QoS Contract Negotiation in Distributed Component-Based Software

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

M.Sc Mulugeta Dinku, Mesfin
geboren am 27. März 1972 in Harar

Gutachter:	Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill (TU Dresden)
	Prof. Dr. rer. nat. habil. Uwe Aßmann (TU Dresden)
	Prof. Dr. Frantisek Plasil (Charles University, Prague, Czech Republic)

Tag der Verteidigung: 15. Juni 2007

Dresden, im April 2007

Abstract

Currently, several mature and commercial component models (for e.g. EJB, .NET, COM+) exist on the market. These technologies were designed largely for applications with business-oriented non-functional requirements such as data persistence, confidentiality, and transactional support. They provide only limited support for the development of components and applications with non-functional properties (NFPs) like QoS (e.g. throughput, response time). The integration of QoS into component infrastructure requires among other things the support of components' QoS contract specification, negotiation, adaptation, etc. This thesis focuses on contract negotiation.

For applications in which the consideration of non-functional properties (NFPs) is essential (e.g. Video-on-Demand, eCommerce), a component-based solution demands the appropriate composition of the QoS contracts specified at the different ports of the collaborating components. The ports must be properly connected so that the QoS level required by one is matched by the QoS level provided by the other. Generally, QoS contracts of components depend on run-time resources (e.g. network bandwidth, CPU time) or quality attributes to be established dynamically and are usually specified in multiple QoS-Profiles. QoS contract negotiation enables the selection of appropriate concrete QoS contracts between collaborating components. In our approach, the component containers perform the contract negotiation at run-time.

This thesis addresses the QoS contract negotiation problem by first modelling it as a constraint satisfaction optimization problem (CSOP). As a basis for this modelling, the provided and required QoS as well as resource demand are specified at the component level. The notion of utility is applied to select a good solution according to some negotiation goal (e.g. user's satisfaction). We argue that performing QoS contract negotiation in multiple phases simplifies the negotiation process and makes it more efficient. Based on such classification, the thesis presents heuristic algorithms that comprise *coarse-grained* and *fine-grained* negotiations for collaborating components deployed in distributed nodes in the following scenarios: (i) single-client - single-server, (ii) multiple-clients, and (iii) multi-tier scenarios.

To motivate the problem as well as to validate the proposed approach, we have examined three *componentized* distributed applications. These are: (i) video streaming, (ii) stock quote, and (iii) billing (to evaluate certain security properties). An experiment has been conducted to specify the QoS contracts of the collaborating components in one of the applications we studied. In a run-time system that implements our algorithm, we simulated different behaviors concerning: (i) user's QoS requirements and preferences, (ii) resource availability conditions concerning the client, server, and network bandwidth, and (iii) the specified QoS-Profiles of the collaborating components. Under various conditions, the outcome of the negotiation confirms the claim we made with regard to obtaining a good solution.

Acknowledgement

First and foremost I would like to thank my advisor Prof. Dr. Alexander Schill for giving me the opportunity to work under the creative atmosphere of his chair of computer networks. His guidance, constant follow-up, critical comments, and prompt feedbacks throughout the study period have contributed a lot to my success. I am so grateful to him and look forward to working with him in the future.

My second advisor Prof. Uwe Aßmann has also played an important role in the successful completion of the thesis by allowing me to make presentations to his research group and providing me critical comments on the draft version of the thesis. My heartfelt gratitude goes to Prof. Dr. Frantisek Plasil for agreeing to be my external reviewer and providing me essential comments on the draft version of the thesis. Special thanks also go to Dr. Steffen Göbel, Dr. Christoph Pohl, and Dr. Steffen Zscaler for their helpful ideas during the initial phase of the study. All the feedbacks and comments I received have helped me straighten out my thoughts on the subject areas of this thesis.

I wish to thank my dear parents whose love and prayers have always been helpful in my life. I am grateful to my dear wife Messeret for her love, constant encouragement, understanding, and support, which have been vital to my success. My children Matthewos and Sitota have been my sources of strength when tackling the otherwise difficult and daunting challenges I have faced during the study.

I want to extend my gratitude to the support of all my friends here in Dresden and elsewhere. The e-mail exchanges with my Bole friends concerning various life issues have been helpful in either sharpening my ideas or giving me a break during the difficult and monotonous times of the study period. I am truly fortunate to have such good friends.

I am always indebted to my Heavenly Father for providing me the strength and courage in everything I endeavor in life.

Finally, very special thanks go to the DAAD (Deutsche Akademischer Austausch Dienst) for their sponsorship of the PhD study including the language course.

TABLE OF CONTENT

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1 Motivation

Component-Based Software Engineering (CBSE) allows the composition of complex systems and applications out of well defined parts (components). This has a number of advantages including effective management of complexity, reduced time to market, increased productivity, and improved quality.

Today's and future applications demand the integration of non-functional requirements in order to meet the various needs of stakeholders. For example, video streaming and eCommerce applications, which are soft real-time, require the integration of non-functional requirements to enable the provision of QoS differentiation in the offered services. Other application areas like Distributed Real-time and Embedded Systems [Wang et al, 2004a] have stringent individual QoS requirements (e.g. bounded response time), which must be satisfied simultaneously. Failure to meet these QoS requirements, e.g. a missed deadline, often leads to serious consequences even if an application may still logically function properly.

Currently, several mature and commercial component models (EJB, .NET, CCM, COM+, etc.) exist on the market. These technologies were designed largely for applications with business-oriented QoS requirements such as data persistence, confidentiality, and transactional support. They provide only limited support for the development of components and applications with non-functional properties (NFPs) like QoS and security.

In order to address the deficiencies of the mainstream component technologies and bring the benefits of CBSE for the development of real-time and multimedia applications, a lot of research has been and is currently being done. These efforts range from creating new component models [Göbel et al, 2004a] to extending the existing mainstream component models [Wang et al, 2004a][Ulbrich et al, 2003] [Vecellio et al, 2002]. To complement these ongoing efforts, the OMG has issued a Request for Proposal (RFP) in Quality of Service for CORBA Components. The Request for Proposals (RFP) solicits proposals that facilitate the support for non-functional requirements in the context of the CORBA Component Model [OMG, 2003].

To support third-party compositions, components must be fully specified in terms of functional and non-functional properties [Szyperski, 2002]. Components in the widely available component models (EJB, .NET, CCM), are specified with interfaces that provide syntactical information about which methods are available and how to invoke them. But, this underspecifies the components. In [Beugnard et al, 1999] four different levels of contracts have been identified for components in a component-based application. These are: syntactic, behavioral, synchronization, and QoS contracts. The explicit consideration of component contracts aims at simplifying the development of component-based applications with non-functional requirements like QoS and security, but it is also a challenging task.

Our focus in this thesis is on addressing issues related to the fourth type of component contract, which is the *QoS contract*. A video decoder component, for instance, in a video streaming application has an input interface that expects a stream from a video server component. It also has an output interface that offers the received stream to a video rendering component. The NFPs associated with the input and output interfaces may be frame rate, delay, and resolution. The values of these properties, unlike the functional properties, are dependent on the level of available resources and QoS obtained from interacting components. Hence, a QoS contract is a dynamic entity unlike a syntactic contract that is static. A component's QoS contract is distinguished into *offered QoS contract* and *required QoS contract* [OMG, 2005]. The *offered QoS contract* of a component specifies the quality values that the component can provide to its clients (other components) while the *required QoS contract* of a component specifies the QoS constraints that the clients of the component must achieve. There is dependency between *offered* and *required QoS contracts.*

1.2 Related Work[1]

The integration of QoS in component infrastructures is a subject of very short history [Bouyssounouse and Sifakis (eds.), 2005]. Such integration must deal with a number of issues. Some of these are the specification of component contracts, contract negotiation, resource management, adaptation, etc. As our focus in this thesis lies on QoS contract negotiation, we analyze the various related works only from the view point of negotiation and associated features.

Over the last decade, many researchers have been working on static and dynamic QoS management mechanisms for distributed applications, which are deployed on *object-oriented* middleware like CORBA or Java RMI. The Quality objects (QuO) framework offers one of the most advanced concepts and necessary tools to integrate QoS into distributed applications based on CORBA [Zinky et al, 1997; Loyall et al, 1998]. QuO defines QDL (Quality Description Language) for describing *contracts*, *system condition objects*, and the adaptive behavior of objects and *delegates*. But QDL does not properly address *component QoS contracts* as it is not possible to express the context dependencies of components [Bouyssounouse and Sifakis (eds.), 2005]. QoS contracts specified in QDL describe the adaptive behavior of the application. However, having to provide this description explicitly is a burden on the programmer (or more specifically on the QoS designer). The solutions proposed in [Miguel, 2001] integrate the resource reservation and QoS IP in Java RMI classes.

The work in [Göbel et al, 2004b], which motivated this work to some degree, offers basic QoS mechanisms but only in a single container. Their approach ties the container-based negotiation to a real-time operating system called DROPS. They have not pursued the

[1] Note that some of the researches presented here in brief and also other related work are discussed broadly in Chapter 2.

case of distributed applications where components are deployed in multiple containers. Moreover, no strategies have been proposed for a multiple-clients scenario.

The QuA project [Staehli et al, 2004; Amundsen, et al, 2004] aims at defining an abstract component architecture that also includes the semantics for general QoS specifications. This work does not explicitly consider component QoS contracts for QoS provisioning. Nevertheless, their notion of *QoS-driven Service Planning* has some similarities to our concept of contract negotiation. They do not, however, consider complexity issues in their service planning.

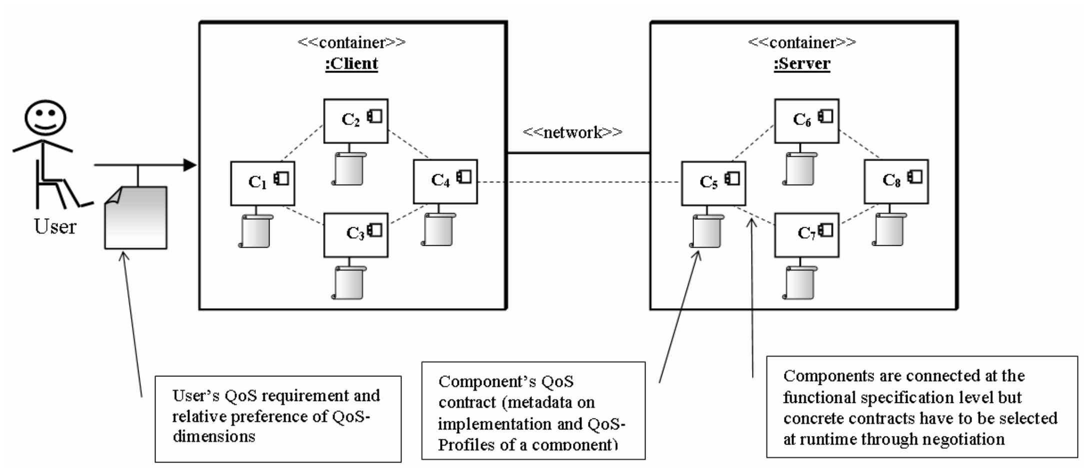
In [Ritter et al, 2003] QoS contract negotiation is applied when two components are explicitly connected via their ports. In the negotiation, the client component contacts the server component by providing its requirement; the server responds with a list of concrete contract offers; and the client finally decides and chooses one of the offers. Their approach covers only the protocol aspect of the negotiation process. They have not pursued the decision making aspects, which are an important element of any automatic negotiation research.

1.3 Problem Statement

As explained with respect to closely related research (subsection 1.2) and also in the several related work that will be discussed in Chapter 2, the integration of QoS Contract negotiation into a component technology has not been sufficiently addressed in previous and current research. This thesis, hence, aims at filling this gap.

For applications in which the consideration of NFPs is essential (e.g. Video-on-Demand and eCommerce), a component-based solution demands the appropriate composition of the QoS contracts specified at the different ports of the collaborating components. The ports must be properly connected so that the QoS level required by one is matched by the QoS level provided by the other. This matching requires the selection of appropriate QoS contracts at each port. When QoS contracts are known statically, the developer or assembler can select the right concrete (provided and required) QoS contracts of each component and compose the whole application during design, implementation, or deployment time. But, for composing QoS contracts that depend on run-time resource conditions (e.g. network bandwidth, CPU) or quality attributes to be fixed dynamically, the selection of appropriate QoS contracts must be carried out at run-time by the process of *QoS Contract Negotiation*. In our approach, the run-time environment, in particular the component container, performs the selection of the appropriate concrete QoS contracts for the components at run-time based on a number of different criteria.

We consider applications whose components are deployed on distributed nodes, for example on the service provider and on the client node. The basic client/server application scenario we are studying is shown in Figure 1.1. As the figure depicts, the problem with QoS contract negotiation is how to find appropriate provided and required QoS contracts of the collaborating components on condition that a user's QoS requirements and preferences, components' QoS contracts, and resource conditions (not shown in the figure) are known.



<<container>>
:Client
<<container>>
:Server
<<network>>
C_1
C_2
C_3
C_4
C_5
C_6
C_7
C_8
User
User's QoS requirement and relative preference of QoS-dimensions
Component's QoS contract (metadata on implementation and QoS-Profiles of a component)
Components are connected at the functional specification level but concrete contracts have to be selected at runtime through negotiation

Figure 1.1: Basic Client / Server Application Scenario

Figure 1.1 depicts a *single client – single server* scenario. There are more general scenarios of the problem such as is the case with (i) multiple-clients, (ii) multi-tier (multiple servers), and (iii) peer-to-peer. In the multiple-clients scenario, more than one client is engaged in negotiation with a server simultaneously. In the multi-tier scenario, chained containers are involved in the provisioning of a service, i.e., a component may require the service of another component located in a separate server container in order to give the requested service to its clients. In a peer-to-peer case, interacting components function both as client and server. Each of the aforementioned scenarios poses its own specific challenges as will be explained in subsequent sections.

1.4 Research Challenges

QoS contracts of components are specified with multiple QoS-profiles. The challenges concerning the selection of appropriate QoS-profiles are listed below.

1) Find a solution that satisfies a number of different types of constraints.

The selections must fulfil a number of constraints that exist in a single container or across containers. For example, there could be resource constraints in the underlying platform, i.e. at each node and the network. There also exist conformance constraints between required and provided QoS contracts for one or more QoS-dimensions (e.g. delay, frame rate, resolution, etc.). Users might have certain minimum QoS requirements towards the application. Several users/clients with possibly different requirements may be involved in the negotiation.

2) Find a "better" solution.

There could be several solutions that satisfy the different constraints mentioned above. But, just fulfilling a user's minimum QoS requirement might not be enough. A solution that maximally satisfies a user's requirement and preferences may be a more desirable choice. The question here is how to choose a "better" solution from the set of possible solutions. One of the difficulties with this lies in defining what "better" means. For instance, what is "better" for the service provider might not be so for the consumer. Problems of this type and those stated in 1) above are known to be NP-hard. Hence, heuristics must be applied to determine the appropriate solutions.

3) Find the solution in an efficient way.

As the negotiation is performed dynamically, considering the performance of the agreement process is important. For example, it may be preferred to agree on some acceptable solution in a reasonable time rather than getting the best solution by taking a much longer time.

4) Integrate (at least from architecture point of view) the whole approach into a component framework.

One of the tasks involved in the integration is how to incorporate the proposed mechanisms used to tackle the challenges listed 1) to 3) above in a component framework. The QoS specification languages currently available support only a small aspect of QoS contract negotiation.

1.5 Scope and Approach

The contract negotiation framework to be proposed presupposes that there exists a means for the specification of NFPs of individual components. QML [Frølund and Koistinen, 1998] and CQML [Aagedal, 2001] are examples of such a specification language. This thesis uses CQML^{+} [Röttger and Zschaler, 2003], which is an extension of CQML, to specify QoS contracts of components.

Different applications may need to consider several NFPs such as reliability, availability, security, performance, etc. Treating all of these together is a complex task and trying to come up with unifying solutions is an even more formidable challenge. In this thesis, we focus on timing properties. We also consider certain aspects of security properties. Furthermore, we target soft real-time applications.

Some of the primary goals of this research are:

1) To propose a mechanism for QoS contract negotiation in a distributed component based application. Similar problems have been proven to be NP-hard [Lee et al, 1999]. In this regard, we want to propose heuristics that help in making the negotiation process efficient.

2) To demonstrate the proposed ideas by taking example scenarios from Video Streaming and Stock Quote Applications. In particular, our work focuses on a *componentized* version of these applications.
3) To propose a contract negotiation framework that can be integrated into a component framework to enable QoS contract negotiation for distributed component based solutions.

One of the guiding principles concerning the integration of QoS management in the component architecture is the separation of "business code" from "QoS management code." This relieves the application developer from the system details and concentrates on the business domain and at the same time makes the developed components more flexible and reusable. This principle of separation is pivotal for our integration of contract negotiation into component architecture.

1.6 Research Contributions

The main contribution of our work is proposing a framework for the QoS Contract Negotiation of components in distributed component based applications that are deployed in a resource-constrained and dynamic environment. Our whole approach is done in the context of CBSE. We use the notion of a component as defined by Szyperski [Szyperski, 2002]. The following aspects make our work distinct from the related works.

1) We have demonstrated the applicability of our approach by: (i) specifying the QoS Contracts of each interacting component, and (ii) composing the provided and required QoS contracts of connected components dynamically through the process of QoS contract negotiation according to a certain negotiation goal. We have validated these steps by taking different scenarios from componentized distributed applications of (i) video streaming (including booking and billing), and (ii) stock quote applications.
2) Our approach addresses the three important topics in automated negotiation research, which are Negotiation Objects, Negotiation Protocols, and Decision Making Models.
3) We have generalized our approach for different application scenarios: (i) multi-tier (chained containers), and (ii) multiple clients.
4) Our QoS Contract Negotiation Framework addresses issues that have not been captured in presently available QoS specification languages (e.g. QML, CQML, CQML$^+$, etc.). In fact, the specification languages cannot describe all relevant aspects of QoS contract negotiation.

In the process of achieving our research goal, we have introduced some unique ideas that would be useful in the integration of QoS in component technologies. These are:

1) Classification of QoS contract negotiation as a multi-phase process: *coarse-grained* and *fine-grained* negotiation. This helps us to better understand the nature of NFPs. Following this classification, NFPs can be categorized as coarse-grained and fine-grained properties.
2) Modeling the component QoS contract negotiation problem as a Constraint Satisfaction Problem (CSP); as Partial CSP, when the original problem is found

to be over-constrained; and as Constraint Satisfaction Optimization Problem (CSOP), when good solutions are required.

1.7 Thesis Overview

This thesis is divided into six chapters. The first chapter elaborates on the motivation for the research and outlines the problem statement, the scope of the research and major contributions of the thesis. In Chapter 2 we examine the state-of-the-art with regard to the integration of component technology and QoS and summarize closely related research and projects that are mainly concerned with supporting QoS issues in object-oriented and component middleware. The related researches are also analyzed in view of the scopes and objectives of this thesis.

In Chapter 3 we look into the core of our approach in detail. Based on the research challenges identified in Chapter 1, we formalize QoS contract negotiation as a constraint solving problem. We show how a phased negotiation simplifies the agreement process and makes it more efficient. Pertaining to such classification, we present different algorithms. Chapter 3 concludes with a case study of a componentized video-streaming application. We evaluate the proposed approach based on this example. Chapter 4 extends the single-client – single-server scenario addressed in Chapter 3 into two other scenarios: multiple-clients and multi-tier. In each case, after analyzing different examples to motivate the scenarios, generalized algorithms will be presented.

Chapter 5 introduces a QoS contract negotiation framework that identifies important active components, data entities, and the interaction of different instances of these. The framework depicts how the different negotiation mechanisms designed in the previous two chapters are applied to realize the framework. In Chapter 6, we summarize the findings presented in this thesis and examine how the work can be taken further.

2 Foundations and Review of Existing Literature

In this chapter, we briefly highlight the state-of-the-art with regard to the integration of component technology and QoS by starting from a discussion on some fundamental aspects of the component technology. We also summarize and analyze closely related research and projects that are mainly concerned with supporting QoS issues in object-oriented and component middleware.

2.1 Component-Based Software Engineering (CBSE)

2.1.1 Introduction

Component-based software development has recently emerged as a key technology to cope with the ever challenging issues of software engineering - the production of complex software systems, increased productivity, etc. The technology, however, isn't new and has been recognized in the software community for the past several decades. For instance, in 1968, in Garmish, Germany, at the NATO software engineering conference, Douglas McIlroy presented a paper with the title *Mass Produced Software Components*, in which he argued that the software industry is weakly founded, and that one aspect of this weakness is the absence of a software components sub-industry.

The recent popularity of this technology can be attributed to a number of factors. According to [Brown and Wallnau, 1996], advances in (i) the object-oriented development approach; and (ii) the economic reality that large-scale software development must take greater advantage of existing commercial software; spur the renaissance in the component-based approach. It has also been viewed by many that the vision of early day components can come to fruition due to the recent advances in technology. According to [Clements, 1995], some recent new exciting aspects of CBSD are: (i) the availability of off-the-shelf components that have a wide range of functionality, (ii) the fact that the coordination and communication infrastructure is being acknowledged as a component that is potentially available in pre-packaged form, and (iii) the realization and attempt to address non-technical issues that must be solved in order to make CBSD work.

It is believed by many experts in the field that the component technology can be considered as the natural evolution of object technology [Meyer, 1999]. It is noted in [Crnkovic, 2001] that there is a strong relation between object-oriented programming (OOP) and components. Component models like COM/DCOM, .NET, EJB, CCM relate component interfaces to class interfaces. Moreover, components adopt object principles of unification of functions and data encapsulation. There are, however, others who advocate a different view. We find in [Brown and Wallnau, 1998] that object technology is neither necessary nor sufficient for CBSE. This implies that it is possible to realize component technology without employing object technology.

The major goals of CBSE are: (i) the provision of support for the development of software systems as assemblies of components, (ii) the development of components as reusable entities, and (iii) the maintenance and upgrading of systems by customising and replacing their components [Heineman and Councill (eds.), 2001]. The object-oriented approach emphasizes programming over assembly (composition) and aims at modelling real-world applications with objects and their interaction.

It appears that there is not a general consensus on the definition of a component in the literature. The most widely used and adopted definition in the software community is the one given by Szyperski, which is: *a software component is a unity of composition with contractually specified interface and fully explicit context dependencies that can be deployed independently and is subject to composition by third parties* [Szyperski, 2002]. Some argue that this definition isn't general enough as it requires components to be binary. Cheesman and Daniels' classification also gives an interesting insight into the notion of a component. According to [Cheesman and Daniels, 2001], a component can exist in several forms during its life time: Component Specification; Component Implementation; Installed Component; and Component Object.

Szyperski's notion of "contractually specified interface and explicit context dependencies only" signifies that: (i) a component should be specified not only with a *provides* interface, i.e. what it provides to interacting components, but also with a *requires* interface, i.e. what it needs from the interacting components, and (ii) the specification should not only include functional properties but also non-functional properties (this includes among others QoS properties, resources required from the underlying platform, etc.). Current component based technologies usually have a *provides* interface and support the specification of functional properties, but address only limited aspect of non-functional properties (e.g. transactions).

We will further explore some of the technical concepts underlying component technology, notably component models, frameworks, and contracts (especially with regard to QoS properties of a component).

2.1.2 Component Models

A component model specifies the standards and conventions imposed on developers of components in terms of the set of component types, their interfaces, and additionally the allowable patterns of interaction among component types [Bachman et al, 2000]. A closely related concept is a component framework that provides a variety of run-time services (e.g. transaction, persistence, etc.) to support and enforce a component model. The specifications of the widely used component models like Sun's EJB or Microsoft's COM+ describe the various standards and conventions to be followed by component or container developers of the respective component technologies. The containers serve as a component framework.

It is to be noted that none of the mainstream component models (i.e. EJB, .NET, COM+) comply with Szyperski's component definition. For instance, the explicit context dependencies criterion isn't met by these component models. Components developed using these models can contain only a *provides* interface. That implies these components can only specify the services they provide or offer to other components. In our work, we require a component to have a *uses* interface in addition to a *provides* interface. The *uses* interface specifies services required by the component from other components.

The component model upon which this thesis builds is the COMQUAD component model [Göbel et al, 2004a], which extended Sun's EJB and the OMG's CORBA Component Model (CCM) with the following concepts: (i) the specification of non-functional issues, (ii) container-managed instantiation and connection of component implementations based on the specific quantitative capabilities of the system, and (iii) streaming interfaces. As far as the connection of component implementations through contract negotiation is concerned, the features provided by the COMQUAD's run-time environment are limited as explained in section 1.2.

2.1.3 Component QoS Contracts

Component contract can be taken to mean *"component specification"* in any form. A *contract* specifies functional or extra-functional properties of a component, which are observable in its interface [Bouyssounouse and Sifakis (Eds.), 2005]. This definition clarifies the relationship between an interface and a contract in that a contract can be seen as specifying constraints on the interface of a component. In [Beugnard et al, 1999] four different levels of contracts have been identified for components in a component-based application. These are: syntactic, behavioral, synchronization, and QoS contracts.

Syntactic contract specifies (i) the operations a component can perform, (ii) the input and output parameters each component requires, and (iii) the possible exceptions that might be raised during operation. *Behavioral contract* constrains values of parameters and of persistent state variables, expressed by pre- and post-conditions and invariants. *Synchronization contract* specifies the global behavior of objects in terms of synchronizations between method calls. *QoS contract* makes constraints on the extra-functional or non-functional properties like response time, throughput, etc. Since our focus in this thesis is on the negotiation of QoS contracts, the subsequent discussions concentrate only on this type of contract.

A component's QoS contract is distinguished into *offered QoS contract* and *required QoS contract* [OMG, 2005]. QoS specification languages are used to describe QoS contracts as the discussion in the next section outlines.

2.1.4 QoS Contract Specification

There exist different description languages for specifying the QoS contract of components. In this section we discuss two such languages: QML [Frølund and Koistinen, 1998] and CQML [Aagedal, 2001] together with CQML$^+$[Röttger and Zschaler, 2003], which is an extension of CQML. At the end, we summarize related issues from an OMG's standard document on the subject of QoS specification.

2.1.4.1 QML

QML has three main abstraction mechanisms for QoS specification: *contract type*, *contract*, and *profile*. A *contract type* defines the dimensions that can be used to characterize a particular QoS category (e.g. performance or reliability). A *contract* is an instance of a contract type and represents a particular QoS specification. The third fundamental concept in QML, i.e. a *profile* associates contracts with interface entities, such as operations, operation arguments, and operation results. An example as adapted from [Frølund and Koistinen, 1998] is given below. This example is given only as a simple demonstration of the three abstractions used by QML. More complex situations can also be specified in QML.

```
interface SomeService {
      void SomeFunction(in Argument ar1);
};

type Performance = contract {
      delay: decreasing numeric msec;
      throughput: increasing numeric mb/sec;
};

contract1 = Performance contract {
      delay < 40msec;
};

PerformanceProfile for SomeService = profile {
      require contract1;
};
```

Although QML is a general-purpose QoS specification language in the sense that it separates specification of QoS from specification of functional aspects, it has its own limitations. For instance, the *profile* is used in QML to define QoS offers of service providers. But, it is not possible to specify a number of profiles that a service provider may offer depending on the prevailing environmental conditions.

2.1.4.2 CQML and CQML$^+$

CQML uses four main abstraction mechanisms to specify QoS contracts. These are: (i) QoS characteristics, (ii) QoS categories, (iii) QoS statements, and (iv) QoS profiles. *QoS characteristics* are defined as user-defined types. QoS characteristics (e.g. frame rate, response time) are then grouped and restricted into specifications of QoS. For this, *QoS statements* that constrain each constituent QoS characteristic within specific ranges of values are defined. These two constructs focus on specification of QoS independent of what interface it annotates and how the QoS mechanism is implemented. A third construct, *QoS profiles*, relates QoS statements to specific components or parts thereof. Finally, *QoS categories* are used to group any of the three aforementioned concepts. The relationship of these abstractions is depicted in Figure 2.1.

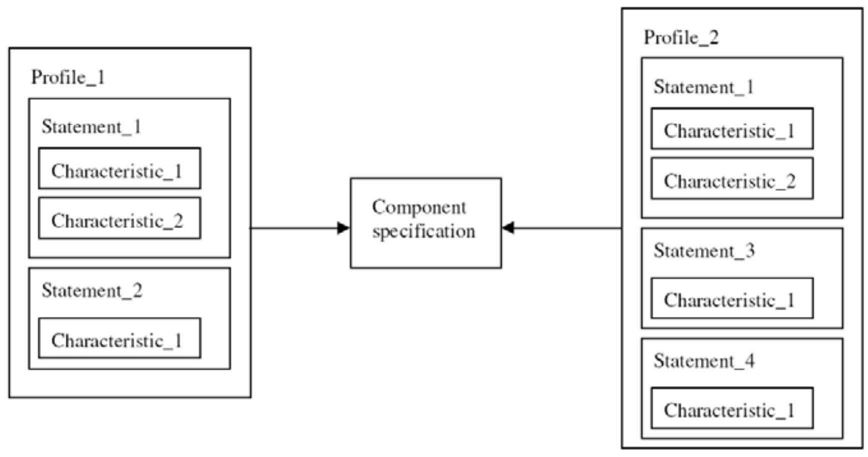


Figure 2.1: CQML overview [Aagedal, 2001]

CQML uses the QoS profile construct to specify the non-functional properties of a component in terms of what a component requires (through a *uses* clause) from other components and what it provides (through a *provides* clause) to other interacting components. The resource demand by the component from the underlying platform isn't captured in the specifications. CQML$^+$, which is an extension of CQML, adds a *resource* clause in the QoS profiles of CQML [Röttger and Zschaler, 2003].

Figure 2.2 shows the specification of non-functional properties of a component that follows Cheesman and Daniel's classification schemes as used by the COMQUAD component model [Göbel et al, 2004a]. CQML$^+$ is used to specify the non-functional properties of the component for the COMQUAD model.

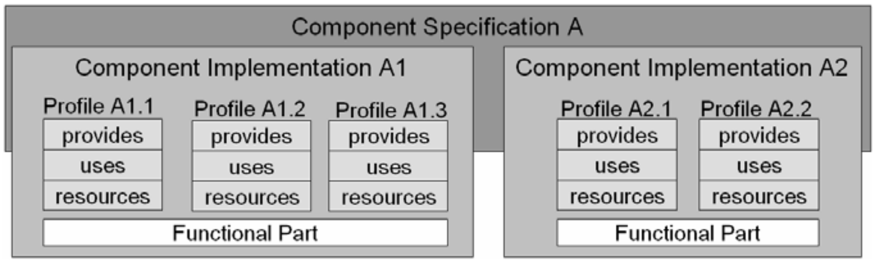


Figure 2.2: Correlation of component specifications, implementations, and NFP profiles [Göbel et al, 2004a]

As shown in Figure 2.2, NFPs are associated with component implementations. There can be multiple implementations of the same functional component specification. Each implementation provides the same functionality, but may have different NFPs. These properties are described using potentially multiple profiles, which represent the operational ranges of the component implementation.

2.1.4.3 UML Profile for Modeling QoS & Fault Tolerance Characteristics and Mechanisms

The OMG's specification document – UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms [OMG, 2005] - defines a set of UML extensions to represent Quality of Service and Fault-Tolerance concepts. In this thesis, we are particularly interested in the section that deals with a QoS framework metamodel that defines the abstract language that supports modelling general QoS concepts.

One of the functional elements of the QoS model used in [OMG, 2005] is the resource-consuming component (RCC). RCC is a processing entity that includes a group of concurrent units of execution, which cooperate in the execution of a certain activity and share common budgets. The budget is an assigned and guaranteed share of certain resources. An RCC has the following associated: i) facets (interfaces provided and synchronously used by RCC clients); ii) receptacles (interfaces synchronously used by this RCC); iii) event sinks (event queues supported by this RCC and asynchronously used by RCC clients); and iv) event sources (event queues asynchronously used by this RCC). UML can model RCC in different ways; in general, classes, components, and interfaces are modeling elements that model the RCCs.

The following have been identified to be important concepts that must be supported in a general QoS modelling language and are incorporated in the QoS Framework metamodel. These are: (i) Definition of QoS Characteristics, (ii) QoS Constraints, (iii) QoS Levels of Execution, and (iv) QoS Adaptation and Monitoring.

QoS Characteristics represent quantifiable characteristics of services such as latency, throughput, capacity, scalability, availability, etc. The QoS Characteristics are specified independently of the elements that they qualify. *QoS Dimensions* are dimensions for the quantification of QoS Characteristics. QoS dimensions also have units (e.g. sec for delay) and direction (an enumeration of increasing, decreasing, and undefined values) that defines the type of order relation (e.g. delay is decreasing while throughput is increasing). When the number of QoS Characteristics is large, and they are especially complex, some mechanisms for grouping are required. Such a grouping is referred to as *QoS Category*. An example of general groupings of quality attributes is performance, which includes quality characteristics such as latency, throughput, etc.

A *QoS Constraint* limits the allowed values of one or more QoS Characteristics. Application requirements or architectural decisions limit the allowed values of quality and the QoS Constraints describe these limitations. The quality constraints are viewed from two points of view: from the *client's point of view* and from the *provider's point of view*. This approach defines two different types of constraints: *constraints required* and *constraints offered*. This is a common approach in the specification of QoS [Aagedal, 2001][Miguel et al, 2002].

Required QoS constraint by a client specifies the non-functional (quality) requirement of a client that the service provider has to support. This constraint limits the space of valid values for the QoS characteristics involved in the service. The QoS characteristics are the dimensions of the quality space, and the Required QoS defines the valid values of this space. Required QoS constraint by a server can be defined either to the client that requests a service from the server or to another server onto which the first server relies in order to provide its service. In the former, the constraint specifies quality requirements that must be achieved by the clients so that the provider can offer the service at a quality level as required by the client. One example is the maximum frequency of invocation that clients must achieve. In the latter, the specification applies to a software element that provides service and uses the service of another provider.

Offered QoS associates the set of QoS characteristics that a software component guarantees for the services it provides. The offered QoS often depends on the QoS provided by the resources and service providers that the software element uses. Offered QoS constraint defined by a provider specifies constraints that the provider must achieve while offered QoS constraint by a client represents the constraint that the client must achieve to the invoking service.

For collaboration among interacting components, a service provider specifies the quality values it supports (provider-Offered QoS) and the requirements that its clients must achieve (provider-Required QoS). A client specifies the quality it requires (client-Required QoS), and the quality that it ensures (client-Offered QoS). In general, the allowed values in client-Required QoS must be a subset of values in provider-Offered QoS, and the allowed values in provider-Required QoS must be a subset of client-Offered QoS. In this case, a QoS Contract will be established between the client and service provider, where the selected values in provider-Offered QoS, provider-Required QoS, client-Required QoS, and client-Offered QoS constitute the QoS contract. While in general it is required to have conformance between a client's and server's QoS offers and requirements, the bi-directional relationship exists only in certain cases as the subsequent sections illustrate. In some cases, the QoS contract cannot be computed statically as it may depend on the resources available or quality attributes fixed dynamically.

2.2 QoS-Enabled Object and Component Middleware

2.2.1 The QuA Project

QuA (Quality of service-aware component architecture) [Staehli et al, 2004; Amundsen et al, 2005] is an open, reflective component architecture that provides an execution environment for components and also offers hooks to which QoS-management components can be attached in order to meet committed QoS-levels during run-time.

QuA has been designed as a minimalist research component architecture with support for QoS-sensitive applications. The QuA architecture is a distributed platform consisting of a set of networked capsules. Each capsule provides a run-time environment for local core QuA objects and platform managed objects. The minimal QuA core enables it to be deployed on a wide range of computers.

An essential part of the QuA's architecture is the *Service Planner*. Together with the resource manager, the *service planner* creates the notion of *platform-managed QoS*. The service planner is performing what is referred to as *QoS-driven service planning*: A process that identifies components, component configurations, and how to compose them; to form a functionally correct service composition, which meets a set of QoS-requirements, such as accuracy, security and real-time [Amundsen et al, 2004].

In this approach applications are viewed as a service that is constructed from sub-services, which again can be constructed from other sub-services. If a service or a sub-service cannot be broken up into services of finer granularity, it is considered an atomic service. Each service has a service type, which defines the service name and the provided functional services in the format of operation signatures and semantic description. There can be one or more alternative implementations for a service type, where the implementations have the same functional properties but different QoS characteristics [Amundsen et al, 2005; Staehli et al, 2004]

The service plan contains eight information elements as shown in Figure 2.3: 1) *dependencies:* requirements to the execution environment, libraries, and static dependencies to front-end or back-end systems; 2) *parameter configuration:* parameters the component composition or component is to be configured with; 3) *composition specification:* a graph specifying the construction of the service, i.e., the composition of service types and the bindings between them; 4) *role:* a role name space and role names for service types and component types in the composition. The same role name in two alternative service plans will be interpreted during reconfiguration as identical services and, hence, not be replaced during dynamic reconfiguration; 5) *offered services:* services/operations that the composition/component offers; 6) *input QoS contract:* QoS values along QoS dimensions that users of this composition/component must adhere to; 7) *QoS model:* a model of the QoS characteristics, which is defined using QoS dimensions independently of the execution environment. The model specifies the possible range of QoS values along the QoS dimensions; and 8) *QoS mapping:* functions that establish the logical relationships between QoS characteristics at different levels.

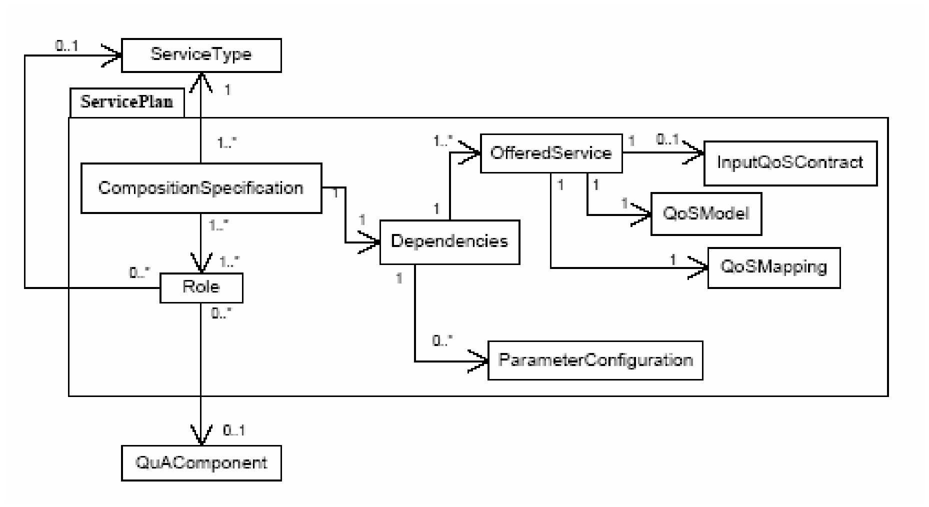


Figure 2.3: Service Plan in QuA's component architecture [Amundsen et al, 2005]

How to choose the correct service composition?

