A Framework for QoS Contract Negotiation in Component-Based Applications

Mesfin Mulugeta and Alexander Schill

Institute for System Architecture
Dresden University of Technology, Germany
{mulugeta|schill}@rn.inf.tu-dresden.de

Abstract. The support of QoS properties in component-based software requires the run-time selection of appropriate concrete QoS contracts at the ports of the interacting components. Such a selection process is called QoS contract negotiation. This paper discusses the architecture of a QoS contract negotiation framework and how it is implemented in our prototype. The framework can be integrated in a component container and act as a run-time support environment when QoS Contracts are negotiated under different application scenarios. Our approach is based on: (i) the notion that the required and provided QoS properties as well as resource demands are specified at the component level; and (ii) that QoS contract negotiation is modeled as a constraint solving problem.

1 Introduction

Component-Based Software Engineering (CBSE) allows the composition of complex systems and applications out of well defined parts (components). In today's mature component models (e.g. EJB and Microsoft's .NET), components are specified with syntactic contracts that provide information about which methods are available and limited non-functional attributes like transaction properties. This underspecifies the components and limits their suitability and reuse to a specific area of application and environment. In [2], component contracts have been identified in four different levels: syntactic, behavioral, synchronization, and QoS. The explicit consideration of component QoS contracts aims at simplifying the development of component-based software with non-functional requirements like QoS, but it is also a challenging task.

For applications in which the consideration of non-functional properties (NFPs) is essential (e.g. Video-on-Demand), 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 must be matched by the QoS level provided by the other. This matching requires the selection of appropriate QoS contracts at each port. Generally, QoS contracts of components depend on run-time resources (e.g. network bandwidth, CPU time) or quality attributes to be established dynamically. QoS contract negotiation involves the run-time selection of appropriate concrete QoS contracts specified at the ports of the interacting components.

In [9], we presented how QoS contract negotiation can be formulated as a constraint solving problem and proposed two-phased heuristic algorithms in a single-client - single-server and multiple-clients scenarios. This paper focuses on a conceptual negotiation framework that can be integrated in a component container to act as a run-time support environment when QoS contracts are negotiated under different application scenarios. We also discuss details of our prototype implementation of the framework. The advantages of having the framework are the following. Firstly, it can be directly used for various types of applications as long as similar QoS contract specification schemes are used. Secondly, the framework can be extended to handle different scenarios either by incorporating new negotiation algorithms or by including more features with respect to the basic components of the framework.

The rest of the paper is organized as follows. In section 2 we examine related work. Section 3 details our QoS contract negotiation framework. Section 4 is devoted to the discussion of how we have implemented the proposed framework by demonstrating the ideas based on an example application scenario. The paper closes with a summary and outlook to future work.

2 Related Work

The work in [4] offers basic QoS negotiation mechanisms in only a single container. It hasn't pursued the case of distributed applications where components are deployed in multiple containers. In [11] 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. This approach covers only the protocol aspect of the negotiation process. It hasn't pursued the decision making aspects of the negotiation.

In [8] 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. QuA [13] aims at defining an abstract component architecture, including the semantics for general QoS specifications. QuA's QoS-driven Service Planning has similarities to our concept of QoS contract negotiation. Complexity issues, however, haven't been accounted for in the service planning.

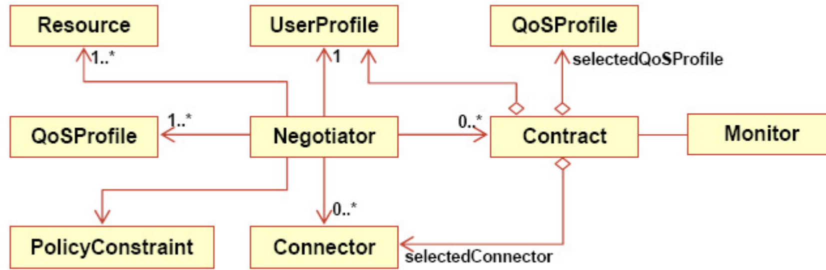


Fig. 1. Architecture of a QoS Contract Negotiation Framework

The Quality Objects (QuO) framework offers one of the most advanced concepts and tools to integrate QoS into distributed applications based on CORBA [7]. In QuO, a QoS developer specifies a QoS contract between the client and object. This contract specifies the QoS that the client desires from the object, the QoS that the object expects to provide, operating regions indicating possible measured QoS, and actions to take when the level of QoS changes. Having to provide the adaptive behavior explicitly in the QoS contract is a burden on the QoS developer. In our work, we have taken the approach that instead of specifying the pre-determined adaptive behavior in the QoS contract, we leave the reasoning on adaptation (or negotiation) to the component containers, which they would perform based on the specification of the component's QoS profiles. One advantage of this approach is that it makes the application development process easier.

