Faculteit der Wiskunde en Natuurwetenschappen

Wiskunde en Informatica

A QoS Provisioning Service for CORBA

afstudeerverslag

Marcel Harkema

december 1999

begeleider: Rein Smedinga

RuG

Authors: Marcel Harkema

Date: December 1999

A QoS Provisioning Service for CORBA

For internal use only at KPN RA-99-31966

Documenthistory

Version	Editor	Date	Explanation	Status
1	Marcel	20 January 2000		Final draft
2	Marcel	28 January 2000		Final

KPN Research

Informationsheet issued with Report RA-99-31966

Title: A QoS Provisioning Service for CORBA

Abstract: Quality of Service (QoS) aspects like performance, security, and reliability are important to middleware systems. Still, there is no support for QoS in current middleware systems. CORBA is an increasingly used middleware component. The QoS Provisioning Service (QPS) adds QoS support to CORBA middleware systems. QPS only uses available mechanisms to add functionality to CORBA and doesn't change the CORBA implementation itself. This allows QPS to be used with all CORBA implementations that support such extension mechanisms. This thesis describes the design of QPS and discusses a prototype implementation.

Author(s): Marcel Harkema
Reviewers: Aart van Halteren (KPN Research), Rein Smedinga (Rijksuniversiteit Groningen)
Department: Communications Architectures and open Systems (CAS)
Project: QoSMOS
Project manager: ir A.T. van Halteren
Project number: 13967
Programme: FOBO
Program manager: ir G. Kuiken
Commissioned by: ir J.M.G.A. Ouderling
Date: December 1999

Person responsible at KPN Research: ir J.M.G.A. Ouderling

Key Words: CORBA Middleware, QoS Provisioning

Mailing List:

KPN Telecom prof ir B.L. de Goede (Staf T&I), P. Morley M.Sc. (CTO)
KPN Research Alg dr KPN Research, dr R&D, hfd STR, hfd RPM, MT CAS, ir. G. Kuiken, ir. F.W. de Vries, ir. A. van Halteren, drs. J. vd Leur, prof. dr.J. Bruijning, ir. J.P. vd Bijl, ir. M.R. Vonder, drs. K.A. Helmholt, dr. R.J. Meijer, drs. M. M. Visser, dr. G. Fabian, prof. dr. W. Jonker, dr. ir. M. Vos, ir. T.G. Eijkman, ir. E.M. Peeters, drs. M.M. Visser, drs. J. Gerlofs, drs. H. Fluks, drs. Ing. R.L.J. Beekhuis, J.D. Bakker, drs. K. J. Koster, drs. J. vd Leur, ir. J.E.P. Wijnands, dr. ir. R. van Buuren

Contents

Management Summary

This is a report on a graduate assignment conducted in the context of a PhD. Research on Quality of Service (QoS) mechanisms for Open Distributed Environments (ODE). An ODE is a conglomerate of hard- and software systems often obtained from several vendors, which are interconnected through a variety of (tele-) communication networks. An ODE is clear manifestation of the integration of Telecommunication and Information Technology into an integrated set of technologies, often referred to as Information and Communication Technology (ICT). The aim of the PhD. Research is gain a better understanding of managing the QoS in large-scale ICT systems. Knowledge on architectures, protocols and QoS management mechanisms of large-scale ICT systems contributes to KPN's goal to provide ICT services.

The QoS provided by an ODE is a very important aspect, since QoS has a direct effect on the perception of the end-user of the quality of the overall services provided by an ODE. A good service provider is capable to manage and control the QoS of ODEs.

The Common Object Request Broker Architecture (CORBA) by the Object Management Group (OMG) is a crucial element of a growing number of ODEs. Unfortunately, the current version of CORBA does not support management and control of QoS. However, there is a lot of research activity aiming to extend CORBA with mechanisms and policies for QoS provisioning.

This document presents a QoS Provisioning Service for CORBA, supporting management and control of QoS. The QoS Provisioning Service doesn't change the implementation of CORBA, and is therefore usable with a wide range of CORBA implementations.

This document is of interest to those who are looking for ways to add QoS provisioning to CORBA in a portable manner, so that it can be used with ORBs from different vendors.

List of Abbreviations

API	Application Program Interface
CBQ	Class Based Queuing
CD	Common Data Representation
CIOP	Common Inter-ORB Protocol
CORBA	Common Object Request Broker Architecture
COSS	Common Object Services Specification
DCE	OSF's Distributed Computing Environment
DPE	Distributed Processing Environment
DRP	Distributed Resource Platform
DII	Dynamic Invocation Interface
DSI	Dynamic Skeleton Interface
ESIOP	Environment Specific Inter-ORB Protocol
GIOP	General Inter-ORB Protocol
IDL	Interface Description Language
IETF	Internet Engineering Task Force
IIOP	Internet Inter-ORB Protocol
IOP	Inter-ORB Protocol
IOR	Interoperable Object Reference
ISO	International Standards Organization
OCI	Open Communications Interface
OMA	Object Management Architecture
OMG	Object Management Group
ORB	Object Request Broker
OSF	Open Software Foundation
QoS	Quality of Service
QPS	QoS Provisioning Service
RFC	Request for Comments
RFP	Request for Proposals
RSVP	Resource ReSerVation Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
WFQ	Weighted Fair Queuing

1 Introduction

Most large organizations use a lot of different computer systems. They differ in their type (from small workstations to mainframes), in the vendor they come from, in the operating systems they run, in the protocols they use to inter-network, etc.

Traditionally most systems in an organization are standalone (personal workstations). These standalone systems may need to share resources (e.g. printers) and data (e.g. databases), so they're often inter-connected via a network.

Distributed Computing allows computer systems to work together in a network. It improves collaboration through internetworking and connectivity, application performance through parallel processing, reliability and availability through replication, scalability and portability through modularity, extensibility through dynamic invocation and cost effectiveness through resource sharing.

Large computer networks and corporate Intranets are heterogeneous. They are made up of mainframes, UNIX workstations and servers, PC systems running different operating systems, different communication hardware, different network protocols, etc. This heterogeneity makes developing distributed software difficult. Besides heterogeneity other issues make developing distributed software difficult, like detecting and recovering from network and host failures.

The lack of standard off-the-shelf frameworks (e.g. for printing, database interaction, and communication) for distributed systems caused developers to build their own software components for distributed systems. This in-house development process is costly and time consuming. It often results in large and hard to maintain distributed software. This problem is known as the *distributed software crisis* [1].

Distributed object computing middleware simplifies development of distributed software by providing a uniform view of heterogeneous networks and operating systems.

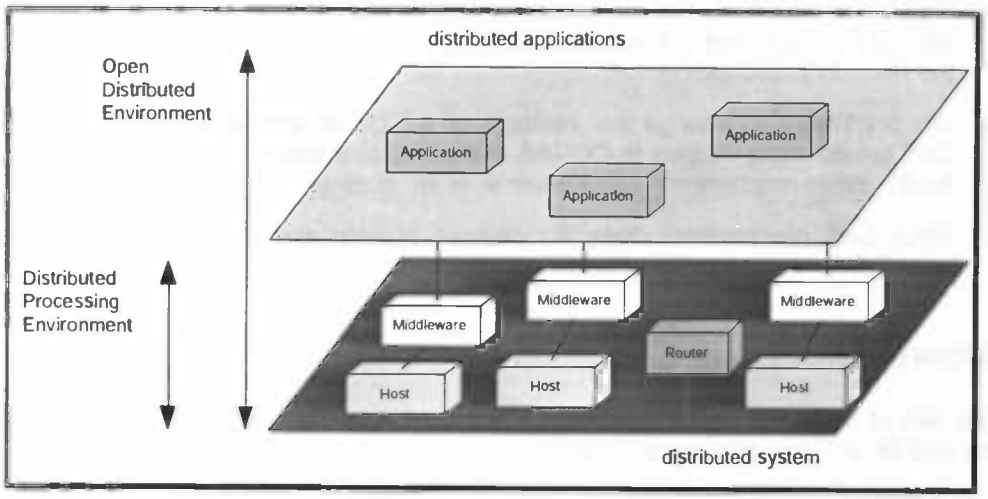

Figure 1: The Open Distributed Environment (DPE)

Middleware is a layer of software that resides between the application and the underlying heterogeneous layers of operating systems, communication protocols and hardware. It

provides *separation of concern*: the middleware isolates hardware, operating systems, and communication protocols from the rest of the system; the applications on top of the middleware. A commonly accepted middleware standard is OMG CORBA.

Figure 1 depicts the described layering. The Distributed Processing Environment (DPE) consists of computer hosts and network elements (such as routers) and the middleware layer on top of them. The collection of computer hosts and network elements alone is known as the Distributed Resource Platform (DRP). The whole environment, i.e. the DPE and the applications, is called the Open Distributed Environment (ODE).

1.1 Problem statement

In addition to functional requirements of distributed software, users also have non-functional requirements, for example performance, security, and reliability. These non-functional requirements are termed Quality of Service requirements. The management and control of QoS is called QoS provisioning.

The OMG CORBA middleware standard doesn't include specifications on how QoS can be specified, controlled and managed. There is a lot of research related to how QoS provisioning can be added to CORBA. However, most research focuses on particular kinds of QoS, like reliability or bandwidth reservation for multimedia applications. They often modify the internals of the ORB and thus create solutions that are not portable between different ORB implementations and generic enough to support a wide range of QoS.

1.2 Research questions

We have identified that CORBA does not support Quality of Service provisioning. We have decided to tackle this problem by designing a so-called QoS Provisioning Service for CORBA.

During the initial phases of the design various questions arose.

1. How are QoS requirements specified by the application?
2. How do we manage and control QoS in the ODE?
3. How does end-to-end QoS propagate through the ODE? The various parts of the ODE (applications middleware, hosts, network elements, etc.) have different views on QoS. For instance, the application would like to have a high level view on QoS, without having to worry about QoS configuration parameters of QoS mechanisms that the network provides.
4. We don't want to change the internals of a CORBA implementation. Can we add QoS provisioning support to CORBA in a portable manner, without changing CORBA itself? What mechanisms can we use in order to do so?
5. What QoS mechanisms does the network provide and how does the middleware benefit from these QoS mechanisms?

1.3 Report objective

The aim of this document is to describe an architecture and implementation that solves the lack of QoS provisioning in CORBA.

1.4 Report Structure

The structure of this document is as follows. For the reader who is not familiar with middleware or Quality of Service terminology chapter 2, architectural concepts, provides

the necessary background information. Chapter 3, engineering concepts, describes mechanisms that can be used to extend the ORB: Pluggable Protocols and Portable Interceptors. Chapter 3 also gives an overview of QML, a specification language for QoS requirements in distributed systems. Chapter 4 discusses the QoS Provisioning Service (QPS) for CORBA that we've designed. The implementation of QPS is described in chapter 5. Chapter 6 presents the conclusions and discusses future work.

2 Architectural concepts

The goal of this thesis is to design a service that adds QoS provisioning to CORBA. This chapter gives an overview of the architectural concepts that we use in this thesis. In the previous chapter the Open Distributed Environment (ODE) was introduced. The ODE is divided in a layer with applications, a middleware layer, and the Distributed Resource Platform (DRP). The DRP and the middleware layer form the Distributed Processing Environment (DPE).

The architectural concepts that we discuss are:

- Middleware, in section 2.1. We discuss CORBA, a commonly used middleware implementation.
- Quality of Service, in section 2.2. Quality of Service is a general term that covers system performance, as opposed to system operation.
- Specification of Quality of Service in CORBA, in section 2.3.
- Quality of Service mechanisms at the network-level, in section 2.4. The network is part of the DRP.

2.1 OMG CORBA middleware

The Object Management Group (OMG) was founded in 1989 by eleven companies, including 3Com, Philips and Sun. It currently has over 800 members. The OMG develops, adopts and promotes standards for distributed application development in heterogeneous environments. The OMG is structured in three bodies, the Platform Technical Committee (PTC), the Domain Technology Committee (DTC) and the Architecture Board. The architecture board manages the consistency and technical integrity of work produced by the Technical Committees.

OMG members contribute technology and ideas in response to 'Request for Proposals' (RFPs). OMG Technology Committees issue these Requests for Proposals. OMG specifications are based on responses to the RFPs. Each vendor can then build implementations based on those standards. This approach ensures interoperability, since all vendors use the same APIs and specifications.

The OMG developed the Object Management Architecture (OMA), an architectural framework for distributed interoperable objects.

This section gives an overview of the Object Management Structure (OMA) and the Common Object Request Broker Architecture (CORBA), an important part of the OMA.

2.1.1 Object Management Architecture

The Object Management Architecture (OMA) is a high-level view of a distributed environment. The OMA is divided in two parts: an Object Model and a Reference Model [2].

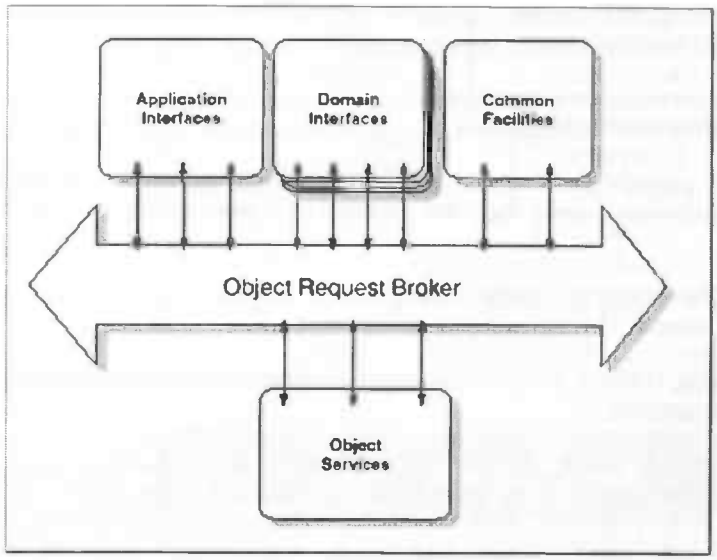

Figure 2: OMG Reference Model

The OMA Object Model defines ***common object semantics*** for specifying the ***externally visible*** characteristics of objects in a standard and ***implementation-independent*** way. The common semantics characterize objects that exist in an OMG-conformant system.

The object model has a small number of concepts:

- objects
- operations
- types
- subtyping

An object can model entities such as persons, boats or documents. Operations can be invoked on objects, an example operation is a query for a person's date of birth. Objects are instances of types, for example an instance of type ***boat*** can be a ***red boat, 4 meters long, with a seating capacity of 6***. A type characterizes the behavior of an object by describing the operations that can be applied to it. There can be relationships between types, for instance a ***speedboat*** could be a subtype of the more generic ***boat***.

A ***client*** can issue requests to objects to perform services. These services can only be accessed through well-defined ***interfaces***, the implementation and location of objects are hidden from the requesting client.

The OMA Reference Model partitions the OMG problem space into high-level architectural components that can be addressed by member-contributed technology. The Reference Model is a conceptual roadmap for assembling these technologies while allowing different design solutions. The Reference Model identifies and characterizes components, interfaces, and protocols that compose the OMA but doesn't define them in detail.

The components identified by the Reference Model are:

- Object Request Broker
- Object Services
- Common Facilities
- Domain Interfaces
- Application Interfaces

The **Object Request Broker** (ORB) component, commercially known as CORBA, is responsible for handling interaction between clients and objects.

The **Object Services** are domain-independent interfaces. Provided are interfaces that standardize life-cycle management of objects, interfaces for security, etc.

Services that provide for the discovery of available services are examples of domain independent services, since they are needed in almost every domain. Two examples of this are:

- The Naming Service, this service allows clients to find objects based on names.
- The Trading Service, this service allows clients to find objects based on their properties.

The **Common Facilities** provide a set of generic application functions that can be configured to the specific requirements of a particular configuration. These interfaces are closer to the user than the Object Services. These facilities are useful for almost every market and are divided into four basic categories: user interface, information management, systems management, and task management. Examples of Common Facilities are interfaces for printing, document management, databases, calendar management, and electronic mail. Common Facilities are commercially known as horizontal **CORBAfacilities**.

The **Domain Interfaces**, also known as vertical CORBAfacilities are divided into market segments, e.g. healthcare, financial services, and telecommunications. These industry-specific specifications are often introduced by organizations outside the OMG that want standards in these areas. Currently, the OMG has standardized facilities for: management of Audio/Video streams, distributed simulation, accounting, oil and gas industry explorations and production, etc.

Application Interfaces are developed specifically for an application. These interfaces are not standardized by the OMG, since they are application-specific. This category of interfaces can be seen as a rest category, where in-house developed interfaces can be positioned. These in-house developments can potentially lead to new RFPs and OMG adopted standards.

2.1.2 Common Object Request Broker Architecture

The Common Object Request Broker Architecture (CORBA) specifies the interfaces and characteristics of the ORB component of the OMA. Using the ORB a client can transparently issue a request to a server object. The ORB hides the location of the server object and what programming language it is implemented in. The next figure depicts this process.

