Franz J. Hauck, Ulrich Becker, Martin Geier,
Erich Meier, Uwe Rastofer, Martin Steckermeier

The AspectIX Approach to Quality-of-Service Integration into CORBA

Technical Report TR-I4-99-09
1999-10-15

**Friedrich-Alexander-University
Erlangen-Nürnberg, Germany**

Operating Systems Dept. (Informatik 4), Prof. F. Hofmann

The *AspectIX* Approach to Quality-of-Service Integration into CORBA

Franz J. Hauck, Ulrich Becker, Martin Geier,
Erich Meier, Uwe Rastofer, Martin Steckermeier
IMMD 4, Friedrich-Alexander University Erlangen-Nürnberg, Germany
AspectIX@cs.fau.de

Abstract

Applications running on a large distributed system face a dynamically changing environment. Thus, an application needs to adapt to new conditions or it will fail to continuously meet its specification. This is especially crucial if the application needs to implement a certain quality of service (QoS). Object-based middleware systems like CORBA implementations are currently not able to directly support QoS-aware applications. AspectIX is a novel and CORBA-compliant middleware architecture, in which so-called aspects define and describe QoS requirements on a per object basis independently from the functional interface. The aspect configuration can dynamically be changed by the client. Additionally, AspectIX allows object developers to implement QoS requirements in arbitrary ways. A partitioned object model, which is the basis of AspectIX, provides full control over the client side of an object invocation and thus allows end-to-end QoS.

1. Introduction

When applications run on a large distributed system, e.g., on the whole Internet, they face a dynamically changing environment. With leaving the local area network and building large scale applications, failures and delays in the network protocol layer affect applications far more severe than in a LAN [17]. Additionally, parts of the application may fail independently for an unacceptably long time. An application may not meet its specification if it is not able to adapt to those conditions. Especially applications that need to implement a certain quality of service (QoS) are affected. If the system conditions change, an application has to adapt alternative mechanisms to implement the required QoS, or it has to signal that the quality of service cannot currently be met. The latter may allow the client to negotiate a less strict quality requirement with the object. Let us give two examples for dynamically changing QoS in those systems:

First, for increased availability and better fault tolerance, replication mechanisms have to be deployed in large applications. Replication needs some strategies to keep replicas consistent. These strategies are the more expensive the more transparent they are for clients and object developers. Application-aware consistency algorithms can improve efficiency, but often also the client has to be taken into account. The client's requirements (QoS requirements in this respect) may dynamically change at run-time and influence consistency strategies.

Second, bandwidth-consuming applications (e.g., video or audio on demand) may temporarily exceed even the resources in a LAN, and thus need to adapt to the currently available resources.

We can conclude that applications need to be self-adaptive and aware of the current quality of service they require and implement. Pure CORBA implementations are not able to directly support quality-of-service requirements. Thus CORBA extensions are necessary to run QoS-aware applications.

CORBA applications are built on the basis of distributed objects. A QoS-aware CORBA application may need objects for which certain QoS requirements can be given, which in turn have to be implemented by those objects. If the requirements cannot be fulfilled the objects have to signal this situation, which allows the objects' client to adapt in certain ways. We can distinguish two different problem domains of QoS support: description and definition of QoS requirements by the client of an object, and implementing the QoS requirements by the object developer.

In the worst case, the description of QoS requirements could be implanted into the functional interface of a distributed object and thus be transparent to middleware systems. It is more appropriate to separate different QoS requirements from each other and from the functional interface (separation of concerns), which demands some integration into the middleware. Some related systems extend CORBA IDL for integrating QoS description [1, 14]. In the *AspectIX* middleware architecture presented in this paper, we follow a different approach: we automatically extend the interface of all CORBA objects by only one method that allows the configuration of QoS requirements. These requirements are named *AspectIX* aspects and are represented by ordinary objects described in CORBA IDL [4].