Based on the system resource availability, the service planner [more specifically through its error prediction function] predicts error along each error-dimension in the error model for the available service compositions, and chooses the service composition that best meets the user QoS-requirements (or more formally chooses the plan with the best utility).

The Error Prediction Functions encode the application developer's knowledge about the service, and predict the error level for the service as a function of error in the sub-services of the service composition and system resource availability (CPU, disk, network, etc). Application developers are free to decide their own format and the complexity of the error prediction functions. Examples of error function implementations might be a straightforward condition statement or a large table with measured errors.

Analysis

The QuA approach has many similarities to our approach. One such similarity is that in the QuA architecture the middleware platform is responsible for the QoS management mechanisms. We use the component containers to manage QoS. The container provides a run-time environment for QoS contract negotiation. Both approaches facilitate the separation of concern between application logic and QoS mechanisms (infrastructure).

In the QuA's approach, service plans specify service composition and parameter configuration of the implementation. The QoS characteristics of the implementation are also described in the service plan. The Service Planner selects the correct service composition, i.e., the service plan that provides best utility for the user, based on the actual context and resource situation. The selected complete service plan is then configured and instantiated. Our approach's central idea is the selection of component implementations and QoS-profiles based on the availability of resources and user's QS requirement. The aim of this selection process is more or less equivalent to the service composition and configuration phase in the QuA's approach. Neverthless, unlike QuA, our approach considers component QoS contracts explicitly and aim to consider the complexity of the selection process (through the use of heuristics).

In applications like video-streaming that involves components like *VideoServer*, *VideoPlayer*, etc., although only few components are involved, the number of possible configurations (or in our case component implementations and QoS-profiles) could explode exponentially. As an example to this, an application that has four components, where two implementations are available for each component and two QoS-profiles exist for each implementation, already results in $4^4 = 256$ configurations. This configuration or selection problem could be difficult and could even be intractable. We believe, the complexity of the problem should be properly considered in order to provide a viable solution.

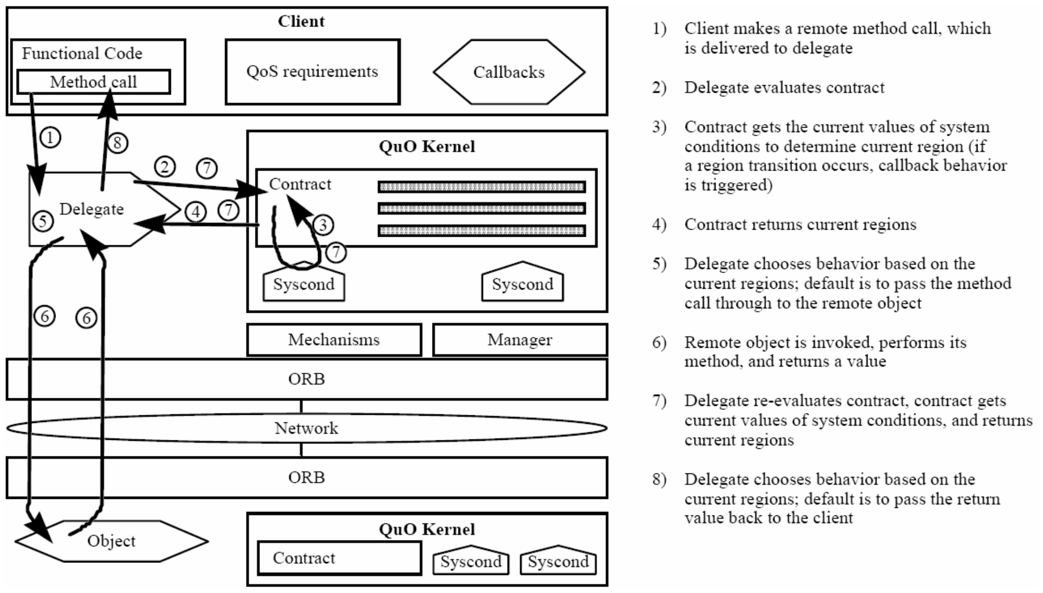
Our approach takes into consideration the interoperability or conformance of interacting components in the selection of implementations and QoS-profiles. This is especially non-trivial for distributed components. Such issues haven't been considered in QuA. The QuA service planner selects the best service composition. The service composition may consist of multiple components or even a single component. Services, sub-services, or an atomic service (component) are selected based on the predicted error.

2.2.2 Quality Objects (QuO) Project

The QuO project was motivated by the distributed object middleware's (e.g. CORBA) lack of support in handling QoS requirements and building systems that can adapt to different levels of QoS. To address this challenge, the BBN technologies developed the QuO framework [Zinky et al, 1997; Loyall et al, 1998; Schantz et al, 2003], which is a QoS adaptive layer of middleware that runs on existing DOC middleware (such as Real-time CORBA and Java RMI) and supports distributed applications that can specify (i) their QoS requirements, (ii) the system elements that must be monitored and controlled to measure and provide QoS, and (iii) the behavior for adapting to QoS variations that occur at run-time.

The QuO framework consists of the following components [Loyall et al, 1998]: (i) a suite of Quality Description Languages (QDL) for describing *contracts*, *system condition objects*, and the adaptive behavior of objects and *delegates*, (ii) the QuO kernel, which coordinates evaluation of contracts and monitoring of system condition objects, and (iii) code generators which weave together QDL descriptions, the QuO kernel code, and client code to produce a single application program.

Figure 2.4 illustrates the steps that can occur during a remote method call in a Quo application (this is an application that adds the QuO framework on top of a traditional CORBA application).



1) Client makes a remote method call, which is delivered to delegate
2) Delegate evaluates contract
3) Contract gets the current values of system conditions to determine current region (if a region transition occurs, callback behavior is triggered)
4) Contract returns current regions
5) Delegate chooses behavior based on the current regions; default is to pass the method call through to the remote object
6) Remote object is invoked, performs its method, and returns a value
7) Delegate re-evaluates contract, contract gets current values of system conditions, and returns current regions
8) Delegate chooses behavior based on the current regions; default is to pass the return value back to the client

Figure 2.4: Example remote method call in a QuO application [Loyall et al, 1998]

Adaptation in a QuO application occurs through the evaluation of contracts, which is triggered by a method call/return or change in a system condition. The contract evaluation may result in a transition from one active region into another and this in turn triggers transition behavior, which consists of client callbacks or method calls on system condition objects. All these sequences of actions are captured and specified in a QuO contract and other mechanisms by a QoS developer. The QoS developers, also refereed as *QoSketeers*, develop QuO contracts, system condition objects, callback mechanisms, and object delegate behavior.

A QuO QoS contract between a client and object in an application consists of the following components:

- A set of nested operating *regions*, each representing a possible state of QoS. The nesting could be into two sets of nested regions, an outer nesting called *negotiated regions* and an inner nesting called *reality regions*. The negotiated regions represent the QoS desired by the client and the QoS that the remote object expects to provide and their predicates consist of system condition objects that interface to the client and object. The reality regions represent the QoS measured in the system and have predicates consisting of system condition objects that interface to system resources. This grouping distinguishes the QoS associated with operating modes of the client and object, which will likely change infrequently, from the measured QoS of the system, which will probably change more frequently
- *Transitions* for each level of regions, specifying behavior to trigger when the active region changes.
- References to *system condition objects* for measuring and controlling QoS. These are either passed in as parameters to the contract or declared local to the contract.

System condition objects are used in the predicates of regions to get values of system resources, object or client state, etc. and used in transitions to access QoS controls and mechanisms.

- *Callbacks* for notifying the client or object. Callbacks are passed in as parameters to the contract and are used in transitions.

QuO represents an example of Aspect-Oriented Programming (AOP), in which a program is divided into aspects of concern, each of which is programmed separately in a language suitable for expressing the particular aspect. The application is constructed by weaving (using code generators) the aspects together into a single, executable application. QuO allows an application developer to separate the aspects of functional behavior, QoS contracts, system state monitoring and control, and alternate implementation and adaptation which would traditionally be interleaved throughout a critical application.

Analysis

The QuO framework offers advanced concepts and necessary tools in order to integrate QoS management into distributed applications based on CORBA. QuO's aspect programming concepts and our container-based approach targets the separation of "business code" from "QoS management code". Nonetheless, there are certain issues that make our approach distinct from that of QuO's. The first is that the ideas of QuO are built on top of (or augmenting) existing object middleware (i.e. CORBA) rather than building the concepts based on CBSE. It has been noted in [Bouyssounouse and Sifakis (eds.), 2005] that in order to be useful in component based systems, contract languages must include facilities for expressing properties typical of components, that is, their context dependencies.

QuO's QoS contracts describe the adaptive behavior of a distributed application. This is made possible by describing all feasible QoS states of the interaction between the client and the remote object and also every transition between these states. The QoS developer is responsible for specifying all the necessary elements in the QoS contract. This is a burden on the QoS developer. Instead of specifying the predetermined adaptive behavior in the QoS contract, why don't we leave the reasoning on adaptation (or negotiation) to the component containers, which they could do based on the specifications of the interacting components' *provided-* and *required-QoS contracts*? One advantage of the container-based approach is that it makes the application development process easier.

The QuO's approach presupposes a tight coupling in the client-object interaction possibly limiting the use of the specified QoS contract only in the anticipated environment and for the QoS desired by the client and the QoS that the remote object expects to provide. But, dynamic applications, i.e. applications created at run-time based on what each constituent component offers and expects requires a loosely-coupled approach. The loose coupling makes it easier to incorporate new client's requirements and also new component implementation's, which may provide new performance advantages (e.g. by implementing a different algorithm).

2.2.3 CIAO (Component Integrated ACE ORB)

CIAO [Wang et al, 2003] is a QoS-enabled implementation of the CORBA component model (CCM) built on top of the ACE ORB (TAO). TAO is an open-source, high performance, highly configurable Real-time CORBA ORB that implements key patterns to meet the demanding QoS requirements of DRE systems. CIAO is developed at Washington University in St. Louis and the Institute for Software Integrated Systems (ISIS) at Vanderbilt University, USA.

CIAO is motivated by the limitations of conventional component middleware (e.g. CCM) for large-scale DRE systems. Examples of DRE applications include distributed sensor networks, flight avionics systems, naval combat management systems, and financial trading systems, all of which typically have stringent QoS requirements. DRE applications often require prioritization of various tasks to ensure critical tasks are handled within their time constraints. Developers are forced to embed QoS provisioning mechanisms imperatively in component implementations when using the conventional component middleware for DRE systems. This creates dependencies between application components and the underlying component framework making the reuse of component implementations very difficult.

To ensure that DRE applications can achieve their QoS requirements, various types of QoS provisioning must be performed to allocate and manage system computing and communication resources end-to-end. QoS provisioning can be performed statically or dynamically. *Static QoS provisioning* involves pre-determining the resources needed to satisfy certain QoS requirements and allocating the resources of a DRE system before or during start-up time. *Dynamic QoS provisioning* involves the allocation and management of resources at run-time to satisfy application QoS requirements [Wang et al, 2003].

CIAO extends CCM to support *static QoS provisioning* as listed below:

- By extending the CCM component assembly descriptor with features that include *QoS provisioning specification* and implementation for required QoS supporting mechanisms.
- By supporting for client configuration specifications to facilitate the configuration of a client ORB to support various QoS provisioning attributes, such as priority level policy. Clients can then be associated with named QoS provisioning policies.
- By enhancing CCM containers to support QoS capabilities, such as various server specific Real-time CORBA policies.
- By supporting installation of meta-programming hooks, such as portable interceptors. Such a capability can be used to inject dynamic QoS provisioning transparently.

The aforementioned features decouple QoS management from component implementations (which are concerned with the business code) and help compose static QoS provisioning capabilities into the application via the component assembly and deployment phases.

Analysis

CIAO's philosophy is a strong adherence to existing OMG specifications such as RT-CORBA and CCM, and the extension of those. Hence, CIAO is tied to RT-CORBA. CIAO facilitates the development of componentized solution to DRE applications, which are hard real-time applications with stringent QoS requirements, by enabling developers to declaratively provision QoS policies end to end when assembling the system. Though CIAO'S main target is the support of static QoS provisioning, dynamic QoS provisioning is also possible by using portable interceptors and other meta-programming hooks.

When we compare DRE systems with other applications such as video-on-demand, which is a soft-real time application, we distinguish certain differences in the QoS requirements of these applications. For example, video-on-demand applications need the integration of non-functional requirements to enable the provision of QoS differentiation in the offered services (i.e. to different users). On the other hand, DRE systems have stringent individual QoS requirements (e.g. bounded response time), which must be satisfied simultaneously. Failure to meet these QoS requirements, e.g. a missed deadline, often leads to serious consequences even if an application may still logically function properly. These differences in QoS requirements make one solution more specific to a certain area of application.

2.2.4 QoS-Aware Middleware [*2k^Q* and Agilos Projects]

Nahrstedt et al have proposed a QoS-aware middleware for ubiquitous and hetrogenous environments [Nahrstedt et al, 2001]. They have also presented an integrated run-time framework for distributed multimedia applications [Baochun et al, 2002; Dongyan et al, 2000a ; Dongyan et al, 2000b].

In [Nahrstedt et al, 2001], they present four key aspects of their QoS-aware middleware system. These are: (i) QoS specification to allow description of application behavior and QoS parameters; (ii) QoS translation and compilation to translate specified application behavior into candidate application configurations for different resource conditions; (iii) QoS setup to appropriately select and instantiate a particular configuration; and finally, (iv) QoS adaptation to adapt to run-time resource fluctuations.

We are interested in the last two aspects, namely, QoS setup and QoS adaptation phases as these are the ones closely related to our QoS contract negotiation. At the stage of run-time instantiation (QoS setup), the middleware selects a suitable configuration for the application which matches the specific resource availability and user preference. They use the terms *application/service configuration/selection/re-selection* in the QoS setup to mean more or less the same thing. The run-time instantiation phase has certain similarities to our QoS contract negotiation. In our case also, the negotiation is driven by the user's QoS requirement and current end-to-end resource condition. In order to make the distinction between our approach and the ones followed Nahrstedt et al clear, let's give some more details.

The QoS-aware middleware architecture proposed in [Nahrstedt et al, 2001] [Baochun et al, 2002] favors applications modeled by a generic *application component model*. In this model a collection of interconnected application components on a single host are viewed as a set of tasks, with input-output dependencies. Beyond a single end host, the entire distributed application can be grouped into clients and services. The collection of clients and services form another directed graph representing the service provider-consumer relations. The graph in Figure 2.5 is called an application functional graph [Baochun et al, 2002].

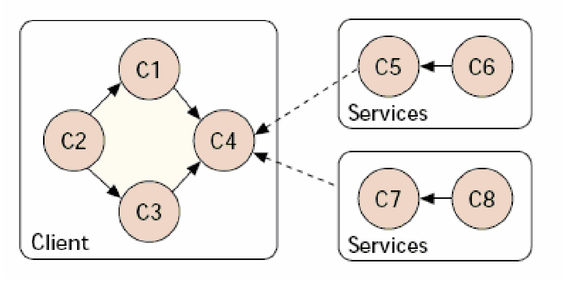


Figure 2.5: An Application Functional Graph as used in the QoS-aware middleware architecture ($2K^Q$ and Agilos Projects)

In ubiquitous and heterogeneous environments, it is desirable for an application to have multiple configurations, each suited to a different QoS requirement, resource condition, or physical environment of users. Based on this idea, multiple application configurations can be defined based on the application functional graph (Figure 2.5). For example, the following two configurations may be defined based on Figure 2.5.

- $Confg_1$ involves the following application components: C2, C1, C4, C5, C6
- $Confg_2$ involves the following application components: C2, C3, C4, C7, C8

$Confg_1$ may be suitable for a high performance client that has sufficient CPU and bandwidth while $Confg_2$ may be suitable for a client with weak CPU and low bandwidth capability. The selection among the different applications configurations ($Confg_1$, $Confg_2$, etc.) is governed by a rule base as exemplified in [Baochun et al, 2002]. For a similar problem, a different configuration strategy has been proposed in [Dongyan et al, 2000b]. Assume each configuration is associated with an end-to-end resource vector. Hence, R_1 and R_2 are the end-to-end resource demand vectors of $Confg_1$ and $Confg_2$ respectively. The proper service configuration $Confg_x$ is chosen such that $R_x < R_{avail}$ and there is no other configuration $Confg_y$, which has better quality and $R_y < R_{avail}$. $Confg_x$ or $Confg_y$ are one of ($Confg_1$, $Confg_2$, etc.).

The basic difference between the approach by Nahrstedt et al as described above and our approach can be explained as follows. The multiple service configurations, which are derived from the Application Functional Graph (Figure 2.5), are defined for the application as a whole and they involve a different set of service components. Moreover, the application is composed prior to run-time. One of the drawbacks of the design-time composition is that once the selections have been made it is difficult to accommodate unexpected run-time changes, which are common in many distributed applications. In our case, we specifically consider the NFPs (QoS contracts) of individual components, which are specified in multiple QoS-Profiles. We would then compose the application dynamically by selecting concrete QoS contracts of each component at run-time based on resource availability, other interacting components' QoS offers and expectations, and user's QoS requirements and preferences. Our approach, which is based on the principles of CBSE, is more general and can be applied to a wider range of application domains.

To explain the flexibility of our approach, let's consider the case of component changes and/or upgrades. How would, for instance, the QoS middleware in [Nahrstedt et al, 2001] address the change/upgrade of some components or how does it cope when new components are available which are to be used in the application? This might necessitate the change of the rule base as explained in [Baochun et al, 2002] (assuming the rule base itself is configurable) to form a different application configuration. In our approach, what is required is only the NFPs specification of the new or upgraded components. Nothing will be affected in our middleware or any of its configurations as the application compositions are performed dynamically according to the QoS contract specifications of the components.

A dynamic QoS-Aware Multimedia Service Configuration is proposed for ubiquitous computing environments in [Xiaohui et al, 2002]. According to [Xiaohui et al, 2002], a model that includes two tiers has been proposed. These are: (i) service composition, and (ii) service distribution tiers. The former is responsible for choosing a set of suitable services, discovered in the current environment, to compose a customized application delivery to any client device. This tier also provides a QoS consistency check to discover and correct inconsistencies of QoS parameters between any two interacting service components. The service distribution tier is responsible for dividing a distributed application into several partitions and dispatching them to different devices according to the current distributed resource availability.

The fact that compositions of services are made dynamically make the proposal in [Xiaohui et al, 2002] similar to our approach. However, in our case, we assume the components are already deployed into their respective nodes and try to find appropriate QoS contracts. In [Xiaohui et al, 2002], the service composition is performed without the knowledge where the components will be deployed. But, what happens when the available resources in the nodes cannot provide the required QoS to the user has not been addressed. Moreover, their approach is specific to multimedia applications only.

2.2.5 The OpenORB Project

OpenORB [Blair et al, 2001] is a middleware platform developed by researchers at Lancaster University. The OpenORB project was motivated by the lack of flexibility of established middleware platforms such as CORBA and DCOM to meet the needs of emerging distributed applications (e.g. safety-critical, real time, etc.), which require capability of both deployment-time *configurability* and run-time *reconfigurability*.

The design of OpenORB uses *component technology* thereby enabling the middleware itself to be built using components in addition to enabling the development of component-based applications (similar to the mainstream component models like EJB). Its design is also highly *reflective* in the sense that the component configurations that comprise both the middleware and the applications are associated with *causally connected* data structures (called meta-structures) that represent aspects of the component configurations, and offer meta-interfaces through which these represented aspects can be inspected, adapted and extended. The reflective middleware architecture of OpenORB adopts the following four principles: (i) a component-based model of computation, (ii) a procedural (which is a more general approach than declarative) approach to reflection, (iii) support *per interface* (or, sometimes, *per component*) meta-spaces, and (iv) structure meta-space as a number of closely related but distinct meta-space models

In reflective systems, one can classify two types of reflections: *structural* and *behavioural*. *Structural reflection* is concerned with the content of a given component while *behavioural reflection* is concerned with activity in the underlying system. These and other aspects of reflection are represented and structured in the OpenORB architecture into four distinct meta-space models [Blair et al, 2001]: (i) Interface Meta Model, (ii) Architecture Meta Model, (iii) Interception Meta Model, and (iv) Resource Meta-Model.

The structural reflection is represented by two distinct meta-models, namely the *interface* and *architecture* meta-models (these have also been termed as *encapsulation* and *composition* meta-models in [Blair et al, 2000]).The interface meta-model provides access to the external representation of a component, i.e. in terms of the set of provided and required interfaces, their methods and associated attributes. The architecture meta-model provides access to the implementation of the component as a software architecture, consisting of two key elements: a *component graph* and an associated set of architectural constraints. The component graph represents the composition of components, which itself is a component. The architecture meta-model can be used to both discover and also make changes to this software structure at run-time.

Concerning behavioral reflection, OpenORB distinguishes between actions taking place in the system, and the resources required to support such activity. These two aspects are represented by the *interception* and *resource* meta-models respectively. The interception meta-model enables the dynamic insertion of interceptors. Such interceptors are associated with interfaces (more specifically, local bindings) and enable the insertion of pre- and post- behavior. Interceptors can also be used to introduce non-functional behavior, such as security checks or concurrency control. The resource meta-model offers access to underlying resources and resource management. The resources meta-model is based around the abstractions of resources and tasks.

In combination with the reflective component model, OpenORB uses a *component framework (CF)* based structuring principle as a main ingredient to the integrity maintenance across reconfiguration operations [Coulson et al, 2002]. Component frameworks are collections of rules and interfaces that govern the interaction of a set of components plugged into them. The use of a hierarchically-structured CFs as the natural scope for adaptation addresses the issue concerned with constraining the scope and effect of reconfiguration, which is one of the fundamental issues in effective reconfiguration management. Meta-level manager/managed pattern can be applied within the resultant hierarchical scopes to maintain integrity in the face of dynamic change (e.g. when a new component is dynamically inserted). According to [Coulson et al, 2002], CF-based adaptation is expected to serve most adaptation needs. Nevertheless, it is also possible in OpenORB to bypass the CF structure and perform ad-hoc unanticipated adaptations directly at the OpenCOM level.

The issues of QoS management with regard to the OpenORB's architecture has been addressed in [Blair et al, 2000] where the paper examines how dynamic QoS management functions are supported in a reflective middleware. According to [Blair et al, 2000], dynamic QoS management can be achieved by introducing *management components* into the component graph structure (accessed via meta-space). Communication between *management* and *managed* components is achieved by an event notification mechanism. The prototype implementation of OpenORB, which has been developed in Python, supports the creation of such management components.

The different styles of management components identified in a dynamic QoS management are:

i.) *Monitoring* – collect statistics on the level of QoS attained by the running system and raise events when problems occur.

ii.) *Control* – divided into *strategy selectors* that select an appropriate adaptation strategy based on feedback from the Monitoring component, and *strategy activators* that implement a particular strategy, e.g. by manipulating a component graph.

The dynamic QoS management as discussed above differs from the traditional QoS management approach as it uses the reflective architecture. Firstly, new management components can be dynamically introduced into the underlying configuration and removed when no longer needed. Secondly, the policy for management is itself open to inspection and adaptation through representation of management components. The OpenORB approach also provides support for the specification of QoS management policies in the formal language of *timed automata*.

Analysis

OpenORB combines the concepts of reflection and component technology to the design and implementation of a generic component-based middleware platform that addresses many of the deficiencies of today's mature component models (e.g EJB or .NET). For instance, the issues in QoS management (e.g. monitoring and adaptation) can be handled in OpenORB by introducing CFs as building blocks together with the concept of reflection. It has also been demonstrated that dynamic QoS management can be realized through management components (which are accessed through the appropriate meta-models). Our work is more focused on how to support non-functional properties of components and component-based applications with non-functional requirements such as QoS through QoS contract negotiation (that includes the issues of negotiation protocol and decision making). Our prototype applies the interceptor approach (the interceptor meta-model is one meta-model developed in OpenORB) in order to realize the QoS contract negotiation.

2.2.6 A QoS Metamodel and its Realization in a CORBA Component Infrastructure

[Ritter et al, 2003] describes a QoS meta-model and its realization in a CORBA component infrastructure. There are two points where the QoS support has to be plugged into the existing CCM architecture. The first is the binding mechanism between two components and the second is the interaction between container and component implementation (business code).

The Binding takes place when two components are explicitly connected via their ports. Ports are externally visible interaction elements. To seamlessly integrate QoS support into the CCM infrastructure this binding procedure has to be extended by a negotiation phase. But this phase should only be started if the components want to negotiate a certain QoS contract. The container of the client component decides on this. The container knows whether the hosted component wants to negotiate a certain QoS contract or not.

Thus, a third party will coordinate the initial binding between two components. This means the business code of each involved component will not directly influence the QoS negotiation. This is done to decouple QoS negotiation from business code.

A QoS-aware component offers the interface `Negotiation`. Client and server components before their connection set-up use this interface to make negotiation. The container of each QoS-aware component provides the implementation of the `Negotiation` interface.

```
interface Negotiation {
      ContractBaseSeq req_contract(in ContractBase min );
      ContractId accept(in ContractBase accept,
            inComponents::CCMObject own_ref)
            raises(AcceptionFailed);
      void terminate_contract(in ContractId contract_id);
      string get_facet_name(in Object facet_ref) raises(NoFacet);
};
```

Analysis

A connection set-up (for two components) is preceded by QoS contract negotiation. Although a third party coordinates the initial binding, it is the (QoS-aware) components which are responsible for the negotiation process. They have to invoke appropriate methods from the Negotiation interface. This implies that QoS-management codes are tied to the component implementations.

In our approach negotiation precedes every service invocation of the application. The service invocation may involve more than two components and the binding is made based on some global user's QoS requirement. If this global user's QoS requirement can be broken down into a requirement on individual component's requirement, then this amounts to the negotiation between two components.

The work only shows how a QoS Provider (which is similar to our integration of contract negotiation into the component framework and with some other differences mentioned previously) fits into a QoS-aware Component Middleware and how actually (or even the strategy) the contract is managed, monitored, and enforced is absent.

2.2.7 Worth-Based Multi-Category QoS Negotiation in Distributed Object Infrastructures

In [Koistinen and Seetharaman, 1998], QoS negotiation is viewed as one key element to dynamic configuration and adaptation. During negotiation two or more distributed agents try to reach an agreement on the QoS that they will attempt to provide to each other. The negotiation is performed dynamically as the system executes. The proposed negotiation mechanisms have the following characteristics:

- The negotiation is concerned with multi-category negotiation instead of limiting the negotiation to specific domains such as multimedia. Category generally refers to QoS aspects such as performance, reliability, or security. Categories typically consist of multiple dimensions.
- *Worth* (sometimes called utility) calculation is applied in the negotiation in addition to satisfying conventional constraints.
- The focus of the negotiation is on QoS for operation invocation rather than for data streams

For the client and server applications, which are referred as agents, a *constraint profile* or simply *profile* is assumed to be specified using QML [Frølund and Koistinen, 1998]. More specifically the term *server profile* and *client profile* are used for the QoS characterization of a server and a client respectively. Both the client and the server have their own representations for both profiles. The client-side server profile describes the client's requirements of the server while the client profile describes its own characteristics. The server-side server profile describes the server's QoS guarantees and its client profile captures the server's requirements for the client.

A typical negotiation scenario is given as below. It is assumed that before a negotiation can start, the client must have obtained a reference to the server with which it wishes to negotiate.

1. The client requests the set of offers that the server supports. It supplies the *client profile* to the server so that the server may present suitable offers.

2. If more than one offer satisfies the *server profile* of the client, the client selects the offer with the highest worth and proffers that one to the server.
3. In this case, let us assume that the server cannot accept the chosen offer. It then computes a counter offer that it sends to the client.
4. If the counter offer was acceptable to the client, it sends a *deal* message back to the server.
5. The server acknowledges the deal.

Analysis

The scenarios considered in [Koistinen and Seetharaman, 1998] are assumed only for a single client – single server negotiations. Multiple-clients negotiation has been left out for future consideration. Moreover, the approach doesn’t take into account performance implications of a negotiation. We have addressed these two issues in this thesis.

The approach concentrates more on the protocol aspects of negotiation, i.e. the exchange of messages and contents of the messages. As far as evaluation of offers is concerned, the way to deal with is that when the client receives several QoS offers for a service, it either selects one of these offers to propose a deal or it proposes a conflict deal. As part of the selection process, the client agent first applies conformance checking to obtain the set of server offers that meet its absolute requirements. The approach says little on how, for example, the server makes its (possibly multiple) proposal or counter-proposals in the first place.

The negotiation mechanisms proposed are based on a distributed object infrastructure (e.g. CORBA or DCOM) and the QoS is specified with QML though it is claimed that the model is platform or language independent. Nevertheless, the concepts are not directly applicable in the context of CBSE. It has been noted in [Bouyssounouse and Sifakis (eds.), 2005] that in order to be useful in component based systems, contract languages must include facilities for expressing properties typical of components, that is, their context dependencies.

2.2.8 Others

The work in [de Miguel et al, 2002] proposes a QoS-aware component framework that extends the EJB container by integrating QoS services such as resource reservation and negotiation. The EJB container implements basic negotiation algorithms and isolates the business components from reservation services. The approach allows clients to negotiate a single QoS dimension of method calls per second. But it does not explain how component contracts can be negotiated in the multiple nodes.

In [Menasce et al, 2004] a model is described where a component provides a set of interrelated services to other components. These components are QoS-aware and are capable of engaging in QoS negotiations with other components of a distributed application. The paper attempts to create a framework for software components that are capable of negotiating QoS goals in a dynamic fashion using analytic performance models. The QoS negotiation between two components occurs by taking performance as a QoS requirement and concurrency level as a means of negotiation element. Our treatment of QoS negotiation is more generic and general, which may be applied for a larger set of problems. Moreover, the container handles the negotiation between components in our case, which enhances the reusability of the components.

Aspects that we believe are important in a framework of the proposed type [Menasce et al, 2004] but not addressed are:

i.) The negotiation between the QoS-aware component and the requester (a client component) can result in requests being accepted, rejected, or in counter offers being made to the requester. It is not clear how the client component makes its own decisions with regard to accepting the counter offer proposed by the server component. While a decision mechanism (using a performance model solver) has been provided for the server component, no mechanism has been proposed for the client component.

ii.) The multiple-clients negotiation is performed one client at a time. No strategies have been formulated for cases in which the negotiation is to be done with multiple clients in a group. This is essential in order to maximize the total utility of the system. The decision making process should not only depend on whether a client's requests can be fulfilled or not or whether already existing clients' commitment will be broken or not but also on meeting the goals of the negotiating parties (for e.g., maximizing utility)

2.3 Web Services

2.3.1 QoS-Aware Middleware for Web Services Composition

[Zeng et al, 2004] presents a QoS-aware middleware platform that addresses the selection of Web services for the purpose of their composition in a way that maximizes user satisfaction expressed as utility functions over QoS attributes, while satisfying the constraints set by the user and by the structure of the composite service. Two QoS-driven selection approaches are described and compared: *local optimization* approach and *global planning* approach.

The local optimization approach performs optimal service selection for each individual task in a composite service without considering QoS constraints spanning multiple tasks and without necessarily leading to optimal overall QoS. The global planning approach on the other hand considers QoS constraints and preferences assigned to a composite service as a whole rather than to individual tasks, and uses integer programming to compute optimal plans for composite service executions.

The service composition and the selection are made based on the service description of the web service, which contains metadata that describe, among others, the capabilities and QoS of a web service. In the scenario discussed in the paper, the QoS dimensions considered are: execution price, execution duration, reputation, successful execution rate, and availability.

In the local optimization, selection is made on a particular web service that executes a given task of a composite service. The selection is made as follows: the system collects information about the QoS of each of the web services. After collecting this information, a quality vector is computed for each of the candidate web services, and based on the quality vectors; the system selects one of the candidate web services by applying Multiple Criteria Decision Making (MCDM). This selection process is made based on:

i) The weight assigned by the user to each criterion.
ii) A set of user-defined constraints (i.e. user's requirement in each of the QoS dimensions of the web-service), which may be specified for a task. In our approach, we don't assume that user's constraints are given on individual components but rather on the whole or composed application. Zeng et al elaborate this assumption as follows: examples of constraints that can be expressed on a given service include duration constraints and price constraints. However, constraints can only be expressed on individual tasks and not on combinations of tasks. In other words, it is not possible to express the fact that the sum of the durations for two or more tasks should not exceed a given threshold.

In the global planning approach, all possible plans associated to a given execution path are generated (at least conceptually speaking) and the one which maximizes the user's preferences while satisfying the imposed constraints is then selected.

Comparison of local optimization and global planning

The computational cost of local optimization is polynomial. The bandwidth cost is very limited: for each task, there are two messages that flow between the composite service execution engine and the service broker (query and result) and three messages that flow between the execution engine and the selected Web services (enable, start, and completed). On the negative side, the local optimization approach has two shortcomings: It cannot consider global trade offs between quality dimensions; and it cannot consider global constraints,

The global planning approach overcomes these shortcomings, but at the price of higher computational and bandwidth cost. Indeed, the global planner first needs to select an optimal execution plan using an expensive algorithm (exponential in some cases). It then needs to monitor all the candidate Web services (whether they are included in the plan or not) thereby consuming considerable bandwidth resources. Finally, when it detects exceptions or changes, it may need to revise the execution plan, again using an expensive algorithm. Another issue with global planning is that users are required to provide relatively complex input (i.e., global constraints and trade offs).

Analysis

The concept of service selection in [Zeng et al, 2004] is similar to our profile selection. There is a difference between our component's Quality Model and Web Service Quality Model used in the paper. A Web Service's quality is described through a quality vector attached to its operations. Our component's quality is described not only by the quality that is provided by its operations but also by the quality it requires from the environment. Thus, in our approach, we explicitly use the information about the required QoS of components – we call these QoS Contracts. This modeling enables components for maximum reusability, and independent deployment and third party composition.

The local optimization is not concerned with conformance relation – this emanates from explicit specification of context dependencies. It doesn't consider resources of the client, server, or the network (this kind of information might be incorporated in the price). What makes the global planning different from the local optimization is that it aims at optimizing the overall QoS. In other respects it is the same. The comments given concerning local optimization also hold for global planning.

3 QoS Contract Negotiation in Multiple Component Containers

3.1 Introduction

The QoS contract negotiation that we want to automate occurs between components in distributed component-based software. The theme of negotiation is present in various fields and, as a result, several definitions have been proposed for it in the literature. To give a wider perspective on what an automated negotiation may involve, we refer to three different descriptions. Whereas the first description is a more general one, the last two are more specific to a particular environment. In [Rosenschein and Zlotkin, 1994] automated negotiation is defined as: the process of several agents searching for an agreement. Agreement can be about price, about military arrangements, about meeting place, about joint action, or about joint objective. The search process may involve the exchange of information, the relaxation of initial goals, mutual concession, lies, or threats.

In [Koistinen and Seetharaman, 1998], QoS negotiation in a distributed object infrastructure is defined as a process used by a client and a server to reach an agreement on QoS characteristics for their services considering their expected loads, network characteristics, and other influential factors. QoS negotiation is necessary for applications to adapt to environments in which resource availability and load varies.

Negotiation also exists in the context of a Dynamic e-Business (DeB), which is also known as a virtual enterprise, where a DeB consists of a variety of dynamic services, offered by different service providers and selected on-demand, that cooperate for a short period of time [Keller, 2002]. In DeB the customer, service integrator, and service providers negotiate and sign an electronic contract (a.k.a. Service Level Agreement, SLA) that specifies the guaranteed quality of the subscribed services and the compensating actions in case of a violation.

We are interested particularly in the negotiation between cooperating components and between the component containers, in the context of a component-based application. This negotiation is driven by a user's QoS requirement and the available resources. The QoS contract negotiation process selects *appropriate* implementations and QoS-Profiles of the interacting components in order to make the requested service available at the required level of quality.

All of the aforementioned aspects of automated negotiation have some common basic characteristics. Three elements have been identified to be important topics in automated negotiation research [Jennings, 2001]. These are *Negotiation Protocols*, *Negotiation Objects*, and *Decision Making Models*. The relative importance of these elements varies according to the negotiation and environmental context.

Negotiation Protocols define the set of rules that govern the interaction. This covers the permissible types of participants, the negotiation states, the events that cause negotiation states to change and the valid actions of the participants in particular states. *Negotiation Objects* define the range of issues over which agreement must be reached. At one extreme, the object may contain a single issue (such as price), while on the other hand it may cover hundreds of issues (related to price, quality, timings, penalties, terms and conditions, etc.). *Decision Making Models* are the decision making apparatus the participants employ to act in line with the negotiation protocol in order to achieve their objectives.

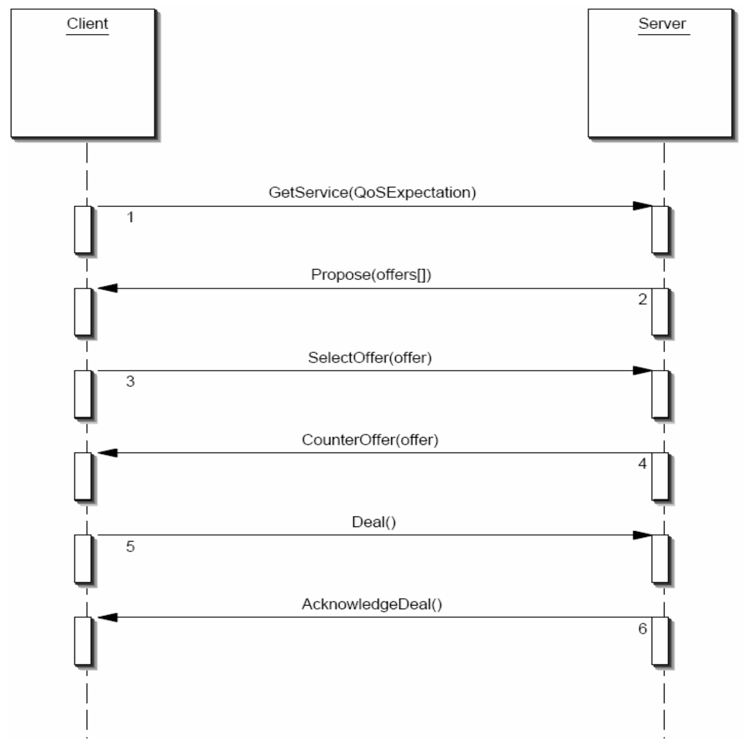


Figure 3.1: A QoS Negotiation Scenario in a Client / Server Application

A minimum requirement of a negotiating agent is the ability to make and respond to *proposals* [Jennings, 2001]. Proposals can be made either independently of other agents' proposals or based on the negotiation history. When one agent makes a proposal, if the interacting agents can only accept or reject other's proposals, then negotiation can be very time consuming and inefficient since the proposer has no means of ascertaining why the proposal is unacceptable. To improve the efficiency of the negotiation process, the recipient needs to be able to provide more useful feedback on the proposals it receives. This feedback can take the form of a *critique* (comments on which part of the proposal the agent likes or dislikes) or a *counter proposal* (an alternative proposal generated in response to a proposal). From such feedback, the proposer should be in a position to generate a proposal that is more likely to lead to an agreement. A negotiation scenario used in [Koistinen and Seetharaman, 1998] is shown in Figure 3.1. In step 3, the client selects the offer with the highest worth value from the multiple offers proposed by the server. In step 4, it is assumed that the server cannot accept the chosen offer. Hence, it computes a counter offer and sends it to the client.

