DESIGN AND EVALUATION OF A QOS PROVISIONING SERVICE

A.T. van Halteren, G. Fábián, E. Groeneveld
KPN Research, PO Box 96, 7500 AB Enschede, The Netherlands.
{a.t.vanhalteren,g.fabian} @kpn.com
SERC, P.O. Box 424, 3500 AK Utrecht, The Netherlands
erik@serc.n

Abstract Middleware provides distributed objects with a software infrastructure that offers a set of well-known distribution transparencies. These transparencies enable the rapid introduction of applications for heterogeneous, distributed systems. However, to support guaranteed Quality of Service (QoS) system-specific QoS mechanisms need to be controlled. Accessing the low-level mechanisms directly by applications crosscuts the transparency offered by the middleware and limits portability and interoperability. The challenge for next-generation middleware is to support application-level QoS requirements, while maintaining the advantages of the distribution transparencies. This paper presents three contributions: (1) An architecture for a QoS-aware software infrastructure for distributed objects (2) A framework for a QoS provisioning service (QPS) and (3) An evaluation of the QPS framework by means of a prototype that supports performance

Keywords: QoS control, adaptive middleware, framework, prototyperequirements.

1. INTRODUCTION

Middleware is gaining wide acceptance as a generic software infrastructure for distributed applications. A growing number of applications are designed and implemented as a set of collaborating objects using object middleware, such as CORBA, EJB and (D)COM(+), as a software infrastructure that facilitates distribution transparent interactions. However, quality aspects of these interactions cannot be specified nor enforced by current object middleware, resulting in a best-effort QoS.

In order to support QoS sensitive applications, system-specific QoS mechanisms such as OS scheduling mechanisms and network reservation mechanisms need to be controlled. Allowing applications to directly access and control these mechanisms would crosscut the distribution transparencies offered by the mid-

dleware layer and would reduce the portability and interoperability of distributed object applications.

To avoid this, next generation object middleware should offer abstractions for management and control of the system level QoS mechanisms. These abstractions should take into account that new interfaces to OS resources and new network protocols are expected to appear as the result of ongoing research efforts. In addition, a changing run-time environment must be handled, such as system and network load that influences the QoS capabilities.

In summary, the architecture of next generation middleware has to meet the challenge of (1) evolutionary changes of the system level QoS mechanisms and (2) run-time dynamic changes of the environment.

1.1. Paper structure

This paper is organised as follows. Section 2 describes an architecture for positioning QoS support in open distributed systems. Section 3 gives an overview of the requirements on a middleware-based software infrastructure that offers QoS support to distributed objects. Section 4 presents our solution in the form of an infrastructure service for QoS provisioning, which is an extensible service for making middleware QoS-aware. Section 5 evaluates this framework and gives an overview of our implementation. Section 6 relates our work to other QoS-aware middleware solutions. Finally, this paper is completed in Section 7 with our conclusions.

2. QOS PROVISIONING IN OPEN DISTRIBUTED SYSTEMS

We use a layered architecture to structure the problem space and position the functions that provide QoS support in an open distributed system.

In this architecture, three functional layers are distinguished, each with distinct responsibilities, offering services to adjacent layers on top and using services of adjacent layers below. Orthogonal to the layers, three planes are identified: data transfer, control and management. The architecture is depicted in Figure 1.

At the middleware layer, the responsibilities of the planes are as follows. The *data transfer plane* consists of the functions that provide the 'traditional', i.e. non-QoS related, distribution transparencies. In case of CORBA, examples of these functions are the Portable Object Adapter (POA), the GIOP protocol engine or a CDR encoder.

The *control plane* is responsible for controlling the QoS mechanisms and monitoring the achieved QoS level to ensure adequate end-to-end quality of service levels. The scope of the control plane is limited to a single association between objects.

