, different solutions can be compared and evaluated with this framework.

A synthesis-based approach [16] can be used to arrive at a complete QoS-control architecture, e.g. for a specific application or system environment. In this approach, requirements are converted into technical problems. For each technical problem, possible solution techniques are sought. The candidate solution techniques are then compared with each other from the perspective of relevance, robustness, adaptability and performance. Whenever a suitable solution technique is found, the fundamental abstractions of this technique are used to derive the architectural abstractions. This process is repeated until all the problems are considered and solved. Finally, the architectural abstractions are specified and integrated within the overall framework. Since solution domain knowledge changes smoothly, this approach provides us with stable and robust abstractions with rich semantics. The discussion of technical issues in Section 4 partially illustrates this approach.

3.2 Generic Control System

The main objective of QoS-aware middleware is to establish and enforce an agreed QoS that satisfies the demands of applications, given the available resources. We observe this is essentially a controlling problem, and therefore the QoS-control framework should be synthesised from the fundamental abstractions of control systems.

A *control system* [3, 8] consists of a *controlled system* in combination with a *controller*. The interactions between the controlled system and the controller consist of observation and manipulation performed by the controller on the controlled system. The building blocks of the control process are shown in figure 2:

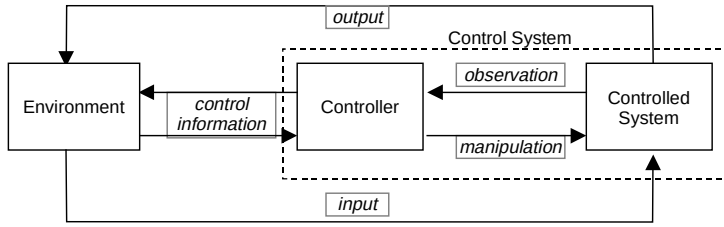

Fig. 2. Building blocks of a control process.

The generic control model abstracts from the type of observation and the type of manipulation that can be employed by the controller on the controlled system. The relationship between the controlled system and the controller can be realised using different strategies. With a *feed-forward control* strategy, manipulation through control actions is determined based on manipulation of the input to the controlled system. A *feed-back control* strategy can be applied for behaviour optimisation. According to this strategy, measurements of the output delivered by the controlled system are compared with a desired behaviour (a *reference*) and the *difference* between them is used by the controller to decide on the control actions to be taken.

3.3 QoS-Control System

In QoS-aware middleware, the 'controlled system' is the middleware functionality responsible for the support of interactions between application objects, while the 'controller' provides QoS control. Here, the environment represents the operational context of the middleware, which consists of application objects with QoS requirements and QoS offers. The middleware platform encapsulates the computing and communication resources at each individual processing node, which may be manipulated in order to maintain the agreed QoS.

Figure 3 shows the specialisation of the generic control model for controlling the QoS provided by a middleware.

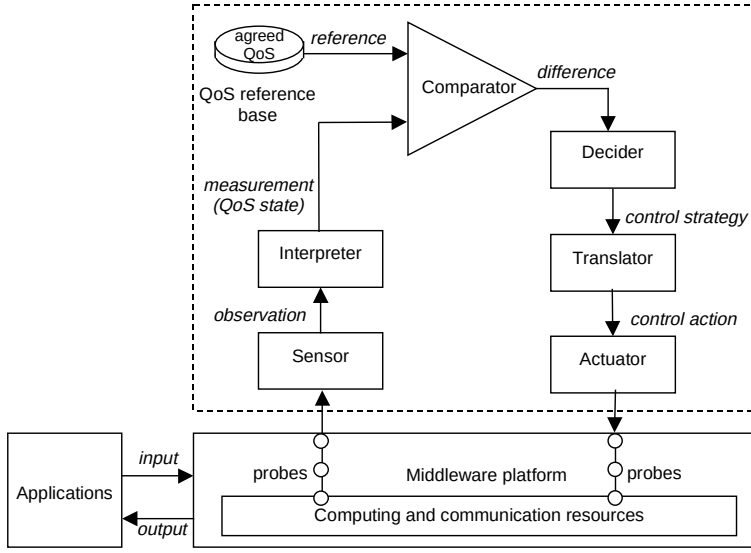

Fig. 3. QoS-control architecture.

In Figure 3 we identify two symmetrical structures, one for handling QoS measurement concerns and another for handling QoS manipulation concerns. A *probe* is a point of observation or manipulation that is available or must be planted in the controlled system, i.e., the middleware platform. Many probes may be planted in the controlled system, for both observations and manipulations.

