Date: October 2008

Quality of Service for CORBA Components Specification

Version 1.1

OMG Document Number: formal/2008-10-02
URL: http://www.omg.org/spec/QOSCCM/1.1/PDF/
Associated File(s)* : http://www.omg.org/spec/QOSCCM/20080601

Source document: QoS for CORBA Components, Beta 1 (ptc/2008-06-04)

* Original file: IDL (ptc/2008-06-11)

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

LICENSES

commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 140 Kendrick Street, Needham, MA 02494, U.S.A.

TRADEMARKS

MDA®, Model Driven Architecture®, UML®, UML Cube logo®, OMG Logo®, CORBA® and XMI® are registered trademarks of the Object Management Group, Inc., and Object Management Group™, OMG™ , Unified Modeling Language™, Model Driven Architecture Logo™, Model Driven Architecture Diagram™, CORBA logos™, XMI Logo™, CWM™, CWM Logo™, IIOP™ , MOF™ , OMG Interface Definition Language (IDL)™ , and OMG SysML™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.htm).

Table of Contents

Preface

About the Object Management Group

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *http://www.omg.org*

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. A catalog of all OMG Specifications is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications

- UML
- MOF
- XMI
- CWM
- Profile specifications

OMG Middleware Specifications

- CORBA/IIOP
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM)

Platform Specific Model and Interface Specifications

- CORBAservices
- CORBAfacilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security specifications

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
140 Kendrick Street
Building A, Suite 300
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult *http://www.iso.org*

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

Courier - 10 pt. Bold: Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

Note – Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to *http://www.omg.org/technology/agreement.htm.*

1 Scope

1.1 Overview

Distributed systems, and especially highly distributed applications, are very dependent on many execution conditions: number of available computers, their location, quality of network connection, heterogeneous hardware, operating systems, and libraries to name only a few. Middleware like the CORBA Component Model is used to make such heterogeneity more transparent. But sometimes a distributed application cannot be written and executed ignoring the features of the underlying system. To keep both the transparency and the flexibility to build adaptable applications, the separation of functional code (i.e., business code) and non-functional code (i.e., container related code) is needed. This separation allows a developer or system administrator to easily adapt an application to particular execution conditions or to change the Quality of Service (QoS) of the application without rewriting the application.

However, not all QoS properties are non-functional; some are strongly linked to the service that a component performs. An example is an audio encoding algorithm whose results should match certain quality criteria, e.g., a maximum data rate. The component implementation, i.e., the business code written by the developer, needs to be aware of the values of these properties.

Separation of functional and non-functional aspects is necessary but not sufficient to allow efficient and flexible adaptation. A component platform has to provide an extensible and flexible architecture for an easy integration of those two separated aspects (functional and non-functional properties) at deployment and run-time.

Dynamic configuration and re-configuration is also an essential point of distributed systems, because in order to adapt an application to a specific execution environment, one has to be aware that the QoS of the execution environment evolves during execution. There is a strong need to be able to make on-the-fly adaptation. That is why it is very important to provide an architecture, mechanisms, and monitoring concepts that allow dynamic adaptation of a running application, explicitly by an administrator, but also automatically. The specification addresses this problem domain in the context of the CORBA Component Model (CCM) [CCM].

The CORBA Component Model supports the separation of functional and non-functional properties for a fixed set of such properties already. Transactions, Persistency, and Security are part of this fixed set as well as component co-locations properties. Some of these properties are completely independent of the component implementation while other properties can be managed by the component implementation by using standardized interfaces. The specification defines concepts that keep this approach, by allowing the injection of non-functional aspects completely transparent to component implementations and gives the component implementation the possibility to manage such properties.

The specification describes the essential extensions that need to be made to the container architecture to allow the support of separation of functional and non-functional properties of CORBA components.

The first aspect of QoS properties within the container is their configuration or – in dynamic environments – their negotiation. The second aspect is the enforcement and realization of the non-functional properties. This specification will deal with three different realization mechanisms:

1. The injection of non-functional properties into the container by means of interception. Container plugins defined in this specification, named QoSEnablers, have the responsibility to provide such interceptors if the QoS pattern they implement need to be based on them. A major part of this document deals with their specification.

2. The configuration of the underlying middleware, for instance the use of certain policies of the portable object adapter, or object references, typically in order for QoS properties to be propagated in a suitable way.

3. The configuration of the underlying transport. This mechanism is relevant, since QoS does not only apply to single components, but to connections between components as well.

The goal of this specification is not only to support the consideration of typical QoS properties like Latency, Throughput, Bandwidth, etc. but also a very different kind of QoS properties (non-functional aspects) like reservation of computing power or constraining the frequency of event submission, which can be useful for saving battery power of a mobile device. Nevertheless, this specification concentrates on the introduction of container extensibility mechanisms in order to allow realization of such properties, and does not deal directly with them, since the needs are very disparate and sometimes application specific.

1.2 Quality of Service

This specification targets on modeling, realizing, and managing Quality of Service (QoS) properties. However, this term needs to be defined for this specification. Quality of Service properties of a component are properties that describe how a component deliver a service to other components. QoS properties do not define what functionality is provided but define how functionality is provided.

It is not easy to clearly draw a line between properties that describe the functionality of a component and properties that describe QoS of a component. In this specification, everything that describes a component with its ports is considered as the functionality. This means every port of a component with all the operation in the corresponding interface or events is considered to be the functionality of a component. Everything else is per definition not the functionality.

Moreover, the term Quality of Service sometimes is not really adequate, since some of the properties do not really describe a quality of a service but cannot be considered as functionality. For example, to enable a tracing that monitors calls between components is not really a quality of the service that the called component provides. For that reason such properties are often called non-functional properties or non-functional aspects instead of QoS properties. In some cases the term extra-functional properties is used as well. In this specification the terms QoS properties and non-functional properties or aspects are used synonymously.

2 Conformance

It is the intent of the specification to support a wide variety of use cases where handling of non-functional aspects can be important. Furthermore, some parts of the specification can be used to develop other container related services that cannot directly be seen as a non-functional or QoS aspect of a component.

Some realizations of non-functional aspects might not require all the parts of the described concepts of the specification to be present. It is also possible to use the Container Oriented Portable Interceptors defined by the specification to realize certain functionality but not the use the QoS Enabler concept to implement them. This is the reason why the specification defines two sets of compliance points by definition of two sets of compliance points.

2.1 Mandatory Compliance Points

A conformant CORBA Component implementation should at least support the following items:

- Container Portable Interceptors

2.2 Optional Compliance Points

A conformant CORBA Component implementation shall support the mandatory compliance points and can optionally support the following additional concepts:

- QoS Enabler

- Extension Container
- Negotiation Interface and Flow Details

3 References

3.1 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

[CCM] CORBA Components Specification, OMG TC Document formal/02-06-65

[CORBA] CORBA Specification, OMG; OMG document number formal/04-03-01

[MOF] Meta Object Facility (MOF) Specification, Version 1.4, OMG document ptc/2001-10-04

[UMLQOS] UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms, OMG Adopted Specification, OMG documents number formal/06-05-02

3.2 Non-normative References

[QEDO] Qedo, QoS Enabled Distributed Objects, an Open Source CCM implementation, http://www.qedo.org

[UMLCCM] UML Profile for CORBA and CORBA Components, Initial Submission, OMG document number mars/2005-11-05

[COACH] COACH Project Home Page, http://www.ist-coach.org, IST Program. Project IST-2001-34445.
1 April 2002 to 31 March 2004

4 Terms and Definitions

For the purposes of this specification, the terms and definitions given in the normative reference and the following apply.

Container Portable Interceptor

An interface at container level that allows the interception of calls at client and server side.

5 Symbols (and abbreviated terms)

CCM	CORBA Component Model
COPI	Container Portable Interceptor
QoS	Quality of Service

6 Additional Information

6.1 Changes to Adopted OMG Specifications

The general principle of this specification is to define only extensions to existing specifications. The concepts defined are targeted on the CORBA Component Model (CCM) and enable the management of QoS properties in a very flexible and general way. The extensions to the CCM specification that are implied by this specification are listed below. No other OMG specification is affected.

- This specification defines the extension of the CCM metamodel by introducing the new package CCMQoS as it is explained in Chapter 7.
- This specification defines an extension to the number of predefined container types as part of the full compliance profile. The container type is named Extension. This container type is described in detail in Section 8.9, "Extension Container," on page 43.
- Extension of the context interface to retrieve reference to container services.
- Extending the enumeration Components::CCMExceptionReason by the values: QOS_ERROR, REGISTRATION_ERROR, SERVICE_INSTALLATION_ERROR.
- Extend the component category by new type extension. This is used to denote components such as QoSEnablers that extend the run-time environment of CORBA Components.
- Extend the Components::StandardConfigurator interface to get component configuration.
- Extend the Components::HomeConfiguration interface to retrieve reference to Configurator.

6.2 How to Read this Specification

The rest of this document contains the technical content of this specification. As background for this specification, readers are encouraged to first read the *CORBA* and the *CORBA Component Model* specifications that complement this. Furthermore, the *TheUML Profile for Quality of Service and Fault Tolerance Specification* explains the concepts and the language how to model QoS properties in general.

6.3 Acknowledgements

The following companies submitted and/or supported parts of this specification:

- Fraunhofer FOKUS
- CEA
- THALES
- ObjectSecurity Ltd.
- Deutsche Telekom / T-Systems

7 Modeling of QoS for CORBA Components

7.1 Scope of QoS Properties

Not all QoS properties are valid for the whole application lifetime. Some are bound to the lifetime of a connection instance; others are to be applied specifically for each call. The following list shows the possible entities to which a QoS property may be bound:

1. Component (type)
2. Group of component instances (Managed by one Home)
3. Component Instance
4. Connection (type)
5. Call

If a QoS property is bound to a component or connection type, only one value is configured for all possible component instances. The configuration of a QoS property related to the connection type corresponds to a global transport configuration, e.g., a general CORBA policy.

If a QoS property is bound to a component or connection instance, its actual scope depends on their lifetime. If a component instance is already declared during the assembly and never deleted until application shutdown, its lifetime – and that of the QoS property as well – is identical with that of the application. In this case, the value of the QoS property may be configured during the deployment and configuration phase. However, if the component is created dynamically or configured at runtime, the QoS value needs to be configured either via application specific means or by using configurator. The latter mechanism is modified by this specification, see Section 8.12, “Dynamic Adaptation of a Running Application,” on page 50.

In case of dynamically created connections, the property might be configured either via application specific means or negotiation. The latter mechanism is standardized by this specification, see Section 8.8, “Negotiation,” on page 36 for details.

An example of a QoS property that is bound to a component type is a specific synchronization mechanism that is needed to avoid race conditions within the component implementation.

In many cases, the implementation of a component does not need to know which QoS properties are configured: the properties are managed by the container and it is an essential benefit that the implementation can be done being aware of these. But in some cases, the implementation needs to access the value of the QoS property, imagine an encoder that should process data trading CPU usage for quality. Therefore, the access to the QoS property needs to be specified through an API.

7.2 CCMQoS Metamodel

The modeling of non-functional properties such as QoS properties requires clearly defined modeling concepts. Such concepts are defined in a platform independent way in the specification “UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms” [UMLQOS]. Chapter 8 of that specification defines a comprehensive metamodel for the description of QoS properties.

The CCM specification [CCM] contains the definition of a metamodel for CCM. This metamodel contains different packages. The BaseIDL package contains basic IDL concepts e.g., for defining interfaces. The ComponentIDL package contains concepts needed for the definition of component types. The CIF package contains concepts needed for the definition of component implementations.

However, the modeling of QoS properties for CORBA Components requires the definition of a link between QoS metamodel and CCM metamodel. This link is defined in the metamodel package CCMQoS. Due to the fact that definition of QoS properties for CORBA Components may have different scopes different links between the metamodels needs to be defined.

The CCM metamodel currently does not contain concepts for defining component instances and component assemblies. This part of the metamodel has been removed, with the goal to define a new and consistent Deployment and Configuration metamodel. A first draft of such a metamodel is presented in the initial submission to the UML2 Profile for CORBA and CORBA Components RFP [UMLCCM].

7.2.1 Package Structure

The link between the QoS metamodel and the CCM metamodel is described in a new package that depends on the QoSConstraint package of the QoS metamodel and on the ComponentIDL package from the CCM metamodel. The overall package structure is shown in Figure 7.1.

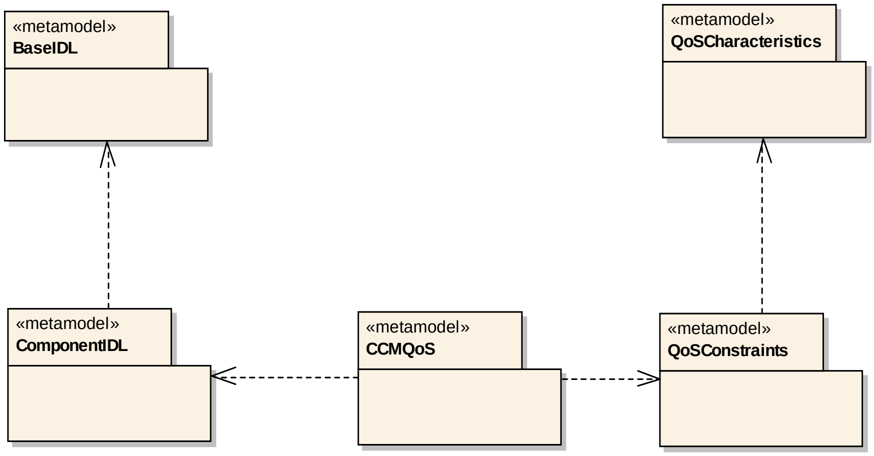

Figure 7.1 - Package Structure

7.2.2 Binding

The CCMQoS package contains definitions to link QoS properties to CORBA Components. This is achieved by defining a Binding metaclass. A binding metaclass correlates a QoS constraint (QoSConstraint) with a component feature definition (ComponentFeature). With this it is possible to describe that a specific QoS constraint is relevant in the context of a specific component type. A ComponentFeature can be any of the ports of a component type or the component type itself. (Figure 7.2)

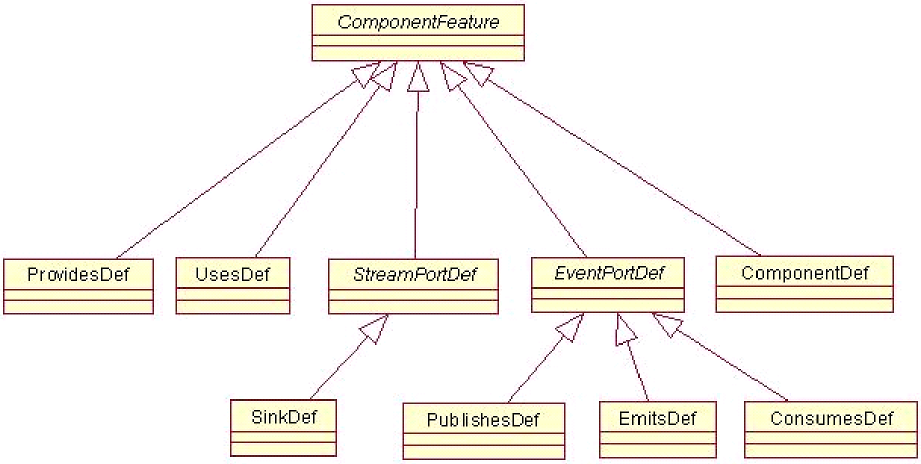

Figure 7.2 - ComponentFeature (from CCM Metamodel)

Binding a QoSContext to a ComponentFeature makes this QoS property relevant for the component type. This means all instances of this component type are related to that QosContext.

The definition of the Binding metaclass is depicted in Figure 7.3.

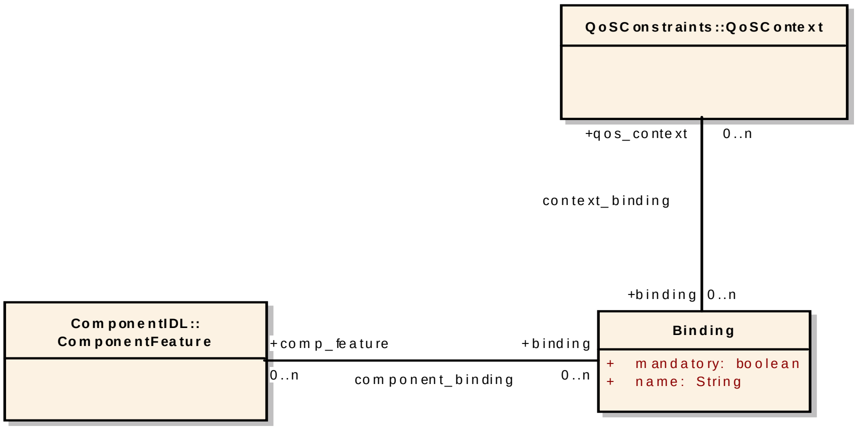

Figure 7.3 - Binding

7.2.2.1 mandatory

This attribute is of type boolean. If it is set to true, the Binding is mandatory. Which means the QoS property bound in any case. If this attribute is set to false, the binding needs not to be valid all the time. This can be useful for systems in phases where fewer resources are available and even the handling of QoS bindings would imply major decrease of system performance. QoS bindings that are not mandatory can be temporarily disabled.

