Extending CORBA with Specialised Protocols for QoS Provisioning

A.T. van Halteren[α], A. Noutash[β], L.J.M. Nieuwenhuis[α], M. Wegdam[α]
[α]KPN Research, P.O. Box 15000, NL-9700 CD Groningen, The Netherlands,
{A.T.vanHalteren, L.J.M.Nieuwenhuis, M.Wegdam}@research.kpn.com
[β]CTIT, P.O. Box 217, NL-7500 AE Enschede, University of Twente, The Netherlands, anoutash@cs.utwente.nl

Abstract

The CORBA layer in a distributed system hides the heterogeneity of the underlying computer network. The interactions of objects located at different computing systems are described in terms of IDL specifications and the ORB takes care of the actual transfer of messages along the wire. In fact, an object interaction is translated into the transfer of GIOP messages over TCP/IP networks (IIOP). The advantages in terms of interoperability and portability are obvious. Currently, OMG is in the process of standardising the Open Communication Interface (OCI). Through OCI, a protocol module can be plugged into any ORB and hence, the distributed application including the ORB can be put on top of any network without changing the application's code, thus implementing network transparency. Obviously, the QoS of distributed applications depends on the QoS of the underlying network protocols, e.g., best effort versus guaranteed bandwidth. Through OCI we are able to use the network protocol that is needed to satisfy the QoS requirements of a specific distributed application. In this paper, we propose to extend CORBA with specialised protocols for QoS provisioning using OCI. We have prototyped protocol plug-in's, including a plug-in that exploits IP Multicast. The IP Multicast plug-in can be used in situations where one client communicates with a group of replicated servers. In fact, we have used this mechanism to implement replication transparency in CORBA. We have shown that the OCI interface can be used for QoS provisioning in CORBA. Based on our hands-on experience, we also have identified some shortcomings in the proposed OCI specification.

1 Introduction

We make two observations about CORBA. The first is that the CORBA specification does not provide any means to influence the Quality of Service (QoS) that is provided. There is no interface defined to do this, or even inquire about the offered QoS. What is offered is simply a best effort QoS. The second observation is that CORBA does not exploit all features of the underlying network. The OMG has only specified one protocol, called IIOP, which uses TCP/IP as a transport protocol. ORB vendors can support other protocols and network technologies, but they are then delivered as proprietary extensions. These extensions are not available in an interoperable manner, although they could be very helpful to fulfil the QoS requirements of an application.

To use different networks than TCP/IP the OMG is in the process of standardising the Open Communication Interface (OCI), as part of the CORBA/IN interworking RFP [1]. The OCI is used in the CORBA/IN interworking specification to use signalling protocols from Intelligent Networks (IN) to convey CORBA method invocations. However, the OCI has a much broader use than IN only. The OCI specifies interfaces within an ORB that will enable a developer to create his own protocol that can be used with any ORB that complies to these interfaces. OCI is primarily intended to enable the usage of CORBA with all kinds of different (non-IP based) networks, like in the CORBA/IN Interworking RFP where it is used to transport GIOP requests over SS7.

This paper describes how OCI can be used to extend an ORB with specialised protocols to support the provisioning of QoS, while remaining interoperable. We believe that distributed systems, such as CORBA based systems, should take more advantage of the available resources in a distributed system. We focus on the network resources and demonstrate how an ORB implementation can be extended with a specialised transport protocol.

1.1 Structure

Section 2 introduces QoS, and describes how QoS provisioning can be included in a CORBA environment. Section 3 explains how the OCI specification works, and how the OCI ensures interoperability between different ORBs. Section 4 describes and evaluates our prototype implementation. Section 5 evaluates OCI and describes related work. Section 6 ends the paper with conclusions and future work.

2 QoS provisioning in CORBA

For a CORBA application object with QoS requirements the CORBA specification does not provide an interface to indicate these requirements to the ORB and the ORB cannot inform application objects whether the QoS requirements are met. This section describes an architecture for incorporating QoS provisioning in CORBA systems.

2.1 QoS provisioning

Provisioning of QoS usually involves a common understanding between two or more parties about the quality characteristics of the service. These parties can be end-users, but they can just as well be software components. This section describes the generic concepts that are used throughout this paper to describe QoS provisioning. These concepts are based on the ISO/IEC QoS Framework [2,3].