The implementation of QoS requirements usually needs middleware support or must circumvent the middleware and directly use services of the underlying operating system. CORBA middleware creates generic stubs for client-to-object communication. As this communication is almost always affected by QoS requirements QoS-aware stubs are needed. Most systems can plug-in extra code into the generic stub for implementing the client-side part of the QoS requirements. In contrast, the *AspectIX* architecture uses a more general model. At the client side there is an own local object that implements parts of the QoS requirements and which can even implement parts of the object's functionality. For the client this local object, which we call a *fragment*, is not distinguishable from an ordinary CORBA stub.

The rest of this paper is organized as follows: Section 2 will give a more detailed motivation for our work. In Section 3, we introduce the *AspectIX* architecture and its approach to QoS integration. Section 4 compares our concepts to related work. In Section 5, we will give our conclusions.

2. A Motivating Example

Let us consider a large distributed application, namely a live Internet auction. Imagine that this application allows users all over the world to enter a saleroom in which at a certain time an auction of exactly one item is run by an auctioneer. The users can bid, and according to some rules the user with the highest bid is obliged to buy the item. All participating users can watch the bids and their history on their local computer screens and if they want they can outbid each other. Users may also give items to the auctioneer for selling them at a later time.

Today, such an application is typically implemented using World Wide Web browsers as graphical user interfaces. A special HTML Frame displays the bids and the comments of the moderating auctioneer. The frame is automatically refreshed by the browser, e.g., every 15 seconds, to distribute the current state of the auction. Buttons on the Web page allow the user to outbid the current bid. Clicking on such a button posts the new bid to a central Web server, which updates its internal auction database and redistributes this information as soon as the Web browsers want to refresh their auction frames.[1]

The current implementation has some problems that immediately have to do with different and changing system conditions on the Internet, e.g., the network connection: When a user has a very poor Internet connection she will not get her auction frame updated within 15 seconds. This is even worse as the auction frame is reloaded regardless of any changes made to it. A user with a good Internet connection will only get new updates of current bids every 15 seconds and not more. As the network conditions can vary a lot, users can experience both a poor and a good connection within the same minute depending on the network load created by other Internet users. In fact, the user does not see what happens: is there no new information or is the refreshing of the frame delayed.

Another problem is that the auction relies on a central Web server and its database. On one hand, this easily preserves complete consistency of the bids and other data. On the other hand, if the server fails the auction has to be cancelled and usually be set up as a new one at a later time. It is almost impossible to use replicated Web servers as current Web browsers and the HTTP protocol do not support them.

Implementing this auction application using CORBA technology [7] would overcome the direct problems stemming from the World Wide Web. So, the browser object (user agent) at the user's side does not need to poll the bids at a central server but could be notified of new bids or comments (see Fig. 2.1). However, the basic problem remains the same: the application is not automatically QoS-aware and not able to adapt to changing conditions. In fact, users with poor or varying Internet connection will still experience problems and will not be informed if the delivery of new bids is delayed.

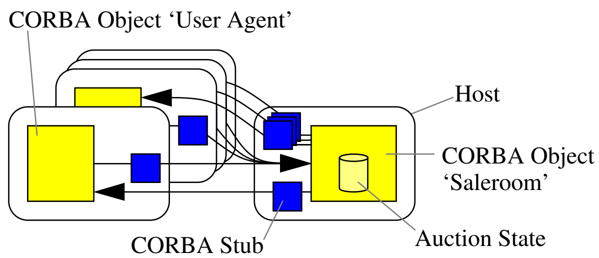

Fig. 2.1 CORBA implementation of a live Internet auction.

The replication of the virtual saleroom would be possible in CORBA, although CORBA does not directly support replication. We would need special objects at the client side that switch server connections if the old one breaks. For some items, a costly replication simply is not necessary. Nevertheless the server switching needs to be implemented at the client side to be prepared for replicated salerooms.