7.2.2.2 name

This attribute is of type string and is used to give the binding a name.

7.3 Notation for QoS

The notation for QoS properties is completely based on the notation defined by UML profile for QoS and the UML Profile for CCM. Furthermore, the upcoming UML2 Profile for CORBA and CORBA Components [UMLCCM] is going to replace the Profile for UML 1.x. Examples on how to use the notations are given in Annex B.

8 Container Architecture

8.1 Introduction

This specification defines concepts and interfaces that allow the integration of non-functional aspects and CCM based applications. The goal is the separation of the realization of such non-functional aspects from the business code as much as possible.

The component server is the run-time environment of a component. The component server is extended by containers that are a more dynamic part of the run-time environment of a component. Containers can be created inside the component server or can be destroyed. From a logical perspective components are created inside a container. Component server and container together dispatch calls, manage lifecycle of components. Although the term Container Portable Interceptors does not directly cover this, it is important to mention that Containers are dependent on the Components Server to provide their functionality.

The injection of non-functional aspects according to the above principle implies the need for some extensions of the container architecture and the definition of APIs between container, component implementations, and QoS related entities. Those entities can be realized by using the QoS Enablers. QoS Enablers are similar to components and are executed in a specialized container. This specification does not require the usage of the QoS Enabler concept to integrate non-functional properties into the container.

In some situations the separation of non-functional aspects from business code will not be possible to the full extent (because QoS logic has sometimes to be intimately related to component logic). In such cases clear interfaces between the component implementation (Executor) and the run-time environment (Container) are defined. This helps making functional and non-functional properties as independent as possible.

For managing QoS properties for CORBA components, it is not sufficient only to extend the container architecture. It is also important to define means for managing QoS properties from outside of the container. This includes the connection between components as well as the deployment of components.

The general container architecture of the CORBA Component Model is explained in the specification of the CORBA Component Model [CCM]. Figure 8.1 is in accordance to the corresponding figure of the CCM specification and contains the general idea of the extension of the container architecture by introducing the Container Portable Interceptors (COPI). Container Portable Interceptors are the means by which Container extensions can effectively become part of the container.

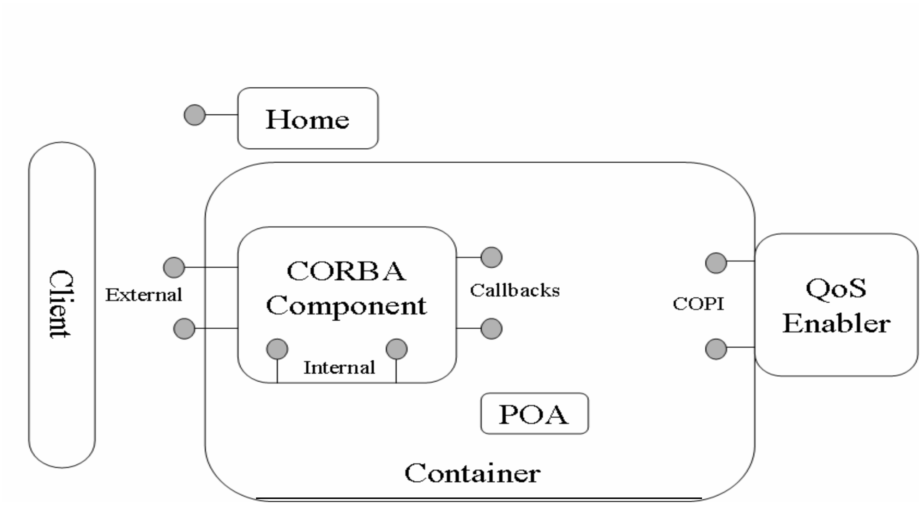

Figure 8.1 - Extended Container Architecture

8.1.1 Requirements

8.1.1.1 Support of CORBA Portable Interceptors

A basic principle, among others, for integration of non-functional properties to CORBA applications is the interception mechanism, which can be used to monitor and possibly change the processing of calls. This mechanism is defined by the specification of the CORBA Portable Interceptors [CORBA]. CORBA Portable Interceptors can be used for integrating CORBA services within CORBA based applications.

The CORBA Component Model emphasizes the clear separation of business logic implementation (Executor) and other run-time related artifacts (Container). This means that user provided code is essentially reduced to the core business logic. All other activities, which are common to plain CORBA based applications are now managed by Container and Component Server and are controlled by the deployment infrastructure. For that reason there is no standard way for accessing and using the CORBA Portable Interceptors from CORBA Components. One of the reasons is that Executors have no direct access to the ORB and cannot make use of **pre_init()** and **post_init()** methods that prevents them from registering interceptors at the ORB. The Container Portable Interceptors shall allow the injection interception mechanism as user provided code in a standard way.

An important addition to the CORBA Portable Interceptors is needed since not only an interface and its implementation is an identifiable entity in the CORBA Component Model but also the component implementation (executor). This means that the information provided in the various interception points has to be enhanced by information about Executor.

8.1.1.2 Migrating from CORBA to CCM

The CORBA Component Model is based on plain CORBA and allows an easy migration from plain CORBA based applications to CCM based applications. The same principle shall be applied to the Interception mechanism of CORBA. The Container Portable Interceptors shall support an easy migration from user code based CORBA Portable Interceptor to user code based on Container Portable Interceptors. This may lead to definition of similar or same API as well as same or similar flow rules.

8.1.1.3 Simple Management of User Code

Since the implementation for doing the actual interception is provided by user and not by container vendor, it is important that handling of such code is as simple as it is to handle component implementations. The COPI specification shall allow easy implementation and management of Container Portable Interceptor code.

8.2 Component Instance Identity

The identity of a component instance is important, in particular associating non-functional properties with them. On the basis of this identity the container is able to associate such properties to an actual call dispatched by it. In contrast to traditional CORBA systems, two important identities are relevant. One is the identity of the client component instance that calls a method on a server component instance and the identity of the server component instance itself.

Currently, the handling of component instance identifiers is not supported by CCM. For the reliable handling of non-functional aspects on the basis of the contract concept a way to identify client and server is needed. This is achieved by associating each component instance with an identifier at instance creation time. The identifier needs to be unique.

In order to achieve this, two different cases have to be distinguished. In case of a static component setup, as useful for rather small embedded systems, a unique identifier can easily be achieved: all component instances are known at compile time and can be associated with a unique instance. In case of dynamic systems in which new component instances might show up, the situation is different. In this situation it is important to identify each component instance with a unique identifier. If this component instance identifier is chosen out of a sufficiently large value space it is very unlikely that a client “guesses” a client identifier belonging to another client. Depending on the desired improbability identifiers could be sequence of octet. In the context of this specification the possibility that one client steals the identity of another by means of spying out the network traffic is not addressed (if this is an issue, encrypted transports need to be used).

To identify a client component instance at server side it is vital that the identity information is transmitted in every call a client makes to give the server side the possibility to check for active contracts for the calling component instance. If a client does not submit its identity, an inter-component QoS agreement cannot be applied.

The container, i.e., the run-time environment of a component, is responsible for managing the association of the identifier and the component instance. Furthermore, the transmission of component instance identities should be completely transparent to the component instances itself. Therefore it has to be handled by the container. The container shall use the Service Context defined for plain Portable Interceptors (PI) to achieve this. This allows the integration of this functionality by standard means, which will in fact keep interoperability at CORBA level.

8.3 Container Portable Interceptors

8.3.1 Introduction

Container Portable Interceptors (COPI) are hooks into the Container Architecture through which the normal flow of execution inside the Container can be intercepted. In general, the Container Portable Interceptors are built upon the principles of the CORBA Portable Interceptors, but modifications are applied to take the container architecture and the component setting into account.

There are two levels of Container Portable Interceptors. The basic level corresponds to the capabilities of the CORBA Portable Interceptors (PI) and the extended ones provide an extended functionality to better control the call chain within the Container.

Container Portable Interceptors support similar programming models as Portable Interceptors do, namely:

1. Client sendsrequest
2. Server receives request
3. Server sends reply
4. Client sends reply

A detailed description on how to implement the basic and the extended COPIs will not be given in this specification. This will leave container vendors the freedom to decide on how to implement this specification. However, a possible implementation variant is that basic COPIs will be realized by wrapping the CORBA Portable Interceptor as part of the container implementation, which means that the basic interception points will be called by a wrapped Portable Interceptor. The extended COPIs can be realized by extending the container implementation and make the calls to the extended interceptions points directly from the container.

8.3.2 Design Principles

The following points are the principles followed in the design of the basic Container Portable Interceptor architecture.

1. Interceptors are called on all container mediated calls that are directed to components. Interceptors are not called for operations for management of run-time (e.g., install_home). Due to the fact that implicit operations (i.e., **get_interface**, **is_a, non_existent**, **get_domain_managers**, and **get_component**) may or may not be ORB mediated, Basic Container Interceptors may or may not be called. Whenever the implicit operations are ORB mediated, interceptors are called; otherwise, they are not called.

2. A basic Container Interceptor can affect the outcome of a request by raising a system exception at any of the interception points. It can stop the request from even reaching the target by raising a system exception.

3. A basic Container Interceptor can affect the outcome of a request by directing a request to a different location at any interception point other than a successful reply.

4. A basic Container Interceptor cannot affect a request by changing a parameter specified by a client. That is, the Basic Container Interceptor cannot modify “in” arguments.

5. A basic Container Interceptor cannot affect a non-exception outcome by supplying the response itself. That is, the basic Container Interceptor cannot modify “out” arguments or the return value.

6. Basic Container Interceptors are independent of other Container Interceptors. That is, a Container Interceptor won’t need to know, and won’t even be told, if there are Container Interceptors executed before or after it.

7. A basic Container Interceptor may make object invocation itself before allowing the current request to execute.

8. There is no provision of making component implementation aware that any Container Interceptor is called.

9. The basic Container Interceptors do not bypass the dispatching of the request within the container.

The following points are the principles followed in the design of the extended Container Portable Interceptor architecture:

10. Basic Container Portable Interceptors and Extended Container Portable Interceptors can be used in parallel. They work independently of each other. From the perspective of basic Container Portable Interceptor, the Extended Portable Interceptors are similar to plain user code. Corollary: Basic and Extended Container Portable Interceptors can communicate between themselves to bypass this principle.

11. Even if the interception points are quite similar to the ones of the basic Container Interceptors different names are used to provide clear separation of the different interception points.

12. A set of general flow rules governs the flow of processing.

8.3.3 General Flow Rules

Container Interceptors are registered with the Component Server. The Component Server logically maintains the order of the list of Container Interceptors.

To accommodate both Client and Server Container Portable Interceptors a set of general flow rules are defined. These flow rules are exactly the same rules as the general flow rules defined for portable request interceptors (section 21.3.2 in CORBA specification [CORBA]), with the following addition:

- Basic and extended Container Interceptors can be used in parallel.
- On client side, starting points of Extended Container Portable Interceptors are called before starting points of Basic Container Portable Interceptors. Ending points of Extended Container Portable Interceptors are called after ending points of Basic Container Portable Interceptors.
- On server side starting points of Basic Container Portable Interceptors are called before starting points of Extended Container Portable Interceptors. Ending points of Basic Container Portable Interceptors are called after ending points of Extended Container Portable Interceptors.

8.3.4 Container Interceptor Interface

All Container Portable Interceptor Interfaces are defined in the module ContainerPortableInterceptor, which is defined in the module Components. All Container Portable Interceptors inherit from the local interface ContainerInterceptor.

The following IDL fragment describes the ContainerInterceptor interface.

```
module Components {
   module ContainerPortableInterceptor {

    struct CustomSlotItem
      {
         string identifier;
         any content;
      };

   typedef sequence<CustomSlotItem> CustomSlotItemSeq;
   struct IntegrationPoint
      {
         string port;
         string operation;
      };

      local interface ContainerInterceptor
      {
         readonly attribute string name;

         attribute unsigned short priority;
         attribute IntegrationPoint registration_info;
         void
         destroy ();
```

```
            void
            set_slot_id(in PortableInterceptor::SlotId slot_id);
        };
    };
};
```

8.3.4.1 name

Each container interceptor may have a name that may be used for administrative purposes. It is possible to use anonymous interceptors. In this case the name is an empty string. There can be more than one anonymous interceptor.

8.3.4.2 priority

Each container interceptor may have a priority that may be used to order the list of registered interceptors. Priority value may help implementor to control execution order of interceptors and some existing dependencies between interceptors. The determination and management of priority values is free.

8.3.4.3 registration_info

This attribute is a structure containing a port and an operation string that informs about the specific intercepted call. If null strings are set, interceptor is registered for all interception points.

8.3.4.4 destroy

This operation can be used by the container to destroy the container interceptor interface to free resources (e.g., when the container is destroyed). The semantics of this operation depends on how the container interceptor is implemented. It might be the case that the container interceptor is implemented by QoS Enabler, which itself will control the lifecycle of the container interceptor interface. In such a case the destroy operation has no other effect than to inform that the container interceptor interface is no longer used and subsequently no interception points will be called.

8.3.4.5 set_slot_id

This operation is used by the container to set the identifier of the slot that is used by the container in the **PortableInterceptor::Current** to handle thread specific information. Container interceptors have to use the slot identified by this id to process call and thread context specific information.

The data in the slot contains **CustomSlotItemSeq** which is a sequence of **CustomSlotItem**. A **CustomSlotItem** is a struct that contains an identifier and a content of type any. A COPI can add a new **CustomSlotItem** at the end of the sequence contained in the slot. The identifier of a **CustomSlotItem** should denote the COPI name. Content could be any content provided as any. The **CustomSlotItemSeq** sequence contained in the slot of **PortableInterceptor::PICurrent** identified by this slot_id will be encoded by the container in the service context of the call and is exchanged between client-side and server-side COPIs. See Section 8.3.6 for details.

8.3.5 Stack Visual Model and Interception Points

Similar to the definition of the CORBA Portable Interceptors, the same Flow Stack visual model is applied. This means to visualize the general flow rules, think of each Container Interceptor as being put on a Flow Stack when a starting interception point completes successfully. An ending interception point is called for each Container Interceptor in the stack. If one of the interceptors raises an exception during the invocation of its starting interception point, only those Interceptors on the stack at that point will be popped and have an ending interception point called.

Although basic and extended Container Portable Interceptors are inherently independent at run-time, they share the same Flow Stack.

8.3.6 COPIServiceContext

The COPIServiceContext struct contains information about the component identifier of the component instances participating in a call. origin_id denotes the client side component instance id. target_id denotes the server side component instance id. When Container Portable Interceptors are used the context_data component of the ServiceContext shall contain a CDR encapsulation of the COPIServiceContext struct, which is defined below.

```
module IOP {
  const ServiceIdCOPI = 23;
};

module Components {
  module ContainerPortableInterceptor {
      struct COPIServiceContext
      {
         CORBA::OctetSeq origin_id;
         CORBA::OCtetSeq target_id;
         CustomSlotItemSeq slot_info;
      };
  };
};
```

8.3.6.1 origin_id

This identifies the client component instance id that is the originator of a call. Whenever a call is not issued by a component instance or the originator id cannot be determined for any reason the sequence should have a zero length.

8.3.6.2 target_id

This identifies the server component instance id that is the target of the call. Whenever the id of the target component instance cannot be determined for any reason the sequence should have a zero length.

8.3.6.3 slot_info

slot_info is used to transmit **a CustomSlotItemSeq** sequence between client side and server side Container Portable Interceptors. **CustomSlotItems** are provided by Container PortableInterceptors during the processing of a call. See Section 8.3.4.5 for details.

8.4 Basic Container Interceptors

Basic Container Interceptors are designed with the very same intension as the Request Interceptors of Portable Interceptor specification are defined for the integration of ORB services into the ORB.

A basic Container Interceptor is designed to intercept the flow of a request/reply sequence through the Component Server and the Container respectively at specific points so that container services and other container extensions such as QoS Enablers can query the request information and manipulate the service context that are propagated between clients and servers.

The primary use of basic Container Interceptors is to enable ORB services and other artifacts that might be used to ensure a certain level of QoS to transfer context information between client and servers.

There are two types of basic Container Interceptors: client-side (8.4.2, ClientContainerInterceptor Interface) and server-side (8.4.5, ServerContainerInterceptor Interface).

A set of Design Principles apply to the Container Portable Interceptors that are very similar to the ones applied to the Portable Interceptors.

8.4.1 Basic Interception Points

Each basic Container Portable Interceptor is called at a number of interception points. Figure 8.2 shows the interception points that might be called in a request reply cycle. The details of the client-side interception points are described in 8.4.3, Client-Side Interception Points. The details of server-side interception points are described in 8.4.6, Server-Side Interception Points.

