DEPLOYING QoS CONTRACTS IN THE ARCHITECTURAL LEVEL

Sidney Ansaloni[1], Alexandre Sztajnberg[2], Romulo C. Cerqueira[1], Orlando Loques[1]

[1]Instituto de Computação – Universidade Federal Fluminense (UFF), Rua Passo da Pátria – Niterói – RJ – Brasil; [2]Instituto de Matemática e Estatística – Universidade do Estado do Rio de Janeiro, Rua São Francisco Xavier, 524 / 6018-D – Maracanã – RJ - Brasil

Abstract: This paper presents an approach to describe, deploy and manage software architectures having dynamic functional and non-functional requirements. The approach is centered on an ADL extended with high-level contracts, which are used to specify the non-functional requirements associated to the architecture of a given application. These contracts are also used to configure the infrastructure required to enforce the non-functional requirements and, during the running time, can be used to guide architecture adaptations, in order to keep them valid in face of changes in the supporting environment. The infrastructure required to manage the contracts follows an architectural pattern, which can be directly mapped to specific components included in a supporting reflective middleware. This allows designers to write a contract and to follow standard recipes to insert the extra code required to its enforcement in the supporting middleware.

Key words: QoS contracts ADL, architectural pattern, dynamic configuration

1. INTRODUCTION

The specification of QoS requirements and the implementation of the corresponding management strategies for the resource providers associated to the requirements are, generally, embedded in the application programming in an ad-hoc manner, mixed with the application's specific code. This lack of modularity makes evolution and code reuse difficult, also making difficult its

verification and debugging. In this context, there is a growing interest for handling quality of services (QoS) aspects in a specific abstraction level[1, 2, 3]. This approach would allow to single out the resources to be used and the specific mechanisms of the native system that will be required by the application, and, if possible, turn automatic the configuration and management of those resources.

The traditional notion of QoS is bound to communication level performance. However, a more recent view of QoS includes characteristics associated to application's non-functional aspects, such as availability, reliability, security, real-time, persistency, coordination and debugging support. Such kind of aspect can be handled by reusable services provided by middleware infrastructures or native systems support. This makes feasible to design a software system based on its architectural description, which includes the functional components, the interactions among those components, and requirements regarding the behavior of system QoS resources. To this end, it has to be provided a means to specify those requirements in the context of the application's architecture description and, also, there has to be available an environment that allows to deploy those requirements over the system resources. In some applications, such environment has to include mechanisms to monitor the resources and to manage adaptations, according to the availability of those resources, in order to guaranty that the QoS requirements are met during run-time.

Among the available techniques to specify QoS constraints, we highlight the concept of contracts [4]. A QoS contract establishes a formal relationship between two parts that use or provide resources, where rights, obligations and negotiation rules over the used resources are expressed. For instance, a parallel computing application can have a QoS contract defining rules to replicate processing resources, in order to guaranty a maximum execution time constraint. According to the specified contract, the application can have its components parallelization degree automatically controlled by the supporting environment. Thus, when the time constraint is not being met by the present configuration, the number of replicas can be raised, if there are available processors [5].

In the previous context, this work presents the CR-RIO framework (*Contractual Reflective - Reconfigurable Interconnectable Objects*)[2,5] conceived to specify and support QoS contracts, associated to the architectural components of an application. The approach helps to achieve separation of concerns[6] facilitating the reuse of modules that implement the computation in other application systems, and allows the non-functional requirements to be handled separately during the system design process. The framework includes a contract description language, which allows the definition of a specialized view of a given software architecture. The

supporting infrastructure required to impose the contracts during run-time follows an architectural pattern that can be implemented by a standard set of components included in a middleware. The results of our investigation point out that the code generation of these components can be automated, except some explicit parts of code related to specific contract and resources classes. In this way, contracts and their respective supporting infrastructures can be reused in different applications.

In the rest of this paper, we initially describe the key elements of the framework including the architecture description language with support to QoS contracts. Next, we present the supporting infrastructure and, based on an example we demonstrate the validity of the framework. Complementing the article we present some related proposals and provide some conclusions.