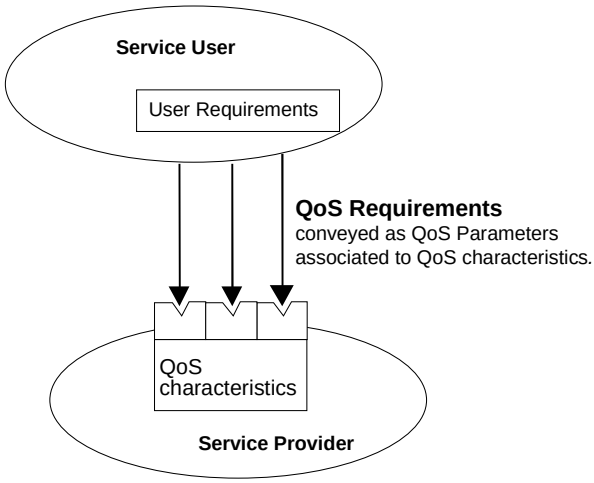

Figure 1: Generic QoS provisioning model

The ISO/IEC QoS provisioning model defines two roles for entities in a distributed system. These two basic roles are Service Provider and Service User. The Service Provider has a number of QoS characteristics, such as availability and response time of the service. The Service User has a number of User Requirements, some of which may be related to the QoS expected from the Service Provider. However, User Requirements do not have to be expressed in terms of the QoS characteristics of the Service Provider. For example, a user requirement could be "the service should always be accessible". Often the User Requirements are expressed as subjective requirements, whereas the Service Provider needs objective requirements in order to handle them [4]. The user requirements must therefore be translated into one or more QoS Parameters that are expressed in terms of the QoS characteristics of the Service Provider. The QoS provisioning model, as depicted in Figure 1, should enable entities to express their quality requirements.

The relevant QoS concepts used in this paper are defined as follows:

QoS characteristic:	A quantifiable aspect of QoS, which is defined independently of the means by which it is represented or controlled. Examples are: delay, throughput, availability and integrity.
QoS requirement:	QoS information that expresses part or all of a requirement to manage one or more QoS characteristics, e.g. a maximum value, a target, or a threshold; when conveyed between service user and service provider, a QoS requirement is expressed in terms of QoS parameters.

The QoS characteristics of the Service Provider are determined by the QoS management functions that the Service Provider has. On the other hand, the QoS management functions effect the QoS characteristics. The relations between these QoS concepts inside the Service Provider are depicted in Figure 2.

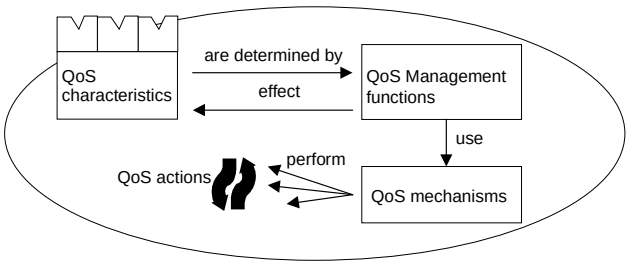

Figure 2: QoS provisioning inside the Service Provider

QoS management function	A function specifically designed with the objective to meet QoS requirements
QoS mechanism	A specific mechanism that may use QoS parameters or QoS context, possibly in conjunction with other QoS mechanisms, in order to support establishment, monitoring, maintenance, control, or enquiry of QoS.

2.2 ORB structure

The generic QoS provisioning model can be applied to the architecture of an ORB. The architectural pattern used for ORB based systems is a layered pattern, with application objects on the top layer, the OS and network layer at the bottom and the ORB in the middle. Figure 3 shows the architectural layers of an ORB based system.

The application objects are faced with a number of interfaces to the ORB, such as the Client IDL stubs, the ORB interface, the skeletons and the object adapter. These ORB components are held together by the ORB core.

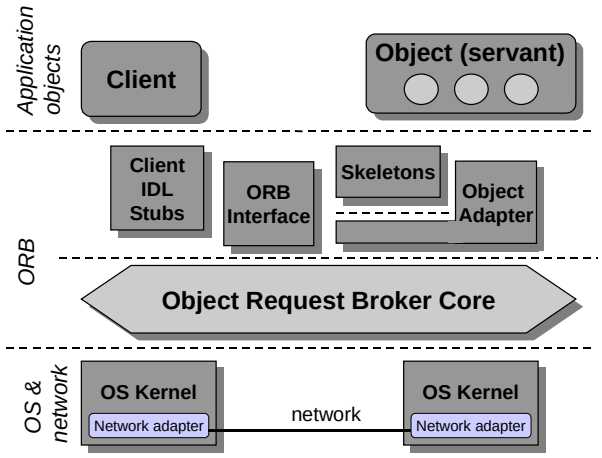

Figure 3: ORB layers and structure

The ORB core deals with the Operating System (OS) and the network environment.

In most current CORBA implementations the architectural distinction between the components can not be found in the implementation code. For example, the code that deals with ORB specific tasks is often intertwined with the code that deals with the network.

The boundaries between these ORB components can be seen as boundaries between Service Users and Service Providers. In the case of QoS provisioning, they can be used as potential boundaries between users and providers of QoS. Candidate QoS Providers can be: the ORB core, the (Portable) Object Adapter, a CORBA implementation object (servant), the network, the OS, the stub and the skeleton.

Based on these observations, we introduce the notion of QoS in CORBA systems.

2.3 QoS in CORBA

The next step is to devise a QoS architecture for CORBA systems. This includes the definition of QoS specific extensions for the current CORBA specification, in order to introduce QoS-awareness for CORBA based applications.

