Component QoS Contract Negotiation in Multiple Containers

Mesfin Mulugeta and Alexander Schill

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

Abstract. 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. QoS contract negotiation is used to select concrete QoS contracts between the collaborating components. This paper presents an algorithm for the QoS contract negotiation of components deployed in multiple component containers. Our algorithm addresses: possible resource constraints at each node and the network, the efficiency of the negotiation process, the selection of a heuristically optimized solution, and over-constrained cases. As a basis to our approach, we used the notion that the required and provided non-functional properties as well as resource demand are specified at the component level. To demonstrate the presented ideas, the interaction of a customer, video provider, and payment provider example scenario is analyzed.

1 Introduction

Component-based software engineering allows the composition of complex systems and applications out of well-defined parts. Nowadays, several mature and commercial component models (e.g. EJB, JavaBeans, .NET, COM+, etc.) exist in the market. However, 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 these mainstream component technologies, a lot of research has been and is currently being done. These efforts range from creating new component models [7] to extending the existing mainstream component models [18][16][17]. In realizing all these, various issues have to be addressed like the specification of component contracts, resource management, negotiation, adaptation, etc. Our work focuses on contract negotiation. Before highlighting the challenges in this area, we first give a brief note on component contracts and the specification languages our work applies.

Components in EJB, .NET, and COM+, are specified only with interfaces that provide syntactical information about which methods are available and how to invoke them. A contract greatly extends the component specification precision [19]. The term contract (in relation to components) can very generally...
be taken to mean "component specification" in any form [3]. Four different levels of contracts have been identified for components in [2]. These are: syntactic, behavioral, synchronization, and QoS contracts. Our focus in this paper is on the fourth type of component contract, which is the QoS contract.

A component’s QoS contract is differentiated into offered QoS contract and required QoS contract [9]. The offered QoS contract specifies the quality values that a component can provide to its clients (other components) while the required QoS contract specifies the QoS constraints that the servers of a component must achieve. There is a dependency between offered and required QoS contracts. There exist different description languages for specifying the QoS contract of components [5] [11]. CQML+ [11], which is an extension of CQML [1], provides the construct QoS-profile that is used in specifying a component’s offered QoS contract, required QoS contract, and its resource demand. The notion of components with different QoS-profiles, which this paper applies, has been employed in the research project COMQUAD [7][5] at TU Dresden.

A component-based application requires the appropriate composition of the contracts at the different ports of the 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 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 QoS contracts of each component and compose the whole application during design, implementation, or deployment time. But, for composing QoS contracts which depend on runtime resource conditions (e.g. network bandwidth, CPU) or quality attributes fixed dynamically, the selection of appropriate QoS contracts must be carried out at runtime through the process of contract negotiation. In our approach, the runtime environment, in particular the component container, performs the selection of the appropriate concrete QoS contracts for the components at runtime.

We tackle the QoS contract negotiation problem by first modeling it as a constraint satisfaction problem. Based on this, we propose an algorithm that selects concrete QoS contracts for components distributed in multiple containers. Some of the issues that we address are: (i) resource constraints at each node and the network, (ii) efficiency of the negotiation process, and (iii) over-constrained cases. We treat efficiency from two different aspects. The first is from the point of view of the selection of QoS contracts that heuristically optimizes the solution for meeting the negotiation goal (e.g. maximizing user’s satisfaction). The complexity of the selection process increases exponentially with the number of components and the QoS contracts they provide. 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.

The paper is structured as follows. Section 2 details our modeling of the contract negotiation as a Constraint Satisfaction Problem. In Section 3 we discuss the proposed approach by providing a heuristic algorithm for the negotiation, a discussion on over-constrained situations, and a comparison on three
architectures for realizing the algorithm. Section 4 demonstrates the presented ideas by taking an example scenario. The paper closes with an examination of related work, a summary, and outlook to future work.

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

Before formalizing the QoS contract negotiation as a CSP, we would describe first the notion of conformance [5] that should exist between QoS-profiles. A conformance between two QoS-profiles exists when the constraints in the QoS contract of the server component satisfies the constraints in the QoS contract of the client component. For e.g., the constraint \( \text{delay} < 5 \) conforms to the constraint \( \text{delay} < 10 \). When the components are distributed across containers, the relationship must take into account the influence of the network and containers between the interacting components as demonstrated in Section 4.

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 near-optimal solution (heuristic optimization) among the set of possible solutions.

A CSP consists of \( n \) variables \( x_1, x_2, ..., x_n \) whose values are taken from finite, discrete domains \( D_1, D_2, ..., D_n \) respectively, and a set of constraints on their values. In general, a constraint is defined by a predicate. That is, the constraint \( p_k(x_{k1}, ..., x_{kj}) \) is a predicate that is defined on the Cartesian product \( D_{k1} \times ... \times D_{kj} \). This predicate is true iff the value assignment of these variables satisfies this constraint [20]. Solving a CSP is equivalent to finding an assignment of values to all variables such that all constraints are satisfied.