3.2 Our Negotiation Model (in a single container)

We describe and illustrate our negotiation model based on the common building blocks identified for any automated negotiation research. In our case the negotiating parties are the components, the user, and the containers (run-time environment). The containers have a role of arbitration. The collaborating components and the user put forward *all* their proposals (i.e. the multiple QoS-Profiles of the components and the user's QoS requirements and preferences) and the container (arbitrator) analyzes all the problems (e.g. constraints) and dictates the solution for the parties (i.e. choose appropriate implementations and QoS-Profiles of the components). After a successful negotiation, contracts are established and will be monitored and enforced by the container. If no agreements can be reached, the user is asked to make a concession (to decrease his/her QoS requirements). In the event that no agreement is reached after the user relaxes his/her requirement, the negotiation terminates with a "conflict deal". A simplified scenario of our negotiation model that involves only a single container is depicted in Figure 3.2.

In steps 3) and 4) of Figure 3.2, it is assumed that the QoS-Profiles are specified by the component developers. In steps 9), 10), and 11), contracts are established for interacting components or between a user and interacting component. In our model we assume that components propose (or specify) all possible offers. The other possibility would have been for components to propose an offer and when this is not accepted by the container, to make counter-offers.

The *negotiation objects* in our case are non functional properties (or the so-called extra-functional properties) of components and containers. Distinction is usually made between non-functional properties (NFPs) and functional properties of a certain system. *Functional properties* of a system relate to the specification of *what* the system should do. The *extra-functional properties*, also called qualities, address *how well* this functionality is (or should be) performed if it is realized. For a general discussion on properties and their classification refer subsection 3.4.1.1.

In the sections and chapters that follow, we present an in-depth discussion of the aforementioned three elements of automatic negotiation in the context of QoS contract negotiation in distributed component-based software. We begin in the next section by formalizing the problem and subsequently discuss the proposed solutions.

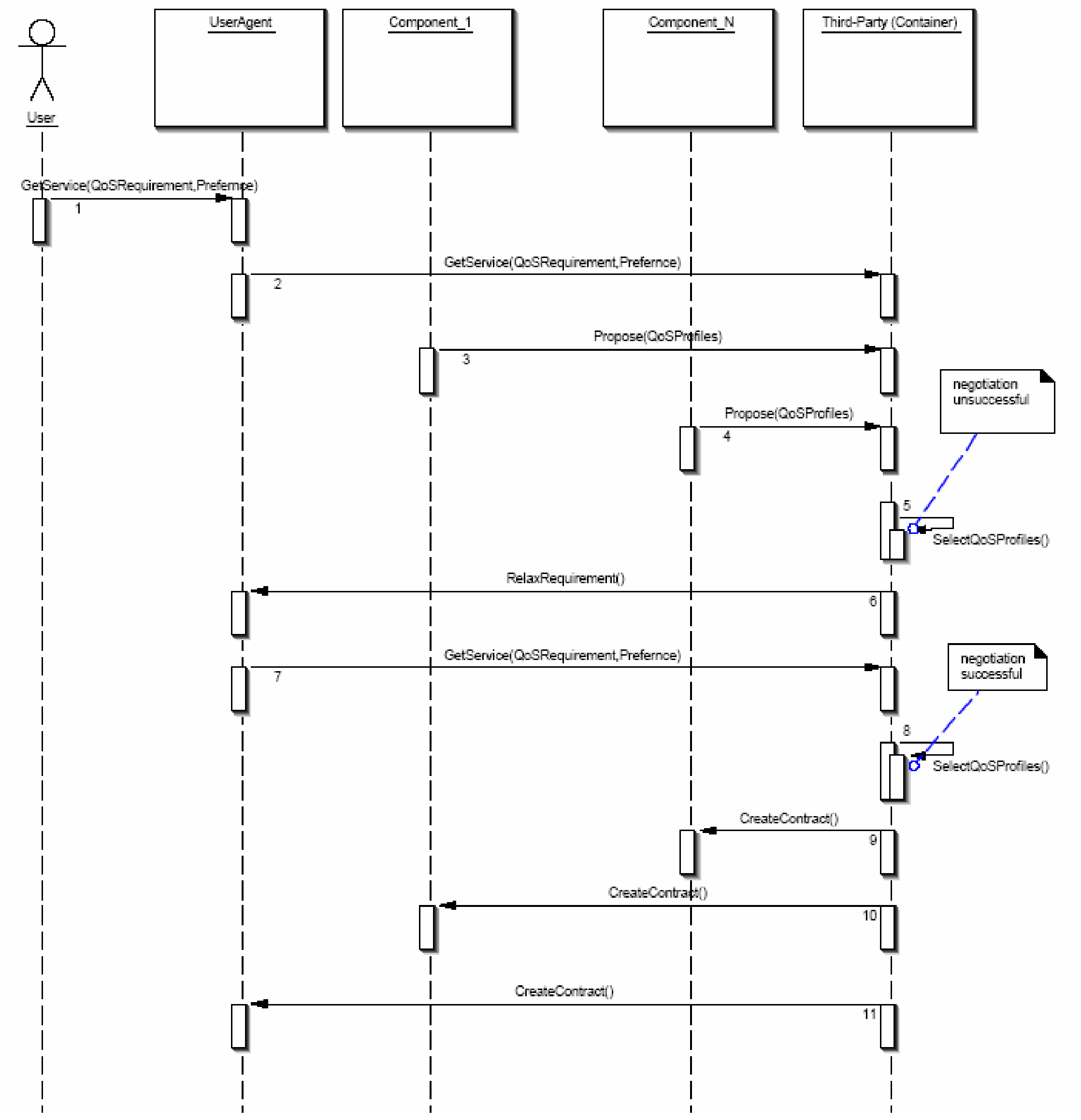


Figure 3.2: A simplified scenario of our negotiation model (in a single container)

3.3 Problem Formulation

Before formalizing the QoS contract negotiation as a Constraint Satisfaction Problem (CSP), we describe briefly the notions of: i) utility value [Lee et al, 1999] of an application, and ii) conformance [Frølund and Koistinen, 1999] that should exist between constraints of interacting components.

3.3.1 Utility

In [Lee et al, 1999], the notion of utility has been used to represent varying satisfaction with QoS changes of an application. A utility value is also defined in [Walpole et al, 1999] as a measure of usefulness, represented by real numbers in the range [0, 1] where 0 represents useless and 1 represents as good as perfect. Our usage of the term utility is similar to these definitions. We use the utility concept in order to compare different potential solutions (resulted from the composition of QoS contracts) in terms of the overall quality they provide.

There are two utility values that we should differentiate – the *application utility* and the *system utility*. The application's utility represents the quality of the provided service of an application as perceived by the user. The system utility is defined for the overall system, with multiple applications or clients.

An application's QoS can be represented by a number of QoS dimensions. For instance, in our video streaming scenario (Section 3.5), the application's quality characteristics are specified by `frame rate` and `resolution` properties. The *VideoServer* component's QoS-Profile is defined based on these NFPs (Figure 3.5 and Table 3.1).

A dimension-wise application utility is defined for each QoS dimension specified for the application. This definition or mapping can be done by an expert of the application domain or the user of the application. In [Lee et al, 1999], dimension-wise utilities are defined by the user. We think that in a contract negotiation framework, a default dimension-wise utility may be provided by an expert. The user can also be given the facility to alter this default, when needed. The QoS mapping on one QoS dimension can differ among users and also from application to application.

We refer to our video streaming scenario to briefly explain how a utility can be defined for an application that involves multiple QoS dimensions. Let's assume the values of the `frame rate` property range from `1 frame/s` to `30 frames/s (fps)`. `30 fps` is mapped to a utility of `1` while `1 fps` may be assigned to a utility of `0`. If the utility function is to be a linear one, these two points are enough to compute the utility function. Accordingly, the utility for `frame rate` is given as: `U(frameRate) = (1/29)(frameRate - 1)` (Figure 3.3). For the `resolution` property, let's assume that the possible values are only three, as shown in Table 3.1. The assigned utilities are then `0.8` for `352x288`, `0.6` for `176x144`, and `0.4` for `128x96`. The fact that no `0` value is assigned as a utility for `resolution` implies that none of the choices are considered to be useless. It is possible, however, that some users may assign a utility of `0` to `128x96`.

It is to be emphasized that neither a user nor an expert can specify the dimension-wise utility values for every QoS points of the application as this is simply impractical due to the many QoS points. But, as explained in [Lee et al, 1999], the utility of selected QoS points could be specified and then these points are interpolated in order to obtain the utility function. The example given above for the dimensional utility of frame rate uses only two points to define a linear utility function (for e.g. Figure 3.3). If an exponentialdecay utility function was chosen (for e.g. Figure 3.4), one would have to determine the constants from a specification by the user for two points (e.g. a utility of 0.5 for 5fps and a utility of 0.95 for 20fps). Having a very slow increasing rate at higher frame rates compared to the linear function, the exponential utility function reflects the fact that the improvement in quality as perceived by humans diminishes at higher frame rates.

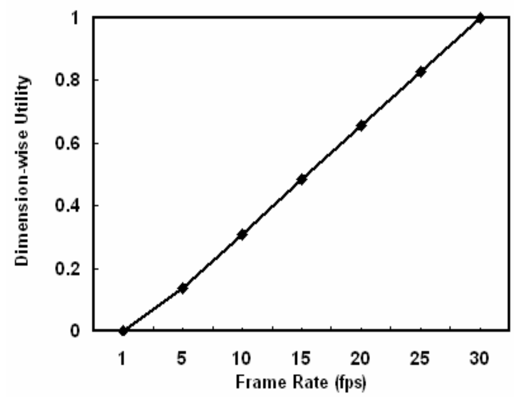


Figure 3.3: Linear dimensional utility

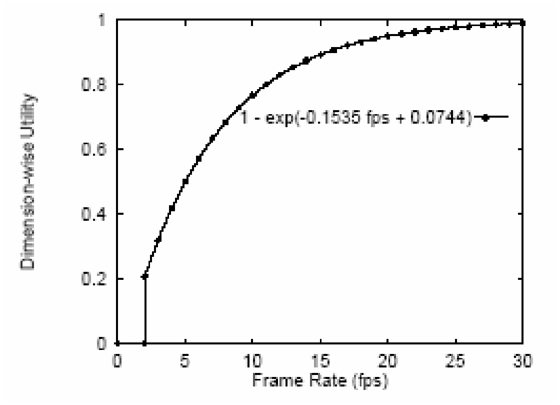


Figure 3.4: Exponential dimensional utility

In an application with multiple QoS dimensions, there could exist a dependency among the QoS dimensions. Two QoS dimensions, Q_a and Q_b are said to be independent of one another if a quality increase along Q_a (Q_b) does *not* increase the resource demands to achieve the quality level previously achieved along Q_b (Q_a). 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 [Rajkumar et al, 1997].

In a video streaming application, `frame rate` and `resolution` are dependent QoS dimensions. The utility function for such an application can be given as a weighted average of the dimension-wise utilities. This technique has been applied in [Lee et al, 1999] [Rajkumar et al, 1997], which is based on the analytic hierarchical process (AHP) [Saaty, 1992] with regard to the computation of weights. We take a user's relative preference towards each NFP as a weight for the corresponding NFP. The sum of all weights must be equal to 1. For example, a user who prefers `frame rate` over `resolution` in a certain video streaming application can assign $w_1=0.6$ for `frame rate` and $w_2=0.4$ for `resolution` as weights. Note that the determination of weights is an area that requires further study and is beyond the scope of this thesis. By using the dimension-wise utilities and the chosen relative weights, the overall utility is computed as shown in Table 3.1.

Profile Nr.	`ICompVideo` interface		Overall utility
	`frame rate in` (s^{-1})	`resolution`	
1.	30	352x288	0.92
2.	30	176x144	0.76
3.	30	128x96	0.72
4.	20	352x288	0.72
5.	20	176x144	0.56
6.	20	128x96	0.52
7.	10	352x288	0.51
8.	10	176x144	0.35
9.	10	128x96	0.31

Table 3.1: Utilities for a *VideoServer* Component

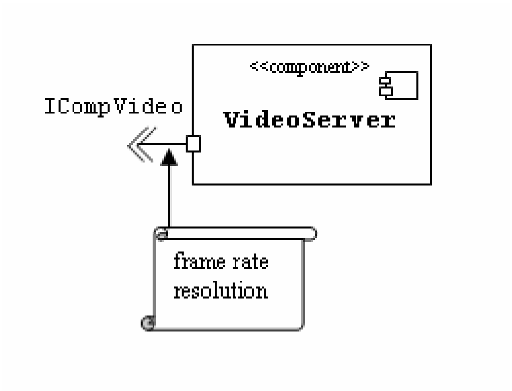


Figure 3.5: NFPs in a *VideoServer* Component

An application may attain the same utility value (for example, same frame rate and resolution as perceived by the user) under different resource conditions through what is known as *resource trade-off*. Compression techniques are applied to achieve resource trade-off. In our approach, the QoS-Profiles specify the NFPs of components. Compression types are considered to be properties of the component together with the QoS dimensions. This simplifies the mapping of Utility values to resources as there is explicitly the compression type in the association although compression is transparent to the user (assuming it is lossless). If the compression type were to be implicit, the QoS-Profiles would have to specify multiple resource assignment in order to achieve a certain QoS point.

During negotiation, we are interested in finding a "better" solution among the set of possible solutions. This is achieved through appropriate selection of QoS-Profiles of the collaborating components. Let's illustrate what we mean by a "better" solution. A more descriptive definition is given to our notion of "better" solution in sub-section 3.3.3. Suppose the user's QoS requirement to be `20 frames/sec` and a resolution of `176x144`. The user's overall utility is thus `(0.6*0.66)+(0.4*0.4)=0.56`. While profiles 1 to 5 (in Table 3.1) all satisfy the user's requirement, profiles 1 to 4 constitute the "better" solution as their utility is higher than the solution that just fulfills the user's requirement. In Table 3.1, the profiles on top give "better" solution than the ones below. Profile 1 represents the optimal solution as it has the highest utility. The selection of QoS-Profiles is determined by the resource availability and the existence of a matching profile of the interacting components. The matching process uses the notion of conformance as discussed in the next section.

3.3.2 Conformance Relationship between Constraints

In QML [Frølund and Koistinen, 1999], a general conformance relation is defined between two constraints. For example, the constraint *`delay<10`* "conforms" to the constraint *`delay<20`*. "Conforms" can be interpreted here as a "stronger than" relationship. The relationship between the two constraints is defined in this way due to the fact that `delay` is a decreasing QoS dimension (smaller values are better). Conformance can be applied to the QoS-Profile matching of interacting components. A QoS-Profile is an association of one or more constraints on a component's implementation. Between two QoS-Profiles of interacting components, a conformance is said to exist when the constraints in the server's provided-QoS contract conforms to the constraints in the client's required-QoS contract. As explained earlier, a QoS-Profile defines the provided- and required-QoS contracts and the associated resource demand of a component.

When components are distributed across containers, the conformance relationship must take into account the influence of the network and containers between the interacting components. Consider the example in Figure 3.6.

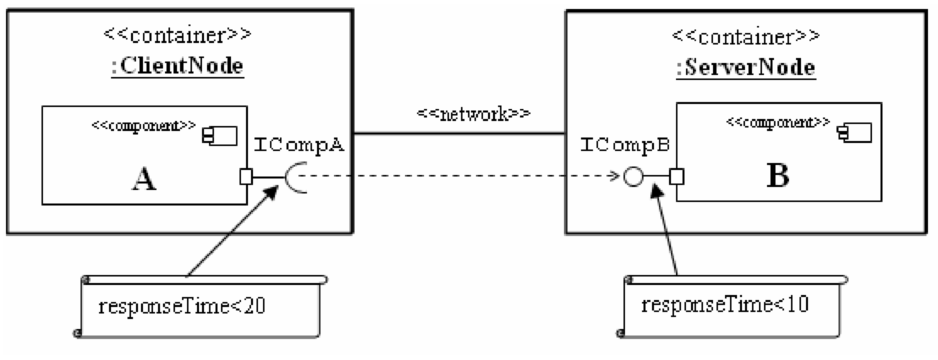


Figure 3.6: Two distributed interacting components that constrain the `response time` property in their provided and required interface

In Figure 3.6, the `responseTime<20` is the constraint in the used interface of *A* (`ICompA`) while `responseTime<10` is the constraint in the provided interface of *B* (`ICompB`). If the delay caused by the network and the containers is estimated to `delay<5`, `responseTime<20` still conforms to `responseTime<10`. In general, `responseTime<`t_A (constraint in `ICompA`) conforms to `responseTime<`t_B (constraint in `ICompB`) if t_A is less than the sum of t_B and the delay introduced by the network and the containers.

Consider now Figure 3.7 that depicts *VideoPlayer* and *VideoServer* components and the considered properties – frame rate and resolution – at each interface. The frame rate and resolution properties attached to the used stream interface of *VideoPlayer* constrain what the *VideoPlayer* expects from *VideoServer*. For the *VideoServer*, the frame rate property is the encoded frame rate of the streamed video. The question now is: how can we check whether the constraint in the *VideoServer* conforms to that required by the *VideoPlayer*?

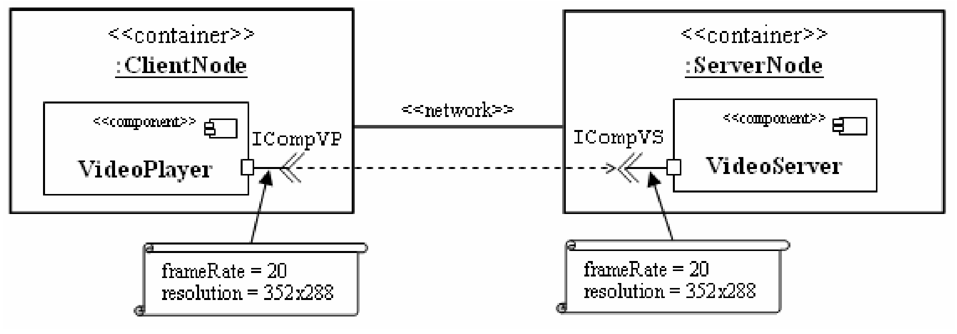


Figure 3.7: NFPs of *VideoPlayer* (at used stream interface) and *VideoServer* (at provided stream interface) deployed on a client and server nodes

In the process of streaming, a packet-loss is said to occur when: (i) the packet never arrives its destination, or (ii) it arrives later than its scheduled play-out time. In the latter case, however, the delayed packet may be useful for subsequently received frames. These facts demonstrate that a frame rate property can be dependent on a different parameter, i.e. a packet-loss rate as opposed to a response time property, which is dependent only on a similar property, i.e., the end-to-end delay due to the network. Deriving a conformance relation when a NFP depends on dissimilar parameters is a difficult task. An in-depth study of the conformance relationship is beyond the scope of this thesis.

In our modeling of streaming applications, we assume no packet-loss to occur when the network is not congested (i.e. there is enough end-to-end bandwidth for the streaming) or there is enough buffer size in the *VideoPlayer* component. We define the conformance relationship for the frame rate and resolution properties based on this assumption as follows:

```
frameRate (in ICompVP) = frameRate (in ICompVS)
resolution (in ICompVP) = resolution (in ICompVS)
```

Conformance relations for security properties are given in Section 4.2.3. We will next formalize the QoS contract negotiation as a constraint satisfaction problem.

3.3.3 QoS Contract Negotiation as a Constraint Satisfaction Problem (CSP)

The objective of the QoS contract negotiation is the selection of QoS-profiles so that the composed application meets the user’s QoS requirements and preferences. As a goal, the negotiation aims at finding a “better” solution among the set of possible solutions.

A CSP consists of n variables $x_1, x_2, \ldots, x_n$, whose values are taken from finite, discrete domains $D_1, D_2, \ldots, D_n$, respectively, and a set of constraints on their values. In general, a constraint is defined by a predicate. That is, the constraint $p_k(x_{k1}, \ldots, x_{kj})$ is a predicate that is defined on the Cartesian product $D_{k1} \times \ldots \times D_{kj}$. This predicate is true iff the value assignment of these variables satisfies this constraint. Solving a CSP is equivalent to finding an assignment of values to all variables such that all constraints are satisfied. Since constraint satisfaction is NP-hard in general, a trial-and-error exploration of alternatives is inevitable [Yokoo and Hirayama, 2000].

For the formalization, we take the variables to be the QoS-Profiles of the collaborating components $C_1, C_2, \ldots, C_n$. There are n variables – $P_1, P_2, \ldots, P_n$ – each representing the QoS-Profiles in use by $C_1, C_2, \ldots, C_n$ respectively. The domain of each variable is the set of all QoS-Profiles specified for a component. The constraints are classified as *conformance*, *user’s* and *resource*. As explained earlier, a QoS-Profile P_i defines the provided and required QoS contracts and the corresponding resource demand of a component, which we would designate as: P_i`.Offered`, P_i`.Required`, and P_i`.Resources` respectively.

In the offered and required QoS contracts, one or more QoS characteristics (e.g. *delay*, *frameRate*) are constrained with values (e.g. *delay < 5* or *frameRate = 15s*$^{-1}$). Suppose d_1, d_2, ..., d_k to be the considered QoS characteristics. The conformance constraint between P_i and P_j is valid if there is conformance between each corresponding QoS characteristics in P_i and P_j. We designate the conformance requirement as (P_i.Required.d_j => P_j.Offered.d_j). This conformance constraint can apply to a single method of the interface implemented by the components or to the entire methods.

As an example, for the application that involves three components as depicted in Figure 3.8, the QoS contract negotiation problem is formalized as in Table 3.2, where all the components are assumed to be deployed in a single container. For multiple containers, the conformance and resource constraints must be modified as follows. The influence of the network and the containers must be incorporated in the conformance constraint for those components connected across containers. For example, for a `response time` (`RT`) property, the constraint can be modified as: (P_i.Required.RT => P_j.Offered.RT + t_{delay}), where t_{delay} is the delay due to the network and containers. t_{delay} is assumed to be a constant for the period the negotiation agreement is valid. In the resource constraint, for two containers case, three relations must be defined instead of only one relation as shown in Table 3.2. These are for components deployed on the client, on the server, and for components connected across containers.

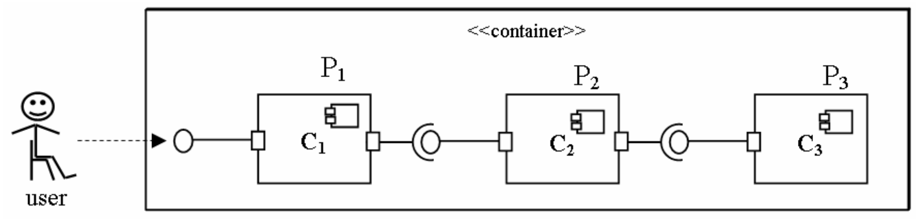


Figure 3.8: Example Component-Based Application

Variables	P_1, P_2, and P_3 (these are QoS-Profiles in use by C_1, C_2, and C_3 - Figure 3.8)
Domains:	for each component, a set of QoS-Profiles are specified by the component developer
User's Constraint	user's QoS Requirement on d_1 > P_1.Offered.d_1 … user's QoS Requirement on d_k > P_1.Offered.d_k
Conformance Constraint	(P_1.Required.d_1 => P_2.Offered.d_1) … (P_1.Required.d_k => P_2.Offered.d_k) (P_2.Required.d_1 => P_3.Offered.d_1) … (P_2.Required.d_k => P_3.Offered.d_k)
Resource Constraint	P_1.Resources + P_2.Resources + P_3.Resources ≤ Resources_{avail}

Table 3.2: Formalizing QoS contract negotiation as a constraint satisfaction problem

Resource types specified in a certain QoS-Profile, `Pi`, could be CPU, memory, and network bandwidth. The resource constraint is based on the following assumption. Suppose there are `n` components - C_1, C_2, …, C_n - deployed in a container and these components need `m` different resource types. Let R_i be the resource demand of C_i, R_{all} is the resource demand of the `n` components, and R_{avail} is the available resource. That is,

$R_i = [r_{i1}, r_{i2}, ..., r_{im}]$ where r_{ij} is the resource demand of C_i for the j^{th} resource type

$R_{avail} = [r_1^{avail}, r_2^{avail}, ..., r_m^{avail}]$

$R_{all} = R_1 + R_2 + ... + R_n$

$R_{all} = [r_{11} + r_{21} + ... + r_{n1}, r_{12} + r_{22} + ... + r_{n2}, ..., r_{1m} + r_{2m} + ... + r_{nm}]$ where the addition used is arithmetic

The resource constraint is said to be met under the following conditions.

$$\begin{aligned} r_1^{avail} &\geq r_{11} + r_{21} + ... + r_{n1} \\ r_2^{avail} &\geq r_{12} + r_{22} + ... + r_{n2} \\ &\cdots \\ r_m^{avail} &\geq r_{1m} + r_{2m} + ... + r_{nm} \end{aligned}$$

There are enough resources for the `n` components if and only if for each type of resource, the resource requirement is not larger than the corresponding available resource.

In CSP there can be several solutions that satisfy all the constraints. Each solution is assumed to be as good as any other one. But, when considering the QoS of applications, some solutions may be "better" than others. One of the challenges we face in the QoS contract negotiation is the choice of a "better" solution according to some goal (for example, user's satisfaction). In order to address this challenge, modelling the negotiation as a Constraint Satisfaction Optimization Problem (CSOP) would be helpful.

The term Constraint Satisfaction Optimization Problems (CSOP) is used to refer to the standard CSP plus the requirement of finding optimal solutions [Tsang, 1993]. A CSOP is defined as a CSP together with an optimization function *f* (this function is also called objective function) which maps every solution to a numerical value. The task in a CSOP is to find the solution with the optimal (minimal or maximal) value with regard to the application-dependent optimization function *f*.

We use as an objective function, *f*, the utility function that is described in sub-section 3.3.1. Defining a utility function is a difficult task due to the inter-dependency of QoS-dimensions. In this thesis, we simply assume that *f* is given as a weighted sum of dimension-wise utilities. An in-depth treatment of utility functions for applications with multiple QoS-dimensions is beyond the scope of this thesis and is still an open problem.

We define next certain terms to clarify their usage in this thesis.

Definition 1: a *solution* to a QoS contract negotiation constitutes a selection of implementations and QoS-Profiles of the cooperating components that fulfils the conformance, user's, and resource constraints.

Definition 2: a solution *A* is a "*better" solution* than another solution *B* if *A's* utility is higher than that of *B*. The successive improvement on a "better" solution would ultimately lead to an optimal solution.

Definition 3: an *optimal solution* is a solution to the QoS contract negotiation problem that gives the highest utility.

The aim of finding a "better" solution is to find a good solution rather quickly as compared to finding the optimal solution, which is usually NP-hard. Multiple parties are normally involved in a negotiation and what is "better" for one (e.g. the service provider) might not be so for the other (e.g. consumer). In this section, our notion of a "better" solution is based on a user's satisfaction. We will explain how to incorporate the interests of two parties in the negotiation when we address the multiple-clients scenario in sub-section 4.1.

It might not be possible to find a solution at all as per Definition 1 above. In this case, the CSP is said to be over-constrained and how this situation is treated is explained in sub-section 3.4.5.

3.4 The Proposed Approach

To find an optimal solution for the resulting CSP (Table 3.2), in a *naïve approach*, all possible assignments of the variables (QoS-Profiles) must be considered. Then, QoS-profiles that give the highest utility and where all constraints are met are selected. This approach requires an exhaustive search of all possible configurations. In general, constraint satisfaction is NP-hard [Yokoo et al, 1998]. For applications with few possible configurations, the naïve approach can be used. In other cases, heuristics must be applied to find good or what we call "better" solutions quickly. We tailor general-purpose heuristic mechanisms to help us find the "better" solution faster. We also propose our own problem-specific heuristics that help in decreasing the search space. Before discussing the various heuristics, we argue why the QoS contract negotiation should be performed in multiple phases.

3.4.1 The need for multiple phases in the QoS contract negotiation

While investigating QoS contract negotiation in various componentized application scenarios, we realize that the negotiation process is complex and less efficient unless some kind of phasing or ordering is applied. The complexity arises from the complex dependency[2] exhibited by component properties and from the different nature of these properties as illustrated below.

1. Compression algorithms affect the QoS contract of components. For instance, in video streaming applications, for different video coding types, the frame rate and resolution properties take different values under the same resource allocation of each component. Actually the purpose of having various coding types is to achieve resource trading between the network bandwidth and the client or server CPU time in order to target environments with certain resource availability levels. If the negotiation between interacting components were to be performed on the three properties (coding type, frame rate, and resolution) together in a single phase, the negotiation would be more complex because of the resource trading. Our desire to find a "better" solution under conditions of resource constraints on clients, server, and network bandwidth further increases the complexity.
2. Components may be implemented to use different communication protocols. Cooperating components must ensure their protocols agree in order to interact with one another. For instance, in a video streaming scenario, a component's implementation may use RTP/UDP, or UDP, or TCP, or custom protocols. For applications with request/reply kind of communication too, different protocols may be specified for each of a component's implementation. To simplify the QoS contract negotiation process, we require the negotiation on protocols be accomplished separately. Moreover, negotiation on the communication service models (e.g. RSVP, best-effort) that should exist between the multiple containers hosting the components must be performed before the QoS contract negotiation.
3. When considering security properties, a component's port may be specified with security protection goals and associated security mechanisms. In [Franz and Pohl, 2004], a five level gradation (*unconditional*, *if_possible*, *dont_care*, *if_necessary*, *on_no_condition*) has been suggested to express protection demands for component interaction. Associated to certain levels of gradation, security mechanisms must also be specified. This can be on key type, key length, encryption algorithm, etc. Negotiating protection goals together with security mechanisms make the agreement process complex.

The above three points demonstrate why a possible classification of properties and their negotiation can simplify the negotiation process. Let's see how efficiency can be achieved through our classification scheme. Consider the application in Figure 3.9, which also shows the considered properties of the cooperating components.

[2] The dependency here is different from the one that exists between provided- and required-QoS contracts, which is captured in a QoS-Profile; or the one that exists between two properties like security and response time, which we also assume to be captured in a QoS-profile.

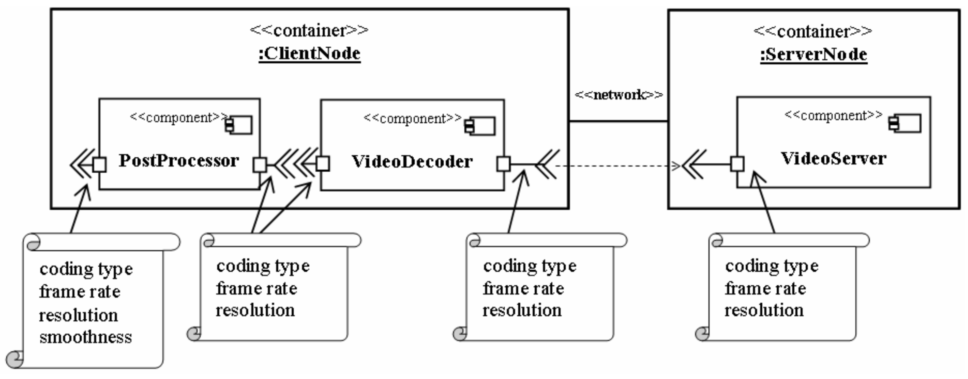


Figure 3.9: Components and considered properties in a video streaming scenario

The NFP specification of the components in Figure 3.9 will be done in such a way that for a certain value of `coding type` multiple values of `frame rate` and `resolution` are specified. If all properties were treated at the same level, the conformance constraint checking between provided-QoS contract and required-QoS contract of interacting components would have to be evaluated on all the properties in an atomic manner. Suppose *VideoServer* is specified by considering 3 coding types, and for each coding type 10 different frame rates and 5 different resolutions are specified, respectively. This gives a total of 150 different alternatives in the component implementations and QoS-Profiles of *VideoServer* alone. Similarly *VideoDecoder* is specified with the same number of the three properties (whose values don't necessarily match those of the *VideoServer's*). Finally, *PostProcessor* is specified with only one coding type and the same number of frame rates and resolutions to those of the other two components. The search space in the conformance constraint checking between the three components constitutes 150*150*50=1125000 possibilities. If classification is considered, i.e. during the first phase negotiation is made on coding type, and in the second phase on the frame rate and resolution, then the search space would be 1*3*3 + 50*50*50 = 125009. The computation of the search space in the second phase is made by assuming that only an agreement is made on one coding property during the first phase. For the above specific example, the reduction in the search space is about 89%. It has to be emphasized, however, that this approach aims at finding a "better" solution with respect to a user's preferences. This may lead to missing out "better" solutions with respect to global optima.

Next, we will give a general discussion on the classification of properties before describing the multiple phases that aim at simplifying the negotiation process and making it more efficient.

3.4.1.1 On Properties and Their Classification

Properties of either components or applications are usually classified along functional and non-functional lines. The functional properties describe *what* the system does and the NFPs describe *how* the system does it. The division of properties into functional and non-functional isn't always clear-cut and can depend on the user's requirement toward the system's services. Usually NFPs of a system encompass the following aspects: QoS-properties (e.g. timeliness, accuracy), security properties (e.g. integrity, confidentiality), and also higher level end-user properties (e.g. fees for used services). In the literature we usually find the terms *non-functional properties*, *extra-functional properties*, *QoS attributes/properties*, *Ilities* used interchangeably. NFPs are classified in a number of different ways depending on the context of use. We will next explore the different approaches on the classification of these properties.

Probably many of the classification schemes base their approach on the ISO's quality model [ISO/IEC, 1999], which provides a standard for a software quality model that can be applied to any software product by tailoring it to a specific purpose. This standard defines a two-part model for software product quality: (i) *internal and external quality*, and (ii) *quality in use*. The first part identifies six quality characteristics (functionality, reliability, usability, efficiency, maintainability, and portability) that are further subdivided into sub-characteristics appropriate for internal and external quality. These characteristics are attributes of the software product that can be measured irrespective of context of use. The second part, i.e. quality in use, represents the user's view of the quality of the system. It is measured in terms of results of using the system, as opposed to measurements of the software product itself. It identifies four characteristics (effectiveness, productivity, safety, and satisfaction) appropriate to characterize quality in use.

QoS characteristics and categories have also been classified in [OMG, 2005] with the purpose of integrating them with the UML specification language and models. It defines particularly those characteristics important to real-time and high confidence systems, which describe the fundamental aspects of the various specific kinds of QoS and create a common framework that relates all of them. Standards like [ISO/IEC, 1999] [OMG, 2005] provide very general quality models and guidelines independent of domains and problems. They are rather more abstract and aim at the identification of characteristics and their grouping into higher level, say category, and lower level, say other category or sub-characteristics. These classification schemes usually say little or nothing about the correlation or dependencies of characteristics, which is usually the concern in a particular domain or problem.

We find also in the literature [Preiss et al, 2001] quality attributes classified as *run-time* and *life-cycle* attributes. The run-time properties are the quality attributes that are discernable at system execution time. Performance, dependability, usability, etc. are categorized as run-time properties. Each property may also be further classified as main-category – sub-category. For instance, the sub-categories identified for performance are: responsiveness, accuracy, footprint, timeliness, schedulability, etc. Life-cycle attributes (e.g. maintainability, portability) cannot be observed at run-time. They characterize different phases in a development and maintenance process. For maintenance, the sub-categories identified are: evolvability, extensibility, modifiability, upgradeability, etc. The classifications made in [Preiss et al, 2001] and [ISO/IEC, 1999] are essentially the same with the former using the terminologies *run-time* and *life-cycle* to mean *external* and *internal* properties respectively. The characteristic – subcharacteristic categorization in the latter is the same as the main-category – sub-category categorization in the former.

In [Rosa et al, 2002] a language for describing non-functional properties, called Process^{NFL}, has been proposed. Process^{NFL} has been designed in order to support particular characteristics of non-functional requirements such as their strong correlations, often conflicts and non-direct implementation nature. We are particularly interested in Process^{NFL} due to its abstractions for the relationships of properties. Non-functional (NF) issues are modelled by Process^{NFL} with the following abstractions: *NF-Attribute*, *NF-Property*, and *NF-Action*.

NF-Attributes model non-functional characteristics that can be categorized in any of the following classes: (i) ones precisely measured (e.g. performance), (ii) ones stated through levels (e.g. security), and (iii) ones just present in the final product (e.g. transaction). A NF-Attribute is usually decomposed into primitive NF-Attributes that are more detailed or closer to implementation elements. This aspect of decomposing an attribute into several primitive ones resembles the characteristic-subcharacteristic categorization used in [ISO/IEC, 1999] [Preiss et al, 2001].

A Non-Functional Action (NF-Action) models either any software aspect or any hardware characteristic that affect the NF-Attributes described earlier. Software aspects mean design decisions, algorithms, data structures, etc. while hardware characteristics concern computer resources available for running the software system. Considering the software aspect, an algorithm used to compress data has a direct influence on the NF-Attribute performance. The NF-Attribute security is not only affected by encryption algorithms but it is actually implemented by them. The notion of "correlation" is also captured in NF-Actions. Correlation refers to the fact that a NF-Action implementing or affecting a NF-Attribute may also have an effect over other NF-Attributes. For example, the NF-Action encryption algorithm implements elements of the NF-Attribute security. Furthermore, it also interferes in the NF-Attribute performance. Hence, the NF-Attributes performance and security are correlated.

A Non-Functional Property (NF-Property) models constraints over NF-Attributes. In practical terms, the constraint defines a subset of NF-Actions that must be actually implemented in order to satisfy the NF-Property. For example, the NF-Property *good performance* expresses a constraint over the NF-Attribute ("be good"). In this particular case, only NF-Actions that contribute to achieve a *good performance* must be taken into account.