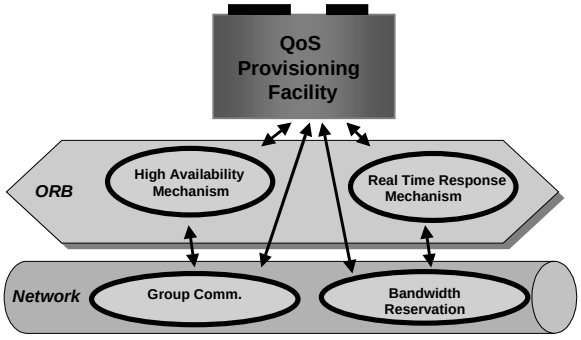

Figure 4: QoS provisioning extensions for CORBA

These extensions could best be positioned as a QoS provisioning facility, that could be used by the CORBA application objects to establish their required QoS. The main concern of the facility is to know the available mechanisms in the ORB and in the network that are suitable for controlling the QoS. Based on this knowledge, the QoS provisioning facility can configure the ORB and the network according to the QoS requirements of the application objects.

The detailed functionality of the QoS provisioning facility is for further study and outside the scope of this paper. To add QoS provisioning to CORBA systems it is necessary that the ORB has a number of QoS mechanisms available, which allow the QoS provisioning facility to control the QoS provisioning capabilities of the ORB.

The QoS experienced by CORBA application objects is determined by several factors such as for example the scheduling of the Operating System (OS) and the de-multiplexing of object invocations by the ORB [5]. Some QoS characteristics are directly related to the QoS characteristics of the network. In this paper we focus on how an ORB can exploit network specific features. Figure 4 depicts an ORB which provides mechanisms for high-availability and real-time that use group communication and bandwidth reservation features of the underlying network.

3 The Open Communication Interface

Ongoing work in the OMG proposes the Open Communications Interface (OCI). The OCI is part of the submission to the CORBA/IN Interworking RFP [1]. This section explains the rationale behind the OCI and describes the impact on current ORB implementations.

3.1 Adding dedicated protocols

The main purpose of an ORB is the transportation of (remote) object invocations between client and server objects. The structure and content of the invocation messages is defined in the Generic Inter ORB Protocol (GIOP) specification [6]. CORBA 2.x compliant ORBs have to implement IIOP, which maps GIOP requests to TCP/IP.

The current state-of-practice is that ORBs are provided as binary components, with no API for managing the transport protocol that is used. Most ORBs make direct use of the TCP/IP interface and hide all transport network details from the application objects. This makes it almost impossible to take advantage of network features that may be useful for an application.

An open interface that allows protocols to be added to the ORB would be very beneficial. This becomes even more apparent when the QoS parameters of the transport network enable the ORB to meet the QoS requirements of the applications objects.

The next section describes how the OCI meets the need for adding protocols to an ORB implementation.

3.2 OCI specification

Despite the fact that most ORBs are provided as a black box, some layering can be distinguished. Each ORB requires a layer that is responsible for exchanging messages, since object invocations require request and reply messages to be exchanged. GIOP is an example of a specification for the messaging layer. For transporting these messages a transport layer can be identified, such as TCP/IP.

The main purpose of the OCI initiative is create more openness for CORBA by defining standard interfaces, specified in IDL, for the messaging and transportation layers. The Remote Operation Interface (ROI) resides between the ORB Core and the messaging layer. The Message Transport Interface (MTI) resides between the messaging and transport layers. The OCI layers and the two interface sets are depicted in Figure 5.

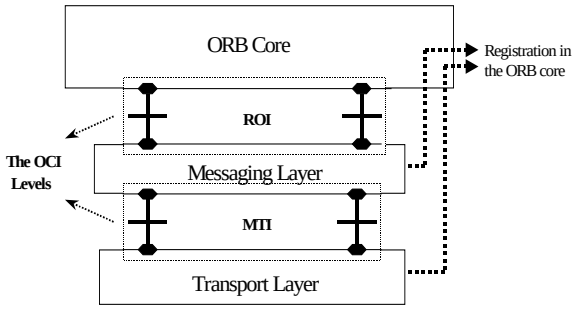

Figure 5: **Communication layers of an ORB**

A major benefit of the OCI is that new messaging and/or transport layers can be added to any ORB implementation because the OCI ensures that the layer is portable between ORB implementations. This leads to the concept of pluggable protocols. An implementation of the ROI or the MTI is called a protocol plugin.

The OCI is specified in terms of locality constrained IDL and a protocol plugin is the implementation of this IDL. The IDL is locality constrained, i.e. no remote interactions on these interfaces are allowed, because protocol plugins are local to the ORB Core. Currently, only detailed IDL specifications of the MTI are available and their definition is based on the Connector/Acceptor pattern [7],[8]. In the following sections we discuss the classes as defined for the MTI.

3.2.1 Client side classes

The following classes bear a responsibility at the client side of a transport plugin. They deal with establishing transport connections to servers and transporting request data.

The Connector class is responsible for establishing a transport channel between a client and server. This channel should behave as a full duplex communication channel. A Connector object creates a communication port and sends a connect request to the server through this port. After connection establishment, the connector object creates a Transport object and offers the reference of this object to the messaging layer.

The Connector Factory class is responsible for creating Connectors. A client may create a connector object whenever a connection to server is needed. A Connector Factory object manages all outgoing connections of a client by controlling the lifecycle of Connector objects.