A *sensor* is a mechanism that uses a probe to obtain observations. Observations can only be useful if they are interpreted in terms of measurements that can be compared with the reference, i.e., they are represented using the same units and have the same semantics. For example, observations can be time moments of the sending of a request and the receiving of the corresponding response. The needed measurement could be the average response time, which implies that the average of the difference between the time moments observed should be calculated in order to generate the measurement. This calculation is performed by an *interpreter*. In general, the interpreter combines observations, which could even come from different sensors, in order to generate measurements.

A *comparator* compares the measurement and an associated reference value (an agreed QoS value), determining the difference. A *decider* gets the difference and applies some algorithm to establish a control strategy, consisting of the objectives to be reached in this execution of the control loop. The control strategy must be translated in a collection of control actions, i.e., manipulations of the controlled system. A *translator* is responsible for translating the control strategy to a collection of control actions. An *actuator* schedules the control actions such that they are carried out using one or more probes. The translator distributes the control actions among the actuators, realising in this way the control strategy.

4. Technical Issues

This section identifies and elaborates a number of the technical issues that have to be addressed in order to realise the QoS control architecture. The first 5 issues correspond to the realisation of the components in our architecture. We explain the requirements, and propose some possible solutions and solution strategies.

4.1 Collecting Observation Values

In order to collect observation values we have to develop probes and sensors. Probes connect the middleware to the control mechanisms and are independent of the actual measurements. Sensors collect the actual measurements and they typically depend on the amount and types of data that are collected.

The fundamental requirement on the probes and sensors is that they must have a minimal impact on the middleware platform. This introduces two issues: (a) how to minimise the impact on the middleware code, and (b) how to map the probes to one or more specific places in the middleware code (cross-cutting problem [9]).

Reflection is a technique in which a system is explicitly represented in terms of a meta-object, allowing one to manipulate the (structure of the) system by manipulating its meta-object. A reflection-based approach suits well to the collection of observation values.

Crosscutting of concerns requires either careful documentation and management of probe insertion points, or entirely new tools and techniques for specifying and implementing crosscut concerns. Recent work in the area of Aspect-Oriented Programming (see, e.g., [9]) addresses these issues.

4.2 Interpretation of Observations

The interpretation process depends on many factors: the involved observation data, the required measurements, and the rules or strategies for interpretation. The number of interpretation rules and their complexity also determine the interpretation process.

The interpretation part should not become a possibly large collection of unstructured ad-hoc code. This implies that a generic model should be developed to define how observations are translated to measurements, such that interpretation code can be reused or generated automatically as much as possible. In case statistical

information determines measurements, a lot of input data may be required, such that the amount of storage and processing should be as much as possible reduced.

The interpretation process is essentially a transformation from a set of input values to a set of output values. The variation in input values lies both in sources, types and time, and depends on the sources of the input, i.e., how the middleware has been designed. The resulting output should be independent of the specific implementation details of certain middleware and applications, and it should be suitable for the comparison process. This means that a common *QoS meta model* should be available that determines the types and values of both measurements and the references.

The interpretation of observations can be done through calculations, heuristics (logic rules), stream interpreters or conversions. We need to model these different techniques in a uniform way, with explicit dependency relations to a structured representation of the observations and measurements. Interpretation rules should all be a specialisation of a single abstraction, i.e., the interpreter. Each individual instantiation can be considered as a *micro-interpreter*. For each QoS measure, there should be a clear specification of the interpretation rules in terms of formulas or guidelines. In Figure 4 at the end of this section the extensions to our architecture to meet the needs of interpretation are shown.

4.3 Determination and Representation of the Difference

The comparator compares the measurement with the reference model and determines the difference. This comparison can vary from subtraction in the simple case of one QoS characteristic with a numeric value, to complex calculations possibly using heuristics in the case of multi-faceted QoS characteristics. The main task of the comparator is to deliver an abstraction of the 'problem to be solved' that is as far from the implementation details of the environment as feasible.

The difference produced by the comparator serves to detect (potential) violations of the QoS. Such violations depend on the agreed QoS. Hence, the difference must be obtained by comparing the actual measurements with corresponding references specified by the agreed QoS.

The difference could be represented as a 'distance' vector, where each element of the vector corresponds to a relevant QoS characteristic.