For the formalization, we take the variables to be the QoS-profiles of the collaborating components \( C_1, C_2, ..., C_n \). There are \( n \) variables \( P_1, P_2, ..., P_n \) each representing QoS-profiles of \( C_1, C_2, ..., C_n \) respectively. The domain of each variable is the set of all QoS-profiles provided for a component. The constraints are classified as conformance, user’s and resource. As explained earlier, a QoS-profile \( P_i \) defines the offered 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. \( \text{delay}, \text{frameRate} \)) are constrained with values (e.g. \( \text{delay} < 5 \) or \( \text{frameRate} = 15s^{-1} \)). Suppose \( d_1, d_2, ..., d_k \) are the QoS characteristics considered, 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 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. This has not been distinguished for the purpose of clarity of the presentation.

As an example, for the application that involves three components (Fig. 1), the QoS contract negotiation is formalized as in Table 1 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 response time (RT) property, the constraint can be modified as: \((P_i.\text{Required}.RT \implies P_i.\text{Offered}.RT + \text{delay in network and containers})\), where \(\text{delay in network and containers}\) is assumed to be a constant for the period the negotiation agreement is valid. In the resource constraint, for two containers case, instead of only one relation, three relations must be provided. These are for components deployed on the client, on the server, and for components connected across containers.

\textbf{Resources} considered in \(P_i\) 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 \(C_i\), \(R_{\text{all}}\) is the resource demand of the \(n\) components, and \(R_{\text{avail}}\) is the available resource. That is,

\[
R_i = [r_{i1}, r_{i2}, ..., r_{im}]
\]

where \(r_{ij}\) is the \(j^{th}\) resource demand of \(C_i\),

\[
R_{\text{avail}} = [r_{1\text{avail}}, r_{2\text{avail}}, ..., r_{m\text{avail}}]
\]

\[
R_{\text{all}} = R_1 + R_2 + ... + R_n
\]

\[
R_{\text{all}} = [r_{11} + r_{21} + ... + r_{n1}, r_{12} + r_{22} + ... + r_{n2}, ..., r_{1m} + r_{2m} + ... + r_{nm}]
\]

where the addition used is arithmetic.

---

Table 1. Component QoS contract negotiation as a constraint satisfaction problem

<table>
<thead>
<tr>
<th><strong>Variables</strong></th>
<th>(P_1, P_2,) and (P_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Domains:</strong></td>
<td>for each component, a set of QoS-Profiles are specified by the component developer</td>
</tr>
<tr>
<td><strong>User’s constraint</strong></td>
<td>user’s QoS Requirement on (d_i &gt; P_i.\text{Offered}.d_i)</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>user’s QoS Requirement on (d_i &gt; P_i.\text{Offered}.d_i)</td>
</tr>
<tr>
<td><strong>Conformance</strong></td>
<td>((P_i.\text{Required}.d_i \implies P_i.\text{Offered}.d_i))</td>
</tr>
<tr>
<td>constraint</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>((P_i.\text{Required}.d_k \implies P_i.\text{Offered}.d_k))</td>
</tr>
<tr>
<td></td>
<td>((P_i.\text{Required}.d_i \implies P_i.\text{Offered}.d_i))</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td><strong>Resource</strong></td>
<td>(P_1.\text{Resources} + P_2.\text{Resources} + P_3.\text{Resources} \leq \text{Resources}_{\text{avail}})</td>
</tr>
<tr>
<td>constraint</td>
<td></td>
</tr>
</tbody>
</table>
We require there are enough resources, and hence the resource constraint is satisfied, if:

\[
\begin{align*}
    r_{1}^{\text{avail}} & \geq r_{11} + r_{21} + \ldots + r_{n1}, \\
    r_{2}^{\text{avail}} & \geq r_{12} + r_{22} + \ldots + r_{n2}, \\
    & \ldots \\
    r_{m}^{\text{avail}} & \geq r_{1m} + r_{2m} + \ldots + r_{nm}
\end{align*}
\]

There are enough resources for the \( n \) components iff for each type of resource, the resource requirement is not larger than the corresponding available resource.

3 The Proposed Approach

To find a near-optimal solution for the resulting CSP (Table 1), in a naive approach, all possible configurations (assignments) must be considered. Then, QoS-profiles that give the highest utility and where all constraints are met are selected. This approach requires exhaustive searching of all possible configurations. In general, constraint satisfaction is NP-hard [20]. For applications with few possible configurations, the naive approach can be used. In other cases, heuristics must be applied to improve the efficiency of the naive approach. Next, we discuss how we have tailored general-purpose heuristic mechanisms and our own problem-specific heuristics.