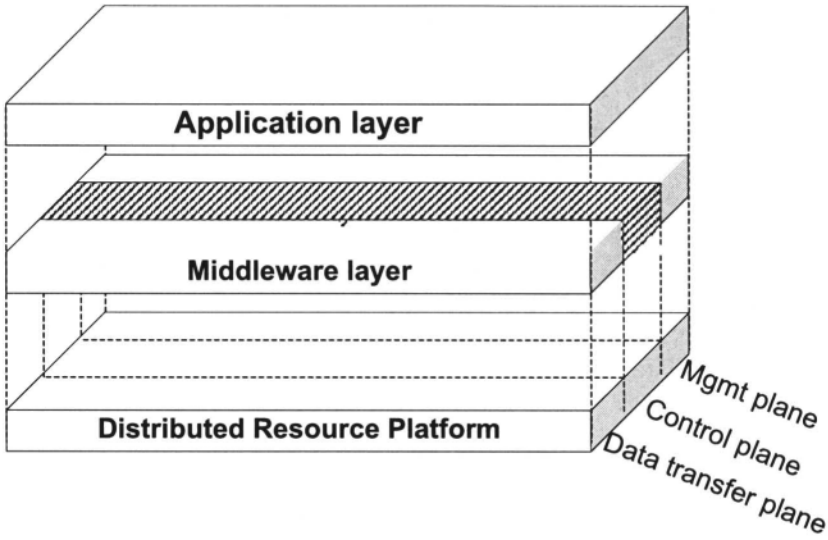

Figure 1. Layers and planes of open distributed systems

The *management plane* contains functions for long term monitoring, such as the gathering of statistics, and the instantiation and configuration of control plane functions. The scope of the management plane is beyond a single association since management actions have an effect on multiple associations.

The focus of this paper is on the control plane; therefore it is depicted as a hatched area in Figure 1.

3. REQUIREMENTS ON A QOS-AWARE MIDDLEWARE

The middleware layer is a natural place for brokering between QoS requirements of applications and the QoS capabilities of operating systems and networks. The aim of a QoS-aware middleware therefore is to provision QoS of applications in a heterogeneous distributed environment. Such a system has to deal with the diversity of low-level resource management mechanisms and the dynamic behaviour of the environment. In the literature, a number of requirements have already been identified on a QoS-aware middleware [11], [17]. The following requirements have been identified and are used as constraints on the design of our QoS provisioning service:

- Applications should be able to specify their QoS requirements using high-level QoS concepts. This frees application developers from having to know how to interact with the available system-level resource control mechanisms. Furthermore, applications become more portable since

they are independent of the lower level mechanisms. Mapping the high level QoS specifications into parameters of resource control mechanisms should be done by the middleware.

- The software infrastructure should be modular and easily extensible with new interfaces to system level QoS mechanisms. This is essential to be able to deploy the software infrastructure on top of a wide variety of hardware, OS, and network infrastructures. Specifically, it should be possible to configure into the middleware components handling the control of different resource management mechanisms dynamically. Consequently, QoS control mechanisms are expected to offer standardised interfaces to the middleware, including reflective interfaces for run-time discovery.
- The software infrastructure should allow adaptive QoS support. In distributed environments the system behaviour is dynamic and only partially predictable. This requires adaptation that can be initiated both at the application and at the system level. At the application level this means that applications can change their QoS requirements dynamically, requesting higher (or lower) QoS. In such a case, the middleware should re-allocate system resources in order to meet the new requirements. Adaptation at system level on the other hand occurs when the availability of system resources drops, due to failure, system reconfiguration, increased user load or other non-predictable factors. Again, the middleware should re-allocate resources, and if possible, this should be completely transparent for applications. If the required QoS cannot be guaranteed, applications should be notified in order to see if they can adapt themselves to lower QoS levels.
- The software architecture should support policies for a) the QoS negotiation between client and server sides, and *b*) balancing and trading functions when resources are interchangeable. Many of the mechanisms used in a QoS-ware middleware are application area dependent. Policies offer a generic way to configure these mechanisms.