3.2.2 Server side classes

The following classes bear a responsibility at the server side of a transport plugin. They deal with accepting transport connections from clients and transporting reply data.

The Acceptor class is responsible for accepting connections. An Acceptor object creates a communication port and waits for connection requests. After receiving a valid request the Acceptor creates a Transport object and offers the reference of this object to the messaging layer.

The Transport class is responsible for transporting messages received from the messaging layer. Fragmentation and re-assembly is a task of the Transport class. A Transport object is deleted when the underlying transport protocol closes the associated connection or when the messaging layer instructs it to close a connection.

3.2.3 Registration classes

The hooks for plugging in a transport layer are formed by two registry objects: the Connector Factory Registry and Acceptor Registry. A transport layer is registered by adding a Connector Factory object and an Acceptor object to the appropriate registries.

The registry for Connector Factory objects is available from the ORB core and the registry for acceptors is available from the Object Adapter. Incorporating these hooks into CORBA specification requires a revision of the standard.

3.2.4 Helper classes

The Buffer class is a helper class, which is used by the Transport class to manipulate transport data. A Buffer object holds data in an array of octets with a position counter. The position counter determines how many octets have already been sent or received. A buffer object can be used to provide an interface to a zero-copy buffer.

3.2.5 OCI interactions

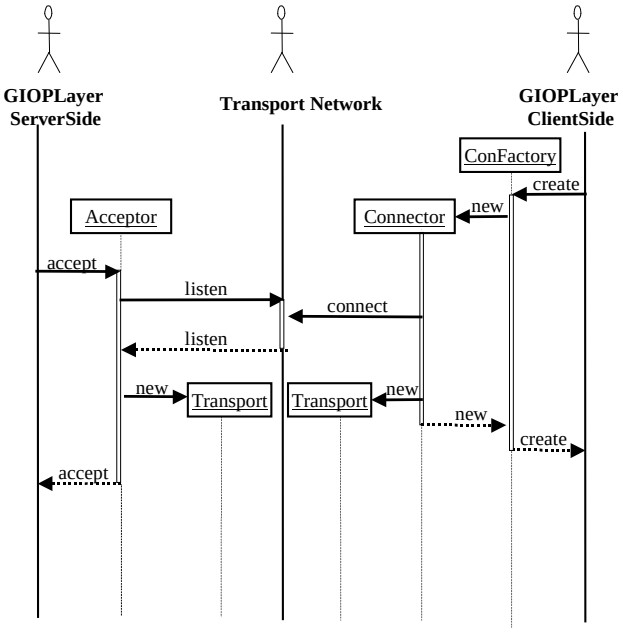

Figure 6: OCI connection establishment

Two phases of the OCI are relevant for explaining the lifecycle and interactions of the OCI objects. This first phase is the connection establishment phase. During this phase a transport connection is established, which is used during the second phase to transport messages. Figure 6 shows the interactions between the messaging layer (that uses the GIOP protocol in our case) and the OCI objects in a UML message sequence diagram. The 'listen' and 'connect' messages in the diagram are implementation specific and depend on the API of the underlying transport network.

The main purpose of the connection establishment phase is to create two associated Transports objects. The Transport objects can then be used to exchange (GIOP) messages. Figure 7 depicts the message sequence for a request from a client to a server. Note that the implementation of the Transport object can hide the fragmenting of messages from the GIOP layer.

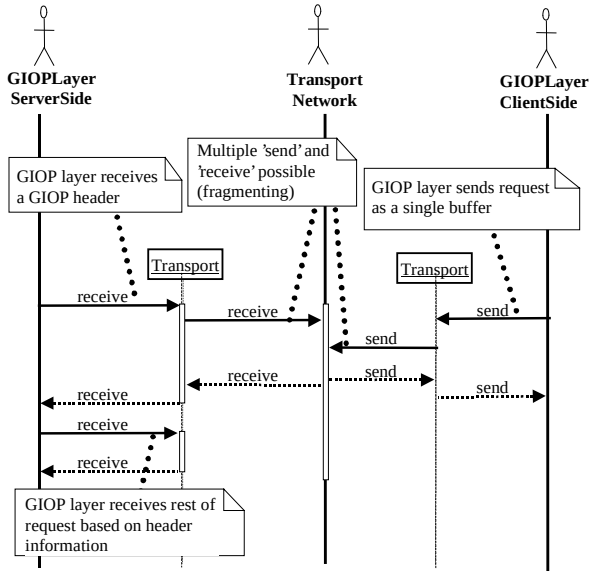

Figure 7: OCI message transport

3.3 Interoperability compromised ?

One of the major goals of the OMG is to define specifications for products that work together. Interoperability between software components that were developed for different hardware platforms and with different implementation languages is mandatory for all CORBA specifications.

IIOP is a key asset of the OMG to reach the interoperability goal. A first glance at the OCI solution may suggest that interoperability can easily be compromised when an ORB is extended with dedicated protocol plug-ins. This section explores how interoperability can still be maintained with pluggable protocols. We first describe how interoperability is achieved with IIOP and then how this is done with MTI compliant protocol plugins.