2. BASIC FRAMEWORK

The CR-RIO framework integrates the software architecture paradigm, which is centered in an architecture description language (ADL), with concepts such as reflection and dynamic adaptation capability[6], which are generally provided in an isolated fashion in middleware proposals described in the literature. This integration facilitates the achievement of separation of concerns, software component reuse and dynamic adaptation capability of applications. CR-RIO includes the following elements:

CBabel, an ADL used to describe the functional components of the application and the interconnection topology of those components, which follow the CR-RIO model. CBabel also caters for the description of non-functional aspects, such as coordination, distribution, planned reconfigurations and QoS. This set of features turns possible submitting CBabel descriptions to formal verification procedures[7]. A CBabel specification corresponds to a meta-description of an application that is available in a repository and is used to deploy the architecture in a given operating environment. In running time this meta-description provides the information required to manage architectural adaptations.

An **architecture-oriented component model**, that allows programming the software configuration of the application; (i) Modules, which encapsulate the application's functional aspects; (ii) Connectors, used in the architecture level to define relationships between modules; in the operation level connectors mediate the interaction between modules; and (iii) Ports, which identify access points through which modules and connectors provide or require services; ports are fundamental to allow component linking with low coupling.

A simple **software design methodology** that encourages the designer to follow a simple meta-level programming discipline, where functional aspects are concentrated in modules (base level) and non-functional aspects are encapsulated in connectors (meta-level). It is worth to point out that some QoS requirements can be directly mapped into connectors, which are equivalent to meta-level components, and can be configured in an application's architecture. For example, the access to real-time communication mechanisms, such as a real-time RMI[8], could be encapsulated into a connector and configured in different architectures.

The **Configurator**, a reflective element that provides services to instantiate, execute and manage applications with distributed configurations. The Configurator provides two APIs: configuration and architectural reflection, through which these services are used, and a persistency mechanism for the architecture meta-level description repository, where the two APIs reflect their operations. The configuration API allows to *instantiate*, *link*, *stop* and *replace* components of a running application. The architectural reflection API allows querying the repository. A specialized module of the application can consult the architecture's configuration and decide to make changes under certain conditions, say, in face of resource changes.

To specify non-functional aspects or quality of service (QoS) aspects related to operational requirements such as processing capacity, fault tolerance, real-time, information persistency, security or communication CBabel employs the concept of architectural contract. In our approach, an architectural contract is a description where two parts express their non-functional requirements, through services and parameters, negotiation rules and adaptation policies for different contexts. The CR-RIO framework provides the required infrastructure to impose and manage the contracts during run-time. Regarding QoS aspects we propose an architectural pattern that simplifies the design and coding of specific components of the infrastructure, consistently establishing the relationship between the Configurator and the QoS contract supporting entities.

3. THE QOS ARCHITECTURAL PATTERN

In our proposal a functional service of an application is considered a specialized activity, defined by a set of architectural components and theirs interconnection topologies; with requirements that generally do not admit negotiation[1]. Non-functional services are defined by restrictions to specific non-functional activities of an application, and can admit some negotiation including the used resources. A contract regulating non-function aspects can

describe, at design time, the use of shared resources the application will make and acceptable variations regarding the availability of these resources. The contract will be imposed at run-time by an infrastructure composed by a set of components that implement the semantics of the contract.

3.1 The QoS Contract Language

Our proposal incorporates concepts from the QML (QoS Markup Language)[4], which were reformulated for the context of software architecture descriptions[2]. A QoS contract includes the following elements:

QoS Categories are related to specific non-functional aspects and described separately from the components. For example, if processing and communication performance characteristics are critical to an application, associated categories, *Processing* and *Transport*, could be described as in Figure 1.

```
01  QoScategory Processing {
02    utilization: decreasing numeric %;
03    clockFrequency: increasing numeric MHz;
04    priority: increasing numeric; }
05  QoScategory Transport {
06    delay: decreasing numeric ms;
07    bandwidth: increasing numeric Mbps; }
```

Figure 1. Processing and Transport QoS Categories

The Processing category (lines 1-5) represent a processing resource where the utilization property is the used percentage of the total CPU time (low values are preferred – decreasing), the *clockFrequency* property represents the processor's operating frequency (high values are preferred – increasing) and priority represents a priority for its utilization. The Transport category (lines 5-7) represents the information associated to transport resources used by clients and servers. The *bandwidth* property represents the available bandwidth for the client-server connection and the *delay* property represents the transmission delay of one bit between a client and the server. The use of those categories, and of the other elements of the language described next, is presented in Section 4.

