UNIVERSITY OF OSLO
Department of Informatics

Service Planning in a QoS-aware Component Architecture

Cand Scient thesis

Øyvind Matheson Wergeland

15th April 2007

Preface

This thesis has taken me much more time to complete than I ever anticipated. I have to thank my supervisor Frank Eliassen for being more patient than I would have assumed. He has taught me that researchers tend to stand upon the toes of other researches, when they should be standing on their shoulders. I indeed hope that I have managed to climb some shoulders in this thesis.

I also have to thank my co-student, friend, and former colleague Tore Engvig for the mutual support and all the fun and interesting time we have spent together in the QuA arena.

Several of the people of in the QuA project has taught and helped me a lot over the years. Special thanks go to Richard Staehli, Hans Rune Rafaelsen, and Eli Gjørven.

Finally I have to thank my ever-supporting wife for pushing me to finish this work.

Oslo, April 2007.

Table of contents

List of figures

List of tables

1. Introduction

1.1. Background

It is commonly agreed that building modern computer systems is a complex task. Sommerville says that even a simple system has "high inherent complexity" (Sommerville 1995, p. v). Abstraction and divide and conquer are the main strategies to cope with this complexity. Component based software engineering (CBSE) offers both strategies; framework mechanisms as persistence, distribution, and transactions can be separated into a component platform and "abstracted away" from the business logic in the components. The components themselves also offer abstractions through encapsulation, as well as being an intuitive target for separation of various business functions. In addition, CBSE promotes reuse, primarily of the framework mechanisms through reusing component platforms, but to some degree also from reusing component implementations.

Abstraction takes other forms as well; an object-oriented design provides encapsulation that effectively "abstracts away" the implementation of the provided functionality. The ordering of functionality in architectural layers abstracts away both the implementation of the layer below the current layer as well as the functionality of the levels further down (see Figure 2 on page 14 for an example of architectural layers).

Another important observation is that many applications share a lot of common functionality. When this shared functionality, also shares the same abstract interfaces, it can be generalized and implemented. The common functionality may often be hard to implement correct, so using a well-known and well-tested implementation saves a lot of development time and cost. Current state of the art is to use CBSE in an attempt to profit from software re-use (Szyperski 2002).

1.1.1. Components and services

Current industry architectures for open distributed systems are often called "service oriented architectures" (Papazoglou and Georgakopoulos 2003). In such architectures, there are usually three distinct entities (see Figure 1):

Service producers offer services that are described in an implementation neutral way.

Service consumers have a need for certain services to fulfill their responsibilities, and consume services from the producers.

Service directories are used by service producers that want to register their services for lookup by service consumers.

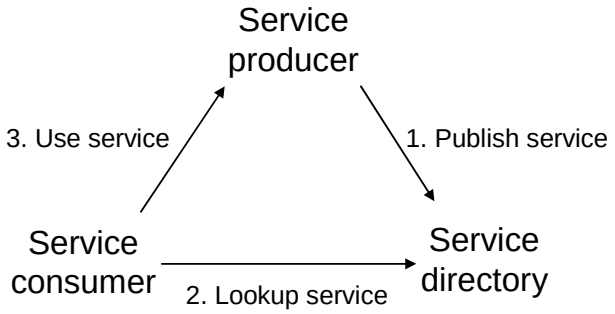

Figure 1: Service oriented architecture

Components and services pair up well; component instances provide services to its environment. And both components and services need to have their interfaces formally

described using some interface description language (IDL). E.g., Web Services may be described using an IDL called Web Service Description Language (WSDL).

A major difference between acquiring a component and a service is the market model. Acquiring a component means that a piece of software is made available for the acquirer and it can then typically be used indefinitely without any additional cost. Using a service is more likely to have a per-execution cost model.

When acquiring a component, the acquirer must provide the necessary resources so the component can execute with the expected quality of service (QoS), while service providers are responsible for the resource availability.

Services may depend on other services to execute, in addition to depend on resources. A *service aggregator* may provide a *managed service* that is composed of *basic services* (Papazoglou and Georgakopoulos 2003). Underlying services as well as resources may be acquired from other parties. Both the functionality and the QoS of the managed service are aggregated. The task of composing services is similar to the task of composing with components, but the composed service is envisioned with a very loose coupling where the composition may be used only once (Gold et al. 2004).

1.2. Problem area

Software components that provide services have certain qualities. There are many models that can be used to describe software qualities, of which ISO 9126 is one. ISO 9126 lists the following main *quality characteristics*:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

These characteristics are further divided into sub-characteristics (for a complete list of sub-characteristics, see Appendix A). Some of these characteristics are static once the component has been implemented, such as its usability, maintainability and portability compliance. Other quality characteristics may be configurable and even dynamically reconfigurable during execution, such as accuracy and security. Sub-characteristics may be broken further down in a recursive manner.

Other terms for these quality characteristics are *non-functional* or *extra-functional*. Some prefers the latter term because these characteristics may be implemented as functions in the software that implement the service, they are just abstracted away from the main functional interface. Quality characteristics may be organized in a hierarchical tree, where a higher-level characteristics consists of some lower level characteristics. Consider a characteristic *audio quality* could refer to the combination of the characteristics *sample rate* and *sample size*. Here, the audio quality level “CD sound” could refer to a sample rate of 44.1 kHz and sample size of 16 bits. Quality characteristics are sometimes also called *quality dimensions*, but some use this term is to refer only to the lower level characteristics.

The quality of an executing service is called *quality of service*, or just *QoS*. The quality of a service depends on the current quality of the different underlying quality dimensions that are relevant for the service. It is not straightforward to measure QoS, as what is the preferred quality depends on the context of the service. For example, the delay may be more important than the signal-to-noise ratio for an audio service in a phone conversation, and vice versa in a

radio broadcast. Also, the metrics used to measure QoS may be subjective, as poor, acceptable, and good usability.

The main problem when trying to achieve high QoS is that the execution environment is not known before the execution time of a service. Some of the parameters may be known, or rather presumed, at development time, but only to a limited extent. Examples of such parameters are the availability of an IP-based network, but not the bandwidth, and a minimum of available RAM, but not the maximum. This problem does not apply to embedded systems, where the environment may be specified exactly.

We say that a system is *QoS-aware* if it is able to trade quality in one dimension with another, e.g., delay with signal-to-noise ratio.

As stated in section 1.1, component-based software engineering is believed to be an effective way to create software systems. The effectiveness is supposed to arise from the ease of reusing already developed functionality, which is made available as components.

Component-based systems should be QoS-aware so that the resources shared among the different component instances running in the systems are utilized not only fairly, but in an optimized manner, depending on the preferences of the service consumers. Sharing resources between concurrent services can be achieved by giving the component platform or framework responsibility to *plan* the services, called *service planning* in this thesis.

For a platform to be able to plan QoS-aware services, it needs an extendible QoS model, as it is impossible to foresee the QoS domains of future services.

1.2.1. QuA

This thesis is written in the context of the Quality Architecture (QuA) project. QuA provides an architecture for a QoS-aware component model, but at the early stage of the work, only a Smalltalk prototype which implemented the basic component model existed, and it did not include any QoS-awareness. Also, the QoS related interfaces where underspecified.

In QuA, service planning is the process of choosing component implementations for a service composition (Staehli and Eliassen 2002), and allocating resources to the implementation instances in a way that maximizes the quality of the provided service, limited by the available resources at the time the service is planned.

The QuA platform is motivated by research in reflective middleware and dynamic reconfiguration, to make it possible for any part of the system to inspect and reconfigure the platform and running services (Staehli and Eliassen 2002).

1.2.2. Specific problem statement

With component implementations on one side, and the operating environment on the other, the first part of the service planning process is to choose the composition and implementations of components to satisfy a service consumer. The problem is to enable the middleware to choose the best available component implementation for given QoS requirement and available resources. As the task of choosing component implementations is most likely to be a common one for the middleware, it would be preferable if the logic for this can be reused across different service types or domains

A straightforward solution to the problem is most likely to be inefficient, as the search space grows rapidly with each implementation and quality dimension to consider. Providing a generic solution that also is efficient for all variations of the problem is not very likely, but a generic solution should at least be efficient enough to be usable for some use cases.

The problem statement can be summarized as follows:

> *How can a component middleware efficiently choose a component implementation based on component type, QoS requirements and available resources?*

1.3. Goal

The goal is to implement a version of the QuA architecture including its service planner component framework. The service planner framework consists of a *composition planner*, *binding planner*, and an *implementation planner*, which cooperate to locate, instantiate, and bind together the component implementations needed to provide a given service. The implementation will be made QoS-aware by including a pluggable *Generic Implementation Planner*. This implementation planner should be able to choose between different available implementations based on a client's QoS requirements and available resources at planning time.

The validity and effectiveness of such a generic planner should be verified using experimental simulations. Both the correctness of the choice and the scalability with respect to the number of implementations and number of quality dimensions specified by the client should be investigated.

An implementation should be selected and configured within one or a few seconds on a typical personal computer per component type. The configuration should have the highest possible utility value with the available resources, or at least very close to this value.

This goal is in line with the second sub goal of QuA, which is "to develop, prototype and experimentally evaluate a QoS aware component architecture with platform managed QoS" (Eliassen et al. 2002, p. 4).

1.3.1. What is not covered

QoS is a large area, and it is not possible to consider all elements in a single thesis. More specifically, several elements are taken for granted in this work, as QoS negotiation, resource reservation and monitoring, QoS policing, QoS monitoring, and adaptation to maintain agreed QoS.

QoS negotiation is the process where the service consumer and service provider negotiate the quality level of the service, possibly involving the cost for the service. The result of such a negotiation is a *contract*.

Reservation and/or monitoring of resources is assumed to be available to the component platform and handled by a *resource manager*. Note that resource reservation is not required, but then QoS adaptation should be in place to allow services to degrade gracefully.

Policing means that the system enforces the client to behave as agreed in the contract from the QoS negotiation. Without policing, any client not keeping to its contract could break the system by flooding it with requests or not releasing resources when supposed to.

Monitoring the actual QoS level, and adaptation of executing services to maintain the agreed quality level, are considered beyond the scope of this thesis.

1.4. Method

This thesis will present a hypothesis for service planning in a QoS-aware component architecture, and test the hypothesis with a prototype for such an architecture, including a possible model for a part of the service planning process.

The hypothesis will be tested by running and analyzing experiments on the provided prototype.

1.5. Result

This thesis will present an implementation of a QuA Java capsule with a Generic Implementation Planner. The Generic Implementation Planner can be applied to select and configure any QoS-aware component for a limited QoS domain, which is a viable solution to the stated problem.

1.6. Overview of the rest of this thesis

Chapter 2 presents the technical background and related work for this thesis, and the QuA architecture is presented in chapter 3. The problem area is described in detail in chapter 4. Chapter 5 contains an analysis of the problem area and a solution proposal, and describes two experiments to evaluate the solution proposal. The design and implementation of the solution proposal is presented in chapter 6, and chapter 7 contains the experiment results. The final chapter 8 contains the evaluation and conclusion of the work.

2. Background and related work

This chapter provides an overview of the most relevant technologies for this thesis. Section 2.1 briefly describes well-known architectures from distributed computing, section 2.2 shows the current industrial standards, and finally in section 2.3 other research projects are discussed.

2.1. Background

2.1.1. Middleware

Middleware is a typical example of software reuse that has emerged from the distributed systems area. The idea is to create reusable communication modules, and hide the fact that the system is distributed over different nodes in a network. There are two major benefits from this approach; the first is that communication is tedious and error-prone to design and implement, so it is very efficient to re-use an existing communication module. The other is that the programming model for the distributed system becomes similar to a non-distributed program, also called *location transparency*. However, as will be pointed out later, location transparency may not always be desired.

There are middleware implementations for distribution mechanisms as remote procedure calls (RPC) for procedural systems, and remote method invocations (RMI) for object-oriented systems. The term middleware points to the placement of this functionality in the middle between the application layer and the operating system layer in the system architecture, as shown in Figure 2. Middleware itself can be further divided into sub-layers, as common middleware services and domain specific services (Schmidt 2002).

There exist numerous middleware specifications and implementations, e.g., Sun RPC, OMG's CORBA, Java RMI, and Microsoft DCOM.

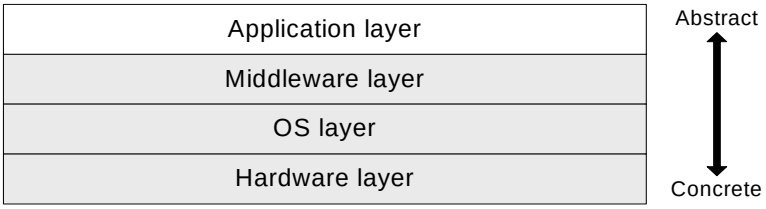

Figure 2: A layered architecture

2.1.2. Components

Component-based software engineering is another approach for providing common mechanisms to applications, where the system is (at least partially) implemented as components that are wrapped by containers. The containers may intercept calls and perform extra functions before the call is forwarded to the actual component. Functionality provided by the container may be authorization of the caller, transaction management, and trace logging.

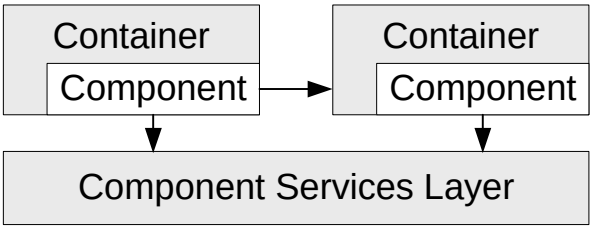

Figure 3: A component architecture

The rules that govern the interaction of components and containers are described in a *component model*. The domain of component models can be divided in two; graphical

components and enterprise components. Graphical components are used for building graphical user interfaces (GUI), and the probably best known model for this is Microsoft's object linking and embedding (OLE) model (Szyperski 2002).

In the enterprise domain, there are several commercial component models as well as research component models. Some of these models are interoperable, as Sun's Enterprise JavaBean (EJB) model and OMG's CORBA Component Model (see section 2.2 below for more on industrial standards).

There are written (and said) much about components in the recent years. Szyperski (2002) is probably the most cited reference for a definition of components, not least because of many disagreements over the phrasing of the definition in the first edition of the book. He defines that a software component has contractually (i.e., formally) specified interfaces and explicit context dependencies only. A component that adheres to the definition can then be deployed and used in compositions by third parties. On the implementation side, neither of the requirements is usually completely supported, as it is hard to specify interface semantics formally, and likewise to specify all explicit context requirements, e.g., the needed stack size for a component that is implemented using a recursive algorithm. On the other hand, it is hard to verify the semantics of an implementation, and it only makes sense to specify context requirements that the platform has mechanisms to handle.

Szyperski (ibid.) does not specify what makes up a component, but requirements such as polymorphic behavior suggest that object-oriented languages are preferred for component development.

A component model is implemented by a *component platform*. A (typical higher-level) component platform may also be built using an existing component model, as OpenCOM, which is implemented using (a subset of) Microsoft COM (Blair et al. 2001). Another example of this is Microsoft OLE which is a superset of Microsoft COM (Szyperski 2002).

Szyperski (ibid.) also advocates the use of *component frameworks*. A component framework is a micro-architecture with a specific area of concern. The framework specifies the contracts for components that are plugged into the framework, thus making the framework configurable. Open ORB 2 (see 2.3.3 below) is a very good example of how component frameworks can be recursively packaged as components and then define dependencies between the component frameworks in the same way as dependencies between single components (Coulson et al. 2002).

2.2. Industrial component standards

Of today's component standards, the following three are important to look into; Microsoft COM for its widespread use on the Microsoft Windows platform, Sun's Enterprise JavaBeans which is heavily used in enterprise systems, and the CORBA Component Model, which is a good reference model for component models.

2.2.1. CORBA/CCM

OMG's Common Object Request Broker Architecture (CORBA), and the CORBA Component Model (CCM) is a programming-language neutral approach to distributed systems and components (Szyperski 2002). Today CORBA is viewed as a platform itself, although CORBA ORBs and services are implemented in several programming languages.

CCM is an interesting model to look at as a reference model, as it is explicit in some of its external dependencies. In CCM, a component defines both its required and provided interfaces, called receptacles and facets, respectively, as well as provided and required event sinks and sources. Figure 4 shows an example CORBA component, where the provided ports are shown on the left hand side, and the required ports on the right.

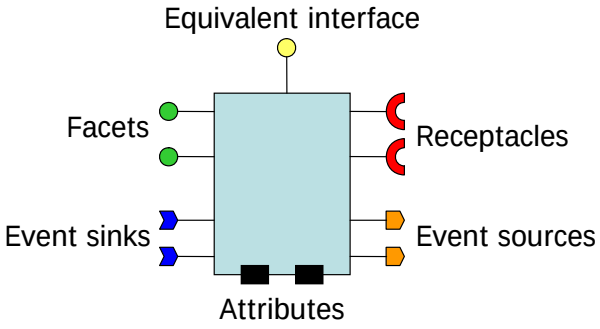

Figure 4: CORBA 3 component overview (based on a figure from Szyperski (2002))

In addition, a CORBA component must implement a special interface, called the *Equivalent interface*. This interface is used by clients programmatically to discover the other interfaces that are provided by the component. Components may also contain attributes, accessed and changed through method calls.

Component implementations may contain implementations for several platforms in the same binary bundle. This is a neat way for a component vendor to ship components, but may make the deployment process on limited devices such as a mobile phone, a bit cumbersome, because the bundle must be stripped of unnecessary implementations before deployed to the device.

Which component implementations to use in a system are chosen by *component assemblers* (Schmidt and Vinoski 2004).

The model only makes dependencies to other components explicit, and not other kind of dependencies, such as resources the component requires to execute. It is also possible for a component to call other CORBA objects that are not listed as receptacles, thus breaking the explicit dependency rule.

QoS is not a part of CCM, but OMG also provides a specification for *Real-time CORBA* (RT-CORBA). An RT-CORBA implementation allows control of some system resources to distributed applications (Schmidt and Vinoski 2001). Management is limited to processing, communication, and memory resources. However, Wang, Balasubramanian, and Gill (2002) argue that it is not sufficient to run CCM on top of RT-CORBA, and that CCM must be extended with QoS mechanisms.

2.2.2. EJB

Enterprise JavaBean (EJB) is Java's component model for distributed systems (Sun Microsystems 2003). In EJB, access to components, or beans, and also between beans, are always through the container. Extra-functional services offered by the container are distribution, transaction management and security. In addition, the platform also offers connectivity services such as naming service, database access and asynchronous messaging.

The EJB container runs within an EJB server. The different EJB server implementations compete on providing additional extra-functional properties as scalability, load balancing and reliability (i.e., failover).

EJB specifies that each bean must provide interfaces for remote and/or local lookup, and remote and/or local interfaces for method access, as well as a deployment descriptor. The deployment descriptor may declare transaction and security requirements. During deployment, the container generates the code that implements the interfaces to be exposed by the container, including transaction and security management.

The developer of an EJB component specifies any required interfaces by listing their names in the bean's deployment descriptor. During deployment, these names must be resolved to the

actual names of their deployed implementations. There is no automatic type checking at deployment time, so the correctness resolving process is left completely to the component deployer. However, this indirection could allow for postponing the resolving until runtime using an implementation broker or trader.

Choosing an implementation is, as in CORBA, executed using the naming service. If an implementation does not implement the assumed interface, the client will receive an exception. It is not specified how beans can be updated runtime, but this is possible at least in theory as long as the exposed interfaces are kept unchanged, and the container directs all new method calls to the implementation, and dereference the old when it is no longer used.