3.3.1 Interoperability with IIOP

CORBA 2.x compliant ORBs rely on IIOP for interoperability between ORB implementations from different vendors. Each ORB vendor implements an IIOP engine, inside its product. Mutual understanding of the messages generated by these IIOP engines is guaranteed because the on-the-wire encoding, the sequences of valid messages and the use of TCP/IP is specified in the standard. Client and servants can be implemented on top of these ORB implementations because a standardised interface to the ORB is available. The IIOP reference points are shown in Figure 8. The reference points are both horizontally and vertically directed, this is an important difference with the interoperability reference points defined by the MTI.

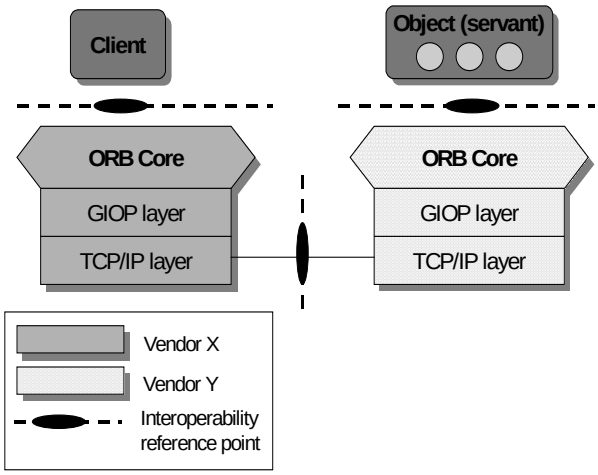

Figure 8: IIOP reference points

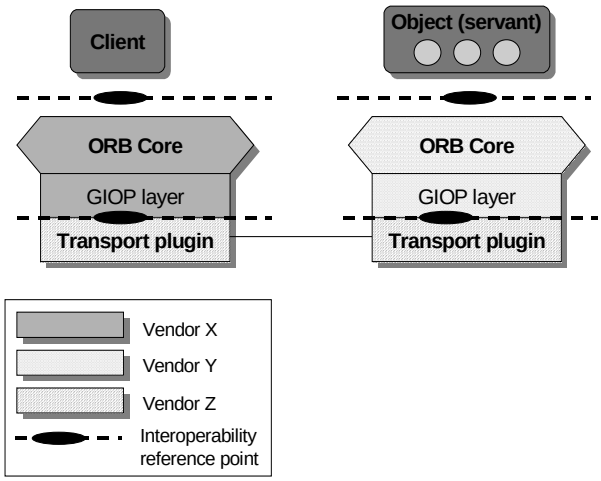

Figure 9: Reference points for protocol plugins

3.3.2 Interoperability with MTI

When ORBs are equipped with the MTI, the interoperability reference points are located at a different position. Protocol plugins encapsulate the valid messages and message sequences, so no agreement for the on-the-wire encoding is needed. The trade-off for loosing the network interoperability reference point is that the MTI itself becomes a reference point. Figure 9 shows how protocol plugins are positioned in a multi-ORB situation. Plugins can be built by a third party and plugged into various ORB implementations.

Changing the interoperability reference points allows extension of the ORB with specialised protocols. With these extensions the ORB can exploit specific features of the network, such as specific QoS characteristics, without compromising the interoperability of application objects.

The OCI provides a good means for extending ORB. If the QoS management functions are able to choose from a number of transport plug-ins, QoS provisioning becomes more feasible in CORBA based systems. In terms of the QoS provisioning model of Figure 2 a transport plug-in can be seen as a QoS mechanism.

4 Prototypes

This section describes our prototypes and their design. We have developed two protocol plugins that use the MTI interface to validate the usefulness of OCI as a building block for QoS provisioning. We first describe the design of a transport layer using UDP/IP and then extend this with the design of a transport layer based on the IP Multicast protocol. Before describing the design issues and patterns, we first describe the requirements of a transport plugin from the GIOP layer and the implications of multiple plugins for object references.

4.1 Transport layer requirements

The GIOP layer imposes a number of requirements on the transport layer. We summarise these requirements, taken from the CORBA specification [6]:

- Connection oriented
- Reliable data transport
- Data is transported as a stream (i.e., no restriction on the length of messages)
- Notification of connection loss

These requirements are very well met by the TCP/IP protocol, but the use of the TCP protocol implies that some of the QoS features of the ORB are fixed. For example, TCP is a point-to-point protocol, which does not exploit the inherent redundancy of network resources and the failure of a TCP connection usually means the loss of service from CORBA applications objects.

To use other protocols than TCP/IP for the implementation of a transport plugin, connection establishment, reliability and packet-to-stream conversion must be taken into account. The use of these protocols for an OCI-compliant transport plugin requires the design of additional functionality on top of the services provided by these protocols. For example, to have reliability with a UDP based transport extra actions are needed, such as acknowledgements and retransmission of data packets.