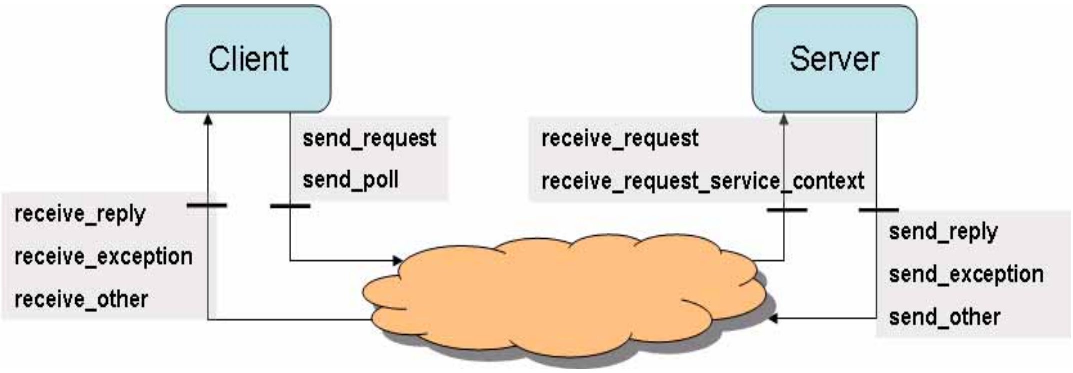

Figure 8.2 - Basic Interception Points

8.4.2 ClientContainerInterceptor Interface

The following IDL fragment describes the ClientContainerInterceptor interface. The operations (interception points) defined in this interface correspond to the operations defined in the ClientRequestInterceptor interface. The important difference between the operations of those two interfaces is related to the request info object that is passed as parameter.

To write a basic client-side Container Portable Interceptor this interface needs to be implemented.

```
module Components {
   module ContainerPortableInterceptor {
      local interface ClientContainerInterceptor  : ContainerInterceptor
      {
      void
         send_request (in ContainerClientRequestInfo info)
         raises (PortableInterceptor::ForwardRequest);

         void
         send_poll (in ContainerClientRequestInfo info);

         void
         receive_reply (in ContainerClientRequestInfo info);

         void
         receive_exception (
            in ContainerClientRequestInfo info)
         raises (PortableInterceptor::ForwardRequest);
```

```
            void
            receive_other (in ContainerClientRequestInfo info)
            raises (PortableInterceptor::ForwardRequest);
        };
    };
};
```

8.4.3 Client-Side Interception Points

8.4.3.1 send_request

This interception point allows an Interceptor to query request information and modify the service context before the request is sent to the server component.

This interception point may raise a system exception. If it does, no other Interceptors' **send_request** operations are called. Those Interceptors on the Flow Stack are popped and their **receive_exception** interception points are called.

The interception point may also raise a ForwardRequest Exception. If an Interceptor raises this exception, no other Interceptors' **send_request** operations are called. Those Interceptors on the Flow Stack are popped and their **receive_other** interception point is called.

Compliant Interceptors shall properly follow completion_status semantics if they raise a system exception from this interception point. The **completion_status** shall be COMPLETED_NO.

8.4.3.2 send_poll

This interception point allows an Interceptor to query information during Time-Independent Invocation (TII) polling get reply sequence.

With TII, an application may poll for a response to a request sent previously by the polling client component or some other client. This poll is reported to the Interceptors, through the **send_poll** interception point and the response is returned through the **receive_reply** or **receive_exception** interception points. If the response is not available before the poll time-out expires, the system exception TIMEOUT is raised and the **receive_exception** is called with this exception.

This interception point may raise a system exception. If it does, no other Interceptors' **send_poll** operations are called. Those Interceptors on the Flow Stack are popped and their **receive_exception** interception points are called.

Compliant Interceptors shall properly follow completion_status semantics if they raise a system exception from this interception point. The **completion_status** shall be COMPLETED_NO.

8.4.3.3 receive_reply

This interception point allows an interceptor to query the information on a reply after it is returned from the server component and before control is returned to the client component.

This interception point may raise a system exception. If it does so, no other Interceptors' **receive_reply** operations are called. The remaining Interceptors in the Flow Stack shall have their **receive_exception** interception point called.

Compliant Interceptors shall properly follow completion_status sematics if they raise a system exception from this interception point. The **completion_status** shall be COMPLETED_YES.

8.4.3.4 receive_exception

When an exception occurs, this interception point is called. It allows an Interceptor to query the exception's information before it is raised to the client component.

This interception point may raise a system exception. This has the effect of changing the exception, which successive Interceptors popped from the Flow Stack receive on their calls to **receive_exception**. The exception raised to the client component will be the last exception raised by an Interceptor, or the original exception if no Interceptor changes the exception.

This interception point may raise also a ForwardRequest exception. If an Interceptor raises this exception, no other Interceptors' **receive_exception** operations are called. The remaining Interceptors in the Flow Stack are popped and have their **receive_other** interception point called.

If the **completion_status** of the exception is not COMPLETED_NO, then it is inappropriate for this interception point to raise a ForwardRequest exception. The request's at most-once semantics would be lost.

Compliant Interceptors shall properly follow completion_status semantics if they raise a system exception from this interception point. If the original exception is a system exception, the **completion_status** of the new exception shall be the same as on the original. If the original exception is a user exception, then the completion_status of the new exception shall be COMPLETED_YES.

Under some conditions, depending on what policies are in effect, an exception (such as COMM_FAILURE) may result in a retry of the request. While this retry is a new request with respect to Interceptors, there is one point of correlation between the original request and the retry: because control has not returned to the client component, the **PortableInterceptor::Current** for both the original request and the retrying request is the same.

8.4.3.5 receive_other

This interception point allows an Interceptor to query information available when a request results in something other than a normal reply or an exception. For example, a request could result in a retry (for example, a GIOP Reply with a LOCATION_FORWARD status was received.); or on asynchronous calls, the reply does not immediately follow the request, but the control shall return to the client component and an ending interception point shall be called.

For retries, depending on the policies in effect, a new request may or may not follow when a retry has been indicated. If a new request does follow, while this request is a new request with respect to Interceptors, there is one point of correlation between the original request and the retry. Because control has not returned to the client component, the request scoped **PortableInterceptor::Current** for both the original request and the retrying request is the same.

This interception point may raise a system exception. If it does, no other Interceptors' **receive_other** operations are called. The remaining Interceptors in the Flow Stack are popped and have their **receive_exception** interception point called.

This interception point may also raise a ForwardRequest exception. If an Interceptor raises this exception, successive Interceptors' **receive_other** operations are called with the new information provided by the ForwardRequest exception.

Compliant Interceptors shall properly follow completion_status semantics if they raise a system exception from this interception point. The **completion_status** shall be COMPLETED_NO. If the target invocation had completed, this interception point would not be called.

8.4.4 Interception Flow for ClientContainerInterceptors

Instances of the ClientContainerInterceptor Interface are registered with the run-time environment. The run-time environment logically maintains an ordered list of client-side Container Interceptors. The interceptor list is traversed in order on the sending interception points and in reverse order on the receiving interception points.

8.4.4.1 Client-side Flow Rules

The client-side flow rules for basic Container Portable interceptors are derived from the general flow rules.

- The set of starting interception points is: **send_request** and **send_poll**. One and only one of these is called on any given request/reply sequence.
- The set of ending interception points is: **receive_reply**, **receive_exception**, **receive_other**. One and only one of these is called on any given request/reply sequence.
- There are no intermediate exception points.
- If and only if **send_request** or **send_poll** runs to completion is an ending interception point called.

8.4.4.2 Additional Client-side Details

If, during request processing, a request is canceled because of an ORB shutdown, which is caused by component server shutdown, **receive_exception** is called with the system exception BAD_INV_ORDER with a minor code of 4 (ORB has shutdown).

If a request is canceled for any other reason (for example, a GIOP cancel message is sent by the ORB), **receive_exception** is called with the system exception TRANSIENT with a standard minor code of 2.

On oneway requests, returning control to the client component may occur immediately or it may return after the target has performed the operation, or somewhere in-between depending on the SyncScope. Regardless of the SyncScope, if there is no exception, **receive_other** is called before control is returned to the client component.

Asynchronous requests are simply two separate requests. The first request receives no reply. The second receives a normal reply. So the normal (no exceptions) flow is: first request - **send_request** followed by **receive_other**; second request - **send_request** followed by **receive_reply**.

8.4.5 ServerContainerInterceptor Interface

The following IDL fragment describes the **ServerContainerInterceptor** interface. The operations defined in this interface correspond to the operations defined in the **ServerRequestInterceptor** interface. The main difference between those operations is the request info object.

To write a basic server-side Container Portable Interceptor this interface needs to be implemented.

```
module Components {
   module ContainerPortableInterceptor {
      local interface ServerContainerInterceptor : ContainerInterceptor
      {
         void
         receive_request_service_contexts (
            in ContainerServerRequestInfo csi )
         raises (PortableInterceptor::ForwardRequest);
```

```
            void
            receive_request ( in ContainerServerRequestInfo info )
            raises (PortableInterceptor::ForwardRequest);

            void
            send_reply ( in ContainerServerRequestInfo info );

            void
            send_exception ( in ContainerServerRequestInfo info )
            raises (PortableInterceptor::ForwardRequest);

            void
            send_other ( in ContainerServerRequestInfo info )
            raises (PortableInterceptor::ForwardRequest);
        };
    };
};
```

8.4.6 Server-Side Interception Points

8.4.6.1 receive_request_service_contexts

At this interception point, Interceptors must get their service context information from the incoming request and transfer it to **PortableInterceptor::Current**'s slots.

This interception point is called before the servant manager is called. Operation parameters are not yet available at this point. This interception point may or may not execute in the same thread as the target invocation.

This interception point may raise a system exception. If it does, no other Interceptors' **receive_request_service_contexts** operations are called. Those Interceptors on the Flow Stack are popped and their **send_exception** interception points are called.

This interception point may also raise a ForwardRequest exception If an Interceptor raises this exception, no other Interceptor's **receive_request_service_contexts** operations are called. Those Interceptors on the Flow Stack are popped and their **send_other** interception points are called.

Compliant Interceptors shall properly follow completion_status semantics if they raise a system exception from this interception point. The **completion_status** shall be COMPLETED_NO.

8.4.6.2 receive_request

This interception point allows an Interceptor to query request information after all the information, including operation parameters, are available. This interception point shall execute in the same thread as the target invocation.

In the DSI model, since the parameters are first available when the user code calls arguments, **receive_request** is called from within arguments. It is possible that arguments is not called in the DSI model. The target may call **set_exception** before calling arguments. The ORB shall guarantee that **receive_request** is called once, either through arguments or through **set_exception**. If it is called through **set_exception**, requesting the arguments will result in NO_RESOURCES being raised with a standard minor code of 1.

This interception point may raise a system exception. If it does, no other Interceptors' **receive_request** operations are called. Those Interceptors on the Flow Stack are popped and their **send_exception** interception points are called.

This interception point may also raise a ForwardRequest exception. If an Interceptor raises this exception, no other Interceptors' **receive_request** operations are called. Those Interceptors on the Flow Stack are popped and their **send_other** interception points are called.

Compliant Interceptors shall properly follow completion_status semantics if they raise a system exception from this interception point. The **completion_status** shall be COMPLETED_NO.

8.4.6.3 send_reply

This interception point allows an Interceptor to query reply information and modify reply service context after the target operation has been invoked and before the reply is returned to the client component. This interception point shall execute in the same thread as the target invocation.

This interception point may raise a system exception. If it does, no other Interceptors' **send_reply** operations are called. The remaining Interceptors in the Flow Stack shall have their **send_exception** interception point called.

Compliant Interceptors shall properly follow completion_status semantics if they raise a system exception from this interception point. The **completion_status** shall be COMPLETED_YES.

8.4.6.4 send_exception

When an exception occurs, this interception point is called. It allows an Interceptor to query the exception information and modify the reply service context before the exception is raised to the client component. This interception point shall execute in the same thread as the target invocation.

This interception point may raise a system exception. This has the effect of changing the exception that successive Interceptors popped from the Flow Stack receive on their calls to **send_exception**. The exception raised to the client will be the last exception raised by an Interceptor, or the original exception if no Interceptor changes the exception.

This interception point may also raise a ForwardRequest exception. If an Interceptor raises this exception, no other Interceptors' **send_exception** operations are called. The remaining Interceptors in the Flow Stack shall have their **send_other** interception points called.

If the **completion_status** of the exception is not COMPLETED_NO, then it is inappropriate for this interception point to raise a ForwardRequest exception. The request's at-most-once semantics would be lost.

Compliant Interceptors shall properly follow **completion_status** semantics if they raise a system exception from this interception point. If the original exception is a system exception, the **completion_status** of the new exception shall be the same as on the original. If the original exception is a user exception, then the **completion_status** of the new exception shall be COMPLETED_YES.

8.4.6.5 send_other

This interception point allows an Interceptor to query information available when a request results in something other than a normal reply or an exception. A request could result in a retry (for example, a GIOP Reply with a LOCATION_FORWARD status was received). This interception point shall execute in the same thread as the target invocation.

This interception point may raise a system exception. If it does, no other Interceptors' **send_other** operations are called. The remaining Interceptors in the Flow Stack shall have their **send_exception** interception points called.

This interception point may also raise a ForwardRequest exception. If an Interceptor raises this exception, successive Interceptors' **send_other** operations are called with the new information provided by the ForwardRequest exception.

Compliant Interceptors shall properly follow completion_status semantics if they raise a system exception from this interception point. The **completion_status** shall be COMPLETED_NO.

8.4.7 Interception Flow for ServerContainerInterceptors

Instances of the ServerContainerInterceptor Interface are registered with the run-time environment. The run-time environment logically maintains an ordered list of server-side Container Interceptors. The interceptor list is traversed in order on the receiving interception points and in reverse order on the sending interception points.

8.4.8 Server-side Flow Rules

The server-side flow rules for basic Container Portable interceptors are derived from the general flow rules.

- The starting interception point is **receive_request_service_contexts**; this interception point is called on any given request/reply sequence.
- The set of ending interception points is **send_reply**, **send_exception**, **send_other**. One and only one of these is called on any given request/reply sequence.
- The intermediate interception point is **receive_request**, which is called after **receive_request_service_contexts** and before ending interception point.
- On an exception, **receive_request** may not be called.
- If and only if **receive_request_service_contexts** runs to completion is an ending interception point called.

8.4.8.1 Additional Server-side Details

If, during request processing, a request is canceled because of an ORB shutdown that is initiated by a component server shutdown, **send_exception** is called with the system exception BAD_INV_ORDER with a minor code of 4 (ORB has shutdown).

If a request is canceled for any other reason (for example, a GIOP cancel message has been received), **send_exception** is called with the system exception TRANSIENT with a standard minor code of 3.

On oneway requests, there is no reply sent to the client; however, the target is called and the server component can construct an empty reply. Since closure is necessary, this reply is tracked and **send_reply** is called (unless an exception occurs, in which case **send_exception** is called).

Asynchronous requests, from the server’s point of view, are just normal synchronous requests. Normal interception point flows are followed.

8.5 Extended Container Interceptor Interfaces

In contrast to the basic Container Portable Interceptors the extended Container Portable Interceptors are designed to overcome some limitations in the design of the CORBA Portable interceptors. This is in particular related to the modification of parameters of requests or replies.

Furthermore, extended Container Interceptors intercept a call on a different level than the basic interceptors do. Basic interceptors mostly work on level of ORB dispatching while extended interceptors work on the level of container dispatching.

An extended Container Interceptor is designed to intercept the flow of a request/reply sequence through the Container at specific points so that container services and other container extensions such as QoS Enablers can query the request information and manipulate the invocation parameters.

The primary use of extended Container Portable Interceptors is to enable ORB services and other artifacts modification of component behavior to ensure a certain level of QoS.

There are two types of extended Container Interceptors: client-side (Section 8.5.2, “StubContainerInterceptor,” on page 23) and server-side (Section 8.5.5, “ServantContainerInterceptor,” on page 26).

8.5.1 Extended Interception Points

Each extended Container Portable Interceptor is called at a number of interception points. Figure 8.3 shows the interception points that might be called in a request reply cycle. The details of the client-side interception points are described in Section 8.5.3, “Stub Interception Points,” on page 24. The details of server-side interception points are described in Section 8.5.6, “Servant Interception Points,” on page 27.

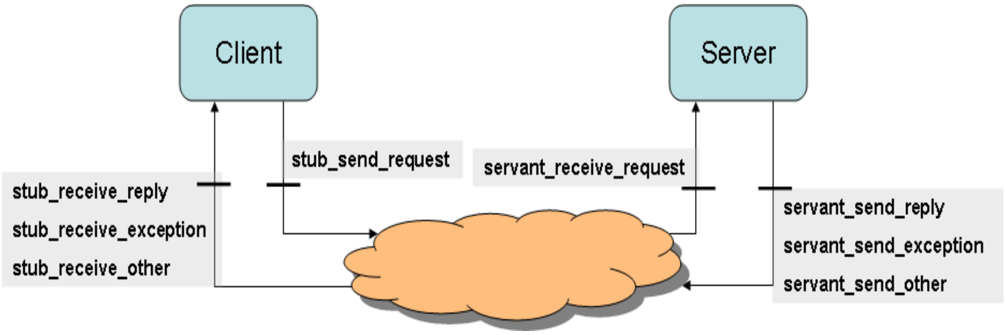

Figure 8.3 - Extended Interception Points

8.5.2 StubContainerInterceptor

The following IDL fragment describes the StubContainerInterceptor interface. To write an extended client-side Container Portable Interceptor this interface needs to be implemented.

```
module Components {
   module ContainerPortableInterceptor {
      local interface StubContainerInterceptor  : ContainerInterceptor
      {
         void
         stub_send_request (
            in ContainerStubRequestInfo info,
            out boolean con)
         raises (PortableInterceptor::ForwardRequest);

         void
         stub_receive_reply (
            in ContainerStubRequestInfo info,
            out boolean con);

         void
         stub_receive_exception (
            in ContainerStubRequestInfo info,
            out boolean con)
        raises(PortableInterceptor::ForwardRequest);
        void
        stub_receive_other (
            in ContainerStubRequestInfo info)
        raises(PortableInterceptor::ForwardRequest);
    };
  };
};
```

8.5.3 Stub Interception Points

8.5.3.1 stub_send_request

This interception point allows an Interceptor to query request information and modify value of the parameters of the call or change the target of the call by directing the call to a different location.