3.1 Variable and Value Ordering

A variable and value ordering is a general purpose heuristics used to solve CSPs efficiently [12]. In this method, the variable to assign next is appropriately selected. Having selected a variable, the value to assign to must also be appropriately chosen. We have tailored this method as we seek a near-optimal solution.

The variables (QoS-profiles) are ordered for assignment by topologically sorting the network of cooperating components. We call the minimal element the front-end component (e.g. C1 in Fig. 1), which is also the one that interacts with the user. The assignment starts from this minimal element. At any point in time, the QoS property of the partially completed assignment can be taken as the offered QoS contract of the front-end. This is one advantage achieved by this ordering as we know right from the start and during successive assignments whether or not a partial assignment meets the user’s constraint. The values of each variable, i.e. the QoS-profiles of each component, must be ordered from higher to lower quality. As contracts involve multiple QoS properties, the ordering is based on first the QoS property that is the most preferred by the user, then the next preferred, etc.

Using a centralized approach, we can find the near-optimal solution by considering all the components (in both containers) as a whole. Starting from the front-end component and its profile at the top of the list, we continue selecting QoS-profiles in the topologically ordered set. In doing so, if we cannot get
Because of the ordering of all profiles from higher to lower quality, the first solution obtained is the near-optimal one. However, the ordering may not necessarily be a heuristics for finding any acceptable solution (but this was not our aim!) as properties like smallest domain size, maximum degree, etc. help in speeding up search in CSPs. But the latter approach would require us to exhaustively look all possible solutions in order to get the one which is near-optimal.

3.2 Structure of the Problem

In the above, we tried to find the near-optimal solution by considering all the components as a whole. But, generally breaking a problem into sub-problems helps in finding the solution quickly. In the two container case, the problem can be divided into three sub-problems as shown by the categorization of components in Fig. 2. This subdivision is logical as separate resource constraints apply to each group. We refer the groups as: components on the client, components connected across containers, and components on the server.

An algorithm that combines the structuring of the problem with the variable and value ordering heuristics can be given as follows. For components on the client, the client container selects appropriate QoS-profiles. In the next steps, based on the already selected profiles, appropriate profiles for components across containers and components on the server are selected by the server container. This algorithm resembles the synchronous backtracking algorithm in [20].

There are two shortcomings of this algorithm: (i) it would lead to more back-trackings if the bottleneck resource were either the network bandwidth or server resources, and (ii) the method is sequential even though the problem has been

![Diagram](image.png)
decomposed into sub-problems and lends itself for concurrent executions. We define a bottleneck resource as the first limiting resource in a step-by-step (from lower to higher quality) selection of QoS-profiles, which aims at finding a near-optimal solution. An algorithm that avoids the above stated shortcomings will have to first identify the location of the bottleneck resource and first selects QoS-profiles that maximally use this resource. The algorithm we propose (Listing 1.1), thus, determines the bottleneck resource first and refines the solution when more QoS-Profiles are available.

Listing 1.1. QoS contract negotiation algorithm for components deployed in client and server containers

```c
enum ComponentGroup { On_Client, On_Server, Across Containers } /* On_Client, On_Server, and Across Containers refer to components on the client, components on the server, and components connected across containers respectively */

void BuildSolution()
{
    betterSolution = true;
    while(betterSolution)
    {
        if(FindConformantProfiles(On_Client))
        {
            if(CheckResourceConstraints(On_Client))
            {
                if(FindConformantProfiles(Across Containers \ On_Client))
                {
                    if(CheckResourceConstraints(Across Containers))
                    {
                        if(FindConformantProfiles(On_Server \ Across Containers))
                        {
                            if(CheckResourceConstraints(On_Server))
                            {
                                CommitSelectedProfiles();
                                continue;
                            }
                            else /* Bottleneck is Server */
                            {
                                betterSolution = false;
                                SetPreviousSelection(On_Server);
                                DoFurtherNegotiations(On_Client \ Across Containers);
                            }
                        }
                        else /* no conformance for components on server */
                        {
                            // backtrack to previous FindConformantProfiles()
                        }
                    }
                    else /* Bottleneck is Bandwidth */
                    {
                        betterSolution = false;
                        SetPreviousSelection(Across Containers);
                        DoFurtherNegotiations(On_Client \ Across Containers);
                        DoFurtherNegotiations(On_Server \ Across Containers);
                    }
                }
                else /* no conformance for components across containers */
                {
                    // backtrack to previous FindConformantProfiles()
                }
            }
            else /* Bottleneck is Client */
            {
                betterSolution = false;
                SetPreviousSelection(On_Client);
                DoFurtherNegotiations(On_Server \ Across Containers);
            }
        }
        else break; /* no conformance for components on client */
    }
```
Listing 1.1 provides an algorithm that finds QoS-profiles that fulfill the user’s, conformance, and resource constraints, if a solution exists and No Solution otherwise. To determine whether or not a solution exists, a list is used by the algorithm. Let the name of this list be selectedProfiles. After the execution of BuildSolution(), selectedProfiles contains the selected profiles if a solution exists, or it is empty for No Solution. No Solution occurs when the algorithm exits before executing CommitSelectedProfiles(). The categorization of components is shown as enumeration type ComponentGroup. Most of the functions invoked in Listing 1.1 take a type of ComponentGroup as input argument. FindConformantProfiles(.) finds QoS-profiles, which are conformant to one another for all the components specified in the input argument. At each iteration FindConformantProfiles(.) improves the solution by one step based on the specified QoS-profiles. CheckResourceConstraint(.) checks whether there is enough resource for the current selection. DoFurtherNegotiation(.) applies the same approach of finding the solution, for a subset of components while QoS-profiles have already been determined for other components. During intermediate steps of the algorithm, QoS-profiles are temporarily selected. We say, a possible solution is obtained when selections are made for all components. This selection is saved on selectedProfiles with CommitSelectedProfiles(). After temporarily selecting profiles and when it is found out that the partial solution would not lead to a solution, the temporary selection would be rolled back to the previous selection with SetPreviousSelection(.)