Moreover TCP/IP is a peer-to-peer transport protocol whereas IP Multicast has a many-to-many group dynamics i.e. a sender can send a message to a group of recipients at once. This property of IP Multicast requires more implementation to deal with group communication among IP Multicast group members. The next three sections describe the required extensions and implementations for supporting other transport protocols.

4.2 Interoperable Object Reference

Extending an ORB with transport plugins means that a CORBA object can potentially be reached through several communication paths. To establish a communication path from a client to a server object, the transport address of the server must be known by the client. CORBA server objects publish this transport address information as part of the Interoperable Object Reference (IOR). An IOR is a flexible data structure, which is generated at run-time by a server object and read by the client application object. The ORB at the client side is responsible for interpreting the contents of an IOR and the details are hidden from the CORBA application developer. Figure 10 shows how an IOR is structured.

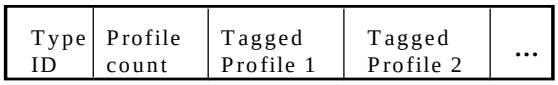

Figure 10: IOR structure

The TypeID indicates the type of an object. The value of a TypeId is derived from the interface definition (IDL)

Tag	Sequence Header	Port Number	Machine IP Address	Object Key

Figure 11: Specified Tagged profile for UDPIOP

Tag	Sequence Header	Port Number	Group IP Address (Class D)	Object Key

Figure 12: Specified Tagged profile for IPMIOP

of an object. The sequence Header field contains the length information of the rest of an IOR. Every IOR has one or more Tagged Profiles and each Profile contains the information for a specific protocol that can be used to access the object. Since we are interested in supporting other transport protocols than TCP/IP we focus on tagged profiles and the information stored in them. Figure 11 shows the design of a tagged profile for UDPIOP and in Figure 12 we show the design of a Tagged Profile for IP Multicast communication.

The Tag is a constant value, which indicates the protocol used in a Tagged Profile. OMG [6] has assigned the Tag value 01 for the IIOP protocol. In our experiments we picked an arbitrary number for each of the plugins. The rest of the Tagged Profile structure is very similar. For the UDP profile we use the IP number and a port number to identify a transport address. For the IP multicast profile, we use a class D multicast address and a port number. Both profiles include an object key, which identifies a particular instance of an object. The combination of TypeId, transport address and the object key should uniquely identify an object.

4.3 Implementing MTI with UDP/IP

UDP/IP is a connection less, thus does not meet the requirements of the GIOP layer. Therefore we have designed and implemented an additional protocol that leverages the features of UDP to the expected level. We denote our intermediate protocol as MTI-UDP protocol.

Connection number (Generated by a counter)	**Message level** (= 0 for control messages = 1 for data messages)	**Message type** (= C for Conn. Request, = R for Conn. Reply, = L for packet lost, = D for packet redelivery)	**Packet number & Data packet (s)** (Is empty if Message type is not L or D)

Figure 13: MTI-UDP Transport level message format

MTI-UDP can establish a connection by sending a transport layer ConnectRequest message (to the remote transport layer, which also contains MTI-UDP) and request for a connection establishment. The remote transport layer responses to this ConnectRequest with a Connection Reply, indicating that it can accept a request. MTI-UDP supports two other messages for reliability: PacketLost and PacketRedelivery. Figure 13 shows a format in which these messages can be embedded. These messages are sent in a UDP Datagram packet therefore host and port information is available in the header of packets.

4.4 Implementing MTI with IP Multicast

We have implemented and named another protocol as MTI-IPM, which uses IP Multicast for transport. MTI-IPM is a connection oriented, reliable, stream based transport protocol. MTI-IPM is a one-to-many protocol. GIOP can use MTI-IPM to multicast a request message but in connection oriented, reliable and stream based manner.

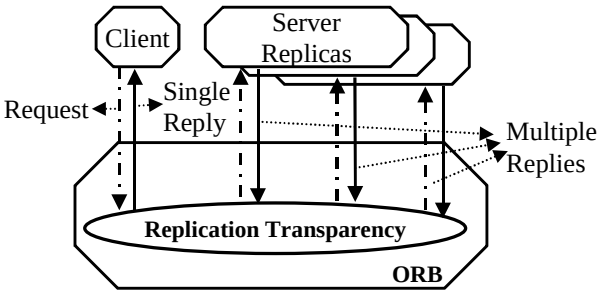

Figure 14: Replication transparancy by Group Abstraction

Connection establishment and reliability issues are solved with a similar approach to MTI-UDP protocol, by sending out additional control messages using the IP multicast service. However in this prototype we multicast a message to a group of server object replicas. Therefore the sender MTI-IPM will receive multiple replies. We have implemented group abstraction in MTI-IPM to accept multiple replies from server replicas but forward one message to GIOP in client side. The initial version of the prototype returns the first reply message generated by a member server object replica to the client.

An important decision was to put the IP Multicast group address in the IOR. This IP Multicast address is a unique address for each group of replicas. It is generated in all of the server objects of the group in the start up phase, without requiring any run time agreement between the members. The object key and object name are used to generate a unique multicast address.

A server object replica can become a member of an IP Multicast group by joining a group. Joining a group is supported by the Java IP Multicast API through the Join method.