Although Process^{NFL} enables capturing the relationship between non-functional properties and the implementation elements and also correlations between properties, this has been done somewhat qualitatively (e.g. with the notions of *in favour* and *against*; or through levels like high, medium, and low). Process^{NFL} uses a more general approach that isn't directly applicable in the context of CBSE (which for e.g. demands notions that capture the relationship between provided and required QoS contracts as discussed in [Reussner et al, 2003]).

On a different spectrum, there have been researches that in one or another aspect reasons about the characteristics of properties and their relationships. For instance, [Larsson, 2004] classifies properties based on composability. This classification is made according to the principles applied in deriving the system properties from the properties of components involved. A list of such classifications is directly-composable properties, architecture-related properties, derived properties, usage dependent properties, and system environment context properties.

3.4.1.2 Coarse-grained and fine-grained properties/negotiation

A coarse-grained property is a component implementation’s property that can be associated with one or multiple fine-grained properties. This association is created by the fact that for a certain value of the coarse-grained property the fine-grained properties can possibly take different values depending on the allocated resource (e.g. CPU, bandwidth, memory). Referring to the justifications we make for the phasing of negotiations, coding type and protocol qualifies to be coarse-grained properties while frame rate, resolution, and smoothness are fine-grained properties. Moreover, according to our classification, security goals specified for the component interfaces represent the coarse-grained properties while associated security mechanisms are fine-grained properties.

Different values of a coarse-grained property are *usually* realized in different component implementations, while the associated fine-grained properties are specified in the QoS-profiles (which describe different operating modes that can be chosen depending on their resource demands and the availability of the underlying resource). But this rule cannot be enforced. Such a rule can be a part of a component model that also incorporates different mechanisms of contract negotiation.

Coarse-grained negotiation is the negotiation on coarse-grained properties while fine-grained negotiation is the one on fine-grained properties. Both negotiations can be done between components and between the component containers. During the negotiation of properties between interacting components, a negotiation dependency exists between a coarse-grained property and its associated fine-grained properties. By this dependency we mean that the negotiation on the associated fine-grained properties should be performed if there is an agreement between the components on the coarse-grained property. A detailed discussion of these negotiation types is given in subsequent sections.

3.4.1.3 Negotiation Ordering

As pointed out earlier, the purpose of classifying negotiation objects (or NFPs) as coarse-grained and fine-grained is to bring about negotiation orderings, thereby simplifying the negotiation process and making it more efficient. A graph that represents such ordering is shown in Figure 3.10. Properties A and B are coarse-grained while C, D, and E are fine-grained properties.

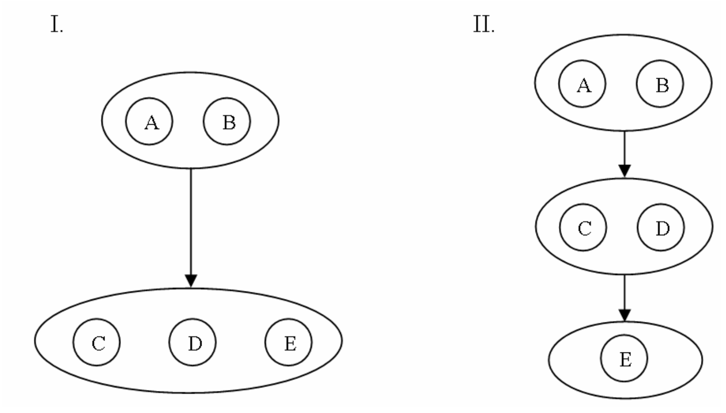


Figure 3.10: Negotiation ordering between coarse-grained (A and B) and fine-grained (C, D, and E) properties

In the first part of Figure 3.10, the fact that A and B are enclosed in a bigger circle indicates that properties A and B are negotiated together. The same thing applies to C, D, and E. The reason for negotiating several properties simultaneously is due to the fact that the QoS-Profiles of components define the multiple QoS properties collectively (this is of course specification language dependent).

In order to achieve efficiency, we can introduce negotiation orderings between fine-grained properties. This is possible, for instance, when certain properties are specified only for some components. Referring to Figure 3.9, the smoothness property is specified only for *PostProcessor* while frame rate and resolution are specified for all components. Under this condition, our strategy is to negotiate first properties that are common to all components to be followed by a negotiation on properties specified for some components. This is depicted in the second part of Figure 3.10 where among the fine-grained properties (C, D, and E), C and D are the common properties that are to be negotiated before E.

3.4.2 Coarse-Grained Negotiation

Container-Container negotiation

In our approach, negotiation exists not only among distributed components, but also between containers. The *container-container negotiation* must precede the component negotiation. The containers may need to agree on the type of communication service model (which also has to be supported by the network between them), like guaranteed, priority-based, or best-effort. A client may not support RSVP (Resource Reservation Protocol) [Braden et al, 1997] while the server supports it. Hence, in such a case, RSVP based guaranteed connections cannot be used and the parties need to agree on a different communication service model.

Coarse-grained Negotiation between Components

As explained in subsection 3.4.1, some examples of coarse-grained properties a component may have are: compression (coding) types, protocol, and security protection goals. In general, a coarse-grained property of a component is generally assumed to be specified with multiple values, which may be ordered or unordered. Ordering is followed to associate a preference with the various values. In our video streaming application scenario, *video coding type* is taken as a coarse-grained property for the interacting components. An example of a coding type property specification for a *VideoPlayer* component is shown in Table 3.3.

	Video Coding
1.	H.264
2.	H.263
3.	MPEG

Table 3.3: A specification for the video coding type property of a *VideoPlayer* component

Similarly specifications are made for components that interact with *VideoPlayer*. For two interacting components, the list of values specified for a certain coarse-grained property might not match. The most preferred value for a client component may be the least preferred for a server component. Whose preference to take in the negotiation (client or server) is thus something to be decided before the negotiation starts. In our present strategy, we first take the client's preference and give priority to it.

To draw a comparison on preferences of values, let us take an analogy from a scenario where a customer buys a shoe from a shoe store. The customer has preferences on the colors of the shoe in the order: black, brown. The seller may also have preferences for selling a shoe of the particular brand desired by the customer as: brown, black, white. Following the customer's preference, the black color is selected. Had we prioritized the seller's preference, we would have a different selection, i.e. brown. In the shoe buying-selling example, a number of attributes may be considered for negotiation like style, color, quantity, and price. For this particular example, a negotiation order can be followed such that first agreements must be reached on style, color, and quantity and when these are agreed upon, there is price negotiation. The three properties may be taken as coarse-grained properties while price can be taken as a fine-grained property.

Coarse-grained negotiation is successful when offer and expectation of the interacting components *conform* or *match* on the concerned property. In general, a conformance relationship is defined for each property to test whether the values of the corresponding properties match to each other. The idea of conformance (see subsection 3.3.2) in coarse-grained and fine-grained properties is the same.

If the negotiation results in the choice of a single value of the particular coarse-grained property, then we say a final decision has been reached on the property's negotiation at this phase. However, if the negotiation leads to an agreement on multiple values (both for ordered and unordered lists), the final decision on the values will have to be postponed until after the fine-grained negotiation. In the shoe buying-selling analogy discussed previously, a final decision is reached after the negotiation of the price, if the buyer and seller agree, say, on black and brown shoes.

Algorithm 3.1 outlines the steps to be executed in the coarse-grained negotiation between a client and server component. It assumes that the multiple values of a property are ordered. The algorithm can be applied for the unordered case as we can always introduce temporary ordering on the unordered list. When multiple coarse-grained properties are specified, Algorithm 3.1 must be applied for each property individually.

Algorithm 3.1: Coarse-grained negotiation algorithm between a client and server component

```
/* INPUT:
Specification of a coarse-grained property for a client and a server component

 OUTPUT: agreed value(s) of a property stored in coarseAgreementList
*/

List coarseAgreementList;

void CoarseGrainedNegotiation()
{
    coarseAgreementList ← Empty;
    for(int I=1; I <= total number of values specified for the coarse-grained property in
    the client component; I++) {
        Take the Ith preference of the client component;
        if( there exists a conformant value specified for the server component ) {
                coarseAgreementList ← the Ith preference of the client component;
        }
    }
}
```

After executing Algorithm 3.1, `coarseAgreementList` may:

i.) be empty – the whole negotiation process terminates with no agreement; or
ii.) have a single value – final decision is reached on the property's value at this phase; or
iii.) have multiple values – final decision on the property's value will have to be made during the fine-grained negotiation phase.

3.4.3 Fine-grained Negotiation

Fine-grained negotiation is the second and last phase of the contract negotiation process where appropriate implementations and QoS-Profiles are selected. This negotiation is performed on the fine-grained properties like frame rate, resolution, delay, and security mechanisms corresponding to the agreed values of the coarse-grained properties in the first phase of the negotiation.

The fine-grained negotiation is also responsible for finding a "better" solution in addition to just picking a solution that satisfies all the constraints. In order to accomplish this task, we extend the CSP into a CSOP framework as explained in subsection 3.3.3. Among the methods for tackling CSOPs are branch and bound (B&B) and genetic algorithms.

To apply the B&B to CSOP, one needs a heuristic function *h* which maps every partial labelling to a numerical value. Let this value be called the *h-value* of the partial label (partial assignment). A global variable, referred here as *BOUND*, will be initialized to minus infinity in a maximization problem. The algorithm searches for solutions in a depth first manner. It behaves like chronological backtracking except that before a partial labelling is extended to include a new label, the *h*-value of the current partial labelling is calculated. If this *h*-value is less than *BOUND* in a maximization problem, then the sub-tree under the current compound label is pruned. Whenever a solution is found, the optimization function, referred as *f*-value, is computed. This *f*-value will become the new *BOUND* (i.e. the best solution so far) if and only if it is greater than the existing *BOUND* in a maximization problem. After all parts of the search space have been searched or pruned, the best solution recorded so far is the solution to the CSOP [Tsang, 1993].

B&B is a very general framework. To completely specify how a process that applies B&B proceeds, we need to define policies concerning selection of the next variable and selection of the next value, which are usually chosen on the basis of efficiency. Moreover, we must also specify the objective and heuristic functions. We propose next a B&B technique for the fine-grained negotiation by explaining the schemes we used to define:

1. Variable and value selection policies
2. Objective function, *f*
3. Heuristic function, *h*

1. Variable and value selection policies

A variable and value ordering is a general purpose heuristics used to solve CSPs efficiently [Russell and Norvig, 2002]. In this method, the variable to assign next is appropriately selected. After choosing the variable for assignment, the value to assign to must also be appropriately singled out. We employ this heuristic to make the B&B method more efficient.

The variables (QoS-profiles) are ordered for assignment by topologically sorting the network of cooperating components. By such ordering, the front-end component (e.g. C_1 in Figure 3.8) becomes the minimal element. Users interact with a front-end component. The assignment starts from the minimal element and from there continues to the connected components, and so on. We assume that the cooperating components form only a tree. Thus, the ordering also forms a partial order. As will be pointed out later, this particular variable ordering simplifies the definition of the heuristic function, *h*.

The possible values of each variable, i.e. the QoS-profiles specified for each component, must be ordered from higher to lower quality. As contracts involve multiple QoS properties, the ordering is first based on the QoS property that is the most preferred by the user, then the next preferred, etc. Generally, we assume user's preferences are different for each QoS property. This value ordering scheme is similar to lexicographic ordering. As an example, let's assume (d_1,d_2,d_3) and (e_1,e_2,e_3) represent the QoS points in the provided QoS contract of two QoS-Profiles of a component that defines contracts for three properties. Then, (d_1,d_2,d_3) is ordered on top of (e_1,e_2,e_3) if $d_1>e_1$; or $d_1=e_1$ & $d_2>e_2$; or $d_1=e_1$ & $d_2=e_2$ & $d_3>e_3$.

2. Objective function, *f*

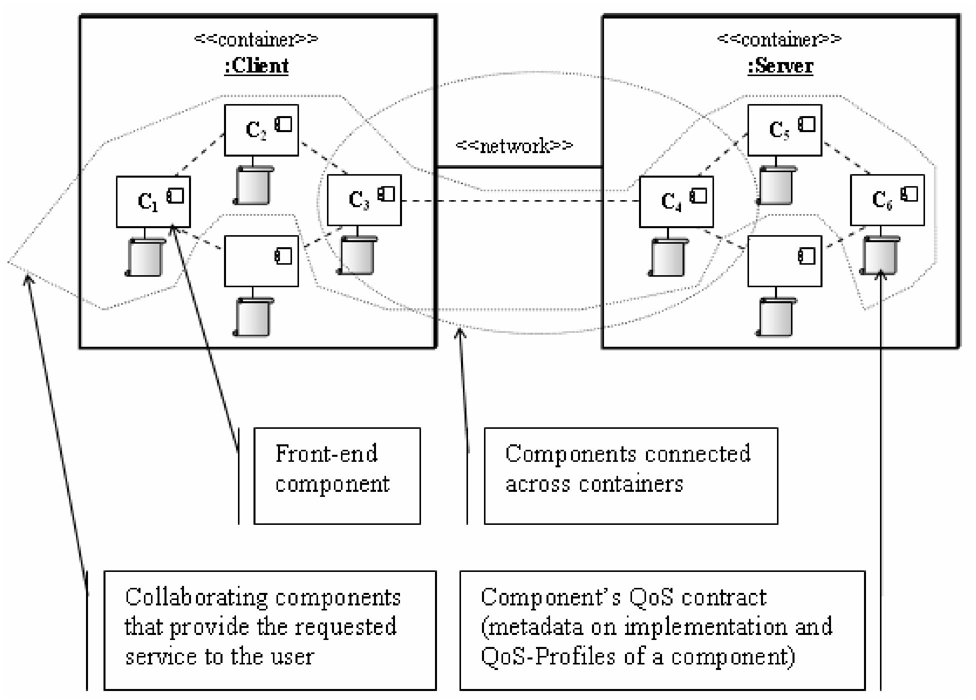
The objective function, *f*, is the utility function that has been described in subsection 3.3.1. The utility value is obtained by computing the utility function at the selected quality points of the application.

3. Heuristic function, *h*

At any point during the assignment of values to variables, the QoS property of the partially completed solution can be taken as the provided-QoS contract of the front-end component. Hence, *h* can be calculated based on the utility function by taking the QoS points in the provided-QoS contract of the front-end component. Because of the ordering strategy of variables we followed, *h* need to be computed only at the beginning of each iteration, that is, when the front-end component is assigned a new value. If the new assignment to the front-end component violates the user's constraint, the choice is retracted and the sub-tree under the particular assignment will be pruned. The process will then re-start with a new assignment.

3.4.3.1 On the Algorithm Design

Based on the problem formalization (subsection 3.3.3), three different types of constraints exist in the QoS contract negotiation. These are: user, conformance, and resource constraints. The conformance constraint is defined for two connected components while user's constraint is specified between a front-end component's provided QoS contract and the user's QoS requirement. Hence, both conformance and user's constraints are *binary* constraints. The resource constraint on the other hand is an *n-ary* constraint, which is specified for a group of components. Figure 3.11 shows the categorization of components where a resource constraint can be specified for each of the groups. For instance, the resource demand of profiles of components deployed in the client container must not be greater than the available resource in the client node. Or, for components connected across containers, the network bandwidth demand in the selected profiles must be smaller than the available bandwidth. As depicted in Figure 3.11, we assume that the components that collaborate to provide the requested service to the user must be known to the algorithm.



<<container>>
:Client
<<container>>
:Server
<<network>>
C_1
C_2
C_3
C_4
C_5
C_6
Front-end component
Components connected across containers
Collaborating components that provide the requested service to the user
Component's QoS contract (metadata on implementation and QoS-Profiles of a component)

Figure 3.11: Components deployed in a client and server container to provide a service to the user

The fact that there exist different types of constraints (binary and n-ary) and components are deployed in multiple containers brings about some unique issues that have to be considered while designing the algorithm. One such issue is the decision on when to check the validity of resource constraints. For this, one possibility is to check the resource constraint after conformant profiles of all components have been selected. But, breaking a problem into sub-problems usually helps in finding the solution quickly [Russell and Norvig, 2002]. Applying this notion, the algorithm will check the resource constraints at three instances – i.e. after conformant profiles have been found for: (i) components deployed in the client container, (ii) components connected across containers, and finally (iii) for components deployed in the server container.

As the components are distributed in a client and server containers, one approach to follow in the algorithm would be to first find QoS-Profiles of components deployed on the client container, which satisfy the three constraints and at the same time give a "better" solution. Then apply the same procedure to components connected across containers and lastly to components deployed on the server container. This approach would have a shortcoming as it could lead to more backtrackings if the *bottleneck resource* were either the network bandwidth or server resources. What we will follow in the algorithm is to find conformant profiles for components deployed on both containers and check the resource constraints three times as explained above. This is done iteratively – from low to high quality – in search of a "better" solution. When a bottleneck resource is found, the algorithm stops (unless some refinements have to be made, which is possible by appropriately choosing ordering on the negotiation of the multiple properties). We will next give the algorithm together with some descriptions.

Algorithm 3.2: Fine-grained negotiation algorithm for components deployed in client and server containers

*/**
INPUT:
1. The QoS-Profiles of each component that are ordered from low to high quality. The collaborating components are assumed to be ordered too as C_1, C_2, …,C_n following the variable ordering strategy explained in subsection 3.4.3.
2. User's QoS requirements and preferences
3. Available resources at the client and server nodes and the end-to-end bandwidth

OUTPUT:
A QoS-Profile for each component that satisfy the user's, conformance, and resource constraints. In addition to fulfilling all the constraints, the selected QoS-Profiles provide a "better" solution (see Definition 2. in subsection 3.3.3)

**/*

```
// constants used as return types
   public static final int NO_RESOURCE = 0;
   public static final int SUCCESSFUL = 1;

   List<QoSProfiles> selectedQoSProfiles;

   enum CG { ON_CLIENT, ON_SERVER, ACROSS_CONTAINERS } // CG is short for component group

   boolean FineGrainedNegotiation()
   {
       if(ConformanceConsistencyCheck()== false) return false;
       selectedQoSProfiles ← Empty;
       BOUND ← user's QoS requirement;
       for(int i=0; i<components[0].profiles.size(); i++) {
              Quality-Point ← provided QoS contract of the i^th profile of components[0];
              if((h(Quality-Point) ≥ BOUND) && (Quality_Point fulfils user's constraint))
              {
                     int result = FindAppropriateProfiles();
                     if(result == SUCCESSFUL) {
                            selectedQoSProfiles ← the newly selected QoS-Profiles;
                            BOUND ← the newly computed BOUND;
                            continue; // try further for a better solution
                     }
                     else if(result == NO_RESOURCE) {
                            break; // this is a termination condition
25                          }
                    }
            }
            if(selectedQoSProfiles is Empty) return false else return true;
    }
30
    int FindAppropriateProfiles()
    {
            FindConformantProfiles(CG.ON_CLIENT);
            if(CheckResourceConstraints(CG.ON_CLIENT)) {
35                  FindConformantProfiles(CG.ACROSS_CONTAINERS\CG.ON_CLIENT);
                    if(CheckResourceConstraints(CG.ACROSS_CONTAINERS)) {
                            FindConformantProfiles(CG.ON_SERVER\CG.ACROSS_CONTAINERS);
                            if(CheckResourceConstraints(CG.ON_SERVER))
                                    return SUCCESSFUL;
40                          else return NO_RESOURCE;
                    }
                    else return NO_RESOURCE;
            }
            else return NO_RESOURCE;
45  }
```

The following is a short description of the variables and functions used in Algorithm 3.2.

Variables or Functions used in Algorithm 3.2	**Description**
`selectedQoSProfiles` (Lines 4, 11, 19, 28)	A list data structure that stores the QoS-Profiles of all components, which fulfil all the constraints
`CG`, `CG.ON_CLIENT`, `CG.ACROSS_CONTAINERS`, `CG.ON_SERVER` (Lines 6, 33-38)	`CG.ON_CLIENT`, `CG.ON_SERVER`, and `CG.ACROSS_CONTAINERS` refer to components on the client, components on the server, and components connected across containers respectively (Figure 3.11). Two functions in this algorithm take a type of CG as input argument.
`ConformanceConsistencyCheck()` (Line 10)	Performs consistency check (i.e. for the conformance constraint) to every connected pair of components. In doing so, it would remove QoS-Profiles from a component if no conformant profile were specified for the connected component.
`BOUND` (Lines 12, 15, 20)	The best `Quality-Point` so far obtained. `BOUND` is initialized to the user's QoS requirement as the algorithm aims at finding a solution that just meets the requirement or even a better one.
`Components[0]` (Line 13)	the front-end component
`components[0].profiles.size()` (Line 13)	The total number of QoS-Profiles specified for `Components[0]`. QoS-Profiles associated with each component are stored in `profiles`, which is a list data structure contained within each component.
`Quality-Point` (Lines 14, 15)	Represents the quality points in terms of the considered properties (e.g. (20fps, 352x288)). This information is

Variables or Functions used in Algorithm 3.2	**Description**
	extracted either from the *provides* or *uses* part of a QoS-Profile.
`h` (Line 15)	The heuristic function that is explained in subsection 3.4.3. `h` is computed only at the beginning of each iteration and `h(Quality-Point) = Quality-Point`
`FindConformantProfiles()` (Lines 33, 35, 37)	Finds QoS-profiles, which are conformant to one another for all the components specified in the input argument. At each iteration this function improves the solution by one step based on the specified QoS-profiles.
`CheckResourceConstraint()` (Lines 34, 36, 38)	Checks whether there are enough resources for the current selection.

A component may belong to two groups in `CG (ComponentGroup)`. For example, a component deployed on the client container and that also communicates across containers belongs to `ON_CLIENT` and `ACROSS_CONTAINERS`. The notation "`\`" in Algorithm 3.2 is read as "less".

3.4.3.2 Algorithm Termination and Complexity

`FineGrainedNegotiation()` in Algorithm 3.2 iteratively searches for a "better" solution. It starts by selecting appropriate profiles from the first component, which is the front-end component, until the last one following the designated component ordering. At each iteration, the conformance checking between two connected components (C_i, C_j) where C_i is the parent of C_j is performed as follows. Starting from (C_1, C_2) where C_1 is the front-end component, a new QoS-Profile is chosen for C_1 at the beginning of the iteration each time selecting a QoS-Profile with a better quality (Lines 13-14). For selecting a profile for C_2, not all of its profiles have to be checked for conformance consistency with that of the already chosen profile of C_1. The checking, nevertheless, is performed starting from the profile of C_2 selected in the previous iteration and moving upwards in the array of profiles from lower to higher quality. This step is repeated for (C_2, C_3) and in general for (C_i, C_j). That is, a profile for C_j is chosen using a similar procedure provided that a profile for C_i has been selected in the previous steps. The termination of the algorithm is guaranteed because of the fact that the network of components constitutes only a tree and the number of QoS-Profiles of each component is finite.

The algorithm could terminate by finding a solution or without finding one. If the list `selectedQoSProfiles` (Lines 4,11,19,28) is non-empty after the execution of `FineGrainedNegotiation()`, then a solution has been found. In this case, `selectedQoSProfiles` contains QoS-Profiles of each component that satisfy user's, conformance, and resource constraints for chosen values of coarse-grained properties. Moreover, the solution obtained gives at least the utility that is required by the user. If `selectedQoSProfiles` remains empty, then there exists no solution that satisfies all the three constraints, which is an over-constrained situation.

In the algorithm design, we assumed that the collaborating components form only a tree. `ConformanceConsistencyCheck()`(Line 10) performs a consistency check to connected components - (C_i, C_j) where C_i is the parent of C_j. The conformance consistency checking runs from `j=n` down to `2` assuming the components are designated as C_1, C_2, …, C_n. This step removes QoS-profiles from the domain of C_i for which no conformant profiles have been specified in C_j. The purpose `ConformanceConsistencyCheck()` is to achieve node- and arc-consistency in terms of the user's and conformance constraints and thus enables the remaining part of the algorithm to run with no backtracking. It has been proved that a search in a binary CSP is backtrack-free if the constraint graph of a problem forms a tree and both node- and arc-consistency are achieved in the problem [Freuder, 1982]. Assuming the total number of components is **n** and the number of QoS-Profiles specified for each component is **d**, the complexity of `ConformanceConsistencyCheck ()` is O(nd^2). As a solution can be found without backtracking, the complexity of finding the first solution in `FineGrainedNegotiation()` is O(nd^2+nd) = O(nd^2). The solution can then be improved to the next higher utility with O(nd).

3.4.4 Combining Coarse-grained and Fine-grained negotiations

As explained earlier, the whole negotiation is performed in two phases: coarse-grained and fine-grained. The following algorithm combines these two phases.

Algorithm 3.3: Negotiation algorithm for components deployed in client and server containers

```
void Negotiation()
{
        CoarseGrainedNegotiation(); // Algorithm 3.1

        if( there is agreement in at least one value on coarse-grained properties ) {
                for( each agreed value in the coarse-grained negotiation) {
                        FineGrainedNegotiation(); // Algorithm 3.2
                }

                Compare the utility of the solutions obtained for the different coarse-
                grained property values;

                Choose the solution that gives the highest utility. If two or more
                solutions have the same highest utility, choose the one that consumes less
                client resource pertaining to the assumed negotiation goal;
        }
}
```

Multiple values could be agreed upon during the coarse-grained negotiation. A good example is the use of different compression algorithms whose purpose is to achieve resource trading between the network bandwidth and the client or server CPU time. This resource trading is one of the reasons for change of quality or utility in the solutions obtained in the fine-grained negotiation.

The choice between multiple solutions is made based on a negotiation goal. Users and service providers usually have conflicting interests. Each wants to maximize its interest. As we are dealing in this Section with a *single client – single server* negotiation, we take the user's satisfaction as the goal that drives the negotiation. In the *multiple clients – single server* scenario (subsection 4.1), we show how the service provider's interest can be incorporated. In general, however, the negotiation goal itself may be the subject of prior negotiation. In this thesis, we limit the negotiations only on the NFPs by assuming the goals have already been agreed upon.

3.4.5 Dealing with Over-constrained Situation

In Algorithm 3.2, the QoS Contract negotiation has been modeled with hard constraints. The algorithm is always expected to give a solution that meets all the constraints. No solution will be provided (i.e. `selectedQoSProfiles` in Algorithm 3.2 remains empty), if at least one constraint is violated. This is inflexible as far as the user is concerned, who may prefer to get a solution, though a degraded one. For instance, if a user requires a response time to a certain operation be less than 5 sec and no configuration can be found that fulfills this demand, then the user could opt for a response of, say, 6 sec rater than no solution. In some instances, it may be difficult to find a solution that meets all constraints whatever the user's requirement may be. Under all these situations, the CSP is said to be *over-constrained*. We discuss next how to generalize Algorithm 3.2 to handle some of the over-constrained cases.

One of the approaches used to deal with an over-constrained CSP is to extend the CSP framework into what is called *Partial Constraint Satisfaction Problem* (PCSP) [Freuder, 1989]. In [Freuder, 1989] maximal constraint satisfaction algorithms, which seek a solution that satisfies as many constraints as possible, have been given. The algorithms compare potential partial solutions based on the number of constraints violated and choose the one that results in minimum number of constraint violations. But these are some general metrics. The particular nature of the problem at hand should dictate the type of strategy to be used in finding the partial solution.

In our case, we have three different types of constraints: user, conformance, and resource. When considering multiple QoS-dimensions, the conformance constraint applies to each dimension. Violating a resource constraint cannot be compared with that of conformance constraint violation and hence simply counting the violated constraints is not applicable to our problem. In search of a partial solution, which is achieved by weakening the problem, we have to examine how each violation affects the outcome and try to identify the constraints that can be tolerated for violation. Allowing either resource or conformance constraints to be violated would make it difficult to predict the QoS property of the whole application. We currently follow the strategy where the user's constraint is systematically violated to reach to a solution. This in effect means, the user's QoS requirement is relaxed in such a way that the QoS property of the application is a degraded one in order to obtain a solution that satisfies the conformance and resource constraints.

Algorithm 3.2 terminates without finding a solution under the following cases:

(i) User's constraint is always violated even though a configuration that meets the resource and conformance constraints exists.
(ii) Resource constraints cannot be met although a configuration that meets the user's constraint exists.
(iii) Conformant constraints cannot be met when trying to find a solution that satisfies the user's constraint.

In the above cases, (i) and (iii) are similar except that we treat the conformance of the user's requirement and the provided QoS contract of the front-end component different from the conformance between required- and provided-QoS contract of any two interacting components of the application. User's constraint relaxation is applied in finding a partial solution for the three cases mentioned above.

For the non over-constrained problem, the algorithm in Algorithm 3.2 improves a solution step-by-step by going one level up in the available QoS-profiles. The relaxation step in the over-constrained case uses a similar approach to get a "better" solution from the available partial solutions. It successively goes one step down on the list of QoS-profiles[3] of the front-end component to degrade the user's QoS requirements. Algorithm 3.2 is then applied to the weakened problem. If this step also terminates without producing a solution, the user's QoS requirement is relaxed further and the above process is repeated. This step runs until a solution is obtained or the user's requirement cannot be further relaxed. The algorithm that incorporates these steps is shown in Algorithm 3.4.

The assumptions we have made concerning how user's requirements and preferences are captured are as follows. A user can define the relative weight of the different QoS properties; and a user can define the minimum quality of each property below which is unacceptable. A more robust user interface can be designed but this is beyond the scope of this thesis. It should however be emphasized that it is essential not to make the user interface complex when trying to make it more robust. One possible user input mechanism that can be incorporated in Algorithm 3.4 in the future is the prioritization of a user towards the QoS properties. Out of the `n` QoS-dimensions considered in the application, the user may be interested only in `m` (where `m < n`) properties. This might imply that the negotiation can continue even if there are contract violations in the `n - m` QoS properties.

Algorithm 3.4: A fine-grained negotiation algorithm that performs user's QoS requirement relaxation during an over-constrained situation

```
/*
INPUT:
1. The same inputs as Algorithm 3.2.
```

[3] The QoS-Profiles are sorted from low to high quality based on user's preference

<u>OUTPUT</u>:
A QoS-Profile for each component that satisfy user's, conformance, and resource constraints. The user's constraint may be relaxed systematically in case of an over-constrained situation.
**/*

```
   List<QoSStatement> userProfile;

   boolean RelaxedFineGrainedNegotiation()
   {
5    userProfile ← user's QoS requirement;
     if(FineGrainedNegotiation() == false) { // Algorithm 3.2
          while(RelaxUserQoSRequirement()) {
               if(FineGrainedNegotiation())// Algorithm 3.2
                    return true;
10        }
          return false;
     }
     return true;
   }
15
   boolean RelaxUserQoSRequirement()
   {
     List<QoSStatement> frontEndProfile;
     for(int i=0; i<components[0].profiles.size(); i++) {
20     frontEndProfile ← provided-QoS contract of the ith profile of components[0];
       if(IsConformant(userProfile, frontEndProfile)) {
          if(i > 0) { // relaxation can be performed
            userProfile ← provided-QoS contract of the (i-1)th profile of components[0];
            return true;
25        }
          else return false; // no more relaxation possible
       }
     }
     return false;
30 }
```

The following is a short description of the variables and functions used in Algorithm 3.4.

Variables or Functions used in Algorithm 3.4	Description
`components[0]` (Lines 19, 20, 23)	The front-end component.
`components[0].profiles.size()` (Line 19)	The total number of QoS-Profiles specified for `Components[0]`. QoS-Profiles associated to each component are stored in `profiles`, which is a list data structure contained within each component.
`frontEndProfile` (Lines 18, 20, 21)	Stores the provided-QoS contract of the front-end component of a particular QoS-Profile.
`userProfile`	This global variable is initialized first with the user's QoS

Variables or Functions used in Algorithm 3.4	Description
(Lines 1, 5, 21, 23)	requirement, which may be specified for one or more QoS properties. Whenever `RelaxUserQoSRequirement()` is invoked, it is updated with a *QoS Statement* that is one step inferior to its previous value.
`IsConformant()` (Line 21)	The function takes two arguments of type *QoS Statement* (this describes a QoS constraint to one or more QoS properties). It returns true if the constraint in the second argument conforms to that of the first.

`FineGrainedNegotiation()` can be replaced by `RelaxedFineGrainedNegotiation()` in Algorithm 3.3 to take advantage of over-constrained considerations.

3.4.6 Centralized, Distributed, and Hybrid Solutions

The algorithms listed in Algorithm 3.1 and Algorithm 3.2 are centralized algorithms. In general, we can identify three approaches for performing the coarse-grained and fine-grained negotiations. These are:

i.) *Centralized Solution*: Only one of the containers makes the selection of QoS-profiles.
ii.) *Distributed Solution*: Both the client and server containers are responsible for the selection of appropriate QoS-profiles.
iii.) *Hybrid Solution*: Both the client and server containers are responsible for the selection of appropriate QoS-profiles. It differs from the pure distributed case as it tries to harness local or concurrent executions after determining bottleneck resources with a centralized approach.

In the centralized solution, the responsible container must have access to the configuration information (meta-data) of all components deployed on the client and server containers. The distributed solution doesn't require this as each container is responsible for the selection of the components' profiles it hosts and communicates with the other container to resolve constraints that concern both of them.

Owing to the nature of our problem, a pure distributed solution results in large inter-container communication overhead and hence we do not consider it as an option. To elaborate on this let us consider the example in Figure 3.12. Assume the client container hosts 2 components C_1 and C_2 and the server container hosts C_3 and C_4. Assume C_2 interacts with C_3 across containers. In the pure distributed solution, the client container finds appropriate profiles for C_1 and C_2. So does the server container for C_3 and C_4. The client must communicate with the server about the selected profile of C_2 as this profile has to match the selection for C_3 and the resource demand of the interaction between C_2 and C_3 must be met by the available network bandwidth. Assume the server finds out that the selected profiles of C_2 and C_3 do not match. In this case, either the client or server has to change their previous selection. Suppose the server changes its selection to match the client's profile and communicates this to the client. It may happen that no matching profiles exist on the client. Even if a matching profile for C_2 can be found, no profile for C_1 might exist that matches the new selection for C_2. In a nutshell, there is no guarantee on the number of communications required to reach an agreement.

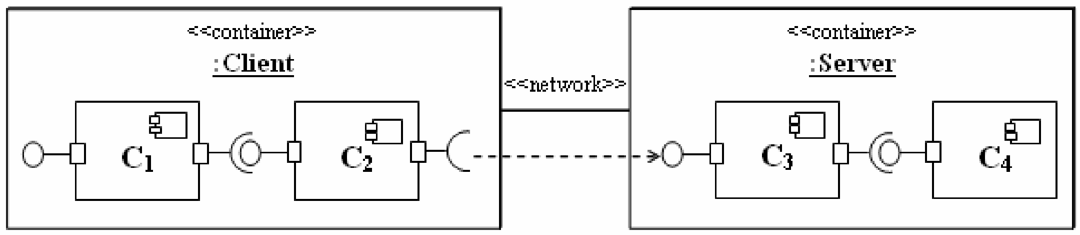


Figure 3.12: Example application in which components are deployed in two containers

In the hybrid solution, the bottleneck resource is first isolated with a centralized approach. Thereafter further negotiations could continue on local containers. The main idea is that some NFPs might be specified only for local components and hence negotiating on these properties only locally helps in speeding up the negotiation process. For example, let's assume that in a video streaming application frame rate and resolution properties are specified for both *VideoPlayer* and *VideoServer*, which are deployed in client and server containers respectively. However, in addition to these NFPs, smoothness property might be specified for *VideoPlayer*. In Algorithm 3.2, all the fine-grained properties are negotiated atomically. That implies for the aforementioned example, Algorithm 3.2 negotiates frame rate, resolution, and smoothness at the same time. The other possibility is to make negotiation orderings on the fine-grained properties such that negotiation is performed first on frame rate and resolution and then on smoothness. This ordering doesn't happen arbitrarily. One notable difference between the three properties is that while frame rate and resolution are specified for components deployed in both containers, smoothness is specified for a component deployed only on one of the containers.

We can think of the hybrid solution as consisting of the following two steps: (i) Centralized solution applied on part of the fine-grained properties; and (ii) localized negotiation applied to client and/or server containers alone. The hybrid solution could speed up the negotiation process under the following conditions: (i) the bottleneck resource is isolated faster (if for e.g. the bottleneck is the network bandwidth), and (ii) the second negotiation step is applied only locally, thereby to a reduced search space.

In order to make comparisons between the centralized and hybrid solutions, we divide the time of negotiations into two phases. Phase 1 is the initial negotiation process where contracts are to be established for the first time. Phase 2 is for re-negotiations that take place after contracts have already been established. Negotiations in Phase 2 are performed to adjust the occurrence of contract violations or to achieve better contracts in case more resources are available.

When considering initial negotiations, both solutions can be advantageous in different aspects. Here, we use the number of inter-container communication and level of concurrency/computation as measures for the comparison. For the centralized solution, inter-container communication is required when:

(i) the QoS-Profiles of interacting components and local resource conditions are exchanged just before the negotiation, and
(ii) the established contracts are communicated finally.

For the hybrid solution, inter-container communication is required when:

(i) the QoS-Profiles of interacting components and local resource conditions are exchanged just before the negotiation (this is used to determine the bottleneck resource),
(ii) the two containers agree to do local negotiations to refine the solution, and
(iii) each container confirms the completion of the independent negotiations and exchanges relevant contracts.

Therefore, as far as minimizing the inter-container communication is concerned, the centralized solution is better than the hybrid solution. On the other hand, the hybrid solution can take advantage of the multiple step negotiation that includes local negotiation and find a solution faster, especially if the required level of computation is high.

When we see this from a different angle, the centralized solution relieves the burden of negotiation for one of the containers (of course at the expense of the other container!) and it could also be simpler for implementation. If there is a shortage of resources in one of the containers while there is abundance on the other, the centralized solution can be more efficient.

The other difference between the two solutions comes in Phase 2, i.e. for negotiations after contracts have been established. As contracts can be violated, monitoring of contracts is constantly performed. For instance, if a contract is established between a *VideoPlayer* and a *VideoServer* component, the contract contains such terms as the frame rate the *VideoPlayer* expects from the received stream and the frame rate the *VideoServer* provides. The contract monitoring checks whether the received stream is not in violation of the contract. When there is a contract violation, re-negotiations are usually performed to gracefully adapt the contract. In a different spectrum, once contracts are established, resources are monitored to see if more resources become available. When this happens, contracts can be upgraded for more quality output through re-negotiations.

In the centralized solution, communication between containers is required to exchange the following information:

(i) any contract violation detected, and
(ii) the monitoring of available resources (which is a periodic activity).

This results in increased communication between containers, thus taking up some of the useful network bandwidth.

However, for the hybrid solution, no such communication is required so long as the solution can be provided locally. The communication would be required if the responsible container cannot perform the negotiations on its own and needs assistance from the other container. Suppose a contract between components in the client is violated because of less available resource in the client node, then other contracts in the server or for components connected across containers aren't affected. Thus the renegotiation can be done only for the contracts in the client. We apply here the idea of *contract re-negotiation dependency*. This dependency captures the effect of violation of one contract on another.

