Enabling Flexible QoS Support in the Object Request Broker COOL

Tom Kristensen and Thomas Plagemann
Center for Technology at Kjeller (UniK), University of Oslo
http://www.unik.no/~tomkri, ~plageman

Abstract

Support of end-to-end Quality-of-Service (QoS) and appropriate high-level programming abstractions are two crucial factors for the development of future telecommunication services and distributed multimedia systems. Today's middleware systems, like the Common Object Request Broker Architecture (CORBA), support high-level programming abstractions, but they do not appropriately support the demands from soft real-time and multimedia applications. We develop a flexible and adaptable middleware that supports a broad range of QoS requirements of distributed multimedia applications. In this paper, we present design, implementation, and preliminary performance evaluation of our first prototype. We have extended the object and message layer of the CORBA implementation COOL with QoS support by (1) enabling QoS specifications for method invocations, (2) integrating bilateral QoS negotiation between client and server, and (3) implementing a simple unilateral QoS negotiation between message layer and transport layer in the Object Request Broker (ORB). In the transport layer of COOL, we have integrated Da CaPo to flexibly support the various QoS requirements. Our work demonstrates that it is possible to support QoS with minor changes in GIOP and the use of a QoS supporting transport protocol.

1 Introduction and Motivation

The availability of multimedia personal computers and workstations, internet services, and high-speed network services has drastically increased the use (and need) of distributed multimedia applications for commercial, private, and other purposes. Such applications range from distance education applications to integrated command and control systems. Independent of the particular application domain, we identify two common factors that are crucial for the development of future telecommunication services and distributed multimedia systems:

- end-to-end Quality-of-Service (QoS) support and
- high-level programming abstractions.

On a first glance, it seems that the necessary technology is already commercially available. Asynchronous Transfer Mode (ATM) networks offer high bandwidth and (guaranteed) QoS to higher layer protocols. Furthermore, implementations of the distributed object-oriented middleware standard Common Object Request Broker Architecture (CORBA) from the Object Management Group (OMG) provide high-level programming abstractions. However, nearly all CORBA implementations (e.g., IONA's Orbix and Inprise's VisiBroker) are based on the communication protocols TCP/IP. It is well known that these transport and network layer protocols do not support QoS. To make things worse, CORBA does not support QoS at the object level, and multimedia streams are not integrated in the Object Request Broker (ORB).

In the MULTE (Multimedia Middleware for Low-Latency High-Throughput Environments) project, we contribute to the solutions of these problems by designing, implementing, and evaluating flexible and adaptable middleware systems that support a broad range of QoS requirements, e.g., low latency, high throughput, and controlled delay jitter. We use IPv4 and IPv6 over traditional 155 Mbit/s ATM and 2.4 Gigabit/s ATM technology developed at the Washington University St. Louis, and the next generation internet protocols IPv6 and RSVP (Resource Reservation Protocol) to provide network QoS. Above the network layer, a flexible (re)configurable protocol system replaces the traditional protocols, e.g., TCP and GIOP (General Inter-ORB protocol), and represents the core element in the MULTE middleware. A flexible protocol system allows dynamic selection, configuration and reconfiguration of protocol modules to dynamically shape the functionality of a protocol to satisfy specific application requirements and/or adapt to changing service properties of the underlying network. The basic idea of flexible end-to-end protocols is that they are configured to include only the necessary functionality required to satisfy the
application for the particular connection. This might even include filter modules to resolve incompatibilities among stream flow endpoints and/or to scale stream flows due to different network technologies in intermediate networks. The goal of a particular configuration of protocol modules is to support the required QoS for requested connections. This will include point-to-point, point-to-multipoint, and multipoint-to-multipoint connections.