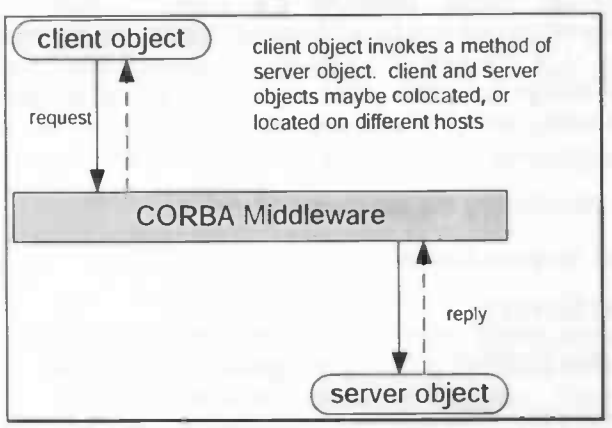

Figure 3: Method invocation with CORBA middleware

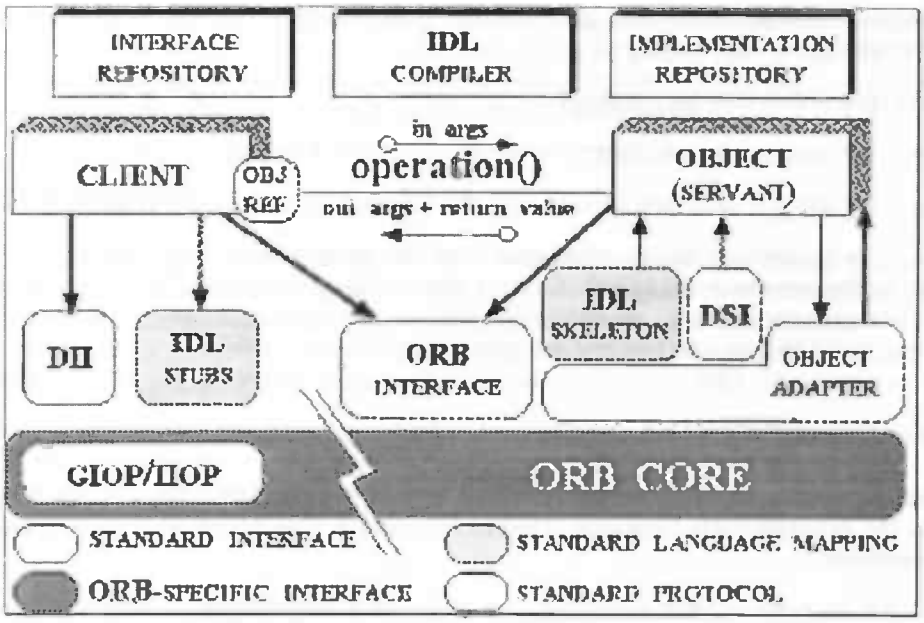

Figure 4: CORBA components

The components of CORBA [3] are:

- ORB Core
- OMG Interface Definition Language
- Interface Repository
- Language Mappings
- Stubs and Skeletons
- Dynamic Invocation and Dispatch
- Object Adapters
- Inter-ORB Protocols

2.1.2.1 The ORB Core

The ORB Core takes care of client/object interactions and provides transparency, i.e. it hides the following properties of an object from the client:

- Object location: the client does not know where a server object is located. The server object may even be located on another machine in another network.
- Object implementation: the client does not know how the server object is implemented. Programming language, operating system, and hardware issues are hidden.
- Object execution state: the client doesn't know whether a server object is activated (i.e. in an executing process). If needed the ORB can transparently start the server object before delivering the request of the client.
- Object communication mechanisms: communication mechanisms such as TCP/IP and shared memory are hidden for the client.

To make a request, the client needs a way to reference the server object. It can do so by using the ***object reference*** of the server object. An object reference is unique (it always

refers to the same object), and also immutable and opaque (i.e. clients cannot 'reach' into object references and modify it).

Object references can be obtained

- By creating a new object, i.e. sending a request to a factory [4] object.
- By using a directory service, e.g. the Naming Service or the Trading Service.
- By converting object references into strings and back, e.g. converting an object reference to a string and storing it into a database or into a file. The string can be converted back to an object reference in a later stage, and it can then be used again to make requests if the service still exists.

2.1.2.2 OMG Interface Definition Language

Object interfaces are defined in the OMG Interface Definition Language (OMG IDL). It hides the programming language used to implement object, and thus ensures language independence.

OMG IDL is a declarative language, not a programming language. IDL doesn't provide control structures and such. It does provide basic types (such as boolean and long), constructed types (structs and unions, similar to C and C++), template types (such as strings and sequences), object reference types and interface inheritance.

2.1.2.3 Language Mappings

The language constructs of OMG IDL need to be mapped on the programming language that is used to implement objects. OMG IDL is also not directly used to implement distributed applications. Some things a language mapping contains are:

- Mapping of an IDL interfaces, in the C++ language mapping this is done by using C++ classes.
- Mapping of IDL basic and structured types, e.g. a C mapping for the IDL `string` type is `char *`.
- Mapping of Object References. An operation on an object can be performed by using the -> operator in C++. This is implemented by overloading the -> operator.
- Mapping of the ORB interface and other *pseudo-objects*, i.e. objects not derived from CORBA::Object. The Object class in the CORBA module is the base interface type for all CORBA objects.
- Mapping of CORBA objects, e.g. CORBA objects can be implemented as abstract data types (a `struct` and a group of functions) in C, in C++ the IDL interfaces are mapped to classes and operations are mapped to member functions of those classes.

OMG has standardized mappings for various programming languages, including C, C++, Java, Ada and Smalltalk.

2.1.2.4 Interface Repository

Every CORBA object is introspective. An `InterfaceDef` object, describing the interface of the object, can be obtained by invoking the `get_interface` method. The Interface Repository (IR) stores the data needed to support introspection. The IR provides support for dynamically accessing and adding OMG IDL interface definitions, it's used when issuing client-server requests via Dynamic Invocation (described later in this document). An interface definition contains a description of the operations it supports, including

parameter types and exceptions it may raise. The interface repository also stores constant values and typecodes. Typecodes are values that describe a type in structural terms. Predefined typecodes include TC_char, TC_boolean, TC_string, ... Typecodes are used to indicate the types of actual parameters of invocations.

2.1.2.5 Stubs and Skeletons

A ***stub*** is a mechanism that creates and issues server requests on behalf of the client, stubs are included in the client implementation. A ***skeleton*** is a mechanism that delivers requests to the CORBA object implementation, skeletons are included in the server implementation. Dispatching requests by using stubs and skeletons is called ***static invocation***. Stubs and skeletons have ***a priori*** knowledge of the OMG IDL interfaces used.

The stub ***marshals*** the request from native programming language format to a format suitable for transmission to the server object. The skeleton does the reverse, it ***de-marshals*** the request to the form required by the programming language in which the server object is implemented.

2.1.2.6 Dynamic Invocation and Dispatch

CORBA provides the Dynamic Invocation Interface (DII) and the Dynamic Skeleton Interface (DSI). They can be viewed as a generic stub and skeleton respectively.

2.1.2.6.1 Dynamic Invocation Interface

DII allows a client to make requests without having compile-time knowledge of the server object. Three types of invocations are supported:

- Synchronous invocation: the client invokes the requests and blocks until it receives a response.
- Deferred synchronous invocation: the client invokes the requests and continues processing. The response is collected in a later stage.
- One-way invocation: the client invokes a request and continues processing. There is no response.

DII is more costly than using static invocation, for example creating a DII request may cause the ORB to transparently access the Interface Repository to obtain type information. The Interface Repository is an object, so a single DII request may require several remote invocations.

2.1.2.6.2 Dynamic Skeleton Interface

DSI provides dynamic dispatch to objects. It allows servers to be written without having static skeletons compiled in. Like DII it provides flexibility in exchange for performance.

2.1.2.7 Object Adapters

The Object Adapter can be seen as glue between object implementations and the ORB. Object Adapters handle object registration (allow programming languages to register implementations for CORBA objects), object reference generation and interpretation, object implementation activation and deactivation, mapping object references to an implementation, security of interactions, and method invocation. CORBA allows for more than one object adapter, so that different implementation styles for objects may be supported, e.g. different Object Adapters for different programming languages.

2.1.2.8 Inter-ORB Protocols

CORBA ORBs based on pre 2.0 specifications lacked interoperability. Each ORB had their own data formats and protocols for remote ORB communications, so ORBs from different vendors couldn't work together. CORBA 2.0 introduced an ORB interoperability architecture, which provides for interoperability between different ORB implementations. This interoperability architecture consists of:

- Object Request Semantics: CORBA IDL
- Transfer and Message Syntax: the General Inter-ORB Protocol (GIOP) and (optionally) Environment Specific Inter-ORB Protocols (ESIOPs)
- Transports: the Internet Inter-ORB Protocol (IIOP) and (optionally) transports for ESIOPs

The General Inter-ORB Protocol (GIOP) specifies message formats and common data representations for inter-orb communication. The Internet Inter-ORB Protocol (IIOP) specifies how GIOP messages are exchanged over a TCP/IP network. Environment Specific Inter-ORB Protocols (ESIOPs) allow interoperability over specific networks.

2.2 Quality of Service

Applications, such as multimedia (e.g. teleconferencing), avionics mission computing (e.g. operational flight programs for fighter aircraft), telecom call processing and E-Commerce are an increasingly important class of distributed applications and require Quality of Service (QoS) guarantees for latency, bandwidth, and reliability [5].

The CORBA specification [3] doesn't define policies and mechanisms for providing end-to-end QoS. Conventional CORBA ORB implementations provide best-effort QoS, i.e. no QoS provisioning. The Internet today is an example of a best-effort Quality of Service. TCP/IP will make an earnest attempt to deliver packets, but may drop packets indiscriminately when resources are exhausted (e.g. congestion) or underlying networks fail [6].

2.2.1 What is QoS?

Quality of Service is a general term that covers system performance, as opposed to system operation [7]. QoS features can be considered non-functional features of a system. The users of a system have both requirements for the functions that are to be performed and for the QoS with which they are performed.

QoS is a pervasive requirement. Meeting QoS involves actions in end-systems, networks, and any other components that end-to-end interactions may pass through.

We can distinguish user perceived QoS and machine measured QoS. For example, we can measure higher availability in a server that goes down once a year than in a server that goes down once a month. An end-user may perceive better quality in the latter service if that service is faster.

2.2.2 Static and dynamic QoS

A common approach for dealing with QoS is determining QoS requirements during the design process and making design choices (such as scheduling rules and assigning priorities to tasks) and physical configuration choices (such as memory and bandwidth) to make sure QoS is met. This is the static approach.

Another extreme are systems which do all QoS handling, within range, during run-time, i.e. dynamically negotiate and adapt to degradations (e.g. switch to lower video resolutions in video streams). This is the dynamic approach.

Most systems' QoS handling lies between the two extremes.

2.2.3 QoS Framework

In this section an overview of the ISO/IEC QoS Framework (ISO/IEC DIS 13236) [8] is given. This framework provides a collection of concepts and terminology that will enable those designing and specifying distributed systems, and those defining communication services and protocols to interact effectively. It describes how QoS can be characterized, how QoS requirements can be specified, and how QoS can be managed. The next figure (source: [9]) provides an overview of QoS management in the OSI QoS model.

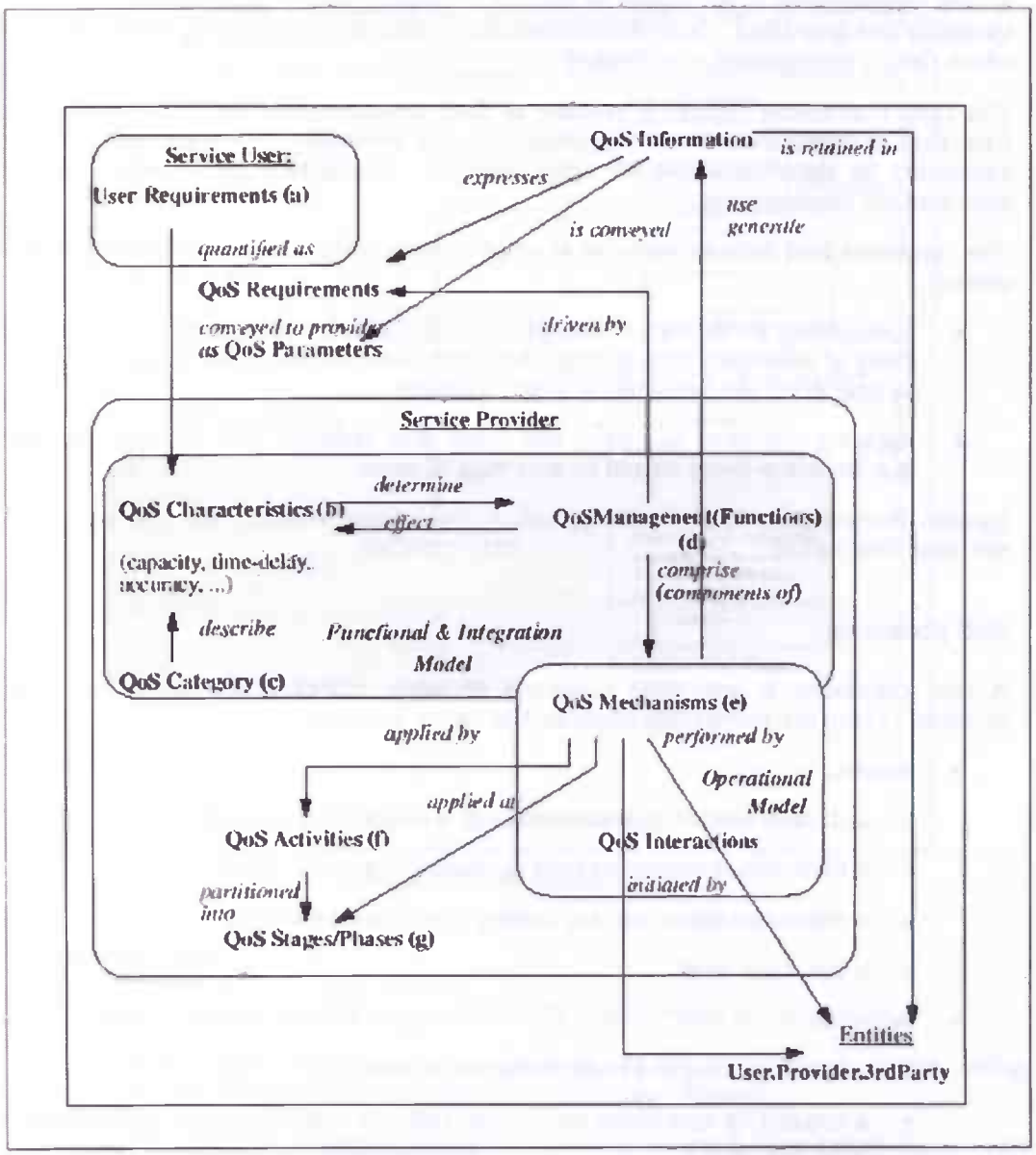

Figure 5: Management of QoS

Quantities such as throughput and reliability are termed QoS characteristics. For a certain application a set of QoS characteristics may be important, for those characteristics various QoS requirements will be defined. Information about these requirements need to be exchanged between parts of the system, this information is termed QoS parameters. QoS management is the term for any set of activities performed by a system to support QoS monitoring, control and administration.

There are two basic roles for entities in a distributed system: Service Provider and Service User. The Service Provider has a number of QoS characteristics, such as response time or availability of a service. The Service User has requirements, which may be related to the QoS expected from the Service Provider. Not all user requirements have to be expressed in terms of the QoS characteristics provided by the Service Provider. User requirements are often subjective, whereas the Service Provider needs objective requirements in order to handle them.

2.2.3.1 QoS characteristics

A QoS characteristic is an aspect of QoS of a system, service, or resource that can be identified and quantified. QoS characteristics are defined independently of the means by which QoS is represented or controlled.

The QoS Framework defines a number of QoS characteristics and categorizes them. Examples of characteristics are response time and availability. Example categories are categories for databases and for video streams. These two applications have very different QoS requirements.

The framework also defines two ways in which a characteristic may be defined in terms of others:

- Specializing an abstract definition. We can specialize the characteristic of transit delay of something by specifying the points between which the delay is defined or by specifying the data units to which it applies.
- Applying statistical functions, like mean and variance. For example specifying that the mean delay should be less than 15 msec.

System designers are free to add and define other characteristics if the existing ones do not meet their needs.

2.2.3.2 QoS parameters

A QoS parameter is any value conveyed between entities of the same or different systems. There are many kinds of parameter types, including:

- Values:
  - a desired level of characteristic, e.g. a target of some kind
  - a minimum or maximum level of characteristic, e.g. a limit
  - a measured value, used to convey historical information
  - a threshold level
- Activities:
  - a warning or signal to take corrective action
  - a request for operations on managed objects relating to QoS, or the results of such operations

2.2.3.3 QoS management functions and mechanisms

A QoS management function is a function designed with the objective to meet one or more QoS requirements. These QoS management functions (QMFs) generally have a number of components, called QoS mechanisms.

A QoS mechanism, performed by one or more entities, may use QoS information (e.g. parameters, context, and requirements), in order to support things like:

- establishment of the conditions to meet the QoS requirement for a set of QoS characteristics
- monitoring of the observed values of QoS
- maintenance of the actual QoS as close as possible to the target QoS
- control of QoS targets
- enquiry upon some QoS information or action
- alerts as result of some event relating to QoS management

The next figure (source: [9]) outlines the QMF relationships in the OSI QoS model.