2.2.3. COM/DCOM/COM+

Microsoft COM is a foundation for several models. It is defined in terms of interfaces and binary interoperability (Szyperski 2002), so a component can be implemented in any language that compiles to such a binary. All COM interfaces must provide the method *QueryInterface* as its first method, which can be used to discover if a component implements a given interface. Also, all COM components must implement a special interface called *IUnknown* that can be used to get access to the QueryInterface method.

The COM platform offers some services to the components running on it, as persistent storage of the state of an instance, uniform data transfer (typically used in a GUI environment for clipboards etc.), and a transaction server is also available.

Distributed COM (DCOM) adds distribution transparency to Microsoft COM. COM+ is a newer version of Microsoft COM. Some of the services from COM are reimplemented in COM+, such as transactional processing. COM+ also offers asynchronous messages and running components in a load-balanced cluster.

2.3. Research projects

In the research area, several projects are investigating reflection as a promising approach for dynamically reconfigurable systems (Kon et al. 2002). This section contains an overview of reflection and some projects which has made major contributions to the field.

The projects that are described here addresses the problem area either by introducing QoS monitoring and/or management mechanisms to component systems, or, in case of the Q-RAM project (Rajkumar et al. 1997), provide an algorithm for maximizing QoS.

There exists numerous research projects in the area, as both researching component models and QoS are of increasing popularity, and it is not possible to cover all such projects. A recent project, COMQUAD (Göbel et al. 2004), researches how aspect oriented programming (AOP) can be used to implement non-functional properties in component models. There are also projects in the mobile computing domain researching component models such as SATIN (Zachariadis and Mascolo 2003), and CASA (Mukhija and Glinz 2004). The former paper briefly describes a generic framework for distributing mobile code and capabilities, while the latter describes a framework where each application is dynamically reconfigurable, but an application is limited to a single node, and consumes services provided by other, remote application. For a more extensive list of projects, see the technical report by McKinley et al. (2004b)

2.3.1. Reflection

Reflection is a system's capability to inspect, evaluate and change itself (Kon et al. 2002). Change through reflection is also called *intercession* (McKinley et al. 2004a).

Reflection is split into *structural* reflection and *behavioral* reflection. Structural reflection is related to the structures in the system, such as class hierarchies, type systems, and state information. Behavioral reflection is defined by Chiba (2000) as "the ability to intercept an operation such as method invocation and alter the behavior of that operation". The *dynamic*

proxy mechanism in Java is a form of behavioral reflection, but it is a weak form because the existing object is not changed, i.e., the change of behavior is only seen by client objects.

The *Equivalent* interface in CCM (see 2.2.1 above) is an example of structural reflection in middleware. Another example is the portable interceptor specification, which provides behavioral reflection to CORBA.

Interfaces such as *IUnknown* and *Equivalent* are called *meta interfaces*, as they give access to meta information about the component. Components may also be represented using meta classes, similar to the class *java.lang.Class* in Java, which can be used to access meta information for a specific class.

Some systems may provide limited reflection capabilities, for example only allowing inspection and not changing the system. Java provides limited reflection by allowing full access to inspection – called introspection – of the type hierarchy, methods and fields, but only fields can be changed.

The evaluation part of reflection, i.e., the system's capability to determine when and how it should change, is naturally limited by the logic implemented within the system. Such logic is almost always limited by a programmer's knowledge (or guess) of what changes are possible. To be able to implement an extensible reflective system, this logic must be pluggable.

2.3.2. dynamicTAO

Kon et al. (2000) describes a reflective and dynamically reconfigurable ORB based on The ACE ORB (TAO). TAO is a statically configurable CORBA ORB, where which services and implementations that are going to be available at runtime, are specified at deployment time. TAO aims at embedded systems where the resources are known in advance. dynamicTAO recognizes the fact that resources vary in space (from computer to computer) and time (on the same computer), and claims that existing middleware systems does not support these variations. As the variations are increasing, a new generation of middleware is needed that support dynamic reconfiguration based on the variations in the system's environment.

Their proposed solution is to use reflection to support both resource monitoring and dynamic reconfiguration. Reflection is implemented using *component configurators* that reify the structure of the ORB and contain the dependencies between the various components. The leaves of this structure are the reconfigurable *categories* as concurrency, security, monitoring, etc. To support dynamic reconfiguration, dynamicTAO implements the strategy pattern (Gamma et al. 1995) to select and change the strategy to use for each category. The strategies are implemented as components, and each component may declare external dependencies to other implementations.

dynamicTAO also provides mechanisms that avoid inconsistent configurations of the ORB. This is necessary since using a reference to an unloaded component may cause not only the caller to fail, but the entire ORB process. Such mechanisms are important when designing systems that are supposed to run over long periods of time.

Since TAO is implemented in C++, dynamicTAO also needed to provide an implementation for dynamic loading and linking. This support is extended from existing capabilities in the Adaptive Communication Environment (ACE).

Reconfiguration is controlled by user level APIs and applications, and is not managed by the ORB itself. It is even possible to store and delete components in a persistent repository during runtime with this API, but there is no meta class for components.

Interceptor components are used to monitor the performance of an ORB instance. The monitor components are loaded and unloaded in the same way as strategy components. The information collected by the monitor is then available for clients, which can use this information to reconfigure the system.

2.3.3. Open ORB 2

Open ORB 2 is another ORB that also is reflective and dynamically reconfigurable.

Open ORB 2 connects component technology with middleware's need to be configurable and dynamically reconfigurable. Configurable means that the same middleware platform can be deployed in different environments with different resources and requirements, while dynamically reconfigurable means that the services running in the middleware, and even the middleware itself, can be reconfigured during runtime (Coulson et al. 2002).

Open ORB 2 is built upon components in a reflective component model, called OpenCOM. OpenCOM is implemented using a subset of Microsoft COM, and there is also an implementation available based on Mozilla XPCOM (Cleetus 2004).

The OpenCOM component model provides three meta models for reflection purposes, the interface meta model, the architecture meta model, and the interception meta model. These meta models are accessed through meta interfaces available in all OpenCOM components.

The interface meta model allows for inspection of the interfaces and receptacles in a component, while the interception meta model allows programmatically insertion and removal of interceptors. These interceptors are run before and/or after the actual method is invoked, but not instead of the invocation. The architecture meta model is fully reflective as it allows both inspection of, and changing of, the coupling of components. Since the meta models also are implemented with OpenCOM components, they also provide reflection. To avoid infinite recursion of meta components, a meta component is not instantiated before it is accessed.

To realize the ORB implementation, a set of component frameworks with responsibility of different concerns are provided. The component frameworks are arranged in layers, as well as being implemented as composite components. A component framework manages the components that are plugged into it, and a higher-level component framework manages the lower-level component frameworks that are plugged into it in the same way, since they are exposed just as components.

This architecture allows for adaptation of the ORB in almost any possible way, where each component framework can be changed through its meta interfaces. To avoid adapting the ORB to an inconsistent state, the component frameworks are composed into a top-level component framework which contains *layer-composition policies* that governs adaptation.

Resource management is handled through a resource model, called a meta-model in Coulson et al. (2002). This model contains resource abstractions and resource managers for different resource types.

Coulson et al. (2002) argue that a major limitation with other middleware platforms is that only basic binding types (remote method invocations, media streams and event handling) are supported by existing middleware, and richer binding types are provided ad-hoc and without explicit dependencies. Therefore, Open ORB 2 provides an extensive binding component framework.

The binding component framework only manages which binding types that will be provided. Adaptation of a binding is managed by the binding type implementation. Binding types are dynamically loaded the first time a binding of that type is requested.

Composition can be adapted through structural reflection using the architecture meta object. To avoid breaking the architecture, Open ORB 2 provides different architectural constraints (Blair et al. 2001), one of which is the type system. In addition, layer-composition policies are realized as components that allow or disallow composition changes (Coulson et al. 2002).

In ReMMoC (Grace, Blair, and Samuel 2003), component frameworks are extended by including a receptacle *IAccept*. The interface is used to govern the reconfiguration policies of that component framework.

2.3.4. Quality Objects

Quality Objects (QuO) is described in (Loyall et al. 1998; Schantz et al. 2002).

QuO identifies the need to separate implementing an application's functional aspects from implementing its QoS aspects, as well as the need to reuse implemented QoS aspects. They reason that developers of distributed object systems that need QoS support bypass the middleware as it lacks the necessary support.

The QuO model extends the CORBA model with QoS support, and at the core of the implementation is a specialized ORB. QoS aspects are divided into *contracts*, *system condition objects*, *callback mechanisms*, object *delegates*, and *mechanism managers*.

System condition objects are used for two purposes; to monitor parts of the system state, and to let the client control desired (but not necessarily provided) service level. It can be argued that the latter should be provided by separate control interfaces. This would provide a cleaner model where additional features such as admission to increase the service level could be controlled by the middleware.

QuO contracts define operating regions with different service levels and transitions between these regions. A region consists of a predicate and possible nested sub-region. The predicate must evaluate to true for the region to become active. Only one region can be active at each level of nesting. In their examples, the top level regions are connected with the system condition objects that the client controls (e.g., client requests an RSVP connection); while the sub-regions are connected to system condition objects that monitors the state of QoS aspects (e.g., RSVP connections are not available in the underlying network).

A QuO contract is similar to a state machine that defines which output signals (callbacks) are to be generated upon which input signals (system condition monitors). QuO does not provide a mechanism for negotiation of such contracts. Also, a QuO contract quickly becomes god-objects (Brown et al. 1998) in the sense of services, as the contract must know the entire service composition to be able to specify the possible adaptations. Thus, QuO contracts may be good for small services or parts of a more complex service, but insufficient for a large and complex service.

The QoS developer – also called *Qoskateer* (Schantz et al. 2002) – that is QoS-enabling an object must include an object delegate that is responsible for in-bound QoS management as contract evaluation during method invocation. The delegate must provide the same interface as the remote object, but is local to the client[1].

The client Qoskateer must provide the necessary callback objects to get signals from the middleware when adaptation requires the client to change its behavior.

Generic QoS properties as availability and real-time data streams are handled by mechanism managers – also called *property managers*.

An important observation by QuO, is that QoS relevant information is bound at different times, namely development time, configuration time, initialization time, negotiation time, and reality time (usually called runtime). Any QoS-aware system will have to accumulate all this information to be able execute its services properly.

QuO shows where and how QoS mechanisms for monitoring and adaptation can be plugged into a middleware platform, given that the platform allows for this; i.e. it must be an open platform. Still, QoS negotiation and policing is missing. It could be possible to support policing by providing standard system condition objects for the supported QoS properties. The ORB could then connect all services requiring the same property to the corresponding system condition object. It seems to be harder to extend QuO with negotiation, as the contracts – which are the subject of negotiation – are highly specialized.

[1] Several of the figures show a delegate also on the server side. This may indeed be useful for monitoring and controlling QoS, especially when an object is shared by several clients.

2.3.5. QoS for EJB

Miguel, Ruiz, and García-Valls (2002) describes a project where the EJB component model is extended with QoS. At the heart of the model is what they call the *resource consuming component* (RCC). Their logical model is based on CCM, where the RCC has facets, receptacles, message sinks, and messages sources, but the implementation is limited to synchronous calls, i.e. facets and receptacles, as it is based on the EJB 1.1 specification. (Message-driven beans that support asynchronous calls did not appear until the EJB 2.0 specification.) The project extends the EJB model with a new component type called *QoSBean*. The QoSBean interface is designed to support both session and entity component types.

To provide the requested QoS, the RCC can require resources and a certain quality on other components that it depends on. The QoS negotiation is based on the reservation scheme, and two algorithms are provided. The RCC container can perform the QoS negotiation process on behalf of the component, but the component can also customize the process, and a QoS context object is available to the component to help implement the customization.

The implementation seems to be limited to QoS negotiation and admission control, and neither policing nor adaptation is supported. Admission control, reservation and scheduling is handled by a *Resource Manager*, while a *QoS Manager* handles distribution of resources between different concurrent services. While the tasks connected to the Resource Manager are thoroughly discussed, the QoS Manager is not explained, neither is it discussed how the system can utilize the available resources in an optimal way.

The QoS for EJB project shows that component models can provide a clear separation of functionality and QoS. Their QoS model is within the assumption/guarantee paradigm, where an RCC can require (assume) minimum resource availability by reservation, and require minimum quality levels of the components it depends on, to provide (guarantee) a certain quality level.

2.3.6. Q-RAM

Rajkumar et al. (1997) presents an analytical approach for resource allocation to a QoS-aware system. Their task is to maximize *the total system utility*, which is defined as the sum of the utility of all the services executing concurrently in a system.

The Q-RAM model takes into account both QoS dimensions that are independent and dependent. Q-RAM defines dependent dimensions as follows: “A QoS dimension, Q_a, is said to be dependent on another dimension, Q_b, if a change along the dimension Q_b will increase the resource demands to achieve the quality level previously achieved along Q_a.”. Their example of this is a system which incorporates an audio stream, where an increase in the sample rate will demand an increase of the CPU consumption, so the encoding process can be able to provide the same level of quality as before the sample rate was increased. It is not pointed out that such dependencies may be the result of the implementation, but once they exist, the system must take the dependencies into account when allocation resources to QoS dimensions.

Their model seems sound, but has some practical limitations on the utility function, which must be “twice continuously differentiable and concave”. Also, the differential utility function must be available to the resource allocation algorithm.

The intuition behind the Q-RAM algorithm is to allocate resources to the service and QoS dimension that makes the utility increase most at any point, until the resource is completely spent or increasing the resource usage does not increase the utility.

As the focus of Q-RAM is the resource allocation model, there is not provided any component or service model within their model, even though services are central in it. On the other hand, the Q-RAM model seems so generic that it may be applied for QoS management to any QoS aware component system.

2.3.7. Aura

The Aura project (Poladian et al. 2004) has an approach similar to QuA and this thesis. They recognize the process of transforming a user's QoS request to capabilities and resources, and the need for an efficient algorithm to select implementations. Aura uses an algorithm implementation from Q-RAM. However, the scope is limited to a single application and there is no reference to component models, although it is most likely to implement their services, called *capabilities*, using components. In addition, Aura shows how the implementation selection algorithm supports reconfiguration by comparing the observed utility with the best computed utility.

2.4. Summary

There exist several component models for middleware. Even the industrial models provide reflection capabilities, even though they are limited. Also, support for several extra-functional properties are available. The models with extended reflection capabilities also offer client applications a fine-grained control over inspection and adaptation of a running system.

There is still certain functionality that is not available in any of the middleware platforms. None of the platforms support choosing a component implementation upon the request of a component type based on resource availability and QoS requirements. Likewise, even though the platforms provide pluggable mechanisms for QoS management such as resource reservation and monitoring, it is assumed that the application must provide the mechanism itself, none of the platforms offers to completely manage QoS on behalf of both clients and components/objects. Q-RAM and Aura are the exceptions, where the platforms handle QoS, but do not provide any component model.

QuO is the only component platform which offers to perform the adaptation of a service based on a specification, the other platforms only opens up the possibility to adapt, but both the "reasoning" and the code to execute the adaptation must still be provided by the application. Aura also offers to provide reconfiguration, by periodically re-calculate to find a more optimal configuration, but how to actually reconfigure a service based on the new calculation is not explained.

With a component based platform, it would be great if it was possible to offer total QoS management to an application, including QoS specification, contract negotiation, resource management, monitoring, policing, and adaptation. It seems to still be a long way, but putting together some of the pieces described in this chapter could look like a step in the right direction.

3. QuA

The Quality Architecture (QuA) project[2] aims at building a component platform where the platform itself manages quality of service (Staehli and Eliassen 2002). QuA recognizes that reflection is a key mechanism for supporting the adaptability that is necessary for QoS management (Staehli et al. 2003).

3.1. A canonical component model

QuA introduces the notion of a canonical component model (Staehli and Eliassen 2002), where key concepts for component models are defined. It is not argued if the canonical model is complete, but such a model is useful for describing component models. The entities identified in the QuA canonical model correspond to the following:

- *Packaged component* – an immutable (binary) value representing a component implementation[3] (called *an X component*, where X is the type implemented by the component).
- *Component instance* – an entity that performs some work and communicates using messages (called *object* in Staehli and Eliassen (2002)[4]).
- *Composition* of component instances.
- *Component platform* – instantiates, manages, and executes the components.
- *Component type* – describes the syntax and semantic of the functional interface to be implemented by a component.

The term *packaged component* is chosen with care in this thesis. In various papers, and specifically in Szyperski (2002), it is often unclear when the term *component implementation* refers to the source code of a component, the (possibly) compiled and then packaged component, an installed component, or the component platform's representation of the implementation, which is used to instantiate the component.

Note that this model is only a suggestion, it is not validated that it is canonical, but it can still serve as a reference model for other component models, as Microsoft COM, EJB, or OpenCOM.

3.2. The QuA component model

QuA is a higher-level component model (Staehli and Eliassen 2002). This means that an implementation can utilize functionality provided by a lower level component model, e.g. OpenCOM (see section 2.3.3 on OpenCOM), but a lower level component model is not necessary. A QuA implementation only need to provide a small set of core functions. In fact, this thesis presents a QuA implementation based on Java 2 Standard Edition (J2SE).

The QuA component model contains the following entities:

- *QuA type* – specifies a syntactic and semantic interface for a component. QuA types may inherit from other types.
- *QuA component* – implements one or more QuA types.
- *QuA component repository* – where the packaged QuA components are installed.
- *QuA capsule* – implements the runtime platform.

---

[2] Joint project between Simula Research Laboratory, Sintef, and University of Tromsø, Norway
[3] Component implementations are in some QuA papers called *blueprints*.
[4] In the QuA papers where the term blueprint is used, "component" is often used as short hand for component instance.

In QuA, even the platform is built using QuA components (called *capsule service implementation components* in Figure 5). The *capsule core* only consists of the minimum functionality needed to load component repositories and, to instantiate and to bind the platform components. In addition, the core defines the QuA types for the platform components. This is in line with the canonical model described above.

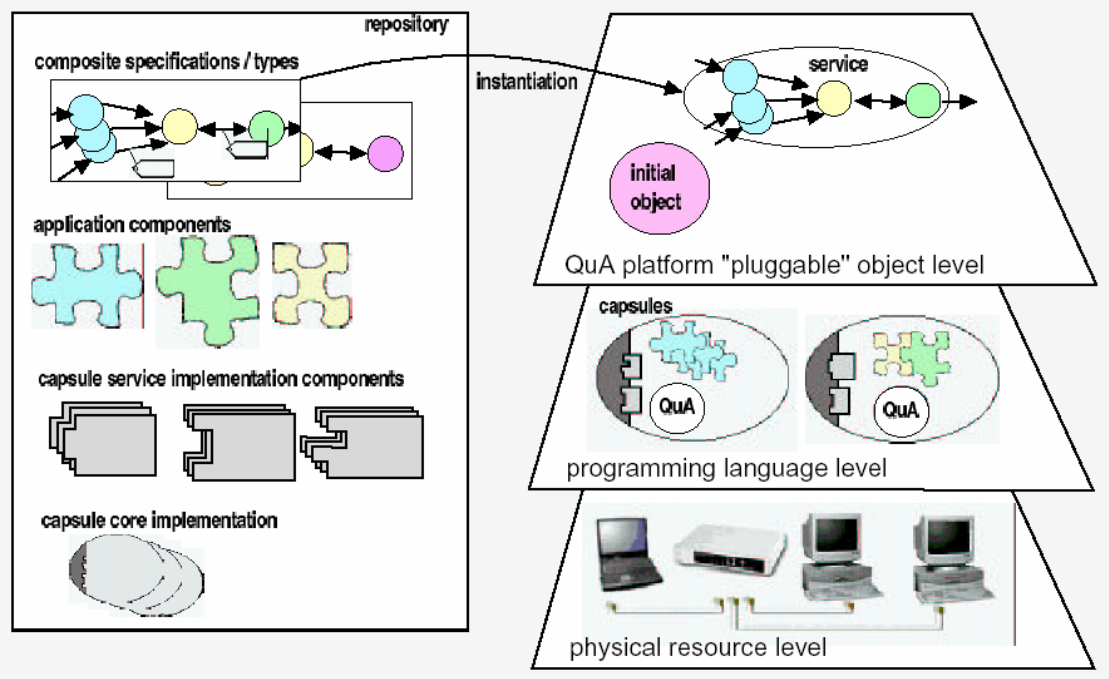

Figure 5: Conceptual view of the QuA architecture (Staehli and Eliassen 2002)

3.2.1. QuA object space

The object space in QuA (or just *QuA space*) is a distributed space, served by of a set of collaborating QuA capsules that may reside on different hosts or nodes. These capsules share a QuA namespace that is inhabited by QuA objects. The QuA platform defines some specific QuA objects, but arbitrary objects can be promoted to QuA objects by registering them with the QuA capsule. This means that any object in any process that instantiates a QuA capsule can be bound to other local or remote QuA objects.