According to [Yokoo et al, 1998], a negative point for a centralized solution is (i) there can be a cost of translating one's knowledge into an exchangeable format and (ii) with some problems, gathering all information to one agent is undesirable or impossible for security/privacy reasons. But, in our case we assumed these as less critical due to the nature of our problem. In the first place, for our problem, privacy issues are not that relevant. Data to be exchanged are QoS-Profiles of collaborating components, and possibly the processor and other pertinent information of the client node, which may be needed to convert the resource demand of components that have been measured in a different environment. We compared the various solutions based on concurrency level and number of inter-container communications.

We conclude this section by summarizing the ultimate approach we are following. Owing to the nature of our problem, which is tightly coupled, we don't consider a pure distributed solution. The main reason is that it results in an undetermined number of inter-container communication, which usually influences the efficiency of the negotiation process. The choice between centralized and hybrid solutions is also not clear-cut. It depends on the number of components and QoS-Profiles involved and the environment conditions. The decisions as to whether to apply the centralized or the hybrid approach can be made just before the start of the negotiation based on some input conditions. The particular choice depends on the cost of inter-container communication and the required level of computation. If the former is expensive, a centralized solution would be better. Nevertheless, if the problem at hand requires a lot of computation and the environment offers faster inter-container communication, the hybrid solution would be the more efficient approach. During run-time re-negotiations, the hybrid solution is the more preferable choice as it avoids unnecessary overheads when it is applicable to make re-configurations locally.

3.5 Example and Validation

The contract negotiation algorithms and protocols that we proposed can be validated by considering various scenarios in the distributed componentized applications (e.g. video streaming and stock quote) that we studied. We generally assume that component developers specify QoS-Profiles through: (i) extensive measurements, or (ii) some analytical means, or (iii) the combination of the two.

In order to validate the proposed approach, we have to produce QoS-Profiles of components. QoS-Profiles contain offered QoS contract, Required QoS contract and resource demand at a component level. In this section, we analyze a video streaming application and examine different scenarios in order to validate our contract negotiation approach for a *single client – single server* setting.

3.5.1 Video Streaming

In this application scenario, the following assumptions have been made:

- A high quality media file is pre-encoded and available for streaming to the clients. The media file is assumed to have a constant bit rate.
- Component's QoS-Profiles are provided by the component developer in a kind of tabular form.

Components that are identified to exist in the Video Streaming application are: *VideoServer* and *VideoPlayer*. The network and the container that exists between the interacting components can be modeled by a connector abstraction. A component assembly diagram depicted in Figure 3.13 shows the NFPs considered in the video streaming application. For *VideoServer*, we assume packets are equally spaced and transmitted in a regulated fashion (in equal space). This removes jitter from the output of *VideoServer*. In actual case, where many video streams are transmitted from a video server, some jitter may be introduced.

VideoServer

This component reads pre-encoded media files from the server or directly reads from a capturing device and streams video and audio to clients. For a given input source (e.g. mpeg1 at 30fps and 640X480), *VideoServer* is capable of streaming at different frame rate, resolution, and coding. Moreover, this component may stream the media in different protocols (e.g. RTP/UDP, UDP, or a custom transport protocol). The component's coarse-grained and fine-grained properties are specified in Figure 3.14 to Figure 3.16. *VideoServer* is abstracted with multiple implementations, which, as is usually the case, differ on the supported coarse-grained properties (Figure 3.14 & Figure 3.15). Within each implementation, a number of different QoS-Profiles can be defined (Figure 3.16).

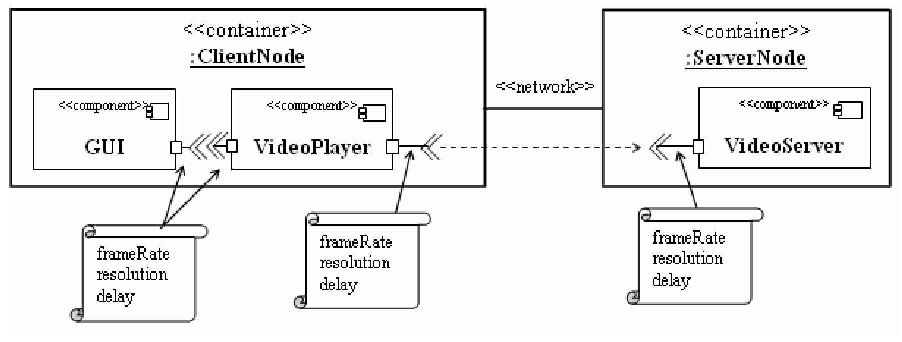


Figure 3.13: A component diagram showing considered NFPs in a componentized video streaming application

	Protocol
1.	RTP/UDP
2.	UDP
3.	TCP

Figure 3.14: *VideoServer's* protocol properties

	Coding
1.	Mpeg
2.	Mpeg4
3.	h264
4.	h263

Figure 3.15: *VideoServer's* coding properties

Profile Nr.	`Provides ICompVideo (frameRate, resolution)`	Resource (bandwidth in Kbps)
1.	30 s^{-1}, 352X288	450
2.	30 s^{-1}, 176X144	250
3.	10 s^{-1}, 352X288	200
4.	10 s^{-1}, 176X144	150

Figure 3.16: The QoS-Profile of VideoServer for mpeg4 coding and RTP/UDP protocol

The QoS-Profiles of *VideoServer* in Figure 3.16 is shown for specific value of coarse-grained properties. Similar specifications should be given for each protocol and coding properties. In this case, we expect to have a maximum of 12 [3 (for protocol) × 4 (for coding)] specifications similar to the one in Figure 3.16. In addition to the coarse grained properties, these specifications would differ in their resource demand. In general, the values of fine-grained properties defined for each coarse-grained property may be different. For instance, not all coding algorithms support similar resolutions in video streaming.

VideoPlayer

The *VideoPlayer* component reads media data streamed by *VideoServer* for playback at the client node. It then passes the uncompressed audio and video data for rendering it over screen and microphone. Two stream interfaces (a sink and a source) are identified for this component. This component, like *VideoServer*, is also capable of receiving streams at different frame rate, resolution, coding, and protocol.

The *VideoPlayer* starts the playback only after the buffer associated with it is filled with the streamed data. During playback, the buffer can get completely emptied (buffer starvation). In this case, the playback is halted until there is complete re-buffering. In our approach, before this happens, a contract violation is encountered. To cope with this violation, a contract re-negotiation must be initiated, which possibly results in a degraded output.

Similar to the properties of *VideoServer*, the various properties of *VideoPlayer* are depicted in Figure 3.17 to Figure 3.19.

	Protocol
1.	RTP/UDP
2.	UDP
3.	TCP

Figure 3.17: *VideoPlayer's* protocol properties

	Coding
1.	Mpeg
2.	mpeg4
3.	h264
4.	h263

Figure 3.18: VideoPlayer's coding properties

Profile Nr.	`uses ICompVideo` `(frameRate, resolution)`	`provides ICompVideo` `(frameRate, resolution)`	Resource (bandwidth in Kbps, CPU time in, memory)
1.	30 s^{-1}, 352X288	30 s^{-1}, 352X288	
2.	30 s^{-1}, 176X144	30 s^{-1}, 176X144	
3.	10 s^{-1}, 352X288	10 s^{-1}, 352X288	
4.	10 s^{-1}, 176X144	10 s^{-1}, 176X144	

Figure 3.19: The QoS-Profile of *VideoPlayer* for mpeg4 coding and RTP/UDP protocol

We have not included jitter in our analysis. We could have considered jitter by including it as a property in the input of the *VideoPlayer*. For minimizing or removing this jitter from the output stream, the component's memory (buffer) requirement is high. This conforms to the fact that the buffer in the player is used to remove the network jitter. To decrease the starting delay of *VideoPlayer*, the memory requirement should be minimized but this happens at the expense of increased jitter in the output. Depending on the particular application (user's requirement), users could prefer a small starting delay at the expense of jitter or otherwise.

Connector

A connector is an abstraction to model the network and containers that exist between interacting components. One way of finding the properties of a connector is through measurements performed off-line at light load conditions in the network and the nodes. A second approach is to obtain the property through run-time probing and prediction just before the contract negotiation begins.

The connector is assumed to have the properties: *end-to-end delay* and *packet-loss rate*. Each property could depend on a number of parameters. For instance, end-to-end delay depends on:

- the data rate of the video stream, and
- the load conditions of the network - the higher the load the higher is the delay due to an increase in the queuing delay at the routers.

The packet-loss rate depends on:

- The load conditions of the network – the higher the load the higher the loss rate.

One can see the dependence of the end-to-end delay to the throughput of the video stream from the requirement of a video conferencing application. For example for a video streaming at 30 frames/s, each frame must be delivered, decoded, and displayed in less than 33ms. For a video of 10 frames/s, this time is 100ms.

How to measure/estimate the connector properties?

Just before the contract negotiation begins, the end-to-end delay (either in the form of RTT (round-trip time) or one-way delay) is measured. The same setup to measure the end-to-end available bandwidth can be used.

Theoretically speaking, an end-to-end delay has a fixed and variable component. When sending a packet from some source to some sink, the delays encountered in the path are classified as: serialization delays (fixed); propagation delays (fixed); and queuing delays (variable) [Prasad et al, 2003]. The end-to-end delay is the sum of serialization, propagation, and queuing delays.

The *serialization delay* of a packet of size L at a link of transmission rate C is the time needed to transmit the packet on the link, equal to L/C. The *propagation delay* of a packet at a link is the time it takes for each bit of the packet to traverse the link (this value depends on the length of the path), and is independent of the packet size. Finally, *queuing delays* can occur in the buffers of routers or switches when there is contention at the input or output ports of these devices.

If we assume there is no congestion in the network, the end-to-end delay measurement doesn't depend on the current load of the network (and also on the data rate as the end-to-end delay is defined on packets and not on frames) and it depends on the packet size of the probe that is used in the measurement and on the length of the path (number of hops). To have a realistic value either the packet size used in the probing is made equal to the packet size of the video packets to be streamed or some kind of estimation should be made. Usually, applications use some default maximum value for the size of the packets. Hence, as a starting point, even without measurement, we can calculate the end-to-end delay by assuming no congestion and from the given values of the packet size and link capacities of each hop.

3.5.2 Demonstrating the Proposed Approach

We first conducted an experiment to specify *VideoPlayer* and *VideoServer*. The *VideoPlayer* component was developed using the JMF [Sun Microsystems, 2001] framework, and *VideoServer* abstracts the video media file, where the streams are pull data sources. The original media file (a 10 minute video clip) was encoded into many files with different frame rate, resolution, and encoding algorithm. This was achieved with the tools: JMF Studio and *Framerate Converter* (evaluation version). When converting the frame rate, the parameters used in *Framerate Converter* were: Interpolation Method – Linear (fast); Interpolation range – 50; and Resample method – Linear (good).

The PC on which *VideoPlayer* ran and the measurement was conducted had a configuration of: AMD Athlon(tm) XP 1600+, 1GB RAM, Windows 2000 Professional, and Java 2 version 5. The PC was connected through a 100 Mbps LAN to the PC that hosted *VideoServer*. The purpose of the whole measurement was not ultimate accuracy but to obtain approximate data, which would enable us to validate our contract negotiation protocol and algorithms. Windows Performance Monitor was used to measure the resource usage of the components. Average bandwidth and CPU percentage time were considered. No anomalous behavior was observed during the experiment.

The measured QoS-Profiles of *VideoPlayer*, with coarse-grained properties `coding=mp42`, `protocol=UDP`, are shown in Table 3.4. Similarly, the measured data for *VideoServer* for the same coding and protocol properties are shown in Table 3.5. As the measurements were taken under light load conditions, it is assumed that the network bandwidth requirement of *VideoServer* was taken to be the same as that for *VideoPlayer*. Moreover, the measured CPU requirements of *VideoServer* are too small (in the range of 0.1%). Hence, the CPU time has been left out from Table 3.5 as it is too small to have influence in our validation.

VideoPlayer			
	uses ICompVideo (resolution, frame rate in s^{-1})	provides ICompVideo (resolution, frame rate in s^{-1})	resource (CPU in %, bandwidth in Kbps, memory KB)
1.	352x288, 30	352x288, 30	13.23, 2165, 31.6
2.	352x288, 15	352x288, 15	8.90, 2146, 30
3.	352x288, 10	352x288, 10	8.91, 2076, 30.2
4.	352x288, 5	352x288, 5	5.90, 1852, 29.5
5.	352x288, 1	352x288, 1	2.31, 1644, 29.2
6.	176x144, 30	176x144, 30	0.97, 321, 25.6
7.	176x144, 15	176x144, 15	0.90, 252, 18.8
8.	176x144, 10	176x144, 10	0.62, 208, 24.4
9.	176x144, 5	176x144, 5	0.39, 135, 24.0
10.	176x144, 1	176x144, 1	0.11, 34, 24.4
11.	128x96, 30	128x96, 30	0.51, 152, 26.1
12.	128x96, 15	128x96, 15	0.38, 120, 25.1
13.	128x96, 10	128x96, 10	0.64, 108, 24.5
14.	128x96, 5	128x96, 5	0.19, 70, 24.2
15.	128x96, 1	128x96, 1	0.04, 24, 22.9

Table 3.4: QoS-Profiles of VideoPlayer for a UDP protocol and mp42 encoding

VideoServer	
provides ICompVideo (resolution, frame rate in s^{-1})	resource (bandwidth in Kbps)
352x288, 30	2165
352x288, 15	2146
352x288, 10	2076
352x288, 5	1852
352x288, 1	1644
176x144, 30	321
176x144, 15	252
176x144, 10	208
176x144, 5	135
176x144, 1	34
128x96, 30	152
128x96, 15	120
128x96, 10	108
128x96, 5	70
128x96, 1	24

Table 3.5: QoS-Profiles of VideoServer for a UDP protocol and mp42 encodingTable 3.4b

Table 3.6 and Table 3.7 show similar measurements with the h263 coding. One can observe from the measurements for h263 and mp42 coding that for the same values in frame rate and resolution the h263 coding demands more CPU time and less network bandwidth as compared to the mp42 coding.

VideoPlayer			
	uses ICompVideo (resolution, frame rate in s^{-1})	provides ICompVideo (resolution, frame rate in s^{-1})	resource (CPU in %, bandwidth in Kbps, memory KB)
1.	352x288, 30	352x288, 30	26.75, 2220, 30.3
2.	352x288, 15	352x288, 15	14.36, 1906, 29.8
3.	352x288, 10	352x288, 10	9.86, 1546, 29.3
4.	352x288, 5	352x288, 5	4.64, 1572, 28.8
5.	352x288, 1	352x288, 1	1.84, 1569, 28.8
6.	176x144, 30	176x144, 30	2.27, 164, 32.2
7.	176x144, 15	176x144, 15	1.04, 86, 25.7
8.	176x144, 10	176x144, 10	1.02, 55, 25.3
9.	176x144, 5	176x144, 5	0.38, 29, 25.1
10.	176x144, 1	176x144, 1	0.20, 7.8, 24.6
11.	128x96, 30	128x96, 30	0.68, 85, 25.8
12.	128x96, 15	128x96, 15	0.48, 42, 25.1

VideoServer	
provides ICompVideo (resolution, frame rate in s^{-1})	resource (bandwidth in Kbps)
352x288, 30	2220
352x288, 15	1906
352x288, 10	1546
352x288, 5	1572
352x288, 1	1569
176x144, 30	164
176x144, 15	86
176x144, 10	55
176x144, 5	29
176x144, 1	7.8
128x96, 30	85
128x96, 15	42

VideoPlayer			
	uses ICompVideo (resolution, frame rate in s^{-1})	provides ICompVideo (resolution, frame rate in s^{-1})	resource (CPU in %, bandwidth in Kbps, memory KB)
13.	128x96, 10	128x96, 10	0.42, 28.8, 25.1
14.	128x96, 5	128x96, 5	0.25, 16, 24.5
15.	128x96, 1	128x96, 1	0.16, 4, 24.0

Table 3.6: QoS-Profiles of VideoPlayer for a UDP protocol and h263 encoding

VideoServer	
provides ICompVideo (resolution, frame rate in s^{-1})	resource (bandwidth in Kbps)
128x96, 10	28.8
128x96, 5	16
128x96, 1	4

Table 3.7: QoS-Profiles of VideoServer for a UDP protocol and h263 encodingTable 3.4b

Let the coarse-grained properties specified for both *VideoPlayer* and *VideoServer* be as listed in Table 3.8 & Table 3.9. Furthermore, the values of these properties are assumed to be ordered as presented there. For each component, QoS profiles that specify different frame rates and resolution are expected to be specified for all combinations of coarse-grained property values.

VideoPlayer				
	Protocol			Coding
1.	RTP/UDP		1.	mp42
2.	UDP		2.	mpg4
3.	TCP		3.	h263

Table 3.8: Protocol and coding properties of *VideoPlayer*

VideoServer				
	Protocol			Coding
1.	RTP/UDP		1.	mp43
2.	UDP		2.	mp42
3.	TCP		3.	mpg4
			4.	h264

Table 3.9: Protocol and coding properties of *VideoServer*

The containers (nodes) at the client and server side likewise have supported properties as shown in Table 3.10 and Table 3.11.

Client container	
Communication model	Protocol
Best-effort	RTP/UDP
	UDP
	TCP
	HTTP

Table 3.10: Communication model and protocol properties of client container

Server container	
Communication model	Protocol
Guaranteed	RTP/UDP
Best-effort	UDP
	TCP
	HTTP
	FTP

Table 3.11: Communication model and protocol properties of server container

The contract negotiation is performed in three steps, at each step selecting appropriate values for the specified properties. Steps II. and III. are combined in Algorithm 3.3.

I. Container-Container Negotiation (subsection 3.4.2)
II. Coarse-grained Negotiations between Components (Algorithm 3.1)
III. Fine-grained Negotiations between Components (Algorithm 3.2)

I. Container-Container Negotiation (subsection 3.4.2)

The properties specified for the two containers are communication model and protocol (Table 3.10 and Table 3.11). The negotiation at this step could affect the result of subsequent negotiations. For instance, if the two containers cannot agree on the RTP/UDP protocol, the next negotiation step wouldn't select this protocol even though both *VideoPlayer* and *VideoServer* support RTP/UDP.

This negotiation results in the selection of:

- one communication model type, i.e., best-effort, and
- four protocols: RTP/UDP, UDP, TCP, and HTTP.

II. Coarse-grained Negotiations between Components (Algorithm 3.1)

We refer to Table 3.8 and Table 3.9 for this negotiation. The algorithm starts from the highest preference of the particular property of the client component (*VideoPlayer* in this case), and searches for a value in the corresponding property of the server component that is *conformant* to the chosen property. The negotiation culminates in the selection of:

- coding: mp42 and mpg4
- protocol: RTP/UDP, UDP, and TCP.

In both properties, multiple values have been selected and the final decision is made after the fine-grained negotiation is performed for each combination of coding and protocol types. Supposing the protocol RTP/UDP and UDP were not selected during the container-container negotiation, the choices of these protocols would have been dropped at this step.

III. Fine-grained Negotiations between Components (Algorithm 3.2)

This is the final stage in the contract negotiation process. For the sake of clarity, we have divided the fine-grained negotiation into two stages: (i) a pre-processing step, and (ii) the main step. During the pre-processing step, necessary actions must be taken to get the input data and properly format them as required by the main step.

Pre-processing Step:

1. Get user's QoS requirement and preference.

   Requirement and preference:

frame rate > 12fps, resolution = 176x144
frame rate is preferred to resolution.

2. Get configuration of all components.

   Some configurations have already been given in Table 3.4 to Table 3.7. We expect to have more of similar data specified for every agreed value of the coarse-grained properties.

3. Get available CPU and memory of all nodes and the end-to-end bandwidth between containers.

   For example, let the available resources be:
   Client: CPU 80%; Memory 500KB
   Server: CPU 50%; Memory 8000KB
   End-to-end bandwidth: 1Mbps

   Data on the microprocessor type and capacity and other pertinent information of each node need to be communicated as well. QoS-Profiles are generally measured on a node different from the one on which the components will be deployed. This makes the specification dependent on the processor speed, particular operating system, etc. of the node on which the measurements have been taken. In order to account for this dependency, either the values in the QoS-Profile must be converted to correspond to the new environment, or the available resources of the node on which the components are deployed should be changed to be consistent to the environment where the measurements have been taken. If the latter is opted for, the available CPU of the client and server given above as available resources have to be obtained after converting to the CPU time where the measurements of the QoS-Profiles of *VideoPlayer* and *VideoPlayer* have been taken respectively.

4. Sort QoS-Profiles of each component according to user's preferences

To demonstrate this step, we take the specifications given in Table 3.4 and Table 3.5. The sorted profiles of *VideoPlayer* and *VideoServer* are shown in Table 3.12 and Table 3.13.

VideoPlayer			
	uses ICompVideo (resolution, frame rate in S^{-1})	provides ICompVideo (resolution, frame rate in S^{-1})	Resource (CPU in %, bandwidth in Kbps, memory KB)
1.	30, 352x288	30, 352x288	13.23, 2165, 31.6
2.	30, 176x144	30, 176x144	0.97, 321, 25.6
3.	30, 128x96	30, 128x96	0.51, 152, 26.1
4.	15, 352x288	15, 352x288	8.90, 2146, 30
5.	15, 176x144	15, 176x144	0.90, 252, 18.8
6.	15, 128x96	15, 128x96	0.38, 120, 25.1
7.	10, 352x288	10, 352x288	8.91, 2076, 30.2

VideoServer	
Provides ICompVideo (resolution, frame rate in S^{-1})	Resource (bandwidth in Kbps)
30, 352x288	2165
30, 176x144	321
30, 128x96	152
15, 352x288	2146
15, 176x144	252
15, 128x96	120
10, 352x288	2076

VideoPlayer			
	uses ICompVideo (resolution, frame rate in s^{-1})	provides ICompVideo (resolution, frame rate in s^{-1})	Resource (CPU in %, bandwidth in Kbps, memory KB)
8.	10, 176x144	10, 176x144	0.62, 208, 24.4
9.	10, 128x96	10, 128x96	0.64, 108, 24.5
10.	5, 352x288	5, 352x288	5.90, 1852, 29.5
11.	5, 176x144	5, 176x144	0.39, 135, 24.0
12.	5, 128x96	5, 128x96	0.19, 70, 24.2
13.	1, 352x288	1, 352x288	2.31, 1644, 29.2
14.	1, 176x144	1, 176x144	0.11, 34, 24.4
15.	1, 128x96	1, 128x96	0.04, 24, 22.9

Table 3.12: Sorted profiles of *VideoPlayer* according to frame rate, resolution

VideoServer	
Provides ICompVideo (resolution, frame rate in s^{-1})	Resource (bandwidth in Kbps)
10, 176x144	208
10, 128x96	108
5, 352x288	1852
5, 176x144	135
5, 128x96	70
1, 352x288	1644
1, 176x144	34
1, 128x96	24

Table 3.13: Sorted profiles of *VideoServer* according to frame rate, resolution

Main Step:

1. Invoke `FineGrainedNegotiation()` (Algorithm 3.2).

   `ConformantConsistencyCheck()` performs arc consistency to connected components as explained in subsection 3.4.3.2. For the specifications given in Table 3.12 and Table 3.13, for every QoS-Profile of *VideoPlayer*, there exists at least one QoS-Profile of *VideoServer* that is conformant. Thus, the given specifications remain the same after executing `ConformantConsistencyCheck()`. After this step, `BOUND` is initialized to the user's QoS requirement (i.e. 176x144, 12fps) and the `selectedQoSProfiles` list is initialized to *empty*.

   The selection of profiles is performed step by step starting from the front-end component, i.e. *VideoPlayer*, and going though all other components according to the ordering of the components. For the front-end component, unlike other components, we assume a conformance constraint to exist (which is actually the user's constraint), between its offered QoS contract and the user's QoS requirement. The solution is then improved in iterations.

   Next, we will describe the outcome at each iteration to demonstrate how the algorithm operates. Only iterations that invoke `FindAppropriateProfiles()` are analyzed below.

1st Iteration

(i) `Quality-Point` $\leftarrow$ 176x144, 15fps

(ii) `FindConformantProfiles(ON_CLIENT)` gets a QoS-Profile for *VideoPlayer* as:

```
profile selectedProfile for VideoPlayer {
      uses  frameRate=15, resolution=176x144;
      provides frameRate=15, resolution=176x144;
      resources CPU=0.9% Memory=18.8KB Bandwidth=252Kbps;
}
```

(iii) `CheckResourceConstraint(ON_CLIENT)` is successful.

(iv) `FindConformantProfiles(ACROSS_CONTAINERS\ON_CLIENT)` will select QoS-Profiles for *VideoServer* as:

```
profile selectedProfile for VideoServer {
      provides frameRate=15, resolution=176x144;
      resources Bandwidth=252Kbps;
}
```

(v) `CheckResourceConstraint(ACROSS_CONTAINERS)` is successful.

(vi) `FindConformantProfiles(ON_SERVER\ACROSS_CONTAINERS)` will not add any additional selected profiles as there are no components on the server other than *VideoServer*.

(vii) `CheckResourceConstraint(ON_SERVER)` is successful in this iteration.

(viii) The `selectedQoSProfiles` list will be updated with the selected QoS-Profiles of *VideoPlayer* ((ii) above) and *VideoServer* ((iv) above). `BOUND` is also updated with 176x144, 15fps.

2nd Iteration

(i) `FindConformantProfiles(ON_CLIENT)` gets a new valid QoS-Profile for *VideoPlayer* as:

```
profile selectedProfile for VideoPlayer {
      uses  frameRate=15, resolution=352x288;
      provides frameRate=15, resolution=352x288;
      resources CPU=8.90% Memory=30KB Bandwidth=2165Kbps;
}
```

(ii) `CheckResourceConstraint(ON_CLIENT)` is successful.

(iii) `FindConformantProfiles(ACROSS_CONTAINERS\ON_CLIENT)` will select a QoS-Profile for *VideoServer* as:

```
profile selectedProfile for VideoServer {
      provides frameRate=15, resolution=352x288;
      resources Bandwidth=2165Kbps;
}
```

`CheckResourceConstraint (ACROSS_CONTAINERS)` is **NOT** successful.

This implies the bottleneck resource is identified to be the network bandwidth. The temporarily selected QoS-Profiles in this iteration (steps (i) and (iii)) are invalidated. The appropriate configurations, hence, are those selected in the previous iteration, i.e. 1st iteration.

2. The fine-grained negotiation (Step 1 above) is repeated corresponding to other values that have been agreed to during the coarse-grained negotiation. A final decision is made by comparing the results of the multiple fine-grained negotiations as explained in Algorithm 3.3. For the sake of brevity, let's assume the only coding type agreed during the coarse-grained negotiation is mp42. Thus, the whole negotiation process ends at this step. The run-time contracts created are shown in Figure 3.20. These contracts contain information about the selected profiles and user's requirements. A contract also contains other attributes as described in subsection 4.1.2.

Contract: User-VideoPlayer

```
UserQoSRequirement {
     uses frameRate>12, resolution=176x144;
}

profile selectedProfile for VideoPlayer {
     uses frameRate=15, resolution=176x144;
     provides frameRate=15, resolution=176x144;
     resources CPU=0.9% Memory=18.8KB Bandwidth=252Kbps;
}
```

Contract: VideoPlayer-VideoServer

```
profile selectedProfile for VideoPlayer {
     uses frameRate=15, resolution=176x144;
     provides frameRate=15, resolution=176x144;
     resources CPU=0.9% Memory=18.8KB Bandwidth=252Kbps;
}

profile selectedProfile for VideoServer {
     provides frameRate=15, resolution=176x144;
     resources Bandwidth=252Kbps;
}
```

Figure 3.20: User-VideoPlayer & VideoPlayer-VideoServer contracts

Next, we will see how the negotiation performs for a different input condition. Let's assume the user's QoS requirement to be the same but the preference to be different, i.e. the resolution to have a higher preference over the frame rate. The one change in the pre-processing step is that the profiles are sorted based on first resolution and then frame rate as shown in Table 3.14 and Table 3.15. Following similar steps as the previous example, the algorithm would select 30fps, 176x144 as the *VideoPlayer's* offered QoS contract. This also turns out to be the optimal solution.

VideoPlayer			
	uses ICompVideo (resolution, frame rate in s^{-1})	provides ICompVideo (resolution, frame rate in s^{-1})	Resource (CPU in %, bandwidth in Kbps, memory KB)
1.	352x288, 30	352x288, 30	13.23, 2165, 31.6
2.	352x288, 15	352x288, 15	8.90, 2146, 30
3.	352x288, 10	352x288, 10	8.91, 2076, 30.2
4.	352x288, 5	352x288, 5	5.90, 1852, 29.5
5.	352x288, 1	352x288, 1	2.31, 1644, 29.2
6.	176x144, 30	176x144, 30	0.97, 321, 25.6
7.	176x144, 15	176x144, 15	0.90, 252, 18.8
8.	176x144, 10	176x144, 10	0.62, 208, 24.4
9.	176x144, 5	176x144, 5	0.39, 135, 24.0
10.	176x144, 1	176x144, 1	0.11, 34, 24.4
11.	128x96, 30	128x96, 30	0.51, 152, 26.1
12.	128x96, 15	128x96, 15	0.38, 120, 25.1
13.	128x96, 10	128x96, 10	0.64, 108, 24.5
14.	128x96, 5	128x96, 5	0.19, 70, 24.2
15.	128x96, 1	128x96, 1	0.04, 24, 22.9

Table 3.14: Sorted profiles of VideoPlayer according to resolution, frame rate

VideoServer	
provides ICompVideo (resolution, frame rate in s^{-1})	Resource (bandwidth in Kbps)
352x288, 30	2165
352x288, 15	2146
352x288, 10	2076
352x288, 5	1852
352x288, 1	1644
176x144, 30	321
176x144, 15	252
176x144, 10	208
176x144, 5	135
176x144, 1	34
128x96, 30	152
128x96, 15	120
128x96, 10	108
128x96, 5	70
128x96, 1	24

Table 3.15: Sorted profiles of VideoServer according to resolution, frame rate

In another scenario, let's take a case where the client resource is the bottleneck with all other conditions being unchanged. The user has still a higher preference for frame rate.

Available resources:

Client: CPU 8%; Memory 500KB
Server: CPU 50%; Memory 8000KB
End-to-end bandwidth: 10Mbps

The CPU times shown above as available resources have been obtained after converting to the CPU time where the measurements of the QoS-Profiles have been taken. Following similar steps as done previously, the final outcome is: 176x144, 30fps as the *VideoPlayer's* offered QoS contract. The bottleneck is felt first at the client.

Finally, we will consider a case of an unsuccessful scenario. This may arise when we need to fulfill user's QoS requirements while resource or/and conformance constraints cannot be met.

Let the input conditions be as follows:

User's requirement and preference:
frame rate > 12fps, resolution = 176x144
resolution is preferred to frame rate

Available resources:
Client: CPU 80%; Memory 500KB
Server: CPU 50%; Memory 8000KB
End-to-end bandwidth: 200Kbps

The 1st iteration of Algorithm 3.2 cannot be completely executed as `CheckResourceConstraint(ACROSS_CONTAINERS)` returns an unsuccessful result. To handle this over-constrained condition, the user's requirement will be relaxed in succession. This step depends on the QoS-Profiles of *VideoPlayer* (Table 3.14). In the first iteration of the relaxation step, the chosen degraded requirement is: resolution = 176x144 and frame rate = 10fps. The negotiation results once more in an unsuccessful outcome. The relaxation is performed to reduce the requirement further to: resolution = 176x144 and frame rate = 5fps. At this step, the negotiation turns out to be successful with selected profiles as below.

```
profile selectedProfile for VideoPlayer {
      uses  frameRate=5, resolution=176x144;
      provides frameRate=5, resolution=176x144;
      resources CPU=0.39% Memory=24KB Bandwidth=135Kbps;
}
profile selectedProfile for VideoServer {
      provides frameRate=5, resolution=176x144;
      resources Bandwidth=135Kbps;
}
```

If the user's requirement had been given in a range (like frame rate 5 – 15 fps), this information would have been used in the relaxation step not to further relax the user's requirement below the given minimum.

3.6 Chapter Summary and Conclusions

This chapter discussed the core of our approach. We started by drawing an analogy between QoS contract negotiation with the findings of the automated negotiation research. We described our model in a simplified form as follows: the negotiating parties are the components, the user, and the containers, with the latter having the role of arbitration. The collaborating components and the user put forward *all* their proposals (i.e. the multiple QoS-Profiles and the user's QoS requirements and preferences) and the container analyzes the problems (i.e. the constraints) and dictates the solution for the parties. After a successful negotiation, contracts are established and will be monitored and enforced by the container. When no agreements can be reached, the user makes concessions by way of automatic relaxation of his/her requirements. In the event that no agreement is still to be reached, the negotiation terminates with a "conflict deal".

We have formalized QoS contract negotiation as a constraint satisfaction problem (CSP). In the CSP modeling, the variables are taken to be the QoS-Profiles of the collaborating components. The domain of each variable consists of the set of all QoS-Profiles specified for a given component. The constraints are classified as *conformance*, *user's* and *resource*. Furthermore, we have shown how to extend the CSP into a CSOP (Constraint Satisfaction Optimization Problem) in order to help us mainly address one of the challenges we face in the negotiation, i.e. the selection of a good solution. The notions of conformance and utility that are crucial to the problem formalization have been discussed by giving examples.

Central to our solution is the classification of the QoS contract negotiation in multiple phases. We have argued that performing negotiation in multiple phases makes the process less complex and more efficient. The various phases we have identified are: *coarse-grained* and *fine-grained* negotiations both for the collaborating components deployed in distributed nodes and for the respective component containers. This classification, we believe, helps better understand the nature of NFPs as illustrated by our categorization of NFPs as coarse-grained and fine-grained properties. A coarse-grained property is a component implementation's property that can be associated with one or multiple fine-grained properties. This association is created by the fact that for a certain value of the coarse-grained property the fine-grained properties can possibly take different values depending on the allocated resource (e.g. CPU, bandwidth, memory). As an example, we can take coding type and protocol properties as coarse-grained properties corresponding to frame rate, resolution, smoothness, and delay properties, which are the fine-grained properties.

Pertaining to the classification scheme described above, we have proposed algorithms both for coarse-grained and fine-grained negotiations that aim to find a "better" solution. We have also shown how the two algorithms can be combined into a single algorithm that may be regarded as the main negotiation algorithm. The fine-grained negotiation algorithm has been developed based on the standard branch-and-bound (B&B) algorithm. As B&B is a very general framework, certain policies and functions must be defined in order to make use of it in the particular problem domain. For this, we have proposed appropriate variable and value selection policies, objective, and heuristic functions. We have also put forward a mechanism for relaxing constraints when the contract negotiation cannot produce a solution. This has been achieved by extending the CSP framework into a partial CSP.

We also discussed the possible architectures for implementing the proposed algorithm. We discussed the pros and cons of three approaches: *centralized*, *distributed*, and *hybrid*. We concluded that owing to the nature of our problem we don't consider a pure distributed solution as an option. The choice between centralized and hybrid solutions is also not clear-cut. It depends on the number of components and QoS-Profiles involved and the environment conditions. The decisions as to whether to apply a centralized or a hybrid approach can be made just before the start of the negotiation based on certain input conditions. The particular choice depends on the cost of inter-container communication and the required level of computation. If the former is expensive, a centralized solution would be better. Nevertheless, for a case that requires a lot of computation and the environment offers relatively faster inter-container communication, the hybrid solution would be the more efficient approach. During run-time re-negotiations, the hybrid solution is the more preferable choice as it avoids unnecessary overheads when it is applicable to make re-configurations locally.

Finally, we validated the contract negotiation algorithms and protocols that we proposed by considering various scenarios in the distributed componentized video streaming application. In order to perform the validation, we have conducted first an experiment to specify the QoS contracts of the two collaborating components in the application: *VideoPlayer* and *VideoServer*. Then, based on the specified data, we have shown with the help of our algorithms how the contract negotiation can be performed in three phases: Container-Container Negotiation between the client and server containers, Coarse-grained Negotiations between *VideoPlayer* and *VideoServer*, and Fine-grained Negotiations between *VideoPlayer* and *VideoServer*.

This chapter has mainly dwelt upon QoS contract negotiation for the *single client – single server* scenario. As we pointed out in section 1.2, there are other important scenarios that need investigation. Two such scenarios - multiple-clients and multi-tier - are the subject of discussion of the next chapter. The general approach we have been following from the outset has been to fully or adequately address the issues for the *single client – single server* case and propose mechanisms for generalizing the solution to other scenarios.

4 Generalizing to Other Scenarios

The contract negotiation algorithms and protocol we proposed earlier are based on the *single client – single server* scenario. There are other interesting scenarios in the real world applications. Some of these are: (i) multiple clients (users), (ii) multi-tier (e.g. a server component uses the service of another component deployed in a different container), and (iii) peer-to-peer. In this section, we would propose mechanisms for extending or adapting our solution to other scenarios.

4.1 Multiple-Clients Scenario

In the real world, service providers offer their service to multiple clients at the same time. In this respect, the *single-client* scenario that we have already addressed in Chapter 3 isn't directly applicable to real world problems. However, as we will demonstrate in the subsequent sections, the single client case can be taken as a basis for the *multiple-clients* scenario, which is the subject of discussion of this section. A pictorial representation of a multiple-clients scenario is shown in Figure 4.1.

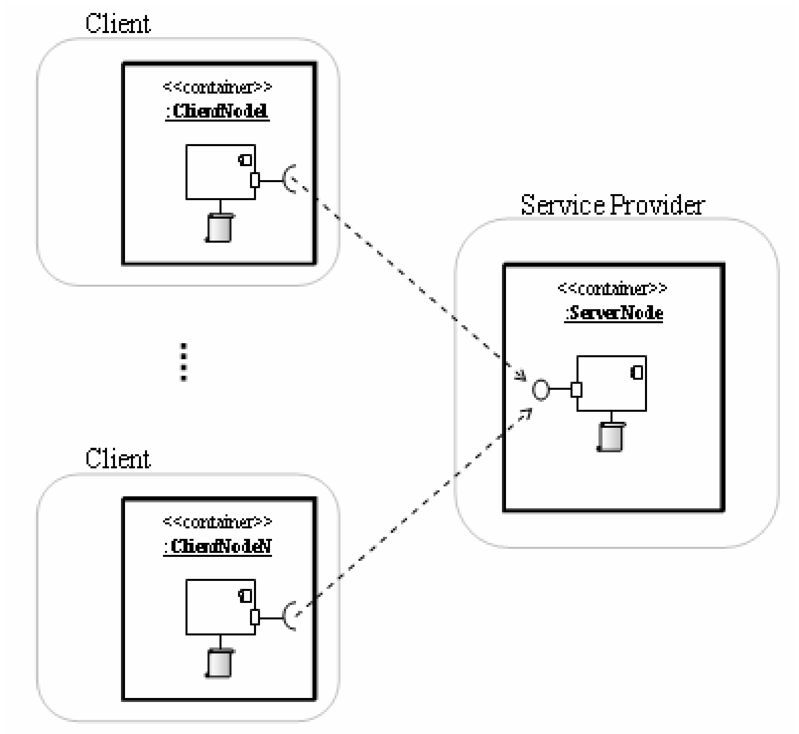


Figure 4.1: Multiple-Clients Scenario

We motivate our discussion of the multiple-clients scenario based on two example distributed applications: video streaming and stock quote. For the former, two different sub-scenarios can be identified.