Measurements and references should be described in such a way that they can be compared (see Section 0). For this purpose we use a QoS meta-model, which consists of a collection of concepts that allow one to specify both the measurement and the reference, and the difference. Another benefit of having a QoS meta-model is the ability to build QoS specification repositories. We adopt and adapt QML [4] to specify the QoS meta-model and its instantiations.

Figure 4 illustrates the use of the QoS meta-model in our overall architecture. The agreed QoS is determined before entering the operational phase, through negotiation based on QoS requirements, QoS offers and the capabilities of the middleware platform. In this paper we assume that the agreed QoS is not modified during the operational phase.

4.4 Controlling Algorithm

The difference or distance vector computed by the comparator may –or may not– define a situation that requires controlling (i.e., correcting) actions to be taken. The controlling algorithm is responsible for selecting an appropriate strategy. The strategy to be chosen depends partially on the specific state and configuration of the middleware. Rather than mixing middleware state and configuration information with the measurements and difference, this information must be available independently. For this reason, we introduce a *middleware control model*. This model is an abstraction (model at a meta-level) of the middleware, which specifies what can be parameterised or tuned in the middleware, or which components can plugged in, deactivated and activated.

The task of the decider is two-fold: firstly to ensure that the agreed QoS can indeed be supported by the middleware platform, and secondly to optimise the overall QoS characteristics, by balancing the different, often contradictory, requirements. In its most general form, controlling is an artificial intelligence task that involves domain knowledge and heuristics about managing and controlling QoS, and the interdependencies between QoS characteristics.

We have not selected a particular solution for the controlling algorithm: our goal is to offer a framework that allows the experimentation with –combinations of– different techniques such as mathematical algorithms, heuristic rules and the use of fuzzy logic as a means of expressing and reasoning about weak but conflicting optimisation [12]. Figure 4 shows the extension of our architecture with an explicit middleware control model.

4.5 Control Strategy and Middleware Manipulation

A control strategy is the output of the controlling algorithm, and it should be an implementation-independent representation of the solution strategy for pursuing certain QoS characteristics. Control strategies are strongly related to the controlling algorithm.

Control actions are abstractions that represent concrete functional behaviour, but are independent of the implementation details of the specific middleware software. Control strategies represent sets of control actions that are to be applied to the middleware in a co-ordinated way. The representation of control strategies must consist of at least the following parts: a) set of control actions; b) a set of probes in the middleware where the control actions can be applied, and c) a co-ordination specification, which could be a script or any other form of executable specification.

There are a few ways to affect the behaviour of a running system like a middleware platform: a) by invoking operations of a local API; b) by modifying the internal state of the system, c) by replacing components of the system with different implementations, and d) by meta-level manipulation of the system itself. A control action can only be a specialisation or instantiation of one of these.

The implementation of control actions through actuators and probes introduces technical issues comparable to the ones discussed in section 0, and therefore they are not discussed further.

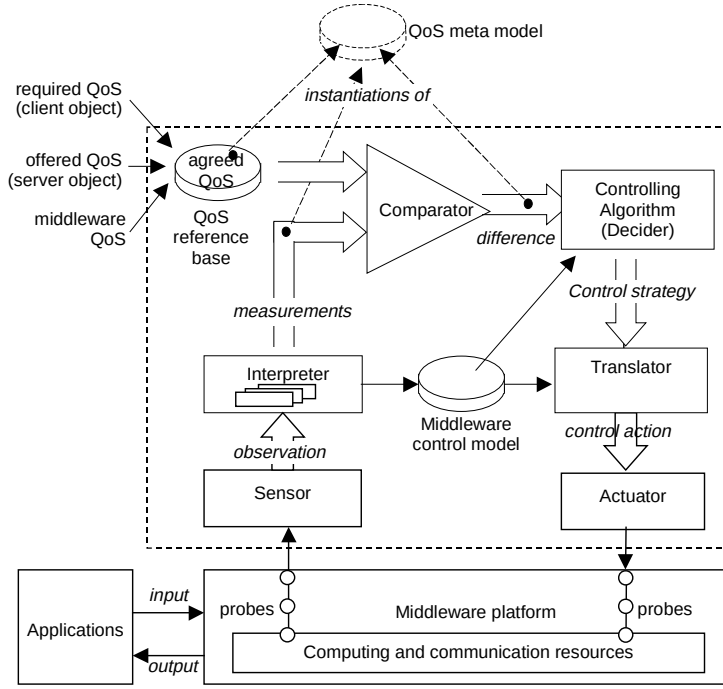

Fig. 4. A detailed version of the architecture incorporating some of the enhancements that are discussed in this section.