Ideally, as a user we would wish to describe the quality-of-service that we expect from such an auction application. As a customer we would like to set up the rate in which we want to be informed about new bids. At the same time we would like to observe whether the system is able to implement it and watch the current update rate. As a vendor we would like to state our expectations on the probability of total failures, or the degree of replication respectively.

1. This scenario is derived from a real live auction at www.ricardo.de.

CORBA does not provide any support for such QoS-aware applications.

3. The *AspectIX* Architecture

This section introduces the *AspectIX* middleware architecture that we developed. Its primary goal is to offer a platform that allows users to describe all kinds of QoS requirements and to adapt to changing conditions. The architecture enables object developers to implement distributed objects that provide QoS support.

AspectIX is designed to be CORBA-compliant [7], which means that CORBA objects can be hosted on and accessed from an *AspectIX* implementation, and that *AspectIX* objects can be accessed from every other CORBA platform, but often with loosing the QoS support.

3.1 Object Model

Shapiro introduced the so-called proxy principle [11], which conceptually considers a local access point to a remote service (a proxy) to be part of that service. In *SOS* [12] and *Globe* [16] this concept has been implemented using objects as basic building blocks of applications. A distributed object is partitioned into multiple distributed parts. A client of such an object has to bind to a local part (proxy) of the object in order to invoke methods on this object (see Fig. 3.1). All communication between the object's parts is transparent to the client.

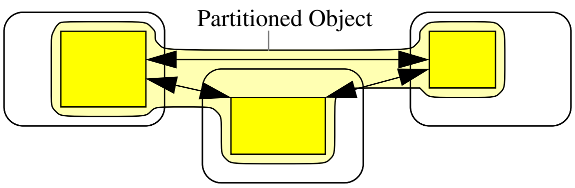

Fig. 3.1 A partitioned object with three parts each placed on a different host.

Such an object model is especially appealing when it comes to quality-of-service requirements by the client of an object. Often, quality-of-service implementations need some support at the client side. Consider a client who needs bandwidth guarantees for data delivered to or from an object. These guarantees can only be implemented by entities controlling both ends of the data transport link (end-to-end QoS) including the client side. Pure stub-based distributed systems, which conceptually consider a stub or remote object reference to be something outside of a distributed object, cannot handle this, because the stub can only forward invocations with predefined semantics, and thus cannot implement QoS requirements. CORBA implementations are stub-based.

AspectIX adopts the partitioned object model. A distributed object is partitioned into *fragments*, and again clients need a local fragment to invoke methods on a distributed object. Each fragment forms a *view* on the distributed object. Access to a fragment, and to the distributed object respectively, is provided by interfaces connected to that view. Those interfaces represent the CORBA object references for CORBA compliance. As long as the client just binds to an object and invokes methods the client will not see any differences to CORBA.

In case of the live Internet auction, the saleroom is modelled as one distributed and partitioned *AspectIX* object. Each user has to bind to that object in order to bid or watch the bids of other users. In fact, the saleroom object is distributed over all computing systems of users who want to participate in the auction (see Fig. 3.2). The interface of the saleroom object is the same for every user, but the object-internal communication is entirely hidden. Also it is hidden in which fragments the actual data of the object are stored.

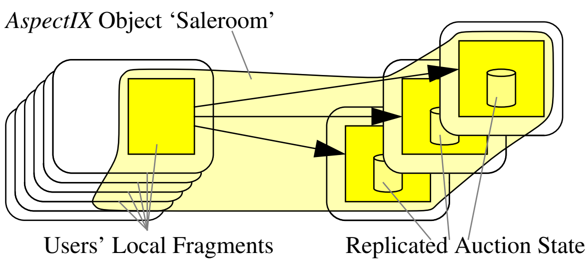

Fig. 3.2 CORBA implementation of a live Internet auction.