3 Framework Architecture and Interaction

3.1 Architecture

Our framework can be seen as a reusable design that consists of the representation of important active components, data entities, and the interaction of different instances of these to enable QoS contract negotiation. Fig. 1 shows the conceptual architecture of our framework represented as a UML class diagram.

`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 declaratively 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. Finally, `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* [10]. We use CQML$^+$ [12][5], an extension of CQML [1], 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 for a `VideoPlayer` component that may be used in video streaming scenarios.

```
QoSProfile aProfile for VideoPlayer {
  provides frameRate=15, resolution=352x288;
  uses frameRate=15, resolution=352x288;
  resources cpu=8.9%; networkBandwidth=2.1kbps; memory=30KB;
}
```

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 off-line 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 QoS-dimensions. 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 values of 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 `Negotiator` on the client node. The `Negotiator` at the client and server side exchange information about the required service and the user's profile before the negotiation begins. `Negotiator` is 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 after systematically 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) [14]. A CSOP consists of variables whose values are taken from finite, discrete domains, a set of constraints on their values, and an objective function. 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 [6], 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 selectedQoSProfileClient;
  QoSProfile selectedQoSProfileServer;
  Connector selectedConnector
  UserProfile userProfile;
  double contractValidityPeriod;
  //...
};
```

If a contract is established between two components deployed in the same container, the clauses of the contract contains the QoS offers and needs as well as the resource demands of the components. That means, the selected QoS-profiles of the client and server components would be clauses in the contract (in this case,

`selectedConnector` and `userProfile` are `null`). If a contract is established 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 (`selectedQoSProfileClient` 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 as policy constraint, 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).

3.2 Interaction

The interaction diagram in Fig. 2 depicts an overall view of the negotiation process. It is assumed that the application's components are deployed in client and server containers. The diagram shows a successful negotiation scenario performed using a centralized approach. The following is demonstrated in Fig. 2.

– 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 (step 1).
– Intercepting the user's request, `Negotiator` at the client's container constructs `UserProfile` and sends a message to the server container, which will identify the components that participate to provide the required service (step 2).
– In steps 3 to 5, the container that is responsible for the negotiation collects QoS contracts specified for the collaborating components, resource conditions at each node and the network, and policy constraints that may be imposed by service providers.
– The responsible container performs the negotiation (step 6) in two phases [9]. In the first phase, negotiation is made on coarse-grained properties (step

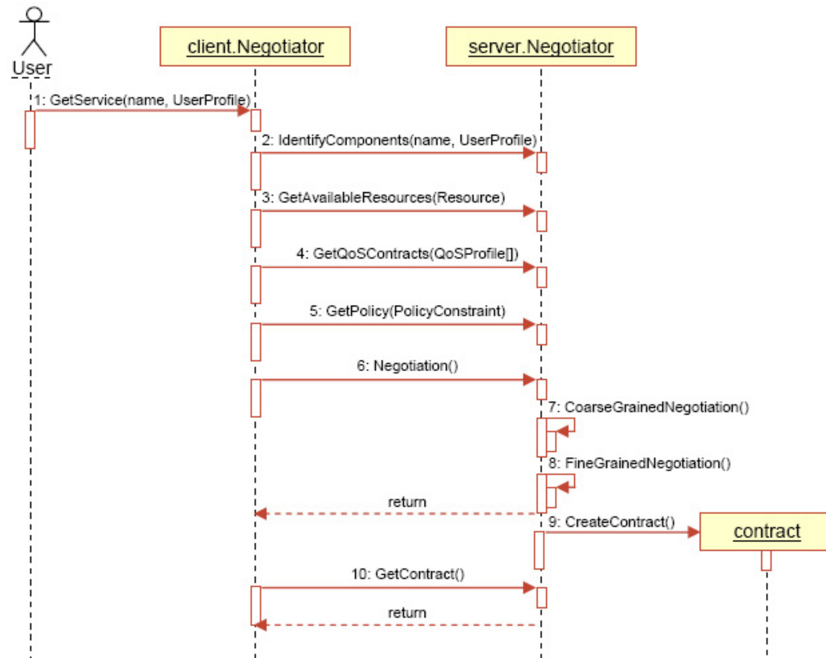


Fig. 2. Interaction between client and server containers (centralized approach)

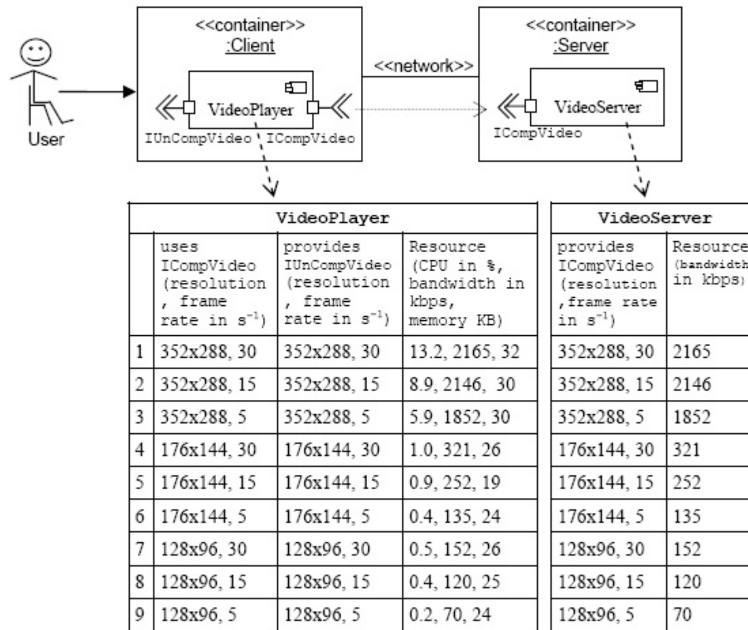
7). When this is successful, negotiation on fine-grained properties continues (step 8).