Our first prototype in MULTE is based on the CORBA 2.0 [15] implementation COOL 4.1 and the flexible protocol system Da CaPo (Dynamic Configuration of Protocols) [19]. We have extended the object and message layer of COOL with QoS support by (1) enabling QoS specifications for method invocations, (2) integrating bilateral QoS negotiation in an extended GIOP protocol, and (3) implementing a simple unilateral QoS negotiation between object layer and transport layer in the ORB. In the transport layer of COOL, we have integrated Da CaPo to flexibly support various QoS requirements. The prototype is implemented on top of the real-time µ-kernel ChorusOS 3.2 to support real end-to-end QoS with appropriate real-time scheduling for threads performing time critical communication and application tasks. In this paper, we concentrate on the mechanisms to enable QoS specification and negotiation through all layers of COOL. With respect to this focus, we present design, implementation, and preliminary performance evaluation of this prototype. Mapping of QoS parameters between the different layers in COOL, corresponding reservation of resources in COOL and Chorus, and support of streams in COOL are beyond the scope of this paper, and solutions to these problems will only be indicated.

The remainder of the paper is structured as follows: Section 2 briefly introduces CORBA and COOL, and Section 3 discusses related work. Our extensions of COOL at object and message layer are described in Section 4, and Section 5 deals with extensions to the transport layer. An overview of the current state of the implementation and a preliminary performance evaluation is given in Section 5. We conclude the paper with a short summary, critical discussion of the presented results, and an outlook to future work in MULTE.

2 Brief Introduction to CORBA and COOL

A CORBA object is “an identifiable, encapsulated entity that provides one or more services that can be requested by a client” [15]. Public interfaces of objects are described in the CORBA Interface Definition Language (IDL). IDL hides the underlying object implementation: clients simply use this interface to invoke methods without caring about location, platform, or implementation of the object. The ORB is the heart of CORBA and implements this transparency. Important interfaces and components of the ORB are: IDL stubs, IDL skeletons, Object Adapter, and ORB core. Normally, IDL compilers are used to generate IDL stubs and IDL skeletons out of IDL specifications. IDL stubs are located at the clients site and marshal parameters in requests and unmarshal parameters in responses. IDL skeletons unmarshal parameters of incoming requests, upcall the object implementation to perform the requested operation, marshal the results and return them to the client. Services provided by the ORB through an Object Adapter often include: generation and interpretation of object references, method invocation, security of interactions, object and implementation activation and deactivation, mapping object references to implementations, and registration of implementations. In the ORB core, OMG’s standard GIOP uses seven messages to send method invocations from client to object implementation, return the response back to the client, cancel requests, handle errors, etc. Therefore, we call this layer in the ORB core the message layer. GIOP in the message layer combined with TCP in the transport layer forms the Inter-ORB Protocol (IIOP), which is mandatory for a CORBA-compliant implementation.

COOL 4.1 [4] is such an CORBA-compliant implementation. Its main components are illustrated in Figure 1. COOL is a modular ORB with focus on the ability to exchange or integrate additional modules with custom made ones. This is especially important for the communication protocols used by COOL. To enable support for multiple protocols and to ease integration of new protocols, COOL is built with generic communication layers [5] for message layer and transport layer protocols. The generic communication layers can be seen as wrappers for the underlying protocols. The supported transport layer protocols are TCP/IP and Chorus IPC, with an associated notification mechanism for each. The generic message protocol layer represents the communication part of operation invocations. COOL supports GIOP and the proprietary COOL protocol in the message layer. It is worth noticing that the Object Adapter is present at both the server side below skeletons and at the client side below stubs. The Object Adapter is designed to optimize colocated scenarios, where client and server runs on the same endsystem.

A method invocation in COOL initiates the GIOP protocol to send a Request message to the object implementation, which in turn requires the establishment of a TCP connection. The GIOP Request message is sent via this connection to the server. The server processes the Request, and sends a Reply message with the results back to the client using the same TCP connection.

Our QoS extensions to COOL are divided in two parts:
Object and message layer extensions add to COOL the ability to specify QoS in IDL definitions for method invocations. Furthermore, they include the propagation of QoS specifications via the ORB to the peer side and to the transport layer.