3.2 Describing QoS Requirements

Adopting the terminology from [13], a *QoS characteristic* is some quantifiable and identifiable aspect of the QoS system (e.g., the delay of a message in transit or the bandwidth of a connection). A *QoS category* is a set of QoS characteristics that are to be managed for a certain application requirement (e.g., fault tolerance or real-time). *QoS parameters* are values related to QoS characteristics. They represent a current state of a characteristic, a desired maximum or minimum level of a characteristic, a threshold, etc.

In *AspectIX*, we adopt the term *aspect* from aspect-oriented programming [5, 3] to describe nonfunctional properties of a distributed object. In the context of QoS, an aspect specifies a set of QoS parameters in order to manage certain QoS characteristics. Thus, a QoS aspect may define the requirements of one QoS category. However, in some cases it may be more appropriate to split a QoS category in several QoS aspects because this allows better reuse of aspect definitions.

An example for an *AspectIX* QoS aspect is the actuality of delivered data. This aspect may have only one QoS parameter determining the maximum time since the delivered data have been considered valid. The data actuality aspect may be useful for the live Internet auction. Setting the maximum time to, let's say 4 seconds, would mean that the highest bid handed out by the local fragment to the user is not more than 4 seconds out-dated (if this QoS requirement can be fulfilled).

Another example is an aspect describing availability. The corresponding QoS parameter specifies the degree of redundancy or, more abstract, the maximal probability of a service failure. This aspect could be used to determine the availability of the saleroom's auction database (in fact, the object's state).

The configuration parameters of an *AspectIX* aspect are described in CORBA IDL as attributes of an aspect configuration object.[2] Every aspect has a unique name, which can be used to identify the aspect and its configuration object. The aspect configuration object for describing data actuality could be written in CORBA IDL like this:

```
module AspectIX {
   module Aspects {

      interface DataActuality {
         attribute unsigned long maximalAge;
                                // in milliseconds
      };
   };
};
```

The aspect for specifying data redundancy or degree of replication could be written as:

```
module AspectIX {
   module Aspects {

      interface DataRedundancy {
         attribute unsigned short degree;
                               // of redundancy
      };
   };
};
```

3.3 Configuring QoS Requirements

When the client has bound to a distributed object it has a CORBA object reference at hand. In *AspectIX* this is called an interface to the object, or to the local view respectively (see Fig. 3.3). All *AspectIX* interfaces have an additional method to retrieve a reference to the so-called view object representing the view to the distributed object. This view object hosts an aspect configuration as a set of configuration objects. The corresponding fragment has to implement all the requirements specified by the aspect configuration, which makes the configuration valid. If the fragment cannot fulfill the requirements the configuration will become invalid.

The client may change the aspect configuration dynamically during run-time to adapt to certain application conditions. The fragment in turn may adapt to a changed configuration by replacing itself by another fragment implementation which is more appropriate to implement the new configuration. This exchange is transparent to the client, who still deals with the same view object and its interfaces.

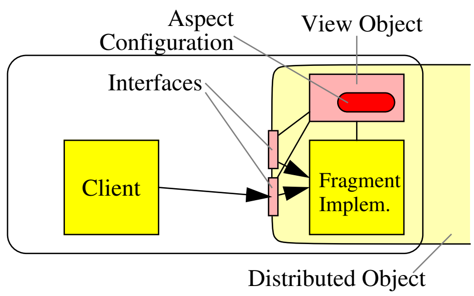

Fig. 3.3 A local fragment with two interfaces and a view object.

Taking Java as the implementation language, a client could change the required actuality of data delivered by the local saleroom fragment with the following code:

```
import de.fau.AspectIX.*;
...
SaleRoom sroom= ...;       // Get saleroom ref.

// Get current aspect configuration
View v= sroom.get_view();
AspectSet aspectConf= v.get_aspects();
DataActuality da=
       aspectConf.get_aspect( "DataActuality" );

// Set new value
da.set_maximumAge( 4000L );

// Set new configuration
aspectConf.set_aspect( "DataActuality", da );
try { v.set_aspects( aspectConf ); }
catch( ASPECTS_INVALID e ) {}
```

The configuration can be retrieved from the view as a set of aspect configuration objects. Each configuration object can be taken out of the set by its name. A changed configuration object can be put back to a set and given to a view, which takes it as a new aspect configuration. The user may hold different configuration sets at hand to switch immediately to a new configuration if necessary.