A component may belong to two groups in ComponentGroup. For example, a component deployed on the client container and that also communicates across containers belongs to OnClient and AcrossContainers. The notation \ in Listing 1.1 is read as ”less” and & is read as ”and”.

3.3 Dealing with Over-Constrained Situations

The algorithm in Listing 1.1 has been modeled with hard constraints. The output is always expected to give a solution that meets all the constraints. No Solution will be provided, 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. 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. Next, we discuss how to extend the algorithm in Listing 1.1 to handle some of the over-constrained cases.

One of the approaches used to deal with over-constrained CSPs is to extend the CSP framework into what are called Partial Constraint Satisfaction Problems [4]. Maximal constraint satisfaction algorithms, which seek a solution that satisfies as many constraints as possible, have been given in [4]. The algorithms compare potential partial solutions based on the number of constraints violated and choose the one that results in a minimum number of constraint violations. These are some general metrics and the nature of the problem at hand should dictate the strategy to be used in finding the partial solution.
In our case, we have three different types of constraints: user’s, conformance, and resource. The conformance and user’s constraints apply to each QoS property. It is difficult to compare violations of resource constraints with conformance constraints and hence simply counting the violated constraints is not applicable. In search of a partial solution, which is achieved by weakening the original problem, it is important to see which constraints to tolerate for violation. Allowing either resource or conformance constraints to be violated would make it difficult to predict the QoS property of the whole application. Currently, we are following the strategy where the user’s constraint is systematically violated to reach to a solution. User’s constraint relaxation helps in finding a solution when, (i) resource constraints cannot be met although a configuration that meets the other constraints exists, and (ii) the conformance constraints cannot be met when searching for a solution.

For the non over-constrained problem, the algorithm in Listing 1.1 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 near-best solution. It successively goes one step down on the list of QoS-profiles (which are sorted based on the user’s preferences) of the front-end component to degrade the user’s QoS requirements. The algorithm in Listing 1.1 is then applied to the weakened problem. If this step also produces No 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 can’t be further relaxed.

The assumptions we have made concerning how user’s requirements and preferences are captured are: a user can define the relative weights of the different QoS properties; and a user can define the minimum quality of the output below which it is unacceptable. This is done corresponding to each QoS property. A more robust user interface can be designed but it is beyond the scope of this paper. 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 we are currently studying to incorporate 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 in only $m$ (where $m < n$) properties. This might imply, the negotiation can continue even if there are contract violations in the $n - m$ QoS properties.

3.4 Centralized Versus Distributed Solution

We have identified three approaches for implementing the algorithm in Listing 1.1: (i) centralized solution - only one of the containers makes the selection of QoS-profiles, (ii) distributed solution - both containers are responsible for the selection of appropriate QoS-profiles, and (iii) hybrid solution - combines centralized and distributed approaches.

In the centralized solution, the responsible container must have access to the configuration information of all deployed components and the resource availability at each container. The distributed solution doesn’t require this as each
container is responsible for the selection of profiles of components it hosts and communicates with the other container to resolve constraints that concern both of them. In the hybrid solution, the bottleneck resource is first isolated using a centralized approach so as to minimize the number of interactions (message exchanges) between the distributed containers and then distributed approach is applied to harness concurrent executions.