Transport layer extensions are performed by integrating the flexible communication system Da CaPo into COOL.

3 Related Work

There are several research activities in the area of end-to-end QoS support in distributed multimedia systems. We concentrate our discussion of related work on OMG activities and QoS support in middleware.

OMG has defined an architecture for multimedia streams called OMG A/V Streams [16] based on CORBA, with a similar approach to IMA MSS [12][13]. The data flow takes place over separate channels outside the ORB core/computational model. OMG A/V Streams only define a limited set of interfaces, especially the control and management interfaces for streams. The Real-Time CORBA specification (RT-CORBA) [17] addresses how the ORB ensures end-to-end predictability with respect to priority propagation, threading schemes, explicit binding, and protocol policies and configurations. Explicit binding is used according to the CORBA Messaging specification [18]. Protocol policies guide in RT-CORBA the selection of protocols, and the configuration of the protocol is depending on the flexibility and nature of the protocol. The configurable properties for TCP in RT-CORBA include send and receive buffer sizes, keep alive, and do not route. There is no way to reconfigure protocols after binding time in RT-CORBA.

There is a considerable amount of research on end-to-end QoS support. Main concern in most of these approaches is appropriate resource management. A survey of these so-called QoS architectures can be found in [1]. In contrast, flexible protocol approaches put main emphasis on the adaptation of protocol functionality to support the requested QoS and to increase protocol performance by decreasing its complexity. A more detailed discussion of flexible protocol systems can be found in [19]. Resource management can be regarded as an orthogonal concept to protocol flexibility. In this work Da CaPo (see Section 5.1) is used as a flexible communication subsystem.

The next step of this development is to integrate the results and experiences of the above mentioned research prototypes into ORBs. The ReTINA project has summarized requirements for a multimedia ORB in [3]. The DIMMA project at ANSA [7] has implemented a real-time, multimedia ORB. The endsystem architecture TAO [21] provides static real-time support for CORBA. The Adapt project [10] using Ensemble [11] has extended COOL with configurable protocol stacks for adaptive multimedia applications.

Apapt exploits the open binding abstraction, providing a number of low-level primitives for other entities use. Also, Adapt only addresses stream communication. Ensemble is also used by the Electra ORB [23] for reliable, fault-tolerant computing, with emphasis on group communication. GOPI [6] is a generic middleware platform, which can be extended with new protocols, new QoS mapping schemes provided by the programmer and so on. Another flexible ORB is Jonathan [8], where arbitrary binding mechanisms may be chosen for the binding between interacting objects.

QoS in middleware has many facets and approaches, and a research prototype of an ORB with end-to-end QoS support developed in an integrated manner is needed. Our work differs from others in that the extensions are carried out using a CORBA conformant platform, and in that the QoS extensions offered as options for the developer to use or not. This paper describes a first step, with focus on QoS support in interfaces and protocol functionality.

4 Object and Message Layer QoS Support

The general task of QoS support at object and message layer can be split up into the following areas of concern: (1) object based QoS specification, (2) QoS negotiation between client and object implementation, and (3) QoS negotiation between message layer and transport layer [2].

4.1 QoS Specification

Traditionally, QoS is related to a particular type of service, namely associations, like transport connections or virtual circuits in ATM. A certain QoS is specified and negotiated for an association and the negotiated QoS is valid for the lifetime of the association, if not re-negotiated. In the context of object-oriented middleware, there is no directly corresponding notion of such a service. Objects, bindings, method invocations, and even para-
ters in method invocations may be seen as a service. Therefore, several alternatives for QoS specification and negotiation in middleware exists with different degrees of granularity, flexibility, and complexity:

- **QoS per object**: (persistent) objects implement and provide methods to remote clients. Thus, the object itself can be seen as a service. It can be associated with a maximum QoS that can be provided by the object, e.g., maximum resolution of an image that can be returned on method invocation. However, clients that are connected via low performance links to the object should be generally enabled to request the same image with a lower resolution, i.e., lower QoS. In other words, specifying QoS per object poses strict limitations on the implementation and use of the object. It would be impossible to design an object with different QoS for different methods and parameters.