In our case the local fragment of the saleroom will immediately try to fulfill the new requirements. In some situations, it may find out that it is not the right fragment implementation to do that, and will load a new implementation, hand over to the new one, and delete itself.

In other situations, none of the available fragment implementations can fulfill the requirements and the configuration becomes invalid. In this case, `set_aspects()` will throw a run-time exception `ASPECTS_INVALID` so that the client is immediately notified. The exception carries additional information and allows the client to find a new and valid aspect configuration.

2. As these objects are not supposed to be distributed we consider to convert them into CORBA values in future versions of our architecture.

3.4 Handling Invalid Aspect Configurations

An aspect configuration can also become invalid if, for example, the network load has changed. If the fragment implementation notices that it can no longer fulfill the configured requirements, it invalidates the configuration. It can also validate the configuration if it detects that the requirements can be fulfilled again.

The client can configure how invocations on the same view are handled while the aspect configuration is invalid. They can be blocked as long as the configuration remains invalid. They can raise a run-time exception, signalling the invalid configuration to the user, or they can ignore the invalidity and proceed as usual.

Blocking is useful for objects which require the configured QoS in any case (e.g., all data have to be encrypted). Throwing an exception makes sense when the client needs to know whether the configuration is invalid or not. Ignoring an invalid configuration is useful for applications that want to proceed anyway regardless of the current QoS conditions (e.g., video on demand).

As an aspect configuration can become invalid at any time, the user might want to know when the configuration becomes invalid or valid. Therefore, the client can set up a callback handler which gets notified if the aspect configuration becomes invalid. Thus the client can adapt to the new condition (e.g., by providing a new aspect configuration) or simply wait until the configuration becomes valid again.

The callback handler and the exception thrown by `set_aspects()` and by method invocations, in case of an invalid configuration, pass additional information to the client so that the client is able to find alternative configurations. The client can specify a list of aspects ordered by their importance. In case of a configuration becoming invalid, the fragment can compute alternative aspect configurations on demand that would be valid if set. These configurations try to be as similar to the current but invalid configuration as possible, especially concerning the aspects most important to the client. In best case, important aspects remain unchanged whereas all other aspects may be changed in such a way that the complete configuration would be valid. The client can now retrieve the alternative configuration using an iterator. If the first alternative does not fit the client's needs it may retrieve another alternative. The more alternatives are computed the less does the importance of aspects matter for the computation. Finally, an alternative configuration may be produced which does also change important aspects.

The concept of alternative configurations makes the handling of invalid aspect configurations much easier, but in some cases alternatives do not make sense, because the client necessarily needs the current configuration to be valid. To avoid unnecessary computation in the fragment, the client can specify how many alternative configurations will be retrieved at most. If set to zero, no alternatives need to be computed.

3.5 Live Internet Auction

In the auction application, the client will bind to a saleroom object and specify the actuality of the bidding information. The local fragment will, in conjunction with the other fragments of the saleroom, check the actuality (e.g., by probe messages and synchronized clocks). If it detects that the actuality cannot be guaranteed it sets the aspect configuration invalid. An alternative aspect configuration computed by the fragment could include a data actuality aspect with less strict `maximalAge` value.

As an alternative to a static setting in the client application, the user interface may allow to configure the actuality aspect (e.g., with a slider). Thus, the user can adapt the application to her own needs. If a user wants to watch the ongoing auction, a less strict actuality will do, whereas if the user wants to buy something she relies on fast information delivery. If the actuality cannot be fulfilled the user interface may show a red warning on the screen.