(i) Each client negotiates with the server about its requirement at a convenient time and in an independent manner (a case of video-on-demand). The content and quality of each stream might vary among the clients.

(ii) The server broadcasts the same video stream to the clients. In this case, the broadcast has to account for the variations of clients in supporting video of different quality, coding, and protocol.

One difference between the above two cases is that in (i) an independent contract between each client and the server is required while in (ii) there is a kind of group contract between the server and a set of clients. In (ii) a negotiation could take place between the server component and other intermediate components that convert the stream quality according to the capabilities of the various clients.

The multiple-clients scenario poses new challenges which didn't exist in our single-client scenario. Some of these are:

(i) New clients constantly send requests for a service. During this period, some contracts of clients haven't yet expired while certain clients leave the system. The new requests might follow a particular pattern or they could even occur in bursts.

(ii) Multiple clients usually have varying requirements and expectations about the QoS delivered by the service provider.

(iii) There is a need for considering new parameters like *contract duration* and *time of service delivery*.

The first characteristic above has an impact on the server's decision making process during the negotiation. The input about existing active contracts and probably future agreed contracts between the service provider and clients together with the new clients request pattern and rate should aid the service provider to forecast its resource availability or workload. This in turn helps the service provider to make appropriate (intelligent) decisions during contract negotiation. The decision might necessitate re-negotiation or even termination of certain active contracts in order to establish new contracts if the existing contracts have actually a lower priority/class.

The second characteristic above emanates from the fact that some customers are willing to pay more for getting high quality service while others are content to get less quality for the amount they pay. For example, a stock quote service could report prices with different levels of timeliness, ranging from real-time to a fifteen minute delay to a 24-hour time lag.

This characteristic might in some cases stem due to variations in the client resources capacity, which is also termed in some literature as client heterogeneity.

Contract duration (validity period) was not considered in our single-client scenario. The reason for this assumption is that it doesn't influence the negotiation process as only one client is assumed to be involved. But, in the multiple-clients case, its consideration does have an impact in light of the first characteristic mentioned above. Next, we will address the three characteristics in more detail and propose a QoS contract negotiation approach for a multiple-clients scenario.

4.1.1 Classes of users and service class

Service providers can offer differentiated QoS to clients/users based on the amount the users pay at the service request time. Decisions on specific payment methods in QoS negotiation may require analyzing different business model options. In this thesis, we do not consider payment properties for handling differentiation in the quality of provided service. We would rather realize the differentiation by assuming the existence of different classes of users who are entitled to get "premium" or "normal" service; or users having golden, silver, or bronze cards, etc. The actual determination of how to classify a user, which could be based on a monthly user fee to the service provider, is out of the scope of this thesis. We simply assume that different classes of users exist.

In order to be able to define classes of users and hence make a QoS differentiation, the services rendered by the service provider must be classified into different *service classes*. We define a *service class* as a service with a common functionality but different quality (e.g. response time, frame rate). The QoS property of the service classes depends on the specified QoS-Profiles of the collaborating components deployed on the server. For instance, if there is only one component on the server, its QoS-Profiles are used to derive the QoS properties of the service classes.

Considering our video streaming scenario, only *VideoServer* is deployed on the server side. If the QoS-Profiles of *VideoServer* are specified for a combination of 5 different frame rates and 5 different resolutions, there are 25 different QoS-Profiles. In principle, 25 different service classes can be defined based on the 25 QoS-Profiles. But, defining large number of service classes might make the management of client requests difficult. Moreover, the rationale for such a large number of service classes in the real world applications is questionable. Besides, it is advantageous to define the quality level of a service class in a range where chosen QoS points can deteriorate within this range reflecting the load conditions on the server. Hence, the 25 different QoS-Profiles can be grouped into 5 service classes (two of them are listed in Table 4.1).

In addition to the QoS, certain other attributes can be associated with a service class, like priority, guarantee, etc. One service class may have a higher priority than the other. This would have an effect on the negotiation when more clients exist than the service provider's capacity allows.

To demonstrate the concept of service class, we take an example from our video streaming scenario. Let's assume that the video streaming provider classifies two service classes: *High Quality* and *Low Quality*. There is a one-to-one mapping between the service class and the user class, which we can identify as *premium* and *normal*. The user associated with *premium* class is entitled to get *High Quality* service while *normal* users get *Low Quality* service. In the subsequent discussions, we use High Quality and Low Quality interchangeably with premium and normal respectively. Table 4.1 shows the quality of service available in the two service classes.

	premium service class (resolution, frame rate in s^{-1})	*normal* service class (resolution, frame rate in s^{-1})
1.	352x288, 30	176x144, 30
2.	352x288, 25	176x144, 15
3.	352x288, 20	176x144, 10
4.	352x288, 15	176x144, 5

Table 4.1: QoS points in a *premium* and *normal* service class

The minimum quality assigned to *premium* and *normal* is 352x288, 15fps, and 176x144, 5fps respectively. The service provider has the obligation to provide the minimum qualities specified in Table 4.1 in both service classes. Nevertheless, a client may prefer to get the service even if it is inferior to the promised quality (e.g. at 128x96, 10fps) rather than not getting a service at all (see subsection 3.4.5). Hence, more service classes could be defined based on the classification in Table 4.1. The following four classes are such an example: (i) premium, (ii) premium for which the minimum can be violated in some narrow range, (iii) normal, and (iv) normal for which the minimum can be violated. Under light load conditions, the service provider could agree to make a contract with 176x144, 30fps (the maximum possible in the service class) as there is no shortage of resources.

4.1.2 More Parameters Considered

The additional parameters that we discuss here are *service duration* (contract duration or contract validity period) and *time of service delivery*. Still there could be other parameters that might affect the negotiation process. For example, when a customer/client requests a service for the first time, the service provider might want to attract him or her by making some concessions. In the rest of the discussion, we consider only service duration and time of service delivery. We give an example as to how these parameters can affect the negotiation process.

i) Service or Contract Duration

Service or contract duration refers to the time interval between the start and end time of the provided service. If an agreement is reached between the service provider and the customer, service duration is the valid period of the contract in which both parties agree to fulfill their obligation.

In our video streaming scenario, *VideoServer* might stream videos of various lengths, which ranges from very short video clips to long movie clips that may last hours. This implies that if the streaming rate is equal to the real-time play back rate, the resource demand of the particular instance of *VideoServer* must be available throughout the length of the video. That is, the contract validity period should be equal to the video length. The service duration also applies to the Stock Quote application scenario in which case customers may specify the time span they need to receive the stock quotes. In general, it is not always possible to know the service duration for some applications. For example, in a VoIP (or video conferencing) system, it is not known beforehand how long the voice and video communication will last.

ii) Time of Service Delivery (waiting time)

Time of service delivery refers to the point in time when the service is available to the customer with reference to the service request time. This can be in the immediate time or after some delay. The users could specify their preferences and give different weights on the QoS attributes and time of delivery.

The consideration of the two parameters described above help the contract negotiation process to be performed in a more flexible and efficient way. For example, the service provider could propose the user multiple offers. This requires knowledge of the available resources at the time of service request and in future times. The available resources can be derived based on the currently *active contracts* and already *agreed future contracts*, if there are any. During the negotiation, if there is not enough resource for the requested service, two possible offers from the server's side are: (i) to deliver the service at a degraded quality at the time of the service request, or (ii) to deliver the service at the requested (or even better) quality at a designated later time. The decision between the two offers can be made based on the preferences (relative weights) of the user towards the quality attributes and the time of service delivery. Users may get encouraged to agree to future contracts in order to get a superior quality.

To elaborate further the effect of contract duration and time of service delivery on the contract negotiation process, we will discuss a case from our video streaming application. Let us assume five contracts exist between the service provider and five different clients at present. $Cont_1$, $Cont_2$, and $Cont_3$ are active contracts while $Cont_4$ and $Cont_5$ are future agreed contracts. Each contract can be represented by the attributes shown in Figure 4.2. Additional attributes like *terms of penalty* (financial consequences when breaking contracts) may need to be included in the contracts. For our purpose, we focus only on the described elements of the contract.

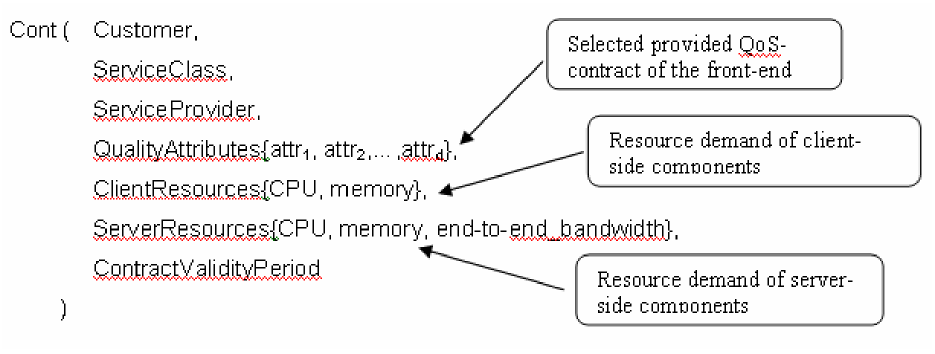

```
Cont (  Customer,
        ServiceClass,
        ServiceProvider,
        QualityAttributes{attr1, attr2,... ,attrd},
        ClientResources{CPU, memory},
        ServerResources{CPU, memory, end-to-end_bandwidth},
        ContractValidityPeriod
     )
```
Selected provided QoS-contract of the front-end
Resource demand of client-side components
Resource demand of server-side components

Figure 4.2: Some attributes of a Contract between a service provider and a user

Figure 4.2 also depicts how part of the attributes of a contract between a user and service provider can be derived from the selected QoS-Profiles of the collaborating components. Some of the attributes in Figure 4.2 (e.g. the resources) might not be required to be visible to the end-user. Nevertheless, they are important for the contract enforcement and monitoring modules of the system.

In Figure 4.2, the first and third attributes represent the involved parties, i.e., customer and service provider. The second attribute describes the service class (or equivalently the user class). The fourth element represents the agreed quality attributes of the service as received by the user. The fifth attribute represents the client resources allocated for all components deployed on the client node. The sixth element specifies the server resources allocated for the service (components deployed on the server). The end-to-end bandwidth is included as server resource for the sake of simplicity. The last attribute is the contract duration, which may have two parts – the start time of the contract and the contract length. Some precautions must be taken when representing contract duration due to problems of time synchronization of distributed nodes. It is assumed that during the contract validity period all of the resources are required.

Let us assume that the five contracts: $Cont_1$, …, $Cont_5$ are as given below. For the sake of generality, the duration of each contract is shown to take different values.

$Cont_1$(C1, premium, SP, Quality{352X288, 30fps}, clie{13%CPU,32KB},
serv{0.1%CPU,20KB,2.2Mbps}, {t_{jj}, 20min})
$Cont_2$(C2, premium, SP, Quality{352X288, 10fps}, clie{9%CPU,30KB},
serv{0.1%CPU,20KB,2Mbps}, {t_{ii}, 10min})
$Cont_3$(C3, normal, SP, Quality{176X144, 30fps}, clie{1%CPU,26KB},
serv{0.1%CPU,20KB,0.3Mbps}, {t_{mm},15min})
$Cont_4$(C4, premium, SP, Quality{352X288, 30fps}, clie{13%CPU,32KB},
serv{0.1%CPU,20KB,2.2Mbps}, {t_k, 10min}) $\rightarrow$ future contract!
$Cont_5$(C5, premium, SP, Quality{352X288, 10fps}, clie{9%CPU,30KB},
serv{0.1%CPU,20KB,2Mbps}, {t_l, 20min}) $\rightarrow$ future contract!

Figure 4.3 depicts the bandwidth resource[4] allocated for each contract with respect to time. The current time is represented as $t_{current}$. Similar diagrams can be drawn for each resource type (e.g. CPU, memory) allocated for the contract.

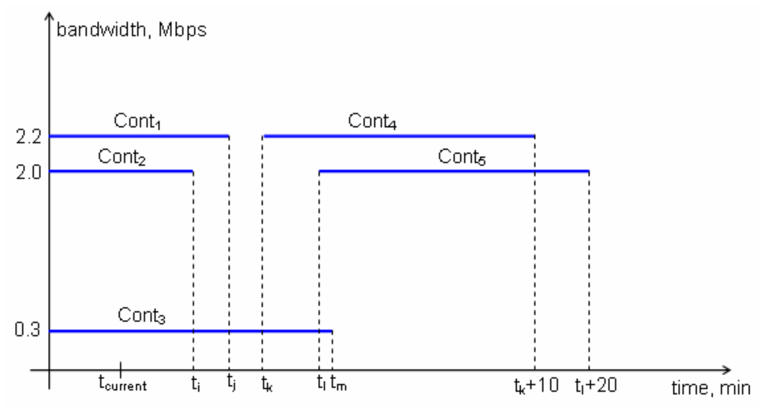


Figure 4.3: Bandwidth usage of active contracts and future agreed contracts

Figure 4.3 is used to derive and forecast available bandwidth at the time of the negotiation and in the future. Similar computations can be made for the other resource types. Diagrams of the total bandwidth usage and available bandwidth as derived from Figure 4.3 are depicted in Figure 4.4. We assumed a maximum of 100Mbps is supported at the server's side in Figure 4.4.

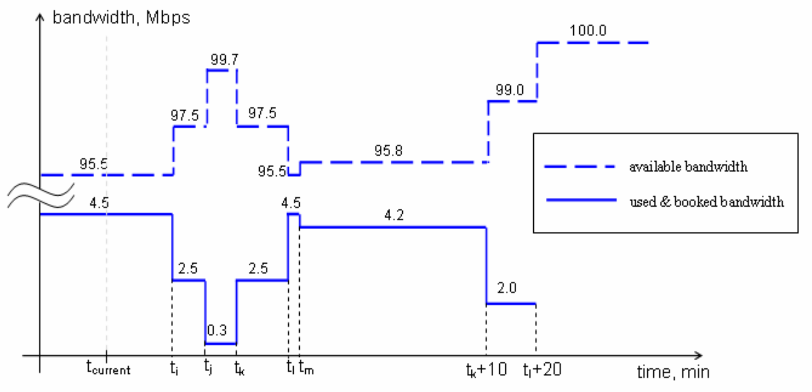


Figure 4.4: Total bandwidth usage of active and future contracts and available bandwidth at the server

In Figure 4.4, it is assumed that no resources are shared among the contracts. That is, each contract exclusively requires the resources described in the contract. However, in some applications like multicasting or broadcasting, different contracts may share server

[4] Strictly speaking, bandwidth is given in [Hz] while data rate is given in [bps]

resources. An additional attribute may be required in Figure 4.2 to represent which contracts share resources. The significance of the diagrams in Figure 4.4 is that the service provider can forecast the available resources at the time of negotiation and thereafter in order to make multiple proposals during resource shortages.

4.1.3 QoS Contract Negotiation in a Multiple-Clients Scenario

In a negotiation between a client (customer) and a server (service provider), the choice of a concrete contract is guided by the negotiation goals of both the client and server. The customer desires to get the maximum possible quality at a given price or service class. On the other hand, the service provider aims at maximizing its revenue through higher and efficient resource utilization. Favoring one client over another (even if both are in the same user class) may also be considered by the service provider due to one client's long-term relationship with the service provider. In this thesis, a service provider's negotiation goal is taken to be efficient (maximum) resource utilization while at the same time fulfilling user's minimum QoS requirements in each service class.

When multiple clients are involved in the contract negotiation, the key question is how to allocate resources to each client. The allocation then determines the QoS points for each client. In the literature, we find many resource allocation approaches for QoS-aware distributed systems [Lee et al., 1999] [Abdelzaher et al, 2003] [Levy et al., 2003] [Park et al., 2001]. In [Lee et al., 1999], a QoS management framework that enables to quantitatively measure QoS, and to analytically plan and allocate resources has been presented. In this model, the quality preferences of the end users are considered when system resources are apportioned across multiple applications such that the net utility that accrues to the end-users is maximized. In [Abdelzaher et al, 2003] a communication server architecture that maximizes the aggregate utility of QoS-sensitive connections for a community of clients even in the case of over-load has been proposed. In this model, the optimal resource allocation policy will always keep the clients with the largest utility for the same resource consumption. In [Levy et al., 2003], a resource allocation strategy is discussed for a clustered web-service environment. In [Park et al., 2001], resource allocation is discussed in the management of service level agreements for multimedia Internet service.

The findings of all of the aforementioned research are important to our problem though our context and emphasis is different. We have found it important to employ in our approach the utility model, which has been applied in all of the mentioned literature. The work in [Abdelzaher et al, 2003] uses the notion of QoS Contract although its usage is slightly different from ours. There are basically two reasons that make our work different from those mentioned above. Firstly, none of them approached the problem in the context of CBSE. For example, in our case, the contract between a customer and a service provider (also known as SLA) depends on and are derived from the selected QoS-Profiles of the collaborating components. Besides, the service classes are defined based on the specified QoS-Profiles of the components. Secondly, the focus of most related approaches is the allocation of server resources. Clients' properties haven't been considered together with the servers' properties. In [Lee et al., 1999], however, there is no such restriction as server side. But, it treats the allocation problem for resources which are shared.

A utility function is defined for each service class. As applications can involve multiple QoS-dimensions, the utility function of an application is defined as a weighted average of the dimension-wise utility functions (see Section 3.6.1). If the application has d QoS-dimensions, the utility of the service class s_i, U_{s_i} is given as:

$$U_{s_i} = \sum_{j=1}^{d} w_j U_j(q) \qquad (1)$$

In Eq(1), $U_j(q)$ is the dimension-wise utility, q designates the chosen QoS point, and $\sum w_j=1$. The weights, w_j, define the relative importance of each dimension-wise utility function. For the example in Table 4.1, two utility functions must be defined for the two service classes. The fact that the qualities in "normal" service class are inferior to those in the "premium" service class doesn't mean that utility values of the former are lower than those of the latter. For users of each class, quality expectations are different and thus the maximum quality in each service class can be assigned utilities of 1.

The overall system utility, U, is defined as a weighted average of the utilities of each user. A user's utility is associated to one of the service classes (Eq(1)). The relative weightings (α_i) assigned to the different service classes capture how important that service class is to the service provider.

$$U = \sum_{clients} \alpha_i U_{s_i} \qquad (2)$$

The required analytical solution is to find QoS points (which depend on the QoS-Profiles of the collaborating components) that maximize U in Eq(2). This solution must be found under the assumption that the following conditions are fulfilled.

(i) the resource demands of all distributed components are met at the server node, the network, and at each client's node;
(ii) user's minimum QoS requirements are met;
(iii) conformance constraints of the QoS-Profiles of the collaborating components are met;
(iv) the chosen solution must maximize U per resource consumption at the server side.

The fourth condition above is especially important for the service provider in order to meet its negotiation goals. To clarify this, let's refer to our video streaming scenario. Let's assume there are two different selections of QoS-Profiles of *VideoServer* and *VideoPlayer*, which result in the same stream quality (i.e. in frame rate and resolution) as perceived by the end-user. As far as resource demands are concerned, one of the solutions demands more network bandwidth while the other needs lesser bandwidth at the expense of more CPU time at the client's side. According to Eq(1), both contracts have the same utility value. But, looking from the service provider's interest, the one that requires lesser bandwidth is chosen so that the server can increase its revenue by negotiating with another client.

Problems of the type in Eq(2) are known to be NP-hard [Lee et al., 1999]. To cope with this difficulty, we resort to a heuristic solution, which we will discuss next. Our solution also considers dynamic situations at the server side like change of the number of clients. We propose the following procedure, which uses our single-client solution as one of its steps. A more formal algorithm will be given at the end of this subsection once we discuss the details listed below. Figure 4.5 depicts the various steps of the procedure.

1. Customers input QoS requirements and quality preferences on the service.
2. Clients send requests for a service.
3. The server containers estimate server resources (e.g. CPU, memory, network bandwidth) to be allocated to each client (or group of clients).
4. A contract negotiation is performed between each client and the server on an individual basis (or group basis). This would result in establishing contracts among the collaborating components and between the customer and service provider.
5. The server checks if some resources are left unutilized after Step 4. If some resources are still available, the server will be engaged in a negotiation with new clients. (Then, Go to Step 3)

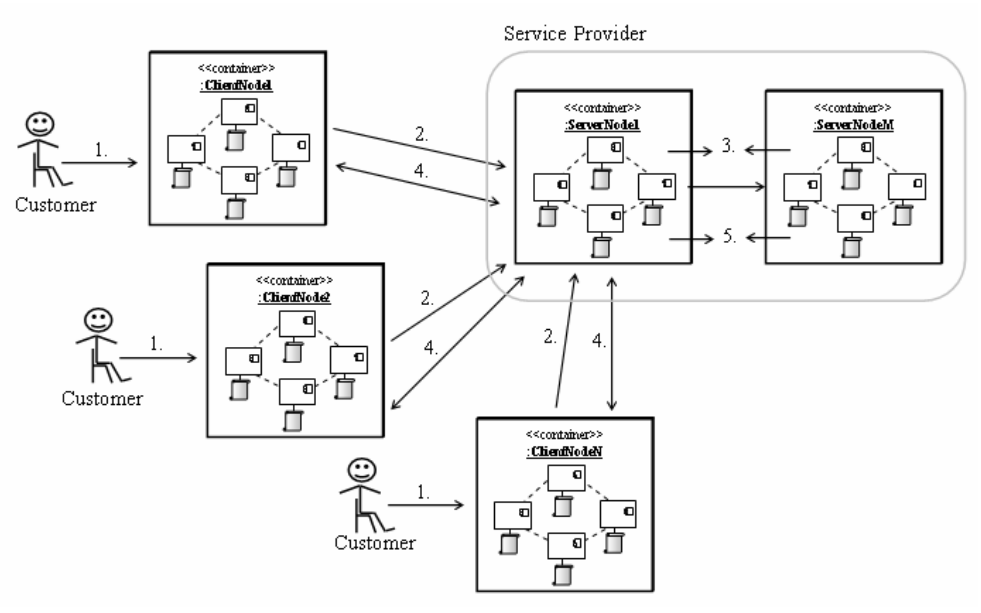


Figure 4.5: Steps in the QoS Contract Negotiation in a Multiple-Clients Scenario

From the five steps stated above, the important ones are the third and the fourth. The fourth step takes into account the particular characteristics of each client like the client's resource conditions and the components deployed on the client container. We have already proposed algorithms that can be applied for Step 4 (see Section 3.6.4). Step 3, i.e. estimating the resources to be allocated to the clients, which are currently negotiating or which are anticipated to arrive at a later time, depends on the current and future resource availability at the servers. Next, we discuss this third step in detail.

4.1.3.1 Resource Allocation Strategy

The resource allocation to the multiple clients depends on the availability of server resources. Instead of treating the allocation in a First Come First Served (FCFS) manner, the decision should be influenced by the total number of clients placing requests and the rate of new arrivals. When only few clients make requests and there is abundant resource, maximum resource is allocated (services are offered at maximum quality). On the other hand, when there are many clients and the available resource is meager, minimum resource is allocated (services are offered with lesser quality).

To elaborate the distinctions between different load conditions, let's see the factors that affect the maximum number of clients (*N*) that can be supported by the service provider concurrently. As far as the service provider is concerned, *N* must be large enough to efficiently utilize the server's physical resources, but small enough to prevent overload and performance degradation, and at least fulfill the minimum QoS for each service class and each client as promised in the contract. In general, *N* depends on the type of service class selected (e.g. premium or normal, Table 4.1), the service mix (how many clients from each service class), etc. Let's assume the server's bandwidth capacity is 100 Mbps, and premium and normal class' minimum bandwidth requirement are 2 and 0.3Mbps respectively. If only premium class is supported, then N=50. If only normal class is supported, N=333. *N* falls between 50 and 333 when mixing service classes. Different alternatives (policies) can be taken as to this mixing as discussed later in over-load case. It is to be noted that *N* has been estimated above only based on the bandwidth requirement of each service class. Nevertheless, *N* can be determined based on only the CPU requirement for instance for CPU-bound applications. In some cases, the combination of all resource type requirements can decide on the size of *N*. The resource(s) that determine the size of *N* are termed as bottleneck resource(s).

In light of the dynamic nature of the load conditions, it would be appropriate to discuss resource allocation under different load conditions and load dynamics. For this reason, we will consider the following three cases:

(i) *light-load* conditions,
(ii) conditions where the clients' request rate is known, and
(iii) *over-load* conditions.

I. Light-Load Case

Light-load is a situation where few clients make requests and there is enough server resource for all of them. During this condition, the total number of clients (those making requests and those already accepted) is much less than *N* (or some percentage of *N*, e.g. *80%N*).

At light load all clients are offered the maximum resource that is allocated for each service class. That is, the application's utility (on a per client basis) takes the maximum value. Referring to Table 4.1, for premium clients, the offered quality is 352x288, 30fps and for normal clients this is 176x144, 30fps. It is to be noted that these maximum offered qualities might not last the whole contract period, when for example there is a burst of client requests. The obligation of the service provider is to provide the minimum quality in each service class.

II. Known Clients' Request Rate Case

At light load conditions, we assumed that the server resource conditions remain always abundant. Therefore, each client is provided with the maximum quality. But, when the number of new clients increases more than those leaving the system, the resources will no longer be abundant. What the server must do in this situation is to re-negotiate the previously agreed contracts to a lower level whereby some resources are released for new clients. This calls for a systematic approach for handling these dynamic conditions to ultimately meet the negotiation goals of the clients and server. Some of the questions to be answered are: do we have to re-negotiate already established contracts whenever new clients request a service? To what level should we re-negotiate established contracts? For example, in Table 4.1, each service class defines four levels. How important is it for the server to know how many clients are anticipated in the future?

Answering some of these questions is important for the overall negotiation process. Simply doing negotiations for all clients (requesting and existing), whenever new clients arrive or with some periodicity, could have an adverse effect on the system. Such schemes make the system less stable due to the frequent changes in the offered QoS level. The answers to some of the aforementioned questions depend on the request rate of clients and the rate at which existing clients leave the system as explained below.

Let's assume the clients' request rate, $C_{req\text{-}rate}$ (in `min`$^{-1}$), and contract duration, D (in `min`) are constants. A given service class' resource requirement depends on the selected QoS point. Let the maximum requirement is R_{max} and the minimum is R_{min}. When R_{max} (R_{min}) is allocated to a single client, the maximum (minimum) quality in the service class is offered to the respective client. In between R_{max} and R_{min}, other intermediate qualities can be served. For R_{max} or R_{min}, m different resource types (e.g. CPU, network bandwidth, disk bandwidth, etc.) are assumed to exist. Let the server's total capacity, $R_{capacity}$, be also given in all dimensions of the resource types.

$$R_{\max} = (r_{1,\max}, r_{2,\max,}, \dots, r_{m,\max})$$
$$R_{\min} = (r_{1,\min}, r_{2,\min,}, \dots, r_{m,\min})$$
$$R_{capacity} = (r_{1,capacity}, r_{2,capacity,}, \dots, r_{m,capacity})$$

Given $C_{req\text{-}rate}$, and D, the maximum number of contracts active at any one time can be computed as:

$$TotalNumberOfContracts = C_{req-rate} \times D \qquad (3)$$

A condition for the client's request rate, where maximum quality can be served for all clients can be derived based on the resource constraint at the server's side, which is given below.

$$C_{req-rate} \times D \times R_{\max} \le R_{capacity} \qquad (4)$$

$$C_{req-rate} \le \frac{R_{capacity}}{D \times R_{\max}} \qquad (5)$$

The $C_{req\text{-}rate}$ in Eq(5) is determined by the bottleneck resource. For example, in video streaming application, the network bandwidth is usually the bottleneck resource. Considering *m*-resource types, Eq(5) can be re-written as:

$$C_{req-rate} \le \min(\frac{r_{1,capacity}}{D \times r_{1,\max}}, \frac{r_{2,capacity}}{D \times r_{2,\max}}, \ldots, \frac{r_{m,capacity}}{D \times r_{m,\max}}) \qquad (6)$$

Thus, if Eq(6) is satisfied,

- all clients can be allocated the maximum resource – i.e. the maximum requirement of each class can be fulfilled by the service provider
- there is no need of re-negotiating established contracts at any time

Considering our video streaming scenario, let the total available bandwidth at the server's side, $r_{i,capacity}$, be 100Mbps, $D=2min$, and the bandwidth requirement of the maximum quality in the given service class, $r_{i,max}$ be 2Mbps. Thus, if $C_{req\text{-}rate} \le 25min^{-1}$, then all clients can be offered the maximum service with no need of contract re-negotiations.

Depending on the value of $C_{req\text{-}rate}$ two other cases might occur as shown in the following relations.

$$\frac{R_{capacity}}{D \times R_{\max}} \le C_{req-rate} \le \frac{R_{capacity}}{D \times R_{\min}} \qquad (7) \text{ or}$$

$$C_{req-rate} > \frac{R_{capacity}}{D \times R_{\min}} \qquad (8)$$

If Eq(7) holds true, all client requests can be fulfilled with no request waiting. But, the offered qualities can be as low as the minimum in each service class. A part of the clients can, however, be served a quality higher than the minimum. This can be decided through policing that may favor one client over another. For Eq(8), not all clients' requests can be fulfilled, even with the minimum quality. This condition leaves the system in an overloaded situation where some clients must always be rejected. For a request rate of $C_{req\text{-}rate}$ that satisfies Eq(8), the rejection rate (by the server) can be computed as:

$$rejectionRate = C_{req-rate} - \min(\frac{R_{capacity}}{D \times R_{\min}}) \qquad (9)$$

In our analysis of known clients' request rate case, we assumed the existence of only a single service class. This assumption is necessary only if the system transitions into an overload condition. It is possible to relax the assumption if we know how much of the server capacity is allocated to clients of each service class.

III. Over-Load Case

Over-load is a situation where the capacity of the service provider isn't sufficient to establish contracts with all of the clients requesting service. Under this condition, the total number of clients (those making requests and those whose contracts haven't expired) is greater than *N* (or some percentage of *N*, e.g. *80%N*).

Suppose the number of requesting and existing clients is *N'* where *N'>N*. Then, *(N'-N)* clients must wait for some period until they get the required service. Let's also assume that the size of the queue length (in number of clients) is *S*. A relation can be derived for *S* where a maximum waiting time for clients can be guaranteed. If t_{wait} is the maximum waiting time before a client's request is served, then:

$$t_{wait} = \frac{S}{N} \times D \qquad (10)$$

For a desired minimum waiting time, the queue length can be fixed using the relation given below.

$$S = \frac{N \times t_{wait}}{D} \qquad (11)$$

If *N'-N>S*, then *N'-N-S* clients must be rejected. Before rejecting these clients, offers must be proposed to the S clients. The offers include the interval clients must wait before getting the service in addition to the QoS. Some clients may reject this offer. The decision on whether or not to accept the offer can be done by the containers if a user's preference on waiting time has been available.

When the system transitions its state from a light-load to over-load condition in an abrupt manner, which may be a result of sudden increase in new client requests, the already established contracts in all service classes must be re-negotiated to the minimum quality in each class. The reason for this is that during light-load conditions maximum resources are allocated to each client. When the system already anticipates overload conditions beforehand, some appropriate re-negotiations times must be chosen.

In general, during the over-load case, termination of some contracts may need to be made in order to create contracts with higher level service classes. Referring to Table 4.1, in order to create a contract with one premium client, 7 normal clients must be terminated. If the utility functions define contract termination costs, it can be calculated what benefits the service provider most. But, this may create bad impressions to certain groups of clients and thus, the parameters that determine termination should not solely be made in terms of monetary benefits (unless the utility values capture all relevant parameters). This is one interesting area that needs further research.

One problem to be solved in the overload case is how to choose the proportions of the various service classes. Different alternatives (strategies) can be taken when mixing the two service classes (e.g. in Table 4.1) under over-load conditions. One option is to select as many clients as the provider could support from a higher priority service class (e.g. premium having higher priority over normal) and when there is enough resource, negotiate with the lower priority clients. This is a strict priority policy. However, in order to be responsive to clients of all classes, a different policy may be followed, where a certain percentage of the resources may be allocated to each class. As an example, 80% of the (bottleneck) resource can be allocated for the higher-priority class and the rest for the lower-priority class. The third possibility is to use utility functions for each service class and choose proportions that would maximize the total utility [Levy et al., 2003]. The percentage in the second policy can be improved through time until a near-optimal solution is found (which is equivalent to analytical means). Owing to the difficulty in defining utility functions that incorporate the effects of all relevant parameters, we recommend choosing the proportions of various classes through experiments of past measurements.

4.1.3.2 Policy Constraints

There are certain behaviors in our system model that cannot be captured in either the utility functions or the negotiation goals. We model these behaviors by policy constraints. Some policies have already been suggested in the previous sub-section, in the case of percentage allocation of resources to each service class.

In this thesis, we define a *negotiation policy* as an explicit representation of the desired behavior of the system during the process of selecting concrete contracts of collaborating components and also the contract (which is also known as SLA) between the user and the service provider. Each negotiation policy is represented in the form of policy constraints and included to our model that already incorporates other constraints (user's, conformance, and resource) for the single-client case. The negotiation policies are assumed to be specified by a *Negotiation Expert*. This is a new role besides the already existing ones: component developer, container developer, etc.

The different policies that need to be defined and incorporated in the negotiation process concern the following areas.

- How to make a choice on the proportions of the various service classes to negotiate during over-load conditions? This can be something to be experimentally re-adjusted so as to reflect the service provider's negotiation goals.
- How to move from one quality level into another when the system gradually transitions from light-load to over-load conditions? The various options are: (i) not to change levels by choosing the minimum quality from the start, (ii) choose maximum first then at some point choose minimum, and (iii) go through all (or part of) the available levels starting from maximum to minimum. A choice of the particular option is made by weighing the cost and benefit of switching levels for the specific application.
- How to favor clients of the same class when re-negotiating contracts? This case is relevant when some resources are released and re-negotiations can be done so as to increase the quality levels of existing contracts. In this case, the resource availability is such that it is not enough to offer maximum quality to all clients.

4.1.3.3 Algorithm

The following algorithm is based on the discussion in subsections 4.1.3.1 and 4.1.3.2 together with the negotiation algorithms developed for the single-client scenario.

Algorithm 4.1: Negotiation algorithm between multiple clients and a server

```
–   Clients send requests for a service.
–   Server container performs policy constraint checks (subsection 4.1.3.2)
    if there is a need to reallocate resources to active contracts.
–   Server allocates resources to the new clients (subsection 4.1.3.1).
–   Server makes a one to one negotiation with the new clients and also with
    existing clients whose contracts have to be renegotiated using Algorithm
    3.3.  This   would   result   in   establishing   contracts   between   the
    collaborating components and between the user and service provider.
–   If there are clients waiting in the queue, the server proposes an offer
    that contains the quality and the maximum waiting time to these clients.
    If any client would not accept the offer, it is rejected. All clients
    that cannot be allocated resource are rejected.
```

4.2 Multi-tier Scenario

In a multi-tier scenario, a component requires the service of another component located in a separate server container in order to give the requested service to its clients. The first server component is acting as a client to the second server. Figure 4.6 shows this scenario where *B* requires the service provided by *C* in order to be able to give its service to *A*. Our *single-client – single-server* scenario already discussed in Section 3 is general enough for the chain of components deployed in a single container. The main difference that lies in the multi-tier scenario is that the chains of components are deployed in different containers.

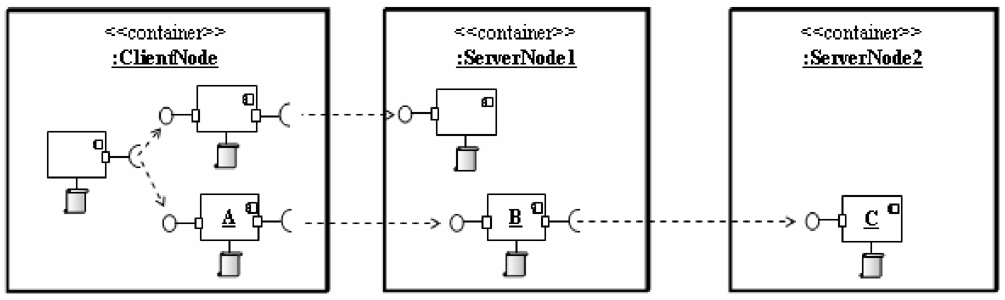


Figure 4.6: A multi-tier scenario

4.2.1 Example

We use a Stock Quote Application (Figure 4.7) to motivate the generalization of our approach to the multi-tier scenario. In this application, stock quotations are monitored by customers/brokers. Stock Quote is a small part of the Stock Trading Application. In stock trading, orders need to be issued to buy or sell shares.

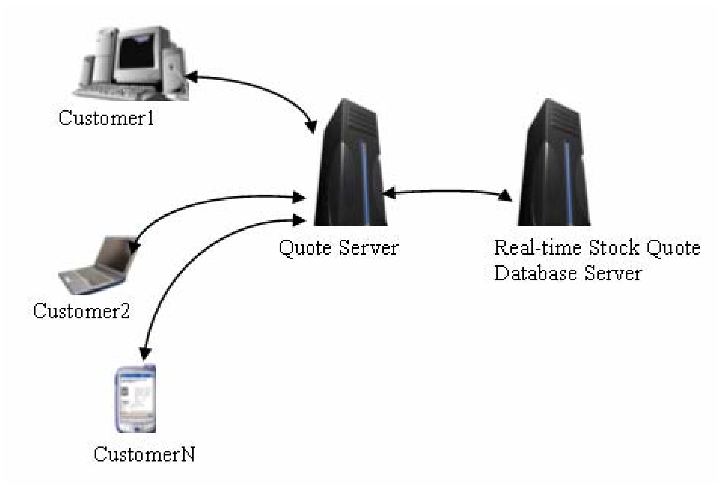


Figure 4.7: A simplified architecture of a stock quote application

As shown in Figure 4.7, the customers/brokers get stock quotes from the quote server. They may have different requirements on the QoS of the delivered service. Some of the interesting NFPs in this application are: (i) the rate at which the stock quotes are delivered, (ii) the response time of the delivered service, and (iii) the stock quote symbols for which the service has to be provided. If the service is available with a price, different customers are willing to pay different values. The service may be delivered with rate updates of 1 sec or less (this is real-time delivery), every 1 minute (a one minute delay), every 15 minutes (a fifteen minute delay), etc.