A **QoS profile** quantifies the properties of a QoS Category. This quantification restricts each property according to its description, working as an instance of acceptable values for a given QoS Category. A component, or a part of an architecture, can define QoS profiles in order to constrain its operational context.

A set of **services** can be defined in a contract. In a service, QoS constraints that have to be applied in the architectural level are described,

and can be associated to either (i) the application's components or (ii) the interaction mechanism used by these components. In that way, a service is differentiated from others by the desired/tolerated QoS levels required by the application, in a given operational context. A QoS constraint can be defined by associating a specific value of a property to an architecture declaration or associating a QoS profile to that declaration.

A **negotiation clause** describes a negotiation policy and acceptable operational contexts for the services described in a contract. As a default policy the clause establishes a preferred order for the utilization of the services. Initially the preferable service is used. According to the described in the clause, when a preferable service cannot be maintained anymore, the QoS supporting infrastructure tries to deploy a service less preferable, following the described order. The supporting infrastructure can deploy a more preferable service again if the necessary resources are again available.

3.2 Support Architecture

CBabel described architectures and QoS contracts are stored as meta-level information. Based on this information a set of middleware components (see Figure 4) composing a well-defined architectural pattern[2] is used to instantiate the application and to manage the contracts.

The **Global Contract Manager** (GCM) interprets a contract description and extracts its service negotiation state machine. When a negotiation is initiated the GCM identifies which service will be negotiated first and sends the configuration descriptions, related to each participating node, and the associated QoS profiles to the **Local Contract Managers** (LCM). Each LCM is responsible for interpreting the local configuration and activating a *Contractor* to perform actions such as resources reservation and monitoring requests. If the GCM receives a positive confirmation from all LCM involved, the service can be attended and the application can be instantiate with the required quality. If not, a new negotiation is attempted in order to deploy the next possible service. If all services in the negotiation clause are tried with no success, an *out-of-service* state is reached and a contract violation message is issued to the application level. The GCM can also initiate a new negotiation when it receives a notification informing that a preferred service became available again.

The **Contractor** has several responsibilities: (a) to translate the properties defined by the QoS profiles into services of the support system and convey the request of those services (with adequate parameters) to the QoS Agents; (b) when required, to map each defined interaction scheme (*link*) into a connector able to match the required QoS for the actual interaction, and (c) to receive *out-of-spec notifications* from the QoS Agents.

The information contained in a notification is compared against the profile and, depending on its internal programming the Contractor can try to make (local) adjustments to the resource that provides the service. For instance, the priority of a streamer could be raised in order to maintain a given frame generation rate. In a case where this is not possible an *out-of-profile* notification is sent to the LCM.

A **QoS Agent** wraps the access to system level mechanisms, providing adequate interfaces to perform resource requests, initializes local system services and monitors the actual values of the required properties. According to the thresholds to be monitored, registered by the Contractor, a QoS Agent can issue an *out-of-spec* notification indicating that a resource is not available or does not meet the specification defined in the profile.

4. EXAMPLE

During our research we developed some prototype examples to evaluate and refine the framework. A virtual terminal in a mobile machine was used to evaluate security and communication aspects in the context of a mobile network[9]. Specifically, a static contract was used to specify security protocol options (*telnet* or *ssh*, and cipher types) and a dynamic contract was used to specify communication channels that can be dynamically reconfigured (reconfiguration can be triggered by changes in available set of channels); in this example it was also demonstrated the composition of both contracts, which was immediately achieved joining theirs negotiation clauses. We developed in [5] the application with real-time requirements, mentioned in the introduction, an application with fault tolerance requirements, and the video on demand application to be presented in the next subsections.

4.1 Video on Demand (VoD)

The scenario of the application is comprised by a server, which stores a collection of video files in the MPEG-2 format, and by clients that connect themselves to the server and initialize a flow to receive and display a selected video. Each client can freeze or resume the video exhibition, in the same way it would be done if the video were locally stored. It is assumed that the clients can run on different platforms, from portable devices to workstations, in which the availability of resources such as CPU capacity and bandwidth can vary. In this context it is necessary to adapt the resources or the application's architecture configuration, depending on the specific operational environment, in order to have the video being exhibited with the expected quality.