If a fragment finds out that it cannot fulfill the current configuration it may consider alternative mechanisms (e.g., data could be compressed to consume less bandwidth, trading off computation against bandwidth). These alternative mechanisms could be implemented in a different fragment implementation which is transparently loaded.

As the saleroom is partitioned into many fragments this structure can be used for arbitrary intra-object communication schemes. If there are many users who enter the saleroom from one place, let's say from one ISP[3] or one company, one of the users' fragments may act as a master receiving update information from the Internet and re-distributing it to the other fragments at the same place. Thus communication costs could be limited and the QoS requirements of more users can be fulfilled. The fragments may build up a spanning tree as used in multicast implementations (if they do not already rely on an appropriate multicast implementation).

The fragment of a vendor may also become a host for replicating the data if the user specified data redundancy. We could imagine an implementation which is much less central than today's CORBA implementations, which would make applications more reliable.

3.6 Intra-Object Communication

In CORBA, object communication is fixed to pre-defined semantics and stub-server–based remote method invocation. Unlike CORBA, *AspectIX* defines only local invocations, namely on local fragments using object references.

3. Internet Service Provider.

For intra-object communication, i.e., communication between fragments of a single distributed object, the architecture has to provide some mechanisms.

As in a QoS-aware application the standard remote method invocation is not enough, an *AspectIX* ORB provides the concept of communication end points (CEPs). A fragment can open such a CEP and attach a stack of protocols to it. These protocols may honor QoS parameters so that communication-related QoS requirements can be directly mapped to protocols (e.g., bandwidth reservation on ATM links for video transport). The fragments may use arbitrary protocols for communication, and they may even switch protocols during their life time. There are three different kinds of CEPs: connectionless, connection-oriented and RPC-based CEPs.

For the live Internet auction, a local fragment may use a reliable multicast protocol to notify the replicated server fragments about a new bid, and a TCP stream for getting the newest bids of other users.

One protocol stack, offered by *AspectIX*, is GIOP over TCP/IP (which makes up IIOP). Thus, an *AspectIX* object is also accessible from standard CORBA ORBs. Also, a fragment can access other CORBA objects using IIOP.

The CEPs and a set of predefined protocols are part of the *AspectIX* architecture. So, there is no need to use communication mechanisms outside of the middleware as it would be the case in most CORBA implementations if the application needs some other communication mechanism than RPC-based method invocation. We imagine that an *AspectIX* ORB may download necessary protocol modules on demand from an external repository. Thus the ORB itself becomes modular and flexible. This is ongoing research and beyond the scope of this paper.

3.7 *AspectIX* Services

As there may exist many different fragment implementations for a single distributed object, the right implementation must be found when the client binds to an object the first time or when a fragment wants to be replaced by another implementation. On one hand, this can be hard-wired into the Interoperable Object Reference (IOR) or into the fragment implementations. On the other hand, for dynamically reacting objects this would be too static. So, we imagine a service which maps an aspect configuration to the appropriate fragment implementation. The implementation can be pointed to by a URL referring to a Java class file. For other language bindings than Java some shared library modules or DLLs[4] have to be identified.

Another service is necessary for locating the fragments of one distributed object. As a new fragment instance has to connect to the other fragments, it has to know the protocols and the contact addresses. As in the Globe project a very sophisticated location service has been already developed [15], we consider to adopt it for *AspectIX*.

4. Dynamic Link Library.

3.8 Implementation of the Architecture

As the *AspectIX* architecture is almost completely designed we are working on an implementation. The first language supported by *AspectIX* is Java as it makes dynamic loading of code very easy. A first prototype which was not yet distributed and did not have any CEPs was finished in summer 1999. It allowed the validation of various internal interfaces (e.g., of the view and interface objects) and disclosed very subtle problems with concurrency when callback handlers and fragment replacements are involved. These problems have been solved and their solution is going to be integrated in our implementation.