The specified QuA object types are *component repository*, *component type*, *packaged component*, and *component instance*, in addition to arbitrary promoted objects. There is no special QuA object type for resources at this level.

3.2.2. QuA capsules

The QuA platform is a distributed platform, where each process runs (at least) one instance of a QuA capsule, similar to CORBA where each process in a CORBA space runs an ORB. The QuA capsule itself is a minimal core where components need to be plugged in to provide the necessary capsule services. Implementing the capsule as components is similar to the Open ORB 2 design (Blair et al. 2001). A small capsule core should be easier than a monolithic system to port to a range of platform, including platforms with limited resources such as PDAs and mobile phones. The necessary capsule components can then be ported independently.

3.2.3. Component repositories

All QuA capsules contain a *volatile repository* where run-time and other volatile objects are registered. Typical volatile objects are component instances and proxies for remote objects. In

addition, a number of *persistent repositories* can be served by each capsule. Persistent repositories will typically contain packaged component types and implementations.

QuA repositories are used to store types, implementations and instance references. In comparison with CORBA (Coulouris, Dollimore, and Kindberg 2001 p. 679), QuA repositories may bee seen both as an *interface repository* and *implementation repository*. However, in comparison with the CORBA implementation repository, the QuA repository is used for actually storing the implementation, while the QuA capsule is responsible for instantiating (*activating* in CORBA terminology) the components.

3.2.4. QuA components

Everything in a QuA capsule is implemented as components, with the exception of the minimum core functionality that is needed to instantiate and bind local components without any QoS management. This is similar to how Open ORB 2 is implemented using OpenCOM components (Blair et al. 2001).

Component types

A component type (or *QuA Type*) describes the syntax and semantics for a service (type). Ideally, both syntax and semantics should be defined formally, but QuA does not define how component types should be specified. Types are named, and may be versioned. A QuA Type is platform independent.

Component implementations

A component may be implemented in any programming language, or *platform*, if a QuA capsule supports that platform, similar to how CORBA object may be implemented in any platform for which a CORBA ORB is provided (Coulouris, Dollimore, and Kindberg 2001 p. 671). One component implementation may implement several component types. Components are compiled (if necessary), and packaged with metadata that describes which types it implements and which capsule version and platform implementation it requires, as well as its name and version.

Component dependencies

If a component requires another component, this should also be part of the component metadata. Components should only require a component type, not a specific implementation, but to allow for some extra flexibility, it may be possible to allow both. It is not specified how this is handled in QuA.

Composite components

Composite components are not clearly defined in QuA. A composite component specifies a set of components and their architecture, i.e., how the component instances are to be bound together. It should be possible to specify compositions of both types and implementations, and provide this specification in a capsule implementation neutral format, however, as with component dependencies, QuA does not specify such a format. QuA names could be used to refer to types and implementations in a packaged composite component, with additional information on the architecture. Any language to describe an object graph could be adopted for this usage.

Packaged components

A component package is an immutable binary value containing the blueprints needed to instantiate this component. The package is prefixed with following information about the contained component

- Which version of the QuA platform this component is implemented for. This can be used to maintain backward compatibility.

- Type implemented by this component, e.g. AudioSource.
- Short name for this component, e.g. Microphone.
- The QuA capsule type this component is implemented for, e.g. Java.

Component instances

Component implementations are instantiated by the QuA capsule and registered in the capsule's volatile repository. A component implementation needs to provide a well defined factory so the capsule is able to create, or *manufacture*, instances of that implementation. A component instance may be a factory for component instances of a different type, e.g. an audio binding factory instance could used to create audio bindings.

3.2.5. QuA/Type meta interface

All QuA components must implement the *QuA/Type* interface. This interface allows for discovering which types the component implements dynamically, similar to the *IUnknown* interface in Microsoft COM and the *Equivalent* interface in the CORBA Component Model.

3.2.6. QuA Names

When a packaged component is *published* to a component repository, its short name is prefixed with the path to this repository. Duplicate names are not allowed. Instead the QuA name contains a version number. In this way, it is possible to identify a QuA component implementation, or type, uniquely in one QuA space.

3.3. QuA services

A QuA component instance, or a collaborating composition of component instances, provides a *QuA service*. A service is defined to be "a subset of input messages to some composition of objects and their causally related outputs" (Staehli and Eliassen 2002 p. 5). The service consumer may either be an entity outside the QuA framework, i.e., a QuA client, or the QuA core (for platform services), or another QuA service. In the latter case, the consumed service is a called a *sub-service*. It is not defined whether a single component instance may serve multiple consumers simultaneously, or how services are scoped in this case.

3.3.1. Bindings and compositions

Component instances, and other objects promoted to QuA objects, may be bound together in a composition. The bindings are performed by binding services, which can create local or remote bindings.

3.3.2. Requesting services

A QuA client requests a service by providing the QuA capsule with a *service specification* (*ServiceSpec*). This specification contains the requested functionality of the service, represented with the QuA type of the service. The service may also be restricted to bind to objects in specific capsules. These objects may represent resources, as specific microphones and speakers in an audio conference, but can really be any QuA object, as long as the objects supports the necessary bindings to be composed as a part of the QuA service.

When requesting a service, the client can provide a *quality specification* (*QualitySpec*) along with the functional specification for the required service. The QualitySpec contains the QoS properties the client expects from the service. It is not defined how QualitySpecs are negotiated, and what happens if a QualitySpec can not be satisfied. For the latter case, the alternatives are to either re-negotiate the QualitySpec, or just reject the service request.

3.3.3. Service planning

Component implementations are not specified by the client, but provided by the platform itself. The composing is delegated to the service planning framework, which is a component framework where different service planners can be hooked in, e.g. a specific service planner for the audio domain.

Service planning consists of three parts:

- Implementation planning – selecting a component implementation
- Binding planning – selecting remote bindings to bind components together
- Composition planning – selecting the complete component composition for a service

3.3.4. Service execution

As stated above, a QuA component instance, or a composition of component instances, executes a service. The service can be within one capsule or distributed – in QuA the idea is that distribution of services should be transparent to the client. A service executes within a *service context*. If the service is distributed, the service context will be distributed as well.

3.4. Summary

Service planning is key to the QuA architecture. This is the main mechanism that QuA uses to be support platform managed QoS, together with leveraging an open and reflective architecture. Another important aspect of the architecture is that also the capsule itself consists of components providing capsule services. This design should make it easier to port QuA capsules to different platforms. QuA also embraces utility functions a generic approach to describe QoS.

4. Problem description

This chapter discusses service planning in the context of the overall QoS problem. At the end, the problem area of this thesis is scoped.

4.1. The overall QoS problem

The overall problem is to execute a service such that it is optimized both with respect to the end users' quality requirements, and the available system resources. If there were infinite – or just always enough – resources available, it would be trivial to create and offer component implementations that satisfy any end user requirement. But in practice in a distributed environment, resources have physical limitations, high prices, or low availability. QoS aware systems are introduced to cope with these limited resources. The challenge for these systems is to optimize resource utilization.

The following detailed description of the problem area is from the point of view for a *component based distributed system*.

4.1.1. Overview of a QoS session

We have established that QoS is not a problem – it is a solution to lack of resources. QoS management is split in two phases, static QoS management and dynamic QoS management (Aurrecoechea, Campbell, and Hauw 1998). The former phase takes place before the service is provided to the service consumer, and the latter phase during the service execution.

In the static QoS phase, the following typically takes place:

- *Describe QoS requirements*. The service consumer provides a formal description of her QoS requirements. These are then provided as the service consumer's input to the next step.
- *QoS negotiation* that results in a *QoS contract*. The QoS contract defines limits on the QoS characteristics that the service must operate within, and optionally also how many resources the service is allowed to consume during service execution.
- *Initial service configuration* is performed by the service provider. The service is configured to use some set of resources to provide a certain QoS level.

After the static phase is completed, the service is ready for use by the service consumer. If the service executes in an environment with resource management, resources are reserved during the static QoS phase, and – unless some reservation fails – the QoS management job is done. More often, resources are not managed in a way that make is possible to reserve all needed resources exclusively for a service, so QoS management continues with the dynamic phase, which consists of the following elements:

- *QoS monitoring* that monitors the *provided QoS*. If the provided QoS level decreases below the agreed QoS level in the contract, the service must be reconfigured.
- *Resource monitoring* – if more resources become available during the service execution, these can be used to increase the provided QoS.
- *Dynamic reconfiguration* of the service takes place based on QoS and resource monitoring. If the service cannot be reconfigured to still satisfy the QoS contract, the service should be terminated. It is also possible to start over with a new QoS negotiation at this point.
- *QoS policing* to avoid that the service consumes more resources than the QoS contract allows.

4.2. The service planning problem

Service planning means that the platform or middleware should provide a service to clients based on functional and QoS requirements (Staehli and Eliassen 2002). In this context, QoS requirements are all extra-functional requirements that apply to services at run-time, such as timing, accuracy, and security. More specific, service planning is the process of identifying and instantiating component implementations, and then configure and bind the instances together to provide the requested service.

This definition places service planning in the final "initial service configuration" step of the static QoS phase. Before the service can be planned, we must assume that the required QoS has been specified and negotiated.

4.2.1. Describing QoS

'QoS description' is almost an oxymoron in a computing context, as qualitative terms such as 'good' and 'bad' does not make any sense in a binary world. Computing is all about numbers; hence there is need for quantitative measures of a service's quality characteristics. A way of obtaining quantitative measures is to break down quality characteristics into a layer of sub-characteristics, with mappings on how to calculate the more qualitative value on the higher level from the more quantitative values on the lower level. CQML (Aagedal 2001) provides a notation for recursive quality characteristics, e.g.:

```
quality_characteristic media_quality {
      domain: increasing enum{bad, average, good}
      audio_quality;
      video_quality;
}
```

Here we define the QoS characteristic, or QoS dimension, 'media_quality' for an audio-video service to have the increasingly better values 'poor', 'average', and 'good', and that this characteristic is made up of the characteristics 'audio_quality' and 'video_quality'. These characteristics must again be defined, and may be broken down this way in several layers, until objectively measurable characteristics are defined. To continue the example, 'video_quality' could be defined as:

```
quality_characteristic video_quality {
      domain: increasing enum{bad, average, good}
      frame_resolution;
      frame_rate;
      color_depth;
}
```

The good, the bad, and the context

After the main quality characteristics have been identified and broken down into objective measures, the mappings between the different layers must be defined. The problem here is that which combination of audio and video quality that maps to the media quality domain depends on the context the media service is used in. What is 'good' quality in one context, may be 'bad' quality in another; consider a (one-way) streaming application in two different cases. In our first case, the user accesses the service using a mobile phone with a limited screen. This user would probably say that a very small frame resolution is 'good'. In our second case, consider a user accessing the service using a regular workstation with a high resolution screen. The latter user would probably map the 'good' frame resolution in the first case to 'bad'. In other words, QoS is highly context dependent.

Tradeoffs

Our former user using her mobile phone probably also has other limited resources such as bandwidth. In the case above, this does not really matter – it is just very inconvenient to carry a 20" LCD panel around in your pocket. But on the other hand, the latter user may also be on

some remote location using a mobile phone link with limited bandwidth, and this connection would not be able to handle a combination of high frame resolution, high frame rate, and high color depth. She would then have to prioritize between these QoS characteristics, and this prioritization must be included in the QoS characteristic mappings to be communicated to the service provider.

Utility functions

One way of representing the overall QoS for a service consisting of several underlying quality dimensions, is to use a utility function, which maps values from all the underlying dimensions to a normalized real value in the interval [0, 1]. A utility value of 0 means that the service is useless to the service consumer, because the quality of one or more dimensions are unacceptable. 1 means that the QoS is as good as desired in all dimensions.

The user's requirements and tradeoffs are captured and mapped to mathematic expressions in the utility function. How this is done, is studied in the field of *human-computer interactions* (Poladian et al. 2004).

4.2.2. Negotiating QoS

The following must be resolved during the QoS negotiation:

- How much resources are required to satisfy the requested QoS
- How much of the necessary resources are available
- And, optionally, the cost of the service

Required resources

Before it is possible to calculate the required resources, it is necessary to figure out which component implementations the service will be composed of, as different implementations will have different resource requirements, e.g., for our video stream example, using MPEG4 instead of MPEG2 will require less bandwidth and more CPU.

This means that already at this stage, the system must figure out alternative service compositions, component implementations, and estimate how much resources the different alternatives will require.

Available resources

Discovering available resources can optionally be combined with reserving the necessary resources. If the service will be distributed among several autonomous resource management domains, resource usage must be negotiated with each domain.

Cost

QoS negotiations are complicated if one or both of the parties do not want to disclose all their preferences. A parameter that is likely not to be revealed from the other party, is the price-to-service-level ratio, i.e. how much the consumer is willing to pay for a certain service level, or how cheap the provider can sell a certain service level.

The service consumer may disclose what she is willing to pay for different levels of QoS to the producer. This opens up for that the party calculating the utility, which is most likely the producer, can take advantage of the other party. Fixed pricings or having a neutral third party broker to negotiate the service are possible solutions to this. Koistinen and Seetharaman (1998) presents an algorithm where two parties can negotiate QoS without disclosing all underlying information, but their approach does require that some level of trust are established between the service provider and the service consumer.

Another complicating factor is the non-autonomous resource domains mention above. The service provider could have to negotiate the cost for the resources the service requires.

However, this is more likely to be done in advance then as a part of the QoS negotiation, e.g. network connectivity is ordered months in advance, but it is also possible to envision resources procured on demand, e.g., some resource provider may have lots of spare computing power, and wants to sell access to this resource. Note that providing resources this way could be treated like providing a (low-level) service.

4.2.3. Initial service configuration

When the negotiated QoS has been accepted, the system can instantiate the components that the service is to be composed of. As we discovered above, it is necessary to find possible service compositions already in the QoS negotiation step.

Single service vs. entire system

Another dimension to the QoS problem, is that the service provider may have a different opinion on what QoS is 'best' than the service consumer. The consumer wants as high as possible utility on one or a few concurrent services, while the producer has two options; serve as many consumers as possible, or maximize the profit of resource usage. We call the former maximizing the *system utility value*, while the latter can be thought of as maximizing the *resource utility value*. The latter is interesting to service producers that are resource consumers of resource providers.

For the consumer using the services provided, the best implementation is the one providing the highest utility value, limited by the current available resources, and possibly the cost of those resources.

The system utility value can be defined to be the sum of the utility of each service provided by the system (Rajkumar et al. 1997). Note that using this definition, serving 1 consumer with 100% utility, is better than serving 99 consumers with 1% utility each, while the number of served consumers could have an impact in the total system utility value. Also note that the utility values must be normalized in such a model, and after services are normalized, it is possible to weigh the utility of some services higher than others.

In a distributed multi-party system where consumers must pay for resource consumption, it may be more likely that the 'best' service is the one which the consumer is willing to pay most for using least resources, but still getting satisfactory quality. A service could even get reduced quality or be terminated by the service provider if another consumer appears that are willing to pay more for the same resources.

4.2.4. Resource monitoring

Resource monitoring is to monitor how much capacity are available for the resources currently used by the service, or possibly used in another configuration of the service. If resource capacity increases, the service can be configured to utilize the increased capacity to increase QoS level.

If resource monitoring discovers that capacity is decreasing with a trend that tells the system that the QoS will be dramatically affected, the system can try to reconfigure the service to use less resources and still provide adequate QoS. Consider our video stream example and imagine that the network throughput decreases so much that the system will have to drop frames because they arrive to late to be played. It could be better to reconfigure the system to decrease the frame size and sustain the frame rate.

It is not the responsibility of resource monitoring to reconfigure the system – it only notifies other parts of the system, which may take actions.

4.2.5. QoS monitoring

QoS monitoring is similar to resource monitoring, except that here is the actual QoS monitored. QoS monitoring will not discover that extra resource capacity is available; in that

case it will only report that the service is running with maximum QoS in the current configuration. This information could be used to reconfigure the system to utilize more resources, but without resource monitoring, the system can not really know if the resources have spare capacity.

On the other hand, QoS monitoring will report that the quality is decreasing, and this information can be used to trigger dynamic reconfiguration, e.g., to utilize the available resources better. Consider QoS monitoring for the video streaming application; the QoS monitoring discovers that the frame rate drops drastically, which could happen due to network congestion. This is another way to trigger the reconfigure of the service to reduce the frame size to maintain the original frame rate.

If the monitored QoS drops below an acceptable lever altogether, the service should be terminated and possibly renegotiated.

When the service has completed execution, a QoS monitoring report can be used to calculate the actual cost of the service.

4.2.6. Dynamic reconfiguration

Dynamic reconfiguration of services, also called service adaptation, may occur at different levels of a service. At the lowest granularity it means that some parameters of an executing component instance are tuned by some QoS manager. A higher level of adaptation is when one ore more executing component instances are replaced with other components, possibly changing the component architecture of the service. At the highest level, when the system is not able to provide the agreed service level, adaptation can incur renegotiation of the service level. If cost is involved, this may mean that the service provider has to lower the price for the service.

Service adaptation may also be applied recursively. Figure 6 shows a video service made up of three component instances, where the video stream composite component consists of four component instances. The lines between the component instances reflect the bindings. Changing some run-time parameter of the video stream component may propagate in three different ways:

1. The parameters of one or more of the sub-components may be altered. E.g., the compression ratio on the encoder is changed.
2. Some of the components are changed, but the component architecture (how the components are bound together) is kept. E.g., the codec could be switched by changing the encoder and decoder components with different implementations.
3. The component architecture is completely changed, e.g. the stream switches to a raw video stream, where the encoder, decoder, and controller components are removed, and only the data stream is kept.

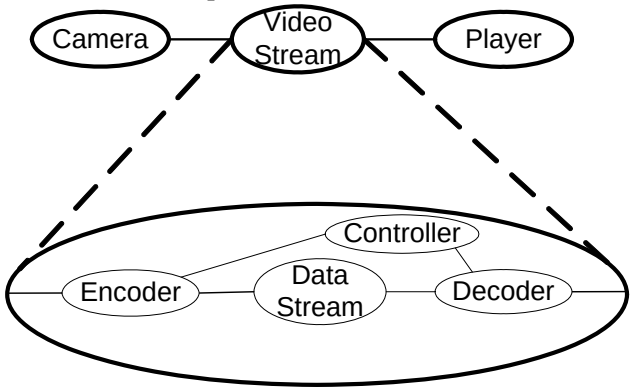

Figure 6: Recursive reconfiguration of composite components

4.2.7. QoS policing

QoS policing is the mechanism that ensures that a service does not utilize more resources than agreed in the QoS contract. QoS policing can utilize resource monitoring and dynamic reconfiguration, and potentially terminate services that uses too much resource capacity.