- **QoS per binding**: a binding in CORBA is an association between the client and a server. The binding is implicitly set up during the first method invocation. Subsequent method invocations use the same binding, but changes in QoS requirements have to be renegotiated for the binding, which in turn requires renegotiation of the transport connection.

- **QoS per method invocation**: each method invocation is marshaled in a single Request message and the result is returned in a single Reply message. These messages are transported over one transport connection and can easily be associated with QoS. However, subsequent method invocations with different QoS require renegotiation of the transport connection. Furthermore, all parameters in a method will have the same QoS.

- **QoS per parameter in each method invocation**: specifying QoS per parameter is the most flexible approach, but also the most complex approach. Specifying QoS per parameter does not fit well into the GIOP scheme using one Request message per invocation, because parameters with different QoS requirement would need each a particular transport connection, which results in the need for multiple messages.

Obviously, QoS per object and QoS per parameter are not appropriate solutions. If QoS re-negotiation is supported, the differences between QoS per binding and QoS per method are only marginal. Therefore, we have decided to support QoS per binding and QoS per method by implementing a method called `setQoSParameter` to enable clients to explicitly define QoS requirements. There are two basic possibilities for the client:

- Never call `setQoSParameter`: no QoS support is required and standard GIOP can be used.
- Call `setQoSParameter`: each method invocation will be supported (if possible) according to the QoS requested in the last `setQoSParameter` call. By calling `setQoSParameter` only once in the beginning of a binding, a per binding QoS will be supported, and setting QoS before each method invocation results in QoS per method.

In order to allow QoS specification for method invocations, we have modified the COOL IDL compiler, called Chic, because it is necessary to generate stubs that handle the QoS specification. Chic uses template files to generate stub source files during compilation of the IDL definitions. These template files are modified by adding the method `setQoSParameter(struct QoSParameter** qp)` in the stub. The client uses this method to specify QoS requirements in an array of QoSParameter structures (see Figure 2-ii) and communicates it to the stub by calling the method. The stub handles the propagation of these QoS requirements to the message and transport layers in the ORB.

Calling the `setQoSParameter` method transforms the implicit binding between the client and the object implementation on the server side to a explicit binding. This explicit binding is controlled by the client application when specifying requirements.

### 4.2 Extensions in GIOP

In order to negotiate the requested QoS for a method invocation between client and server, it is necessary to either introduce a new object level negotiation protocol, or to extend GIOP. With respect to our goal, to introduce only minimal changes and support backwards compatibility, we have decided to implement QoS negotiation with minimal extensions in GIOP and allow the usage of standard GIOP. These extensions are concerned with the following three tasks: (1) differentiating both GIOP versions, (2) integrating QoS parameters in GIOP messages, and (3) the negotiation rules.

We use the version field in the GIOP message header (see Figure 2) to inform the receiver of a message whether standard GIOP (i.e., major version number 1, minor version 0) or our QoS extension (i.e., major version number 9, minor version 9) is used.

The QoS requirements are specified by the client and communicated to the stub by the `setQoSParameter` function. We have extended the format of the Request
message to include these QoS parameters by adding the field `qos_params` (see Figure 2-ii), and we have extended the IDL compiler Chic to marshal method invocation and QoS parameters into the extended Request message.

```
struct MessageHeader {  // GIOP header
    char magic[4];
    Version GIOP_version;
    boolean byte_order;
    octet message_type;
    unsigned long message_size;
};
enum MsgType {  // GIOP messages
    Request,
    Reply,
    CancelRequest,
    LocateRequest,
    LocateReply,
    CloseConnection,
    MessageError
};
```