4. THE QOS PROVISIONING SERVICE

The QoS Provisioning Service (QPS) is a control plane service, because its actions are limited to a single association between a client and a server object. Such an association is also referred to as a client-server binding.

QPS acts as a broker between the application level QoS requirements and the available QoS mechanisms of the Distributed Resource Platform (DRP). In addition, QPS is responsible for maintaining a QoS agreement for the lifetime of a binding. Effectively, QPS is a broker and controller for QoS agreements.

The following sections give an overview of the lifecycle of QPS controlled bindings and present how each phase of the lifecycle is supported by QPS.

4.1. QPS lifecycle model

The lifecycle of bindings controlled by QPS revolves around the QoS level offered ($Q_{offered}$) by a server object, the QoS level required ($Q_{required}$) by a client object and the agreed QoS level (Q_{agreed}). The purpose of QPS is to control the resources of the DRP in such a way that some Q_{agreed} is negotiated and maintained for the lifetime of the binding. This agreed QoS is the result of a matchmaking process between the offered QoS of the server object and the required QoS of the client.

Figure 2 shows the five lifecycle phases of QPS for a single client-server binding. The lifecycle phases are inform, negotiate, establish, operate and release.

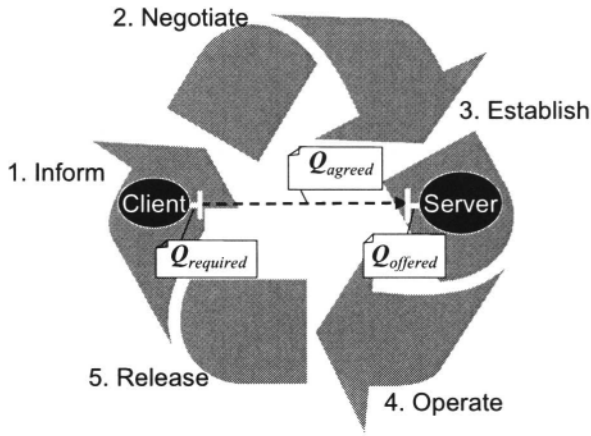

Figure 2. QoS support lifecycle in QPS

In the inform phase the client specifies its $Q_{required}$ and the server specifies its $Q_{offered}$. The offered QoS is defined as the intended QoS that a server can offer if sufficient resources are available at the time a client binds to the server. QPS will generally accept a $Q_{offered}$, unless a single server object by itself intends to consume more resources than the computing system where it is deployed can offer.

During the negotiate phase, QPS initiates a negotiation procedure between the client, the server and the DRP to see if an agreement can be reached. A successful negotiation results in a Q_{agreed} that is then associated with the binding, and resources are reserved for the binding. During the establish phase, QPS assigns the resources that have been reserved during the previous phase, so other bindings cannot claim these resources. These can be communication, storage and processing resources.

Once sufficient resources have been assigned to the binding, Q_{agreed} must be maintained. This means correcting drifting quality levels, for example, by re-allocating system resources or, in case it is not possible, by informing applications to take appropriate actions. Applications can subsequently decide to lower their $Q_{required}$ and request a re-negotiation, or end their binding. This is the operate phase. Finally, when the client does not further need the binding (this is indicated explicitly by the client) system resources are released.

In this paper we focus on the first three phases. Details of the operate phase design can be found in [1]. We don't describe the release phase, since this phase is only concerned with releasing the resources claimed during the negotiate and establish phase. The next sections describe the QPS components and how QPS supports the inform, negotiate and establish phases.

4.2. QPS implementation

The QPS framework is implemented as a CORBA service and is designed to use standard CORBA extension hooks. QPS uses the Portable Object Adapter [16], the Portable Interceptor [10] and the Open Communications Interface [6]. Since QPS uses standardized ORB extension hooks, it can work with any standard ORB implementation that implements these extension hooks.