4.3. Problem scope

As we see, the QoS problem area is quite wide. This thesis will look at the following elements:

- How to programmatically describe QoS and available resources, independently from different component implementations.
- How a QoS description can be used to select a single component implementation.
- How a component instance can be configured to maximize QoS, given limited resource availability.

This is summarized as:

> *How can a component middleware efficiently choose a component implementation based on component type, QoS requirements and available resources?*

This leaves several important mechanisms that must exist in a complete QoS aware system as beyond the scope of this thesis:

- QoS negotiation
- Resource and QoS monitoring
- Dynamic reconfiguration
- QoS policing

Note that the way utility functions are used in this thesis, they should not be considered to be a part of QoS negotiation. It is expected that the system maximizes the utility value with the current given available resources, and also cost is not a part of the utility functions. A utility function could possibly be the outcome of a prior QoS negotiation.

In addition, the following optional elements are also considered beyond the scope:

- Cost
- Resource management

The narrowed problem area thus does not cover an entire QoS aware system by itself, but it is neither sufficient to look only at the areas beyond the scope of this thesis.

4.4. Summary

In this chapter, the QoS problem area has been described, and the problem for this thesis has been narrowed down to a part of the problem area; how to select a component implementation in a specific environment.

5. Analysis

This chapter starts with presenting a hypothesis for QoS-aware service planning. It then goes on with analysis on how the hypothesis can be implemented and tested, and finally describes two experiments that can be used to evaluate the hypothesis.

5.1. Method

The overall method used in the thesis, is to present a hypothesis and test it with an experiment. To be able to test this way, the experiment is described as a set of test cases and goals.

5.2. Hypothesis

QuA states that service planning requires specific service planner for different QoS domains, called applications domains (Staehli and Eliassen 2002). However, since QuA provides a generic approach to QoS modeling using utility functions (ibid.), a generic component model (ibid.), and a generic resource model (Abrahamsen), it could also be possible to provide a generic solution to select component implementations for a specific service. The hypothesis to test can be formulated as:

> *It is possible to make a generic solution to selecting and configuring a component implementation when the QoS model and resource model are generic, with as high level of QoS as possible, given limited resource availability.*

Intuitively, a possible obstacle for this hypothesis is that the solution space is so big that any generic solution will not be efficient enough or precise enough. Efficient here means how fast an implementation and configuration is selected, while precise means how far from the highest possible level of QoS the selection actually is.

The inspiration for the hypothesis is the solution of Q-RAM (Rajkumar et al. 1997), which describes a very similar problem; how to maximize the resource usage given a set of services. Transformed to a component based platform, the components for the services are already selected in Q-RAM, but only satisfying a minimum level of QoS.

5.3. Prototype background

To be able to conduct the experiment, we need a prototype of a component based platform. The early QuA capsule prototype written in Smalltalk by Staehli based on (Staehli and Eliassen 2002) founds a good basis. Important aspects of the QuA platform that the capsule needs to support includes:

- Pluggable service planners
- Component repositories
- QoS descriptions
- Resource descriptions

The prototype needs to provide components for:

- Basic implementations of capsule core services – for bootstrapping the capsule
- Example component implementations to select and configure
- The generic implementation planner itself

The prototype can show that the proposed solution is feasible. It is also possible to test a prototype for precision and effectiveness.

5.3.1. Component types and QoS models

In general, a component type is described by its syntactic and semantic specifications (Szyperski 2002). In order to specify QoS and calculate resource requirements for given QoS levels, all service consumers and component implementations must agree on which QoS dimensions that are known for a specific type, hence this must be a part of the type specification. We call the set of QoS dimensions that a type specifies the type's *QoS model*.

A QoS model may include other QoS models. E.g., a QoS model for timing, defining QoS dimensions as delay and jitter, may be included by QoS models for both audio and video, and a QoS model for multimedia may include the audio and video QoS models.

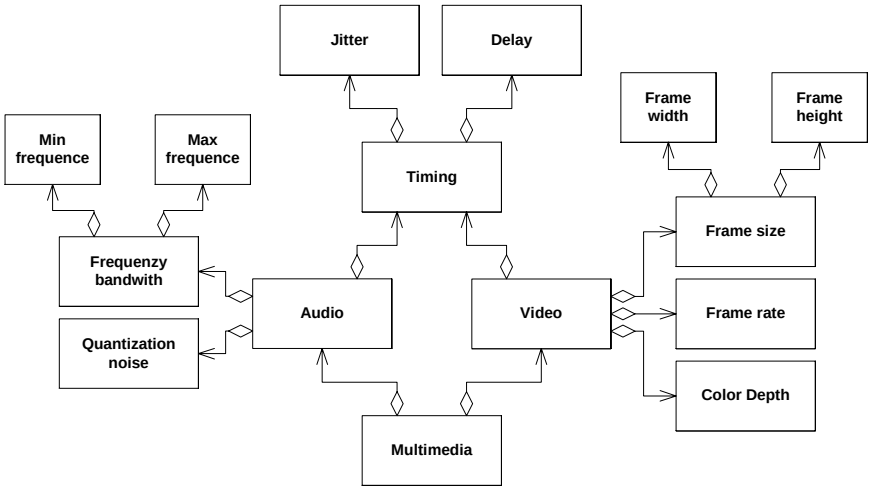

Figure 7: Example QoS model for multimedia

Figure 7 shows such a QoS model. The utility for a service of this type would only depend on the values from the QoS dimensions in the leaf nodes of the QoS model, with objective, quantitative values; hence an implementation of QoS models can be simplified to only consist of the leaf dimensions.

A *QoS statement* holds values for all the leaf QoS dimensions of a QoS model. The statement is just a value, and can represent values such as current monitored QoS, minimum QoS, or maximum QoS.

5.3.2. Utility function interface

Depending on the context, the function to calculate the utility value may combine the QoS dimensions in any possible way. We can define a generic interface for a utility function to accept a single argument; a QoS statement. An issue with a generic interface is that the QoS model known to the utility function may differ from the QoS statement. A missing value for a QoS dimension is most likely to cause an error when computing the utility value, and its existence should be asserted by the platform before invoking the function. Values for QoS dimensions unknown to the function can be ignored; implying that the quality in those dimensions do not affect the utility value.

To normalize the utility values for different service requests, we define the value range for utility functions to be a real number in [0, 1], where 0 is unusable, and 1 is as good as possible in the context the service is requested in.

The utility may be different for the same objective QoS level, depending on the subject of interest; e.g., an audio service with a frequency cutoff at 4 kHz (used for speech) and a delay of 0.5 seconds could yield a utility of 0.2 for a client that requests music to be played, while it could yield a utility of 0.9 for a client that requests to listen to a debate.

5.3.3. Service requests

In QuA, a service is defined as an output trace with and a causally related input trace (Staehli and Eliassen 2002). More specific, a service has type that specifies both its syntax and semantics, which can be described in a *functional specification*. The required extra-functional properties of the service can be described in a *quality specification*. A *service request* consists of a both the functional specification and the quality specification.

Functional specification

The simplest service request only specifies the type of the service, and the platform is free to realize the service in any way to enhance the quality of the system.

The functional specification may also hold additional requirements, as binding to specific resources; e.g. a specific microphone and loudspeaker. It may also be desirable to request a specific implementation of a type to override the platform, or even a specific instance of a component implementation.

A realization of the functional specification may hold references to the following elements:

- Components types to include in the service.
- Components implementations to include in the service.
- Component and resource instances to include in the service.
- List of unidirectional component bindings (service architecture description).

Quality specification

The quality specification must at least provide the utility function, so the system can calculate the utility of any given QoS statement. To simplify service planning (and potentially QoS monitoring), it should contain two QoS statements in addition:

- Minimum QoS (minQoS) values for each QoS dimension. If the quality drops below the given value in any of these dimensions, the service becomes useless for the service consumer, and the utility value drops to 0.
- Maximum QoS (maxQoS) values for each QoS dimension. There is no point for the system to enhance the quality in any dimension above this value. If all dimensions are at maximum or higher, the utility value is 1.

The utility function is defined in the range [minQoS, maxQoS] for all the QoS dimensions in its quality model, and it must be *increasing* in this range. The utility value would never decrease because of an increase in some QoS dimension, so this should not be a problem when specifying a utility function. To support the case where increasing the quality in one QoS dimension does not have an effect unless the quality in some other QoS dimension also increases, it is not required that the function is *strictly increasing*. Since the maxQoS statement is required, the system can tell if maximum quality is not provided in a QoS dimension, even though the utility value does not increase with every increase in this dimension.

The quality specification needs to provide a mechanism to calculate the utility value for a given QoS level, represented as a QoS statement. In a distributed environment, this can be realized in a number of ways, as mobile code (e.g., a packaged component), remote method calls, a platform neutral function language, or limiting the service planner to run in the capsule that requests the service.

In a component platform, it is tempting to choose providing utility functions as packaged components, but in a language independent component system, such as CCM or QuA, this will still limit the function to be executed on the underlying platform it was written on. Remote method calls solves this problem, but as the function may be executed almost

continuously during service planning and QoS monitoring, this may impose a large overhead. A platform neutral function language is the only approach where the system can choose freely where to execute the function, but this requires a formal language description and execution environment, e.g., an interpreter, to be a part of the component platform.

With a prototype that only supports one language, it is sufficient to provide the utility function as a packaged component.

The QoS model in this thesis is a *quality model*, as opposed to the *error model* described by Staehli and Eliassen (2004). An error model can be viewed as the inverse of a quality model, as it measures the *error* or how far from ideal a service is operating.

5.3.4. Describing resources

Several resource models have been suggested throughout the years, as the *General Resource Model* in the *UML Profile for Schedulability, Performance, and Time*, and *CQML+* (Abrahamsen). Resources may exist at different abstraction layers, e.g. CPU cycles may be abstracted to operating system processes or application threads, as shown in Table 1.

Abstraction	**Processing**	**Memory**
Application	Thread	Buffers
OS	Process	Virtual memory
HW	CPU cycles	Swap space and physical memory

Table 1: Resource abstraction layers

Resources are of either spatial or temporal character (Rajkumar et al. 1997), as shown in Table 2.

Memory	spatial
CPU cycles	temporal
Throughput	temporal

Table 2: Example of spatial and temporal resources

To be more precise, all resources are temporal, i.e. any resource can be shared over time. Also, a spatial resource can be broken down to units that are only temporal, e.g. a single memory block (Abrahamsen).

Temporal resources can also be divided between resources that allow time-sharing, and resources that are only of exclusive use (Abrahamsen). For the latter type, a resource manager could emulate a time-sharing mechanism.

Resources can be described similar to QoS. They need to be described at the lowest level that they can be monitored or managed

5.3.5. Algorithm for the Generic Implementation Planner

The rationale for providing a generic algorithm to choose component implementation and configuration is to avoid implementing separate algorithms for each QoS model.

Each QoS-aware component must provide a function for resource requirement based on a QoS statement. This function is used by the algorithm to verify if the system can provide the necessary resources for a given QoS level.

Requiring that the service request must provide the minQoS statement, we can quickly disregard component implementations that require more resources than currently available, under the assumption that to be able to provide increased QoS, more resources are required for that component. This is a sound assumption – a component requiring *fewer* resources to provide higher QoS would be completely backwards.

First, the implementation planner locates all QoS-aware component implementations of the given type, using an implementation broker. Before the actual search for the best possible

utility is begun, there are two possible cut-offs; first the planner checks the resource requirements for the maxQoS statement for all the implementations. If there are enough resources available for any of the implementations, any of these can be selected immediately. If all implementations require more resources than currently available to satisfy the maxQoS statement, the second cut-off is to check the resource requirement for the minQoS statement for all of the implementations. Any implementation that requires more resources to satisfy the minQoS statement than currently available is disregarded as described above at this point.

The implementation planner's task now, is to maximize the utility value in the space defined by [minQoS, maxQoS). As the only requirement on the utility function is to be an increasing function, i.e., if the QoS level increases in any one if its defined QoS dimensions, the utility can not decrease, the QoS space must be search in all dimensions, until there are not enough resources available to increase the quality in some dimension. This search can be implemented as a depth-first search.

The problem is similar to the algorithm presented by Rajkumar et al. (1998) in the way that we have a set of discrete and dependent QoS dimensions, for which we want to maximize the utility value, Rajkumar et al. (1998) prove that the problem of maximizing the resource allocation with discrete and dependent QoS dimensions is an inexact 0-1 knapsack problem, which is NP-hard. It is not formally proven that the variation of the problem in this thesis also is NP-hard, but its close relation with the problem in Q-RAM suggests it is.

5.4. Experiment description

This section describes two experiments, one where an audio codec is selected and configured, and one where a raw video stream is configured. The former experiment is designed to analyze the precision of the algorithm, and the latter experiment is designed to analyze the effectiveness of the algorithm.

5.4.1. Select and configure audio codec

In this experiment, the Generic Implementation Planner should select and configure the audio codec component implementation that provides the highest utility, with 1 or 2 QoS dimensions in the quality specification, and limited resources.

The QoS dimensions are frequency bandwidth and quantization noise, and the required resources are network bandwidth and CPU. We will map these dimension to quantified values for objective measurement.

Frequency bandwidth

Frequency bandwidth should be a fairly objective measurement by itself. The human ear is capable of hearing sounds of approximately 20 to 20.000 kHz (Tanenbaum 1996 p 724). According to the Nyquist theorem it is sufficient to sample twice the highest frequency to be reproduced (Tanenbaum 1996 p. 81)[5]. The sample rate for audio CDs, which is 44.1 kHz, should then be sufficient to reproduce all audible frequencies, however some claim that using higher sampling rates will provide reproduced audio that sounds closer to its original form to the human ear (Dunn 1998). Digital audio tapes (DAT) provides 48 kHz sample rate, and DVD audio supports up to 192 kHz sample rate!

For simplicity, we disregard the minimum frequency of the frequency bandwidth in the experiment, setting the frequency bandwidth equal to the maximum frequency. We then map the frequency bandwidth to just below ½ of the sampling rate, which can be used to represent this QoS dimension.

[5] Tanenbaum is slightly wrong according to Wikipedia (2007), which states the Shannon-Nyquist theorem as follows (my emphasis): "Exact reconstruction of a continuous-time baseband signal from its samples is possible if the signal is bandlimited and the sampling frequency is *greater* than twice the signal bandwidth."

Quantization noise

When transforming an analog signal to a discrete value for digital representation, some information is always lost, as shown in Figure 8. If the quantization of audio is crude enough, it is audible to the human ear, and is known as quantization noise (Tanenbaum 1996 p. 725).

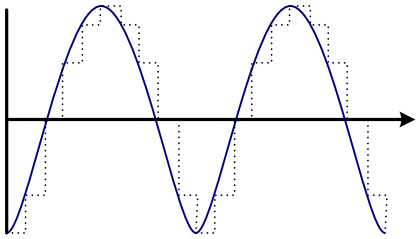

Figure 8: Quantization of analog signal

Each audio sample is represented with a finite number of bits. Audio CDs uses 16 bits to represent each sample per channel, while DVD audio uses 16, 20, or 24 bits per channel. Higher number of bits provides less quantization noise.

Compression

Audio can be compressed in various ways. An obvious compression scheme is to reduce the sample rate. This is a lossy compression where the information of higher tones is lost. The compression ratio is then equal to the ratio between the original and produced sample rates.

Another way to compress audio is to reduce the sample size. One scheme is to use coarser values to represent each sample, e.g. 8 bits instead of 16 bits. Reducing the sample size will increase the quantization noise.

The intuitive approach is to distribute the sample values linear, but the human ear picks up sound on a logarithmic scale (Tanenbaum 1996 p 724), which is leveraged in phone systems, that uses either the *A-law* or *μ-law* algorithm for compression, both uses logarithmic distribution of the quantified values (ITU-T 1993).

There are also other audio compression schemes that use different schemes for compression audio, as MP3 (Peterson and Davie 2000, p. 557).

It is possible to measure the quantization noise when compressing audio, but since the ideal audio signal is analog, some error has already been introduced. However, the perceived quality of audio is strongly subjective, which is studied in psychoacoustics. ITU-T has developed subjective methods for measuring the perceived quality of speech codecs, called Mean Opinion Score (ITU-T 1996).

For this experiment, we will assume that the sample size reflects the quantization noise.

QoS space

Table 3 shows the two QoS dimensions for the experiment, and the domain used to represent values on these dimensions. For simplicity, we ignore timing QoS dimensions as delay and jitter. Both the processing in the audio codec and network transport will impose delay. Jitter may be caused by variations in CPU and network availability during the execution of the service. Adding a buffer at the audio sink may reduce jitter, but this will increase the delay.

QoS dimension	Domain	Direction	Suffix
Sample rate	Real numbers	Increasing	kHz
Sample size	Integers	Increasing	bit

Table 3: QoS dimensions for audio codec experiment

QoS aware components

For this experiment, we will mock two imaginary QoS aware audio codec components. Both are configurable in the following ranges:

- Sample rate [8.0, 44.1] kHz, in steps of 0.1 kHz
- Sample size [8, 16] bits, in steps of 1 bit

Both components only support one audio channel (mono).

The first component implementation (**raw**) mocks a simple codec that will only downsample raw real-time audio. This component requires CPU based on the output sample rate, and both the sample rate and sample size depend on available network throughput.

For the second component implementation (**compressor**), we mock a codec that utilizes CPU to compress raw or downsampled audio. This component is capable of loss-less compression of real-time audio up to 25%, but at a cost of additional CPU. Note that the resource requirement for an actual loss-less encoder would depend on the actual content of the media stream. E.g., gaps of silence can be compressed much more efficient than rock music.

Resource requirements

The resource requirements for the two components are shown in Table 4.

Resource	Raw	Compressor
CPU	Sample rate * 0.3	Sample rate * 0.3 * (compression/50 + 1)
Bandwidth	Sample rate * sample size	Sample rate * sample size * (1-compression)

Table 4: Component resource requirements

Figure 9 shows the bandwidth requirements for the **raw** audio codec component. The output of this function could be fed into a data stream component which may add overhead in packet headers and possible retransmission of packets. The sample frame size is also relevant to compute network packet sizes, as frames should not be split between packets, as well as network path minimum MTU, which should be detected to avoid packet fragmentation.

For raw audio, the sample frame size is simply the sample size, but various codecs, as MP3, may groups information for several samples into one frame, and add a header with time and codec information (Peterson and Davie 2000 p 558).

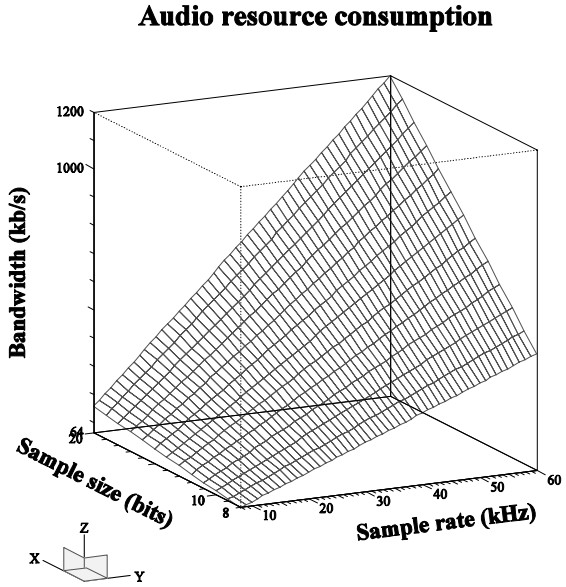

Figure 9: Bandwith requirements for the raw audio codec component

Utility function

Figure 10 shows the utility function we will use in the experiment. The related minimum and maximum QoS constraints are shown in Table 5.