Owing to the nature of our problem (which is not loosely coupled), a pure distributed solution results in large inter-container communication overhead and hence we do not consider it as an option. 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 could take place after contracts have already been established. The number of inter-container communication and level of concurrency are taken as measures for the comparison.

For both the centralized and hybrid solutions, inter-container communication is required when: the QoS-profiles of interacting components and local resource conditions are exchanged just before the negotiation. The processor and other pertinent information of each node need to be exchanged together with the QoS-profiles. This is used to convert the resource demand of components that have been measured in a different environment.

In addition, in the centralized solution, inter-container communication is required when: (i) the established contracts are communicated finally. For the hybrid solution, inter-container communication is also required when: (i) the two containers agree to concurrently do negotiations locally, and selected QoS-Profiles that concern both are communicated, and (ii) each container confirms the completion of the independent negotiations and exchanges relevant established contracts.

The centralized solution performs better than the hybrid one in terms of minimizing the inter-container communication. On the other hand, the hybrid solution can take advantage of the concurrent executions of both containers and attain a better solution, especially if the required level of computation is high.

As contracts can be violated, monitoring of contracts is constantly performed. For instance, if a contract is established between VideoPlayer and VideoServer (see Section 4), the contract contains values of the parameters frameRate the VideoPlayer expects from the received stream and the frameRate 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 contract re-negotiation phase, the centralized solution requires the exchange of information between the containers, when (i) any contract violation is detected, and (ii) during monitoring of available resources (which is a periodic activity). Nevertheless, for the hybrid solution, no such communication is required as 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.

The decision for applying either a centralized or hybrid solution can be made just before the start of the negotiation. 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 (e.g. due to large number of components and QoS-profiles) and the environment enables relatively faster inter-container communication, the hybrid solution would be the more efficient approach. During run-time re-negotiations, the hybrid solution is preferred as it avoids unnecessary overheads when it is applicable to perform re-configurations locally.

3.5 Negotiation Ordering

Negotiation ordering is about negotiating one property before another, for example, security properties being negotiated before response time. In this paper, we follow the strategy of negotiating all properties in an atomic manner, i.e. without any order. This is natural as dependencies may exist among the properties. For example, for a QoS-Profile that specifies response time and security properties, the response time among others is influenced by the security attributes selected. If the key length in the encryption is large and some sophisticated algorithm is used, the response time will be higher. This makes it logical to negotiate response time and security properties in an atomic manner.

We are currently working on a strategy that follows order of negotiation between what we call coarse-grained and fine-grained properties. For example, negotiations on whether or not to do confidential communication must precede negotiation on particular security mechanisms. Or, negotiation on protocol or coding properties (coarse-grained) in a video streaming application must go before frame rate and resolution (fine-grained) negotiation. The rationale for this ordering is the fact that negotiation on certain properties makes sense if there is an agreement on other properties.

The procedure we can follow in the coarse-grained property negotiation between a client and server component is outlined as below. It assumes that each property is specified with multiple ordered values.

1. Take the first preference of the client component
2. Search for a conformant value in the corresponding property of the server component
3. If there is a matching, register the conformant pair in a temporary agreement list.
4. Proceed to the next value in the list of preference of the client component and go to step 2)
5. The procedure ends when all values are exhausted in the client component’s preference list

At the end of the coarse-grained negotiation, the state of the temporary agreement list (Step 3 above) may: (i) be empty - the whole negotiation process stops,
or (ii) have a single value - the coarse-grained negotiation is concluded at this step, or (iii) have multiple values - final choices have to be made during the fine-grained negotiation phase.

For the fine-grained negotiation, we use the algorithm in Listing 1.1. Doing this two-level negotiation reduces the complexity of the search. This is also demonstrated in the Example Section. Incorporating strategies for negotiation ordering between two fine-grained properties (e.g. frame rate and resolution) or between two different QoS categories (e.g. security and timeliness) as may be required by users is the subject of our future research.

4 Example

An application scenario (Fig. 3) we discuss involves the interaction of a customer, video stream provider, and payment provider in the process of video streaming and payment transactions. Two use cases considered are 'order service' and 'stream video'. The 'order service' is concerned with the handling of a user’s request for a movie up to the payment process while 'stream video' involves the streaming of the movie from the video provider to the customer.

The Booking component receives, through the GUI, information about the title of the movie, customer’s data (e.g. name, ID), payment method, etc. Booking contacts the movie and customer databases to retrieve pertinent information about the movie (if it exists) like the amount of charge, the mode of payment, etc., and the customer’s credit card information. Booking then sends the credit card number together with the charge to PaymentProcessor for the latter to process the payment. After successfully transacting the payment, the requested movie will be streamed from the VideoServer to the VideoPlayer. VideoServer reads pre-encoded media files from the server and streams video and audio to clients. It is capable of streaming at different frame rates, resolutions, media codings, and protocols. VideoPlayer reads the streamed media for playback at the customer’s node.