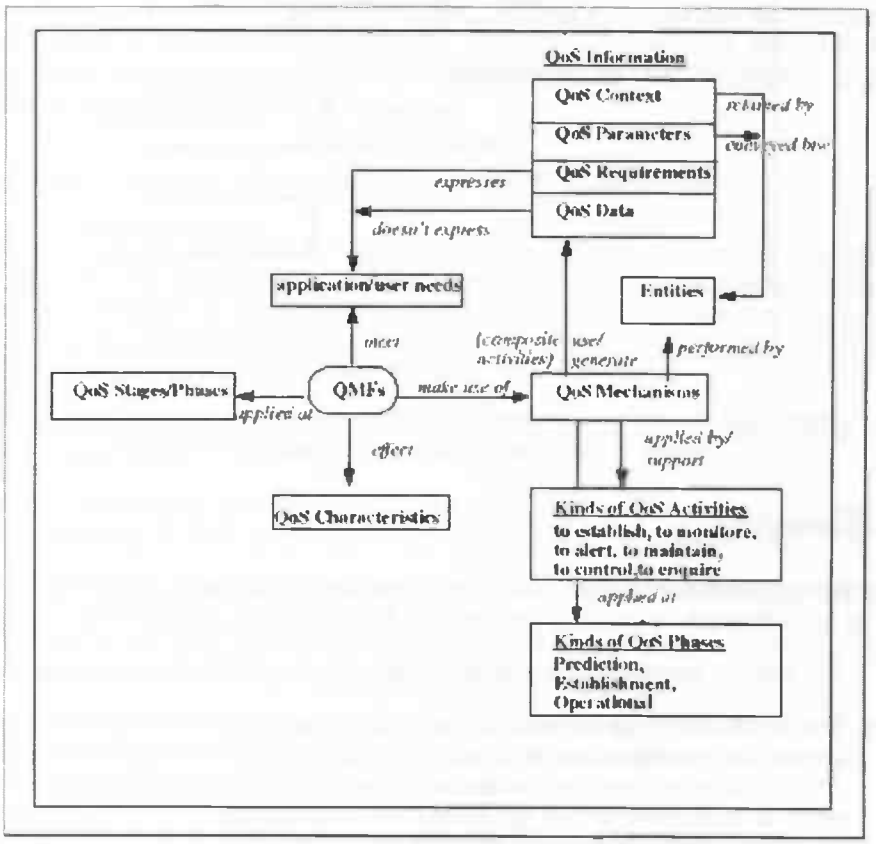

Figure 6: QMF relationships

2.2.3.4 QoS management

The framework defines several stages when QoS management is used:

- a priori - QoS requirements may be built into the systems, by design, sizing, procurement of suitable services, or by dedicating resources
- before initiation (prediction phase) - QoS requirements can be conveyed to some or all of the participating entities before an activity is initiated, for example, reserving resources before a connection that would use them is established
- at initiation of an activity (establishment phase) - negotiating QoS requirements at (connection) establishment time
- during the activity (operational phase) - handling changing QoS requirements, due to changing user requirements, detected performance loss, explicit indications from the service provider, or explicit indications from one or more third parties

- after the activity (historical phase) - after the activity took place, for example, performance monitoring

The next figure (source: [9]) displays the stages when QoS management is used.

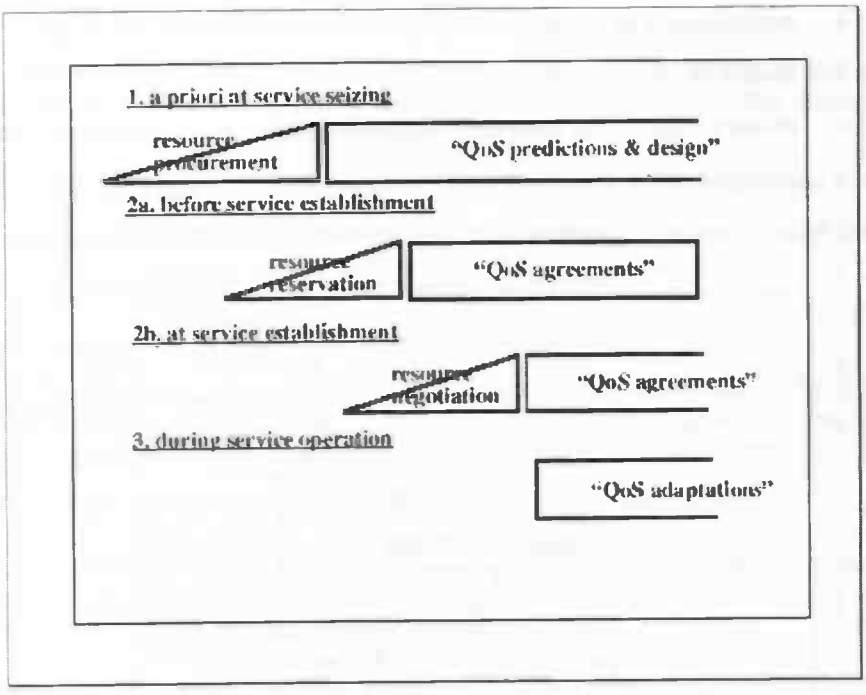

Figure 7: QoS phases, stages of service of evolution

2.2.3.5 QoS agreements

During the establishment phase the entities involved try to reach agreement on the QoS that is to be delivered. It involves determining for each QoS characteristic concerned:

- The type of agreement, e.g. one or more targets, limits, thresholds, etc.
- The level of the agreement, meaning the strength of the agreement, whether it should be monitored, and if so, what action (e.g. re-negotiation, aborting the activity, aborting another competing activity of lower precedence, ...) to perform when a requirement can no longer be met (temporarily or permanently). For examples, best-effort QoS (as soon as possible), compulsory QoS (the QoS is monitored, an action is taken when it degrades below a certain level, resources are not reserved but shared), guaranteed QoS (except for rare events such as equipment failure, resources are reserved).

2.3 Quality of Service specification

In conventional CORBA ORBs there are no APIs for specifying end-to-end QoS requirements. This chapter describes developments in QoS design and specification for distributed systems, QoS in OMG CORBA, and QoS on the network.

2.3.1 QoS design and specification

Deciding which QoS properties should be provided is an important part of the design process. This decision should be made as early as possible, since late consideration of QoS aspects often lead to increased development and maintenance costs and systems that do not meet user requirements.

To capture component-level QoS properties, [10] introduces a language called QML, QoS Modeling Language. They also extend the Uniform Modeling Language (UML), the ***de facto*** standard specification language [11] [12], to support the definition of QoS properties. QML can be used to specify QoS requirements in the design of class models and interfaces of distributed object systems.

The Quality of Service for CORBA Objects (QuO) framework [13] is another research project that aims to provide QoS abstractions to CORBA objects. It extends OMG IDL with a QoS Description Language (QDL). QDL specifies an application's expected usage patterns and QoS requirements for a connection to an object. A connection is an QoS-aware communication channel. The ***expected*** and ***measured*** QoS behaviors of a connection are characterized through a number of QoS regions. During connection establishment the client and server negotiate about a certain region. This region captures the expected QoS behavior of the connection. After the connection is established the actual QoS level is continuously monitored. The client is notified if the ***expected*** QoS level is no longer within the negotiated region. The client and server can then negotiate a new region or perform some other action.

2.3.2 Developments in CORBA QoS by the OMG

The OMG recognizes the need for QoS-enhanced systems and wrote the OMG QoS Green Paper [7]. The objective of the OMG Green Paper is to define and specify the set of necessary extensions to the OMA, the CORBA standards and the QoS standards that will support QoS management in CORBA-based systems. The paper hasn't been finished, and the QoS workgroup of the OMG has been dissolved.

Other OMG initiatives taking QoS into consideration are CORBA Messaging [14] and Real-time CORBA [15].

The CORBA Messaging specification defines a standard QoS framework that allows setting and querying of QoS at multiple client-side levels. These levels are:

- ORB level: QoS policies at this level control all requests made using this ORB
- Thread level: QoS policies at this level control all requests issued from a given thread. These policies override the ORB level policies.
- Object reference level: QoS policies at this level control requests made using that object reference. Policy settings at this level override both thread level policies and ORB level policies.

The specific QoS features addressed in the CORBA Messaging specification include delivery quality, queue management and message priority. All qualities are defined as interfaces derived from `CORBA::Policy` (Notation: ***Policy*** is an IDL interface defined in the ***CORBA*** module), policies are part of CORBA specification. Interfaces derived from `CORBA::Policy` can be used to give access to methods that affect the operation of a CORBA service. CORBA Messaging introduces new policy interfaces for QoS. These policies can be used to request specific QoS, for example by setting QoS parameters for certain QoS characteristics.

With CORBA Messaging applications can specify their QoS requirements to the ORB in a portable, protocol independent, and convenient way.

CORBA Messaging also features new invocation models, Asynchronous Method Invocation (AMI) and Time-Independent Invocation (TII). These new invocation models fix the shortcomings of the standard CORBA invocation models (e.g. lack of a truly asynchronous method invocation model).

Real-time CORBA is an optional set of extensions tailored to equip ORBs to be used as a component of a Real-time system [15]. It is based on the CORBA 2.2 and CORBA Messaging (it uses the policy framework of the CORBA Messaging) specifications. One

of the goals of the specification is to provide a standard for CORBA implementations that supports end-to-end predictability.

2.4 Quality of Service at the network-level

QoS is a pervasive requirement [7]. Meeting QoS involves actions in end-systems, networks, and any other components that end-to-end interactions may pass through. This section contains a brief overview of QoS in networks.

There are three basic service levels of end-to-end QoS across heterogeneous networks:

- Best-Effort Service (also known as ***lack of QoS***)
- Differentiated Service (also known as ***soft QoS***)
- Guaranteed Service (also known as ***hard QoS***)

Best-Effort QoS is basic connectivity without performance guarantees. The Internet of today is an example of a best-effort service; TCP/IP doesn't guarantee performance. Best-effort is suitable for many applications, such as e-mail or file-transfers.

Differentiated QoS means that some traffic is treated better than the rest. It is a statistical preference, not a hard and fast guarantee. Examples of better treatment are faster handling for some traffic, more bandwidth on average, or lower loss rate on average. The Differentiated Services (Diff-Serv) working group [16] in the IETF is developing standards and definitions for differentiated services [17] [18] [19] [20].

Guaranteed QoS is an absolute reservation of network resources, typically bandwidth, for specific traffic. There are various technologies that provide guaranteed service, one of them is RSVP-signaling [21] [22] in combination with IP-WFQ, both are described later in this chapter. The Integrated Services (Int-Serv) working group [23] in the IETF is developing standards and definitions for guaranteed services.

2.4.1 Signaling QoS with Int-Serv RSVP

QoS signaling is a form of network communication, it provides a way to signal requests to a neighboring network node (host, router, ..). An IP network can use part of the IP packet header to request special handling of priority traffic, for example. End-to-end QoS requires that every single element in the network path (switch, router, firewall, host, client, etc.) delivers its part of the QoS. Signaling is used to coordinate this. For differentiated service IP Precedence [24] is used; 3 precedence bits in the IPv4 headers ToS (Type of Service) field are reserved for specifying the importance of IP packets. For guaranteed service RSVP (Resource ReSerVation Protocol, RFC-2205) [22] can be used.

RSVP can be used to dynamically reserve network bandwidth. It enables applications to request a specific bandwidth for a data flow. Hosts and routers use RSVP to deliver QoS requests to the routers on the paths of the data stream and to maintain router and host state to provide the requested service.

2.4.2 Network packet queuing with WFQ

To handle possible overflow of network packets, a network element can sort the traffic. IP routers use queuing mechanisms such as CBQ and WFQ to classify packets according to the specified QoS.

The Weighted Fair Queuing algorithm (WFQ) [25] is a queuing technique that:

1. Schedules certain traffic to the front of the queue, e.g. interactive traffic to reduce response time.
2. Fairly shares the remaining bandwidth.

The "weight" is determined by:

1. Requested QoS. For example, RSVP can be implemented using WFQ to allocate buffer space and to schedule packets.
2. Flow throughput (weighted-fair), by default WFQ favors lower-volume traffic flows over higher-volume ones, i.e. it favors an e-mail message over a large FTP download.

There are other packet scheduling algorithms that can be used in combination with RSVP, for example Class Based Queuing (CBQ) [26].

3 Engineering concepts

In the previous chapter we discussed some architectural concepts. This chapter is concerned with engineering concepts that are used in this thesis. One of the constraints listed in the research questions in chapter 1 is that we aren't allowed to change the inner working of CORBA for our QoS Provisioning Service. We do however need to add functionality in the ORB. We need to filter CORBA method invocations to see if there are QoS requirements for them. If there are QoS requirements for a filtered method invocation we need to use a QoS mechanism at the network-level to implement these requirements. In short we need to add the following functionality:

- Support to filter method invocations.
- Support to use QoS mechanisms available at the network-level.

Method invocations can be filtered using a so-called interceptor. This chapter discusses Portable Interceptors. If we could make an alternative for IIOP to transport GIOP messages we could use that to implement QoS at the network-level. Pluggable Protocols support addition of protocol plug-ins to the ORB in a portable manner.

Another research question is how Quality of Service requirements are specified by the application. Section 3.3 introduces QML, also briefly mentioned in the previous chapter.

3.1 Portable Interceptors

Interceptors provide a manner to extend the functional path of a method invocation with additional functionality (located outside the ORB). The CORBA 2.2 specification defines two kinds of interceptors as an optional extension to the ORB. The two interceptor kinds are request-level interceptors and message-level interceptors.

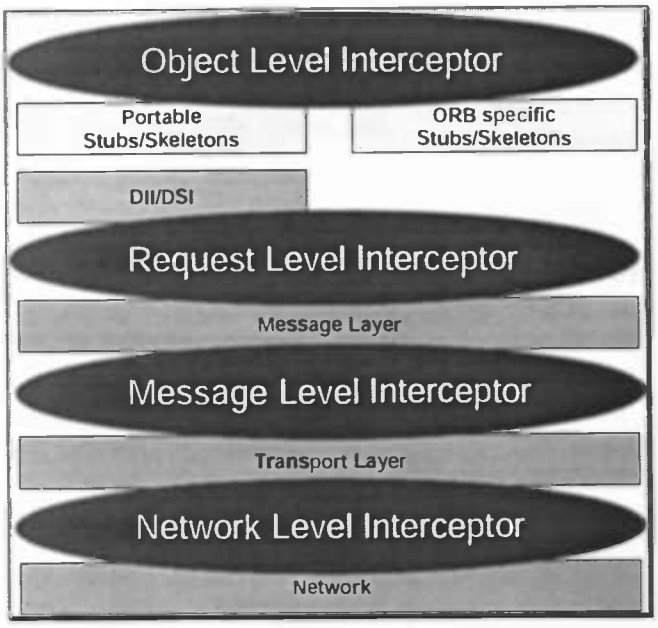

Figure 8: Interception points

Request-level interceptors perform transformations on a structured request. Request-level interceptors are executed whether the client and target are collocated and remote. Typical uses for request-level interceptors include transaction management, access control, and replication. The request-level interceptor is able to find the target object, operation name, context, parameters, and (when complete) also the result of the invocation. The request-level interceptor may decide not to forward the request for further processing, this can be used to implement access control. Request-level interceptors allow interception at four points related to request processing:

- after a client ORB receives a request from a client and before transmitting it to a target
- after a target ORB receives a request and before dispatching it to a method
- after a target ORB regains control after a method invocation and before transmitting the reply to the client
- after a client ORB receives a reply and before returning it to the client

Message-level interceptors are only executed when a remote invocation is required. Message-level interceptors perform transformations on an unstructured buffer, this buffer contains the request transformed to a message suitable for network transfer. A typical use for message-level is encryption.

Interceptors in CORBA 2.2 are under-specified. This lack in specification makes it hard to write portable interceptors, since different vendors may provide different interceptor support. The OMG issued a Request For Proposal (RFP) for Portable Interceptors [27] aiming for a more portability for interceptors between ORBs and also more functionality and interception points. Two new interceptor types are proposed, object-level (or IOR management) interceptors and network-level interceptors. Object-level interceptors can be used to monitor life-cycle events, like object reference creation and first-time usage of an object reference. Network-level interceptors can be used for connection management, examples include interception of connection establishment and closure.

3.2 Pluggable Protocols

This section describes ORB interoperability with the GIOP and IIOP protocols and introduces the concept of pluggable protocols.

GIOP/IIOP is sufficient for applications that require best-effort QoS, but not sufficient if the application has more stringent QoS requirements (which require other protocols to be used). Pluggable protocol frameworks such as ORBacus' OCI and TAO's Pluggable Transports [28] [29] make it easy to plug-in new protocols.

3.2.1 Interoperability between ORBs

Interoperability allows different ORB implementations to communicate. CORBA 1.0 lacked specification for inter-ORB communication. Due to the lack of such interoperability specification, ORB implementations from different vendors couldn't work together.

The CORBA 2.x specification [3] supports protocol-level ORB interoperability via the General Inter-ORB Protocol (GIOP). It also specifies the Internet Inter-ORB Protocol (IIOP), which is a mapping (specialization) of GIOP to TCP/IP [30].

Here we give an overview of the CORBA Protocol Interoperability Architecture.

3.2.1.1 Interoperable Object References

An important interoperability issue in CORBA is how to locate (address) objects. The Interoperable Object Reference (IOR) stores information needed to locate and communicate with an object over one or more protocols. An object reference identifies a

single object and associates one or more paths through which that object can be accessed. The IOR is a sequence of profiles, each profile describes a protocol specific way to access the object. For example, the IIOP profile includes the IP address and a TCP port number. The Object Adapter component of CORBA generates and interprets object references.

3.2.1.2 General Inter-ORB Protocol (GIOP)

The GIOP specification consists of the following:

- A Common Data Representation (CDR). CDR is a transfer syntax that maps IDL data types from their native host format into portable representation.
- Message formats. GIOP messages are exchanged between ORBs to facilitate object requests, locate object implementations, and manage communication channels.
- Transport assumptions. Several assumptions are described concerning any network transport layer that may be used to transfer GIOP messages. Connection management and constraints on message ordering are also specified.

3.2.1.2.1 Common Data Representation

Common Data Representation (CDR) is a transfer syntax that maps IDL data types from their native host format into a portable representation. It has the following features:

- Variable byte ordering, two frequently used byte ordering formats, little-endian and big-endian, are supported.
- Aligned primitive types, which allow for efficient handling of data by architectures that enforce data alignment in memory.
- A complete IDL mapping of all IDL basic data types. IDL data types are marshaled into an encapsulation. The encapsulation is a stream of octets. In CORBA an octet is defined as a sequence of 8-bits. An octet itself doesn't undergo conversion when transmitted over the network by CORBA.