4.5 Prototype evaluation

We have developed two transport plug-ins. We have successfully plugged them into the ORB. We have experienced that the standard messaging protocol (GIOP) can smoothly interact with other protocols than TCP/IP if

these protocol are leveraged to be connection oriented, reliable, and stream oriented.

We are aware that the IP multicast plugin does not support all the nice features of a flexible and dynamic group communication protocol. Currently, the number of replicas in a group is static and no support for synchronisation between group members is implemented. These features have not been added to the prototype, because our focus was on evaluating the MTI as a means for QoS provisioning in CORBA and not on implementing a feature rich group communication protocol.

5 Evaluation and related work

5.1 Evaluation of MTI

We have implemented two protocol plugins that comply with the MTI interface of OCI. The implementation activities have focused on developing a transport plug-in that supports replication transparency and encapsulates IP Multicast.

Unfortunately at this stage only one ORB implementation, i.e. ORBacus [9], supports the MTI. It is therefore not yet possible to test if another ORB implementation can seamlessly integrate with our plugins.

From our prototyping efforts we have learned that the OCI specification should be further detailed in order to remove some ambiguities. An example of such an ambiguity is the synchronisation between the messaging layer (GIOP) and the transport layer with respect to the Buffer object (see Section 3.2.4 for an explanation of the Buffer object). Another ambiguity is the management of the protocol plugins. When multiple transport protocols are plugged in to an ORB, it is not defined which transport protocol should be used. The choice for a particular plugin is left to the ORB core and cannot be influenced by the CORBA application objects.

Although the OCI specification contains some ambiguities, our overall experiences with OCI are positive. We think that the ability to plugin a protocol to an ORB makes CORBA a more open and extendible architecture. The OCI provides the necessary functionality to add transport protocols to an ORB, but could be extended in at least two ways. First, interfaces for managing the configuration of messaging layers and transport layers must be added. Second, a management interface for adding plugins at run-time would prevent plugin developers from recompiling the entire ORB. This could even allow for dynamically downloading plug-ins.

5.2 Related work

In our IP multicast plugin we claim that group communication can increase the availability of CORBA objects by making them fault-tolerant. The current implementation of the multicast plugin provides only limited support for fault tolerance. More advanced group communication mechanisms, such as ISIS [10], Horus [11] or Ensemble [12] could be used to really improve the availability of CORBA objects.

Orbix-Isis [13] is an example implementation that addresses the integration of a group communication protocol with a standard ORB. However, the integration is done as a proprietary extention to the Orbix ORB, and no public interface is defined between the ORB code and the Isis code. The Orbix-Isis integration effort can not be reused with other ORB implementations and therefore interoperability between Orbix-Isis and other ORB implementations seems very unlikely.

In [14] and [15] an architecture is described for Quality of Service of CORBA Objects. This architecture is called QuO and extends IDL with a QoS Description Language (QDL). QDL is used to specify an application's expected usage patterns and QoS requirements for a connection to an object. The application can adapt its behaviour depending on the status of a connection, but it cannot influence the QoS, which is possible in our approach.

Another relevant initiative in this direction is *The Realize middleware* [16]. Realize extends CORBA with object replication. It does this by intercepting IIOP messages at the TCP/IP layer, and diverting them to a so called Replication Manager that uses the earlier mentioned QoS Description Language to decide how many replicas are required. The Totem multicast group communication system is used for multicasting the messages. Our approach manipulates messages at the GIOP level, while this approach manipulates messages at the lower TCP level. possibly causing extra overhead in analysing the content of the TCP request. Another difficulty with this approach is that the TCP messages have to be intercepted in an operating system dependent manner. In addition, the notion of plugging in a different protocol or QoS mechanism is not taken into account.

Globe [17] is an object based framework in which an object (or servant) is not necessarily located on a single host like with CORBA. Instead it is left to the developer how an object is distributed over a number of hosts. Globe claims to offer an efficient solution for replication, especially in a wide-area context. A benefit of Globe is more flexibility is choosing the replication (or another QoS related) mechanism, but compliance with CORBA is not possible (without loosing this flexibility) and interoperability in general between two independently developed Globe-based applications might be a problem. The implementation of Globe seems much less developed than the implementation of CORBA.

HTTP-NG [18] is a proposed solution to improve the current HTTP protocol. The architecture of HTTP-NG is based on a three layer model, consisting of a transport layer, a remote invocation layer and a web application layer. The architecture enables web applications to run over other protocols than TCP/IP and seems to match

closely with the layering of OCI. Although the current focus of HTTP-NG is on supporting web applications and not on generic distributed object systems, our approach for QoS provisioning could be applied to HTTP-NG.

6 Conclusion and Future work

6.1 Conclusions

We have successfully used the Open Communication Interface for extending an ORB with specialised protocols instead of the normal IIOP. The ability to extend an ORB with specialised protocols is a necessary, but not sufficient, requirement for QoS provisioning in CORBA. To this extent OCI seems very beneficial but also requires a more detailed specification. Our experience with the OCI shows that it provides the necessary interfaces to plugin other transports than TCP/IP to an ORB. However, the specification is not clear in all parts and could cause interoperability problems when implemented by other ORB vendors. For example, when multiple transport protocols are plugged in to an ORB, it is not defined which transport protocol is used. This is left to the ORB developer. The problem of managing communication layers becomes even more apparent when other messaging layers than GIOP become available.