Figure 3 shows the standard ORB components and the QPS specific extension components. On the server side, the POA is extended with a dedicated Servant-Locator and a Negotiator object for managing servants with an offered QoS. On the client side QPS provides a QoSRepository (QR) interface for managing QoS requirements of clients.

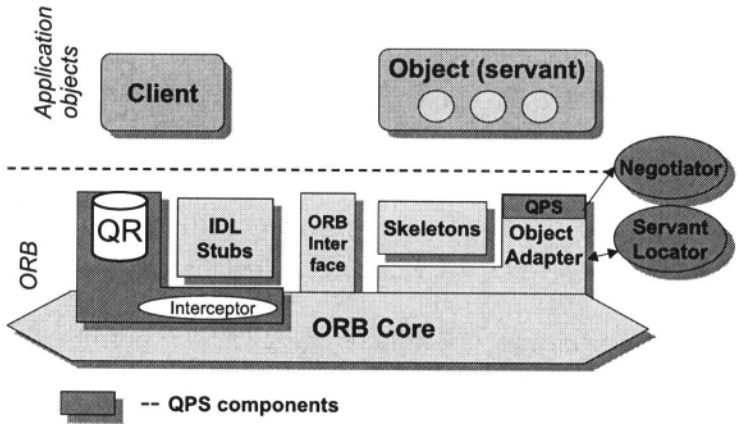

Figure 3. QPS components

4.3. Inform phase

Server application objects with offered QoS are registered at a dedicated POA, the QPS Object Adapter (QOA). These objects also register their $Q_{offered}$ with the QOA. QoS offers are expressed as XML documents. The structure of a QoS specification in XML format is inspired by the QML specification [5]. Use of XML technology has the advantage that standard parsers are available and that additional QoS dimensions can easily be incorporated into the QoS specification format. Clients can use the same XML document structure for specifying $Q_{required}$ and register their requirements with the QoSRepository.

The QR and the QOA provide the same generic interfaces to client and server objects for informing QPS about their required and offered QoS.

4.4. Negotiate phase

The purpose of the negotiation phase is to reach an agreement between the client and the server about a sustainable and acceptable QoS level Q_{agreed}. The QoS negotiation is initiated by a client application that instructs QPS to start the negotiation. As a result, QPS sends $Q_{required}$ to the server-side, using a DII request directed at the target object. $Q_{required}$ is received by the QOA and forwarded to the negotiator together with an identifier for the target object. The negotiator uses this identifier to obtain the $Q_{offered}$ and then calculates a Q_{agreed}. Calculation of Q_{agreed} can exploit different strategies. The default strategy in QPS results in a Q_{agreed} that is equal to $Q_{offered}$, whenever $Q_{required}$ is smaller than $Q_{offered}$ if sufficient resource are available such that $Q_{offered}$ can indeed be guaranteed. Negotiation fails when $Q_{required}$ is bigger than $Q_{offered}$. The default negotiation strategy can be overridden by a more sophisticated way to reach an agreement.

At any time in the lifetime of a binding, a client can register a new $Q_{required}$ and can initiate a negotiation.

4.5. Establish phase

When the negotiation is successful we enter the establish phase. At this stage the client and server have a common understanding of Q_{agreed} and resources at the middleware and DRP layers have been reserved. During the establish phase the resources are assigned to the binding and administered so that they can be released when the binding is released. In addition, a monitoring and control loop is instantiated to ensure that Q_{agreed} can be maintained for the operate phase.

5. QPS EVALUATION

The QPS components have been implemented in order to validate the feasibility of the QPS architecture[12]. In addition to the framework, a QPS plugin has been implemented that is able to reserve network resources. The design of this plugin, called QIOP, and its use for reserving network resources for CORBA method invocations are presented in the next sections.

5.1. QIOP

QIOP is an inter-ORB protocol that conveys standard inter-ORB messages via dedicated channels offering guaranteed QoS for messages sent through these channels. QIOP offers an ORB all the facilities needed to convey General Inter-ORB Protocol (GIOP) messages, in a similar way as IIOP does. The IIOP protocol specifies how GIOP messages are transported over TCP/IP connections. However, the IIOP protocol cannot provide guarantees on throughput and/or delay for message delivery. With QIOP such guarantees can be provided. In QIOP, RSVP control messages are used to investigate the availability of network resources.