This interception point may modify the parameters of the current request by using the **ContainerStubRequestInfo** object.

This interception point may raise a system exception. If it does, no other Interceptors' **stub_send_request** operations are called. Those Interceptors on the Flow Stack are popped and their **receive_exception** interception points are called.

If a system exception is raised by any interceptor at this interception point, the call is not handled by the ORB and no basic interception point will be called.

The interception point may also raise a ForwardRequest Exception. If an Interceptor raises this exception, no other Interceptors' **send_request** operations are called. Those Interceptors on the Flow Stack are popped and their **receive_other** interception point is called.

Compliant Interceptors shall properly follow completion_status sematics if they raise a system exception from this interception point. The **completion_status** shall be COMPLETED_NO.

This interception point may also end the current call without exception but by providing appropriate return values. This is done by setting the boolean output parameter **proceed_call** to false and by modifying the parameters (i.e., return, inout, and out parameter). If this non-exception completion of a call occurs, no other Interceptors' **stub_send_request** operations are called. Those Interceptors on the Flow Stack are popped and their **stub_receive_other** interception point is called. If call shall continue normally, the **proceed_call** parameter shall be set to true.

8.5.3.2 stub_receive_reply

This interception point allows an interceptor to query the information on a reply after it is returned from the server component and before control is returned to the client component.

This interception point may modify the return parameter of the current request by using the **ContainerStubRequestInfo** object.

This interception point may raise a system exception. If it does so, no other Interceptors' **stub_receive_reply** operations are called. The remaining Interceptors in the Flow Stack shall have their **receive_exception** interception point called.

Compliant Interceptors shall properly follow completion_status sematics if they raise a system exception from this interception point. The **completion_status** shall be COMPLETED_YES.

This interception point may also end the current call without exception but by providing appropriate return values. This is done by setting the boolean parameter **proceed_call** to false and by modifying the parameters (i.e., return, inout, and out parameter). If this non-exception completion of a call occurs, no other Interceptors' **stub_receive_reply** operations are called. Those Interceptors on the Flow Stack are popped and their **stub_receive_other** interception point is called. If call shall continue normally, the **proceed_call** parameter shall be set to true.

8.5.3.3 stub_receive_exception

When an exception occurs, this interception point is called. It allows an Interceptor to query the exception's information before it is raised to the client component.

This interception point may raise a system exception. This has the effect of changing the exception, which successive Interceptors popped from the Flow Stack receive on their calls to **receive_exception**. The exception raised to the client component will be the last exception raised by an Interceptor, or the original exception if no Interceptor changes the exception.

This interception point may raise also a ForwardRequest exception. If an Interceptor raises this exception, no other Interceptors' **receive_exception** operations are called. The remaining Interceptors in the Flow Stack are popped and have their **receive_other** interception point called.

If the **completion_status** of the exception is not COMPLETED_NO, then it is inappropriate for this interception point to raise a ForwardRequest exception. The request's at most-once semantics would be lost.

Compliant Interceptors shall properly follow **completion_status** semantics if they raise a system exception from this interception point. If the original exception is a system exception, the **completion_status** of the new exception shall be the same as on the original. If the original exception is a user exception, then the **completion_status** of the new exception shall be COMPLETED_YES.

This interception point may also end the current call without exception but by providing appropriate return values. This is done by setting the boolean parameter **proceed_call** to false and by modifying the parameters (i.e., return, inout, and out parameter). If this non-exception completion of a call occurs, no other Interceptors' **stub_receive_exception** operations are called. Those Interceptors on the Flow Stack are popped and their **stub_receive_other** interception point is called. If call shall continue normally, the **proceed_call** parameter shall be set to true.

8.5.3.4 stub_receive_other

This interception point allows an Interceptor to query information available when a request results in something other than a normal reply or an exception but the control shall return to the client component and an ending interception point shall be called.

For retries, depending on the policies in effect, a new request may or may not follow when a retry has been indicated. If a new request does follow, while this request is a new request with respect to Interceptors, there is one point of correlation between the original request and the retry. Because control has not returned to the client component, the request scoped **PortableInterceptor::Current** for both the original request and the retrying request is the same.

This interception point may raise a system exception. If it does, no other Interceptors' **stub_receive_other** operations are called. The remaining Interceptors in the Flow Stack are popped and have their **stub_receive_exception** interception point called.

This interception point may also raise a ForwardRequest exception. If an Interceptor raises this exception, successive Interceptors' **stub_receive_other** operations are called with the new information provided by the ForwardRequest exception.

Compliant Interceptors shall properly follow **completion_status** semantics if they raise a system exception from this interception point. The **completion_status** shall be COMPLETED_NO. If the target invocation had completed, this interception point would not be called.

8.5.4 Interceptor Flow for StubContainerInterceptors

Instances of the **StubContainerInterceptor** Interface are registered with the run-time environment. The run-time environment logically maintains an ordered list of client-side Container Interceptors. The interceptor list is traversed in order on the sending interception points and in reverse order on the receiving interception points.

8.5.4.1 Client-side Flow Rules

The client-side flow rules for extended Container Portable interceptors are derived from the general flow rules.

- The starting interception point is: **stub_send_request**; this interception point is called on any given request/reply sequence.
- The set of ending interception points is: **stub_receive_reply**, **stub_receive_exception**, **stub_receive_other**. One and only one of these is called on any given request/reply sequence.
- There are no intermediate exception points.
- If and only if **send_request** runs to completion is an ending interception point called.

8.5.4.2 Additional Client-side Details

If, during request processing, a request is canceled because of an ORB shutdown, which is caused by component server shutdown, **stub_receive_exception** is called with the system exception BAD_INV_ORDER with a minor code of 4 (ORB has shutdown).

If a request is canceled for any other reason (for example, a GIOP cancel message is sent by the ORB), **stub_receive_exception** is called with the system exception TRANSIENT with a standard minor code of 2.

On oneway requests, returning control to the client component may occur immediately or it may return after the target has performed the operation, or somewhere in-between depending on the SyncScope. Regardless of the SyncScope, if there is no exception, **stub_receive_other** is called before control is returned to the client component.

8.5.5 ServantContainerInterceptor

The following IDL fragment describes the **ServantContainerInterceptor** interface.

To write an extended server-side Container Portable Interceptor this interface needs to be implemented.

```
module Components {
   module ContainerPortableInterceptor {
      local interface ServantContainerInterceptor : ContainerInterceptor
      {
         void
         servant_receive_request (
            in ContainerServantRequestInfo info,
            out boolean proceed_call)
         raises (PortableInterceptor::ForwardRequest);

         void
         servant_send_reply (
            in ContainerServantRequestInfo info,
            out boolean proceed_call);

         void
         servant_send_exception (
            in ContainerServantRequestInfo info,
            out boolean proceed_call)
         raises (PortableInterceptor::ForwardRequest);

         void
         servant_send_other (
            in ContainerServantRequestInfo info)
         raises (PortableInterceptor::ForwardRequest);
      };
   };
};
```

8.5.6 Servant Interception Points

8.5.6.1 servant_receive_request

This interception point allows an Interceptor to query request information after all the information, including operation parameters, are available. This interception point shall execute in the same thread as the target invocation.

This interception point may modify the parameters of the current request by using the **ContainerServantRequestInfo** object.

This interception point may raise a system exception. If it does, no other Interceptors' **servant_receive_request** operations are called. Those Interceptors on the Flow Stack are popped and their **servant_send_exception** interception points are called.

This interception point may also raise a ForwardRequest exception. If an Interceptor raises this exception, no other Interceptors' **servant_receive_request** operations are called. Those Interceptors on the Flow Stack are popped and their **servant_send_other** interception points are called.

Compliant Interceptors shall properly follow **completion_status** semantics if they raise a system exception from this interception point. The **completion_status** shall be COMPLETED_NO.

This interception point may also end the current call without exception but by providing appropriate return values. This is done by setting the boolean parameter **proceed_call** to false and by modifying the parameters (i.e., return, inout, and out parameter). If this non-exception completion of a call occurs, no other Interceptors' **servant_receive_request** operations are called. Those Interceptors on the Flow Stack are popped and their **servant_send_other** interception point is called. If call shall continue normally, the **proceed_call** parameter shall be set to true.

8.5.6.2 servant_send_reply

This interception point allows an Interceptor to query reply information after the target operation has been invoked and before the reply is returned to the client component. This interception point shall execute in the same thread as the target invocation.

This interception point may modify the return parameters of the current request by using the **ContainerServantRequestInfo** object.

This interception point may raise a system exception. If it does, no other Interceptors' **servant_send_reply** operations are called. The remaining Interceptors in the Flow Stack shall have their **servant_send_exception** interception point called.

Compliant Interceptors shall properly follow **completion_status** semantics if they raise a system exception from this interception point. The **completion_status** shall be COMPLETED_YES.

This interception point may also end the current call without exception but by providing appropriate return values. This is done by setting the boolean parameter **proceed_call** to false and by modifying the parameters (i.e., return, inout, and out parameter). If this non-exception completion of a call occurs, no other Interceptors' **servant_send_reply** operations are called. Those Interceptors on the Flow Stack are popped and their **servant_send_other** interception point is called. If call shall continue normally, the **proceed_call** parameter shall be set to true.

8.5.6.3 servant_send_exception

When an exception occurs, this interception point is called. It allows an Interceptor to query the exception information before the exception is raised to the client component. This interception point shall execute in the same thread as the target invocation.

This interception point may raise a system exception. This has the effect of changing the exception that successive Interceptors popped from the Flow Stack receive on their calls to **servant_send_exception**. The exception raised to the client will be the last exception raised by an Interceptor, or the original exception if no Interceptor changes the exception.

This interception point may also raise a ForwardRequest exception. If an Interceptor raises this exception, no other Interceptors' **servant_send_exception** operations are called. The remaining Interceptors in the Flow Stack shall have their **servant_send_other** interception points called.

If the **completion_status** of the exception is not COMPLETED_NO, then it is inappropriate for this interception point to raise a ForwardRequest exception. The request's at-most-once semantics would be lost.

Compliant Interceptors shall properly follow **completion_status** semantics if they raise a system exception from this interception point. If the original exception is a system exception, the **completion_status** of the new exception shall be the same as on the original. If the original exception is a user exception, then the **completion_status** of the new exception shall be COMPLETED_YES.

This interception point may also end the current call without exception but by providing appropriate return values. This is done by setting the boolean parameter **proceed_call** to false and by modifying the parameters (i.e., return, inout, and out parameter). If this non-exception completion of a call occurs, no other Interceptors' **servant_send_exception** operations are called. Those Interceptors on the Flow Stack are popped and their **servant_send_other** interception point is called. If call shall continue normally, the **proceed_call** parameter shall be set to true.

8.5.6.4 servant_send_other

This interception point allows an Interceptor to query information available when a request results in something other than a normal reply or an exception. This interception point shall execute in the same thread as the target invocation.

This interception point may raise a system exception. If it does, no other Interceptors' **servant_send_other** operations are called. The remaining Interceptors in the Flow Stack shall have their **servant_send_exception** interception points called.

This interception point may also raise a ForwardRequest exception. If an Interceptor raises this exception, successive Interceptors' **servant_send_other** operations are called with the new information provided by the ForwardRequest exception.

Compliant Interceptors shall properly follow **completion_status** semantics if they raise a system exception from this interception point. The **completion_status** shall be COMPLETED_NO.

8.5.7 Interception Flow for ServantContainerInterceptors

Instances of the ServantContainerInterceptor Interface are registered with the run-time environment. The run-time environment logically maintains an ordered list of server-side Container Interceptors. The interceptor list is traversed in order on the receiving interception points and in reverse order on the sending interception points.

8.5.7.1 Server-side Flow Rules

The server-side flow rules for extended Container Portable interceptors are derived from the general flow rules.

- The starting interception point is **servant_receive_request**; this interception point is called on any given request/ reply sequence.
- The set of ending interception points is **servant_send_reply**, **servant_send_exception**, **servant_send_other**. One and only one of these is called on any given request/reply sequence.
- There is no intermediate interception point.
- If and only if **servant_receive_request** runs to completion is an ending interception point called.

8.5.7.2 Additional Server-side Details

If, during request processing, a request is canceled because of an ORB shutdown, which is initiated by a component server shutdown **servant_send_exception** is called with the system exception BAD_INV_ORDER with a minor code of 4 (ORB has shut down).

If a request is canceled for any other reason (for example, a GIOP cancel message has been received), **servant_send_exception** is called with the system exception TRANSIENT with a standard minor code of 3.

On oneway requests, there is no reply sent to the client; however, the target is called and the server component can construct an empty reply. Since closure is necessary, this reply is tracked and **servant_send_reply** is called (unless an exception occurs, in which case **servant_send_exception** is called).

Asynchronous requests, from the server's point of view, are just normal synchronous requests. Normal interception point flows are followed.

8.6 Request Information

Each interception point is given an object through which the Interceptor can access request information. Client-side and Server-side interception points as well as basic interception points and extended interception points are concerned with different information. **ContainerClientRequestInfo** is passed to basic client-side interception points and **ContainerServerRequestInfo** is passed to basic server-side interception points. **ContainerStubRequestInfo** is passed to extended client-side interception points and **ContainerServantRequestInfo** is passed to extended server-side interception points. There is information that is common to all information objects, so they all inherit from a common interface: **ContainerRequestInfo**.

8.6.1 ContainerRequestInfo

RequestInfo interfaces are used to provide information about a specific invocation at a container interceptor. The **ContainerRequestInfo** interface is a base interface for all RequestInfo interfaces.

```
module Components {
   module ContainerPortableInterceptor {
      local interface ContainerRequestInfo {
         readonly attribute CORBA::OctetSeq origin_id;
         readonly attribute CORBA::OctetSeq target_id;
         readonly attribute FeatureName name;
      };
   };
};
```

8.6.1.1 origin_id

This attribute is read only and contains the instance identifier of the component, which initiated an invocation. The sequence of octets is of zero length if no id is associated with the calling component or whenever the id of the calling component cannot be determined. (See details on component instance identity in Section 8.2, "Component Instance Identity," on page 11).

8.6.1.2 target_id

This attribute is read only and contains the identifier of the component, which receives an invocation. The sequence of octets is of zero length if no id is associated with the component or whenever the id cannot be determined. (See details on component instance identity in Section 8.2, "Component Instance Identity," on page 11).

8.6.1.3 name

This attribute is read only and contains the name of the port that is used for the current call. If the request information is provided client-side interception point, the name of the port must correspond to a port of the client component that is a Receptacle, a Publisher, or an Emitter.

If the request information is provided to a server-side interception point, the name of the port must correspond to a port of the server component that is a Facet or a Consumer. Furthermore, on server-side interception points this attribute may be an empty string, which indicated that the server component's equivalent is the target of the call.

Note – Using a special identifier for equivalent interface would constrain the number of valid names of ports of a component. Since ports always do have a name, an empty identifier is used to identify the component interface.

8.6.2 ContainerClientRequestInfo

ContainerClientRequestInfo interface is a local interface and provides information about invocations done by components on client-side. This interface is provided for basic client-side interception points.

The following IDL fragment defines the **ContainerClientRequestInfo** interface.

```
module Components {
   module ContainerPortableInterceptor {
      local interface ContainerClientRequestInfo : ContainerRequestInfo
      {
         PortableInterceptor::ClientRequestInfo request_info();
      };
   };
};
```

8.6.2.1 request_info

This operation returns the **ClientRequestInfo** object defined in **PortableInterceptors** module. This object contains information about current invocation. The rules for validity of attributes and operations of this **ClientRequestInfo** object defined for CORBA Portable Interceptors apply here as well.

Note – Using the exact same data structure as used in CORBA Portable Interceptors supports easy migration of applications based on CORBA Portable Interceptors to Container Portable Interceptors.

8.6.3 ContainerServerRequestInfo

ContainerServerRequestInfo interface is a local interface and provides information on invocations received by components at server side. This interface is provided for basic server-side interception points.

The following IDL fragment defines the **ContainerServerRequestInfo** interface.

```
module Components {
   module ContainerPortableInterceptor {
      local interface ContainerServerRequestInfo : ContainerRequestInfo
      {
         PortableInterceptor::ServerRequestInfo request_info();
      };
   };
};
```

8.6.3.1 request_info

This operation returns the **ServerRequestInfo** object defined in **PortableInterceptors** module. This object contains information about current invocation. The rules for validity of attributes and operations of this **ServerRequestInfo** object defined for CORBA Portable Interceptors in [CCM] apply here as well.

Note – Using the exact same data structure as used in CORBA Portable Interceptors supports easy migration of applications based on CORBA Portable Interceptors to Container Portable Interceptors.