4.6 Feasibility of the Overall Control Loop

The performance overhead introduced by our architectural framework has to be carefully considered when using the framework in practical settings. The technical solutions should not make the overall QoS worse than what it would be without them. Several QoS requirements are related to performance (e.g., delays and throughput). Implementations of our architecture may require a lot of additional activities and overhead, which may conflict with the QoS requirements they try to enforce. By adopting a tailorable framework approach, we may choose to build instances of the framework with components ranging from simple, low-overhead components up to complex components. This approach can help coping with the performance overhead by using more efficient versions wherever necessary. In the future, the use of a meta-controller to switch dynamically between different versions may be considered.

Feed-back control loops may make the controlled system oscillate between two undesirable states, depending on the corrective measures and their effects. In some cases, mathematical models based on control theory can help predicting whether the system is stable during operation, allowing one to avoid oscillation. In case mathematical models are not available or are not precise enough, some heuristics may show whether the system is stable or not. Alternatively, additional (meta-level) controllers could be introduced to detect instability and take measures to avoid it, e.g., by actuating on the controlling algorithm. The use of fuzzy logic in the controlling algorithms may also help to avoid that the control loop oscillates during operation.

5. Scenario

This section demonstrates our architectural framework by means of a scenario for a simple CORBA application using a naming service.

5.1 Scenario Set-Up and Use of QML

Figure 5 depicts a QoS provisioning scenario for a CORBA naming service application. The application consists of a client object that intends to invoke a method on a `NamingContext` object through a QoS-aware ORB.

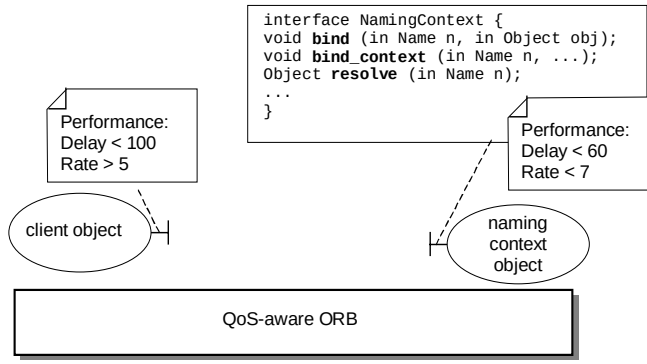

Fig. 5. A naming service application scenario.

The offered QoS of the server object and the required QoS of the client object are depicted in a simplified form:

- the client object requires a *delay* (time necessary to complete a request) smaller than 100, and a supported *rate* (number of requests per time unit) of at least 5;
- the `NamingContext` object offers a delay smaller than 60, and a supported rate up to 7.

We use QML [4] to express the required or offered QoS of an object. The QoS is specified using the QoS dimensions of a QML *contract type*. Figure 6 shows a possible QML contract type that defines the relevant QoS characteristics in this scenario, viz. delay and rate.

```
type PerformanceType = contract {
  delay : decreasing numeric msec;
  rate: increasing numeric req/sec;
};
```

Fig. 6. A QML contract type

```
PerformanceType contract {
  delay < 60; //maximum delay
  rate  >  7; //minimum rate
};
```

Fig. 7. A QML contract

QoS contract types should be defined by the QoS meta-model discussed in Section 4. The QoS that an application object requires or offers is specified by a QML *contract*. A QML contract puts constraints on the dimensions of the corresponding QML contract type. Figure 7 shows a possible instance of the QML contract type of Figure 6. This contract corresponds to the offered QoS of the naming context object in Figure 5; the required QoS of the client object can be defined in a similar way.

5.2 Binding Establishment

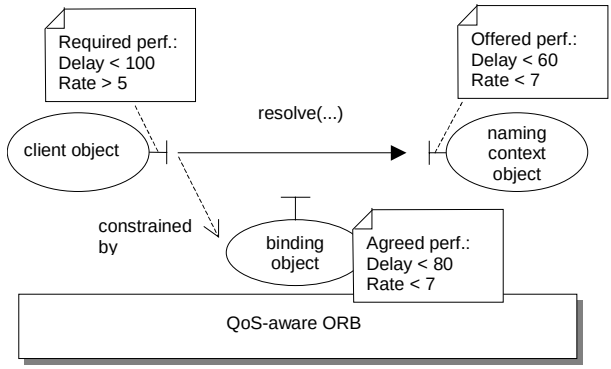

Fig. 8. An established binding with an agreed QoS.