QoS dimension	minQoS	maxQoS
Sample rate (kHz)	8.0	44.1
Sample size (bits)	8	16

Table 5: Minimum and maximum QoS constraints for audio codec experiment

The utility function to compute the surface is defined as follows:

Sample rate utility function u_r

8 kHz <= sample rate < =44.1 kHz:

$u_r = \log_{37.2}(\text{sample rate} - 6.9)$

Sample size utility function u_s

8 bits <= sample size <= 16 bits:

$u_s = (\text{sample size}-7) / 9$

Combined audio utility function u_a

$u_a = u_r * u_s$

The logarithm base of the sample rate factor is chosen to normalize the function in the interval [0, 1]. The function implies that increase in sample rate is preferred over increase in sample size up to certain point.

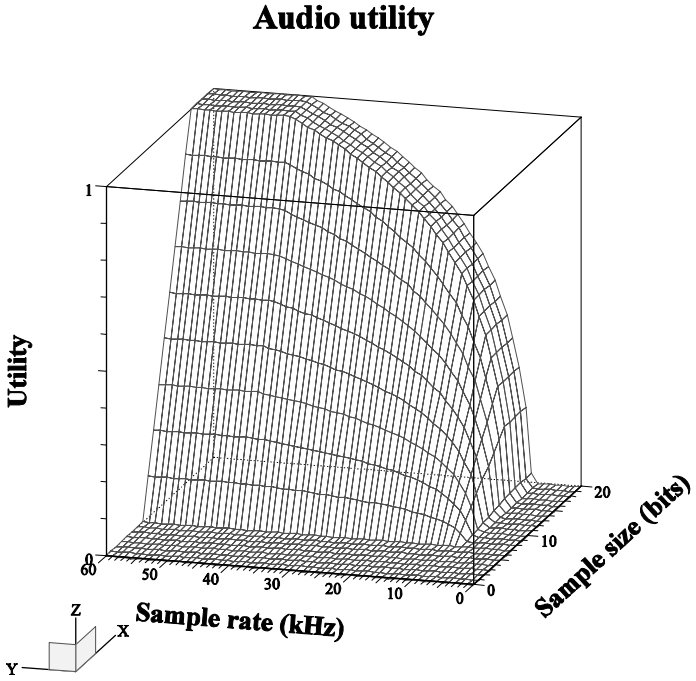

Figure 10: A utility function for audio

Scenarios

To investigate how the algorithm behaves under varying conditions, we will run 6 different scenarios, adjusting the available components, available resources, and number of QoS dimension the service consumer has requirements on. Table 6 shows the resource availability for the various scenarios. For the scenarios with only one QoS dimension, only the corresponding factor of the utility function is used.

Scenario	Available components	QoS dimensions	Utility function	CPU	Bandwidth
1	**Raw**	Sample size	u_s	55%	50-150 kb/s
2	**Raw**	Sample rate	u_r	55%	50-150 kb/s
3	**Raw**	Sample rate and sample size	u_a	55%	50-150 kb/s
4	**Raw** and **compressor**	Sample size	u_s	55%	50-150 kb/s
5	**Raw** and **compressor**	Sample rate	u_r	55%	50-150 kb/s
6	**Raw** and **compressor**	Sample rate and sample size	u_a	55%	50-150 kb/s

Table 6: Experiment scenarios for audio codec experiment

The available bandwidth will be increased in steps of 10 kb/s. The variation in bandwidth should cover the points where increase in sample size should be preferred to increase in sample size in scenario 3 and 6.

5.4.2. Configure video stream

In this experiment, the Generic Implementation Planner should configure a raw video stream that provides the highest utility, with 1, 2, or 3 QoS dimensions in the quality specification, and limited network bandwidth. The QoS dimensions are frame size, frame rate, and color depth.

Frame size

The frame size is the product of frame height and frame width. Assuming the aspect ratio of 4:3 in standard television broadcast (Tanenbaum 1996 p. 728), we can represent the frame size with only one of these dimensions. The frame height in European television is 576 visible lines (ibid.). In this experiment, we will use frame height as the QoS dimension.

Frame rate

The frame rate is simply the number of video frames displayed per second. At 50 frames per second, the human eye is not able to recognize the frames as individual images (Tanenbaum 1996 p. 727), however a frame rate of 25 Hz is sufficient to view smooth movement (Tanenbaum 1996 p. 728).

Color depth

The color depth is the number of different colors used in the video stream; the higher number of different colors we want to reproduce, the higher number of bits must be used to represent each pixel. Digitizing color is similar to sampling audio – if too few bits are used, we get a visible quantization error. When using 8 bits for each of the additive primary colors, (red, green, and blue, also referred to as RGB), we get a color depth of $2^{24} \approx 16.8$ million different colors, which is more than the human eye is capable of distinguish (Tanenbaum 1996 p. 729). In this experiment, we will use the number of bits per pixel to represent the color depth.

QoS space

Table 3 shows the three QoS dimensions for the experiment, and the domain used to represent values in these dimensions.

QoS dimension	Domain	Direction	Suffix
Frame rate	Integers	Increasing	Hz
Frame height	Integers	Increasing	line
Color depth	Integers	Increasing	bit

Table 7: QoS dimensions for video stream experiment

QoS aware component

For this experiment, we will only mock one imaginary QoS aware video stream component, configurable in the following ranges:

- Frame rate [1,25] Hz, in steps of 1 Hz
- Frame height [30, 576] lines, in steps of 1 line
- Color depth: [8, 24] bits, in steps of 1 bit

Resource requirements

The network bandwidth requirement for streaming raw video, with no frame or network packet overhead, is (in kb/s):

frame size * frame rate * color depth / 1000

Frame size can be expressed as:

frame height^2 * 4/3

Utility function

The utility function for this experiment consists of too many dimensions to be visualized, but it is similar to the utility function for the audio codec experiment, with the addition of an extra logarithmic component. The related minimum and maximum QoS constraints are shown in Table 8.

QoS dimension	minQoS	maxQoS
Frame height (lines)	30	576
Frame rate (Hz)	1	25
Color depth (bits)	8	24

Table 8: Minimum and maximum QoS constraints for video stream experiment

The utility function to compute the surface is defined as follows:

Frame height utility function u_h

30 lines <= frame height <= 576 lines:

u_h = (frame height - 29) / 546

Frame rate utility function u_r

1 Hz <= frame rate <= 25 Hz:

$u_r = \log_{25.1}(\text{frame rate} + 0.1)$

Color depth utility function u_c

8 bits <= color depth <= 25 bits:

$u_c = \log_{17.1}(\text{color depth} - 6.9)$

Frame height and rate video utility function u_f

$u_f = u_h * u_r$

Combined video utility function u_v

$u_v = u_h * u_r * u_c$

As in the select audio codec experiment, the logarithm bases are chosen to normalize the functions in the interval [0, 1].

Scenarios

To be able to perform timing operations, we need to design the various scenarios to be possible to satisfy the minimum required QoS, but not the maximum required QoS.

Table 9 lists the scenarios we will use for this experiment. The available network bandwidth will be varied in steps if 500 kb/s.

Scenario	QoS dimensions	Utility function	Bandwidth
7	Frame height	u_h	500-4000 kb/s
8	Frame height and frame rate	u_f	500-4000 kb/s
9	Frame height, frame rate and color depth	u_v	500-4000 kb/s

Table 9: Experiment scenarios for video stream experiment

5.4.3. Goals

To be usable, the algorithm needs to be *effective* and *precise*. As suggested earlier, the problem the algorithm is trying to solve is most likely NP-hard, but if the solution space is small enough, it can be searched fast enough. The algorithm should find a solution with the highest possible utility value with the available resources, or at least very close to this value.

Effectiveness

As a service may consist of a large set of components, the implementation planner will have to run several times during service planning. It is hard to define a concrete goal for how fast the algorithm must to be useful in practical appliance, but the algorithm implementation probably has to finish within one or a few seconds per component type on a typical personal computer.

It will be interesting to measure the effectiveness of the algorithm as the number of QoS dimensions is increased in the experiments. 2-3 QoS dimensions may be insufficient for real-world components, e.g., a multimedia stream component supporting both video and audio constraints would be configurable in all the five QoS dimensions in the two experiments described here, and the quality in each dimension would probably depend on the available network bandwidth and CPU. The suggested NP-hard nature of the problem requires services with few QoS dimensions to be planned very quick, should the algorithm be usable for services with more QoS dimensions.

Precision

If we call the utility value provided by the configuration provided by the algorithm $u_{selected}$, and the highest possible utility with the available resources $u_{possible}$, we can define the precision of the algorithm as $u_{selected} / u_{possible}$. It is hard to find $u_{possible}$, but we can try to reason around $u_{selected}$ with the component configuration output from the implementation planner.

In the audio experiment, we can get an indication of the precision by looking at how the algorithm selects configuration around the point where increasing sample size yields higher utility than increasing sample rate. To find this point, we can compare the derivative of each of the components f = utility(sample size) and g = utility(sample rate).

$$f(x) = \frac{x-7}{9} \qquad g(x) = \log_{37.2}(x-6.9)$$

$$f'(x) = \frac{1}{9} \qquad g'(x) = \frac{1}{(x-6.9)\ln(37.2)}$$

$$\frac{1}{9} = \frac{1}{(x-6.9)\ln(37.2)}$$

$$x = \frac{9}{\ln(37.2)} + 6.9$$

$$x \approx 9.39$$

This shows that for sample rates above 9.4 kb/s, increasing the sample size yields a relative higher utility than increasing the sample rate.

How much of the available resources that are utilized is another indication of the precision of the algorithm, assuming that the algorithm does select the component implementation and configuration that utilizes the available resources in the best way. It may also be that the selected component does not require all the available resources of some resource types.

5.5. Summary

This chapter has presented a hypothesis for a generic approach to QoS-aware service planning in a component based software architecture. The QoS requirements for two domains, audio encoding and video streaming, have been described. Experiments have been detailed, with a set of scenarios for each domain. Finally the goals for the experiments – precision and effectiveness – have been discussed.

6. Designing and implementing service planning

The implementation consists of a QuA Java capsule core, basic implementations of the core services, and the QoS-aware Generic Implementation Planner, which implements the algorithm described in section 5.3.5. This chapter discusses the design and implementation, which is based on a previous prototype.

6.1. Porting issues

The Java capsule core is a port to Java from the prototype in Smalltalk, developed by Staehli and based on the architecture described by Staehli and Eliassen (2002). The port was not straightforward, as Smalltalk has stronger support for reflection than Java, e.g., inspecting the method call stack to find the current service context. Also, the loose type system of Smalltalk (method arguments are not typed), were intriguing the porting. Smalltalk does not support interfaces the way Java does, but abstract classes were provided to define the interfaces for some implementation classes.

The main additions in the Java version compared to the Smalltalk version, are that QuA types are represented as Java interfaces, several of the core services are made into components, a package format for QuA component implementations is defined, along with a format for persistent repositories, and a repository implementation that reads this format.

The Smalltalk prototype also lacked models for representing QoS and resources, which are required for our experiments. A simplified resource model, based on the QuA resource model (Abrahamsen), and a simple QoS model, based on CQML (Aagedal 2001), have been created for the QuA Java capsule.

6.2. Capsule core design model

The QuA capsule itself is made up of component frameworks, similar to Open ORB II. Component frameworks in the capsule core are the service planner framework, broker framework, repositories framework, and resource management framework. All the frameworks can be configured using alternative implementations of the core services that make up each component framework.

This section provides an overview of each of the capsule core packages shown in Figure 11.

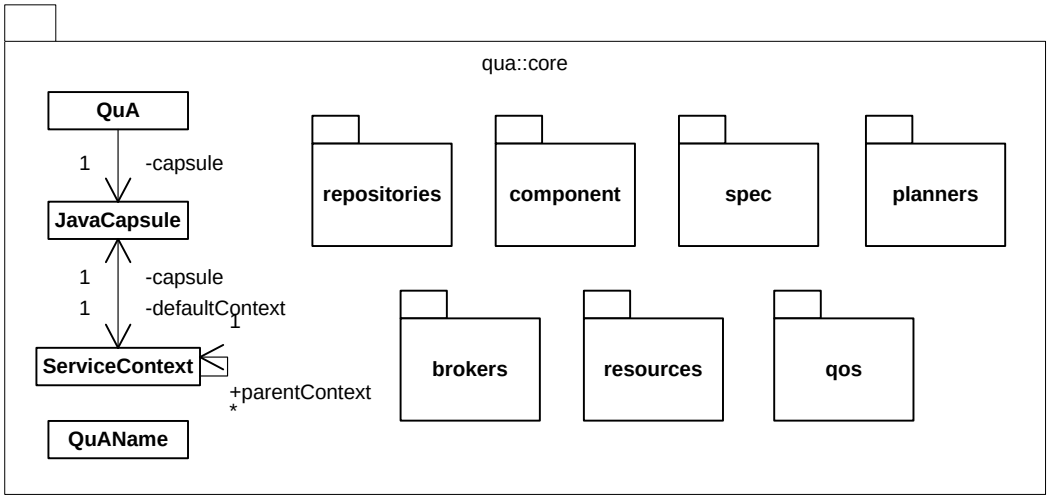

Figure 11: Static UML structure of the capsule core packages

6.2.1. Package qua.core

The most central core classes reside directly in the qua.core package. They are shown in Figure 12.

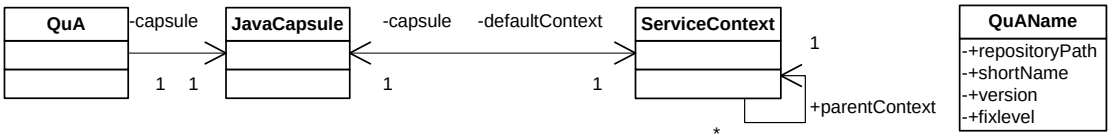

Figure 12: Static UML structure of the qua.core package

The **QuA** class is the entry point for clients of the QuA Java capsule. It provides methods for configuring, initializing, and shutting down a capsule instance, request services to be planned, and execute planned services in a QuA service context.

Capsule instances are represented by instances of the **JavaCapsule** class. These instances hold the configuration of available repositories, service planners and resource managers. JavaCapsule is also capable of instantiating local components, which is leveraged when providing QuA components as capsule services.

An instance of the **ServiceContext** class is intended to represent the context of the current service. Contexts are arranged in a hierarchy. In this implementation, only a default, local context is provided, but it could be extended to be distributed between several QuA capsules.

Instances of **QuAName** represents known objects in the QuA space, as repositories, component types, component implementations, and component and resource instances. The syntax for QuA names are defined as follows:

QuAName ::= repositoryPath [“/” shortName [“:” version “.” fixlevel]]

QuA naming conventions

The capsule itself is not named, but each Java capsule has a volatile repository which is named. The volatile repository name can be used to identify the capsule.

The /qua/types repository path is reserved for types. The type names are mapped to Java interface names by converting the path name part of the name to a Java package name. Since Java does not support versioning of classes and interfaces, the version information is lost for types. An example of this mapping is the QoSAware type, which has to be implemented by all QoS-aware components:

/qua/types/QoSAware:1.0 → qua.types.QoSAware

An interface repository has not been implemented, implying that all interfaces that represents QuA types must be available in the class path for the Java capsule at runtime. However, the persistent repository could be extended to also operate as an interface repository.

6.2.2. Package qua.core.repositories

The design model for repositories is shown in Figure 13. Both persistent and volatile repositories are named.

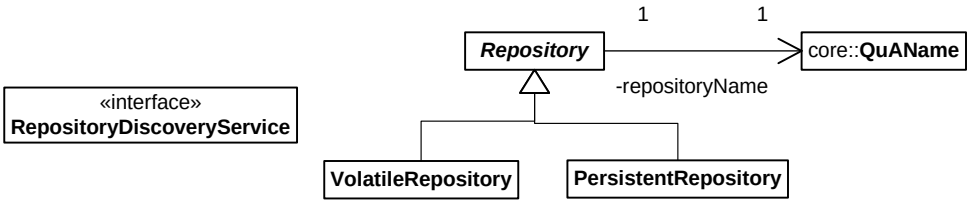

Figure 13: Static UML structure of the qua.core.repositories package

Each capsule instance contains one **VolatileRepository** instance, where it stores all the QuA objects that it knows, such as references to other repositories, component implementations cache, component instances, and arbitrary QuA objects.

An instance of **PersistentRepository** mounts a zip file that contains a set of packaged QuA component implementations. Each capsule instance mounts zero or more persistent repositories. The current implementation only supports reading the file, requiring external tools to generate persistent repository files, but the implementation can be extended to write to the zip file when component implementations are published to persistent repositories. Copying zip files between capsule nodes is a possible way of distributing packaged components.

The **RepositoryDiscoveryService** is a core service that is used by the capsule to discover repositories. An implementation of this service may discover remote repositories in other capsule instances.

Initial repositories and the discovery service implementation are configured when instantiating a Java capsule.

6.2.3. Package qua.core.component

A packaged QuA component implementation consist of meta-information and a platform-specific component implementation. For the Java capsule, the platform-specific implementation is a Java archive (JAR) file, with the required attribute "QuAComponent" in the manifest file. The attribute is used to specify the façade class of the component. The façade class must have a public, default constructor, which is used to instantiate the component.

The classes shown in Figure 14 are used to represent component implementation meta information, and load and instantiate packaged component implementations.

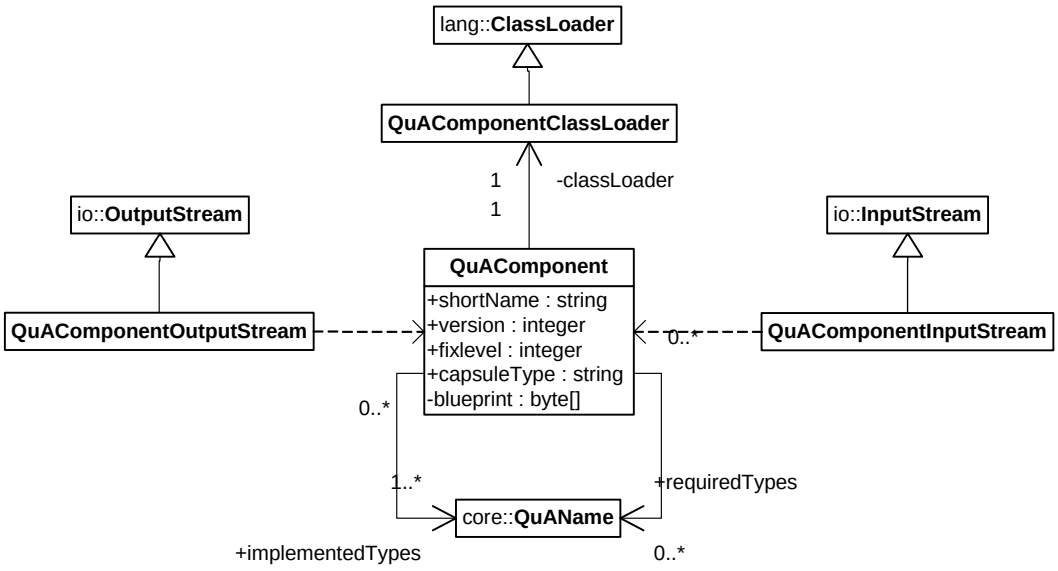

Figure 14: Static UML structure of the qua.core.component package

Dynamic component loading

Instances of the **QuAComponent** meta class is created by PersistentRepository by using a **QuAComponentInputStream** to read a packaged QuA component from the repository file. **QuAComponentOutputStream** can be used to serialize a QuAComponent instance to a remote capsule or another persistent repository. QuAComponent is a meta-class, and instances of QuAComponent instances can represent QuA component implementations for any QuA capsule type.