QIOP builds on the acceptor/connector pattern [15]. It uses the Open Communication Interface (OCI) [3] to register and interact with the ORB. Figure 4 shows how a QIOP transport connection is established.

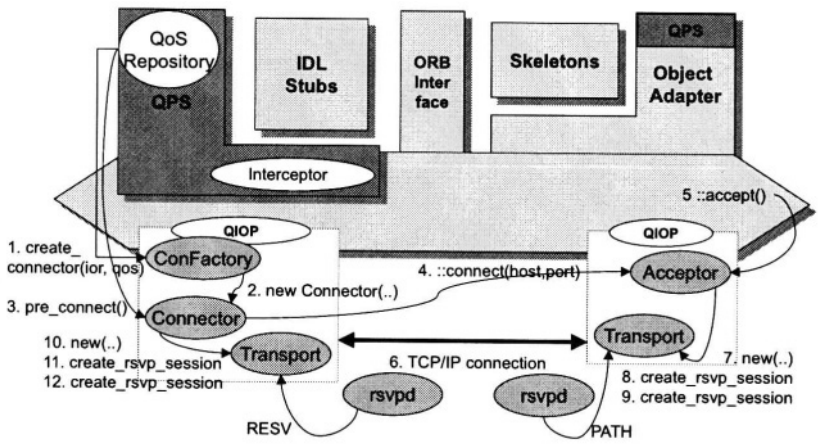

Figure 4. QIOP interactions

The QoSRepository uses the QIOP ConFactory to create a Connector. The Connector establishes a TCP/IP connection with the server side and creates QIOP transport objects. These transport objects create two RSVP sessions, one for network traffic from the client side to the server side and one for network

traffic in the opposite direction. This is necessary because RSVP can only reserve network resources for a unidirectional flow.

The Transport objects create RSVP reservations for both RSVP sessions according to Q_{agreed}.

5.2. Results of QPS & QIOP

To demonstrate the benefits of QIOP over IIOP, we've conducted some experiments. The demonstration system consists of three PCs running Linux. One PC serves as a host for client objects, another PC serves as a host for a server object. The third PC is configured as a router with two Ethernet interfaces that connect to the client and server hosts. All PCs run the KOM-RSVP implementation [7]; furthermore, the client and the server hosts run ORBs with QPS and QIOP extensions.

In the experiment, two client objects are running on the client host; one with a QoS requirement $Q_{required}$ and one without a QoS requirement. Both clients connect to a single server object (i.e. they use the same object reference). As a result, the client without QoS requirements will communicate using IIOP whereas the other client will communicate using QIOP. To show the behaviour of QIOP in a saturated network, a heavy data stream, with occasional bursts was injected into the network. Figure 5 shows the response times of the two clients.