(i) Standard GIOP header and message types

```
struct QoSParameter {
    unsigned long param_type;
    unsigned long request_value;
    long max_value;
    long min_value;
};
struct RequestHeader {  // the only GIOP message modified
    IOP::ServiceContextList service_context;
    unsigned long request_id;
    boolean response_expected;
    sequence<octet> object_key;
    string operation;
    sequence<QoSParameter> qos_params;  // only field added
    Principal requesting_principal;
};
```

(ii) The QoSParameter struct and extended Request header

The request message is sent over a transport connection with corresponding QoS to the server. In case of the first method invocation, it is necessary to establish a new transport connection with corresponding QoS. Subsequent method invocations might use the same transport connection. However, changes in QoS requirements have to be reflected in reconfigurations of the transport connection. If it is not possible to satisfy the QoS requirements at the transport layer, we inform the calling client using an exception. Otherwise, the Request message with QoS parameters will be sent to the server. The server unmarshals the Request message and passes it to the object implementation. If the object implementation is not able to support the requested QoS, it sends a negative acknowledgement (NACK) to the client with the standard CORBA exception mechanism (see Figure 3-i). Otherwise, the server returns the results within a standard Reply message with the requested QoS (see Figure 3-ii).

![Figure 2: GIOP message header and extended Request header](image1)

![Figure 3: QoS negotiation scenarios](image2)

In summary, QoS negotiation at object and message level can be performed by using the version field in the GIOP header, extending the Request message with QoS parameters and utilizing the exception message in case the requested QoS cannot be supported. Figure 4 illustrates the entire QoS negotiation procedure at object and message layer in COOL.

![Figure 4: Combined method invocation and QoS negotiation](image3)
4.3 Interface to Transport Level

In order to pass the QoS parameters from message layer to transport layer, the abstract class defining the generic transport protocol is extended with the setQoSParameter method. This method is inherited by the class actually implementing the transport protocol. Obviously, TCP does not implement the setQoSParameter method, but Da CaPo does and setQoSParameter propagates the QoS parameters to Da CaPo. Within Da CaPo, these QoS parameters are mapped to a particular protocol configuration, network resources, and operating system resources. If it is impossible for Da CaPo to reserve sufficiently enough resources, it informs the client with an exception that it cannot support the requested QoS. Thus, this interface implements a simple unilateral QoS negotiation.

5 Transport Level QoS Extensions

On the Chorus OS platform, the COOL ORB supports TCP/IP and Chorus IPC at the transport layer. None of these protocols support QoS and the necessary flexibility. Therefore, we add Da CaPo as a third alternative in the transport layer. In this section, we briefly explain the general idea of Da CaPo, its re-design for the Chorus µ-kernel, and its integration into the COOL architecture.

5.1 Da CaPo Overview

The Da CaPo (Dynamic Configuration of Protocols) system is based on a three layer model that splits communication systems into the layers A, C, and T and is described further in [19] and [20]. Endsystems communicate via the transport infrastructure (layer T), representing the available communication infrastructure with end-to-end connectivity (i.e., T services are generic). In layer C, the end-to-end communication support adds functionality to T services such that at the AC-interface services are provided to run distributed applications (layer A). Layer C is decomposed into protocol functions instead of sublayers. Each protocol function encapsulates a typical protocol task like error detection, acknowledgment, flow control, de- and encryption, etc. Data dependencies and independencies between protocol functions are specified in a protocol graph. Protocol functions can be realized by different protocol mechanisms, for example, the function error detection can be performed by mechanisms like parity bit, CRC16, CRC32, etc. Protocol mechanisms correspond to specifications and can be implemented as software or hardware modules. Modules implementing the same protocol functions are characterized by different properties such as throughput characteristics or degrees of error detection and correction. The unified module interface allows free and unconstrained combination of modules to protocols. Naturally, different protocol configurations, i.e., different combination of modules in a so-called module graph, support different QoS.