GIOP defines two kinds of octet-streams:

- Messages described in Message formats (3.2.1.2.2).
- Encapsulations described in the above text.

3.2.1.2.2 Message formats

The GIOP 1.0 specification defines 7 message types. They're summarized in the table below.

Message Type	*Originator*	*Value*
Request	Client	0
Reply	Server	1
CancelRequest	Client	2
LocateRequest	Client	3
LocateReply	Server	4
CloseConnection	Server	5

MessageError	Both	6

It lists the message type, the originator of the message and the value used to identify the message type in a message header. There are messages for sending requests and receiving replies (**Request**, **Reply**), locating objects (**LocateRequest**, **LocateReply**), connection management (**CloseConnection**), letting the server know that the client no longer expects a reply to a pending **Request** or **LocateRequest** message and error handling (**MessageError**). A **MessageError** is sent in response to a request when the version number or request value in the message header are unknown and to messages that are not properly formed (e.g. messages that do not have the proper magic value in their header).

3.2.1.2.3 Transport assumptions

Because GIOP is designed to be implementable on a wide range of different transports, several assumptions regarding the behavior of these transports have been made. The GIOP expects that:

- The transport is connection-oriented.
- The transport is reliable. Bytes are to be delivered in the order they're sent, at most once, and positive acknowledgement of delivery should be available.
- The transport is a byte-stream. No size limitations, fragmentation, or alignments are enforced.
- The transport provides reasonable notification of disorderly connection loss. If the peer process aborts, or the peer host becomes unreachable (due to a crash or network connectivity loss), a connection owner should receive some notification of this condition.

The server doesn't actively initiate connections. It should be prepared to accept client connect requests. A client can send a ***connect*** request to a server. The server may ***accept*** the request and set up a new, unique connection with the client, or it may ***reject*** the request, e.g. due to lack of resources. Each side, server or client, may ***close*** an open connection.

A transport (e.g. UDP [31]) might not directly support the model, in this case an extra layer has to be made on top of that transport which implements the missing features [32].

3.2.1.3 Internet Inter-ORB Protocol (IIOP)

The mapping of GIOP to TCP/IP [30] is called Internet Inter-ORB Protocol (IIOP). ORBs must support IIOP in order to be CORBA 2.0 ***interoperability compliant.*** IIOP guarantees out-of-the box interoperability between CORBA ORBs.

3.2.1.4 Environment Specific Inter-ORB Protocols (ESIOPs)

CORBA 2.x allows vendors to define Environment Specific Inter-ORB Protocols (ESIOPs) in addition to GIOP/IIOP. ESIOPs are optimized for particular environments, e.g. telecommunications or avionics. ESIOPs can define unique data representation formats, ORB messaging and transport protocols, and object addressing formats. CORBA 2.x defines one EISOP protocol: DCE-CIOP [3].

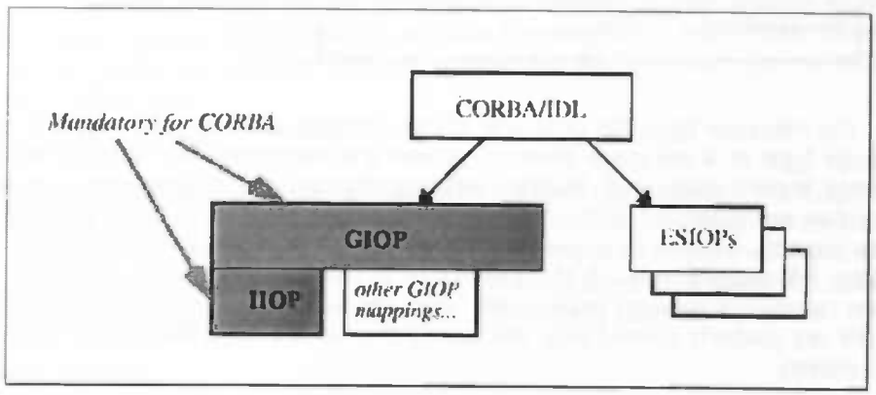

Figure 9: Inter-ORB protocol relationships

3.2.2 Protocol limitations of Conventional ORBs

GIOP/IIOP works well for conventional request/response applications that only require best-effort QoS. High-performance, real-time, and/or embedded applications require optimized protocol implementations, because they can't tolerate the overhead, jitter, latency, and/or message footprint size of GIOP/IIOP [33].

Domains that require optimized transport and/or messaging protocols include teleconferencing and real-time applications.

Conventional CORBA implementations have the following limitations for performance sensitive applications:

- Static protocol configurations: most ORBs only support a limited number of statically configured transport or messaging protocols, only GIOP/IIOP in most cases.
- Lack of protocol control interfaces: most ORBs don't allow applications to define protocol policies and attributes.
- Single protocol support: most ORBs don't support simultaneous use of multiple inter-ORB protocols.
- Lack of real-time protocol support: most ORBs have no support for specifying and enforcing real-time protocol requirements.

With CORBA 2.2 it is difficult to add new protocols. The ORB internal structures are not always accessible to protocol implementers. Even when the internal structures are visible, via interfaces, those interfaces are proprietary. A newly developed protocol can then only be used for a particular ORB.

A solution is to define common interfaces. The Open Communications Interface [34] [35] defines those interfaces.

3.2.3 Open Communications Interface

The Open Communications Interface (OCI) defines common interfaces for pluggable protocols.

It consists of two parts:

- The Message Transfer Interface (MTI)
- The Remote Operation Interface (ROI)

The Message Transfer Interface can be used for connection-oriented, reliable protocols, such as a TCP/IP plug-in. If the ORB uses the GIOP messaging protocol then such a

plug-in implements IIOP. If a plug-in for a non-reliable or non-connection-oriented protocol is to be built, the plug-in should take care of it.

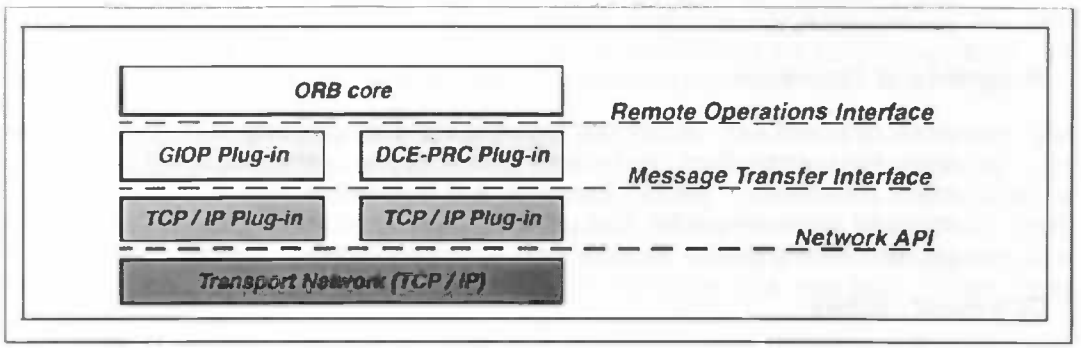

Figure 10: OCI interfaces

The Remote Operation Interface can be used to support protocols that directly implement remote procedure calls. GIOP falls in this category, since it basically implements remote procedure calls. Other protocols can be DCE-RPC and TCAP (Transaction Capabilities Application Part, part of SS7) [35].

Of course two plug-ins (one for the MTI and one for the ROI) can be combined. For example, a GIOP (ROI) plug-in together with a TCP/IP (MTI) plug-in implements IIOP.

3.2.3.1 Message Transfer Interface Summary

The Message Transfer Interface is based on the Acceptor and Connector design patterns [36]. This section describes the OCI classes, the next figure shows the class diagram. The Buffer and Info classes are not shown in the figure.

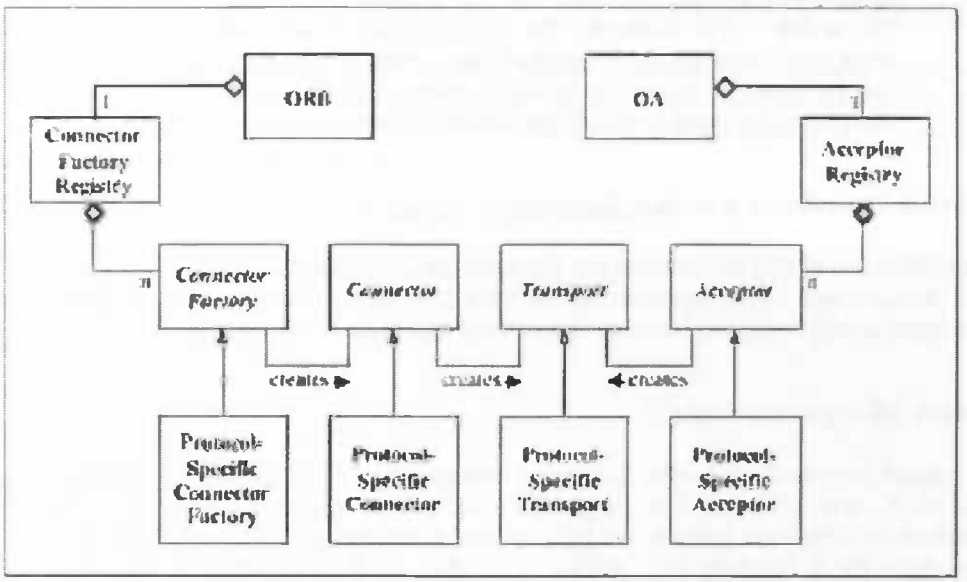

Figure 11: OCI class diagram

Buffer

A buffer is an object holding an array of octets and a position counter. The position counter determines how many octets have been received or sent.

Transport

The Transport interface allows sending and reception of octet streams in the form of Buffer objects. Both blocking and non-blocking send and receive operations are available, as well as operations that handle time-outs and detection of connection loss.

Acceptor and Connector

Acceptors and Connectors are factories for Transport objects. A Connector is used to connect clients to servers. An Acceptor is used by a server to accept client connection requests. Acceptors and Connectors also provide operations to manage protocol-specific IOR profiles. This includes operations for comparing profiles, adding profiles to IORs, or extracting object keys from profiles.

Connector Factory

A Connector factory is used by clients to create Connectors. No special Acceptor factory is necessary, since an Acceptor is created just once on server start-up and then accepts incoming connection requests until it is destroyed on server shutdown. Connectors, however, need to be created by clients whenever a new connection to a server has to be established.

Acceptor and Connector Registries

The ORB provides a Connector Factory Registry and the Object Adapter provides an Acceptor Registry. These registries allow the plugging-in of new protocols. Transport, Connector, Connector Factory and Acceptor must be written by the plug-in implementers. The Connector Factory must then be registered with the ORB's Connector Factory Registry and the Acceptor must be registered with the Object Adapter's Acceptor Registry.

Info

Info objects provide information on Transports, Acceptors and Connectors. A TransportInfo provides information on Transport, an AcceptorInfo provides information on an Acceptor and a ConnectorInfo provides information on a Connector. For example, the AcceptorInfo object can be used to find out on what hostname and port number the server is listening, the TransportInfo object can be used to query the network address of the client, and the ConnectorInfo object can be used to query the network address of the server.

3.2.3.2 Remote Operations Interface Summary

The definition of this interface is not yet finished according to OCI specification in [35] and this feature will be dropped in future OCI specifications (see next section). For my graduate assignment the Remote Operations Interface is not needed.

3.2.3.3 Future OCI specifications?

Humboldt University in Berlin, Deutsche Telekom and OOC (makers of ORBacus) plan to initiate a new Request For Proposals (RFP) for pluggable protocols. The Remote Operations Interface feature will be dropped from the specification, since there was no real need for it; the new RFP will focus on plug-ins for the message-transfer level. The term 'Pluggable Protocols' will most likely be changed to 'Pluggable Transports'.

3.3 QML

The Quality of Service Modeling Language (QML) is a general-purpose Quality of Service specification language, it is not tied to any particular domain, such as real-time or multi-media systems, or to any particular QoS category, such as reliability or performance [37] [38] [10]. QML also extends the Uniform Modeling Language (UML), the ***de facto*** standard specification language [11] [12], to support the definition of QoS properties.

QML can be used for QoS specification in the design of class models and interfaces of distributed object systems.

For the QoS Provisioning Service we've adopted QML's terminology and we also use QML to specify the QoS, so we need to provide an overview of the terminology here. It's not a complete overview, because that would be outside the scope of this document.

QML uses three abstraction mechanisms to specify QoS: contract types, contracts, and profiles. **Contract types** represent specific QoS categories, like reliability and performance. In a contract type the **dimensions** are defined that characterize a particular QoS category. Dimensions have domain values that may be ordered. QML has 3 kinds of domains: set, enumerated, and numeric. A **contract** is an instance of a contract type. **Profiles** associate contracts with interfaces (IDL interfaces in our case) and their operations.

These definitions are best clarified by an example. In the following figure two contract types, ***Performance*** and ***Reliability,*** are specified. Performance has 2 dimensions: delay and throughput. Reliability has 3 dimensions: numberOfFailures, TTR, and availability.

```
Type Performance = contract
  {
    Delay: decreasing numeric msec;
    Throughput: increasing numeric mb/sec
  };

Type Reliability = contract
  {
    numberOfFailures: decreasing numeric no/year;
    TTR: decreasing numeric sec;
    availability: increasing numeric
  };
```

Figure 12: QML contract type examples

All dimensions in this example have numeric domain values. If a numeric domain is defined as 'increasing' then the QoS is better with larger values. An example of this is the availability dimension; larger values are better. Numeric domains defined as 'decreasing' have better QoS with smaller values.

Next we define a contract ***SystemReliability*** of type Reliability.

```
SystemReliability = Reliability contract
  {
    numberOfFailures < 10;
    TTR
      {
        percentile 100 < 2000;
        mean < 500;
        variance < 0.3
      };
    availability > 0.8
  };
```

Figure 13: QML contract example

All dimensions now have various constraints attached to them. The number of failures should not be 10 times a year or more. The time to repair (TTR) should be less than 2000 seconds for each service failure, the mean TTR should be less than 500 seconds, and the variance of the TTR should be less than 0.3. Finally, eighty percent availability is

required. QML discerns simple and aspect constraints. Simple constraints use comparisons using the '<', '>', '<=', '>=', '==' operators. Aspect constraints allow more complex constraints. Aspects are defined in QML as statistical characterizations. QML includes four aspects: percentile, mean, variance, and frequency.

In the following figure the profile *rateServerProfile* is specified. This profile is attached to the *RaterServerI* interface. For all operations of RateServerI the SystemReliability contract is required. In addition, refinements (additional QoS requirements) for the operations latest and analysis are defined.

```
rateServerProfile for RateServerI = profile
  {
    require SystemReliability;
    from latest require Performance contract
      {
        delay
          {
            percentile 80 < 20;
            percentile 100 < 40;
            mean < 15
          };
      };
    from analysis require Performance contract
      {
        delay < 4000
      };
  };
```

Figure 14: QML profile example

4 The QoS Provisioning Service

Chapter 2 and 3 described the architectural and engineering concepts that we use for the design and implementation of the QoS Provisioning Service. This chapter presents the design of the QoS Provisioning Service. We begin by describing a demonstration scenario.

4.1 A demonstration scenario

The purpose of the demo scenario presented in this section is to make our problem statement more tangible and to express the need for QoS provisioning in CORBA. The following figure shows a local network setup of 4 hosts and a PC based router.

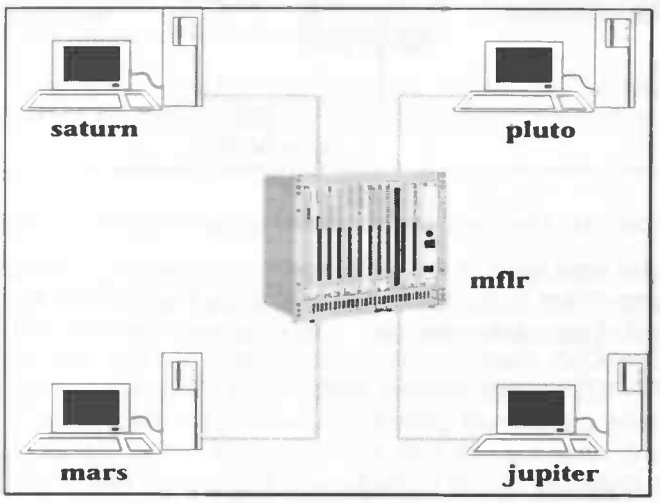

Figure 15: The local network the scenario describes

The router is called `mflr`, which stands for 'my first Linux router'. It is a PC based router that runs the Linux 2.2 operating system [39]. The other machines are `saturn`, `pluto`, `mars` and `jupiter`. There's heavy traffic between `mars` and `jupiter`, for instance a multimedia stream, that highly saturates the network. This traffic causes congestion on the local network, because the router is unable to route the large amount of packets it receives. This can be caused because the router isn't fast enough to process all the packets it receives, or if the network isn't fast enough to transfer the combined traffic generated by the hosts. Both `saturn` and `pluto` run an ORB. The CORBA client application running on `saturn` performs method invocations on a server object that is located on `pluto`. The congestion on the local network causes packets containing these method invocations to be dropped occasionally. If we're not happy with best-effort QoS network communication between `pluto` and `saturn` then we have a problem, it is not possible to use another QoS for our method invocations; unless we extend our CORBA environment with support for QoS provisioning. If we have such support for QoS provisioning then it should be possible to put requirements on the round-trip delay of method invocations for instance. These requirements are then monitored and mapped to network QoS mechanisms. The QoS Provisioning Service can decide to send the method invocations over the network using Diff-Serv Expedited Forwarding for example, if it detects degradation in the QoS.

In other words, we would like to specify, monitor, and control Quality of Service in CORBA middleware; we would like to have QoS provisioning for objects and their operations (i.e. method invocations). CORBA doesn't support this. We have decided to tackle this problem by designing a QoS Provisioning Service for CORBA.