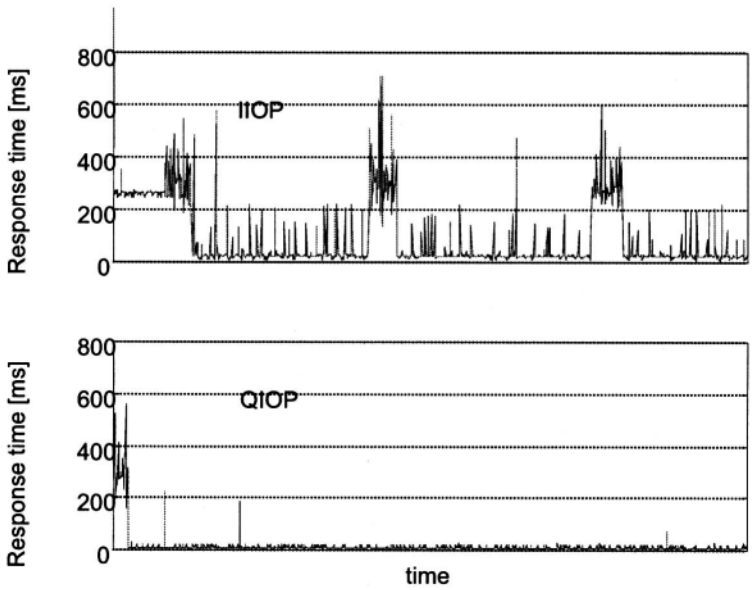

Figure 5. Response times in saturated network

The figure shows that the response times for messages carried over QIOP are not sensitive to heavy traffic bursts on the network (they all stay below 100 ms), whereas messages carried over IIOP can really suffer from large delays. This demonstrates that applications with more stringent requirements on the response time of remote object invocations can benefit from QPS with a QIOP plugin.

6. RELATED WORK

QoS-aware middleware is being developed in several projects, with different focuses. To limit the scope of this overview, we describe here only those systems that *a*) enable applications to specify their QoS requirements using high-level language concepts and *b*) realise resource adaptation. Per focus, three main QoS-aware middleware groups can be identified: general purpose middleware, real-time middleware and multimedia middleware.

The general purpose architectures differ mainly in their approach to resource reservation and adaptation mechanisms. QuO [18] is a CORBA based framework for configuring distributed applications with QoS requirements. It comes with a suite of description languages that allow applications to specify the interdependencies between QoS properties and system objects, thereby configuring the adaptive behaviour of the underlying system. QuO however uses code-weaving. This means that at compile time the code for QoS control is entangled with code for data-transfer. As a result, QuO only allows QoS mechanisms to be added at design time. Quartz [17] is another QoS-aware middleware. In Quartz, QoS concepts are introduced at system and application levels, with configurable mappings between the two. In Quartz, however, there is no reconciliation mechanism between required and realisable QoS. System agents that configure and monitor resources and balance their use carry out adaptation. MULTE-ORB [8][11] is another QoS-aware middleware that supports configurable multiple bindings. A QoS requirement is specified per binding, together with policies for negotiating QoS and for performing connection management. The QoS configuration and management system is however ChorusOS and SunOS specific. Similarly to QPS, in MULTE-ORB, QoS is controlled by feedback control loops.

OMG's Real-Time CORBA (RT-CORBA) specification [9] is targeted at real-time distributed systems. Applications specify policies that guide selection and configuration of protocols. RT-CORBA supports explicit binding in order to validate the QoS properties of bindings. After binding time, however, protocols may not be reconfigured. TAO [14] is a real-time CORBA ORB implementation targeted at hard real-time systems.

QoS aware multimedia middleware concentrates on QoS provisioning for multimedia streams. The requirements for such a platform are specified in

the reTINA project [13]. Multimedia platforms are developed in the DIMMA project [2] at APM in Cambridge, and in the Adapt [4] project that extends the COOL ORB. From the previously mentioned platforms, TAO implements the CORBA A/V streaming. Furthermore, Quartz and MULTE-ORB support streaming too.

7. CONCLUSIONS

Next generation middleware must meet the challenge of evolutionary changes and run-time changes in a heterogeneous distributed computing environment, in order to provide distributed objects with support for QoS. This paper presents an architecture for QoS-aware middleware that separates the QoS support functions from 'traditional' data transfer functions. The QoS support functions at the middleware layer are further separated into control and management plane functions.

The QoS Provisioning Service (QPS) is our framework that enables control plane functions to be added to off-the-shelf object middleware, for controlling the QoS of individual client-server associations. QPS follows a 5 phase lifecycle model to establish and control a QoS agreement between a client and a server.

The QPS framework implementation presented here uses standard CORBA extension hooks, which makes QPS a portable CORBA service. We have presented QIOP, a communication module that uses RSVP for reserving network resources and demonstrated its performance benefits compared to communication over IIOP.