The application can be implemented to use different communication models such as request/reply (polling) and publish/subscribe models. When comparing the two models, request/reply could lead to higher bandwidth usage as it can saturate the server and the network by making many requests, even if the value of the stock has not changed. But, such unnecessary saturations could be minimized if the server and the customer had made prior agreements on each other's requirements and expectations. This can be achieved with our contract negotiations of the components at the client and server.

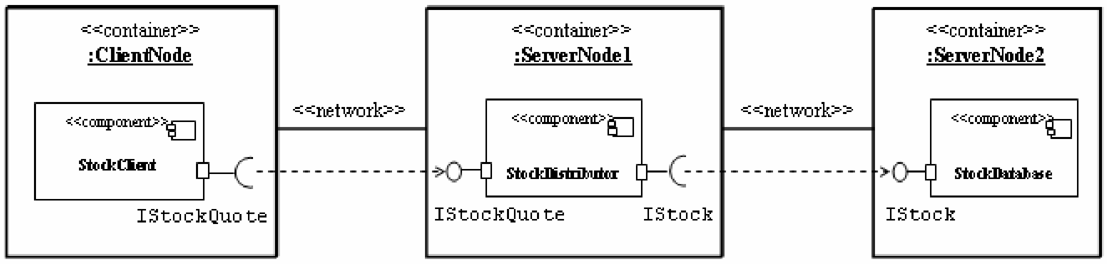


Figure 4.8: Components deployed in three different nodes for a stock quote application

The Stock Quote Scenario we discuss involves three components; *StockClient*, *StockDistributor*, and *StockDatabase* as shown in Figure 4.8. We will next describe these components and specify their non-functional properties at the various interfaces.

StockDatabase

The *StockDatabase* component is installed on a node where the database server is located. This component provides access to the stock database and guarantees a certain response time when providing this service to its clients. Whenever there are changes in the stock prices (which can happen at arbitrary time), the pertinent values are updated to the *StockDistributor*. Various models can be used for the update. Either the *StockDistributor* polls the *StockDatabase* at different frequencies using the request/response model; or the *StockDatabase* pushes the changes to *StockDistributor* using its event interface. Even if *StockDatabase* handles the updates of a number of stock symbols, the agreement with *StockDistributor* could be only on a certain proportion of the total number of stock symbols. The total number of stock symbols used in the update affects the resource requirement as shown in Table 4.2. Let's arbitrarily assume that *StockDatabase* is responsible for updates of a total of 30 stock symbols (e.g. IBM, MSFT, GM, YHOO, etc.).

In Table 4.2, it is assumed that *StockDatabase* updates stocks every 150ms. In reality, all stock prices might not change at this interval. Nonetheless, we take the worst case scenario. In order to specify the QoS-Profiles of each component in Figure 4.8, some measurements or analytical methods must be employed. We don't claim to have a methodology for precisely specifying the profiles as this is out of the scope of this thesis. Nevertheless, we will suggest some techniques for computing the throughput requirements of the components. To calculate the throughput, let's assume that (i) for each update a data volume of 1KB is transported per stock symbol, and (ii) all the required data is to be transported within 100msec at every update. The bandwidth demand in Table 4.2 is computed based on these assumptions. The specification in Table 4.2 assumes that updates to *StockDistributor* can be agreed in a stack of 10 symbols or 30 symbols. Nevertheless, when the need arises, *StockDistributor* and *StockDatabase* may negotiate on update of a single symbol. For the sake of completeness, Table 4.2 may need to incorporate specifications for every symbol.

StockDatabase			
	`Stock symbol`	`provides IStock` (response time in msec, update frequency in sec^{-1})	`Resource` (CPU in %, bandwidth in Kbps, memory KB)
1.	1st to 10th	10, 150	50%, 800, 40
2.	11th to 20th	10, 150	50%, 800, 40
3.	1st to 30th	15, 150	80%, 2400, 60

Table 4.2: QoS-Profiles of *StockDatabase*

StockDistributor

StockDistributor gets the required data from *StockDatabase* and distributes its service to clients at different delays, for example, starting from real-time to delays in minutes and hours. *StockDistributor* also expects a certain response time from *StockDatabase*. The *StockDistributor* can make agreements with its clients for a periodic update of the stocks, for example, every 100ms, every second, every minute, etc. The update frequency of 100ms is effectively a real-time update. The *StockClient* may be interested in one or more stock symbols. As in the case of *StockDatabase*, the resource demand shown in Table 4.3 depends on the number of requested stock symbol updates.

The resource demand in Table 4.3 has been specified for the two interfaces (`IStock` and `IStockQuote`) separately. The distinction is made concerning the bandwidth requirement. The CPU and memory demands are shown to be the same, i.e. some maximum requirement is taken when *StockDistributor* is engaged in receiving updates for 10 symbols and distributing updates for a single stock symbol. It is assumed that the update data to be transported to a client is at 100ms, 150ms, 1000ms, and 1000ms for update frequencies of 120min^{-1}, 20min^{-1}, 1min^{-1}, and 0.05min^{-1} respectively. The total number of QoS-Profiles to be specified for *StockDistributor* would have been much larger if each and every combination of properties at the two interfaces were to be listed.

StockDistributor			
uses IStock (response time in msec)	Provides IStockQuote (response time in sec, invocation frequency in min^{-1})	Resource (IStock) (CPU in %, bandwidth in Kbps, memory KB)	Resource (IStockQuote) (CPU in %, bandwidth in Kbps, memory
20	80, 120	5, 800, 20	5, 80, 20
20	100, 20	5, 800, 20	5, 53, 20
20	500, 1	5, 800, 20	5, 8, 20
20	500, 0.05	5 800, 20	5 8, 20

Table 4.3: QoS-Profiles of *StockDistributor*

StockClient

The *StockClient* needs to connect to *StockDistributor* and get periodic updates of stock values for one or more symbols. *StockClient* captures various expectations of stock updates as required by customers. This is shown in Table 4.4. The same assumption holds as in Table 4.3 as far as the bandwidth requirement is concerned.

The QoS provided by *StockDistributor* could depend on the way its clients use the service. For example, the throughput that *StockDistributor* can provide may depend on how frequently a client calls the service. In order to account for this dependency, a QoS contract needs to include requirements and properties for both clients and service providers. This implies that the *StockClient* and *StockDistributor* components should incorporate the amount of invocation to be provided (for *StockClient*) and to be expected (for *StockDistributor*). This is achieved by considering the update/invocation frequency property as described in Table 4.3 and Table 4.4.

StockClient			
	Stock symbols	uses IStockQuote (response time in msec, invocation frequency in min^{-1})	Resource (CPU in %, bandwidth in Kbps, memory KB)
1.	1st symbol	150, 120	5, 80, 20
2.	1st symbol	200, 20	5, 53, 20
3.	1st symbol	1500, 1	5, 8, 20
4.	1st symbol	1500, 0.05	5, 8, 20
5.	1st to 10th symbols	150, 120	10, 800, 80
6.	1st to 10th symbols	200, 20	10, 530, 80
7.	1st to 10th symbols	1500, 1	10, 80, 80
8.	1st to 10th symbols	1500, 0.05	10, 80, 80

Table 4.4: QoS-Profiles of *StockClient*

Now, let's examine some negotiation scenarios in the componentized stock quote application (Figure 4.8). The *StockClient* sends requests to *StockDistributor* for a quote on one or more stock symbols at a particular update frequency (for example, every 500ms) and for a certain response time (for example, 150ms). If data about the requested stock symbols is already available at *StockDistributor*, this case is reduced to the *single* *client – single server* scenario that we have addressed in subsection 3.4. But, if the requested data is not available at *StockDistributor*, the negotiation must include all three component containers. In this section, we particularly are interested on how to handle the QoS contract negotiations when more than two containers are involved in the negotiation.

4.2.1.1 Discussion

It is insightful to compare our componentized stock quote application scenario with implementations of the same application scenario that use a different infrastructure. The stock quote application scenario has been persistently used in [Schmidt and Vinoski, 2004][Natarajan et al, 2004] and in several of their columns to demonstrate how applications are developed with the CORBA Component Model (CCM). CCM is a component middleware that addresses limitations with earlier generations of CORBA 2.x Distributed Object Computing (DOC) middleware.

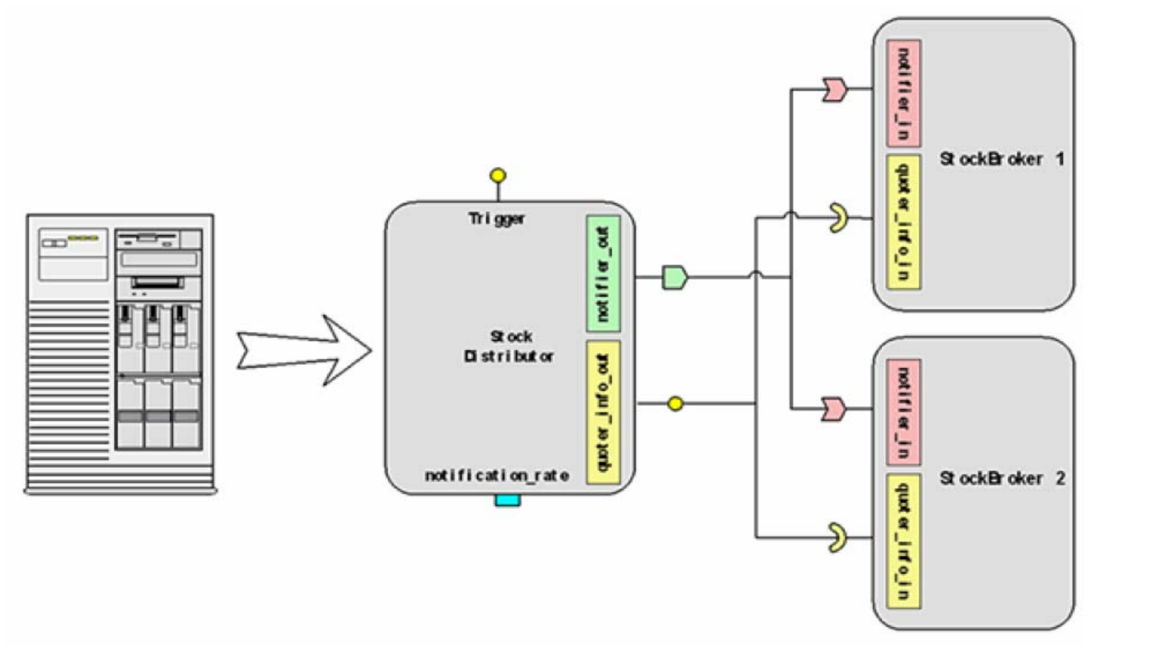


Figure 4.9: StockBroker and StockDistributor component ports

The component diagram in Figure 4.9 illustrates how the stock quote application can be designed using CCM [Schmidt and Vinoski, 2004]. This application operates as follows. The *StockDistributor* component publishes events to indicate that a particular stock's value has changed. This component will monitor the real-time stock database and, when the values of particular stocks change, will push a CCM *eventtype* containing the stock name via a CCM event source (`notifier_out`) to the corresponding CCM event sink (`notifier_in`) of one or more *StockBroker* components. The *StockBroker* components that consume this event will then examine the stock name stored in the event. If they are interested in the stock, they can invoke a request/response operation via their CCM receptacle (`quoter_info_in`) on a CCM facet (`quoter_info_out`) exported by the *StockDistributor* component to obtain more information about the stock. The `notification_rate` attribute is used to control the rate at which the *StockDistributor* component checks the stock-quote database and pushes changes to *StockBroker* subscribers.

Both our scenario and the one realized with the CCM components in Figure 4.9 follow the CBSE principles. But, the basic difference of the two approaches lies in the handling of NFPs (QoS contracts). The components in Figure 4.9 are specified with syntactic contracts only. In our case components' QoS contracts (NFPs) are specified in addition to the syntactic contracts. The incorporation of QoS makes the component specification more robust and brings about a number of advantages to the application. But, it is also a challenging task. Some of the differences between our scenario and the one in Figure 4.9 are listed below.

i.) The ports of the components in Figure 4.9 can be directly connected. The CCM deployment and configuration (D&C) tools can alleviate the need for developers of application components to perform connection "plumbing" programmatically [Schmidt and Vinoski, 2004]. But, the D&C doesn't perform QoS contract negotiations. In our case, the connection of components' ports must be performed by a QoS contract negotiation process.

ii.) A differentiation of QoS of the provided service can be achieved in our approach. This is made possible by the robust specification of components and the automatic run-time QoS contract negotiation process we proposed. In the application in Figure 4.9, such QoS differentiation isn't possible. There is only a very limited flexibility in this regard. The system administrators can use the `notification_rate` attribute to control the rate at which the *StockDistributor* component checks the stock-quote database and pushes changes to *StockBroker* subscribers.

iii.) In our approach, the components can be deployed in environments with different resource availability conditions and they perform their functions properly. But, the application in [Schmidt and Vinoski, 2004] cannot cope with such changes.

4.2.2 QoS Contract Negotiation in a Multi-tier Scenario

In order to handle negotiations among three or more containers, we can extend the algorithm proposed for the two containers case (Sections 3.5 and 3.6). Three groupings of components have been identified for the two containers case, as shown in Figure 3.11. The three groups, which have been classified based on the resource constraint relation, are components deployed in the client container, components connected across containers, and components deployed in the server container. By the same token, more groupings can be created when there is more chaining of containers. For instance, the following five groupings are identified for the three containers (Figure 4.8).

i.) Components deployed in clientNode (e.g. *StockClient*),

ii.) Components connected across clientNode and serverNode1 (e.g. *StockClient* and *StockDistributor*),
iii.) Components deployed in serverNode1 (e.g. *StockDistributor*),
iv.) Components connected across serverNode1 and serverNode2, (e.g. *StockDistributor* and *StockDatabase*) and
v.) Components deployed in serverNode2 (e.g. *StockDatabase*)

Algorithm 3.2 can be systematically modified as in Algorithm 4.2 to handle negotiations on the three containers where the deployed components have been categorized into five groups. As can be seen from Algorithm 4.2, the modification applied on Algorithm 3.2 is only the inclusion of lines 19-26 in the `FindAppropriateProfiles()`. The whole approach can be easily generalized to cases that involve more than three containers in the chain.

Algorithm 4.2: Fine-grained negotiation algorithm for components deployed in a client and two-tiered server containers

```
   // constants used as return types
      public static final int NO_RESOURCE = 0;
      public static final int SUCCESSFUL = 1;

      enum CG { ON_CLIENT, ON_SERVER, ACROSS_CONTAINERS } // CG is short for component group
 5
      boolean FineGrainedNegotiationMultiTier()
      {
          // the same code as FineGrainedNegotiation() in Algorithm 3.2
      }
10
      int FindAppropriateProfiles()
      {
        FindConformantProfiles(CG.ON_CLIENT);
        if(CheckResourceConstraints(CG.ON_CLIENT)) {
15          FindConformantProfiles(CG.ACROSS_CONTAINERS1&2\CG.ON_CLIENT);
            if(CheckResourceConstraints(CG.ACROSS_CONTAINERS1&2)) {
                FindConformantProfiles(CG.ON_SERVER1\CG.ACROSS_CONTAINERS1&2);
                if(CheckResourceConstraints(CG.ON_SERVER1)) {
                    FindConformantProfiles(CG.ACROSS_CONTAINERS2&3\CG.ON_ SERVER2);
20                  if(CheckResourceConstraints(CG.ACROSS_CONTAINERS2&3)) {
                        FindConformantProfiles(CG.ON_SERVER2\CG.ACROSS_CONTAINERS2&3);
                        if(CheckResourceConstraints(CG.ON_SERVER2))
                            return SUCCESSFUL;
                        else return NO_RESOURCE;
25                  }
                    else return NO_RESOURCE;
                }
                else return NO_RESOURCE;
            }
30          else return NO_RESOURCE;
        }
        else return NO_RESOURCE;
      }
```

4.2.3 Consideration of Security Properties

In Figure 4.10, we extend the video streaming scenario (Figure 3.13) to include a payment service provider in the process of video streaming and payment transactions. The *BookingMgr*, *MovieMgr* and *CustomerMgr* components are deployed in the same server as *VideoServer*. The approach we proposed for a multi-tier scenario can be applied when the components are deployed in separate servers.

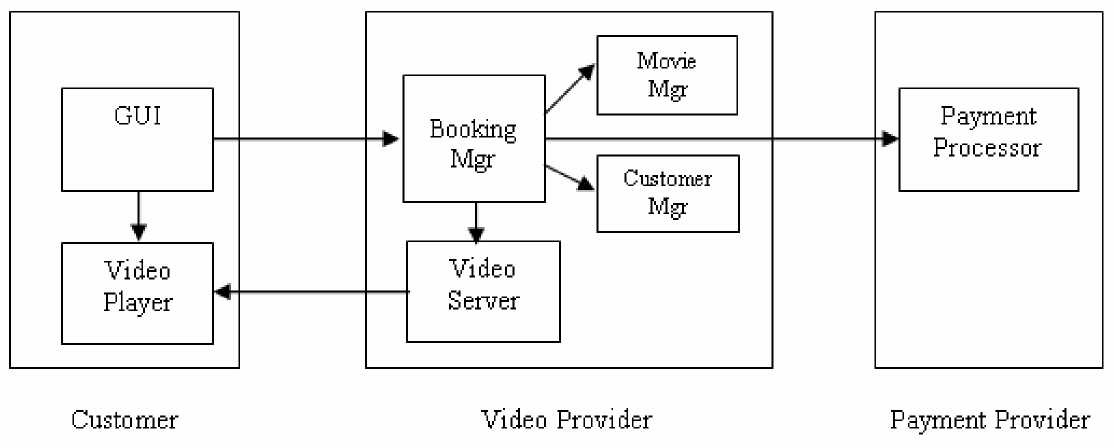


Figure 4.10: Interaction of Components in a Video Streaming Application that includes Billing

The *BookingMgr* component receives, through *GUI*, customer's request and performs a number of operations. The customer provides information about the title of the movie, his credential (e.g. name, ID, etc.), the payment method, etc. *BookingMgr* contacts *MovieMgr* to get information about whether the required movie exists or not. If it exists, *MovieMgr* returns the amount of charge, the mode of payment, etc. to *BookingMgr*. At the same time *BookingMgr* contacts *CustomerMgr* to get information about the customer's credit card information. Then *BookingMgr* sends the customer's credit card number to *PaymentProcessor* for the latter to process the payment. After successfully transacting the payment, the requested movie will be streamed from *VideoServer* to *VideoPlayer*.

In this section, we concentrate on the use case *"order service"*, which involves the handling of a user's request for a movie up to the payment process. In order to simplify our analysis, we combine *BookingMgr*, *MovieMgr*, and *CustomerMgr* into one component *Booking*, as shown in Figure 4.11.

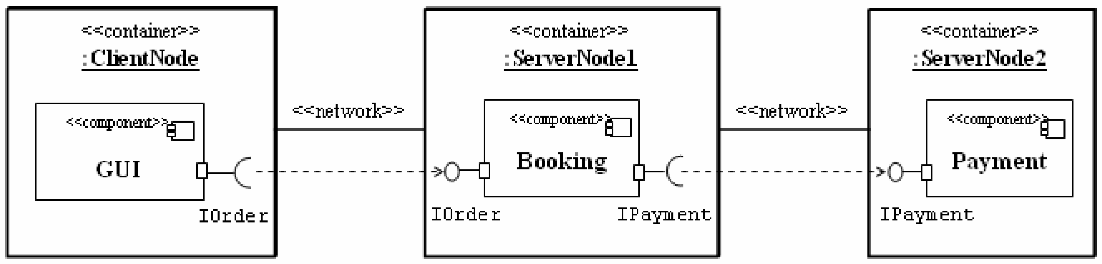


Figure 4.11: Collaborating components for the "order service" use case in the Video Streaming Application that includes billing

We considered in the previous application scenarios only performance related QoS properties like response time or frame rate. For the "order service" use case, the cooperating components and a user have security requirements. One such requirement by a user is the confidential transmission of his/her credit card number to the payment processor. We can assume this communication to be implemented with the help of encryption techniques. In addition to security, the user may have performance requirements as well.

As indicated earlier, the QoS contract specification language that we apply in the negotiation is CQML$^+$. But, this language has no features for specifying security *offer* and *expectations* of components. To handle the contract negotiation for the "order service" use case, we have to define the QoS-Profiles of the components with our own extension of the CQML$^+$ to represent the security properties of components. For this extension, we applied the concepts of a component's *required* and *ensured* security properties from [Khan, et al, 2000]. According to [Khan, et al, 2000], a *required* security attribute is an invariant in a sense that it is the required property of a component that other interested parties must satisfy during the composition according to the contract. It is a precondition the component must ensure that the security attribute is provided, and its validity is ensured. Similarly, an *ensured* security attribute is a post-condition in a sense that it is the responsibility of the component to maintain the committed security *assurance* during the composition of the contract.

In Figure 4.11, a user expects his/her performance and security requirements to be fulfilled through *GUI*, the front-end component. When *GUI* is implemented, it is specified to operate in different QoS-Profiles. For its service, *GUI* relies on *Booking*, which in turn requires a service from the *Payment* component. Each of the components is specified in terms of their NFPs. The *Booking* component implements two interfaces: a uses interface `IPayment` (to get a service from *Payment*) and a provides interface `IOrder` (to offer the required service to *GUI*). Likewise, the *Payment* and *GUI* components implement a provides interface `IPayment` and a use interface `IOrder` respectively.

The specified profiles of components in Figure 4.11 are shown in Table 4.5. PP^{-} is the payment provider's private key whereas *Shared* is a particular shared secret key between the video and payment providers. RSA, DES, and 3DES are encryption algorithms. Only three profiles have been shown. In reality, the number of specified profiles could be much more than that, for example when additional keys, key length, and algorithms are considered in the encryption/decryption.

Note that the resource demands of each profile haven't been shown in Table 4.5. One possible approach to approximately specify the resource demands is to extend the technique described in [Meyerhöfer and Neumann, 2004] for measuring response time. This measurement will be done under particular security settings of the components. Performance monitoring tools (for example, the one integrated in the Windows Operating System) can be used to estimate the average resource requirements while the response time is being measured. A precise specification of components is beyond the scope of this thesis.

GUI	Booking		Payment
`uses IOrder (response time in sec)`	`provides IOrder (response time in sec)`	`uses IPayment (key, algorithm, response time in sec)`	`provides IPayment (key, algorithm, response time in sec)`
7	13	PP$^-$, RSA, 13	PP$^-$, RSA, 10
9	7	Shared, 3DES, 7	Shared, 3DES, 5
18	5	Shared, DES, 5	Shared, DES, 3

Table 4.5: QoS-Profiles of GUI, Booking, and Payment implementations

When performing contract negotiation on the three distributed components (Figure 4.11), one of the issues to resolve is the conformance of selected profiles. Conformance can be defined both on coarse-grained and fine-grained properties. [Franz and Pohl, 2004] have suggested a matrix for the matching of security goals (coarse-grained) but haven't discussed concerns related to the security mechanisms (fine-grained properties) implemented by components. Conformance on fine-grained security properties exist when the constraints of the interacting components match exactly on all the parameters (*key type*, *key length*, and *algorithm*). For matching offer and expectation on *response time*, the "stronger than" relationship described in subsection 3.3.2 is used. This must take into account the transmission delay between the customer's workstation and video provider's server (for *GUI* and *Booking*) and between the video provider's server and the payment provider's server (for *Booking* and *Payment*).

One interesting issue is whether there should be order in the negotiation of the NFPs, for example security being negotiated before performance (response time). In our earlier discussion, we followed order of negotiation between coarse-grained and fine-grained properties. For e.g., negotiations on whether or not to do confidential communication must precede negotiation on particular security mechanisms. Or, negotiation on protocol properties of components in video streaming application must precede frame rate negotiation. The strategy we have been following for fine-grained properties (e.g. response time, frame rate, resolution) has been to do negotiation of all properties in an atomic manner, i.e. without any order, if the distributed components are specified with all these properties. If certain properties were specified only for components deployed in one of the containers, ordering of negotiations could be attained on the fine-grained properties. The QoS specifications languages we are aware of do not have features that enable the application of the different negotiation strategies (e.g. ordering of negotiation). But, our contract negotiation framework supports the use of different negotiation strategies.

One requirement in this scenario, however, could be to do negotiation first on security attributes and only when this is successful, to proceed to performance negotiations. There exists usually a dependency between NFPs of a component. The *response time* property among others is influenced by the security attributes selected. If for e.g. the *key length* in the encryption is large and some sophisticated algorithm is used, the *response time* will be higher. That is why the profiles specified for the components define security and performance properties together. Thus, if we followed the strategy of ordering of negotiation on security and response time, the result of the first would influence the outcome of the second.

Next let us see how we can adapt the two container solutions to the multi-tiered case. In the centralized solution, it is one of the containers that performs the contract negotiation. The other containers must provide information about QoS-Profiles of hosted components and their resource conditions. For the *"order service"* use case discussed above, let us assume that the responsible container is the one in the video provider. The containers in the customer and payment provider must send the QoS-Profiles of *GUI* and *Payment* components, and the available CPU, memory, etc. of each node. Data on end-to-end available bandwidth and end-to-end delay must be obtained by the responsible container.

The selection of QoS-Profile of *Payment* (similar to other components) is determined by: (i) the selected QoS-Profile of Booking, (ii) the network bandwidth available for the connection of Payment and Booking, and (iii) the resource availability at the payment provider node. The QoS contract negotiation proceeds by making use of Algorithm 4.2. Finally relevant agreed contracts are sent to each container. Comparisons between the centralized, distributed, and hybrid solution made for two container case are valid for applications with a higher number of containers.

5 A QoS Contract Negotiation Framework

A general contract negotiation framework is needed so that container developers can tailor it to fit their purpose. Here the framework can be seen as a reusable design of a contract negotiation system in which the design consists of the representation of important active components, data entities, and the interaction of different instances of these. Figure 5.1 shows the conceptual architecture of our framework represented as a UML class diagram.

5.1 Architecture

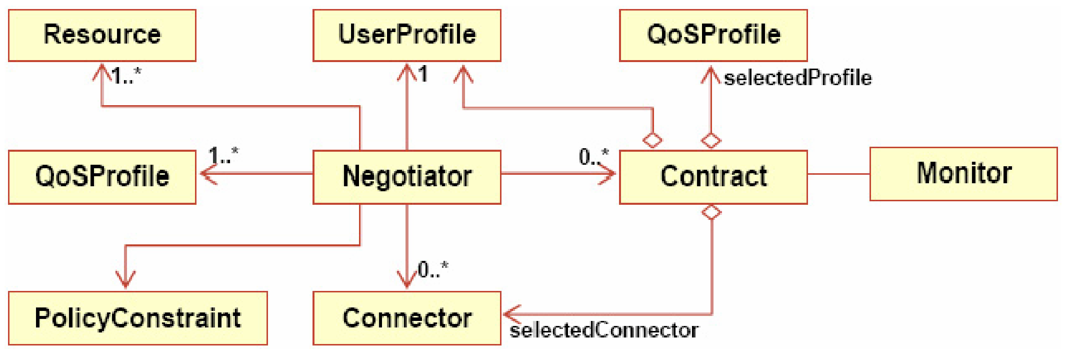


Figure 5.1: Architecture of a Contract Negotiation Framework

Negotiator coordinates and performs the contract negotiation on behalf of the interacting components. In order for *Negotiator* to decide on the solution (i.e. selection of appropriate concrete QoS contracts at the ports of components), it has to make reference to: (i) the QoS specification of all the cooperating components, which is assumed to be available in the form of one or more QoS profiles, (ii) user's QoS requirement and preferences, (iii) available resource conditions, (iv) network and container properties, and (v) policy constraints. After a successful negotiation, *Negotiator* establishes contracts, which will have to be monitored and enforced by the container. Next, we describe the different building blocks of our framework.

QoS Profile

Component's QoS Contracts are specified with one or more QoS profiles. A component's QoS contract is distinguished into *offered QoS contract* and *required QoS contract* [OMG, 2005]. As indicated earlier, we use CQML$^+$ [Röttger and Zschaler, 2003][Göbel et al, 2004a], an extension of CQML [Aagedal, 2001], to specify the offered- and required-QoS contract of a component. CQML$^+$ uses the *QoS-Profile* construct to specify the NFPs (provided and required QoS contracts) of a component's implementation in terms of what qualities a component requires (through a *uses* clause) from other components and what qualities it provides (through a *provides* clause) to other interacting components, and the resource demand by the component from the underlying platform (through a *resource* clause). The *uses* and *provides* clauses are described by a QoS statement that constrain a certain quality characteristic in its value range. A simplified example shown below depicts these elements.

```
QoSProfile aProfile for C {
      provides providesClause;
      uses usesClause;
      resources resourcesClause;
}
```

It is assumed that the component developer specifies the QoS-Profiles after conducting experiments and measuring the provided quality, required quality, and the resource demands at the component level.

Connector

Connector is an abstraction of the network and the containers that exist between interacting components deployed on multiple nodes. A communication channel may have a number of QoS properties. For example, it introduces a delay. The connector properties are used when matching conformance between provided- and required-QoS contracts of components interacting across containers.

It is assumed that the values of the connector properties are available to Negotiator before negotiation starts. Two possible approaches for estimating the values are: (i) Off-line measurement - the required properties are measured offline by applying different input conditions (e.g. throughput) and load conditions in the network and end-systems; and (ii) On-line measurement - the properties are measured during the application launch and/or at run-time.

User Profile

UserProfile is used to specify the user's QoS requirements and preferences. The user's requirement may be specified for one or more QoSdimensions. Additional parameters such as user class need to be defined when considering, for example, a multiple-clients scenario. UserProfile is assumed to be constructed by the run-time system after obtaining the user's request for a given service. The user might be given the chance to select attributes from one of many templates supplied for the application or specify the attributes himself.

Resource

Resource is used to store information about the available resources at the nodes and the end-to-end bandwidth between nodes in which components are deployed. Monitoring functions are used to supply data about a node's load conditions on CPU, memory, etc. It is assumed that the available resources are monitored at run-time. Changes in available resources might initiate renegotiation.

Negotiator

A user's request to get a service is first intercepted by the *Negotiator* on the client node. The *Negotiator* at the client and server side exchange information about the required service and the QoS requirement of the user before the negotiation begins. *Negotiator* is also responsible for selecting appropriate QoS-Profiles of the interacting components that should satisfy a number of constraints (e.g. user's, resource, etc.). It is also responsible for finding a good solution from a set of possible solutions. *Negotiator* creates *Contract* after successfully performing the negotiation. For an unsuccessful negotiation, the selection process is repeated by relaxing the user's QoS requirement.

In order to accomplish the stated responsibilities, *Negotiator* relies on our modeling of the QoS contract negotiation as a Constraint Satisfaction Optimization Problem (CSOP) [Tsang, 1993]. A CSP consists of variables whose values are taken from finite, discrete domains, and a set of constraints on their values. The task in a CSOP is to assign a value to each variable so that all the constraints are satisfied and a solution that has an optimal value with regard to the objective function is found. The objective function maps every solution to a numerical value.

In the above modeling, we take the variables to be the QoS-Profiles to be used for the collaborating components. The domain of each variable is the set of all QoS-Profiles specified for a component. The constraints identified are classified as conformance, user's, and resource. As an objective function, we use an application utility function [Lee et al, 1999], which is represented by mapping quality points to real numbers in the range [0, 1] where 0 represents the lowest and 1 the highest quality.

Contract

The creation of contracts proceeds after the selection of appropriate concrete QoS-profiles of the interacting components. Contracts may exist between components deployed in the same or different containers. In the case of a front-end component, a contract exists between this component and the user. A simplified abstraction of *Contract* is given below.

```
public class Contract {
      QoSProfile selectedProfileClient;
      QoSProfile selectedProfileServer;
      Connector selectedConnector
      UserProfile userProfile;
      double contractValidityPeriod;
      //…
};
```

If the contract is made between two components deployed in the same container, the clauses of the contract should contain the QoS offers and needs as well as the resource demands of the components. That means, the selected QoSprofiles of the client and server components would be clauses in the contract (in this case, `selectedConnector` and `userProfile` are null). If the contract is made between components across containers, a selected connector is also part of the contract. For a contract between a user and the front-end component, a user's profile would become part of the contract (`selectedProfileClient` and `selectedConnector` are null in this case). Note that resources required from the underlying platform are included in the contract through the QoS-profiles. Additional parameters such as contract dependencies, etc. also need to be defined in the contract in order to facilitate contract monitoring and enforcement.

Monitor

After contracts are established, they can be violated for a number of reasons like a shortage of available resources. *Monitor* constantly monitors contracts to assure that no contract violations would occur and in case one occurs, some corrective measures should be taken through contract re-negotiations.

Policy Constraints

As described previously, *Negotiator* uses a CSOP framework to find good solution. The CSOP framework in turn relies on the specification of constraints and a utility function in order to find appropriate solutions. There are, however, certain behaviors that cannot be captured in utility functions. Such behaviors are modeled by *PolicyConstraint*, which can be defined as an explicit representation of the desired behavior of the system during contract negotiation and re-negotiation. *Negotiator* can achieve, for instance, different optimization goals based on varying specifications in the policy constraints. For e.g., the service provider might want to allocate different percentages of resources to different user classes (e.g. premium and normal users).

5.2 Interaction

The interaction diagram in Figure 5.2 depicts an overall view of the negotiation process between the Negotiator's on client and server. It assumes that the Negotiator on the server side has information on the QoS contracts of all components that reside on client and server containers. This is a centralized approach of contract negotiation. Figure 5.3 shows a distributed approach of the contract negotiation process as an activity diagram to clearly show the concurrent tasks in the client and server containers. Figure 5.3 assumes that the local contract negotiation is performed by the respective containers unlike the case in Figure 5.2 where the local contract negotiations are made centrally on the server container.

In Figure 5.2 the following is illustrated (note that the figure depicted a successful negotiation scenario):

- A user requests the application for a service by providing the service's name (e.g. playing a given movie or performing payment for usage of a particular operation) together with his/her QoS and preference needs, i.e. UserProfile (step 1).
- The container, upon receiving the user's requirement, identifies which components need to participate in order to provide the required service (step 2).

- In the centralized approach (which can be done by either client or server container), the container which is responsible for the negotiation, must collect the QoS contracts of all collaborating components and resource conditions at each node and the network (step 3).
- The responsible container performs the negotiation (step 4) in two phases. In the first phase, negotiation is made on coarse-grained properties (step 5). When this is successful, negotiation on fine-grained properties continues (step 6).
- The responsible container creates all contracts. A contract is established between any two interacting components (step 7).
- The client container retrieves relevant contracts from the server container (step 8). These are contracts between components deployed in the client container or between components connected across containers.

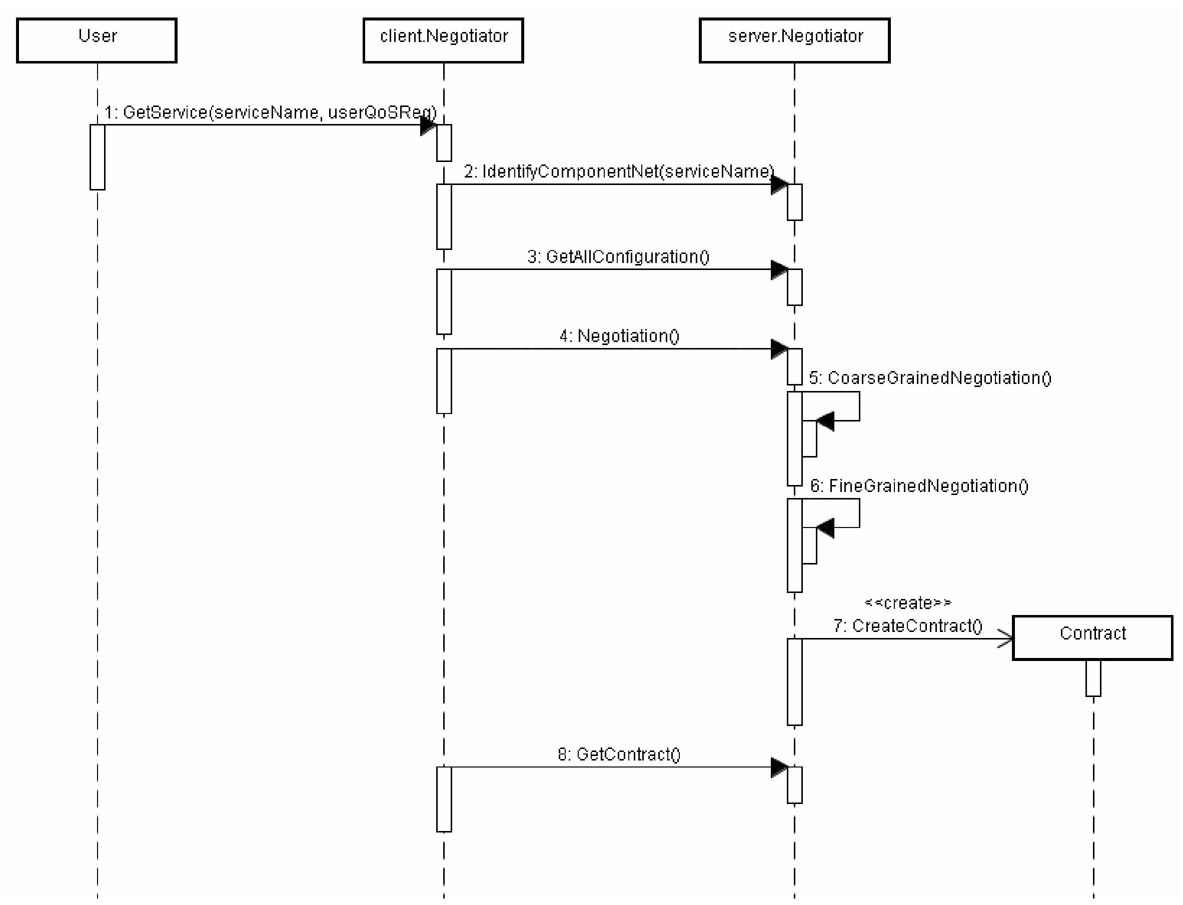


Figure 5.2: Interaction between client and server containers (Centralized Approach)

Figure 5.3 shows the interaction between the client and server negotiators in a purely distributed solution. The different actions are explained as below.