As far as 'order service' is concerned, customers may have security and performance requirements. They require credit card information be sent to PaymentProcessor confidentially. At the same time, when the payment transaction is made, it should not be slow. Some users may prefer stronger security mechanisms and tolerate the response time. PaymentProcessor and Booking are specified
Table 2. QoS-Profiles of GUI, Booking, and PaymentProcessor Implementations

<table>
<thead>
<tr>
<th>GUI</th>
<th>Booking</th>
<th>PaymentProcessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>uses IOrder</td>
<td>provides IOrder</td>
<td>provides IOrder</td>
</tr>
<tr>
<td>(response time in sec)</td>
<td>(response time in sec)</td>
<td>(key, algorithm, response time in sec)</td>
</tr>
<tr>
<td>7</td>
<td>13</td>
<td>PP, RSA, 13</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>Shared, 3DES, 7</td>
</tr>
<tr>
<td>18</td>
<td>5</td>
<td>Shared, DES, 3</td>
</tr>
</tbody>
</table>

and implemented to give different levels of security and response time to entertain the various requirements. For ‘stream video’, the user’s requirements are higher resolution and good frame update rate, where relative preferences on these attributes may vary depending on the video’s content.

The collaborating components for ‘order service’ include GUI, Booking, and PaymentProcessor while for ‘stream video’, VideoPlayer and VideoServer. In the former the front-end component is GUI while in the latter VideoPlayer.

Let us assume that Booking implements two interfaces: a uses interface IOrder (to get a service from PaymentProcessor) and a provides interface IOrder (to offer the required service to GUI). Likewise, PaymentProcessor and GUI implement a provides interface IPayment and a uses interface IOrder respectively. For the sake of readability, the profiles of these components are tabulated as in Table 2. The resource demands of each profile haven’t been shown in Table 2. This is provided in Table 3. The concepts of specifications from [8] has been applied to incorporate security attributes in the QoS-profiles of components. In Table 2, PP− is the payment provider’s private key whereas Shared is a particular shared secret key between the video and payment providers.

VideoServer implements a provides interface ICompVideo. VideoPlayer implements two interfaces: a uses interface ICompVideo and a provides interface IUnCompVideo. The COMQUAD component model supports streams as special interface types [7] and allows to specify non-functional properties for them. We conducted an experiment to specify VideoPlayer and VideoServer. The VideoPlayer component was implemented using the JMF [14] framework and the VideoServer component abstracts the video media file, where the streams are pull data sources. The original media file has been encoded into many files with differing frame rates, resolutions, and encoding algorithm.

The PC where the VideoPlayer ran and the measurement was conducted has a configuration of: AMD Athlon(tm) XP 1600+, 1GB RAM, Windows 2000 Professional, 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 have approximate data, which can enable us to validate our contract negotiation protocol and algorithm. Windows Performance Monitor has been used to measure the resource usage of the components. During the measurements, average bandwidth and CPU percentage time have been considered. The measured QoS-profiles of VideoPlayer and VideoServer, with properties UDP protocol and mp42 coding, is given in Table 3.
Table 3. QoS-Profiles of VideoPlayer and VideoServer Implementations