4.2 Approach

'*Specify, monitor, and control*' hints at using a feedback-control model as the basis of our design. The following figure shows a high-level view of such a feedback-control model:

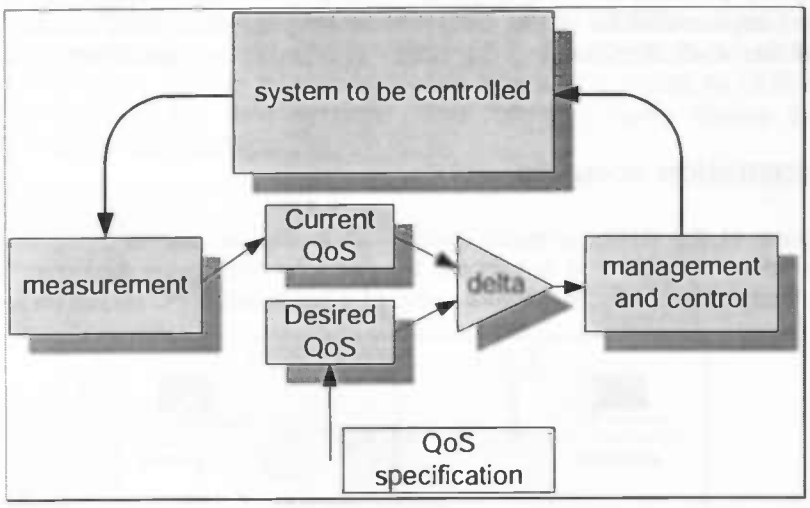

Figure 16: Feedback-control model supporting QoS provisioning

The system to be controlled is the ORB and the underlying network capabilities. The QoS requirements of the system are represented by the **Desired QoS** component. The actual measured QoS, the delivered QoS, is represented by the **Current QoS** component. The **QoS measurement** component updates the data in Current QoS component. The **QoS measurement** component measures the delivered QoS in the system. The **Delta** component generates a vector, the QoS error vector, containing the QoS requirements that are not met by the system. This vector is used by the **QoS Controller** (encapsulated by the management and control part of the figure) to adapt the system, so that it does meet the QoS requirements.

We have designed a service based on such a feedback-control model to monitor and control QoS requirements for CORBA method invocations. We've named it QPS, which stands for QoS Provisioning Service.

The remainder of this chapter is as follows: section 4.3 lists the requirements we have for QPS, section 4.4 describes the design of QPS, and section 4.5 presents a résumé of the design and explains how the requirements are met.

4.3 Requirements

The QoS Provisioning Service should implement QoS provisioning for CORBA method invocations. It should be possible for the client application to specify QoS requirements run-time. The QPS should select suitable mechanisms to implement the QoS requirements, to measure and monitor the QoS, and to adapt when the provided QoS degrades.

The QPS should be extensible. There are many Quality of Service dimensions, QPS cannot implement them all. Therefore it should be possible to add support for a QoS dimension without having to change QPS itself. So, QPS should support QoS dimension plug-ins. Also, there are many mechanisms that implement QoS at the network-level and

in the future new mechanisms may become available. The QPS can't implement all of them, so it should also allow for loading transport plug-ins, preferably at run-time.

The QPS is going to be used to increase the performance of the application, so its mechanisms to monitor the Quality of Service, to select QoS mechanisms, and to make other run-time decisions should not hog the system as a whole. This means that the overhead of QPS should be kept as low as possible.

Finally, QPS should be portable, i.e. ideally it should be possible to use QPS with another CORBA implementation without having to change any of the source code of the ORB and QPS. The QPS should not use ORB specific mechanisms. In addition, the interface provided by QPS to application objects should conform to the generic portability requirements on an ORB.

In short the requirements we have for the QoS Provisioning Service are:

1. Implement QoS provisioning for CORBA method invocations.
2. Support a wide range of QoS dimensions.
3. Be extensible, i.e. it should allow for run-time loading of classes that handle a certain QoS dimension and classes that allow for using QoS mechanisms of the network.
4. Be portable, i.e. it should not be specific to a certain ORB implementation and provide a portable interface to application objects.
5. Be minimal. Keep the overhead of QPS on top of the ORB as low as possible, i.e. measurement and control of QoS by QPS should not hog the system as a whole.

4.4 Design

The design is based on the phases identified by the ISO/IEC Quality of Service Framework [8]. This framework identifies three phases when QoS management is used: the prediction phase, the establishment phase, and the operational phase.

The following activities in the QoS Provisioning Service can be identified, based on the phases of the ISO framework:

During the prediction phase

- Load plug-ins that provide various QoS mechanisms into the system.
- Definition of QML contract types and contracts.

During the establishment phase

- Specification of QoS for operations in IDL Interfaces, more specific attaching QML profiles to IDL interfaces.
- Configuration, i.e. see if the QoS requirements can be met using the current available resources.

During the operational phase

- Enforcing QoS.
- Measuring the current QoS.
- Detect degradation in QoS.
- Adapt the system when the QoS degrades.

Each of these activities is described in detail in the sections to come.

4.4.1 Prediction phase

The QoS requirements that the application expects from the ORB are specified in the QoS Modeling Language (QML) [37]. One of the design goals of QML is to be generic, i.e. the specification language should not be specific to one or more QoS dimensions. This goal fits with our goal of allowing a wide range of QoS dimensions.

During the prediction phase the QML contract types and contracts are specified. To store QML contract types and contracts there is a **QoS Specification Repository**. This repository is also used to store QML profiles, which are typically specified during the establishment phase described in the next section. To extend this repository a QoS specification IDL interface 'Repository' is provided, in the CORBA module QPS. This interface has operations to add and interpret QML specifications of contract types, contracts, and profiles. It also has an operation to attach a profile to CORBA objects, which is explained in the next section. The following figure shows the relevant components and the IDL interface.

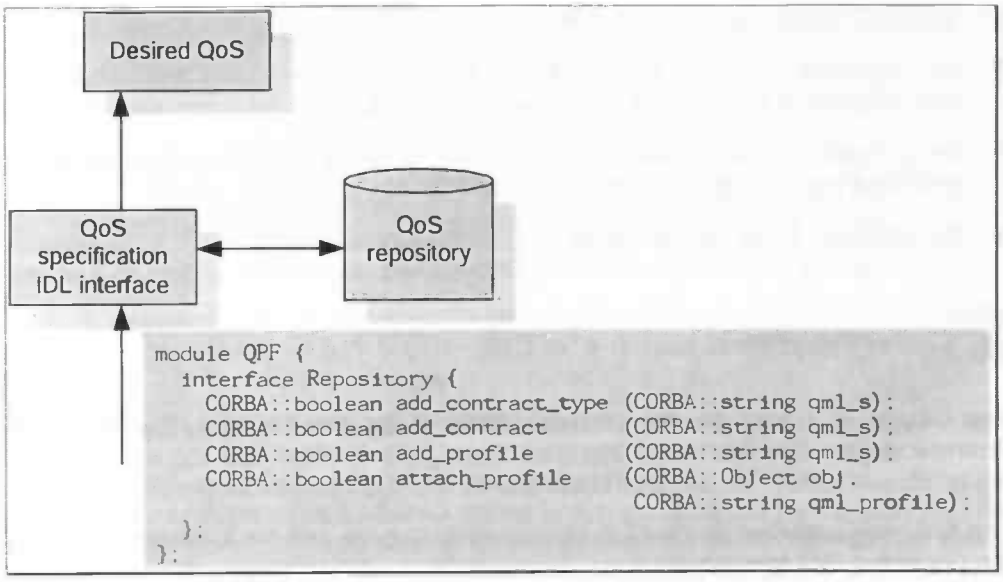

Figure 17: QoS specification with QPS

The add_* operations return a TRUE value when the QML specification given as a parameter has no syntax errors. Existing contract types, contracts and profiles will be replaced if new ones are added that have the same name. There are no checks whether or not contract types and contracts are semantically correct during the prediction phase. This is done in the establishment phase.

Another activity is to load required functionality (QoS mechanisms for particular sets of QoS dimensions) into the system. The QPS has QoS Dimension plug-ins and QoS mechanism plug-ins.

QoS dimension plug-ins

The QPS doesn't know about specific dimensions. It is designed to be generic, allowing a wide range of QoS dimensions to be supported. For instance, QPS doesn't know about the QML QoS dimension 'delay'. Support for a QoS dimension can be added run-time by loading it as a plug-in. The QoS dimension plug-ins are described in more detail later in this text.

QoS mechanism plug-ins

QoS mechanism plug-ins expose network level QoS mechanisms to the middleware layer. This is needed because QPS needs to have control of the configuration of those QoS mechanisms to deliver QoS. This plug-in type is also discussed later in this text.

4.4.2 Establishment phase

During the establishment phase applications QoS requirements of the application are negotiated. The establishment phase is divided in two steps: specification and configuration. In the specification step the QoS requirements of the client application are translated to a QPS native format. This is done by a QML to QPS translator. The second step is configuring the ORB and the network level transports according to the specified QoS requirements. Here we implement QoS using the various QoS mechanisms at our disposal.

4.4.2.1 Attaching Profiles to CORBA Objects

To attach a QML profile to a CORBA object the operation `attach_profile` is provided. The Desired QoS model contains the QoS requirements for CORBA objects that have a QML profile attached to them. Attaching a QML profile to a CORBA object changes the Desired QoS model. The QPS splits the QML profile into **profile parts**. Each profile part contains the relevant data from the QML profile for one or more operations of the CORBA object. For example, given the following QML profile QPS distinguishes three profile parts:

```
rateServerProfile for RateServerI = profile
{
  require SystemReliability;
  from latest require Performance contract
  {
    delay
    {
      percentile 80 < 20;
      percentile 100 < 40;
      mean < 15
    };
  };
  from analysis require Performance contract
  {
    delay < 4000
  };
};
```

Figure 18: Example QML profile

The profile parts QPS derives from the above QML profile are listed in the following three figures. The first profile part consists of the constraints listed in the ***SystemReliability*** QML contact, discussed in the introductory section about QML (section 3.3). The other two profile parts are refinements for the operations ***latest*** and ***analysis***.

```
numberOfFailures < 10;
TTR { percentile 100 < 2000; mean < 500; variance < 0.3; }
availability > 0.8;
```

Figure 19: Profile part relevant for all operations of RateServerI except *latest* and *analysis*

```
numberOfFailures < 10;
TTR { percentile 100 < 2000; mean < 500; variance < 0.3; }
availability > 0.8;

delay { percentile 80 < 20; percentile 100 < 40; mean < 15; }
```

Figure 20: Profile part relevant for the operation *latest*

In this example we see that for the operation ***latest*** of the ***RateServerI*** IDL interface the Performance contract has been refined. So we have different QoS requirements for ***latest*** than we have for other operations of ***RateServerI***. Therefore ***latest*** has its own profile part in QPS. The Performance contract has also been refined for the ***analysis*** operation of ***RateServerI***, so analysis also has its own profile part.

```
numberOfFailures < 10;
TTR { percentile 100 < 2000; mean < 500; variance < 0.3; }
availability > 0.8;

delay < 4000;
```

Figure 21: Profile part relevant for the operation *analysis*

The QPS uses profile parts to conveniently group all QoS requirements for an operation or a group of operations. Each profile part has a unique identification number, the profile_part_id.

The Desired QoS Model

The Desired QoS model contains QoS requirements for CORBA objects with QML profiles attached to them. The QoS requirements are grouped per profile_part_id. The QPS uses its own format for representing QoS requirements, the format is influenced by QML's concepts. The QoS Specification Interface component translates QML source to this representation. The following figure shows the layout of the data-structure of the Desired QoS Model:

```
[(profile_part_id, (QoS dimension, (contraint),
                                   (...),
                                   (constraint)),
                   (QoS dimension, ...)),
 (profile_part_id, #inherit ...,
                   (QoS dimension, ....))]
```

Figure 22: The Desired QoS Model

The #inherit construct indicates QML profile refinement, for example ***#inherit 1*** indicates refinement of profile_part_id 1.

An example Desired QoS model containing the three profile parts looks as follows:

```
[(1. (numberOfFailures. (<. 10. no/year)).
     (TTR.              (percentile. 100. <. 2000).
                        (mean. <. 500).
                        (variance. <. 0.3)).
     (availability.     (>. 0.8))).
 (2. #inherit 1.
     (delay.            (percentile. 80. <. 20 msec).
                        (percentile. 100. <. 40 msec).
                        (mean. <. 15 msec))).
 (3. #inherit 1.
     (delay.            (<. 4000 msec)))]
```

Figure 23: Example Desired QoS model

4.4.2.2 Configuring Quality of Service

The QPS intercepts all method invocations sent by the client application. If a method has QoS requirements, i.e. if QPS can find a profile part to go with the method invocation, QoS negotiation is triggered. There are 2 possibilities: (1) QoS was already negotiated for this profile part, and (2) QoS is negotiated for the profile part for the first time. In the first case QPS uses the already negotiated QoS configuration to send the method invocation to the server object. In the second case QPS needs to select a network level transport that is able to implement the QoS requirements set for that method invocation.

The QPS Inter-ORB Protocol (QIOP)

QIOP is a high-level protocol that implements a transport plug-in (pluggable protocol) for the higher-level GIOP messages. QIOP itself is an umbrella for various transport channels and doesn't know how to send data over the network. It relies on (sub) plug-ins for that, not to be confused with pluggable protocols. These plug-ins allow QPS to use Quality of Service mechanisms available at the network-level. Plug-ins for QIIOP are loaded into the system during the prediction phase. One plug-in could provide best-effort QoS using TCP/IP, another plug-in could provide group communications to enhance reliability, yet another plug-in could provide support for expedited forwarding of IP packets.

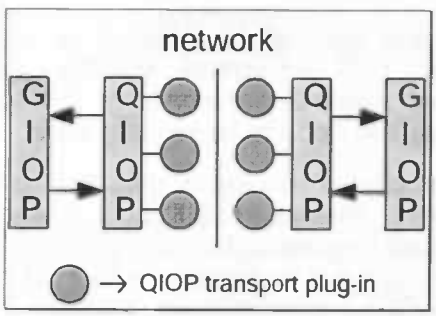

Figure 24: Transport plug-ins

A communication channel opened by a QIOP plug-in is called a **QIOP QoS channel**. A QoS configuration for a profile part is a mapping from a profile part id to a QIOP QoS channel.

QIOP QoS channel plug-ins have various responsibilities. They have to implement receive and send functions to transfer octet-streams over the network, connectors and acceptors, functions to shutdown a connection, etc. They also have to implement functions that are called by QPS to check whether or not QoS requirements can be met. QPS uses these functions to configure QoS based on the requirements specified by the user of QPS.

Configuring a QIOP QoS channel that implements the QoS specified in a profile part

First QIOP tries to match one of the existing QoS channels with the specified QoS in the profile part. It asks each QoS channel whether they conform to the QoS as specified in the profile part or not. Each QoS channel should implement the `boolean conforms_to (profile_part_id)` method for this. If that method returns `true` then QIOP routes all method invocations related to the profile part through this QoS channel. If none of the current QoS channels conform to the QoS as specified in the profile part, then QIOP asks each of the QoS channel plug-ins whether they can create a QoS channel that conforms to the specified QoS. The QoS channel factory of a QIOP plug-in should implement the method `connect_using (profile_part_id)` for this. The method should return a null pointer if it's unable to create such a QoS channel. It should return a reference to the newly created QoS channel when it is able to (and has created) a QoS channel that conforms to the given QoS specification. In case none of the QIOP plug-ins is able to create a QoS channel matching the QoS in the profile part, then QIOP throws an exception.

A QIOP QoS channel plug-in has to know certain things the about QoS dimensions (one or more) that it provides QoS network-level mechanisms for. For example, a QoS channel plug-in that is able to prioritize traffic may support the QoS dimension 'delay'. The QoS channel plug-in and the QoS dimension plug-in have to agree on the definition of 'delay' and how it is measured. The following figure depicts the relationship between both plug-in types.

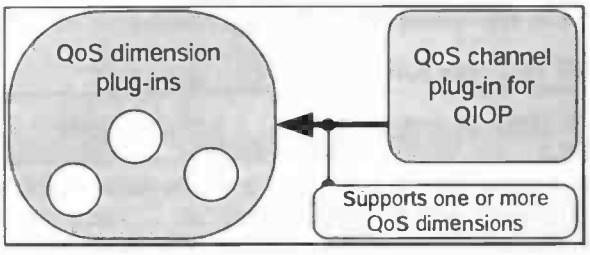

Figure 25: Relationship between QoS channel plug-ins and QoS dimension plug-ins

4.4.3 Operational phase

During the operating phase the QoS is enforced, i.e. QPS uses QoS mechanisms available to implement the QoS requirements that the application has specified. The provided QoS is monitored. When degradation in the QoS is detected the system should be adapted so that the required QoS can be provided again.

The operational phase is divided into four steps. Enforcing QoS (1) involves routing CORBA method invocations through a QIOP QoS channel that implements the QoS requirements for that method invocation. Measuring QoS (2) involves measuring the QoS performance of the system, so that degradation in QoS (3) can be detected. Finally, the system should adapt its QoS configuration (4) when the QoS performance degrades. Each step is described in a separate section.

4.4.3.1 Enforcing Quality of Service

As mentioned in the previous section, QPS intercepts all method invocations generated by the client application. Portable Interceptors are used to implement this. An interceptor at request-level inspects each method invocation and tries to map it to a profile part id. The ORB's interceptor implementation provides the interceptor with a pointer to a data-structure that contains the request. If the request-level interceptor is able to map the method invocation to a profile part, then the service context list of the request is extended with a new service context containing information regarding the profile part id. The following figure depicts this procedure.