- The responsible container creates all contracts. A contract is established between any two interacting components and between a user and the front-end component (step 9).
- The client container retrieves relevant contracts from the server container (step 10). These are contracts between components deployed in the client container or between components connected across containers.

4 Implementation and Example

4.1 Example

As a proof-of-concept, we have developed a prototype of the framework proposed in Fig. 1. Our prototype implements all the elements of the framework with the exception of `PolicyConstraints` and `Monitor`. The framework has been used to negotiate QoS contracts at run-time for a video streaming application scenario. In the future we plan to extend our implementation to include contract monitoring and the consideration of policy constraints. The prototype has been implemented in Java as also demonstrated by the code snippets shown below.



VideoPlayer			
	uses ICompVideo (resolution, frame rate in s^{-1})	provides IUnCompVideo (resolution, frame rate in s^{-1})	Resource (CPU in %, bandwidth in kbps, memory KB)
1	352x288, 30	352x288, 30	13.2, 2165, 32
2	352x288, 15	352x288, 15	8.9, 2146, 30
3	352x288, 5	352x288, 5	5.9, 1852, 30
4	176x144, 30	176x144, 30	1.0, 321, 26
5	176x144, 15	176x144, 15	0.9, 252, 19
6	176x144, 5	176x144, 5	0.4, 135, 24
7	128x96, 30	128x96, 30	0.5, 152, 26
8	128x96, 15	128x96, 15	0.4, 120, 25
9	128x96, 5	128x96, 5	0.2, 70, 24

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

Fig. 3. A video streaming scenario and QoS-profiles of VideoPlayer and VideoServer implementations

The video streaming application scenario that we used involves a `VideoServer` component deployed in a server container and a `VideoPlayer` component deployed in a client container (Fig. 3). We use the COMQUAD component model [5] that supports streams as special interface types and allows to specify non-functional properties for them. `VideoPlayer` implements two interfaces: a *uses* interface `ICompVideo` and a *provides* interface `IUnCompVideo` while `VideoServer` implements a provides interface `ICompVideo`. `VideoPlayer`'s `ICompVideo` is connected to `VideoServer`'s `ICompVideo` to receive video streams for a playback at the client's node.

We conducted an experiment to specify the QoS-Profiles of `VideoPlayer` and `VideoServer`. The `VideoPlayer` component was implemented using Sun's JMF framework and the `VideoServer` component abstracts the video media file that has been pre-encoded into many files with differing frame rates, resolutions, protocols, and coding algorithm. Fig. 3 depicts some of the measured QoS-Profiles of `VideoPlayer` and `VideoServer`, with UDP protocol and mp42 coding. Note that these QoS-Profiles depend on the content of the video. During the measurements, average bandwidth and CPU percentage time have been considered. The bandwidth requirement of `VideoServer` is taken to be the same as that of `VideoPlayer`. The measured CPU requirements of `VideoServer` are too small (in the range of 0.1%) and hence have been left out from Fig. 3.

4.2 Implementation

Next, we will see how the framework elements are instantiated and how they interact during the QoS contract negotiation, which is initiated when a user sends his request to get a service. Our subsequent discussion roughly follows the sequence diagram in Fig. 2.

For the example in Fig. 3, a user requests to watch a video clip that is streamed from the video provider to the user. The involved components are `VideoPlayer` and `VideoServer`. A user also sends his QoS requirements from which `UserProfile` is constructed. `UserProfile` is initialized with the user's QoS requirement, i.e. $frameRate \geq 12s^{-1}$ and $resolution = 176 \times 144$.