8.6.4 ContainerStubRequestInfo

ContainerStubRequestInfo interface is a local interface and provides information about invocations done by components on client-side. This interface is provided for extended client-side interception points. The following IDL fragment defines the **ContainerStubRequestInfo** interface.

```
module Components {
   module ContainerPortableInterceptor {
      local interface ContainerStubRequestInfo : ContainerRequestInfo
      {
         attribute Dynamic::ParameterList arguments;
         readonly attribute string operation;
         attribute any result;
         attribute Object target;
         attribute any the_exception;
      };
   };
};
```

8.6.4.1 arguments

This attribute provides the parameters that are part of the current invocation. The arguments are provided as a list of **Dynamic::Parameter**.

8.6.4.2 operation

This attribute provides the name of the operation that is subject of the current invocation.

8.6.4.3 result

This attribute is an any containing the result of the operation invocation.

If the operation return type is **void**, this attribute will be an **any** containing a type code with a **TCKind** value of **tk_void** and no value.

8.6.4.4 target

This attribute is an object pointing to the target of the current invocation.

8.6.4.5 exception

This attribute is an any providing the exception that is received by the client.

8.6.5 ContainerServantRequestInfo

ContainerServantRequestInfo interface is a local interface and provides information on invocations received by components at server side. This interface is provided for extended server-side interception points.

The following IDL fragment defines the **ContainerServantRequestInfo** interface.

```
module Components {
   module PortableInterceptor {
      local interface ContainerServantRequestInfo : ContainerRequestInfo  {
         attribute Dynamic::ParameterList arguments;
         readonly attribute string operation;
         attribute any result;
         attribute Components::EnterpriseComponent target;
         attribute any the_exception;
      };
   };
};
```

8.6.5.1 arguments

This attribute provides the parameters that are part of the current invocation. The arguments are provided as a list of **Dynamic::Parameter**.

8.6.5.2 operation

This attribute provides the name of the operation that is subject of the current invocation.

8.6.5.3 result

This attribute is an any containing the result of the operation invocation.

If the operation return type is **void**, this attribute will be an **any** containing a type code with a **TCKind** value of **tk_void** and no value.

8.6.5.4 target

This attribute contains an object of type **Components::EnterpriseComponent**. This object is the Executor that is used for the current invocation.

8.6.5.5 exception

This attribute is an any providing the exception that is returned to the client.

8.7 Registering Container Interceptors

Container Interceptors are intended to be used for integration of new services and functionality to the run-time environment of components, namely Components Server and Container. They are means for getting access to the request processing within the run-time environment.

In contrast to the ORB Interceptors Container Interceptors can be registered to the Component Server at any time after creation of a CCM run-time and before destruction of this run-time.

To allow a flexible registration of Container Interceptors the following registration interfaces shall be used. For client and server side there is one registration interface for basic Container Interceptors and one registration interface for Extended Container Interceptor.

8.7.1 Client Registration Interface

The **ClientContainerInterceptionRegistration** interface shall be used to register and deregister client container interceptors.

```
module Components {
   module ContainerPortableInterceptor {
      local interface ClientContainerInterceptorRegistration {
         Components::Cookie
         register_client_interceptor (
            in ClientContainerInterceptor ci);

         ClientContainerInterceptor
         unregister_client_interceptor (

            in Components::Cookie cookie)
         raises(InvalidRegistration);
      };
   };
};
```

8.7.1.1 register_client_interceptor

This operation registers a **ClientContainerInterceptor** interface to the run-time environment. If the registration is successful a **Components::Cookie** is returned. This **Cookie** value can be used to identify the registration and needs to be used for a subsequent unregister operation to unregister the **ContainerPortableInterceptor**.

After successful registration of a Client Container Portable Interceptor the run-time environment will call appropriated interception points of this interceptor for all request/replies sequences.

8.7.1.2 unregister_client_interceptor

This operation unregisters a previously registered Client Container Interceptor. This operation expects a **Cookie** value to identify the Interceptor that was previously registered. The result of the operation is the Container Interceptor that is now unregistered. If the provided Cookie value does not correspond to a previously registered Interceptor, the InvalidRegistration exception is raised.

8.7.2 Server Registration Interface

The **ServerContainerInterceptorRegistrationInterface** shall be used to register and unregister server container interceptors.

```
module Components {
   module ContainerPortableInterceptor {
      local interface ServerContainerInterceptorRegistration {
         Components::Cookie
         register_server_interceptor (
            in ServerContainerInterceptor ci) ;

         ServerContainerInterceptor
         unregister_server_interceptor (
            in Components::Cookie ck)
         raises(InvalidRegistration);
      };
   };
};
```

8.7.2.1 register_server_interceptor

This operation registers a **ServerContainerInterceptor** interface to the run-time environment. If the registration is successful, a **Components::Cookie** is returned. This Cookie value can be used to identify the registration and needs to be used for a subsequent unregister operation to unregister the Container Portable Interceptor.

After successful registration of a Server Container Portable Interceptor the run-time environment will call appropriated interception points of this interceptor for all request/replies sequences.

8.7.2.2 unregister_server_interceptor

This operation unregisters a previously registered Server Container Interceptor. This operation expects a **Cookie** value to identify the Interceptor that was previously registered. The result of the operation is the Container Interceptor that is now unregistered. If the provided **Cookie** value does not correspond to a previously registered Interceptor, the InvalidRegistration exception is raised.

8.7.2.3 Stub Registration Interface

The **StubContainerInterceptionRegistration** interface shall be used to register and unregister client container interceptors.

```
module Components {
   module ContainerPortableInterceptor {
        local interface StubContainerInterceptorRegistration {
            Components::Cookie
            register_stub_interceptor (
                in StubContainerInterceptor ci) ;

            StubContainerInterceptor
            unregister_stub_interceptor (
                in Components::Cookie ck)
            raises(InvalidRegistration);
        };
    };
};
```

8.7.2.4 register_stub_interceptor

This operation registers a **StubContainerInterceptor** interface to the run-time environment. If the registration is successful a **Components::Cookie** is returned. This **Cookie** value can be used to identify the registration and needs to be used for a subsequent unregister operation to unregister the Container Portable Interceptor. After successful registration of a Stub Container Portable Interceptor the run-time environment will call appropriated interception points of this interceptor for all request/replies sequences.

8.7.2.5 unregister_stub_interceptor

This operation unregisters a previously registered Stub Container Interceptor. This operation expects a **Cookie** value to identify the Interceptor that was previously registered. The result of the operation is the Container Interceptor that is now unregistered. If the provided **Cookie** value does not correspond to a previously registered Interceptor, the InvalidRegistration exception is raised.

8.7.3 Servant Registration Interface

The **ServerContainerInterceptorRegistrationInterface** shall be used to register and unregister server container interceptors.

```
module Components {
    module ContainerPortableInterceptor {
        local interface ServantContainerInterceptorRegistration {
            Components::Cookie
            register_servant_interceptor (
                in StubContainerInterceptor ci);

            ServantContainerInterceptor
            unregister_servant_interceptor (
                in Components::Cookie ck)
            raises(InvalidRegistration);
        };
    };
};
```

8.7.3.1 register_servant_interceptor

This operation registers a **ServantContainerInterceptor** interface to the run-time environment. If the registration is successful, a **Components::Cookie** is returned. This **Cookie** value can be used to identify the registration and needs to be used for a subsequent unregister operation to unregister the Container Portable Interceptor.

After successful registration of a Servant Container Portable Interceptor the run-time environment will call appropriated interception points of this interceptor for all request/replies sequences.

8.7.3.2 unregister_servant_interceptor

This operation unregisters a previously registered Servant Container Interceptor. This operation expects a **Cookie** value to identify the Interceptor that was previously registered. The result of the operation is the Container Interceptor that is now unregistered. If the provided **Cookie** value does not correspond to a previously registered Interceptor, the InvalidRegistration exception is raised.

8.7.4 InvalidRegistration Exception

An InvalidRegistration exception is raised by the unregister operations in case the provided **Cookie** value does not correspond to a previously registered interceptor or the interceptor has already been unregistered. This exception is defined as follows.

```
module Components {
   module ContainerPortableInterceptor {
      exception InvalidRegistration { };
   };
};
```

8.8 Negotiation

8.8.1 Introduction

Components may require or offer certain QoS characteristics. Matching QoS requirements and QoS offers is important for proper operation of a system. However, in some cases the requirements of a client side component cannot be clearly identified before execution time (e.g., it might depend on user requirements). On the other side sometimes the QoS properties offered by a server-side component cannot be determined before run-time, because of dependencies to the execution environment of the components (e.g., available network interface, main memory, CPU clock). For that reason additional concepts are defined to allow the dynamic agreement on specific QoS characteristics. Instead of a static assignment of QoS properties to a particular component or connection between components, a dynamic way to agree on QoS properties at run-time is required. Such an agreement is called negotiation; it has to be negotiated to check if the server side component can fulfill the client-side requirements. Such a negotiation may take place at connection establishment, for example when a facet is connected to a receptacle. A negotiation may also take place later, whenever the requirements or the offers may change. For a successful agreement on a set of QoS properties it is important that the component instance id of the client component is transmitted in the call context. This allows the server side to correlate a certain agreement on QoS properties with a specific client component instance and enforcing the agreement later on, when the client invokes service of the server, see Section 8.2, “Component Instance Identity,” on page 11.

8.8.2 Constraint Description

In particular for the agreement on specific QoS properties between client and server components, the description of the client side requirements is important. Client side requirements can be modeled by using the UML Profile for QoS. At negotiation time an extract of these modeled QoS information has to be transmitted to the server for requesting particular QoS properties (i.e., by calling the **require_qos** operation).

```
module Components {
   module QoS {
      struct QoSInstance {
          string dimension;
          any value;
      };

      typedef sequence<QoSInstance> QoSInstances;

      struct QoSConstraint {
          string characteristic;
          QoSInstances instances;
      };

      typedef sequence<QoSConstraint> QoSConstraints;
   };
};
```

8.8.2.1 QoSConstraint

As part of the negotiation process the client side QoS requirements are formulated in terms of instances of **QoSConstraint**. A **QoSConstraint** corresponds to a particular **QoSCharacteristics**. The name of that **QoSCharacteristics** is identified by the attribute characteristics. A QoSRequirement shall also contain a sequence of **QoSInstances**. These **QoSInstances** contain the concrete resource requirements.

8.8.2.2 QoSInstance

A **QoSInstance** is a concrete QoSValue that is part of a **QoSConstraint**. The member dimension is of type string and denotes the QoSDimension the **QoSInstance** corresponds to. The member value is of type any and contains the concrete value of this **QoSInstance**. This type of this any corresponds to the unit of the dimension.

8.8.3 Negotiation Interface

```
module Components {
    module QoS {
        interface Negotiation {

            Components::Cookie
            require_qos(
                in QoSConstraint requirements,
                in CORBA::OctetSeq client_id)
            raises (CCMException);
            void
            release_qos (in Components::Cookie ck);
        };
    };
};
```

8.8.3.1 require_qos

This operation is used to express requirements on particular QoS properties of a client. These requirements are expressed as QoSConstraints. The receiving party of this call shall check whether the requirements can be fulfilled or not. This may or may also imply the reservation of resources. If this operation returns without exception the server side agrees to fulfil the requirements of the client side. In that case the operation returns a **Cookie** to identify this agreement. This **Cookie** can be used to release this agreement in subsequent **release_qos** call. Whenever the server side is not able to fulfill the client's requirements it shall return a CCMException with the reason QOS_ERROR. This does mean that no agreement on QoS properties between client and server can be found.

8.8.3.2 release_qos

This operation can be used to release a particular agreement on QoS properties. This agreement is identified by the **Cookie** provided as parameter. The server can free all resources that might have been allocated for this agreement before.

8.8.4 Provision of Negotiation Interface

To facilitate the negotiation process, i.e., the agreement on certain QoS properties, the negotiation interface is provided by every component that is deployed into a QoS-aware container. To have minimal impact on the CCM Architecture the Negotiation interface shall be offered by each component as a default facet. This default facet is not part of any component definition, it can be seen as a virtual facet. The implementation and the management of this facet (e.g., the navigation capabilities) are added only by the QoS-aware container.

The negotiation facet has the identifier **ccm_qos_negotiation**. This facet is of type **Components::QoS::Negotiation**. In case a component is deployed to a QoS-aware container the navigation operation **provide_facet** with the parameter **_ccm_qos_negotiation** shall return such an interface. In case of a QoS-unaware container the **provide_facet** operation raises an InvalidName exception.

8.8.5 Definition of Negotiation Flow

8.8.5.1 Connection Flow Details

The CORBA Component Model supports the explicit connection establishment of components. This means that all interactions between CCM components only happen if an explicit connection establishment phase is accomplished before. Figure 8.4 shows a typical connection setup, where a facet port of a server component is connected to a receptacle port of a client component. The entity that controls the connection setup (3rd_party) can be a deployment tool in case of constructing the initial configuration of a CCM based system or it can be even another application component that connects components at run-time.

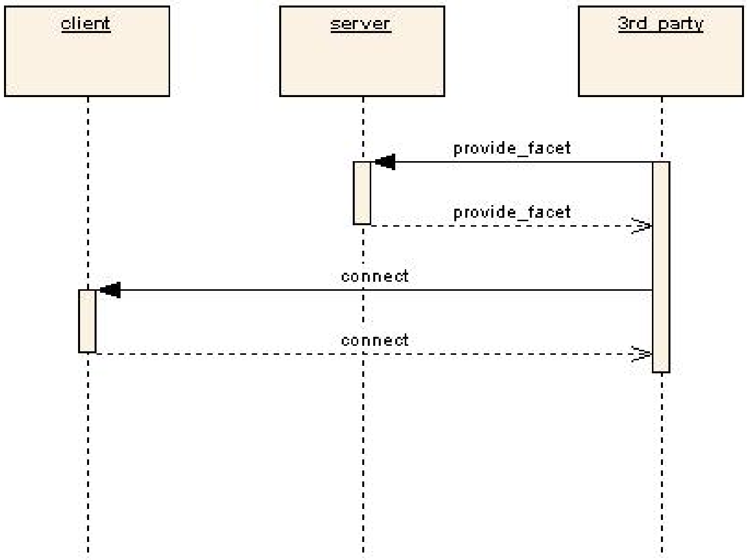

Figure 8.4 - Connection Setup between CORBA Components

A non-exceptional completing of the connection operation indicates that the connection between the components is established. In the case that a client component has some specific QoS requirements, the connection setup shall not complete without proper agreement on certain QoS properties between server component and client component. For that reason the following negotiation flow has to be followed at connection setup.

Receiving the connect operation the client side shall call the **require_qos** operation at the server component that is about to be connected to the client component. The operation provides as parameter the description of the QoS requirements of the client component as ConstraintDescriptions (Section 8.8.2, “Constraint Description,” on page 36). On the server side these requirements shall be evaluated. If the server side can fulfill the requirements of the client side it has to return a cookie value identifying the agreement between client and server component. If the server component side cannot fulfill the client requirement, it has to raise a CCMException with reason QOS_ERROR. If the server side raises this exception, the client has two options. This first option is to repeat the **require_qos** call with a different **ConstraintDescriptions**. This can be repeated if this fails again. The other option is to let the connection setup fail due to the reason that the QoS constraints cannot be supported. In this case the client will raise a Components::InvalidConnection exception.

The following picture represents a connection setup with a successful connection setup where client requirements can be fulfilled by the server side.

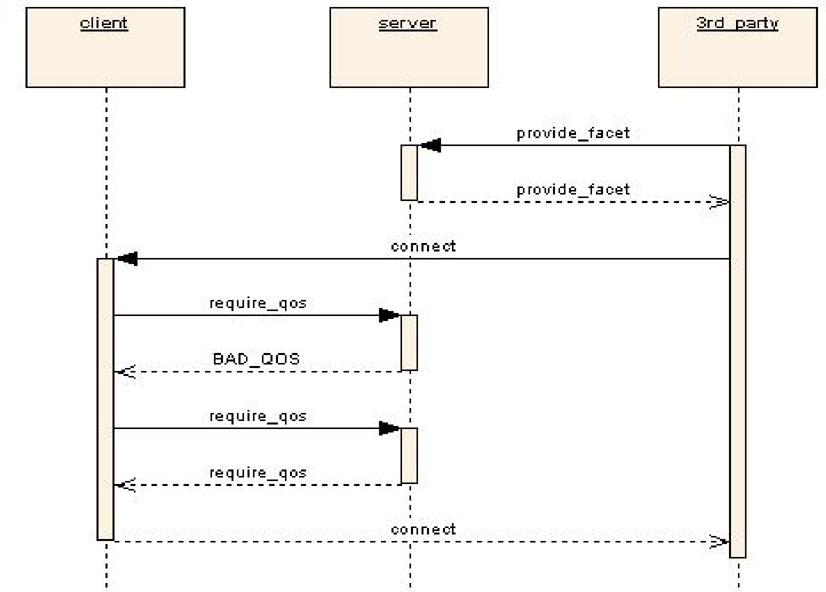

Figure 8.5 - Connection Setup with Negotiation and Option 1

The following picture shows a connection setup that fails due to the reason that an agreement on a specific QoS property cannot be reached between client and server.