4. Related Work

The *SOS* operating system was built for the context of multimedia applications in a local network [12]. It implements Shapiro's proxy principle in form of fragmented objects (FOs) [11]. SOS adopts its own protocols and has a different concept for realizing intra-object communication. All communication is either modelled by so-called channels or by other predefined fragmented objects. The *AspectIX* approach of communication end points allows more flexibility (e.g., also legacy code can be accessed by standard protocols) and is more open for new protocols and QoS requirements.

Globe is another system that is built on partitioned objects [16]. Its goal is to overcome scalability problems in wide-area applications by deploying transparent replication techniques. Considering QoS requirements, Globe concentrates on replication and has a sophisticated framework for building fragments with diverse replication mechanisms. In *AspectIX*, such frameworks are still missing for other QoS requirements. Unlike *AspectIX*, Globe was never designed for being CORBA compliant.

With the *CORBA Messaging* document [8], the OMG adds some QoS support to CORBA. As CORBA only offers remote method invocation the QoS requirements are restricted to this communication scheme (e.g., priorities on requests).

TAO [10] and *Electra* [6] are two ORBs that support dedicated QoS requirements on a CORBA-compliant middleware platform. TAO supports real-time requirements, Electra realizes fault tolerance. Both systems can hardly be used for applications demanding general or mixed requirements.

dynamicTAO [9] is based on TAO and allows the dynamic exchange of ORB components (e.g., of the marshalling or threading component). Similar to TAO and Electra, dynamicTAO is only suited for one dedicated application scenario, because the exchange of components affects all running applications. In *AspectIX*, there can be QoS requirements on a per object basis.

A more detailed comparison is necessary to systems extending CORBA by general QoS support like *QuO* [14, 17] and *MAQS* [1, 2].

MAQS extends CORBA IDL to describe QoS requirements. An object may have an IDL interface which may refer to several of those QoS requirements, of which one can dynamically be negotiated at run time. In *AspectIX*, multiple aspects may be configured in parallel (e.g., data actuality <u>and</u> redundancy). MAQS' QoS definitions also include methods for the client and the server object (e.g., methods to support group membership or state update in case of a redundancy QoS). We think that those methods should be entirely hidden to the client. Aspects should describe what the requirements are and not how they are achieved. Unlike MAQS, we consider trading as a separate issue which can be solved without any effect to our middleware architecture.

QuO also extends CORBA IDL by several other languages. It has a sophisticated concept for adapting the object implementation in case of changing system conditions. At this time, the developers of *AspectIX* fragments have to do that on their own, but we work on a supporting framework. Like MAQS, QuO adopts the standard stub-based object model of CORBA. A local object called *delegate* is in charge of the client-side implementation of QoS (in MAQS this object is called *mediator*). Thus, it is difficult to integrate functional properties with QoS. In *AspectIX*, we could have a local fragment implementing a data cache for the distributed object. This cache could relate to a data actuality and a data redundancy QoS requirement. However, the cache is quite application-specific. MAQS and QuO are not designed to include the cache code into their client-side QoS object.

5. Conclusion

We introduced a novel approach to QoS integration into CORBA. While there is almost no change to the existing CORBA interfaces, the object model is transparently changed to a partitioned model. This model offers the possibility to control the client side of a distributed object. Combined with the description of QoS requirements in form of *AspectIX* aspects this makes distributed objects QoS-aware. Clients can configure their desired service quality and get notified if it is no longer achievable.

For the developer of a distributed object there is much freedom of choice how to implement the different fragments of such an object. Communication end points attached to a certain protocol stack allow a variety of communication patterns within a distributed object.

A prototype implementation has to show in the near future how the architecture behaves in practice. Ongoing research is about how to support the object developer in providing different QoS aspects. We work on building a framework in which the developer can integrate the functional code of the object and which adds most of the QoS code automatically.

Acknowledgements

This work has been funded as part of project OVEST by the Bavarian Research Foundation, Sun Microsystems Munich, Siemens ATD Erlangen, 3Soft Erlangen, and by an IBM Research Award granted by the IBM Zürich Research Lab.