Applications specify their requirements within a service request, and Da CaPo configures in real-time layer C protocols that are optimally adapted to application requirements, network services, and available resources. Figure 5 illustrates the Da CaPo layers, the runtime environment, and the module graph building the communication protocol.

The original version of Da CaPo is designed and implemented on SunOS 4.1.3, where Da CaPo is linked together with the application with a single thread [20]. The new design and implementation of Da CaPo uses the multithreading support on the target platforms ChorusOS and SunOS. Each module in Da CaPo is executed by a single thread. The Da CaPo modules are C++ objects inheriting a base class, the modules implement the packet handling methods for data and control information. Actual system calls and library routines available on the different operating systems are hidden from the module programmer in C++ wrappers. Modules exchange pointers to packets over message queues as illustrated in Figure 6. Each module has two message queues associated: one for data and one for control information. The packets are situated in shared memory accessible by Da CaPo modules, and optionally by network device drivers and applications. The management component is responsible for configuring the module graph, monitoring, reconfiguration, and signalling.

5.2 Integration into COOL Communication Architecture

The COOL ORB communication subsystem is split into two parts as mentioned in Section 2, where a number of message protocols is able to use a number of underlying
transport protocols. As illustrated in Figure 7, there exist two ways to integrate Da CaPo into COOL’s communication architecture. Alternative (i) in Figure 7 shows Da CaPo integrated as another transport protocol below the generic transport layer. Da CaPo is then forwarding messages formatted according to the message protocols above. Alternative (ii) in Figure 7 is a more Da CaPo centric approach, where message protocols are seen as ordinary Da CaPo modules performing this specific task. In order to use the latter approach, message protocols have to be wrapped into Da CaPo modules performing COOL specific functionality regarding formatting of incoming and outgoing messages, interacting with client side stubs, and interacting with server side object adapter to locate object implementations.

In the extended COOL prototype, alternative (i) is implemented to develop the extensions described in Section 4 in parallel with the integration of Da CaPo. This way to integrate Da CaPo follows the generic communication framework in COOL and is easier to implement.

The generic transport protocol is represented by a C++ class wrapped around Da CaPo, that implements the virtual methods in order to perform requests and replies. That is to send the formatted message from a message protocol to perform a request, and wait for and receive a reply in form of a message. A new A-module in Da CaPo makes up the interface between the module graph and COOL runtime.

In Figure 8 the class hierarchy for the transport protocol layer is shown. The integration of the transport protocols are encapsulated in classes, where the generic transport protocol is represented by the _COOL_ComChannel class. The actual implementations inherits from this class, and implements the virtual methods from _COOL_ComChannel to perform it’s functionality. For Chorus IPC buffering is done transparent by the communication subsystem in ChorusOS. But the TCP/IP implementation needs to handle buffer management, this is done in the _TcpBuffer class. Da CaPo handles its own buffers in the Da CaPo runtime environment and only inherits _COOL_ComChannel. In addition the base class _ComManager is specialized for the different transport protocols in order to manage and control the different communication channels or connections in use. The class COOL_InputCallback enables integration of external events as X Events, socket I/O events and so on. The linked lists provided by _dlink and _dlist manage the buffers and channels in use.

The methods of interest implemented in _DacapoComChannel is:

- **call**: messages is sent when calling the method call with the request message as parameter. This method will wait for a reply, and implements the two-way method call.
- **send**: the method send also takes a request message as parameter, but will not wait for a reply. send implements the one-way method call.
- **reply**: replies to requests received earlier are sent as parameter in the reply method.
defer: in the deferred synchronous mode the reply is delayed, the defer method takes a request to act upon later as parameter.

notify: asynchronous replies is handled by calling notify, which calls a registered procedure to receive the reply.

cancel: to terminate the waiting for an asynchronous reply, cancel is called with the appropriate request message as parameter.