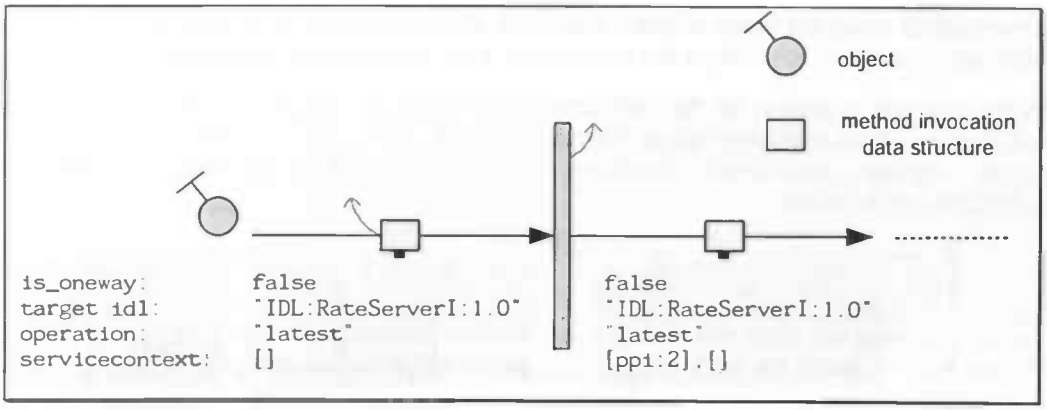

Figure 26: Matching method invocations with profile parts

The QIOP protocol filters the service context list of each GIOP message that it has to send over the network. If QIOP finds a service context containing a profile part id hint, then it routes the message through the QoS channel that is configured for that profile part.

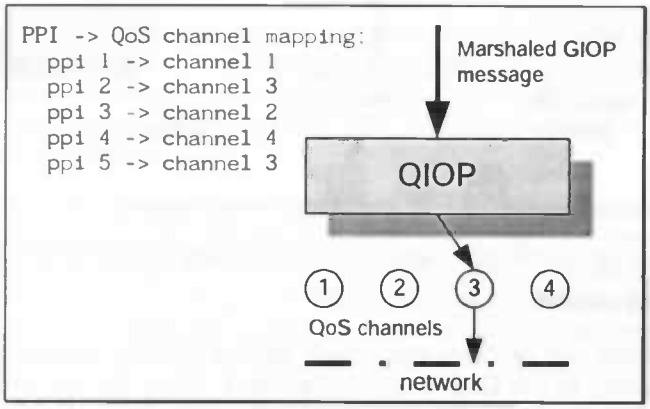

Figure 27: QIOP filters service contexts looking for profile part ids

In this example (see figure) QoS channel 3 should be used to transfer method-invocations over the network related to profile part id 2. QIOP maintains a mapping from profile part id to a QoS channel reference. This mapping is configured when QoS is negotiated.

4.4.3.2 Measuring Quality of Service

To monitor the QoS we need to measure it. The QPS has sensor components for this task. The interceptors inform sensor components of method-invocation. The interceptors only filter method-invocations, they delegate measuring of QoS to the sensor components. There can be different kinds of sensors in the system, located at the interception points of the interceptors that are in use. The sensor receives the method name, target object name, and whether or not the method invocation is one-way. The sensor is then able to lookup the matching profile part id, if any. This profile part id is also returned to the interceptor, so that the interceptor can build a new service context that includes the profile part id.

A QoS dimension plug-in can subscribe to sensor notifications, by registering a notification handler for it. Sensors support a pre-defined set of notifications, for instance request-level sensors offer start-method-invocation and end-method-invocation notifications. Subscribing to sensor notifications is done by the method **configure_sensors_for (profile_part_id)** that all QoS dimension plug-ins should implement. This method is called when a profile that has QoS requirements for the QoS

dimension is attached to an object. The QoS dimension plug-in should also implement notification handlers to process the notification it will receive from the sensors.

As an example, a plug-in for the QoS dimension 'delay' is typically interested in receiving notifications of method invocation startup and end, so it can calculate the delay. The plug-in registers notification handlers for start-method-invocation and end-method-invocation notifications.

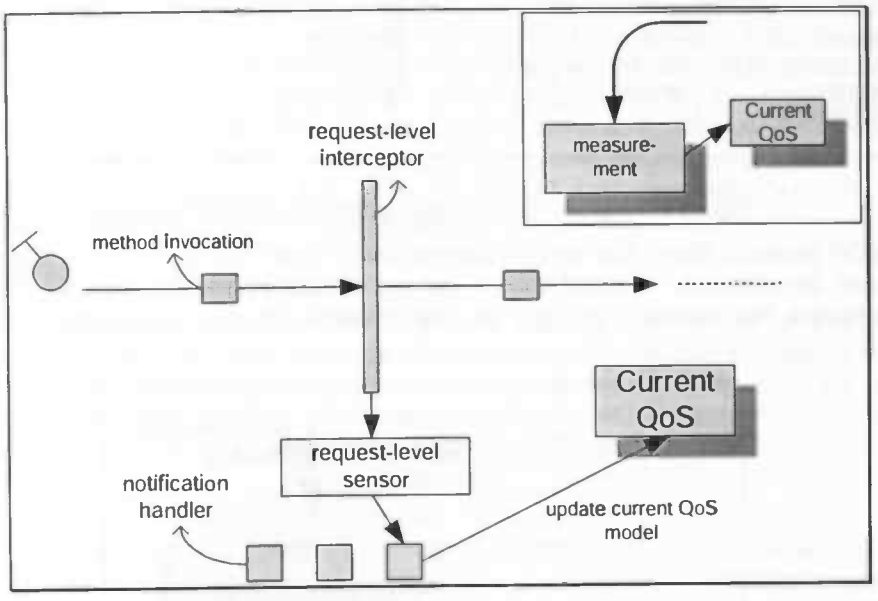

Figure 28: Notification handling and updating the current QoS model

The Current QoS Model

Notification handlers can process these notifications, and thus measure QoS. They send these measurements to the QoS model component, along with the profile part id and the name of the QoS dimension. The QoS model component then updates the current QoS model.

The Current QoS model contains the measured QoS for each profile part. The meaning of the values depend on the QoS dimension and constraint type, depending on the QML specification. The QPS initializes the values in the current QoS model with a special *N/A* (not available) value, denoting that there is no measured QoS yet.

4.4.3.3 Detecting degradation in QoS

After processing the notifications, the sensor invokes a function `delta` of the QoS model to generate an error vector. This error vector contains information about profile parts for which QoS has degraded, i.e. for which QoS is not met. The error vector looks like this:

```
((profile_part_id,
  ((QoS dimension,
    ((measured value, constraint),
     ...,
    ),
   (QoS dimension, ...)
  ),
 (profile_part_id, ...)))
```

Figure 29: Format of the error-vector

If, for instance, the delay requirements of profile part id 2 are not met, then the error vector looks something like this:

```
(
  (2, ((delay, ((96, (percentile, 100, <, 40 msec)))))
)
```

Figure 30: Sample error-vector

96 percent of the invocations related to profile part id 2, in this case the operation latest of RateServerI, has a measured delay value that is less than 40 msec. However, 100 percent is desired.

The error vector generated by delta is inspected by a QoS controller component, which then tries to adapt the system's QoS configuration so that desired QoS is met in future method invocations.

4.4.3.4 Adapting the system after QoS degradation

The controller will try to reconfigure the QoS settings for each profile part id listed in the error vector. It uses an algorithm similar to the one used for QoS negotiation. The QoS controller remembers what configurations failed to deliver the requested QoS. The algorithm that is used to adapt the system after QoS degradation has been detected consists of the following steps:

1. Ask the factory of the plug-in that is currently used to deliver QoS for the profile part id if it can configure a new channel that *does* meet the QoS requirements for the profile part id. A reference to the QoS channel currently in use is provided so the factory knows the current settings. Factories that fail to configure a QoS channel for profile part id are remembered by the QoS controller, so that they won't be asked again to create a QoS channel for the profile part id.
2. If the factory of the QoS channel currently in use is unable to configure a QoS channel for the profile part id, then the QoS controller will ask other open QoS channels whether they can provide the QoS for the profile part id or not.
3. If not, then the QoS controller will traverse the list of QoS channel factories asking them if they can create a QoS channel that implements the QoS requirements of profile part id. Channels that are remembered not to be able to deliver the QoS are skipped.
4. If no QoS channel matching the QoS requirements of the profile part id could be assigned, then the QoS controller throws an exception to inform the client application.

4.5 Résumé

We defined the following requirements for the QoS Provisioning Service. In retrospect, are these requirements met?

Implement QoS provisioning for CORBA method invocations

The QPS does provide QoS provisioning for method invocations. The approach is to use a feedback-control loop as the basis of a service that allows QoS to be specified by the application. The feedback-control loop measures and controls the QoS of the system.

Support a wide range of QoS dimensions

The QPS supports run-time loading of QoS dimension plug-ins, so it is able to support a wide range of QoS dimensions. Support for new QoS dimensions can be added easily to QPS. All that needs to be done is implement a new class (a subclass of an abstract

dimension class in QPS) and implement the interface that QPS expects from a QoS dimension plug-in. In addition the implementing the QoS dimension plug-in an existing QoS channel plug-in should be adapted to support this dimension or a new QoS channel plug-in should be implemented that support the new dimension. The QPS also defines interfaces that such QoS channel plug-ins should implement.

Be extensible

The QPS should be extensible, i.e. it should allow for run-time loading of classes that handle a certain QoS dimension and classes that allow for using QoS mechanisms of the network. The QPS is also able to load transport plug-ins for the QIOP protocol at run-time. This support for loading plug-ins at run-time requires that the language that QPS is implemented in and the operating system support dynamic loading of libraries/classes. Most combinations of operating system and programming languages are able to support dynamic loading, for instance Java, C++ on Win32 systems, and C++ and modern Unices. We believe the system is quite extensible.

Be portable

The QPS should be portable, i.e. it should not be specific to a certain ORB implementation. The QPS uses portable interceptors and pluggable transports to extend the ORB, each ORB supporting these extensions mechanisms can be used in combination with QPS, so QPS itself is quite portable. Also, the applications interface with QPS using IDL, so applications access QPS in a portable manner.

Keep overhead as low as possible

The QPS should be fast i.e. measurement and control of QoS by QPS should not hog the system as a whole. The QPS however is additional functionality, and thus requires CPU time. It's important that the performance of QPS is predictable, i.e. the performance penalty should be constant for each method-invocation.

The performance of the system depends on the performance of the interceptor mechanism of the CORBA implementation. We think that the performance penalty of using interceptors is minimal. The implementation of the interceptors in QPS uses hash-tables to lookup QoS requirements for method invocations. Depending on the hash-table implementation this could also be a minimal and constant performance hit.

The QIOP protocol has to inspect the data it transports to filter service contexts (described in the section about Enforcing QoS). This is also a reasonably constant performance penalty, depending on the size of the service context list (each service context in the list is inspected by QIOP).

In conclusion, QPS does invoke performance penalties. We believe that the loss of performance is minimal given the CPU power of computer systems today. Adding more CPU power can compensate the loss of performance caused by the overhead of QPS. A CORBA system with QPS is less optimal for method-invocations that have no QoS requirements, since these method-invocations are filtered by the interceptor mechanism anyway.

5 Implementation of the QoS Provisioning Service

This chapter describes our prototype implementation of the QoS Provisioning Service. We start with an overview of the CORBA implementation that we've used. We describe the key parts of the implementation in sections 5.2, 5.3, 5.4 and 5.5. This chapter concludes with a section about the status of the implementation.

5.1 ORBacus

Our prototype is implemented on top of ORBacus 3.1.3 [34]. ORBacus is an open-source [40] implementation of the CORBA 2.0 standard. ORBacus has C++ and Java language mappings. For this prototype we use the C++ mapping. All source of the prototype is written in C++. ORBacus supports the Open Communications Interface (OCI). Our QIOP protocol is plugged into the ORB using OCI.

5.2 The QoS model

The first part of the prototype that we've built is the group of classes that implement the QoS model. Our QoS model, implemented by the class **QoSModel**, can be extended with QoS requirements specified in QML. The internal classes that the QoSModel class uses are based on parts of the QML. For instance, QML includes constraints on dimensions and has two kinds of those constraints: simple constraints and aspect constraints. The QoSModel class' internal classes include SimpleConstraint and AspectConstraint.

5.2.1 Specification of QoS

We chose not to implement a full QML parser and compiler. Therefore the QoS repository is also not implemented. Instead, we use a newly defined intermediate language to specify QoS requirements. The internal class **Interpreter** of the QoSModel class implements an interpreter for this intermediate language and takes care of constructing a desired QoS model. The syntax of the intermediate language is divided into two parts: specification of profile parts and attaching profile parts to IDL interfaces and their operations.

The following figure is depicts how profile part specification is done. The syntax is the same as we used in our description of the desired QoS model in the previous chapter. Profile part 2 is a refinement of profile part 1, as indicated by using the #inherit construct.

```
[(1. (Delay.            (percentile. 80. <. 20 msec).
                        (percentile. 100. <. 40 msec).
                        (mean. <. 15 msec))).
 (2. #inherit 1.
     (Delay.            (<. 4000 msec)))]
```

Figure 31: Profile part specification example

The syntax of attaching profile parts to IDL interfaces and their operations is demonstrated in the next figure. Profile part 1 is attached to the IDL interface Hello, and

is default for all operations in that interface. Profile part 2 is attached to the operation hello in the IDL interface hello. So, all operations in the IDL interface Hello have QoS requirements listed in profile part 1, except the operation hello that has additional requirements (refinement) listed in profile part 2.

```
@ IDL:Hello:1.0 *       -> 1
@ IDL:Hello:1.0 hello   -> 2
```

Figure 32: Attaching profile parts to IDL

Applications add QoS specifications written in the intermediate language described above to the desired QoS model using the add operation of the QoSModel class. The next figure depicts this.

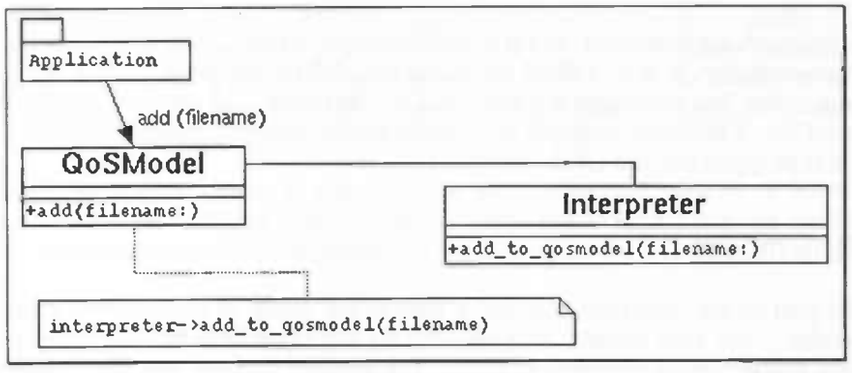

Figure 33: Specifying QoS

5.2.2 How QoS is stored in the desired QoS model

The QoSModel class maintains a hash table of instances of the ProfilePart class. The hash table key is of type profile_part_id_t. A ProfilePart class holds a list of instances of the DimensionConstraints class. A DimensionConstraints instance holds QML constraints related to the same QML dimension. The Constraint abstract class represents a QML constraint. QML distinguishes two constraint types, simple constraints and aspect constraints. These are represented by the SimpleConstraint and AspectConstraint classes respectively. The following figure depicts an UML class diagram of the above.

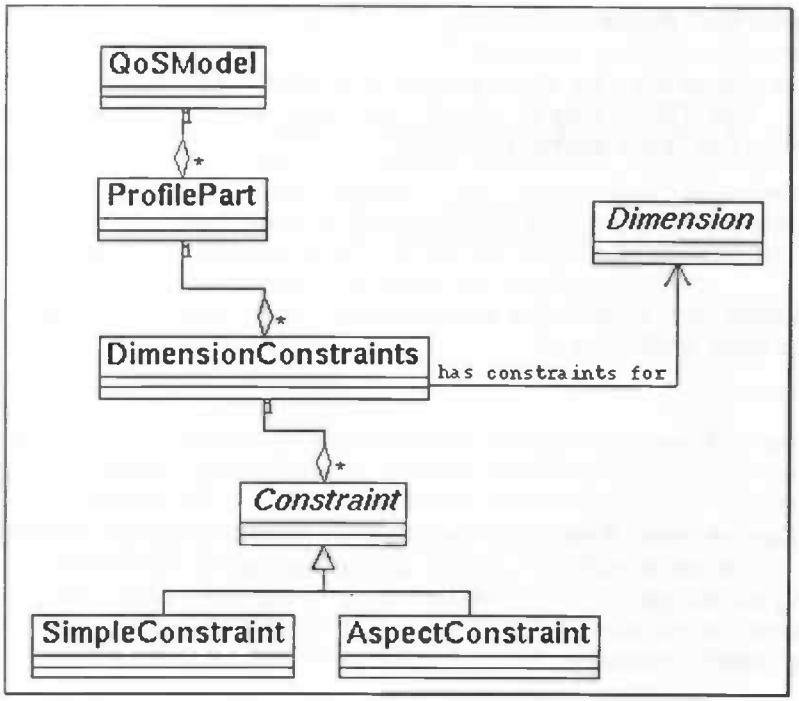

Figure 34: QoS model structure

The Interpreter class is responsible for extending the QoS model with new profile parts. These profile parts are specified in a QPS intermediate language, described in the previous section. The following figure is the sequence diagram for QoS specification. The application can add QoS specifications to the QoS model using the **add (filename)** operation of **QoSModel**. The QoSModel add (filename) operation asks the **Interpreter** class to interpret the specifications in the filename by calling the **interpreter (filename)** method. This method parses all the profile part specifications in the given filename, and creates a **ProfilePart** instance for each of them. Each profile part may contain constraints related to one or more QML dimensions. QML constraints for a QML dimension are grouped in an instance of **DimensionConstraints**. After parsing and creating a **ProfilePart**, **Interpreter** adds the profile part to the **QoSModel** using **add (ProfilePart *)**.