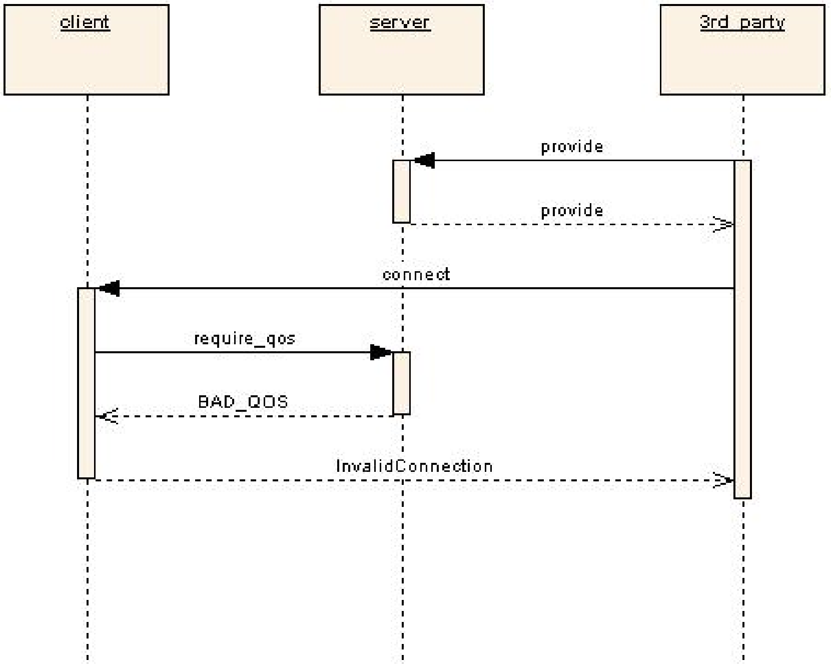

Figure 8.6 - Connection Setup with negotiation and Option 2

The negotiation of QoS constraints requires two things, the first one is external interfaces for the negotiation and the second one is the internal representation of the negotiation. This means that the server side container may allocate resources to accept a request for a specific QoS requirement or refuses it otherwise. The container is responsible for involving appropriate entities into a negotiation.

8.8.5.2 Disconnection Flow details

Links between CORBA Components shall be disconnected explicitly by calling the corresponding disconnect operation at the client component. This is usually done by the party that created the connection. In any way the disconnecting entity needs the **Cookie** value that was returned by the **connect** operation. Whenever a disconnection occurs the client component shall inform about the disconnection in order to allow the server side component to free allocated resources that are dedicated to that specific connection.

The following picture shows the usual way of disconnecting two components.

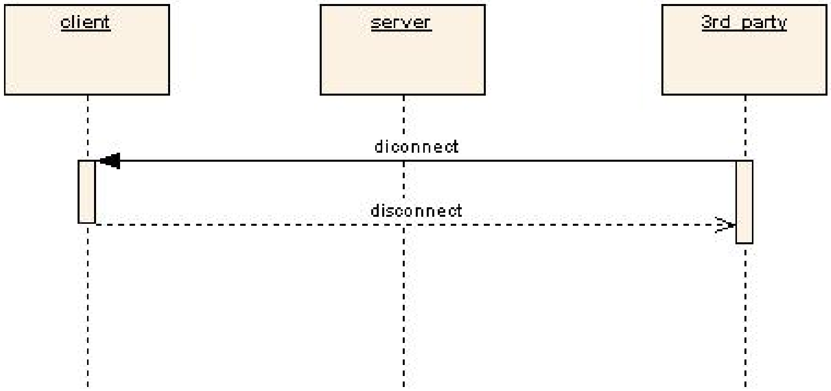

Figure 8.7 - Disconnecting Components

The following flow shall be applied whenever a QoS agreement between client and server has been established (i.e., a **require_qos** call was return without exception). Whenever the client side receives a disconnect call it shall call the **release_qos** operation providing the cookie value that was returned by a preceding **require_qos** call.

The following picture shows how components are disconnected in case a QoS agreement exists and the right cookie value is provided with the **release_qos** call.

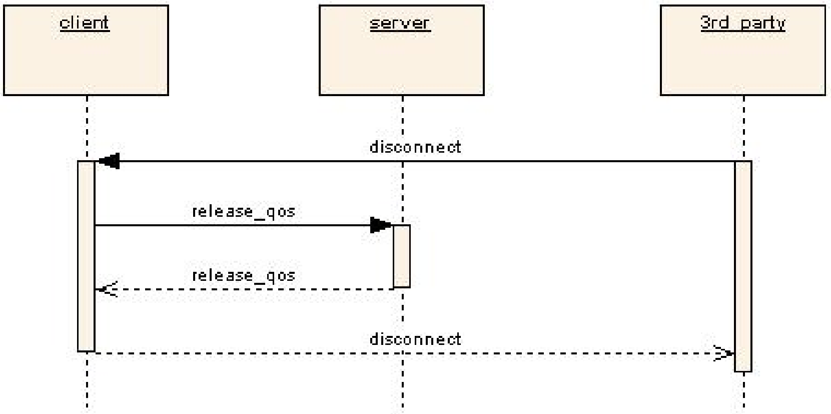

Figure 8.8 - Disconnecting Resource with Releasing Resources

8.8.5.3 Re-negotiation flow details

In dynamic environments the agreement on a particular QoS is needed not only at connection setup but also later in the lifetime of a connection. This could be because the requirements of the client component may change over time.

In such cases the following flow has to be applied. The client shall call the **release_qos** operation to allow the server side to free allocated resource. The next step is a **require_qos** call. The parameter of this operation shall be an instance of ConstraintsDescriptions that expresses the changed requirements. In case the requirements cannot be fulfilled by the server it will raise a CCMException with reason QOS_ERROR exception. If this happens the client has two options. First option is to repeat the call **require_qos** with different **ConstraintDescriptions**. The second option is to disconnect the interface that is currently bound to the client's receptacle.

The following picture shows the general flow details that occur at re-negotiating.

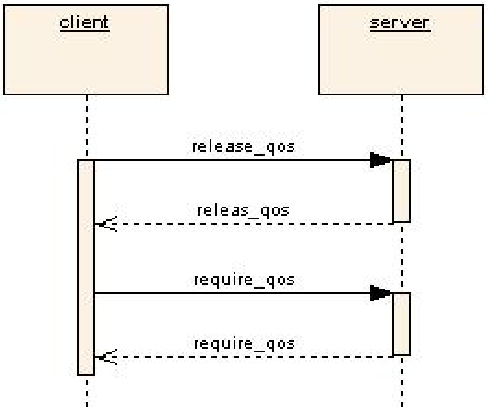

Figure 8.9 - Successful Re-negotiation

In case a renegotiation fails (e.g., possibly because of unavailable resources) the client shall make a disconnection of the interface that is currently bound to the client's receptacle. Figure 8.10 illustrates this flow detail.

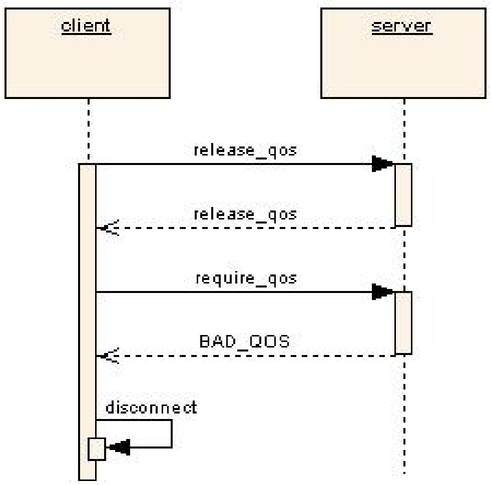

Figure 8.10 - Unsuccessful Re-Negotiation Flow

8.9 Extension Container

8.9.1 Introduction

The CORBA Component Model defines the Component Server and the Container as the standard run-time environment for components. Container vendors can specialize this run-time environment by providing specific container implementation or by introducing completely new container types. However such extensions are vendor specific. The extension container is a means for deploying run-time extensions in a standard way. The extension container can host specific components, which are not typical application components.

The extension container provides an internal interface to the hosted components, which in particular supports the control of the run-time of application components. For example this internal interface, the extension context, provides access to the COPI registration interfaces.

Components hosted by the extension container shall realize a specific call-back interface, the extension component interface.

8.9.2 ExtensionContext

The **ExtensionContext** interface is offered by the container to a component implementation. The following IDL fragment defines this interface.

```
module Components {
   local interface ExtensionContext : CCMContext {

      Components::ContainerPortableInterceptor::ClientContainerInterceptorRegistration
      get_client_interceptor_registration ()
         raises (CCMException);

      Components::ContainerPortableInterceptor::ServerContainerInterceptorRegistration
      get_server_interceptor_registration ()
```

```
        raises (CCMException);

      Components::ContainerPortableInterceptor::StubContainerInterceptorRegistration
      get_stub_interceptor_registration()
        raises (CCMException);

      Components::ContainerPortableInterceptor::ServantContainerInterceptorRegistration
      get_servant_interceptor_registration()
        raises (CCMException);

      Cookie
      install_service_reference(
        in string service_id, in Object objref)
        raises (CCMException);

      Object
      uninstall_service_reference(in Cookie ck)
        raises (CCMException);

      QoSPropertyRegistration
      get_qos_property_registration()
       raises (CCMException);
   };
};
```

8.9.2.1 get_client_interceptor_registration

This operation returns a **ClientContainerInterceptionRegistration** interface, which can be used subsequently to register a COPI of type **ClientContainerInterceptor**. If a registration interface is not available, an exception of type CCMException with a reason REGISTRATION_ERROR shall be raised.

8.9.2.2 get_server_interceptor_registration

This operation returns a **ServerContainerInterceptionRegistration** interface, which can be used subsequently to register a COPI of type **ServerContainerInterceptor**. If a registration interface is not available, an exception of type CCMException with a reason REGISTRATION_ERROR shall be raised.

8.9.2.3 get_stub_interceptor_registration

This operation returns a **StubContainerInterceptionRegistration** interface, which can be used subsequently to register a COPI of type **StubContainerInterceptor**. If a registration interface is not available, an exception of type CCMException with a reason REGISTRATION_ERROR shall be raised.

8.9.2.4 get_servant_interceptor_registration

This operation returns a **ServantContainerInterceptionRegistration** interface, which can be used subsequently to register a COPI of type **ServantContainerInterceptor**. If a registration interface is not available, an exception of type CCMException with a reason REGISTRATION_ERROR shall be raised.

8.9.2.5 install_service_reference

This operation can be used to register a reference to container services, which are run-time extensions hosted by the extension container. After successful installation of such a service reference the service reference can be resolved by plain applications components by using the **resolve_service_reference** operation defined in the **CCMContext** interface (see 8.10, Modification of CCMContext interface). The name under which the service reference is bound is provided with the **service_id** parameter. Whenever this name is already in use, this operation shall return an exception of type CCMException with reason SERVICE_INSTALLATION_ERROR. In a non-exceptional case this operation returns a **Cookie**, which uniquely identifies this service reference installation. This **Cookie** value can be used for subsequent calls of **uninstall_service_reference**.

8.9.2.6 uninstall_service_reference

This operation can be used to uninstall a service reference that was previously installed by calling **install_service_reference**. The **Cookie** value that was returned as a result of the installation call has to be provided as parameter to this operation. If the **Cookie** does not identify a currently installed service reference, the operation raises a CCMException with reason SERVICE_INSTALLATION_ERROR.

8.9.2.7 get_qos_property_registration

This operation returns a QoSPropertyRegistration interface, which can be used subsequently to register a **QoSPropertyInstance**. If a registration interface is not available, an exception of type CCMException with a reason REGISTRATION_ERROR shall be raised.

8.9.3 ExtensionComponent

The **ExtensionComponent** interface is offered by a component implementation that is executed in an Extension container. The interface is defined by the following IDL fragment.

```
module Components {
   local interface ExtensionComponent : EnterpriseComponent {

      void
      set_extension_context (in ExtensionContext ctx)
         raises (CCMException);

      void
      ccm_remove ()
         raises (CCMException);
   };
};
```

8.9.3.1 set_extension_context

The **set_extension_context** operation is used to set the **ExtensionContext** of the component. The container calls this operation after a component instance has been created. This operation is called outside the scope of an active transaction. The component may raise the CCMException with the SYSTEM_ERROR minor code to indicate a failure caused by a system level error.

8.9.3.2 ccm_remove

The **ccm_remove** operation is called by the container when the servant is about to be destroyed. It informs the component that it is about to be destroyed. The component may raise the CCMException with the SYSTEM_ERROR minor code to indicate a failure caused by a system level error.

8.10 Modification of CCMContext interface

To allow normal application components to access the services provided by a container service the CCMContext interface is used. A number of container services that can be used by the component implementation are already defined. This includes security and transaction service. However, the number of services is fixed since their service interface is partly provided by the **CCMContext** interface itself.

It is the intension of this modification to make the number of container services that may be available to application components open. This is achieved by defining the operation **resolve_service_reference** as part of the CCMContext interface.

```
module Components {
   local interface CCMContext
   {
      Principal get_caller_principal();

      CCMHome get_CCM_home();

      boolean get_rollback_only() raises(IllegalState);

      Transaction::UserTransaction
      get_user_transaction() raises(IllegalState);

      boolean is_caller_in_role(in string role);

      void set_rollback_only() raises(IllegalState);

      /* QoS4CCM */
      Object
      resolve_service_reference(in string service_id)
      raises (CCMException);
   };
};
```

8.10.1 resolve_service_reference

This operation returns references to container services in a generic way. The parameter **service_id** identifies the service that shall be resolved. In case the provided service id corresponds to a service that is provided by the run-time environment the operation returns an object reference to that service. The Executor can narrow this reference to the specific service interface to make use of the container service.

If no service with the specified service id is known to the container, the operation will raise a CCMException with reason OBJECT_NOT_FOUND.

8.11 QoS Enabler

8.11.1 Introduction

Implementing the QoS extension that should be part of the run-time environment (component server and container) could be done in a proprietary way by modifying the container. This specification defines concepts for developing and integrating such extension in a standard way. This is achieved by using the component concept, which means that the run-time extensions can be realized as components. These components differ from plain application components in that they are deployed into containers of a particular type (container category). This type is the extension container type as defined in 8.9, Extension Container. Components that are deployed in this container and that are responsible for managing particular QoS properties are called QoSEnablers.

A QoS Enabler is responsible for bringing additional functionality into the container. A QoSEnabler is concerned with a particular QoS characteristic. It depends on the implementation strategy whether only one instance of a QoSEnabler is responsible for managing a QoS characteristic in run-time environment (component server) or several instances of the same QoSEnabler type are used for that.

A QoSEnabler can use different techniques to provide the QoS functionality to the run-time environment. In case the QoSEnabler needs to monitor the interactions between components it can use the Container Portable Interceptors (COPI) to realize this. Furthermore, the QoSEnabler can use COPI interface also to modify call chain to change the behavior of components, to ensure certain QoS properties.

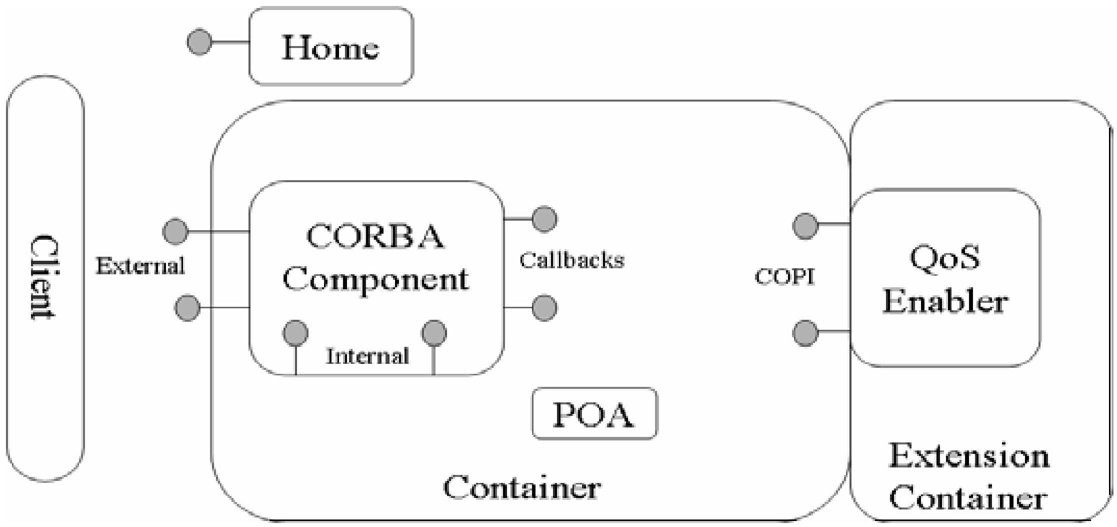

Figure 8.11 - Extension Container Category

The QoS Enabler can offer a special usage interface that can be used by a component implementation (executor) to retrieve QoS category specific information. On the other side the executor may provide special call-back operations that give the QoS Enabler the opportunity to configure the component implementation (executor) before the actual request is performed by that executor. This means the executor has to be QoS aware because it has to behave according to the configuration done by the QoS Enabler.