The basic architecture of the example should fit two types of client: (i) high processing availability, with high-speed access to the server and (ii) medium processing availability, with dial-up modem access to the server. In principle, clients of type (i) have enough processing and communication resources to exhibit the video in the original MPEG-2 format. Clients of type (ii), with limited resources, can only exhibit the video in and alternative format, say H.261.

```
01  module Client_Server {
02     port provide, request;
03     module Client { out port request; } player;
04     module Server { in  port provide; } server;
05     instantiate server at serverHost;
06     instantiate player;
07     link player.request to server.provide;
08  } vod;
09  start vod;
```

Figure 2. VoD application Architecture Description

Figure 2 presents the CBabel description of the application's architecture, composed by a client (*player* - line 3) and a server (*server* – line 4), and their connection topology; communication is made effective through the player's *request* port and the server's *provide* port (lines 5-7). Note that this interconnection could be detailed, by defining a specific connector to mediate the client-server interaction, encapsulating the necessary communication mechanisms. However, as the non-functional restrictions include interaction aspects, the use of connectors in this architecture will be defined explicitly in a contract.

4.2 QoS Contract

The QoS contract of this example considers that two services can be used: (i) the exhibition of the video in the MPEG-2 format or (ii) in the H.261 format, according to the availability of resources at the specific client platform. To deploy any of these services in the client's node, the resources to be handled are those related to the host's processing characteristics and to the client-server communication channel properties.

The QoS categories for processing and transport, and their properties to specify the VoD application contract, are those presented previously in Figure 1. In the example it is considered that the client has to have a CPU with a minimum operating frequency of 700 MHz and a maximum of 50% of used CPU time to exhibit video in the MPEG-2 format. The exhibition of video with the H.261 format will demand from the CPU, by its turn, only a

minimum frequency of 266 MHz and a maximum CPU time usage of 70%. In the example we are not considering static reservation of CPU time, in order to illustrate a contract renegotiation activity. Please note that in a dynamic context, even with CPU reservation, a contract could be invalidated by another contract with higher priority.

In the example, the MPEG-2 requires a bandwidth greater than 1.5 Mbps and a transport delay lower than 50 ms to sustain an acceptable video stream, while videos in H.261 format require a minimum bandwidth of 56 Kbps and can tolerate delays up to 200 ms. Other transport properties could be taken into account in this case, such as the *jitter* or data loss rate; for the sake of simplicity they were not included in the *Transport* QoS Category.

```
01  contract {
02     service {
03      instantiate player at clientHost with cpu_01;
04      link player to server by comTransport with network_01;
05    } MPEG_video;
06    service {
07       instantiate player at clientHost with cpu_02;
08       link player to server by H-261.comTransport
09                                          with network_02;
10    } H-261_video;
11    negotiation {
12       MPEG_video -> H-261_video;
13       H-261_video -> out_of_service;
14    }
15  } vod;
16  profile {
17     Processing.clockFrequency >= 700;
18     Processing.utilization <= 50;
19  } cpu_01;
20  profile {
21     Processing.clockFrequency >= 266;
22     Processing.utilization <= 70;
23  } cpu_02;
24  profile {
25     Transport.delay <= 50;
26     Transport.bandwidth >= 1.5;
27  } network_01;
28  profile {
29     Transport.delay <= 200;
30     Transport.bandwidth >= 0.056;        // 56 kbps
31  } network_02;
```

Figure 3. VoD application QoS Contract

Based on the previous requirements the application's contract can be described as in Figure 3. The *MPEG_video* service (lines 2-5) defines the QoS constraints for the architecture parts that participate in the MPEG video

exhibition. The creation of a *player* component instance (line 3) in a client machine is associated to the *cpu_01* processing QoS profile. The interconnection of the *player* and *server* ports are bound to the *network_01* QoS profile (lines 25-27), being the communication provided by a connector that encapsulates the communication transport mechanism (line 4). The mentioned profiles specify, respectively, the constraints to the *Processing* and *Transport* QoS Categories properties, relevant to this contract. Thus, to create the player instance, the *clockFrequency* of the node has to be at least 266 MHz and then the CPU *utilization* has to be less than 70%. The *H-261_video* service description follows a similar procedure. The *cpu_02* (lines 20-23) and *network_02* (lines 28-31) profiles represent the requirements for the H.261 video exhibition. Note that, for this service, the interaction of the components is mediated by a connector that encapsulates the MPEG-2 to H-261 conversion mechanism. Additionally to the MPEG-2 and H.261, other formats could be supported by using specific decoders, encapsulated in connectors; e.g., the *bitmap* format that can be exhibited on PDAs and cell-phone video matrixes.