To avoid namespace clashes for the Java implementation of QuA components, i.e. two different components uses the same class name for different classes, each component is loaded through a separate instance of **QuAComponentClassloader**. The class loader reads

the JAR file that is represented as the *blueprint* byte array, and calls the default constructor of the façade class specified in the manifest file.

If two component implementations uses a Java library that is not available as a QuA component, both of the components must include the library in their package, and it will be loaded twice into memory, once by each class loader. Whenever possible, such libraries should be made into separate components to avoid redundant class loading.

6.2.4. Package qua.core.spec

Clients requesting a service need to programmatically create a specification of the service. The specification *may* also contain a specification of the desired quality of the service, but the capsule does not guarantee any QoS, as this requires a QoS-aware service planner to be configured in the service context that will plan and execute the service.

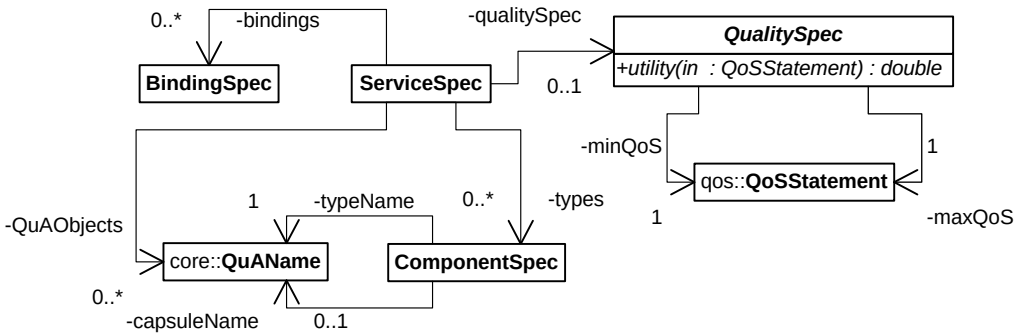

Figure 15: Static UML structure of the qua.core.spec package

The QuA capsule client must provide an instance of a **ServiceSpec**, which is the main class of the service specification. It consists of the composition of the desired service; required current QuA objects, required component types, and the required bindings between the components and objects. The latter is provided using instances of the **BindingSpec** class. Note that the list of QuA objects may contain both component instances as well as other arbitrary objects that have been promoted to QuA objects.

Instances of **ComponentSpec** may hold an additional constraint on a new component instances; the volatile repository (i.e., capsule instance) that the component must be instantiated by.

Clients that require a QoS-aware service must provide an implementation of the abstract **QualitySpec** class, including a *utility* method for the utility function to be used when planning the service. QualitySpec instances hold two QoS statements: *minQoS*, which is the lowest quality that will satisfy the client, and *maxQoS*, the service level that there is no point to exceed.

The following predicates must hold for a QualitySpec:

1. utility(maxQoS) = 1
2. $\forall qos : qos \in [minQoS, maxQoS) \land 0 < utility(qos) < 1$
3. $\forall a, b : a, b \in [minQoS, maxQoS] \land a < b \land utility(a) \leq utility(b)$

The second and third predicate ensure that the function is non-decreasing in the interval it is defined. Note that it is not necessary to define the function outside the interval [*minQoS*, *maxQoS*]. The service planner uses the QoS constraints defined in *minQoS* and *maxQoS* to avoid stepping out of the interval.

6.2.5. Package qua.core.planners

As shown in Figure 16, this package only contains the interfaces for the three components that make up a service planner. The service planner implementations to use in the default service context, are configured when instantiating a Java capsule.

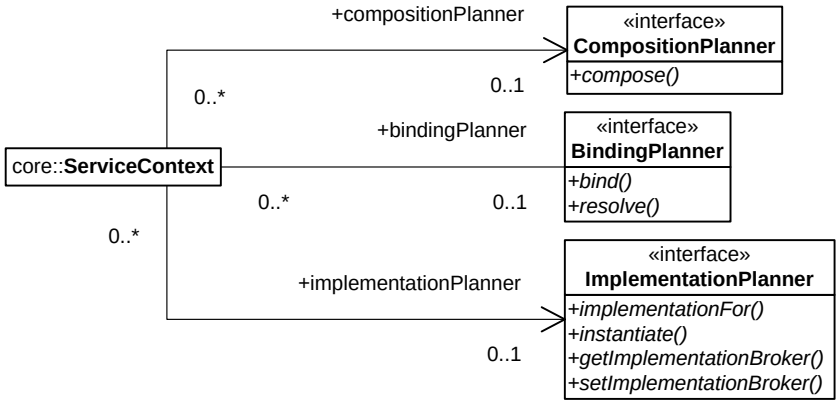

Figure 16: Static UML structure of the qua.core.planners package

CompositionPlanner implementations have the overall responsibility for instantiating the entire service object graph and bind it together.

Implementations of **ImplementationPlanner** use an ImplementationBroker to locate a component implementation that implements a given component type, and on request instantiate a given component. It is the ImplementationPlanner that must select between components if several components implement the same type.

A **BindingPlanner** implementation is responsible for resolving registered QuA objects to be used in a service, and creating bindings to both local and remote QuA objects.

6.2.6. Package qua.core.brokers

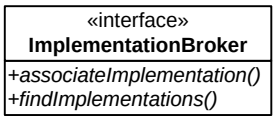

Figure 17: Static UML structure of the qua.core.brokers package

The only specified broker in this capsule implementation is the **ImplementationBroker**, which is responsible for searching repositories for component implementations, and registering discovered component implementations. An ImplementationBroker implementation is configured when instantiating a Java capsule.

6.2.7. Package qua.core.qos

The QoS model is simplified to not represent hierarchical and enumerated QoS dimensions, though enumeration could be represented as a limited range of natural numbers. Combining the QoS dimensions is left to the discretion of the utility functions, so this model is sufficient for the Generic Implementation Planner.

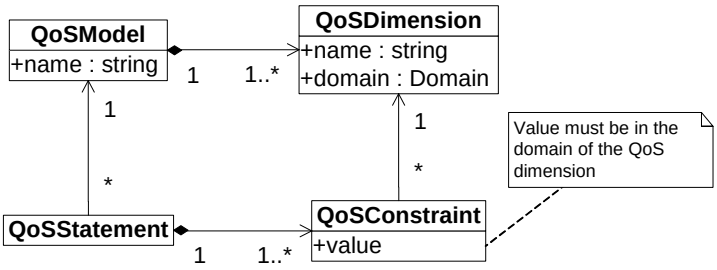

Figure 18: Static UML structure of the qua.core.qos package

QoSDimension is the basic building block in the QoS model.

An instance of **QoSModel** contains a set of QoSDimensions that is associated with a service domain. Examples of QoS models are:

- **TimingQoSModel**, consisting of delay and jitter.
- **AudioQoSModel**, consisting of sample rate and sample size.
- **VideoQoSModel**, consisting of frame rate, frame size, and color depth.

QoSConstraint sets a constraint on a single QoS dimension. The constraint must be a value in the domain specified by the dimension.

QoSStatement consists of a set of QoS constraints on the QoS dimensions of the related QoS model. A QoS statement may have different semantics depending on the context. Some example of QoS statements are minimum QoS, maximum QoS, and current QoS. If a constraint for a dimension is not included in a QoS statement, it means that the quality of that dimension does not matter, even though the dimension is included in the QoS model of the statement.

6.2.8. Package qua.core.resources

For this implementation, we assume a resource manager that allows reserving the resources that it manages. We also assume that both local and distributed resources are managed through a single resource manager.

The implementation includes a simplified version of the QuA resource model (Abrahamsen). It does not support hierarchical resource types, and the differences between spatial, temporal and exclusive resources are left to the discretion of the Resource Manager. The implementation includes a plug-in mechanism for resource managers.

The resource design model is symmetric to the QoS resource model.

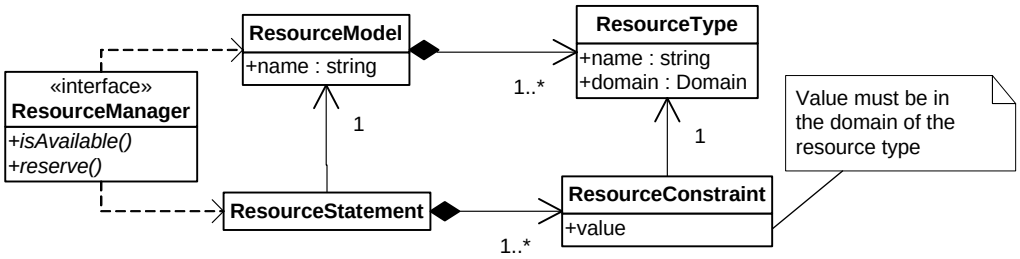

Figure 19: Static UML structure of the qua.core.resources package

ResourceType is the basic building block in the model. Resource types in QuA are typically in the application abstraction layer. Examples of resource types are:

- Memory buffers
- Threads
- Bandwidth

ResourceModel represents the resources that can be managed by the current QuA space. A resource model consists of a set of resource types. Examples of resource models are:

- **PCResourceModel**, consisting of CPU, memory, and peripherals such as microphone, loudspeakers, and display.
- **LANResourceModel**, consisting of network bandwidth.

ResourceConstraint sets a limitation on a single resource.

ResourceStatement is a collection of resource constraints. The semantics of resource statements differ depending on the context. A statement can mean the current reserved or used capacity of the resources, the current free capacity of the resources, the total capacity or the required capacity of a resource. Components specify their resource requirements with resource statements.

A **ResourceManager** implementation is typically capable of managing resources defined in a specific resource model. The resource model for a component implementation may differ from the resource model currently managed. If a component requires a resource type that is not known by the resource manager, the service planner should reject this component, as the resource reservation will fail. The resource manager implementation to use can be configured when the Java capsule is instantiated

6.3. Capsule service components

To provide a working QuA Java capsule, we need at least some basic implementations of the core services for service planning, implementation broker, and the repository discovery service.

6.3.1. BasicServicePlanner

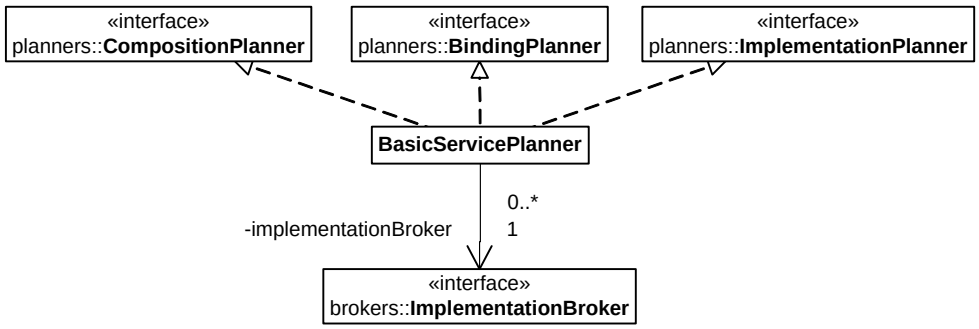

Figure 20: The BasicServicePlanner component

This is a basic implementation of a complete, non-QoS-aware service planner. As shown in Figure 20, the **BasicServicePlanner** implements all the three service planner interfaces, but each part is designed to work in a context where the other service planners are provided by other implementations.

6.3.2. BasicImplementationBroker

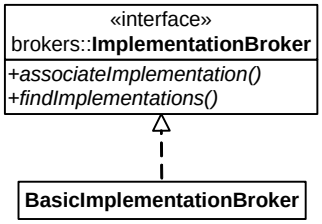

Figure 21: The BasicImplementationBroker component

The **BasicImplementationBroker** is used by the BasicServicePlanner to find component implementations of given QuA types. It uses the repository discovery service to find local repositories, and search these repositories. Remote repositories are not supported by the BasicImplementationBroker.

6.3.3. BasicRepositoryDiscoveryService

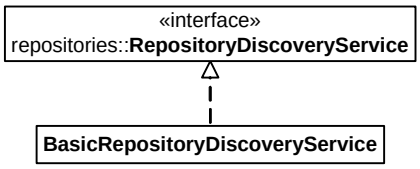

Figure 22: The BasicRepositoryDiscoveryService component

The **BasicRepositoryDiscoveryService** is a simple implementation of a repository discovery service. It simply "discovers" local and remote repositories by the client telling the component about them.

6.4. Instantiating the QuA Java capsule

The QuA Java capsule is configured to a set of properties. If the basic core component implementations above are stored in a persistent repository file "basic.rep", the following properties can be used to instantiate a capsule named "/capsule1", which mounts the persistent repository as "/basic" and loads core components from this repository:

```
qua.VolatileRepository.name=/capsule1
qua.PersistentRepository_1.name=/basic
qua.PersistentRepository_1.file=basic.rep
qua.planner.CompositionPlanner.component=/basic/BasicServicePlanner:1.0
qua.planner.ImplementationPlanner.component =/basic/BasicServicePlanner:1.0
qua.planner.BindingPlanner.component=/basic/BasicServicePlanner:1.0
qua.broker.ImplementationBroker.component=/basic/BasicImplementationBroker:1.0
qua.repository.DiscoveryService.component=/basic/BasicRepositoryDiscoveryService:1.0
```

The properties can be provided as a file or programmatically when instantiating a capsule through the QuA class. It is also possible to replace core components dynamically at runtime.

Figure 23 shows the overview of the QuA Java capsule, including the basic core components.

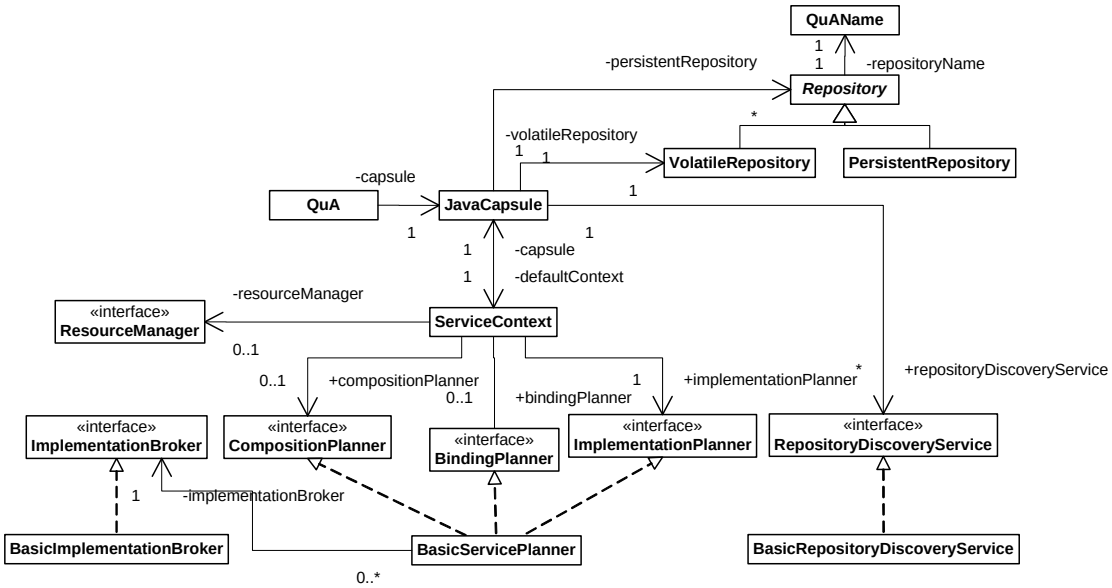

Figure 23: Static UML structure of the QuA Java capsule

6.5. Adding QoS awareness to the QuA Java capsule

So far the QuA Java capsule is not really QoS-aware. It defines models for QoS specification and resource management, but it does not use these models to create QoS-aware services. To be able to create QoS-aware services, we need three more pieces to the capsule implementation:

- A way of marking components as QoS-aware and calculate their required resources.
- A QoS-aware service planner that implements the generic implementation selection and configuration algorithm.
- A resource manager for checking available resources, and resource reservations.

6.5.1. A QuA type for QoS-aware components

One way of marking components as QoS-aware, is to require these components to implement a specific QuA component type. We define the type /qua/types/QoSAware for this purpose, which maps to the Java interface **QoSAware.**

The service planner needs to figure out required resources for various QoS configurations of the QoS-aware components. To calculate the required resources, we simply extend the QoSAware type with a method to calculate required resources based on a QoSStatement, thus forcing all QoS-aware components to implement this method. The method can return a list of one or more resource statements, in case more than one resource configuration can be used to provide the same level of QoS. This is similar to the programming model of QRR (Frølund and Koistinen 1990).

A drawback to this solution is that the implementation planner must instantiate all the components it is considering to use, and since the required resources must be calculated for every iteration of the algorithm, it should instantiate the components in the local capsule, which limits it to consider component implementations for the capsule type is it running on, i.e. only QuA components for the QuA Java Capsule in our case.

Alternatives to this approach could be to describe the resource requirements in a QuA capsule platform neutral language, or provide helper components for required resource calculations implemented on several QuA capsule platforms. The meta information for a QoS-aware component would then have to include either of these.

Finally, the service planner need a method to configure the QoS-aware with the selected QoSStatement and ResourceStatement. The ResourceStatement must of course be one of the statements returned by the call to *requiredResources*.

The complete specification for QoSAware is shown in Figure 24.

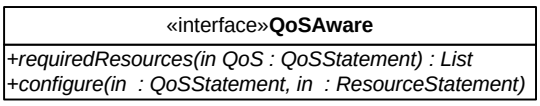

Figure 24: The interface for /qua/types/QoSAware

6.5.2. The GenericImplementationPlanner

The heart of our QoS-aware service planner is the implementation planner that will select and configure the best suited QoS-aware component implementation for the provided QualitySpec and available resources. The **GenericImplementationPlanner** implements the algorithm outlined in section 5.3.5 with a recursive depth-first search through the space defined by the minQoS and maxQoS properties of the QualitySpec. Figure 25 shows the GenericImplementationPlanner and the interfaces it uses for selecting and instantiating an implementation

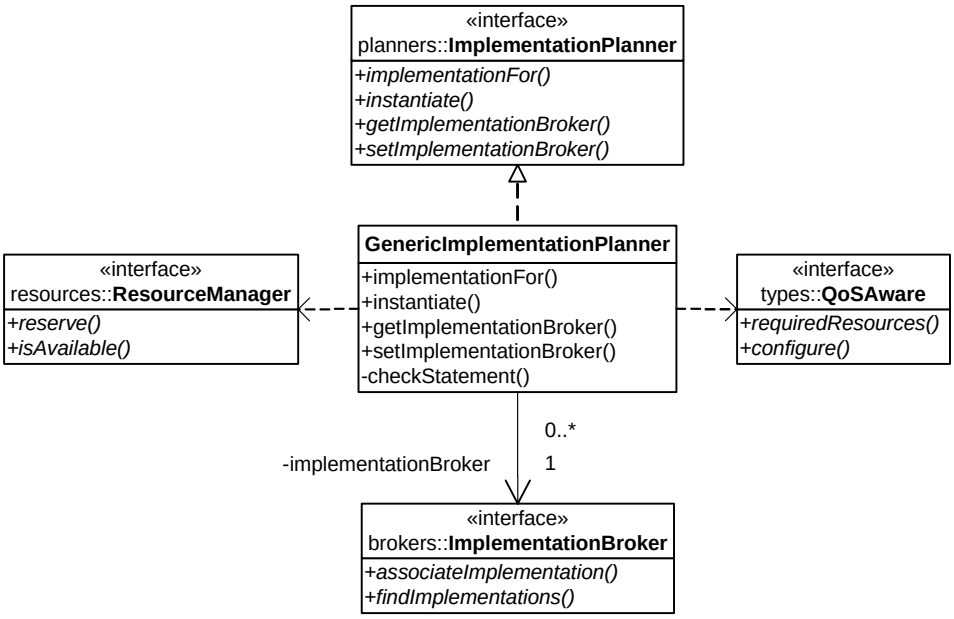

Figure 25: Static UML structure of the QoS-aware implementation planner

When asked to instantiate a given type, the GenericImplementationPlanner first asks the configured ImplementationBroker of available implementations, using the *findImplementations* method. When it has retrieves these component implementations, it creates one instance of each, and then starts the search for which implementation and configuration to use.