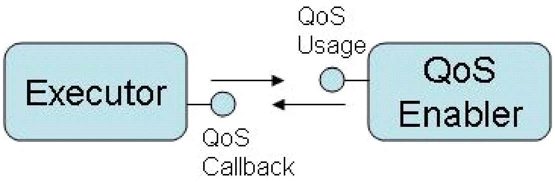

Figure 8.12 - QoS Callback

8.11.2 QoS Usage Interface

If a component wants to retrieve special information about the state of a particular QoS related property it needs to get a reference to the appropriate QoS Enabler. For this purpose it uses the **resolve_service_reference**() operation at the **CCMContext** interface as defined in 8.10, Modification of CCMContext interface.

The service_id parameter corresponds to the QoS characteristics. Whenever a QoSEnabler offers such a QoS Usage interface it has to register this interface to the run-time environment by using the install_service_reference operation provided by the extension context. It is QoS category dependent whether a QoS Enabler offers such a QoS usage interface or not.

In some very constrained environments, it can be more efficient to design services as interfaces, not necessarily part of a component. Since the service is co-localized with components, it can be designed as an interface accessible at container level. The following interface is a standard base interface for services and is designed to allow service configuration when installing reference using the **install_service_reference()** operation.

A service is identified with the characteristic it manages which is part of the **QoSConstraint** definition. A QoSConstraint shall also contain a sequence of **QoSInstances**. These QoSInstances are the concrete resource requirements.

```
module Components
{
 module QoS
 {
  local interface QoSUsage : public ExtensionComponent
  {
   attribute readonly string characteristic;

Cookie install_qos_property( in QoSInstances instances )
                         raises(CCMException);
void uninstall_qos_property( in Cookie ck ) ;
  } ;
 } ;
} ;
```

8.11.2.1 characteristic

This attribute identifies a **QoSCharacteristic** corresponding to the **service_id parameter** of the i**nstall_service_reference** operation.

8.11.2.2 install_qos_property

This operation is called by the container, to set the service configuration defined in **QoSInstances** that are description of the constraints. The QoSInstances can be expressed in a **QoSPropertyInstance** and bound to a component feature if needed. Whenever the service is not able to process a **QoSValue** it shall return a **CCMException** with the reason **QOS_ERROR**. This means that the QoS property can't be installed or managed by service.

8.11.2.3 uninstall_qos_property

This operation is called by the container to unset some QoS properties managed by the service and uninstall related **QoSPropertyInstance** if exist.

8.11.3 QoSCallback Interface

In some cases the component implementation, the Executor, may be QoS-aware. This means it can realize specific QoS properties on its own, though the management of QoS properties is to some extent delegated to a QoS Enabler. This executor is delivered to the QoS Enabler via the servant COPI. The member target is part of the **ContainerServantRequestInfo** object that is provided at interception points.

To get a reference to the QoS Callback interface the Executor Locator approach has to be used. This means that the Executor provided to the COPI is in fact an Executor Locator. A subsequent call of **obtain_executor** with the name **_ccm_qos_callback** can be used to get a reference to such a call-back interface.

8.11.4 Packaging and Deployment of QoS Enablers

Since QoS Enablers are supposed to be dynamic parts of a component server it is most likely that a QoS Enabler is loaded into a component server at run-time. Nevertheless, this should not exclude the possibility to build pre-configured component server with a fixed set of QoS Enablers. But in any case it should be possible to load a QoS Enabler at run-time later on.

Two things need to be achieved to support this behavior. The first one is the physical transportation of the dynamic library to the target node. The second one is the loading of the library by calling the appropriate entry point. Both can be accomplished by using standard CCM means for deployment and configuration. For the physical transportation the mechanism provided by the ComponentInstallation is used.

The same mechanism that is used for loading component libraries is used for loading QoS Enabler libraries. For that reason a container of the category Extension shall be instantiated in the component server. QoS Enablers do not need to run in a container exclusively they can be installed in any other already running extension container.

8.11.5 Monitoring

To determine whether a certain QoS contract is fulfilled or violated a QoS Enabler may use special monitoring mechanisms. These mechanisms may be provided by a container or the QoS Enabler may use self provided monitoring capabilities. For example a QoS Enabler needs to measure whether the provided bandwidth does not fall under that level defined by the active QoS agreement. Because the monitoring depends very much on the QoS category and also on the implementation of a QoS Enabler this specification defines no architecture for that. If a contract is violated, the QoS Enabler is responsible for terminating the contract.

8.12 Dynamic Adaptation of a Running Application

In a most general way, the QoS Enabler life cycle may need to be managed over the component life cycle. This means that creation, deletion, and configuration of QoSEnabler have to be done independently of corresponding component means. Container vendors are free to exploit CCM capacities to add specific operations for QoSEnabler configuration, however configuration at run-time may become constraining:

- Client may have to manage QoSEnabler owned by components with heterogeneous configuration interface.
- Client implementation may be generic code and not be aware about specific component definition.
- Client in an embedded environment may not be allowed to use CORBA dynamic capacities.

To help container vendors to implement standard and homogeneous dynamic configuration mechanisms the **StandardConfigurator** and the HomeConfiguration definition have been extended.

8.12.1 Modification of StandardConfigurator interface

```
module Components {
   interface StandardConfigurator : Configurator {
      void set_configuration (in ConfigValues descr);
      /* QoS4CCM */
      ConfigValues get_configuration( in CCMObject comp )
                                 raises (WrongComponentType);
   };
};
```

8.12.1.1 get_configuration

This operation returns a sequence of **ConfigValue** instances containing the configuration of the target component. If the target component is not of the type expected by the configurator, the **get_configuration** operation shall raise the **WrongComponentType** exception.

8.12.2 Modification of HomeConfiguration interface

```
module Components {
   interface HomeConfiguration : CCMHome {
      void set_configurator ( in Configurator cfg);
      /* QoS4CCM */
      Configurator get_configurator();
      void set_configuration_values ( in ConfigValues config);
      void complete_component_configuration (in boolean b);
      void disable_home_configuration();
   };
};
```

8.12.2.1 get_configurator

This operation returns a Configurator reference to the standard configurator that manages this home. This one may be able to process particular configuration values and to call specific user interface of component and component's home.

8.13 Binding QoSConstraint with Component Feature

Even if the interceptor mechanism allows for an injection of non-functional properties at interception points, the level of control by manipulating the calling thread is limited. It is for instance possible to block this thread, but not to ensure that a pool of threads with a certain priority should handle a request - as it is possible in RT-CORBA. The necessity to configure the underlying middleware, for instance the use of certain policies of the portable object adapter, or object references, may be designed with new interfaces as well as interceptors.

This section introduces two interfaces based on interception principle:

1. The **QoSPropertyInstance** is designed to provide a means for non-functional properties definition. A **QoSPropertyInstance** has to be referenced inside the run-time environment in association with a component feature (ComponentFeature). The container manages the activation and deactivation of the **QoSPropertyInstance**. This interface can be necessary to configure particular QoS constraints not related to interception mechanisms. The considered QoS properties and the way they are managed in a CCM framework is vendor specific.

   Some examples of integration points:

   - At connection time the connect operation of the container can use this interface to configure communication protocol.
   - Another key concern is the creation of a component instance, and the way its facets take shape. The CORBA and real-time CORBA Portable object adapter propose different strategies to activate an object. This can be considered at container level to configure POA policies at this stage.

2. The **QoSPropertyInstanceRegistration**, as well as container interceptor registration, shall be used to register and unregister **QoSPropertyInstance**.

8.13.1 QoSPropertyInstance

This interface allows associating a QoS constraint (via QoSInstances) to component features (ComponentFeatures, meaning facets, receptacles, …).

The following IDL fragment defines the QoSPropertyInstance interface.

```
module QoS {
   local interface QoSPropertyInstance {
      attribute readonly string functionality;
      attribute QoSInstances qos_instances;
      void activate( in string operation, in ParameterList parameters)
              raises (CCMException );
      void deactivate(  in string action,
                        in ParameterList parameters);
   };
};
```

8.13.1.1 functionality

This attribute provides the name of the functionality that the QoS property corresponds to. It corresponds to the component feature name (e.g., facet or receptacle name).

8.13.1.2 qos_instances

This attribute is a sequence of **QoSInstance** (QoSInstances). This set of tag/value pairs describes the property. QoS instances can be priority with RTCORBA::Priority value or transport protocol policy with standard or vendor specific protocol policy value.

8.13.1.3 activate

This operation is called by the container (for example, at connection or facet activation stage) to activate a property on a correlated component feature (see Section 7.2.2, “Binding,” on page 6).

The action parameter allows to identify the kind of activation to process (on activation, on connection, …). The parameters parameter allows to process the activate operation on object references (e.g., **CORBA::Object** representing a connection). This is a ParameterList because, depending on the kind of configuration, these objects can be IN, INOUT, or OUT. If activation can’t be realized, the operation shall return a CCMException with the reason QOS_ERROR.

8.13.1.4 deactivate

This operation is called by container to disable a property associated to a component feature.

8.13.2 QoSPropertyInstanceRegistration

The following IDL fragment defines the **QoSPropertyInstanceRegistration** interface.

```
module QoS {
   local interface QoSPropertyInstanceRegistration {
      Cookie register_qos_property( in QoSPropertyInstance instance );
      void unregister_qos_property( in Cookie ck )
                                                   raises(InvalidRegistration );
   };
};
```

8.13.2.1 register_qos_property

This operation may be called by a service to register a **QoSPropertyInstance** interface to the run-time environment and to bind the property to a functionality of the component. If the operation is successful, it returns a **Cookie** to identify the binding. This Cookie value can be used to identify the registration and needs to be used for subsequent operation to unregister the property.

8.13.2.2 unregister_qos_property

This operation unregisters a previously registered property. This operation expects a **Cookie** value to identify the QoS property that was previously registered. If the provided Cookie value does not correspond to a previously registered property, the InvalidRegistration exception is raised.

Annex A: Components.idl

(Normative)

This section gives a summary of all IDL definitions that are added or changed by this specification. The modifications are made based on the file Components.idl part of the document ptc/02-10-04.

A.1 CCMExceptionReason

```
module Components {
   enum CCMExceptionReason
   {
      SYSTEM_ERROR,
      CREATE_ERROR,
      REMOVE_ERROR,
      DUPLICATE_KEY,
      FIND_ERROR,
      OBJECT_NOT_FOUND,
      NO_SUCH_ENTITY,
      /* extended by QoS4CCM */
      QOS_ERROR,
      REGISTRATION_ERROR,
      SERVICE_INSTALLATION_ERROR
   };
}; // end module components
```

A.2 Module ContainerPortableInterceptor

```
module Components {
   module ContainerPortableInterceptor {
      struct CustomSlotItem
      {
         string identifier;
         any content;
      };
      typeid CustomSlotItem
      “IDL:omg.org/Components/ContainerPortableInterceptor/CustomSlotItem:1.1;

      typedef sequence<CustomSlotItem> CustomSlotItemSeq;

      typeid CustomSlotItemSeq
      “IDL:omg.org/Components/ContainerPortableInterceptor/CustomSlotItemSeq:1.1;

      struct COPIServiceContext
      {
         CORBA::OctetSeq origin_id;
         CORBA::OCtetSeq target_id;
         CustomSlotItemSeq slot_info;
      };

      typeid COPIServiceContext
      “IDL:omg.org/Components/ContainerPortableInterceptor/COPIServiceContext:1.1;

      typedef sequence<CustomSlotItem> CustomSlotItemSeq;

      struct IntegrationPoint
      {
         string port;
         string operation;
      };

      struct COPIServiceContext
      {
         CORBA::OctetSeq origin_id;
         CORBA::OCtetSeq target_id;
         CustomSlotItemSeq slot_info;

      local interface ContainerRequestInfo {
         readonly attribute CORBA::OctetSeq origin_id;
         readonly attribute CORBA::OctetSeq target_id;
         readonly attribute FeatureName name;
      };

      local interface ContainerClientRequestInfo : ContainerRequestInfo
      {
         PortableInterceptor::ClientRequestInfo request_info();
      };

      local interface ContainerServerRequestInfo : ContainerRequestInfo
      {
         PortableInterceptor::ServerRequestInfo request_info();
      };

      local interface ContainerStubRequestInfo : ContainerRequestInfo
      {
         attribute Dynamic::ParameterList arguments;
         readonly attribute string operation;
         attribute any result;
         attribute Object target;
         attribute any the_exception;
      };

      local interface ContainerServantRequestInfo : ContainerRequestInfo  {
         attribute Dynamic::ParameterList arguments;
         readonly attribute string operation;
         attribute any result;
         attribute Components::EnterpriseComponent target;
         attribute any the_exception;
      };

      local interface ContainerInterceptor
      {
         readonly attribute string name;
         attribute unsigned short priority;
         attribute IntegrationPoint registration_info;

         void
         destroy ();

         void
         set_slot_id(in PortableInterceptor::SlotId slot_id);
      };

      local interface ClientContainerInterceptor  : ContainerInterceptor
      {
      void
         send_request (in ContainerClientRequestInfo info)
         raises (PortableInterceptor::ForwardRequest);

         void
         send_poll (in ContainerClientRequestInfo info);

         void
         receive_reply (in ContainerClientRequestInfo info);

         void
         receive_exception (
            in ContainerClientRequestInfo info)
         raises (PortableInterceptor::ForwardRequest);

         void
         receive_other (in ContainerClientRequestInfo info)
         raises (PortableInterceptor::ForwardRequest);
      };

      local interface ServerContainerInterceptor : ContainerInterceptor
      {
         void
         receive_request_service_contexts (
            in ContainerServerRequestInfo csi )
         raises (PortableInterceptor::ForwardRequest);

         void
         receive_request ( in ContainerServerRequestInfo info )
         raises (PortableInterceptor::ForwardRequest);
```

```
   void
   send_reply ( in ContainerServerRequestInfo info );

   void
   send_exception ( in ContainerServerRequestInfo info )
   raises (PortableInterceptor::ForwardRequest);

   void
   send_other ( in ContainerServerRequestInfo info )
   raises (PortableInterceptor::ForwardRequest);
};

local interface StubContainerInterceptor  : ContainerInterceptor
{
   void
   stub_send_request (
      in ContainerStubRequestInfo info,
      out boolean con)
   raises (PortableInterceptor::ForwardRequest);

   void
   stub_receive_reply (
      in ContainerStubRequestInfo info,
      out boolean con);

   void
   stub_receive_exception (
      in ContainerStubRequestInfo info,
      out boolean con)
   raises(PortableInterceptor::ForwardRequest);

   void
   stub_receive_other (
      in ContainerStubRequestInfo info)
   raises(PortableInterceptor::ForwardRequest);
};

local interface ServantContainerInterceptor : ContainerInterceptor
{
   void
   servant_receive_request (
      in ContainerServantRequestInfo info,
      out boolean proceed_call)
   raises (PortableInterceptor::ForwardRequest);

   void
   servant_send_reply (
      in ContainerServantRequestInfo info,
      out boolean proceed_call);
   void
   servant_send_exception (
      in ContainerServantRequestInfo info,
      out boolean proceed_call)
   raises (PortableInterceptor::ForwardRequest);

   void
   servant_send_other (
      in ContainerServantRequestInfo info)
   raises (PortableInterceptor::ForwardRequest);
};

exception InvalidRegistration { };

local interface ClientContainerInterceptorRegistration {
   Components::Cookie
   register_client_interceptor (
      in ClientContainerInterceptor ci);

   ClientContainerInterceptor
   unregister_client_interceptor (
      in Components::Cookie cookie)
   raises(InvalidRegistration);
   };


local interface ServerContainerInterceptorRegistration {
   Components::Cookie
      register_server_interceptor (
      in ServerContainerInterceptor ci) ;

   ServerContainerInterceptor
      unregister_server_interceptor (
      in Components::Cookie ck)
      raises(InvalidRegistration);
};

local interface StubContainerInterceptorRegistration {
   Components::Cookie
      register_stub_interceptor (
      in StubContainerInterceptor ci);

   StubContainerInterceptor
      unregister_stub_interceptor (
      in Components::Cookie ck)
      raises(InvalidRegistration);
};

local interface ServantContainerInterceptorRegistration {
   Components::Cookie
      register_servant_interceptor (
            in ServantContainerInterceptor ci) ;

        ServantContainerInterceptor
            unregister_servant_interceptor (
            in Components::Cookie ck)
            raises(InvalidRegistration);
    };

  }; // end module ContainerPortableInterceptors
}; // end module Components
```

A.3 Interface CCMContext

```
module Components {
  local interface CCMContext
  {
    Principal get_caller_principal();

    CCMHome get_CCM_home();

    boolean get_rollback_only() raises(IllegalState);

    Transaction::UserTransaction
    get_user_transaction() raises(IllegalState);

    boolean is_caller_in_role(in string role);

    void set_rollback_only() raises(IllegalState);

    /* QoS4CCM */
    Object
    resolve_service_reference(in string service_id)
    raises (CCMException);

  };

  interface StandardConfigurator : Configurator
  {
    void set_configuration (in ConfigValues descr);

    /* QoS4CCM */
    ConfigValues get_configuration( in CCMObject comp )
        raises (WrongComponentType);

   };

  interface HomeConfiguration : CCMHome
  {
    void set_configurator ( in Configurator cfg);

    /* QoS4CCM */
```

```
        Configurator get_configurator();

        void set_configuration_values ( in ConfigValues config);
        void complete_component_configuration (in boolean b);
        void disable_home_configuration();

  };

};
```

A.4 Interface ExtensionContext

```
module Components {
  local interface ExtensionContext : CCMContext {

    Components::ContainerPortableInterceptor::ClientContainerInterceptorRegistration
    get_client_interceptor_registration ()
      raises (CCMException);

    Components::ContainerPortableInterceptor::ServerContainerInterceptorRegistration
    get_server_interceptor_registration ()
      raises (CCMException);

    Components::ContainerPortableInterceptor::StubContainerInterceptorRegistration
    get_stub_interceptor_registration()
      raises (CCMException);

    Components::ContainerPortableInterceptor::ServantContainerInterceptorRegistration
    get_servant_interceptor_registration()
      raises (CCMException);

    Cookie
    install_service_reference(
      in string service_id, in Object objref)
      raises (CCMException);

    Object
    uninstall_service_reference(in Cookie ck)
      raises (CCMException);

    QoSPropertyRegistration
    get_qos_property_registration()
      raises (CCMException);
  };
};  // end module Components
```

A.5 Interface ExtensionComponent

```
module Components

  local interface ExtensionComponent : EnterpriseComponent {
        void
        set_extension_context (in ExtensionContext ctx)
            raises (CCMException);

        void
        ccm_remove ()
            raises (CCMException);
    };

};
```

A.6 Module QoS

```
module Components {
    module QoS {
        struct QoSInstance {
        string dimension;
        any value;
    };

    typedef sequence<QoSInstance> QoSInstances;

    local interface QoSPropertyInstance {
        attribute readonly string functionality;
        attribute QoSInstances qos_instances;
        void activate( in string operation, in ParameterList parameters);
        void deactivate(  in string operation,
                          in ParameterList parameters);
    };

    local interface QoSPropertyInstanceRegistration {
        Cookie register_qos_property( in QoSPropertyInstance instance );
        void unregister_qos_property( in Cookie ck )
            raises(InvalidRegistration );
    };

        struct QoSConstraint {
            string characteristic;
            QoSInstances instances;
        };

        typedef sequence<QoSConstraint> QoSConstraints;

        interface Negotiation {

            Components::Cookie
            require_qos(
                in QoSConstraint requirements,
                in string client_id)
            raises (CCMException);
            void
            release_qos (in Components::Cookie ck);
        };

        local interface QoSUsage : public ExtensionComponent {
            attribute readonly string characteristic;
            Cookie install_qos_property( in QoSInstances instances )
                raises(CCMException);
            void uninstall_qos_property( in Cookie ck ) ;
        } ;

    }; // end module QoS
}; //end module Components
```

Annex B: Examples

(Non-normative)

This Annex contains examples to demonstrate how this specification facilitates the development of QoS aware CORBA Component based applications. The examples presented here are developed based on the CCM implementation Qedo [Qedo].