Very important is that OCI keeps interoperability. Instead of standardising the protocol used to communicate between client and server, in OCI an ORB internal interface is specified to achieve interoperability.

Our IPM Prototype is a first step towards replication transparency and reliability in CORBA. It is an example of how to extend CORBA using the OCI to achieve QoS provisioning. We have found that OCI has some ambiguities, but most importantly OCI does not adequately specify a policy interface to select a plug-in. This policy interface is required, and should be standardised to be able to create portable application code.

OCI specifies two interface layers, we have only used and evaluated the MTI interfaces. The ROI interfaces, which allows a designer to replace GIOP with another messaging layer, is something for which we see no immediate need.

6.2 Future work

In the current prototype of the IP Multicast plug-in the group abstraction and virtual synchrony are very primitive. We are considering using Cornell's Ensemble or some other group communication mechanism to make our plug-in more sophisticated.

Although we consider IP Multicast to be a natural fit for a plug-in to achieve reliability, we want to try different approaches to be able to compare them and to do some benchmarking.

In our work so far we have limited ourselves to an IP network, but in some cases it could be much more interesting to use other kinds of networks, e.g. ISDN or ATM.

In the current situation, what plug-in's will be available and used has to be decided design-time. If a good policy object would exist, the decisions on what plug-in's to have available could be delayed untill compile-time, and the decision which plugin is used until run-time. But ideally, what plug-in will be available and what will be used will both be decided at the latest time possible, thus at run-time. This approach seems feasible for certain implementation languages. For example in a Java scenario one could make a plug-in-repository available, which has Java byte-code that implements the OCI interface together with a formal description on the required (network and computing) resources and offered QoS. Depending on the required QoS from the application (and thus from the user), the policy object can query the plug-in repository which plug-in best fits the needs.

Acknowledgements

The authors would like to thank Olaf Kath from Humboldt University for fruitful discussions.

Reference List

[1] OMG. Revised version of the AT&T/TelTec/GMD Fokus IN/CORBA submission. 1998. OMG document number: telecom/98-06-03

[2] Information Technology - Quality of Service - Guide to Methods and Mechanisms - Technical Report Type III. 1997. Notes: ISO/IEC TR 13243 (Editor's draft 1.0) JTC1

[3] ISO/IEC. Information Technology - Quality of Service - Framework. 1997. Geneva. Notes: (ISO/IEC JCT1/SC21 N13236)

[4] L.J.N. Franken, Quality of Service Management: a Model-Based Approach 1996. University of Twente, CTIT. Ph.D.

[5] D.L. Levine, S. Flores-Gaitan, and D.C. Schmidt, An Emperical Evaluation of OS Support for Real-time CORBA Object Request Brokers. June 1999. Vancouver, British Columbia, Canada.

[6] OMG. The Common Object Request Broker Architecture: Architecture and Specification, formal/98-01-01. 1998. Revision 2.2

[7] D.C.Schmidt, Acceptor and Connector: Design Patterns for Initializing Communication Services eds. R.Martin, F.Buschman, and D.Riehle. 1997. Addison-Wesley. Reading, MA.

[8] D.C.Schmidt, A.Gokhale, T.H.Harrison, and G.Parulkar. Architectures and Patterns for Developing High-performance, Real-time ORB Endsystems. In: *Advances in Computers*, ed. Marvin Zelkowitz. Academic Press, 1999.

[9] OOC. http://www.ooc.com/ob. 31-1-1999.

[10] ISIS project. June 1999. http://simon.cs.cornell.edu/Info/Projects/ISIS/ISIS.html

[11] R.v. Renesse, K.P. Birman, and S. Maffeis, Horus, a flexible Group Communication System *Communications of the ACM*, vol. 1996.

[12] Hayden, Mark. The Ensemble System. 1998. Cornell University Technical Report, TR98-1662.

[13] IONA Homepage. June 1999. http://www.iona.com/

[14] Zinky, J., Bakken, D. E., and Schantz, R. E. Architectural support for QoS for CORBA Objects. 1998.

[15] R. Vanegas, J.A. Zinky, J.P. Loyall, D.A. Karr, R.E. Schantz, and D.E. Bakken, QuO's Runtime Support for Quality of Service in Distributed Objects AnonymousAnonymous1998. Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98).

[16] P.M. Melliar-Smith, L.E. Moser, V. Kalogeraki, and P. Narasimhan, The Realize middleware for replication and resource management September 1998. In Proceedings of Middleware'98.

[17] A. Bakker, M.v. Steen, and A.S. Tanenbaum, Replicated Invocations in Wide-Area Systems. 1998. Proc. Eighth ACM SIGOPS European Workshop.

[18] HTTP-NG. June 1999. http://www.w3.org/Protocols/HTTP-NG/