<table>
<thead>
<tr>
<th>VideoPlayer</th>
<th>VideoServer</th>
</tr>
</thead>
<tbody>
<tr>
<td>provides IUnCompVideo (resolution, frame rate in s⁻¹)</td>
<td>provides ICompVideo (resolution, frame rate in s⁻¹)</td>
</tr>
<tr>
<td>uses ICompVideo (resolution, frame rate in s⁻¹)</td>
<td>Resource (CPU in %, bandwidth in Mbps, memory MB)</td>
</tr>
<tr>
<td>Resource (CPU in %, bandwidth in Mbps, memory MB)</td>
<td></td>
</tr>
<tr>
<td>352x288, 30</td>
<td>352x288, 30</td>
</tr>
<tr>
<td>352x288, 15</td>
<td>352x288, 15</td>
</tr>
<tr>
<td>352x288, 10</td>
<td>352x288, 10</td>
</tr>
<tr>
<td>352x288, 5</td>
<td>352x288, 5</td>
</tr>
<tr>
<td>352x288, 1</td>
<td>352x288, 1</td>
</tr>
<tr>
<td>176x144, 30</td>
<td>176x144, 30</td>
</tr>
<tr>
<td>176x144, 15</td>
<td>176x144, 15</td>
</tr>
<tr>
<td>176x144, 10</td>
<td>176x144, 10</td>
</tr>
<tr>
<td>176x144, 5</td>
<td>176x144, 5</td>
</tr>
<tr>
<td>176x144, 1</td>
<td>176x144, 1</td>
</tr>
<tr>
<td>128x96, 30</td>
<td>128x96, 30</td>
</tr>
<tr>
<td>128x96, 15</td>
<td>128x96, 15</td>
</tr>
<tr>
<td>128x96, 10</td>
<td>128x96, 10</td>
</tr>
<tr>
<td>128x96, 5</td>
<td>128x96, 5</td>
</tr>
<tr>
<td>128x96, 1</td>
<td>128x96, 1</td>
</tr>
<tr>
<td>352x288, 30</td>
<td>2165</td>
</tr>
<tr>
<td>352x288, 15</td>
<td>2146</td>
</tr>
<tr>
<td>352x288, 10</td>
<td>2076</td>
</tr>
<tr>
<td>352x288, 5</td>
<td>1852</td>
</tr>
<tr>
<td>352x288, 1</td>
<td>1644</td>
</tr>
<tr>
<td>176x144, 30</td>
<td>321</td>
</tr>
<tr>
<td>176x144, 15</td>
<td>252</td>
</tr>
<tr>
<td>176x144, 10</td>
<td>208</td>
</tr>
<tr>
<td>176x144, 5</td>
<td>135</td>
</tr>
<tr>
<td>176x144, 1</td>
<td>34</td>
</tr>
<tr>
<td>128x96, 30</td>
<td>152</td>
</tr>
<tr>
<td>128x96, 15</td>
<td>120</td>
</tr>
<tr>
<td>128x96, 10</td>
<td>108</td>
</tr>
<tr>
<td>128x96, 5</td>
<td>70</td>
</tr>
<tr>
<td>128x96, 1</td>
<td>24</td>
</tr>
</tbody>
</table>

As the measurements have been taken under light load conditions, it is assumed that the bandwidth requirement of VideoServer is 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 as it is too small to have influence in our validation.

Suppose the user’s QoS requirements and preferences for ‘stream video’ are:

– frameRate > 12fps, resolution = 176X144
– first preference is frame rate, then resolution

Let the available resources are: Customer: CPU 80%; Video Provider: CPU 50%; and end-to-end bandwidth: 1Mbps. The available CPU times given are already converted to be consistent with the environment where the QoS-profiles have been measured.

Before running the algorithm of Listing 1.1, the QoS-profiles of VideoPlayer and VideoServer must be sorted according to the user’s preference, i.e., first on frame rate and second in resolution. The QoS-profiles in Table 3 are sorted first on resolution and then in frame rate. The main steps in the negotiation are executed as follows.

BuildSolution() (Listing 1.1) is invoked to iteratively select heuristically optimized QoS-profiles. For a front-end component, FindConformantProfiles() uses both conformance and user’s constraints to find conformant profiles.

– 1st Iteration

• FindConformantProfiles(On_Client) gets a QoS-profile for VideoPlayer with the offered QoS contract: 15fps, 176X144
- CheckResourceConstraint(OnClient) is successful.
- FindConformantProfiles(Across_Containers\On_Client) will select a QoS-profile for VideoServer with the offered QoS contract: 15fps, 176X144
- CheckResourceConstraint(Across_Containers) is successful.
- FindConformantProfiles(On_Server\Across_Containers) will not add any additional selected profiles as there are no components on the server other than VideoServer.
- CheckResourceConstraint(On_Server) is successful in this iteration.

2nd Iteration
- FindConformantProfiles(On_Client) gets a new valid QoS-profile for VideoPlayer with the offered QoS contract: 15fps, 352X288
- CheckResourceConstraint(On_Client) is successful.
- FindConformantProfiles(Across_Containers\On_Client) will select a QoS-profile for VideoServer with the offered QoS contract: 15fps, 352X288
- CheckResourceConstraint(Across_Containers) is NOT successful. The temporarily selected QoS-profiles in this iteration (first and third steps) are invalidated. The appropriate configurations, hence, are those selected in the previous iteration, i.e. 1st iteration.

The bottleneck resource has been determined to be the network bandwidth in the 2nd iteration. Negotiations, however, would continue in the server and client containers with the DoFurtherNegotiations(.). As VideoPlayer and VideoServer are the only components on the client and server respectively, further negotiations in the two containers wouldn’t produce new profiles.

The selected profiles for VideoPlayer and VideoServer are the ones with the offered QoS contract 176X144, 30fps. Although the profiles with 176X144, 15fps satisfied all constraints, an improved solution was sought as the negotiation’s goal was the maximization of the user’s satisfaction.