```
public class UserProfile {
  List<QoSStatement> uses;
  //...
};
userProfile = new UserProfile({frameRate=12, resolution=176x144});
```

The other element from the framework that needs to be instantiated before the negotiation is started is `Resource`. For the example in Fig. 3, three instances of `Resource` are used as illustrated below. Let's assume that the available resources at the client and server nodes as well as the end-to-end network bandwidth are as indicated below.

```
public class Resource {
  double cpu; // in percentage
  double memory; // in KB
  double networkBandwidth; // in kbps
  //...
};
clientResources = new Resource(80, 150, null);
serverResources = new Resource(50, 200, null);
endToEndBandwidth = new Resource(null, null, 1000);
```

For the video streaming example, the QoS contracts are specified with multiple QoS profiles as shown in Fig. 3. A QoS-Profile for `VideoPlayer` is, for example, represented as:

```
  QoSProfile aProfile for VideoPlayer {
    provides frameRate=15, resolution=352x288;
    uses frameRate=15, resolution=352x288;
    resources cpu=8.9%; networkBandwidth=2.1kbps; memory=30KB;
  }
```

We used a data structure called `ComponentMeta` to store the QoS-Profiles of each component (i.e. a component's meta data) as shown below. Additional variables are also needed to be defined. `tempSelectedProfile` holds temporarily selected profiles during the negotiation process while `selectedProfile` stores the selected profile after the negotiation is concluded. `currentProfilePos` indicates the position of the temporarily selected profile in the array of QoS profiles, which are stored from low to high quality. Although not shown in the code snippet below, `ComponentMeta` defines, among others, the `Set` and `Get` functions for the variables it defines.

```
public class ComponentMeta {
```

```
  List<QoSProfile> profiles = null;
  QoSProfile selectedProfile = null;
  QoSProfile tempSelectedProfile = null;
  int currentProfilePos=0;
25  //...
};
ComponentMeta c[] = new ComponentMeta[N];
enum CG {On_Client, On_Server, Across_Containers} // component's group
```

`ComponentMeta` is instantiated for each cooperating component. It is assumed that `profiles` in `ComponentMeta` are populated with values after parsing the QoS specification that is declaratively available as an XML file to the run-time system. CG (Line 28) is used to identify whether a component is deployed on the client, server, or connected across containers.

An instance of `Connector` is required during the negotiation when components are deployed in distributed nodes. The following code snippet assumes that only a delay property is specified.

```
public class Connector {
30  List<QoSStatement> properties = new ArrayList<QoSStatement>();
  //...
};
connector = new Connector({delay=0.001});
```

`Negotiator` uses instances of `c[i]`, `userProfile`, `connector`, `clientResource`, `serverResource`, and `endToEndBandwidth` described above when performing the negotiation. Additional variables are also required for the particular algorithms used in the negotiation.