References

[1] C. R. Becker, K. Geihs: "QoS as a competitive advantage for distributed object systems: from enterprise objects to global electronic market." In *Proc. of the 3rd Int. Enterprise Distr. Obj. Comp. Conf.*, EDOC '98 (La Jolla, Cal.), 1998.

[2] C. R. Becker, K. Geihs: "Generic QoS specifications for CORBA." In *Proc. of the KiVS Conf., Kommunikation in Verteilten Systemen* (Darmstadt, March 1999), Springer, Informatik aktuell, 1999. pp. 184–195.

[3] F. J. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, M. Steckermeier: "AspectIX: A middleware for aspect-oriented programming." In *Object-Oriented Technology,* ECOOP'98 Workshop Reader, LNCS 1543, Springer, 1998.

[4] F. J. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, M. Steckermeier: *AspectIX: An aspect-oriented and CORBA-compliant ORB architecture.* Techn. Report TR-I4-98-08, IMMD IV, Univ. Erlangen-Nürnberg, Sep. 1998.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda , C. Lopes, J.-M. Loingtier, J. Irwin: *Aspect-oriented programming.* Techn. Report SPL97-008 P9710042, Xerox Palo Alto Res. Center, 1997.

[6] S. Maffeis: "Adding group communication and fault-tolerance to CORBA." In *Proc. of the USENIX Conf. on Object/Oriented Techn. and Sys.* – COOTS (Monterey, Cal., June 1995).

[7] Object Management Group, OMG: *The Common Object Request Broker: Architecture and Specification.* Rev. 2.3, OMG Doc. formal/98-12-01, June 1999.

[8] Object Management Group, OMG: *CORBA Messaging.* OMG Doc. orbos/98-05-05, May 1998.

[9] M. Román, F. Kon, R. H. Campbell: *Supporting dynamic reconfiguration in the dynamicTAO reflective ORB.* Techn. Report UIUCDCS-R-99-2085. Dept. of Comp. Sci., Univ. of Illinois at Urbana-Champaign, Feb. 1999.

[10] D. C. Schmidt, D. L. Levine, S. Mungee: "The design of the TAO real-time object request broker." In *Comp. Comm.* **21**(4); Elsevier Science. April 1998.

[11] M. Shapiro: “Structure and encapsulation in distributed systems: the proxy principle.” In *Proc. of the 6th Int. Conf. on Distr. Comp. Sys.* – ICDCS (Cambridge, MA, May 19-23, 1986), IEEE Comp. Soc.,Wash., DC, 1986. pp 198–205.

[12] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, C. Valot: “SOS: an object-oriented operating system – Assessment and perspectives.” In *USENIX Comp. Sys.* **2**(4), 1989. pp. 287–337.

[13] C. Sluman, J. Tucker, J. P. LeBlanc, B. Wood: *Quality of Service (QoS)*. OMG Green Paper, Ver. 0.4a, OMG Doc. ormsc/97-06-04, June 1997.

[14] R. Vanegas, J. A. Zinky, J. P. Loyall, D. Karr, R. E. Schantz: “QuO’s runtime support for quality of service in distributed objects.” In *Proc. of the Int. Conf. on Distr. Sys. Platforms and ODP, Middleware ’98* (The Lake District, UK), Springer, 1998.

[15] M. van Steen, F. J. Hauck, G. Ballintijn, A.S. Tanenbaum. "Algorithmic design of the Globe wide-area location service." In *The Comp. J.* **41**(5), 1998, pp. 297–310.

[16] M. van Steen, P. Homburg, A.S. Tanenbaum: “Globe: a wide-area distributed system.” In *IEEE Concurrency*, Jan.–March 1999, pp. 70–78.

[17] J. A. Zinky, D. E. Bakken, R. E. Schantz: “Architectural support for quality of service for CORBA objects.” In *Theory and Practice of Object Sys.* **3**(1), 1997.