During the binding establishment phase, the client object requests the establishment of a binding with the `NamingContext` object. Whether such a binding is successful depends on whether the application object and the middleware platform together can satisfy the required QoS of the client object. If so, an agreed QoS is established and the middleware should take appropriate actions, such as:

- update of the QoS reference base (see Figures 3 and 4);
- instantiation of sensors that can be used for measurements during the operational phase;
- instantiation of actuators and/or other configuration settings to prepare support for the agreed QoS. For example, when a configurable transport protocol is used, a connection with certain characteristics may be set up.

Figure 8 depicts a possible result of a successful binding establishment phase, where a QoS is agreed and a binding object is created that is aware of the agreed QoS. The offered QoS of the middleware in this case could be an (added) delay less than 20, and a supported rate less than 100. For the QoS characteristics in this scenario, the constraints on the negotiation process that led to the agreed QoS are as follows:

- delay: $QoS_{offered}(server) + QoS_{offered}(middleware) \leq QoS_{agreed} \leq QoS_{required}$
- rate: $QoS_{required} \leq QoS_{agreed} \leq minimum(QoS_{offered}(server), QoS_{offered}(middleware))$

5.3 Operational Phase

During the operational phase, the client object invokes requests and obtains replies from the `NamingContext` object. Invocations may trigger measurements or the installation of timers to measure the actual offered QoS. If the measured QoS approaches certain thresholds related to the agreed QoS stored in the QoS reference base, the control system may take precautions, either proactively or reactively. Proactive control actions are taken when a "danger zone" has been entered, but before a violation has been detected on the agreed QoS; reactive control actions are applied if a violation of the agreed QoS has been detected. Control actions include scheduling of a request on a high-priority thread, transmitting the request over a transport network with priority routing, installing a protocol plug-in that takes advantage of the network QoS by differentiating between the priority of network packets, or optimising delivery to multiple recipients through a multi-cast protocol.

For example, the handling of QoS during the operational phase, where (for reasons of space) we focus on delay, involves the following steps:

- To determine the actual delay, we can measure the time that elapses between the sending of a request and the receipt of a corresponding reply. *Sensors* are responsible for collecting this information from relevant *probes* (e.g., timer, interceptor) planted in the middleware platform.
- The *interpreter* translates the observations received from the sensors into measurements that can be usefully compared to the agreed delay (reference) value that was established for this binding. For example, the delay observations in a certain time period may be used to compute an (average) delay measurement.
- The *comparator* compares the measurement values with the corresponding reference value. The result could be an element in a distance vector:

  $<\Delta Delay, \Delta Rate>$, where $\Delta Delay = Delay_{reference} - Delay_{measured}$

  In a concrete case, we may have a delay 'distance' 20; this means that the actual offered delay has reached 80% of the maximum allowed delay.
- The *controlling algorithm* must decide, based upon the difference values, and other state information, how to deal with the situation. In this case, we assume that over 80% of the maximum delay is the 'danger zone', which requires specific actions to speed up the transport of the request/reply messages. A suitable *control strategy* that may improve the delays, is the activation of a faster/prioritised transport protocol (e.g., RSVP [6]).
- The *translator* uses state and configuration information from the *middleware control model* to determine the availability of the appropriate transport protocol, and the location where it should be plugged-in. The resulting control actions consist of plugging in the protocol on the client side, plugging in the protocol on the server side, and setting the priorities upon this transport plug-in.
- The control actions are performed by the *actuator*, which needs to access the middleware software for plugging in the protocols, and must perform the right priority/delay settings for each of the protocol instantiations. The *probes* used by the actuator can be APIs and/or global variables.

The steps that have been described here are performed repeatedly, either triggered by an internal clock, or by events (timers that expire, requests that are sent or received, etc).

6. Conclusion

In this paper we have presented an architecture to support QoS-aware middleware. We have introduced some assumptions and general concepts for using QoS-aware middleware. The key part, and focus of this paper, is the *QoS-control system*. The QoS controller in this system observes and, if necessary, manipulates the state of the controlled system, i.e. the middleware platform that supports distributed applications. The design of the QoS controller is an architectural framework that is based on models from control theory. This should ensure its stability with respect to evolving requirements, and its applicability to a wide range of controlling techniques.

The QoS-control architecture was discussed in more detail by examining a number of technical issues that must be addressed when realizing the proposed architecture. For each of these issues, we discussed requirements and corresponding solutions or solution approaches. We illustrated our proposal by describing a simple example of an application with QoS requirements, and how this would be dealt with in the proposed architecture.