The recursive method that is used to search the QoS space is *checkStatement*. It checks the output from the *requiredResources* method in QoSAware with *isAvailable* in the configured ResourceManager. If a QoSStatement requires more resources than currently available, it stops further searching the QoS dimension that is currently being searched.

The GenericImplementionPlanner also reserves the required resources when instantiating the selected component, and finally it calls *configure* on the selected component instance.

If services are planned in parallel, there is a race condition for reserving resources that has recently been verified to be available between the service planner instances.

The *implementationFor* and *instantiate* methods of GeneralImplementationPlanner are synchronized to ensure that each planner instance only plan one implementation at a time.

6.5.3. A dummy resource manager

The resource manager is mocked with a dummy resource manager that can be manipulated through the *setAvailable* method for the experiments, as shown in Figure 26.

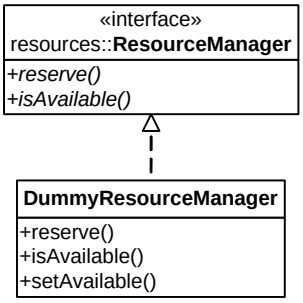

Figure 26: Static UML structure of the dummy resource manager

6.5.4. QoS-aware components

When implementing QoS-aware components, the developer is has to understand the QoS model defined for the type of her component, and figure out how to calculate the resource requirements for her implementation. If it is not possible to deduce the resource requirements directly from the implementation, she may have to use heuristics from experimenting with the component for the calculation.

The component must also be made configurable when instantiated, and preferable re-configurable during execution, to be able to support dynamic QoS management.

For our experiments, we only need dummy components, but their resource requirement calculations still have to reflect reality. The components we will use are shown in Figure 27.

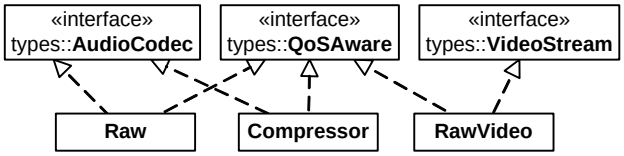

Figure 27: Static UML structure of the dummy QoS-aware components

6.6. Summary

This chapter has discussed a Java implementation of a QuA capsule prototype in Smalltalk. The capsule has been extended to provide core services as QuA components with basic implementations of the services. The service planner has then been extended with a QoS-aware implementation planner, and a QuA type for QoS-aware components has been introduced. The Java implementation will be used to run the experiments described in chapter 5.

7. Experiment results

This chapter contains the results from running the experiments described in chapter 5, using the QuA Java capsule from chapter 6.

7.1. Experiment environment

The environment was an Intel Pentium 4 3.2 GHz CPU with hyperthreading and 1 GB of RAM, running Windows XP. The CPU and operating system should reflect a typical "home PC". Regarding the memory size, executing the audio codec experiment only consumed 11-12 MB, and the video stream experiment consumed 18.6 MB of RAM, so the experiments should be reproducible on a computer with significantly less memory installed.

The QuA capsule ran on top of Sun's Java Development Kit 1.6.0 with no tuning parameters set.

7.2. Select audio codec

This section contains the results from running the scenarios and an analysis of the produced values.

Before running the actual scenarios, the service was planned once so the QuA capsule had discovered and loaded the component implementations. The initial planning also ensured that the JVM had loaded all classes into memory. If this had not been done, the first service to be planned had suffered a penalty when discovering and loading component implementations, which are cached for later service planning in the same capsule instance.

The scenarios described in section 5.4.1 are repeated in Table 10.

Scenario	Available components	QoS dimensions	CPU	Bandwidth
1	**Raw**	Sample size	55%	50-150 kb/s
2	**Raw**	Sample rate	55%	50-150 kb/s
3	**Raw**	Sample rate and sample size	55%	50-150 kb/s
4	**Raw** and **compressor**	Sample size	55%	50-150 kb/s
5	**Raw** and **compressor**	Sample rate	55%	50-150 kb/s
6	**Raw** and **compressor**	Sample rate and sample size	55%	50-150 kb/s

Table 10: Experiment scenarios for audio codec experiment revisited

The **compressor** component implementation was selected by the planner for all configurations in scenario 4, 5 and 6. This is caused by the available CPU, which the component could utilize to reduce the required bandwidth.

The x-axis shows the variations in available bandwidth in all the following charts. Figure 28 through Figure 33 shows the QoS configurations selected by the generic implementation planner for the different scenarios.

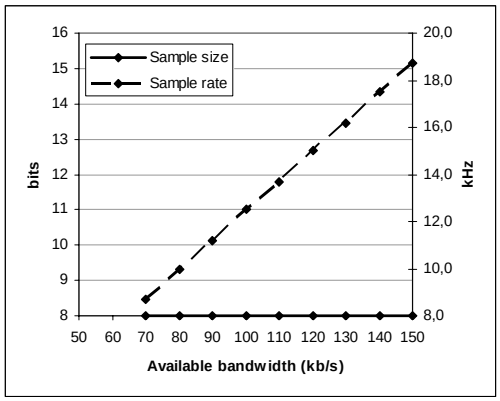

Figure 28: Selected configurations for scenario 1

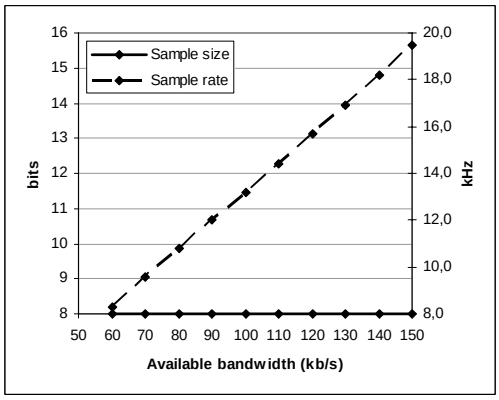

Figure 29: Selected configurations for scenario 4

The main difference between scenario 1 and 4 is that the **compressor** component makes it possible to serve the audio stream already at 60 kb/s available network bandwidth. In addition, we get a slightly higher sample rate in scenario 4 due to compression. The sample size stays at 8 bits, the minimum supported size.

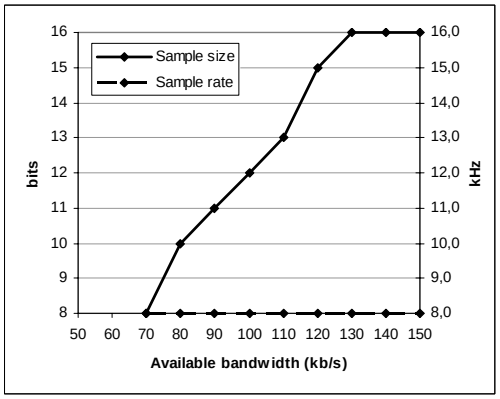

Figure 30: Selected configurations for scenario 2

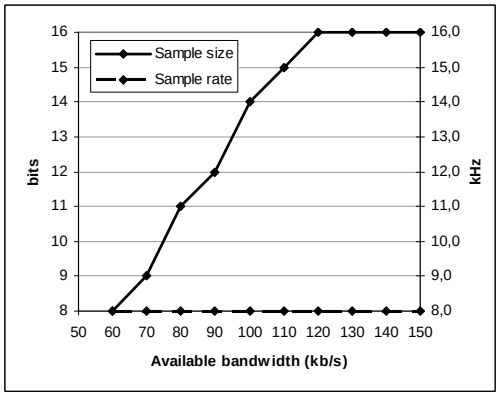

Figure 31: Selected configurations for scenario 5

We see the same pattern when comparing the selected configurations for scenario 2 and 5; due to compression, scenario 5 supports a higher increase in sample rate than scenario 2. When the maximum requested sample rate is achieved, it is no longer increased.

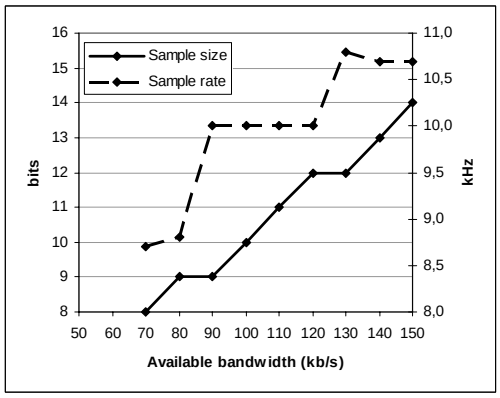

Figure 32: Selected configurations for scenario 3

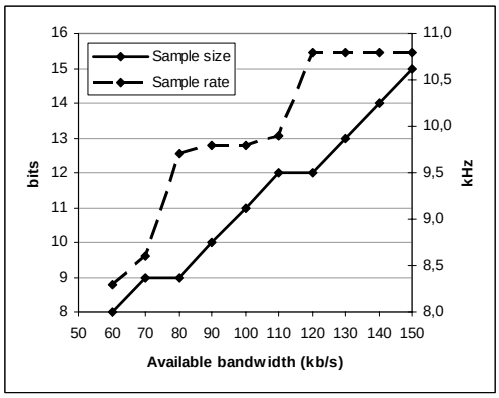

Figure 33: Selected configurations for scenario 6

In scenario 3 and 6, we see that the sample rate increases to between 9.7 and 10 kHz, before the sample size starts to increase. This corresponds with our calculation in section 5.4.3 that showed that the utility function would prefer increasing the sample rate to 9.4 kHz before increasing the sample size. The sample rate is only increased when there is available bandwidth that is not usable by an increase in the sample size, e.g., increasing the sample size to 10 bits at 90 kb/s available bandwidth in scenario 3, would force a sample rate of 9.0 kHz, which is above the 9,4 kHz threshold calculated in section 5.4.3.

There are some interesting observations at 130 kb/s available bandwidth in scenario 3. There is actually enough bandwidth to provide a sample size of 13 bits and 10 kHz sampling rate, but at this point, increasing the sample rate from 0.8 kHz to 10.8 kHz and keeping the sample size at 12 bits yields a slightly higher utility value, 0.20908 versus 0.20857. Also interesting, this configuration consumes slightly less bandwidth than the less optimal configuration, 129.6 kb/s versus 130 kb/s. The higher utility is due to that the sample rate can be increased relatively more than the sample size at this point. At the next step, some of the increase in sample rate is traded for an increase in the sample size at 140 kb/s available bandwidth.

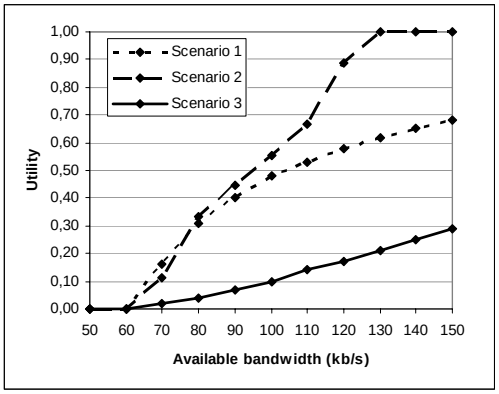

Figure 34: Utility values for scenarios 1-3

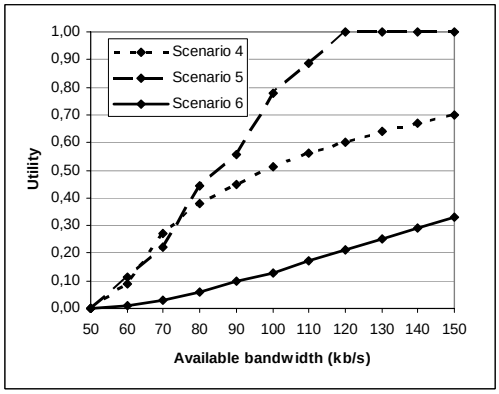

Figure 35: Utility values for scenarios 4-6

Figure 34 and Figure 35 shows the utility values for all the scenarios. As expected, the utility values for scenarios 2 and 5 follow the selected sample sizes, as the utility function is linear. Similarly, the utility values for scenarios 1 and 4 grow logarithmically with the linear increase in sample rate, which should not be a surprise either.

It is harder to analyze the utility values for scenarios 3 and 6. We should see a close to linear increase in the utility until the sample size reaches the maximum value, and then a logarithmic increase as the sample rate increases with the increasing available bandwidth. By extending scenario 3 to be able to reach a utility of 1, it may be simpler to verify the configured utility values with the expected. This requires scenario 3 to be increased to:

16 bits/sample * 44.1 k samples/s = 705.6 kb/s ≈ 710 kb/s

The selected configurations of this extended scenario are shown in Figure 36, and the utility values in Figure 37. Here we see clearly that the utility increases quite quickly to 0.40 at 180 kb/s available network bandwidth, when the sample size has reached its maximum value, and the utility then grows logarithmically as the sample rate grows linearly.

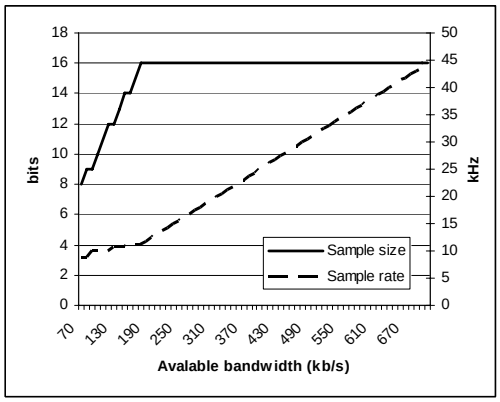

Figure 36: Configurations for extended scenario 1

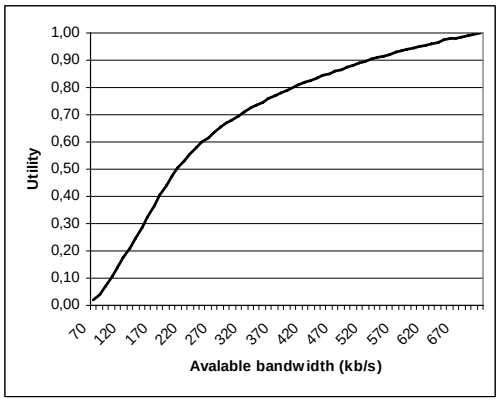

Figure 37: Utility values for extended scenario 1

Figure 38 through Figure 41 shows how much of the available resources that were utilized by the service configurations. In scenario 2, we see that utilization between 80 and 120 kb/s suffers from each increase in available bandwidth is 25% more than is needed to increase the sample size with 1 bit, thus it is not possible to use all the available bandwidth at all steps in this scenario. When we compare the sample sizes for scenario 2 in Figure 30 with scenario 5 in Figure 31, we see that in scenario 5, sample sizes are 1 bit larger than in scenario 2 for the same interval. The extra available bandwidth has been utilized by using the available CPU to compress the larger samples.

We see that in both scenario 2 and 5, the resource utilization drop when the utility reach 1. In all the other scenarios, the utilization of available network bandwidth is close to 100%.

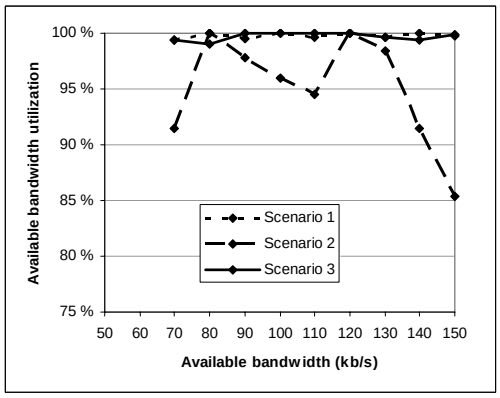

Figure 38: Utilization of available bandwidth for scenarios 1-3

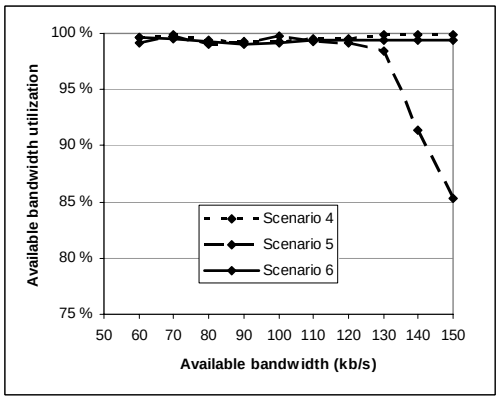

Figure 39: Utilization of available bandwidth for scenarios 4-6

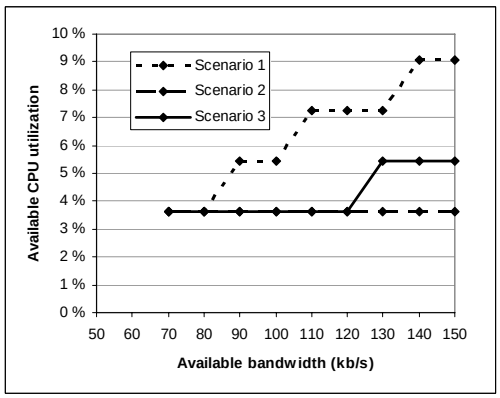

Figure 40: Utilization of available CPU for scenarios 1-3

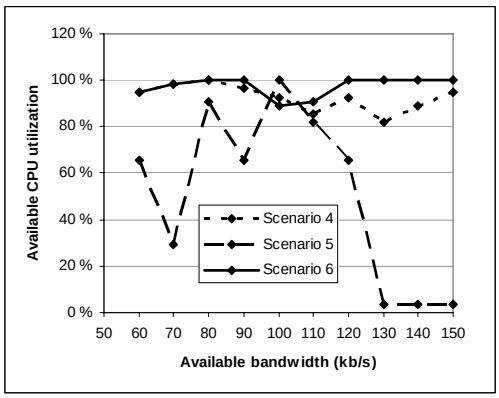

Figure 41: Utilization of available CPU for scenarios 4-6

Scenarios 1 through 3 only require a small portion of the available CPU. The utilization of available CPU in scenarios 4 through 6 follows the compression levels, which are shown in Figure 42. It looks like utilizing network bandwidth has been preferred to CPU when there have been two or more possible configurations to provide the same utility, but there is nothing in the algorithm that may have caused this behavior. The algorithm treats all possible configurations with the same utility equal, so this must be a side-effect of the current implementation.

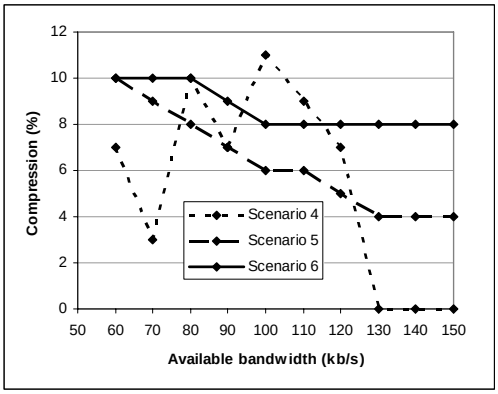

Figure 42: Selected compression for scenarios 4-6

This is where we could plug in a *Resource Planner* using a *resource utility function* to select between configurations that are equal in *service utility*, but with different resource requirements. The resource utility function could be used to weigh the cost of the different resources, i.e., which resource statement to select after a QoS configuration been selected. Table 11 shows the resource configurations to select between for in one of the steps in scenario 5.