6 Implementation and Evaluation

A stepwise and parallel approach is chosen for the work described in this paper. The extensions for object and message level QoS are carried out independent of the port of Da CaPo to Chorus. Da CaPo is ported in a straightforward manner and tested on Chorus with a simple file transfer application and a throughput test application. Measurements of throughput for different protocol configurations using different packet sizes is summarized in Figure 9, the numbers are given in Mbps. The protocol configuration is protocol stacks with the measuring A module which sends dummy packets from a pre-allocated buffer on the sender side, on the receiver side received packets per time interval is counted, the packet buffers are released and throughput in Mbps is calculated. The T module used encapsulates TCP. The C modules is an idle-repeat-request (IRQ) module and dummy modules that just forwards the packets without altering the packets.

The results in Figure 9 shows the expected increase of throughput for bigger packets using the same protocol stack. More interesting is to investigate how the throughput is influenced by introducing more C modules between the A and T modules. When using the dummy modules this indicates how much performance is suffering from the module interfaces and packet forwarding. As we see the throughput for a given packet size is little affected when the number of dummy modules are increased from 0 to 40. A protocol stack with 40 modules are probably not realistic, but are chosen as an extreme to show how the throughput are affected. The low throughput for the IRQ C module is caused by the ineffective flow control of the idle-repeat-request protocol, and points out that careful evaluation of protocol functionality is needed.

The measurement results for Da CaPo ported to Chorus show the same behaviour characteristics as the results for the original Da CaPo [19] on SunOS. We therefore conclude that the cost of the flexibility is neglectable when measuring the throughput, what is crucial is careful design of the overall end-to-end protocol.

The integration of the single-threaded Da CaPo implementation in COOL is finished and is currently subject to testing and measurement. The next step is to use the QoS requirements given by the application to configure protocols on the fly in this prototype. The multithreaded version of Da CaPo described in Section 5.1 is designed and under development. Further work with Da CaPo includes implementing the new multithreaded Da CaPo, and integration of this new version of Da CaPo into COOL.

The extensions and modifications related to object and message level QoS, including the extension of Chic and GIOP, are implemented and tested. In order to compare the runtime efficiency of the original GIOP implementation and our extended version, we analyze the response times of remote invocations in both versions. Our first measurements are performed with the time command running a simple client application on one node, doing method invocation on another node. The results of these measurements show no differences in response time for both versions. The conclusions from these results are twofold: (1) differences between both GIOP versions are negligible, and (2) it requires a more sophisticated analysis and evaluation approach to identify these negligible differences and to evaluate our COOL extensions. The latter is subject of ongoing work.

7 Conclusions and Further Work

This paper describes design and implementation of extensions in COOL to flexibly support QoS. An extension of GIOP is implemented in the ORB runtime, together with a simple bilateral QoS negotiation scheme. The IDL compiler Chic generates stubs with a method to set QoS parameters specified by the application. These QoS parameters are delivered via the stub to the message
and transport protocol layers and propagated to the peer side. The flexible communication system Da CaPo is integrated into COOL ORB and configures protocols to fulfill the applications QoS requirements.

We have demonstrated that minor extensions in the GIOP standard (i.e., in the Request message), will enable QoS support in CORBA. Our preliminary performance evaluation indicates that QoS negotiation at the message layer does not introduce performance degradation.

The work described in this paper focuses on CORBA request/reply invocations. The next step is to use the gathered knowledge to extend COOL ORB with QoS support for multimedia streams. Support for stream interactions need an extended IDL to specify stream interfaces with QoS specification for different flows. A stream object adapter supporting the generated stream stubs and skeletons will be developed. A new IDL compiler (back end) targeted to Da CaPo will be used to produce stubs (A-modules) integrating marshaling as part of Da CaPo’s responsibility and functionality.

As a long time goal, we intend to use our results and experiences reported in this paper to develop prototypes for the next generation of middleware platforms that are based in the marriage of component technology and reflection [9].

Acknowledgements

We thank Ragnvald Blindheim for his work on design and implementation of the object and message layer extensions.

References