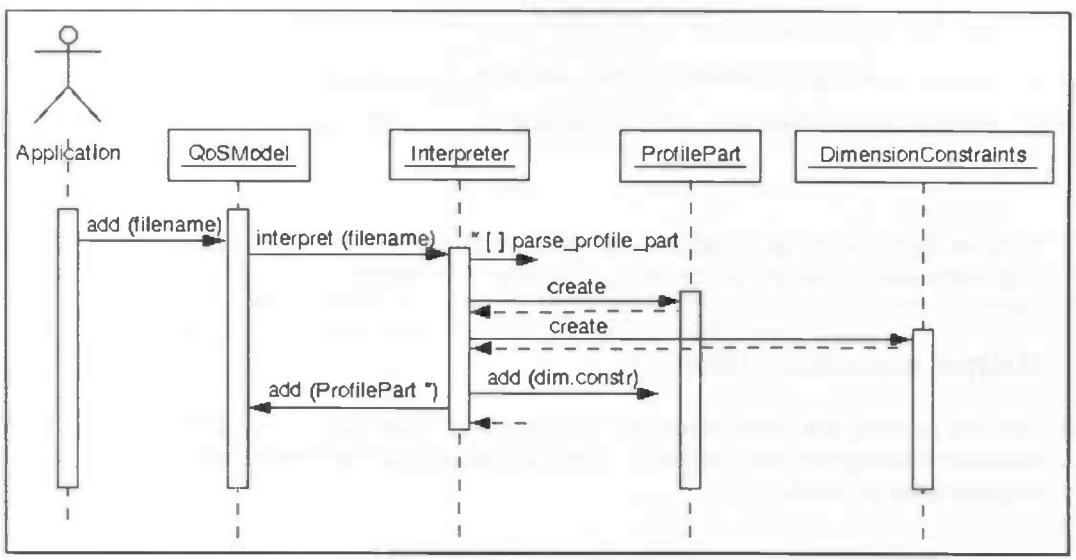

Figure 35: Sequence diagram for QoS specification

5.2.3 Querying the QoS model

The QPS prototype uses the Visitor pattern [4] to allow QIOP plug-ins to do conformance checking. The QIOP plug-in should sub-class the **ConstraintVisitor** class and implement the two virtual **accept ()** methods:

```
class ConstraintVisitor {
  virtual bool accept (SimpleConstraint *);
  virtual bool accept (AspectConstraint *);
};
```

The QoSModel class implements a method that can be called to visit all the constraints related to a given profile part id:

```
class QoSModel {
  ...
  bool visit (ConstraintVisitor *, profile_part_id_t);
  ...
};
```

The visit method visits all the constraints that are related to the given profile part id and for each constraint it calls the proper accept method in the given constraint visitor (depending on the type of the constraint: simple or aspect). Visitor allows us to hide the representation of the internal data-structures of the QoSModel class, yet allowing QIOP plug-ins to inspect constraints.

The method described above is currently the only supported method of traversing the constraints in the QoS model. The QoSModel class itself doesn't use the visitor pattern, since it is able to traverse its data structures directly.

5.2.4 Storing QoS measurements

The QoSModel class also implements the current QoS model. The measured QoS values are stored with the desired QoS in the SimpleConstraint or AspectConstraint classes. The QoSModel class has the following operations that can be used to manipulate the current QoS model.

```
class QoSModel {
  ...
  enum measure_op_t {
    MOP_ASSIGN, MOP_UNION, MOP_DIFFERENCE, MOP_INTERSECTION
  }
  void process_measurement (enum measure_op_t, profile_part_id_t,
                            Dimension *, int);
  void process_measurement (enum measure_op_t, profile_part_id_t,
                            Dimension *, float);
  void process_measurement (enum measure_op_t, profile_part_id_t,
                            Dimension *, string, ...);
  ...
};
```

QoS measurements are updated by notification handlers, called by sensors. Sensors and notification handlers will be discussed in a later section.

5.3 Method invocation filtering

First we discuss the class diagrams of Interceptors and Sensors and discuss the relation between Interceptors and Sensors. After that we look at the interceptor and sensor at the request-level in detail.

5.3.1 Interceptors

We extended ORBacus with a prototype implementation of Portable Interceptors, based on an initial submission [41] to the Portable Interceptors RFP [27]. The prototype of the portable interceptor submission was implemented for an EURESCOM project [42] [43].

This implementation is not mature yet, various modifications had to be made in order to make it usable for our QPS prototype. These modifications included both fixes for bugs and implementation of missing features.

For this prototype we've implemented an interceptor at request-level. Its super-class is the **RequestInterceptor** class that the Portable Interceptors implementation provides. The interceptor implements three methods: **pre_invoke** and **post_invoke**. The **pre_invoke** method is called when a method is invoked, the **post_invoke** method is called when the method invocation finishes (if it's not a one-way method invocation). In the previous chapter we showed that interceptors do two things in QPS:

- Initiate the enforcement of QoS
- Initiate the measurement of QoS

Initiation of the QoS enforcement consists of adding a new service context to the request data-structure that includes the profile part id of the QoS if a method invocation has QoS requirements. Measurement of QoS is initiated by informing a sensor component at the same interception level of method invocation. The **post_invoke** method of the interceptor is only involved in measurement of QoS. The **pre_invoke** method is involved in both enforcement and measurement of QoS. Support for changing the service context list of a request data-structure had to be added to the Portable Interceptors implementation. The interfaces were already in place, but the method bodies were empty.

The following figure depicts the class diagram of the Portable Interceptor class at request-level. **POI_RequestInterceptor** is provided by the Portable Interceptors implementation, and **RequestInterceptor** is the request-level interceptor of QPS.

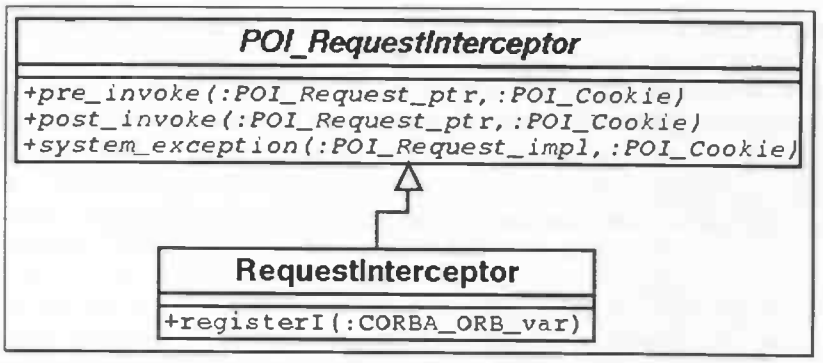

Figure 36: Request-level Portable Interceptor class diagram

The **system_exception** method is not used by QPS. The **registerI** method registers the interceptor at the ORB.

5.3.2 Sensors

The prototype includes a sensor component at request-level. The interceptor at request-level informs this sensor component of method invocation start and end.

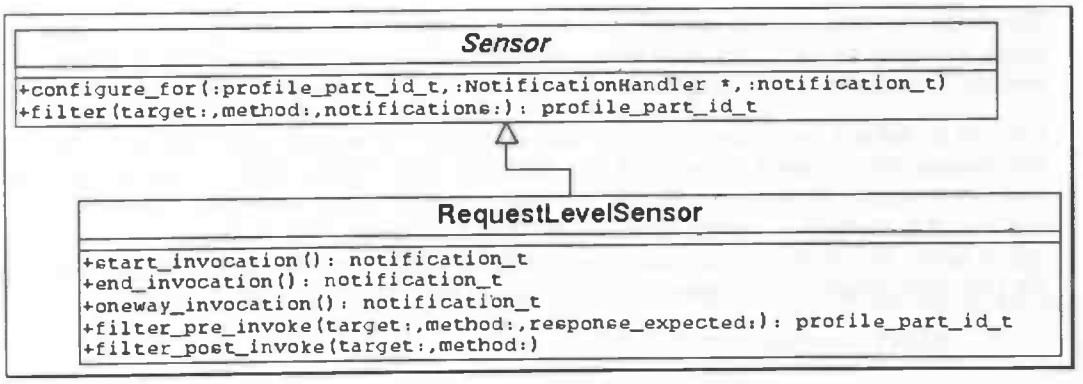

Figure 37: Sensor class diagram

The **configure_for** method of the abstract class Sensor is used to add notification handlers to the sensor. Notification handlers are registered per profile part id. The first parameter is the profile part id we want to monitor notifications for, the second parameter is a reference to the notification handler object, and the third parameter is a bit-vector containing the notifications that should be monitored. The **filter** method is used to filter a method invocation, it's never called directly by an interceptor; it's a helper method that can be used by filter methods in subclasses of **Sensor**. The **RequestLevelSensor** class for instance implements two filter methods that wrap the filter method of the **Sensor** class.

5.3.3 An in-depth look at the request level interceptor

The following source fragment is the **pre_invoke** method of **RequestInterceptor**. Some obscure details have been omitted. We see that **RequestLevelSensor** (in the namespace QPS) is asked to filter the method invocation. When a profile part is found. the service context list is extended with a new service context containing the profile part id.

```
PortableInterceptor::InterceptionResult
RequestInterceptor::pre_invoke (PortableInterceptor::ServerRequest_ptr the_request,
                                PortableInterceptor::Cookie &the_cookie)
{
  // Filter the method invocation.
  QPS::profile_part_id_t ppi =
    QPS::RequestLevelSensor::instance () -> filter_pre_invoke (
      the_request -> get_target () -> target_object () -> _OB_typeId (),
      the_request -> get_operation (),
      the_request -> response_expected () ? true : false);
  // If there is a profilepart for the method invocation then
  // update the service context.
  if (ppi > 0)
  {
    /* Put profile-part-id into context data. */
    IOP_ServiceContextList *ctxs = the_request -> get_contexts ();
    IOP_ServiceContext *ctx = new IOP_ServiceContext;
    ctx -> context_id = 1975; // Randomly chosen context id.
    char str[72]; snprintf (str, sizeof str - 1, "ppi %lu", ppi);
    int count = strlen (str) + 1;
    ctx -> context_data.length (count);
    CORBA_Octet* oct = ctx -> context_data.data ();
    strcpy ((char *)oct, str);
    ctxs -> append (*ctx);
    try {
      the_request -> set_contexts (*ctxs);
    }
    catch (...) {
      exit (EXIT_FAILURE);
    }
  }
  return PortableInterceptor::PROCEED_REGULAR;
}
```

The post_invoke method doesn't have to extend the service context list, and thus only informs the RequestLevelSensor that the method invocation has ended.

```
void
RequestInterceptor::post_invoke (POI_Request_ptr  the_reply,
                                 POI_Cookie       the_cookie)
  {
    QPS::RequestLevelSensor::instance () -> filter_post_invoke (
        the_reply -> get_target () -> target_object () -> _OB_typeId (),
        the_reply -> get_operation ());
    CORBA_string_free ((char *) the_cookie);
  }
```

5.3.4 An in-depth look at the request-level sensor

The following code fragment configures the sensor to call a notification handler when a notification is generated for a certain profile_part_id. Notifications are implemented as bit-vectors.

```
void
Sensor::configure_for (profile_part_id_t     profile_part_id,
                       NotificationHandler  *handler,
                       notification_t        notifications)
{
  ProfilePartData *p;
  if ((p = spd [profile_part_id]) == 0) {
    p = spd [profile_part_id] = new ProfilePartData;
  }
  ProfilePartData::HandlerData *hd = new ProfilePartData::HandlerData;
  hd -> notifications_ = notifications;
  hd -> handler_ = handler;
  p -> notification_handler_l.push_back (hd);
}
```

Sensor is an abstract class. Subclasses of Sensor, e.g. RequestLevelSensor, define notifications and use the functionality implemented in Sensor to distribute these notifications. Distributing notifications is done by the filter method of the Sensor class. First the filter method looks up the profile_part_id that is attached to the given method invocation. If there is a profile_part_id attached to the given method invocation, then the given set of notifications are distributed to the notification handlers that are configured to handle them. After handling the notifications the QoS model is asked to perform a delta operation. The delta operation compares the current QoS model with the desired QoS model, and reconfigures the system when QoS degradation is detected. The profile_part_id is returned to the caller.

```
profile_part_id_t
Sensor::filter (const string target, const string method,
                notification_t notifications)
{
  profile_part_id_t p;
  bool dodelta = false;

  if ((p = QoSModel::instance ()
           -> mapto_profile_part_id (target + "!" + method)) == 0
   && (p = QoSModel::instance ()
           -> mapto_profile_part_id (target + "!" + "*"))    == 0) {
      return 0;
  }

  for (list<Sensor::ProfilePartData::HandlerData *>::iterator
       it  = spd[p] -> notification_handler_l.begin ();
       it != spd[p] -> notification_handler_l.end ();
       ++it) {
    Sensor::ProfilePartData::HandlerData *hd = *it;
    if ((hd -> notifications_ & notifications).any ()) {
      hd -> handler_ -> notify (p, notifications);
      dodelta = true;
    }
```

```
  }

  if (dodelta) {
    QoSModel::instance () -> delta ();
  }

  return p;
}
```

The RequestLevelSensor class defines two wrapper method around `Sensor::filter`, `filter_pre_invoke` and `filter_post_invoke`. These methods are called by the request-level interceptor of QPS.

```
profile_part_id_t
RequestLevelSensor::filter_pre_invoke (const string target,
                                       const string method,
                                       const bool   response_expected)
{
  profile_part_id_t ppi;
  if (response_expected) {
    ppi = this -> filter (target, method, start_invocation_);
  } else {
    ppi = this -> filter (target, method, oneway_invocation_);
  }
  return ppi;
}

void
RequestLevelSensor::filter_post_invoke (const string target,
                                        const string method)
{
  this -> filter (target, method, end_invocation_);
}
```

5.4 The QIOP OCI Transport

The QIOP OCI transport is a copy of the IIOP OCI transport that already was available in ORBacus. For the prototype we were interested in providing priority scheduling of packets sent by the ORB. We have chosen DiffServ to implement this at the network-level. QIOP plug-ins may mark packets sent by QIOP with a DiffServ marker.

5.4.1 Plug-ins

The central component of the QIOP protocol is the plug-in manager. The manager has various responsibilities:

- Provide mechanism and interface for factory plug-in loading, i.e. loading QIOP QoS channel plug-ins.
- Act as a registry for these plug-ins.
- Provide filtering mechanisms for octet streams. These are used by QIOP's implementation of the OCI_Transport_impl class.
- Provide mechanisms to configure and reconfigure QIOP QoS channels for profile parts. Configure() is used by send_filter() reconfigure is used by delta() of QoSModel.

The plug-ins have various responsibilities:

- Identification, via the method `name`
- Configuration of QoS for a profile part, via the methods `configure` and `reconfigure`
- Reporting its configuration, via the `get_configuration` method

- Providing hooks for the pre and post send filters of the plug-in manager, via the method `send_pre_filter` and `send_post_filter`

The following is a class definition for a plug-in that implements DiffServ expedited-forwarding [20]:

```
class QIOP_EF: public QPS_ORB::OCI_QIOP_TP_QoS_Mechanism {
  public:
    virtual string name              (void);
    virtual bool   configure         (QPS::profile_part_id_t);
    virtual bool   reconfigure       (QPS::profile_part_id_t);
    virtual string get_configuration (QPS::profile_part_id_t);
    virtual void   send_pre_filter   (OCI_Transport *);
    virtual void   send_post_filter  (OCI_Transport *);
};
```

The `send_pre_filter` is of particular interest; it marks the packets with the DiffServ expedited-forwarding marker (0x2e):

```
int fd = oci_transport -> handle ();
int v = 0x2e;
assert (setsockopt (fd, SOL_IP, IP_TOS, &v, sizeof (v)) == 0);
```

The `send_post_filter` turns off the marking of packets. The send filter hooks provided by the plug-ins are used by the plug-in manager. The plug-in manager filters all outgoing traffic of the QIOP protocol. If a QPS service context is detected, then the send pre and post filters of the configured plug-in are called.

5.4.2 Shortcomings of OCI

QIOP currently doesn't support plug-ins that need their own file-descriptor. To accommodate this more flexible handling of I/O events is needed. Currently the GIOP layer of ORBacus directly uses the file-descriptor owned by the OCI transport for event handling. It's not possible for an OCI pluggable protocol to have more than one file-descriptor. Pluggable protocols that may require more than one file-descriptor can not be implemented without resorting to ad-hoc solutions, unless parts of the ORBacus source are modified. The following figure depicts the event-handling problem.