```
public class Negotiator {
35  ComponentMeta c[] = null;
  Resource clientResources = null;
  Resource serverResources = null;
  Resource endToEndBandwidth = null;
  UserProfile userProfile = null;
40  Connector connector = null;
  UserProfile boundProfile = null;//initialized to user's QoS requirement
  Contract contracts[] = null;
  //...
  void GetUserProfile() {// gets data for userProfile }
45  void GetAvailableResources() {// gets data for the various resources }
  void GetQoSContracts() {// gets data for c[i].profiles}
  boolean FineGrainedNegotiation() {//performs fine grained negotiation}
};
```

`FineGrainedNegotiation()` uses the standard branch and bound (B&B) [14] technique to find a solution for a problem modeled with a CSOP. To apply B&B to our problem, we need to define policies concerning selection of the next variable and selection of the next value. We must also specify the objective and heuristic functions. In our implementation, the variables (QoS-profiles to be used by each component) are ordered for assignment by topologically sorting the network of cooperating components. The assignment starts from the minimal element (i.e. the front-end component, for e.g. `VideoPlayer` in Fig. 3) and from there continues to the connected components, and so on. The possible values of each variable, i.e. the QoS-profiles specified for each component, must be ordered from lower to higher quality.

The heuristic function, `hValue`, maps every partial labeling (assignment) to a numerical value and this value is used to decide whether extending a partial labeling to include a new label would result in a "better" solution. 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, `hValue` (Line 53) 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, `hValue` needs to be computed only at the beginning of each iteration, that is, when the front-end component is assigned a new value (Line 52). 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.

```
   boolean FineGrainedNegotiation()
50 {
     if (ConformanceCheck() == false) return false;
     for(int i=c[0].GetCurrentProfilePos();i<c[0].profiles.size(); i++) {
       if(hValue(c[0].profiles.get(i).provides)>hValue(boundProfile.uses)){
         c[0].SetTempSelectedQoSProfile(components[0].profiles.get(i));
55       c[0].SetCurrentProfilePos(i+1);
         if(FindAppropriateProfiles()) {
           for(int k=0; k<components.length; k++) {
             c[k].SetSelectedQoSProfile(c[k].GetTempSelectedQoSProfile());
           // change bound with the new value
60         boundProfile = new UserProfile();
           boundProfile.uses = c[0].GetSelectedQoSProfile().provides;
         } else break; // break is a termination condition
       }
     }
65 }
```

`ConformanceCheck()` (Line 51) performs conformance consistency check to every connected pair of components: (C_i, C_j) where C_i is the parent of C_j. It removes QoS-Profiles from the domain of C_i for which no conformant profiles have been specified in C_j. It returns false if there cannot be conformance between at least two connected components.

```
   int FindAppropriateProfiles()
   {
     FindConformantProfiles(CG.On_Client);
     if(CheckResourceConstraints(CG.On_Client)) {
70     FindConformantProfiles(CG.Across_Contaners\CG.On_Client);
       if(CheckResourceConstraints(CG.Across_Contaners)) {
         FindConformantProfiles(CG.On_Server\CG.Across_Contaners);
         if(CheckResourceConstraints(CG.On_Server))
           return 1; // successful
75       else return 0;
       } else return 0;
     } else return 0;
   }
```

`FindConformantProfiles()` (Line 79) 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. `IsMatching()` (Line 92) checks the conformance between two interacting components. Conformance [3] exists between two QoS-profiles

of interacting components when the server's provided-QoS contract conforms to the client's required-QoS contract. If the interacting components are on different containers (as identified by `AreOnDifferentNodes()` (Line 90)), the connector properties are required during conformance check. A component may belong to two groups in CG (Line 28). 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 (Lines 70,72) is read as "less". `CheckResourceConstraint()` (Lines 69,71,73) returns true when there are enough resources for the current selection.