Compression (%)	CPU (%)	Bandwidth (kb/s)
3	16	69.8
4	21	69.1
5	26	68.4
6	31	67.7
7	36	67.0
8	40	66.2
9	45	65.5
10	50	64.8
11	55	64.1

Table 11: Compression and resource usage alternatives for scenario 5 at 70 kb/s available bandwidth

Finally, we will look at how long time it took to plan the different services, shown in Figure 43 and Figure 44. The planning times were averaged over 100 runs of each service request. The simpler scenarios and even scenario 3 take negligible time to plan. We see a linear increase in planning time for scenario 4, but even this scenario is planned in less than 40 milliseconds.

The linear increase in scenario 4 from scenario 1 is due to the time it takes to calculate the various resource requirements for the compression levels supported by the **compressor** component.

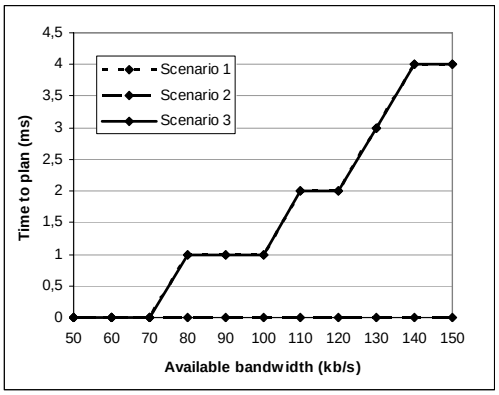

Figure 43: Time to plan service for scenarios 1-3

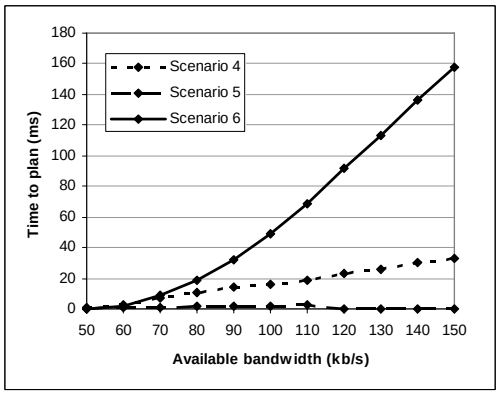

Figure 44: Time to plan service for scenarios 4-6

7.3. Configure video stream

The scenarios for the video stream experiment are repeated in Table 12. We will concentrate on looking at algorithm efficiency for this experiment.

Scenario	QoS dimensions	Utility function	Bandwidth
7	Frame height	u_h	500-4000 kb/s
8	Frame height and frame rate	u_f	500-4000 kb/s
9	Frame height, frame rate and color depth	u_v	500-4000 kb/s

Table 12: Experiment scenarios for video stream experiment revisited

The utility values are plotted in Figure 45. As we see, only scenario 7 reached a utility of 1.0 at 4000 kb/s available bandwidth.

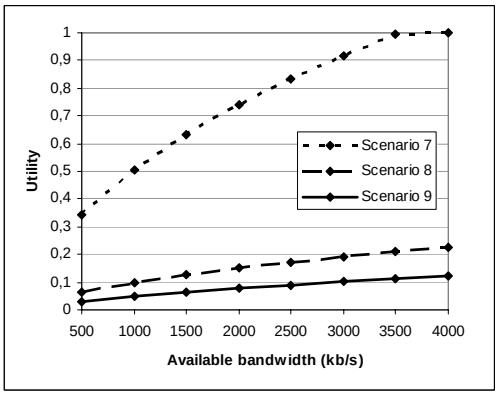

Figure 45: Utility values for scenarios 7-9

Figure 46 shows that the time spent planning with 1 and 2 QoS dimensions are comparable to the results for the audio codec experiment. The planning time is negligible for 1 QoS dimension, and within 200 ms for 2 QoS dimensions. However, at 3 QoS dimensions we see a significant increase in planning time, reaching a maximum of above 1600 ms at 4000 kb/s available network bandwidth.

If we draw the planning time graphs with number of QoS dimensions on the x-axis, as shown in Figure 47, we see a clear hint of an exponential increase in planning time, as the number of QoS dimensions increases. This strengthens the assumption that this problem is NP-hard.

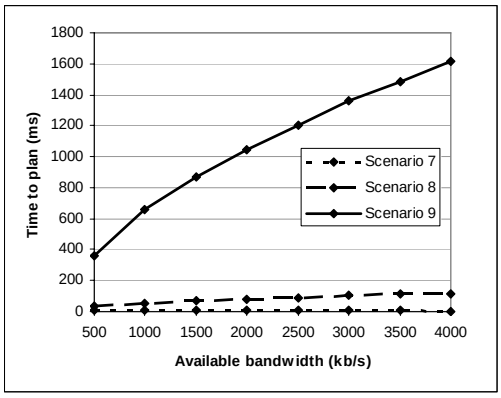

Figure 46: Time to plan for scenarios 7-9, by available bandwidth

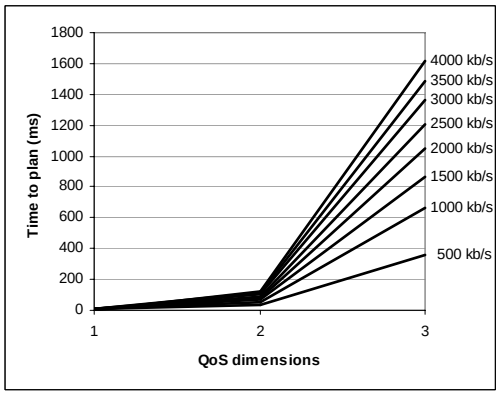

Figure 47: Time to plan for scenarios 7-9, by number of QoS dimensions

7.4. Summary

This chapter has described the results of executing the experiments. Only one of the scenarios needed to be extended to provide better input for analysis. The experiments have been run successfully, and the results will be evaluated in the next and final chapter.

8. Evaluation and conclusion

This chapter evaluates the test of the hypothesis presented chapter 5, and attempts to conclude on how well the problem stated in chapter 4 has been solved. The problem statement and hypothesis are repeated below.

Problem statement:

How can a component middleware efficiently choose a component implementation based on component type, QoS requirements and available resources?

Hypothesis:

It is possible to make a generic solution to selecting and configuring a component implementation when the QoS model and resource model are generic, with as high level of QoS as possible, given limited resource availability.

8.1. Experiment evaluation

8.1.1. Model evaluation

The QuA Java capsule prototype has demonstrated through the successful execution of the experiments that it is indeed possible to use generic approaches to specifying QoS and resource availability in a component based software architecture. The prototype also has showed that these approaches can be combined with a generic solution for selecting and configuring component implementations. This confirms the validity of the hypothesis for a limited number of QoS dimensions.

8.1.2. Precision evaluation

The experiments show that the precision of the algorithm is only bound by the step size in the QoS dimensions. As long as the utility is less than 1, at least one of the limited resources is almost completely exhausted.

8.1.3. Effectiveness evaluation

At 3 QoS dimensions, we see that finding the optimal component configuration takes up to 1.5 seconds, and increases with the search space (bandwidth). Spending 1.5 seconds per component is in within the stated goal.

More disturbing, is the exponential increase in planning time with increasing number of QoS dimensions, and Figure 47 shows that the increase is steep even for exponential functions.

The search space, and thus the search time, could be reduced by increasing the step size for some QoS dimensions, trading decreased precision for increase in effectiveness.

In other words, even though the stated goal has been achieved, we can not conclude on the general effectiveness of the algorithm.

8.2. Generic Implementation Planner feasibility

It seems that a generic approach to selecting a component implementation is feasible within the boundaries of the experiments in this thesis, i.e., 3 QoS dimensions. However, if the number of QoS dimensions is increased to 4, it is uncertain if the algorithm is effective enough, and at 5 QoS dimensions it may be useless. The algorithm needs to be tested with experiments with 4 and 5 QoS dimensions to conclude on this observation though.

If the QoS dimensions that depend on the same set of resources are grouped together, the algorithm could be used to maximize the utility for each of these sets, and then combine the results. This would reduce the complexity of the NP-hard part of the problem, but requires knowledge of component implementations to create these groups.

A divide-and-conquer approach like this could probably be done in a specific implementation planner, which could be programmed with knowledge of how QoS dimensions are realized in a specific QoS domain. The QuA service planner could select a specific implementation planner based on the QoS model associated with the requested QuA component type.

The generic implementation planner was introduced as an alternative to implementing a specific service planner for each QoS domain that a QuA capsule should support. Implementing a QoS domain specific service planner has a cost, in terms of development time. The generic approach could be used when a specific planner is not (yet) available. It may also be useful to experiment with when developing a specific planner, to gather heuristics of possible component configurations, and for comparing the effectiveness of a specific planner in benchmarking.

8.3. Open questions

There are some loose ends to the solution proposed in this thesis. First of all, is that the effectiveness of the algorithm with more than 3 QoS dimensions is unknown.

Another open question is how can the service utility and resource requirements be calculated in a heterogeneous, distributed QuA space, i.e., a QuA space consisting of capsules and components implemented on different platforms.

Finally, how can subjective QoS preferences be transformed to a utility function based on objective measurements? Using utility functions for specifying and measuring QoS depends on this transformation.

8.4. Conclusion

This thesis has proposed a solution to the described problem, through presenting and evaluating a hypothesis. The solution is viable, at least as long as the space of the QoS domain it is applied to is not too big.

In other words, a generic approach to planning QoS-aware services in a component based software architecture is possible on the level of selecting and configuring component implementations. The implemented algorithm may also be reused within specific service planners.

Appendix A – Overview of ISO 9126 – External and Internal Quality Metrics

The following in an excerpt from Aagedal (2003), and is provided here for reference.

Functionality

The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions.

Suitability

The capability of the software product to provide an appropriate set of functions for specified tasks and user objectives.

Accuracy

The capability of the software product to provide the right or agreed results or effects with the needed degree of precision.

Interoperability

The capability of the software product to interact with one or more specified systems.

Security

The capability of the software product to protect information and data so that unauthorized persons or systems cannot read or modify them and authorised persons or systems are not denied access to them.

Functionality compliance

The capability of the software product to adhere to standards, conventions or regulations in laws and similar prescriptions relating to functionality.

Reliability

The capability of the software product to maintain specified level of performance when used under specified conditions.

Maturity

The capability of the software product to avoid failure as a result of faults in the software.

Fault tolerance

The capability of the software product to maintain a specified level of performance in cases of software faults or of infringement of its specified interface.

Recoverability

The capability of the software product to re-establish a specified level of performance and recover the data directly affected in the case of a failure.

Reliability compliance

The capability of the software product to adhere to standards, conventions or regulations relating to reliability.

Usability

The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions.

Understandability

The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use.

Learnability

The capability of the software product to enable the user to learn its application.

Operability

The capability of the software product to enable the user to operate and control it.

Attractiveness

The capability of the software product to be attractive to the user.

Usability compliance

The capability of the software product to adhere to standards, conventions, style guides or regulations relating to usability.

Efficiency

The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions.

Time behaviour

The capability of the software product to provide appropriate response and processing times and throughput rates when performing its function, under stated conditions.

Resource utilisation

The capability of the software product to use appropriate amounts and types of resources when the software performs its function under stated conditions.

Efficiency compliance

The capability of the software product to adhere to standards or conventions relating to efficiency.

Maintainability

The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.

Analysability

The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.

Changeability

The capability of the software product to enable a specified modification to be implemented.

Stability

The capability of the software product to avoid unexpected effects from modifications of the software.

Testability

The capability of the software product to enable modified software to be validated.

Maintainability compliance

The capability of the software product to adhere to standards or conventions relating to usability.

Portability

The capability of the software product to be transferred from one environment to another.

Adaptability

The capability of the software product to be adapted for different specified environments without applying actions or means other than those provided for this purpose for the software considered.

Installability

The capability of the software product to be installed in a specified environment.

Co-existence

The capability of the software product to co-exist with other independent software in a common environment sharing common resources.

Replaceability

The capability of the software product to be used in place of another specified software product for the same purpose in the same environment.

Portability compliance

The capability of the software product to adhere to standards or conventions relating to portability.

References

Abrahamsen, Espen. Unpublished. Resource Model for the QuA Platform.

Aurrecoechea, Christina, Andrew T. Campbell, and Linda Hauw. 1998. A Survey of QoS Architectures. *Multimedia Systems* 6 (3):138-151.

Blair, Gordon, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fabio Costa, Hector Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui Moreira, Nikos Parlavantzas, and Katia Saikoski. 2001. The Design and Implementation of Open ORB 2. *IEEE Distrib. Syst. Online* 2 (6).

Brown, William H., Raphael C. Malveau, Hays W. "Skip" McCormick III, and Thomas J Mowbray. 1998. *Anti Patterns*. New York City: John Wiley & Sons, Inc.

Chiba, Shigeru. 2000. Load-time structural reflection in Java. Paper read at ECOOP 2000.

Cleetus, Anita Maria. 2004. An Implementation Of The OpenCOM Core CFs. MSc dissertation, Distributed Systems Engineering, University of Lancaster.

Coulouris, George, Jean Dollimore, and Tim Kindberg. 2001. CORBA case study. In *Distributed Systems - Concepts and design*. Essex: Pearson Education Ltd.

Coulson, Geoff, Gordon S. Blair, Michael Clarke, and Nikos Parlavantzas. 2002. The design of a configurable and reconfigurable middleware platform. *Distributed Computing* 15 (2):109-126.

Dunn, Julian. 1998. Anti-alias and anti-image filtering: The benefits of 96kHz sampling rate formats for those who cannot hear above 20kHz. Paper read at 104th AES Convention, at Amsterdam.

Eliassen, Frank, Richard Staehli, Jan Øyvind Aagedal, Arne-Jørgen Berre, Anders Andersen, and Gordon Blair. 2002. Excerpt from Quality of Service Aware Component Architecture Project Proposal: Simula Research Laboratory, SINTEF Oslo, University of Tromsø.

Frølund, Svend, and Jari Koistinen. 1990. Quality of Service Aware Distributed Object Systems. Paper read at 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99), at San Diego.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison Wesley.

Gold, Nicolas, Andrew Mohan, Claire Knight, and Malcolm Munro. 2004. Understanding service-oriented software. *IEEE Software* 21 (2):71-77.

Grace, Paul, Gordon Blair, and Sam Samuel. 2003. Interoperating with Services in a Mobile Environment: Lancaster University.

Göbel, Steffen, Christoph Pohl, Simone Röttger, and Steffen Zschaler. 2004. The COMQUAD Component Model - Enabling Dynamic Selection of Implementations by Weaving Non-functional Aspects. Paper read at 3rd International Conference on Aspect-Oriented Software Development, at Lancaster, UK.

ITU-T. 1993. Pulse Code Modulation (PCM) of Voice Frequencies. In *ITU-T Recommendation G.711*: International Telecommunication Union.

———. 1996. Methods for subjective determination of transmission quality. In *ITU-T Recommendation P.800*: International Telecommunication Union.

Koistinen, Jari, and Aparna Seetharaman. 1998. Worth-Based Multi-Category Quality-of-Service Negotiation in Distributed Object Infrastructures. Paper read at Enterprise Distributed Object Computing Workshop, 3-5 Nov 1998, at La Jolla, CA, USA.

Kon, Fabio, Fabio Costa, Gordon Blair, and Roy H. Campbell. 2002. The case for reflective middleware. *Communications of the ACM* 45 (6):33-38.

Kon, Fabio, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. 2000. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. Paper read at Middleware 2000.

Loyall, Joseph P., Richard E. Schantz, John A. Zinky, and David E. Bakken. 1998. Specifying and measuring Quality of Service in distributed object systems. Paper read at ISORC, at Kyoto, Japan.

McKinley, Philip K., Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. 2004a. Composing Adaptive Software. *IEEE Computer* 37 (7):56-64.

———. 2004b. A Taxonomy of Compositional Adaptation. Michigan: Software Engineering and Network Systems Laboratory, Department of Computer Science and Engineering, Michigan State University.

Miguel, Miguel Angel de, José F. Ruiz, and Marisol García-Valls. 2002. QoS-Aware Component Frameworks. Paper read at 10th International Workshop on Quality of Service, May 2002.

Mukhija, Arun, and Martin Glinz. 2004. A Framework for Dynamically Adaptive Applications in a Self-Organized Mobile Network Environment. Paper read at 24th International Conference on Distributed Computing Systems Workshops.

Papazoglou, M. P., and D. Georgakopoulos. 2003. Service-Oriented Computing. *Communications of the ACM* 46 (10):25-28.

Peterson, Larry L., and Bruce S. Davie. 2000. *Computer Networks - A Systems Approach*: Morgan-Kaufman.

Poladian, Vahe, João Pedro Sousa, David Garlan, and Mary Shaw. 2004. Dynamic Configuration of Resource-Aware Services. Paper read at 26th International Conference on Software Engineering, at Edinburgh.

Rajkumar, Ragunathan, Chen Lee, John P. Lehoczky, and Daniel P. Siewiorek. 1997. A Resource Allocation Model for QoS Management. In *18th IEEE Real-Time System Symposium*.

———. 1998. Practical Solutions for QoS-Based Resource Allocation. In *RTSS*.

Schantz, Richard E., Joseph P. Loyall, Michael Atighetchi, and Partha Pal. 2002. Packaging Quality of Service Control Behaviours for reuse. Paper read at ISORC, at Washington DC, USA.

Schmidt, Douglas C. 2002. Middleware for real-time and embedded systems. *Communications of the ACM* 45 (6):43-48.

Schmidt, Douglas C, and Steve Vinoski. 2001. Object Interconnections: Real-time CORBA, Part 1: Motivation and Overview. *C/C++ Users Journal*.

———. 2004. Object Interconnections: The CORBA Component Model: Part 1, Evolving Towards Component Middleware. *C/C++ Users Journal*.

Sommerville, Ian. 1995. *Software Engineering*. 5th ed. Essex: Addison-Wesley.

Staehli, Richard, and Frank Eliassen. 2002. QuA: A QoS-Aware Component Architecture.

———. 2004. Compositional Quality of Service Semantics. Paper read at Specification and Verification of Component Based Systems (SAVCBS04), Workshop at ACM SIGSOFT.

Staehli, Richard, Frank Eliassen, Gordon Blair, and Jan Øyvind Aagedal. 2003. QoS-Driven Service Planning in an Open Component Architecture.

Sun Microsystems. 2003. Enterprise JavaBeans Specification, Version 2.1: Sun Microsystems.

Szyperski, Clemens. 2002. *Component Software*. Edited by C. Szyperski. 2nd ed, *Component Software Series*: Addison-Wesley.

Tanenbaum, Andrew S. 1996. *Computer Networks*. 3rd ed: Prentice-Hall.

Wang, Nanbor, Krishnakumar Balasubramanian, and Chris Gill. 2002. Towards a Real-time CORBA Component Model. Paper read at OMG Workshop On Embedded & Real-Time Distributed Object Systems, July, at Washington D.C.

Wikipedia. 2007. *Shannon-Nyquist sampling theorem*, 2007-03-05 2007 [cited 03-05 2007]. Available from http://en.wikipedia.org/wiki/Nyquist-Shannon_sampling_theorem.

Zachariadis, Stefanos, and Cecilia Mascolo. 2003. Adaptable mobile applications through SATIN: exploiting logical mobility in mobile computing middleware. Paper read at 1st UK-UbiNet Workshop, 25-26 September 2003, at Imperial College, London.

Aagedal, Jan Øyvind. 2001. Quality of Service Support in Development of Distributed Systems. Ph.D. thesis, Department of Informatics, University of Oslo, Oslo.

———. 2003. Quality of Service Support in Software Architecture. In *Lecture no 8 in IN-MMO - Modellering med objekter*.