B.1 Example: Tracing

This example demonstrates the integration of a specialized tracing functionality into the container. This monitoring simply logs every call that is made between components. As a result, traces between components are produced and can be presented to a human. This may possibly help to identify problems in an application.

The example was developed as a showcase of the COACH project [COACH]. This example only uses the basic interceptors. This is sufficient since the tracing does not need to change anything on the call processing.

The tracing property is an example of a very general service that can be integrated into the container. The concepts defined by this specification offer the possibility to integrate very different tracing approaches. In this case log events are produced and sent to a tracing server. An interactive Web page can query the tracing server later on to present the traces.

The tracing property is applied to every component instance of the assembly. The property is also not directly linked to a limited resource and therefore a negotiation is not needed here. Monitoring can be simply switched on. Of course it should imply some run-time overhead.

The example assembly contains a very simple Hello World application. The interaction between them should be monitored. The general scenario is depicted in the figure below.

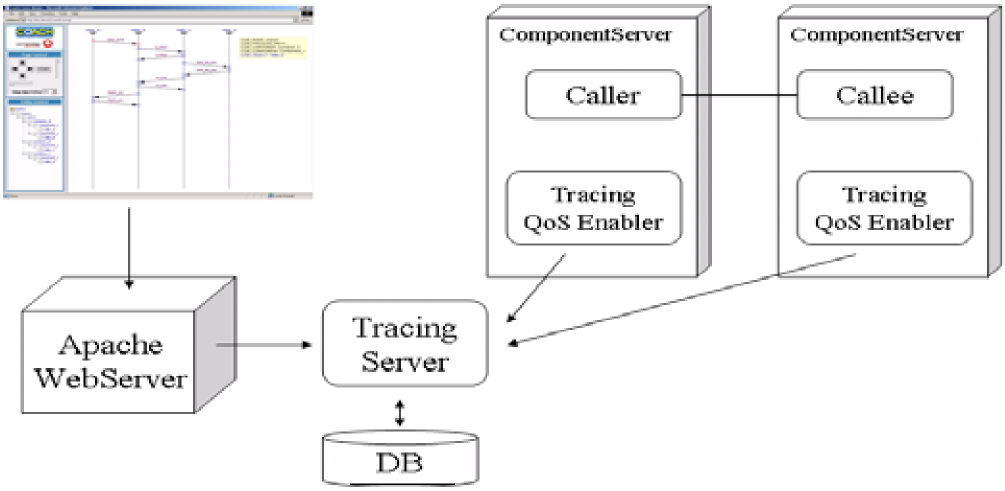

Figure B.1 - Tracing Scenario

B.1.1 Modeling

The first task is to model the example application itself. Since it is a very simple hello world application where one component calls a simple operation on another component, only two component types are needed. These component types are defined by the following IDL fragment.

```
module HelloWorld {

  interface Hello {
    void say ();
  };

  component Callee {
    provides Hello the_hello;
  };

  home CalleeHome manages Callee {};

  component Caller {
    uses Hello hi;
  };

  home CallerHome manages Caller {};
};
```

The same definition can be displayed with a UML2 Profile [UMLCCM].

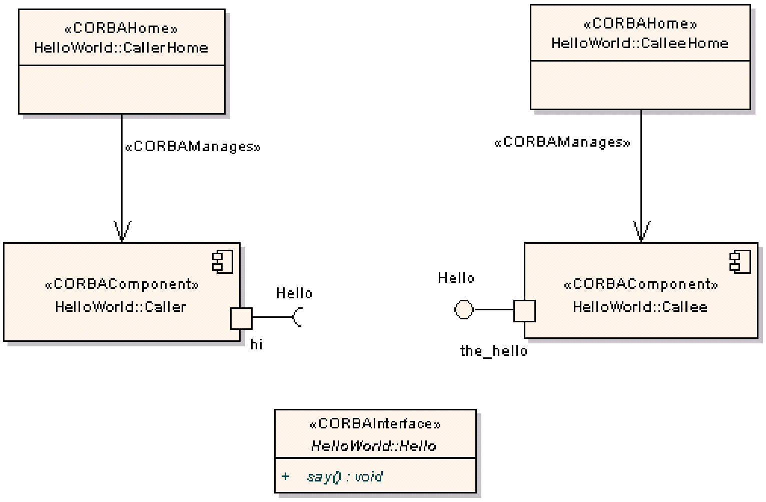

Figure B.2 - Component types of the HelloWorld Example

The next step is the definition of the Tracing QoSCharacteristic. The only dimension this QoSCharacteristic needs to have is the location of the TracingServer. The Tracing Server offers an interface that can be used by Tracing QoSEnablers to send Tracing events to. The reference to the interface of this location is defined as a CORBA NameService name (e.g., “Services/TracingService”). Taking this into account the definition of the QoSCharacteristic Tracing could look as follows.

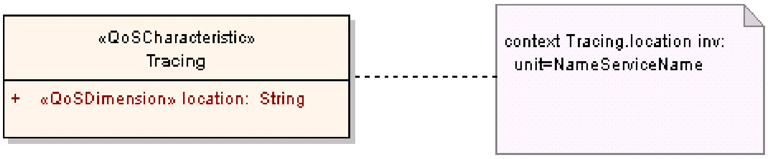
«QoSCharacteristic»
Tracing
+ «QoSDimension» location: String
context Tracing.location inv:
unit=NameServiceName

Figure B.3 - QoSCharacteristic Tracing

Finally the assembly needs to be defined. In this example only one instance of the client component and one instance of the server component is needed. Furthermore, the constraint for the Tracing Characteristics needs to be defined. Since this is not related to a negotiation no specific Constraint such as QoSoffered of QoSRequired needs to be used.

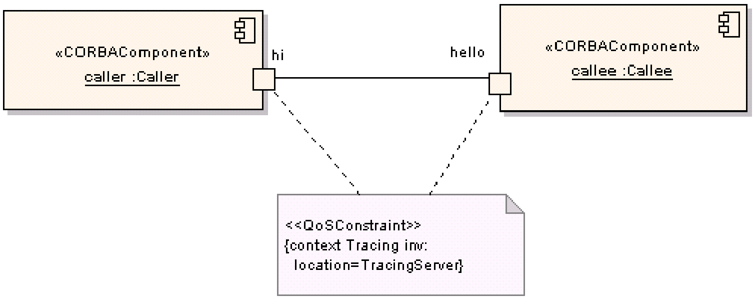

Figure B.4 - QoSConstraint Tracing for Hello World

B.2 Example: Processing Throughput

This example is an implementation of the Throughput Characteristics as defined by the UML Profile for QoS. This characteristic is applied to a simple computation scenario. One Server component instance offers the service to compute something (operation compute) multiple client component instances uses this service. But at least one of them has a specific QoS requirement. It requires to be able to make a certain number of calls in a specific time interval.

In this example negotiation is used. Because it cannot be determined before deployment that such requirements can be fulfilled at run-time. It could be possible that other requirements need to be fulfilled.

B.2.1 Modeling

In this example two component types are used, one component type for the client side and one for the server side. While the server offers a port and the client requires a port of the same type. The following definitions apply.

```
module Computation {

    interface Computing_Service {
        long compute (in long argument_of_function);
    };

    component Client {
        uses Computing_Service computing_server;
    };

    home ClientHome manages Client {};

    component Server {
        provides Computing_Service computing_interface;
    };

    home ServerHome manages Server {};
};
```

For illustrational purposes the same types are displayed as UML2 models. Using a UML2 Profile for CCM [UMLCCM].

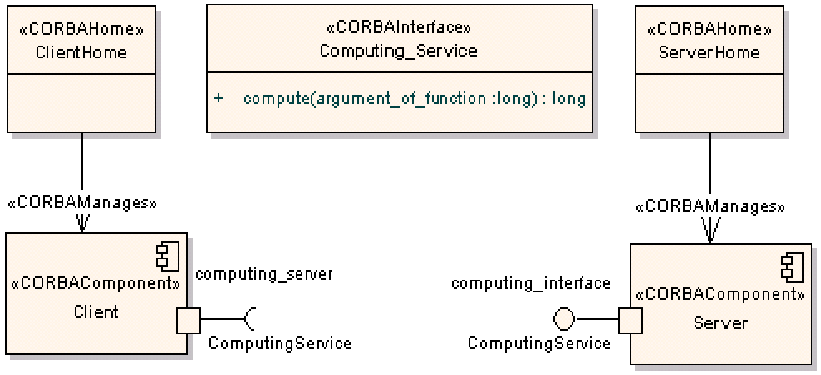

Figure B.5 - Component Types of Computation Example

The UML Profile for QoS defines the abstract QoS Characteristics Throughput. This characteristic has one dimension rate. The Unit of rate is not defined since it is an abstract characteristic. Furthermore, it has an interval of observation, where the rate is averaged.

The following diagram represents the definition of the processing throughput characteristic. This diagram is based on [UMLQOS].

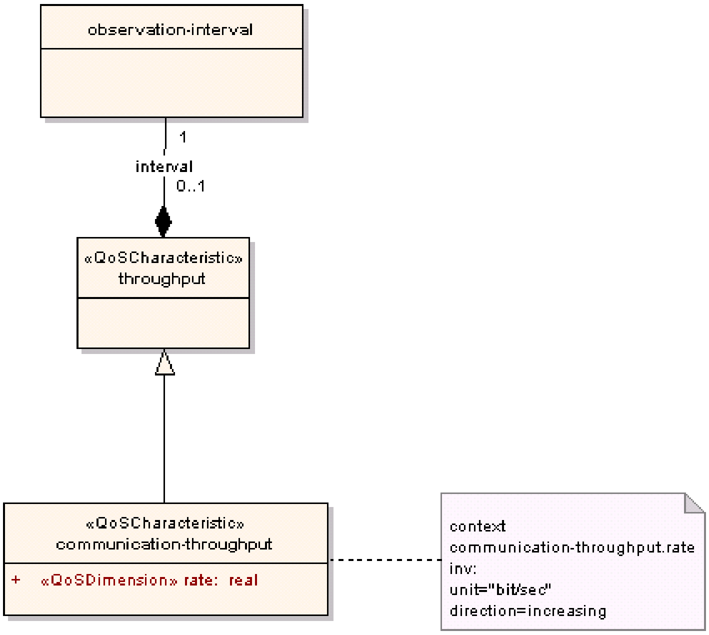

Figure B.6 - QoS Characteristics Communication Throughput

The next step is the definition of a QoS constraint base on the processing-throughput characteristic. This constraint defines the allowed values of the characteristic. In this case the specific client might have the requirement to make 3 calls per second averaged over an interval of 2 seconds. The definition of such a requirement in UML is depicted below. Furthermore it has to be expressed that the server instance is capable of providing control over the processing-throughput.

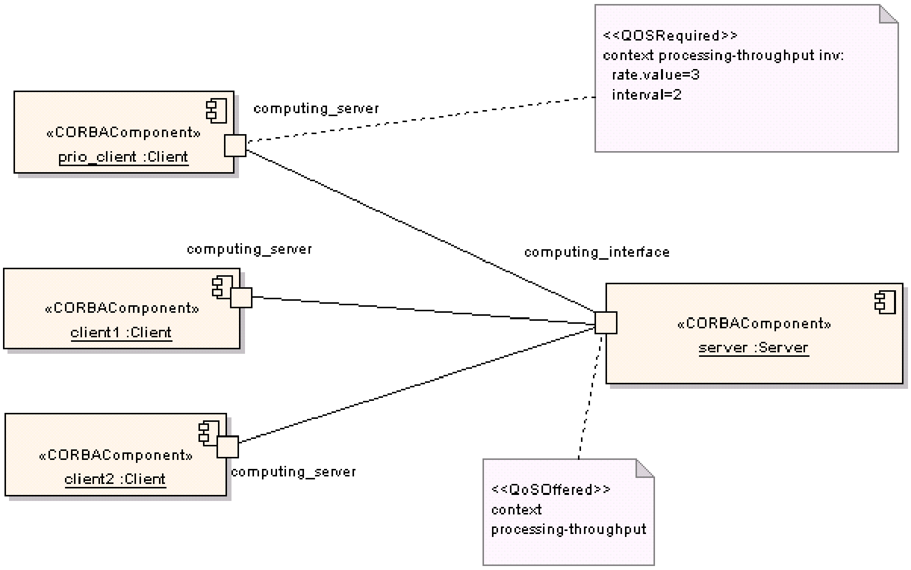

Figure B.7 - QoSConstraints for Computation example

B.3 Example: Encryption

This example describes a way to encrypt specific parameters and return values of calls respectively. The idea behind this is to encrypt the parameter of a method invocation at the client side and to decrypt this parameter at the receiving side. This functionality can be achieved by using underlying security mechanisms like SSL as well. But in this specific solution, only a very specific portion of information is protected, while all other information remains unprotected.

This example uses the same functional example as the Processing Throughput example, which is the compute example. Multiple client request a service from a server component. To protect the result of the computation conducted by the server component from getting read by unintended parties it is encrypted with a specific algorithm. On the receiving side the result is decrypted again and delivered to the client component.

B.3.1 Modeling

First of all the definition of a QoS characteristic needs to be done. In this case it is fairly simple since it is only needed to switch encryption on or off. It is not necessary to define further QoSDimensions for this QoSCharacteristic.

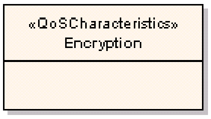

Figure B.8 - Encryption QoS Characteristics

In this example two instances of the client component are connected to one instance of a server component.

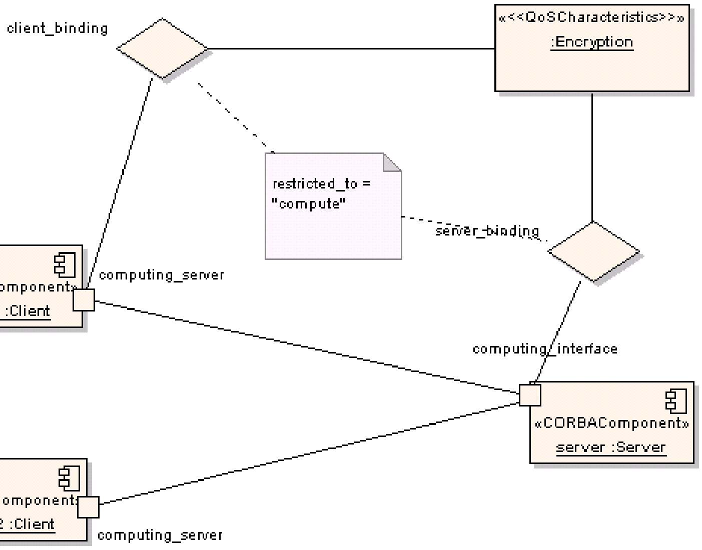

Figure B.9 - QoS Constraint Encryption for Computation Example

INDEX