```
   void FindConformantProfiles(CG componentGroup)
80 {
     int lowerIndex = GetLowerIndex(componentGroup);
     int higherIndex = GetHigherIndex(componentGroup);
     if(lowerIndex==0)
       int startingIndex = lowerIndex+1; // as the profile of the
85           // front-end component has been already selected
     else
       int startingIndex = lowerIndex;
     for(int j=startingIndex; j<=higherIndex;j++) {
       Connector tempConn = null;
90     if(AreOnDifferentNodes(c[j-1],c[j])) tempConn = connector;
       for(i=c[j].GetCurrentProfilePos(); i<c[j].profiles.size(); i++) {
         if(IsMatching(c[j-1].GetTempSelectedQoSProfile().uses,
                   c[j].profiles.get(i).provides, tempConn)) {
           c[j].SetTempSelectedQoSProfile(c[j].profiles.get(i));
95         c[j].SetCurrentProfilePos(i); break;
         }
       }
     }
   }
```

Let's next see the outcome of a negotiation for the example depicted in Fig. 3. Suppose the various input conditions are:

- user's QoS requirement: $frameRate > 12fps, resolution = 176 \times 144$; resolution is preferred over frame rate,
- resource availability: at the client's node, CPU=80%, memory=150KB; at the server's node, CPU=50%, memory=200KB; and the end-to-end bandwidth is 1Mbps,
- QoS contracts of the components are as given in Fig. 3.

As the first solution, `FineGrainedNegotiation()` selects the 5th QoS profiles of `VideoPlayer` and `VideoServer`, i.e. the ones with the offered QoS contract of $176 \times 144, 15fps$. Further iterations improve the solution and ultimately the 6th QoS-profiles of `VideoPlayer` and `VideoServer` are selected. The next step is to establish contracts between `VideoPlayer` and `VideoServer` and between `VideoPlayer` and User. These contracts will then be monitored and enforced by the run-time system.

4.3 Experiences

In the various application scenarios we studied, we had to conduct an experiment to specify the QoS contracts of components. We used these data to check the

validity of our approach and its prototype implementation. In our prototype 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 gives a solution that has the highest utility as far as the most preferred QoS dimension is concerned. The run-time complexity of the negotiation algorithm is $O(nd^2)$ where n is the total number of cooperating components and d is the number of QoS-Profiles specified for each component. Such a complexity is achieved by assuming that the cooperating components form a tree so as to achieve a non-backtracking solution (Lines 68-77).

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. 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.

5 Conclusions and Outlook

We presented a QoS contract negotiation framework that can be integrated in a component container using the interceptor pattern, which enables adding cross-cutting concerns like contract negotiation. The framework acts as a run-time support environment when QoS contracts are negotiated in various applications. As a proof-of-concept we have developed a prototype of the proposed framework. This paper discussed the implementation details of the prototype. We also illustrated how the framework can be applied to perform negotiation in a componentized video streaming example application. In the future we plan to extend our implementation to include contract monitoring and the consideration of policy constraints.

References

1. J. Ø. Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo, 2001.
2. A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *IEEE Computer*, 32(7):38–45, July 1999.
3. S. Frolund and J. Koistinen. Quality-of-Service specification in distributed object systems. *IOP/BCS Distributed Systems Engineering Journal*, December 1998.
4. S. Göbel, C. Pohl, R. Aigner, M. Pohlack, S. Röttger, and S. Zschaler. The COMQUAD component container architecture and contract negotiation. Technical Report TUD-FI04-04, Technische Universität Dresden, April 2004.

5. S. Göbel, C. Pohl, S. Röttger, and S. Zschaler. The COMQUAD Component Model—Enabling Dynamic Selection of Implementations by Weaving Non-functional Aspects. In *3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, Lancaster, UK, 22–26 Mar. 2004.
6. C. Lee, J. Lehoczky, R. Rajkumar, and D. P. Siewiorek. On quality of service optimization with discrete qos options. In *IEEE Real Time Technology and Applications Symposium*, pages 276–, 1999.
7. J. Loyall, R. Schantz, J. Zinky, and D. Bakken. Specifying and measuring quality of service in distributed object systems. In *Proc. 1st Int'l Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98, Kyoto, Japan)*, Apr. 1998.
8. D. A. Menascé, H. Ruan, and H. Gomaa. A framework for qos-aware software components. In *The fourth international workshop on Software and performance*, pages 186–196, Redwood Shores, CA, USA, 2004.
9. M. Mulugeta and A. Schill. An approach for QoS contract negotiation in distributed component-based software. In *In the 10th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2007)*, Medford (Boston area), Massachusetts, USA, July 2007. To appear.
10. Object Management Group. UML profile for modeling quality of service and fault tolerance characteristics and mechanisms, v1.0. OMG Document, May 2006. URL http://www.omg.org/docs/formal/06-05-02.pdf.
11. T. Ritter, M. Born, T. Unterschutz, and T. Weis. A QoS metamodel and its realization in a CORBA component infrastructure. In *proceedings of the Hawaii International Conference on System Sciences*, 2003.
12. S. Röttger and S. Zschaler. CQML$^+$: Enhancements to CQML. In J.-M. Bruel, editor, *Proc. 1st Int'l Workshop on Quality of Service in Component-Based Software Engineering, Toulouse, France*, pages 43–56. Cépaduès-Éditions, June 2003.
13. R. Staehli, F. Eliassen, and S. Amundsen. Designing adaptive middleware for reuse. In *ARM '04: Proceedings of the 3rd workshop on Adaptive and reflective middleware*, pages 189–194, New York, NY, USA, 2004. ACM Press.
14. E. P. K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London and San Diego, 1993.