An initial proof of concept of our approach has been performed in [5]. The prototype, based on the ORBacus implementation of CORBA, measures the QoS by using Portable Interceptors during system operation, and controls QoS at the transport level. Control actions are performed through a pluggable transport protocol that prioritizes IP packets using DiffServ [10] features.

In the paper, we hinted at several topics for interesting future work. These topics address the further development and prototyping of our control architecture, as well as exploring controlling strategies and algorithms that could not be considered so far. In addition, we like to profit from results of related works:

- One of the characteristics of our proposal is that the architecture is largely independent of the specific implementation architectures of middleware systems. The QoS controller is separate from the middleware (and applications) and may interact with these through a number of *probes* (a generic term for interfaces that abstracts from specific implementations). Conceptually (and possibly implementation-wise), this is a reflective model; our QoS controller observes and manipulates the middleware at a meta-level. Several other proposals for reflective middleware have been made, e.g. [2].
- A middleware framework for QoS adaptation has been described in [11]. Both a task control model and a fuzzy control model have been used in this framework to formalise and calculate the control actions necessary to keep the application QoS between bounds. This framework shares many design concerns with our framework, although it has been targeted to the control of applications.
- OMG currently develops Real-time CORBA facilities in the scope of the CORBA 3.0 standard [15]. These facilities allow one to manipulate some middleware characteristics that influence the QoS, such as, e.g., the properties of protocols underlying the ORB and the threading and priority polices applied to the handling of requests by server objects. These facilities are defined in terms of interfaces that have to be implemented in the middleware platform, generalising in this way the control capabilities of the platform.

7. References

[1] G. Blair, J.-B. Stefani. *Open distributed processing and multimedia*. Addison-Wesley, 1998.

[2] G. Blair, C. Coulson, P. Robin, M. Papathomas. An architecture for next generation middleware. In: Procs. of *Middleware 2000*, 191-206. Springer-Verlag, London, 1998.

[3] A.C.J. de Leeuw. *Organisaties: management, analyse, ontwerp en verandering -een systeemvisie*. Assen/Maastricht, Van Gorcum, 1990.

[4] S. Frølund, J. Koistinen. *Quality-of-service specification in distributed object systems*. Techn. report HPL-98-159. HP Labs, Palo Alto, USA, 1998.

[5] M. Harkema. *A QoS Provisioning Service for CORBA*. MSc thesis, Univ. of Groningen, Groningen, The Netherlands, 1999.

[6] IETF. *Resource ReserVation Protocol (RSVP). Version 1 Functional Specification*. IETF RFC 2205, Sept. 1997.

[7] ITU-T | ISO/IEC. *ODP Reference Model Part 1. Overview* (ITU-T X.901 | ISO/IEC 10746-1). May 1995.

[8] W.J.M. Kickert, J.P. van Gigch. A metasystem approach to organisational decision-making. In: J.P. van Gigch (ed.), *Decision making about decision making: meta-models and metasystems*, 37-55, Abacus Press, 1987.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin. Aspect-Oriented Programming. In: Procs. of *ECOOP '97*, Springer-Verlag LNCS 1241, June 1997.

[10] K. Killki. *Differentiated services for the Internet*. Macmillan Computer Publishing, 1999.

[11] Baochun Li, Klara Nahrstedt. A control-based middleware framework for quality of service adaptations. *IEEE J. on Sel. Areas in Comms*. Vol. 17, No. 9, 1632-1650, Sept. 1999.

[12] E.H. Mamdani and S. Assilian. An experiment in linguistic synthesis with a fuzzy logic controller. *Intl. J. of Man-Machine Studies* 7, 1-13, 1975.

[13] OMG. CORBA components. OMG TC Document orbos/99-02-05. March 1999.

[14] C. Szyperski. *Component software*. Addison-Wesley, 1998.

[15] D.C. Schmidt, F. Kuhns. An overview of the real-time CORBA specification. To appear in: *IEEE Computer* special issue on Object-oriented real-time distributed computing, June 2000.

[16] B. Tekinerdogan. *Synthesis-based software architecture design*. PhD thesis. Univ. of Twente, Enschede, The Netherlands, 2000.

[17] R. Vanegas, J.A. Zinky, J.P. Loyall, D. Karr, R.E. Schantz, D.E. Bakken. QuO's runtime support for quality of service in distributed objects. In: Procs. of *Middleware 2000*, 207-222. Springer-Verlag, London, 1998.