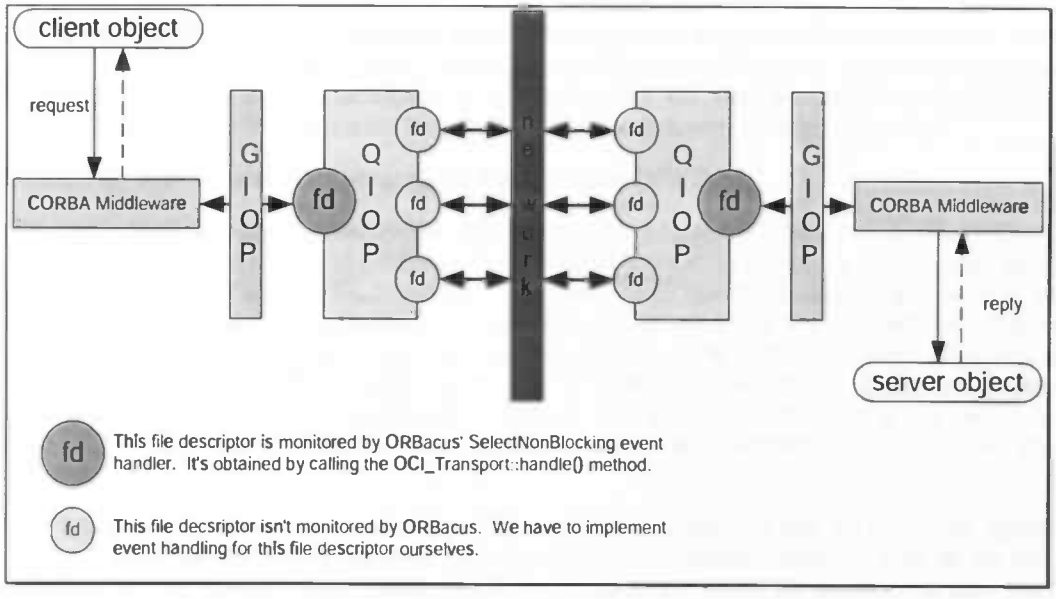

Figure 38: Event handling problems

Furthermore, OCI only allows protocols to be plugged into the ORB at compile-time. Also, parts of the ORB source have to be changed to register the OCI plug-in:

- The QIOP acceptor and connector had to be registered in the IIOPInit.cpp source of ORBacus.
- A method `get_qnet_object` had to be added in ORB.cpp of the ORBacus source, to allow CORBA applications to create an IOR for the QIOP protocol so that they can connect to a QIOP server ORB and request a reference to an object.
- Support to convert a string to an IOR had to be added to ORB.cpp as well.

5.5 Résumé

Our prototype has shown that QoS can be provided for CORBA without changing the implementation of the ORB itself. To accomplish this the ORB needs to provide two extension mechanisms: portable interceptors and pluggable protocols. The most difficult part of the prototype implementation was the propagation of QoS configuration settings through the ORB. A portable interceptor filters method-invocations and extends the service context if QoS requirements are specified for it. The service contexts should be filtered by the QIOP pluggable protocol so that QoS can be provided for the method-invocation.

The Portable Interceptors are sufficient for filtering method-invocations. However, the OCI pluggable protocol mechanism has some shortcomings, mostly related to handling of I/O events in the QoS mechanisms that can be plugged into the QIOP protocol. The current implementation of QIOP doesn't support QoS mechanisms that have their own file-handle. Therefore, only marking of packets is supported by QIOP, sufficient for implementing QIOP plug-ins that support DiffServ.

6 Conclusions & Future work

Conclusions:
Middleware is a layer of software that resides between the application and the underlying heterogeneous layers of operating systems, communication protocols and hardware. It provides *separation of concern*: the middleware isolates hardware, operating systems, and communication protocols from the rest of the system; the applications on top of the middleware.

Quality of Service (QoS) aspects like performance, security, and reliability are important to middleware systems. Still, there is no support for QoS in current middleware systems.

A commonly accepted middleware standard is OMG CORBA. This document describes the design and implementation a QoS provisioning service for CORBA middleware.

In this chapter we present our conclusions and suggestions for future work. We come back to the research questions stated in the introductory chapter.

How are QoS requirements specified by the application?

QoS requirements are specified by the application in the QoS Modeling Language (QML). The QML is a general-purpose Quality of Service specification language, it is not tied to any particular domain, such as real-time or multi-media systems, or to any particular QoS category, such as reliability or performance. We chose to use QML since it doesn't require changes to IDL, which meets one of our constraints that we aren't allowed to change CORBA. Another reason for using QML is that it is a generic QoS specification language, which coincides with our goal of supporting a wide range of QoS dimensions.

The QPS has a QoS repository that can be used to store QML specifications.

How do we manage and control QoS in the DPE?

We've designed a QoS provisioning service based on a feedback-control model. QoS can be specified by the application. The QPS configures the system according to these specifications and the control-loop of QPS, based on the feedback-control model described in chapter 4, measures and controls the QoS during system operation.

How does end-to-end QoS propagate through the DPE?

The application can specify its QoS requirements using QML. These specifications (QML contract types, contracts, and profiles) are stored in the QoS repository. The application can attach a QML profile to IDL interface. It is possible to specify QoS per operation in an IDL interface. When a QML profile is attached to an IDL interface the desired QoS model of QPS is updated with the new QoS requirements. The QPS monitors all method invocations of the CORBA middleware. For method invocations that have QoS requirements in the desired QoS model it tries to find a QoS mechanism at the network-level. This QoS mechanism is then used to transfer the method invocation over the network.

The following figure contains the two QoS interface points in the ODE. The upper interface is used by the application to specify its QoS requirements. This is an IDL interface to the QPS. The lower interface is used by QPS to implement the QoS requirements specified by the application. This interface consists of a collection of APIs that are available at the network-level.

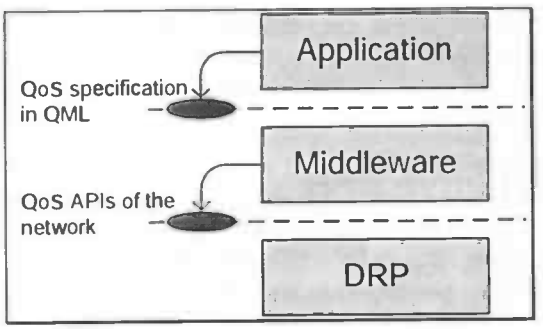

Figure 39: QPS QoS interfaces in the ODE

Can we add QoS support to CORBA in a portable manner?

We have built a prototype of the QoS Provisioning Service. The prototype is implemented in C++. We have chosen ORBacus, an open-source CORBA implementation, as our middleware component. The QPS is designed to be source portable between different CORBA implementations. It should be possible to use QPS with another CORBA implementation without having to change the source of QPS or the source of the CORBA implementation. In order to make this possible QPS relies on two extension mechanisms that should be available in the CORBA implementation. These mechanisms are Portable Interceptors and Pluggable Protocols. Both mechanisms are currently not included as mandatory parts of the CORBA specification, but are on the way to become OMG standards.

Portable Interceptors are being standardized at the time of writing [41]. Not all interception points described in chapter 3 of this thesis are included in upcoming Portable Interceptors specification. However, interceptors at request-level are part of the specification. This interception point is critical to QPS, since this point is used to enforce QoS for method invocations.

There is reluctance in the OMG to standardize Pluggable Protocols. Standardization of Pluggable Protocols is important to services like QPS, in order to be ORB independent. Some ORB vendors do not want to give access to the mechanisms to transfer GIOP messages over the network, allowing third-party software to be ORB independent is not in their best interest. At least two open-source CORBA implementations (ORBacus and TAO) do support Pluggable Protocols however. Sadly, these two implementations do not use the same interfaces.

Given this lack of standardization of Portable Interceptors and foremost Pluggable Protocols, the parts of QPS that depend on this features are isolated from the rest of QPS in order to ease porting to another ORB implementation.

What QoS mechanisms does the network provide and how does the middleware benefit from these mechanisms?

The QPS uses QoS mechanisms available at the network-level to implement the QoS requirements specified by the application in QML. The network may provide various QoS mechanisms, for instance Diff-Serv, Int-Serv, and IP multicast. The network does not provide a generic API to access these mechanisms. Most mechanisms themselves don't even have a good API. The lack of a generic QoS API at the network-level makes it impossible to write a pluggable protocol for CORBA that leverages network QoS mechanisms to the ORB. We decided to implement a generic pluggable protocol that doesn't implement a transport type (like TCP/IP or UDP) and that also doesn't communicate directly with QoS APIs that may be available at the network-level. Instead, QPS supports loading of plug-ins at run-time (this is not the same as a pluggable protocol). This is depicted in the following figure.

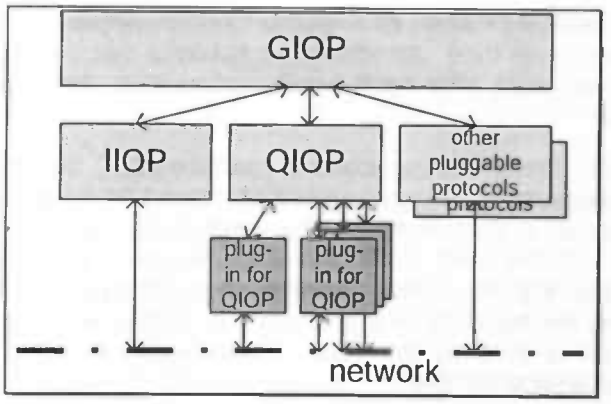

Figure 40: A high-level view of the QIOP protocol

Each plug-in is tailored towards one or more QoS mechanisms available at the network-level. This way QoS mechanisms available on the network can be used by QPS without having to integrate support for it in QIOP and even without having to restart the application that QPS provides QoS for. Instead, a plug-in that implements a simple interface expected by QIOP has to be constructed.

The Open Communications Interface (ORBacus' pluggable protocol support) doesn't have all the necessary features to support this. In chapter 5 we presented these shortcomings of OCI and proposed extensions. For the prototype we implemented an ad-hoc solution for these problems.

Future work:
The QPS is an initial implementation of a QoS control framework for CORBA, but it is far from finished. This section makes suggestions for further research, proposes work to enhance and extend the design of QPS, and lists shortcomings of the current prototype.

An important issue for further research is the performance of QPS. Performance should be measured and based on these measurements, suggestions for improvements in the design and implementation could be made. Critical points in the performance of QPS include the Portable Interceptors mechanisms, the Interceptors that QPS uses, and the overhead generated by the QIOP protocol.

Another issue that hasn't been explored is the suitability of QPS in a real-time environment. Is the behavior of QPS itself predictable? Can QPS be used to enforce hard real-time QoS requirements? QPS focuses on network level QoS, the current version isn't able to use other QoS mechanisms, like scheduling algorithms available in real-time ORB implementations.

Currently, it is not possible to use QIOP plug-ins as components of other QIOP plug-ins. This can be useful for building QIOP plug-ins that support group communication (multicast). Such a group communication plug-in could use other QIOP plug-ins to handle the actual communication. Management of these helper plug-ins could then be done by the group communication plug-in itself. Another example of when stackable QIOP plug-ins could be useful is supporting encryption. A separate QIOP plug-in could handle the actual encryption, and other QIOP plug-in could handle the communication over the network.

The QPS currently only measures QoS at the client-side of a method invocation and it isn't able to measure QoS at the remote-side. QIOP could be extended to allow for exchanging QoS measurements between client-side and server-side ORBs. For instance, QPS is currently not able to measure the delay of a method invocation, i.e. how long it takes to transfer the method invocation from the client ORB to the remote ORB. Only the round-trip delay can be measured, since measurement is done at the client side in this case.

The QPS doesn't provide configuration options to select the actions that should be taken if QoS degrades. It currently tries to adapt the QoS configuration, and throws an

exception to the application when all available QoS resources that are supported by the QIOP plug-ins have been tried. An alternative behavior that the application could desire is to receive an exception after each failed configuration, for example to allow for re-negotiation of QoS.

The changes to the Open Communications Interface (OCI) that we proposed could be designed and implemented. Pluggable protocols cannot be loaded into ORBacus at run-time, they're pluggable at compile-time. This requires that parts of QPS that are used by QIOP are linked with the ORB itself. Run-time pluggable protocols would allow more flexibility and cleaner implementation. Another shortcoming of OCI is that doesn't specify anything regarding the handling of I/O events. In ORBacus the event-handling of OCI pluggable protocols is done by the ORB. Implementing custom event-handling is not possible without changing the ORB.

Finally, the prototype doesn't implement the QoS repository. A QML QoS repository could be implemented based on the design in chapter 4, including a QML parser and compiler.

7 References

[1] D.C.Schmidt, The ADAPTIVE Communication Environment,1994.

[2] Object Management Group, Description of New OMA Reference Model, Draft 1, 1996. OMG Document ab/96-05-02.

[3] Object Management Group, The Common Object Request Broker: Architecture and Specification 2.2,1998.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1995. Addison-Wesley. ISBN 0-201-63361-2.

[5] D.C.Schmidt, A.Gokhale, T.H.Harrison, and G.Parulkar, A High-performance Endsystem Architecture for Real-time CORBA, 1997.

[6] Douglas E.Comer, Internetworking with TCP/IP Volume 1: Principles, Protocols and Architecture, 1995. Prentice-Hall International (UK) Limited, London. ISBN 0-13-216987-8.

[7] Object Management Group, Quality of Service Green Paper, 1997. OMG Document 97-12-06 version 0.4a, June 1997.

[8] ISO/IEC, Information Technology - Quality of Service - Framework, 1997. ISO/IEC Document JCT1/SC21 N13236.

[9] Jan de Meer and Abdelhakim Hafid, The Enterprise of QoS, 1998.

[10] S.Frolund and J.Koistinen, Quality of Service specification in Distributed Object Systems 5, 1998.

[11] Object Management Group, UML Semantics version 1.1, 1997. OMG Document ad/97-08-04.

[12] Object Management Group, UML Notation Guide version 1.1, 1997. OMG Document ad/97-08-05.

[13] John A.Zinky, David E.Bakken, and Richard Schantz, Architectural Support for Quality of Service for CORBA Objects 3, 1997.

[14] Object Management Group, CORBA Messaging Joint Revised Submission, 1998. OMG Document orbos/98-05-05.

[15] Object Management Group, Realtime CORBA 1.0 Joint Submission, 1998. OMG Document orbos/98-12-05.

[16] IETF DiffServ working group, Differentiated Services (diffserv) charter http://www.ietf.org/html.charters/diffserv-charter.html

[17] K.Nichols, S.Blake, F.Black, and D.Black, RFC 2474: Definition of the Differentiated Services Field (DS field) in the IPv4 and IPv6 Headers, 1998.

[18] D.Black, S.Blake, M.Carlson, E.Davies, Z.Wang, and W.Weiss, RFC 2475: An Architecture for Differentiated Services

[19] K.Nichols, V.Jacobson, Cisco, L.Zhang, and UCLA, A Two-bit Differentiated Services Architecture for the Internet (draft), 1999.

[20] V.Jacobson, K.Nichols, Cisco, K.Poduri, and Bay Networks, RFC 2598: An Expedited Forwarding PHB, 1999.

[21] R.Braden, ISI, D.Clark, MIT, S.Schenker, and Xerox PARC, RFC 1633: Integrated Services in the Internet Architecture: an Overview, 1994.

[22] R.Braden, L.Zhang, S.Berson, S.Herzog, and S.Jamin, RFC 2205: Resource ReSerVation Protocol (RSVP), 1997. Information Sciences Institute. http://info.internet.isi.edu/in-notes/rfc/files/rfc2205.txt

[23] IETF IntServ working group, Integrated Services (intserv) charter http://www.ietf.org/html.charters/intserv-charter.html

[24] Andrew S.Tanenbaum, Computer Networks 3rd edition 1996. Prentice Hall Professional Technical Reference. ISBN 0-13-349945-6.

[25] J.C.R.Bennett and H.Zhang, Hierarchical Packet Fair Queueing Algorithms 5, 1997.

[26] Sally Floyd, Notes on CBQ and Guaranteed Service (Draft), 1995. http://www.aciri.org/floyd/cbq.html

[27] Object Management Group, Portable Interceptor RFP, 1998. OMG Document orbos/98-09-11.

[28] Washington University, TAO - The ACE ORB. http://www.cs.wustl.edu/~schmidt/TAO.html

[29] F.Kuhns, C.O'Ryan, D.C.Schmidt, and J.Parsons, The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware, 1999. http://www.cs.wustl.edu/~schmidt/PDF/pluggable_protocols.pdf.gz

[30] J.Postel, Transmission Control Protocol - DARPA Internet Program Protocol RFC 793 1981. Information Sciences Institute. http://www.ietf.org/rfc/rfc0793.txt

[31] J.Postel, User Datagram Protocol RFC 968 1980. Information Sciences Institute. http://www.ietf.org/rfc/rfc0968.txt

[32] A.T.van Halteren, A.Noutash, L.J.M.Nieuwenhuis, and M.Wegdam, Extending CORBA with specialised protocols for QoS provisioning, 1999.

[33] C.O'Ryan, F.Kuhns, D.C.Schmidt, and J.Parsons, Applying Patterns to Design a High-performance, Real-time Pluggable Protocols Framework for OO Communication Middleware, 1999. http://www.cs.wustl.edu/~schmidt/pluggable_protocols.ps.gz

[34] Object Oriented Concepts, ORBacus. http://www.ooc.com/ob/

[35] Object Management Group, Revised version of the AT&T/TelTec/GMD Fokus IN/CORBA submission, 1998. OMG Document telecom/98-06-03.

[36] D.C.Schmidt, Acceptor and Connector: Design Patterns for Initializing Communication Services, 1997. Addison-Wesley. Reading, MA.

[37] S.Frolund and J.Koistinen, QML: A Language for Quality of Service Specification, 1998. http://www.hpl.hp.com/techreports/98/HPL-98-10.html

[38] S.Frolund and J.Koistinen, Quality of Service Aware Distributed Object Systems, 1999. http://www.hpl.hp.com/techreports/98/HPL-98-142.pdf

[39] Linus Torvalds, Alan Cox, David S. Miller, Donald Becker, Stephen Tweedie, Remy Card et al, The Linux operating system. http://www.linux.com/

[40] Open Source Initiative, Open Source website, 1999. http://www.opensource.org/.

[41] Expersoft, GMD Fokus, Objective Interface Systems, Object Oriented Concepts, Adrion, Humboldt University Berlin, KPN Research, and Deutsche Telekom AG, Initial Submission Against The Portable Interceptors RFP, 1999. OMG document orbos/99-04-10.

[42] EURESCOM Project P910, Project Internal Report 3.3: Interceptors Architecture and Specification for P910, 1999.

[43] EURESCOM Project P910, Project Internal Report 3.5: Interceptors Implementation, 1999.