Future work includes the application of the QPS lifecycle to manage the QoS of multimedia streams, and implementing a more advanced interface between QPS and system level QoS control mechanisms.

Acknowledgments

The contributions of this paper are the result of our participation in the AMIDST project (`http://amidst.ctit.utwente.nl`). We thank the project members and in particular Lodewijk Bergmans, Marten van Sinderen, Luís Ferreira Pires and Ing Widya for their valuable discussions and input.

References

[1] L. Bergmans, A. van Halteren, L. Ferreira Pires, M. van Sinderen and M. Aksit, *A QoS-Control Architecture for Object Middleware*, Proceedings of the 7th IDMS Workshop, October 17-20, 2000, Enschede, The Netherlands.

[2] DIMMA Team, *DIMMA Design and Implementation,* ANSA Phase III, Technical Report APM.2063.01, Sept. 1997.

[3] N. Fischbeck, E. Holz, O. Kath and V. Vogel, *Flexible support of ORB interoperability,* 1999.

[4] F. Fitzpatrick, G.S. Blair, G. Coulson, N. Davies and P. Robin, *Supporting Adaptive Multimedia Applications through Open Bindings,* 4th International Conference on Configurable Distributed Systems (ICCDS'98), Annapolis, Maryland, USA, May 1998.

[5] S. Frolund and J. Koistinen, *Quality of Service Specification in Distributed Object Systems Design,* Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS), Santa Fe, New Mexico, April 27-30, 1998.

[6] A.T. van Halteren, A. Noutash , L.J.M. Nieuwenhuis and M. Wegdam, *Extending CORBA with specialized protocols for QoS provisioning.* Proceedings of International Symposium on Distributed Objects and Applications (DOA'99), September 1999.

[7] M. Karsten, J. Schmitt and R. Steinmetz, *Implementation and Evaluation of the KOM RSVP Engine,* IEEE InfoCom 2001.

[8] T. Kristensen and T. Plagemann, *Enabling Flexible QoS Support in the Object* Request *Broker COOL*, 20th International Conference on Distributed Computing Systems (ICDCS'00), Taipei, Taiwan, April 2000.

[9] Object Management Group, *The Common Object Request Broker: Architecture and Specification* OMG document formal/00-10-33. October 2000.

[10] Object Management Group, *Portable Interceptors*, OMG Document orbos/99-12-02 ed., December 1999.

[11] T. Plagemann, F. Eliassen, B. Hafskjold, T. Kristensen, R.H. Macdonald and H.O. Rafaelsen *Flexible and Extensible QoS Management for Adaptive Middleware.* International Workshop on Protocols for Multimedia Systems (PROMS 2000), Cracow, Poland, October 2000.

[12] http-//quamj.sourceforge.net/

[13] Chorus Systems, *Requirements for a Real-Time ORB*, ReTINA, Tech. Report RT/TR-96-8, May 1996.

[14] D.C. Schmidt, A.S. Gokhale, T.H. Harrison and G. Parulka,. *A High-Performance End System Architecture for Real-Time CORBA.* IEEE Communications Magazine, Vol. 35, No. 2, February 1997, pp. 72-77.

[15] D.C. Schmidt, *Acceptor and Connector: Design Patterns for Initializing Communication Services",* in Pattern Languages of Program Design (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA, Addison-Wesley, 1997.

[16] D.C. Schmidt and S. Vinoski, *C++ Servant Managers for the Portable Object Adapter*, SIGS C++ Report, Vol. 10. No. 8, September 1998.

[17] F. Siqueira and V. Cahill, *Quartz: A QoS Architecture for Open Systems,* 20th International Conference on Distributed Computing Systems (ICDCS'00), Taipei, Taiwan, April 2000.

[18] J. Zinky, R. Schantz, J. Loyall, K. Anderson and J. Megquier *The Quality Objects (QuO) Middleware Framework.* Workshop on Reflective Middleware (RM 2000), New York, USA, April 2000.