- A user sends a request for a particular service (step 1). Upon receiving a request, the client and server identify the collaborating components deployed in the respective containers (step 2a & 2b). In step 3a & 3b, each container accesses the repository to get the QoS contracts of the components identified in step 2 and also collects information about the local resource conditions.
- Negotiation is performed by each container locally (step 4a & 4b). Note that this step comprises both coarse-grained and fine-grained negotiations.
- In step 5, the client communicates to the server about the selected QoS-Profiles of client components connected across containers. The server checks if these profiles conform to the ones it selected and also if there is enough network bandwidth (step 6). If all the constraints are met, the negotiation will end by establishing contracts (step 13a & 13b). Otherwise, the server repeats the negotiation in view of the information it received from the client (step 7).
- In step 8, the server sends to the client the selected profiles of components that are relevant. The client checks for consistency of all constraints in step 9 and if so control will move to step 13a & 13b to establish contracts. Otherwise more negotiation is done on the client's side (step 10) that takes into account the previous outcome of the server's negotiation. Steps 6 to 12 are repeated until all constraints are satisfied. Finally, in a successful scenario contracts are established among collaborating components (step 13a & 13b).

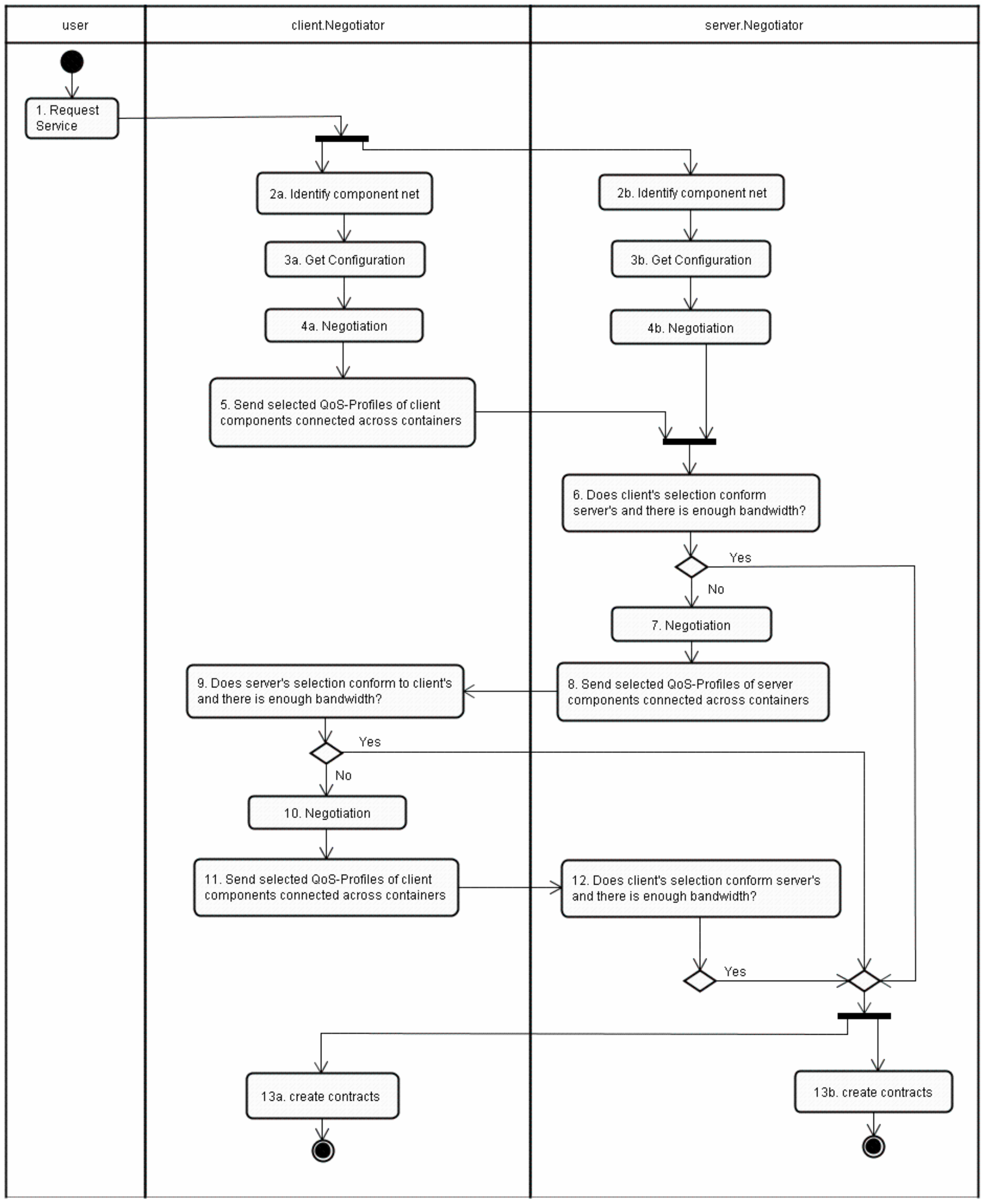

user
client.Negotiator
server.Negotiator
1. Request Service
2a. Identify component net
2b. Identify component net
3a. Get Configuration
3b. Get Configuration
4a. Negotiation
4b. Negotiation
5. Send selected QoS-Profiles of client components connected across containers
6. Does client's selection conform server's and there is enough bandwidth?
Yes
No
7. Negotiation
8. Send selected QoS-Profiles of server components connected across containers
9. Does server's selection conform to client's and there is enough bandwidth?
Yes
No
10. Negotiation
11. Send selected QoS-Profiles of client components connected across containers
12. Does client's selection conform server's and there is enough bandwidth?
Yes
13a. create contracts
13b. create contracts

Figure 5.3: Interaction between client and server containers (Pure-Distributed Approach)

Figure 5.4 depicts one possible interaction between a client and server negotiators in a hybrid solution, which involves both centralized and distributed approaches. In steps 1 & 2, the requested service is communicated to the server. Step 3 identifies the collaborating components. In Step 4, the server obtains all the required metadata about the component and the resource conditions. In step 5, the server performs negotiation on properties defined for all components. Selected QoS-Profiles of components are communicated to the client (step 6). Steps 7a & 7b continue the negotiation with respect to properties specific to the local components. Finally, contracts are established.

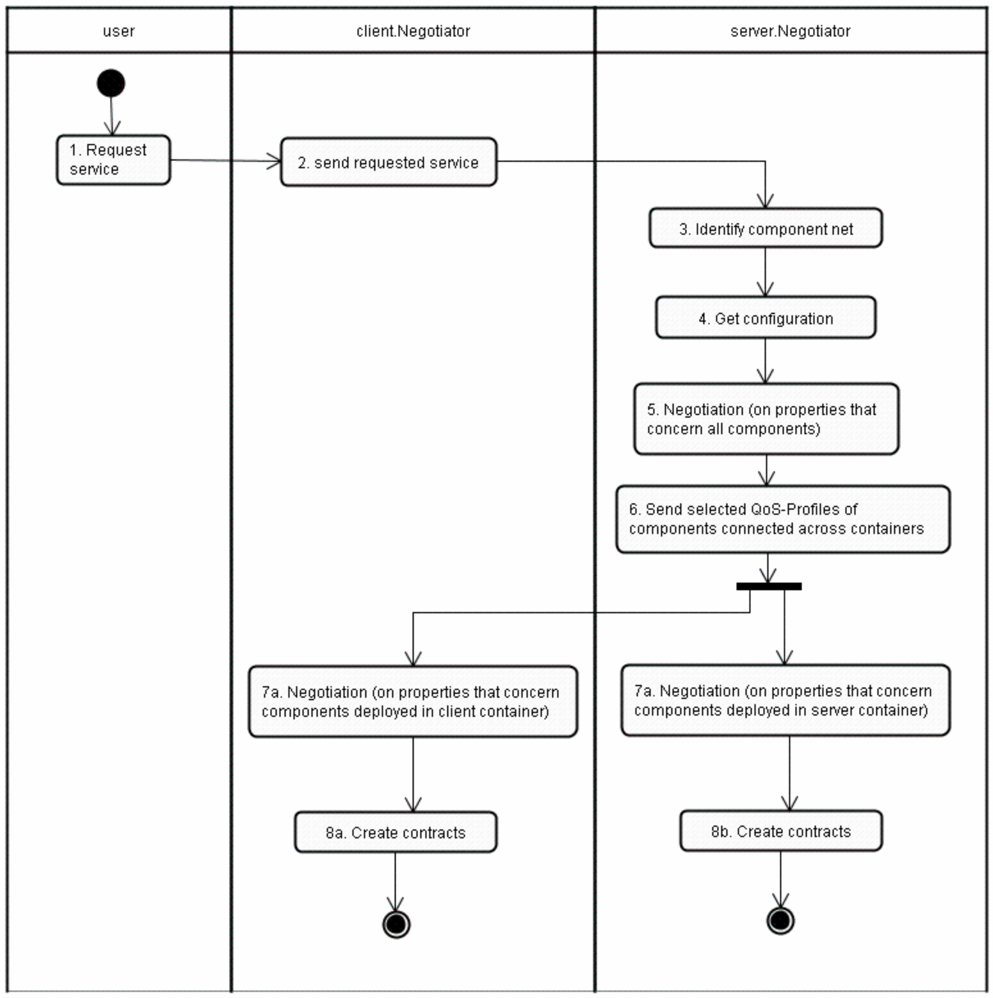

user
client.Negotiator
server.Negotiator
1. Request service
2. send requested service
3. Identify component net
4. Get configuration
5. Negotiation (on properties that concern all components)
6. Send selected QoS-Profiles of components connected across containers
7a. Negotiation (on properties that concern components deployed in client container)
7a. Negotiation (on properties that concern components deployed in server container)
8a. Create contracts
8b. Create contracts

Figure 5.4: Interaction between client and server containers (Hybrid Approach)

Figure 5.5 shows one possible interaction in a multiple-clients scenario. It focuses on how the client and server containers interact to select concrete contracts (i.e. on the decision making process rather than the protocol aspect). The negotiation is started when a client makes a request for a particular service. Only a few of the existing clients is shown in the diagram. *Client_i* and *Client_k* are new clients while *Client_x* is a client that has already established a contract with the server. The particular negotiation scenario in Figure 5.5 results in establishing contracts for *Client_i* and *Client_k* and a re-negotiated contract for Client_x. Steps 3 and 4 are based on the discussions in subsections 4.1.3.1 and 4.1.3.2 while steps 5, 6, and 8 use Algorithm 3.3 in subsection 3.4.4.

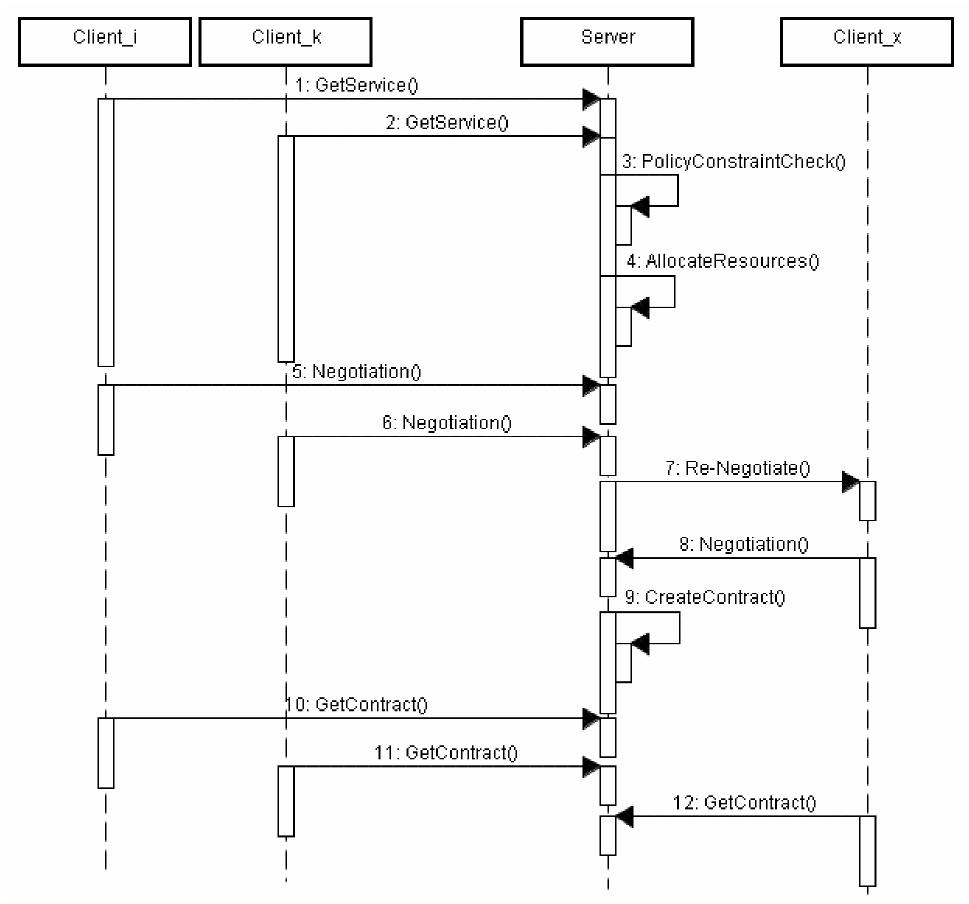


Figure 5.5: Possible interaction between multiple clients and a server

5.3 Negotiation Settings

The negotiation objects, which are the particular NFPs upon which negotiation is to be performed, must be specified and available declaratively in an XML configuration file so that the contract negotiators carry out the negotiation in a phased manner as demonstrated in subsection 3.4. Moreover, policing helps to capture some aspects of negotiation as illustrated in subsection 4.1.3.2 and must be available declaratively to the negotiator. Let the XML file `negotiation.xml` contains the aforementioned configuration information. Note that the components' NFPs (QoS contracts) are specified using CQML$^+$ and are available in an XML file (`cqmlplus.xml`) during the negotiation. `cqmlplus.xml` doesn't contain information about either the precedence (order) of negotiation objects or negotiation policies.

Besides, some important negotiation parameters must be captured in `negotiation.xml` as QoS specification languages do not normally support these issues. For instance, the relative importance of a QoS-dimension in a multi-QoS dimension negotiation must be specified. This is both useful for a satisfied negotiation or in cases where priorities need to be taken for an unsatisfied negotiation.

In our video streaming scenario, negotiation settings at the server and client containers may be provided as follows.

At the server container:

```
<Negotiation>
   <component name=”VideoServer”>
      <coarse-grained name=”coding” value=”mpeg”/>
      <coarse-grained name=”coding” value=”h261”/>
      <coarse-grained name=”protocol” value=”rtp”/>
      <fine-grained name=”frame-rate” weight=”0.4”/>
      <fine-grained name=”resolution” weight=”0.6”/>
   </component>
   <policy>
      <service-class name=”normal”>
            <load-condition name=”over-load”>
                  <property name=”proportion” value=0.3/>
            </load-condition>
      </service-class>
   <policy>
</Negotiation>
```

At the client container:

```
<Negotiation>
   <component name=”VideoPlayer”>
      <coarse-grained name=”coding” value=”h261”/>
      <coarse-grained name=”protocol” value=”rtp”/>
      <fine-grained name=”frame-rate” weight=”0.4”/>
      <fine-grained name=”resolution” weight=”0.6”/>
   </component>
</Negotiation>
```

6 Conclusions and Outlook

Motivated by the need to support the development of distributed component-based applications with non-functional requirements like QoS and security, the main focus of this thesis has been to propose a QoS contract negotiation framework that is employed when collaborating components with NFPs are connected to provide the required service. We summarize below: (i) the most important achievements, the results obtained, and a critique of the approach, (ii) the future work that would advance the ideas in the thesis, and (iii) a final remark.

6.1 Conclusions

This thesis presented a general framework for QoS contract negotiation in distributed component-based software. In this framework, the component containers perform the contract negotiation at run-time when a service is requested by a client. Each component's QoS contracts are assumed to be specified in terms of the *provided* and *required* QoS as well as the associated resource demand. The QoS contracts generally depend on run-time resources and quality attributes fixed dynamically. We have studied three componentized distributed applications to motivate the problem as well as to validate the proposed approach. The analyzed applications are: (i) video streaming, (ii) stock quote, and (iii) billing (to evaluate certain security properties).

We addressed the QoS contract negotiation problem by first modelling it as a constraint satisfaction optimization problem (CSOP). As a basis for this modelling, the provided and required QoS as well as resource demand are specified at the component level. We argued that performing QoS contract negotiation in multiple phases simplifies the negotiation process and make it more efficient. Pertaining to this classification, we presented heuristic algorithms that comprise *coarse-grained* and *fine-grained* negotiations for collaborating components deployed in distributed nodes in the following scenarios: (i) single-client - single-server, (ii) multiple-clients, and (iii) multi-tier scenarios.

The negotiation algorithm in the single-client – single-server scenario (Algorithm 3.3) provides a solution that has the highest utility as far as the most preferred QoS dimension is concerned with a run-time complexity of $O(nd^2)$ where n is the total number of cooperating components and d is the number of QoS-Profiles specified for each component. Algorithm 3.3 assumes that the cooperating components form a tree to achieve a non-backtracking solution. Algorithm 4.1 and 4.2 generalize our approach to the case of multiple-clients and multi-tier scenarios respectively. Algorithm 4.1 uses heuristics to provide a good solution. Finding a globally optimal solution is left for future research.

Under the circumstances that there are not enough resources to satisfy a user's QoS requirements, or conformant QoS-Profiles cannot be found when trying to get a solution that meets the user's constraint, Algorithm 3.4 finds a solution that effectively performs a

QoS trade-off. To achieve this trade-off, a user's preferences towards each QoS dimension and the utility function are used. Under these conditions, the selected QoS-Profiles maximize the most preferred property at the expense of the less preferred ones for dependent QoS dimensions. For instance, if a user prefers frame rate over resolution then the QoS trade-off is performed by successively trying to maintain the user's QoS requirements on frame rate and look for lower values of resolution and only to go to QoS-Profiles with lower frame rate after QoS-Profiles that have lower resolution values have been exhausted.

Efficiency of the negotiation process was one of the goals we addressed in the heuristic algorithms. We treated efficiency from two different aspects. The first is from the point of view of the selection of QoS contracts that provide good solutions based on a certain negotiation goal (e.g. maximizing user's satisfaction). This has been achieved in the single-client – single-server scenario by appropriate value and variable ordering policies in the fine-grained negotiation that employs branch-and-bound techniques. The second is from the standpoint of the number of inter-container message exchanges and the level of concurrency that can be achieved in the negotiation process. As far as this view is concerned, a hybrid solution, that combines centralized and decentralized approaches, has been found to perform well for realizing the proposed algorithms while a purely distributed solution is inappropriate due to the lack of predictability on the number of inter-container communications required to reach an agreement.

We developed a prototype that implements all the proposed negotiation algorithms for the purpose of validating the approach. The prototype has been developed with the intention of integrating it into a component framework using the interceptor pattern that enables adding cross-cutting concerns like contract negotiations. We conducted an experiment to specify the QoS-Profiles of the involved components in one of the applications we studied. In a run-time system that implements our algorithms, we simulated different behaviors concerning: (i) user's QoS requirements and preferences, (ii) resource availability conditions concerning the client, server, and network bandwidth, and (iii) the specified QoS-Profiles of the collaborating components. Under various conditions, the outcome of the negotiation confirms the claim we made earlier with regard to the outputs of Algorithms 3.3, 3.4, and 4.1.

It is to be noted that our entire approach extensively depends on the QoS-Profiles of the collaborating components. The component developer specifies the QoS-Profiles after conducting experiments and measuring the provided quality, required quality and the resource demand at the component level. Because of the negotiable (dynamic) nature of QoS properties, multiple QoS-Profiles are needed to specify QoS contracts. Given the same application, constituting components, and same environment, the outcome of the QoS contract negotiation can depend on the specified QoS-Profiles. One of the drawbacks of this is that the solution obtained might not be the optimal one. In order to overcome such a discrepancy, there must be some standard way of specifying QoS contracts, which might be done by either using measurements or analytical means. This is, however, beyond the scope of the thesis.

6.2 Future Work

Globally Optimal Solutions

One area deserving exploration is to give negotiation results by analytically finding globally optimal solutions. The optimal solution should take into account the interests of users and service providers, which is usually conflicting. In order to accomplish this, the nature of utility functions should be examined. Defining a utility function is a difficult task due to the inter-dependency of QoS-dimensions and various parameters. In this thesis, we simply assumed that a utility function is given as a weighted sum of dimension-wise utilities without a further study as to how these weights are determined. Additionally, more parameters need to be incorporated into a utility function in the context of the multiple-clients scenario (i.e. from the point of view of service providers). Some parameters that need to be incorporated concern: how to differentiate two clients of the same service class during over-load case and contract termination costs.

Other Scenarios

We have demonstrated our approach basically for client/server applications. But, other scenarios for the emerging areas like service-oriented or peer-to-peer computing merit further investigation. In a component-based peer-to-peer application, a component is both a service consumer and provider. Moreover, next generation peer-to-peer applications are highly decentralized and dynamic, which might consist of a large number of peers that may join and leave at any time. These and other challenges need to be examined when extending our framework to the new computing models.

Consideration of More Properties

Due to the scope of the thesis, we considered only certain QoS properties during negotiation. One interesting area that deserves future research is the incorporation of cost/price of service in the QoS contract negotiation. This requires examination of appropriate resource allocation models, which may be market-based, suitable pricing models (e.g. pay-per use), etc.

6.3 Final Remark

In a nutshell, our work can be taken as an important step forward in the realization of a component technology that enables the development of applications with non-functional requirements like QoS from components whose QoS contracts have been specified. Our QoS contract negotiation framework, which can be integrated in a component container, act as a run-time support environment when QoS Contracts are negotiated under different scenarios.

References

[Aagedal, J. Ø., 2001] Quality of Service Support in Development of Distributed Systems. PhD thesis, University of Oslo.

[Abdelzaher, T.F., Shin, K.G. and Bhatti, N., 2003] "User-Level QoS-Adaptive Resource Management in Server End-Systems," IEEE Transactions on Computers, vol. 52, no. 5, pp. 678-685, May, 2003.

[Amundsen, S., Lund, K., Eliassen, F. and Staehli, R., 2004] "QuA: Platform-Managed QoS for Component Architectures," Norsk Informatikkonferanse (NIK) 2004

[Amundsen, S., Lund, K., Griwodz, C. and Halvorsen, P., 2005] "QoS-aware Mobile Middleware for Video Streaming," In: 31st EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA). IEEE Computer Society, pages 54-61, 2005

[Bachman, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R. and Wallnau, K., 2000] "Volume II: Technical Concepts of Component-Based Software Engineering," Technical Report, CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, May 2000

[Baochun, L., Dongyan, X. and Nahrstedt, K., 2002] "An Integrated Run-time QoS-Aware Middleware Framework for Distributed Multimedia Applications," ACM Multimedia Systems Journal, Special Issue on Multimedia Middleware, 8(5):420-430, 2002.

[Beugnard, A., Jezequel, Jean-Marc., Plouzeau, N. and Watkins, D., 1999] Making components contract aware. IEEE Computer, 32(7):38–45, July 1999.

[Blair, G., Andersen, A., Blair, L., Coulson, G. and Snchez, D., 2000] "Supporting Dynamic QoS Management Functions in a Reflective Middleware Platform", IEE Proceedings on Software Engineering, Vol. 147, No. 1, pp 13-21, February 2000.

[Blair, G., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R., Parlavantzas, N. and Saikoski, K., 2001] "The Design and Implementation of OpenORB v2.," IEEE Distributed Systems Online, vol. 2 no 6, Special Issue on Reflective Middleware , 2001.

[Bouyssounouse, B. and Sifakis, J. (eds.), 2005] "Embedded Systems Design: The ARTIST Roadmap for Research and Development," ISBN: 3-540-25107-3. LNCS 3436, 2005.

[Braden, R., Zhang, L., Berson, S., Herzog, S. and Jamin, S., 1997] Resource ReSerVation Protocol (RSVP) Version 1 Functional Specification. Request For Comment 2205, IETF, 1997

[Brown, A.W. and Wallnau, K.C., 1996] "Engineering of component-based systems," iceccs, p. 414, Second IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'96), 1996.

[Brown, A.W. and Wallnau, K.C., 1998] "The Current State of CBSE," IEEE Software, vol. 15, no. 5, pp. 37-46, Sept/Oct, 1998

[Cheesman, J. and Daniels, J., 2001] "UML Components: A Simple Process for Specifying Component-Based Software," Addison Wesley Longman, Inc., 2001

[Clements, P.C., 1995] "From Subroutines to Subsystems: Component-Based Software Development," American Programmer, V8#11, Cutter Information Corp., November 1995

[Coulson, G., Blair, G., Clarke, M. and Parlavantzas, N., 2002] "The design of a configurable and reconfigurable middleware platform," Distributed Computing, pp. 109-126, 2002.

[Crnkovic, I., 2001] "Component-based software engineering - new challenges in software development," in Software Focus, vol. 2, no. 4, pages 127-133, 2001

[Dongyan, X., Wichadakul, D. and Nahrstedt, K., 2000a] "Multimedia Service Configuration and Reservation in Heterogeneous Environments," 20th International Conference on Distributed Computing Systems (ICDCS 2000), Taipei, Taiwan, April 2000.

[Dongyan, X., Wichadakul, D. and Nahrstedt, K., 2000b] "Resource-aware configuration of ubiquitous multimedia services," IEEE International Conference on Multimedia and Expo (II), vol.2, no.pp.851-854, 2000

[Franz, E. and Pohl, C., 2004] "Towards Unified Treatment of Security and Other Non-functional Properties," Workshop on AOSD Technology for Application-Level Security, MAR 2004, Lancaster, UK.

[Freuder, E. C., 1989] "Partial constraint satisfaction," in Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, IJCAI-89, Detroit, Michigan, USA, pages 278–283, 1989.

[Freuder, E. C., 1982] "A sufficient condition for backtrack-free search," J. ACM Vol.29, No.1 January 1982, 24-32

[Frølund, S. and Koistinen, J., 1998] “Quality-of-Service Specification in Distributed Object Systems,” in Distributed Systems Engineering Journal, Dec. 1998.

[Frølund, S. and Koistinen, J., 1999] "Quality-of-Service Aware Distributed Object Systems", in proceedings of the 1999 USENIX Conference on Object-Oriented Technologies and Systems (COOTS).

[Göbel, S., Pohl, C., Röttger, S. and Zschaler, S., 2004a] “The COMQUAD component model – enabling dynamic selection of implementations by weaving non-functional aspects.” In International Conference on Aspect-Oriented Software Development (AOSD’04), Lancaster, UK, 22–26 Mar. 2004. ACM.

[Göbel, S., Pohl, C., Aigner, R., Pohlack, M., Röttger, S. and Zschaler, S., 2004b] “The COMQUAD component container architecture and contract negotiation,” Technical Report TUD-FI04-04, Technische Universität Dresden, April 2004.

[Heineman, G.T. and Councill, W.T. (eds.), 2001] “Component-Based Software Engineering: Putting the Pieces Together,” Addison-Wesley, Boston, MA, June 2001

[ISO/IEC, 1999], Information Technology - Software product quality - Part 1: Quality model, ISO/IEC, Report: 9126-1,

[Jennings, N. R., Faratin, P., Lomuscio, A. R., Parsons, S., Sierra, C. and Wooldridge, M., 2001] "Automated negotiation: prospects, methods and challenges," Int. Journal of Group Decision and Negotiation 10 (2) 199-215.

[Keller, A., Kar, G., Ludwig, H., Dan, A. and Hellerstein, J., 2002] “Managing Dynamic Services: A Contract-based Approach to a Conceptual Architecture,” Proc. NOMS 2002: 8th Network Operations and Management Symposium, Florence, Italy, 15-19 Apr. 2002.

[Khan, K. Md., Han, J. and Zheng, Y., 2000] "Security Characterization of Software Components and Their Composition," tools, p. 240, 36th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS-Asia'00), 2000.

[Koistinen, J. and Seetharaman, A., 1998] "Worth-Based MultiCategory Quality-of-Service Negotiation in Distributed Object Infrastructures", HP Technical Report, 1998.

[Larsson, M., 2004] “Predicting Quality Attributes in Component-based Software Systems,” PhD Thesis, Mälardalen University, March 2004

[Lee, C., Lehoczky, J., Rajkumar, R. and Siewiorek, D., 1999] "On Quality of Service Optimization with Discrete QoS Options," In proceedings of the IEEE Real-time Technology and Applications Symposium, June 1999

[Levy, R., Nagarajarao, J., Pacifici, G., Spreitzer, M., Tantawi, A. and Youssef, A., 2003] “Performance Management for Cluster Based Web Services,” Proceedings of 8th IFIP/IEEE International Symposium on Integrated Network Management (IM 2003), Colorado Springs, USA, March 2003.

[Loyall, P., Schantz, R. E., Zinky, J. E. and Bakken, D. E., 1998] "Specifying and measuring quality of service in distributed object systems," Proc.1st IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 98), April 1998.

[Menasce, D. A., Ruan, H. and Gomaa, H., 2004] “A framework for QoS-aware software components,” In The fourth international workshop on Software and performance, pages 186–196, Redwood Shores, CA, USA, 2004.

[Meyer, B., 1999] “The significance of components,” in Software Development, Beyond Objects Column, Nov. 1999

[Meyerhöfer, M. and Neumann, C., 2004] “Testejb - a measurement framework for EJBs,” In I. C. (Hrsg.), editor, International Symposium on Component-based Software Engineering (CBSE7), volume 3054 of Lecture Notes in Computer Science, Edinburgh, Scotland, 24–25 May 2004.

[Miguel, M., 2001] “Solutions to Make Java-RMI Time Predictable” In Proceedings of 4th International Symposium on Object-Oriented Real-Time Distributed Computing. ISORC’2001. IEEE, May 2001.

[Miguel, M., Ruiz, J. F. and Garca-Valls, M., 2002] “QoS-Aware component frameworks,” In 10th International Workshop on Quality of Service (IWQoS 2002), Miami Beach, USA, May 2002.

[Nahrstedt, K., Dongyan, X., Wichadakul, D. and Baochun, L., 2001] "QoS-aware middleware for ubiquitous and heterogeneous environments ," Communications Magazine, IEEE , vol.39, no.11pp.140-148, Nov 2001

[Natarajan, B., Schmidt, D.C. and Vinoski, S., 2004] Object Interconnections: the CORBA Component Model, Part 3: The CCM Container Architecture and Component Implementation Framework, C/C++ Users Journal, September, 2004.

[OMG, 2003] “Quality of Service for CORBA Components, Request For Proposal, OMG Document: mars/2003- 06-12”

[OMG, 2005] "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms," OMG Document -- ptc/05-05-02

[Park, J-T., Baek, J-W. and Hong, J. W-K., 2001] "Management of Service Level Agreements for Multimedia Intemet Service Using an Utility Model", IEEE Communications Magazine, Vol. 39 Issue 5, May 2001, pp.100-106

[Prasad, R. S., Murray, M., Dovrolis, C. and Claffy, K. C., 2003] "Bandwidth estimation: Metrics, measurement techniques, and tools," IEEE Network, Jun 2003.

[Preiss, O., Wong, J. and Wegmann, A., 2001] "On Quality Attribute Based Software Engineering," euromicro, p. 0114, 27th Euromicro Conference 2001: A Net Odyssey (euromicro'01), 2001.

[Rajkumar, R., Lee, C., Lehoczky, J. and Siewiorek, D., 1997] "A Resource Allocation Model for QoS Management," in Proceedings of the IEEE Real-Time Systems Symposium, December 1997.

[Reussner, R. H., Poernomo, I. H. and Schmidt, H. W., 2003] "Contracts and Quality Attributes for Software Components," in Proceedings of the Eighth International Workshop on Component-Oriented Programming (WCOP'03), June 2003.

[Ritter, T., Born, M., Unterschütz, T. and Weis, T., 2003] "A QoS Metamodel and its Realization in a CORBA Component Infrastructure," In Hawaii International Conference on System Sciences (HICSS-36), Hawaii, USA, January 2003.

[Rosa, N., Cunha, P. and Justo, G., 2002] "ProcessNFL: A Language for Describing Non-functional Properties," Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9, IEEE Computer Society, Washington, DC, USA. 2002.

[Rosenschein, J. S. and Zlotkin, G., 1994] "Rule of Encounter: Design conventions for automated negotiation among computers," The MIT Press, 1994.

[Röttger, S. and Zschaler, S., 2003] "CQML$^+$: Enhancements to CQML." Edited by Jean-Michel Bruel, Proc. 1st Intl. Workshop on Quality of Service in Component-Based Software Engineering, Toulouse, France.

[Russell, S. and Norvig, P., 2002] "Artificial Intelligence: A Modern Approach," Second Edition, Prentice Hall, Englewood Cliffs, New Jersey, 2002.

[Saaty, T., 1992] "Multicriteria Decision Making – The Analytic Hierarchy Process," Technical Report, University of Pittsburgh, RWS Publications, 1992.

[Schantz, R. E., Loyall, J., Rodrigues, C., Schmidt, D. C., Krishnamurthy, Y. and Pyarali, I., 2003] Flexible and Adaptive QoS Control for Distributed Real-time and Embedded Middleware. The ACM/IFIP/USENIX International Middleware Conference, June 2003, Rio de Janeiro, Brazil.

[Schmidt, D.C. and Vinoski, S., 2004] Object Interconnections: The CORBA Component Model, Part 2: Defining Components with the IDL 3.x Types, C/C++ Users Journal, April, 2004.

[Staehli, R., Eliassen, F. and Amundsen, S., 2004] "Designing Adaptive Middleware for Reuse," In Middleware 2004 Companion, 3rd Workshop on Reflective and Adaptive Middleware, 2004.

[Sun Microsystems, 2001] "Java Media Framework API Guide," Nov. 2001

[Szyperski, C., 2002] "Component Software: Beyond Object-Oriented Programming," Second Edition, Component Software Series, Addison-Wesley Publishing Company.

[Tsang, E.P.K., 1993] "Foundations of Constraint Satisfaction," Academic Press Limited, 1993

[Ulbrich, A., Weis, T., Geihs, K. and Becker, C., 2003] "DotQoS – a QoS extension for .NET Remoting," In International Workshop on Quality of Service (IWQoS), volume 2707 of Lecture Notes in Computer Science (LNCS), pages 363–380, Monterey, CA, 2003.

[Vecellio, G., Thomas, W. M. and Sanders, R., 2002] "Containers for Predictable Behavior of Component-based Software", in Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly, Orlando, Florida, May 19-20, 2002.

[Walpole, J., Krasic, C., Liu, L., Maier, D., Pu, C., McNamee, D. and Steere, D., 1999] "Quality of service semantics for multimedia database systems," Proc. Data Semantics 8: Semantic Issues in Multimedia Systems IFIP TC-2 Working Conference, Jan. 1999, pp. 393-412

[Wang, N., Schmidt, D. C., Gokhale, A., Rodrigues, C., Natarajan, B., Loyall, J. P., Schantz, R. E. and Gill, C. D., 2003] "QoS-enabled Middleware," in Middleware for Communications, Qusay Mahmoud, Ed. Wiley and Sons, New York.

[Wang, N., 2004a] "Composing Systemic Aspects into Component-Oriented DOC Middleware," PhD Thesis, Washington University, May 2004

[Wang, N. and Gill, C. D., 2004] "Improving Real-Time System Configuration via a QoS-aware CORBA Component Model", in Proceedings of the Hawaii

International Conference on System Sciences (HICSS), Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, HICSS 2003, Honolulu, Hawaii, January, 2004.

[Weis, T., Plouzeau, N., Amoros, G., Donth, P., Geihs, K., Jézéquel, J.-M. and Sassen, A.-M., 2003] Chapter - QCCS: Quality Controlled Component-Based Software Development, In Business Component-Based Software Engineering. Kluwer Academic Publishers, 2003. – ISBN 1–4020–7207–4

[Xiaohui, G. and Nahrstedt, K., 2002] "Dynamic QoS-aware multimedia service configuration in ubiquitous computing environments," Proceedings of the 22nd International Conference on Distributed Computing Systems, vol., no. pp. 311- 318, 2002

[Yokoo, M., Durfee, E. H., Ishida, T. and Kuwabara, K., 1998] "The Distributed Constraint Satisfaction Problem: Formalization and Algorithms," IEEE Transactions on Knowledge and DATA Engineering 10(5).

[Yokoo, M. and Hirayama, K., 2000] "Algorithms for Distributed Constraint Satisfaction: A Review," Autonomous Agents and Multi-Agent Systems. 2000.

[Zeng, L., Benatallah, B., Ngu, A., Dumas, M., Kalagnanam, J. and Chang, H., 2004] "QoS-Aware Middleware for Web Services Composition," IEEE Transactions on Software Engineering, vol. 30, no. 5, pp. 311-327, May, 2004.

[Zhang, X., Lesser, V. and Wagner, T., 2003] "A Two-Level Negotiation Framework for Complex Negotiations," iat, p. 311, IEEE/WIC International Conference on Intelligent Agent Technology (IAT'03), 2003.

[Zinky, A., Bakken, D. E. and Schantz, R. E., 1997] "Architectural Support for Quality of Service for CORBA Objects," Theory and Practice of Object Systems, 3(1), 1997.

Appendix A - Abbreviations

AOP	Aspect Oriented Programming
B&B	Branch and Bound
CBSD	Component-Based Software Development
CBSE	Component-Based Software Engineering
CCM	CORBA Component Model
CIAO	Component Integrated Adaptive Communication Environment (ACE) ORB
COM	Component Object Model
COMQUAD	Components with Quantitative Properties and Adaptivity
CORBA	Common Object Request Broker Architecture
CQML	Component Quality Modeling Language
CSOP	Constraint Satisfaction Optimization Problem
CSP	Constraint Satisfaction Problem
DCOM	Distributed Component Object Model
DOC	Distributed Object Computing
DRE	Distributed Real-Time Embedded
DROPS	Dresden Real-Time Operating System
EJB	Enterprise Java Bean
IP	Internet Protocol
ISO/IEC	International Organization for Standardization / International Electro-technical Commission
JMF	Java Media Framework
NFP	Non-Functional Property
OMG	Object Management Group
ORB	Object Request Broker
PCSP	Partial Constraint Satisfaction Problem
QDL	Quality Description Language
QML	QoS Modeling Language
QoS	Quality of Service
QuO	Quality Objects
RCC	Resource-Consuming Component
RMI	Remote Method Invocation
RSVP	Resource Reservation Protocol
RT-CORBA	Real-Time CORBA
RTP	Real-time Transport Protocol
SLA	Service Level Agreement
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UML	Unified Modeling Language
XML	Extensible Markup Language