Let’s consider now the case of an unsuccessful negotiation, which arises when trying to fulfill user’s QoS requirements while resource and/or conformance constraints can’t be met. Assume the user’s requirements and preferences are as previous and the available resources are: Customer: CPU 80%; Video Provider: CPU 50%; end-to-end bandwidth: 200Kbps.

The 1st iteration of Listing 1.1 can’t 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 (note that the available QoS-profiles are sorted according to the user’s preferences). In the first iteration of the relaxation step, the chosen degraded requirement is: frameRate = 15fps and resolution = 128X96. At this step, the negotiation turns out to be successful. If the user’s preference were first resolution and then frame rate, the relaxation step would have tried the sequence 176X144, 10fps; 176X144, 5fps; etc. and the negotiation would have been successful at 176X144, 5fps. If the
user’s requirement had been given in a range (like frame rate 5 - 15 fps or resolution 128X96 - 800X600), then this information would have been used in the relaxation step not to further relax the user’s requirement below the given minimum.

Table 3 shows QoS-profiles of VideoPlayer and VideoServer for a UDP protocol and mp42 encoding. Suppose VideoPlayer is specified with RTP/UDP and UDP protocols and mp42, mpg4, and h263 coding; and VideoServer specified with RTP/UDP, UDP, and TCP protocols and mp43, mp42, mpg4, and h264 codings. For all combinations of coding and protocol, QoS-profiles similar to Table 3 have to be specified. The coarse-grained property negotiation between VideoPlayer and VideoServer (see Section 3.5) culminates in the selection of: (i) mp42 and mpg4 coding, and (ii) RTP/UDP and UDP protocol. In both properties, multiple values have been selected and the final decision is made by the fine-grained negotiation, which is demonstrated above by following Listing 1.1 step-by-step.

For ‘order service’, assume the user has a higher preference to security than response time. Before making the contract negotiation for the three components (GUI, Booking, and PaymentProcessor), the QoS-profiles must be ordered first on security properties and then on response time, as it is done in Table 2. The algorithm in Listing 1.1 is applied to select profiles that provide the near-optimal solution. Conformance on security properties between two interacting components exists when the constraints on all the parameters (key type, key length, and algorithm) match exactly. For matching offer and expectation on response time, the conformance relationship 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).

Our algorithm selects appropriate profiles by improving from lower to higher quality to maximally satisfy the user’s requirements and preferences. For QoS-profiles that contain both security and response time properties (Table 2), if the user has a higher preference to security, the step-by-step increment leads to better security (e.g. from Shared, DES to PP^−, RSA), but this affects the response time negatively. In the case where conflicting properties exist the algorithm maximizes the satisfaction of the most preferred property.

5 Related Work

The work in [6] treats contract negotiation only in a single component container and has not pursued the case of distributed contract negotiation. Their approach ties the container-based negotiation to a real-time operating system called DROPS. But, in our case, the negotiation is not dependent on a particular platform. Furthermore, our algorithm treats the over-constrained case.

In [10] 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. The contract negotiation phase is 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. Our approach differs in that negotiation precedes every service invocation of the application. The service invocation may involve more than two components and hence the negotiation takes place among all the collaborating components. Furthermore, we have described the various contract negotiation mechanisms like negotiation goal, efficiency of the negotiation process, and over-constrained situations in a generic manner.

QuA [13] aims at defining an abstract component architecture, including the semantics for general QoS specifications. The proof of concept is provided by implementing an open framework for platform managed QoS. In QuA’s QoS-driven Service Planning has similarities to our concept of contract negotiation. Complexity issues, however, haven’t been accounted for in the service planning.

6 Conclusions and Outlook

A QoS contract negotiation is applied to select appropriate QoS contracts between collaborating components. Important challenges to be addressed are possible resource constraints at each node and the network, the efficiency of the negotiation process, and the selection of a heuristically optimized solution. Component QoS contracts are specified in QoS-profiles that contain the required and provided non-functional properties as well as resource demand. Due to its dynamic nature, the QoS contract is specified with multiple QoS-profiles.

We tackled the problem first by modeling the negotiation process as a constraint satisfaction problem. Based on this, we proposed an algorithm that selects concrete QoS contracts for the distributed components that meet the stated challenges and mechanisms for relaxing constraints when the contract negotiation couldn’t produce a solution. We also found out that a hybrid solution, that combines centralized and decentralized approaches, performs well for realizing the proposed algorithm.

We are currently working on incorporating the case of multiple clients in our negotiation model. In this paper, the negotiation goal we assumed was only the maximum satisfaction of one client (the user). But, in the multiple client scenario, we want to include, among others, the service provider’s needs as a parameter in the negotiation goal. We are planning to introduce negotiation policy constraints, to our existing model, which enables us to capture the different negotiation goals.

References