The negotiation clause of this contract (lines 11-14) defines the priority order between the services. The *MPEG_video* service has to be preferably provided in relation to the *H-261_video* service. If there are no resources available to attend any of these services, an *out-of-service* state is reached and the application cannot run.

4.3 Mapping the contract into the architectural pattern

The implementation of the QoS contract of the example-application using the proposed architectural pattern is depicted in Figure 4. Each participant node has a running instance of the *Local Contract Manager*, the specific *Contractor* for the VOD application and *QoS Agents* associated to the resources to be controlled in each specific platform. The *Configurator* (Section 2) and the *Global Contract Manager* can be instantiated in a node dedicated to manage applications or in the same node were the application's server is running. The *H-261* connector only takes part of the configuration when the *H-261_video* service is deployed. It can also be observed that the *comTransport* connector has a distributed implementation.

The sequence diagram presented in Figure 5 depicts the interactions between the CR-RIO components to establish the MPEG_video service to a player running in a node, which is connected to the server through an Ethernet network. When starting the procedure to load the application the Configurator and the GCM are already running. As the first step, the GCM retrieves the associated QoS contract; all further actions are guided by this contract. Initially the GCM creates instances (create()) of the LCM in the

nodes where the application components are to be instantiated. Next, it selects a service to be used (in this case, the MPEG_video) and initializes a negotiation activity, sending to the LCMs the information related to this service, including the associated QoS profiles (cpu_01 and network_01). Each LCM extracts from the received information the QoS characteristics that have to be considered and instantiates (*create()*) (a) the QoS Agents that provide the interfaces (management and event generation) to the resources used by the service, and (b) the application specific Contractor, that will interpret the service information and will interact with the QoS Agents to impose the desired properties.

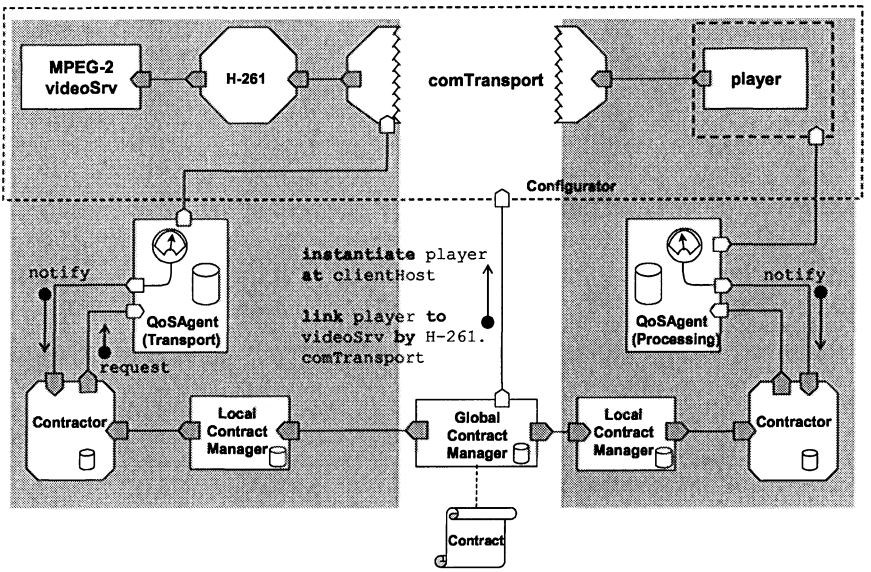

Figure 4. Mapping the VoD application contract in the architectural pattern

In the client node, the LCM identifies the processing resources that have to be managed (based on the `instantiate` ADL's primitive that creates an instance of the *player* module – QoS contract, line 3). In the server node, the local LCM identifies (based on the `link` ADL's primitive that interconnects the *player* module to the *server* module – QoS contract, line 4) that it will be responsible for the management of the transport resources (the adopted semantics is to assign to the server side the responsibility for managing QoS requirements that involves two peers). When the LCM instantiates a Contractor it also sends to it the profiles that have to be attended. In the sequence, the Contractor interacts with the QoS Agents to request resources and to receive relevant events regarding the status of the resources. In this example, the *Processing* QoS Agent verifies the operating frequency of the

CPU and is responsible for monitoring the CPU load (*utilization*). Also, observe that the client-server communication channel uses some kind of resource reservation put in effect through the *Transport* QoS Agent.

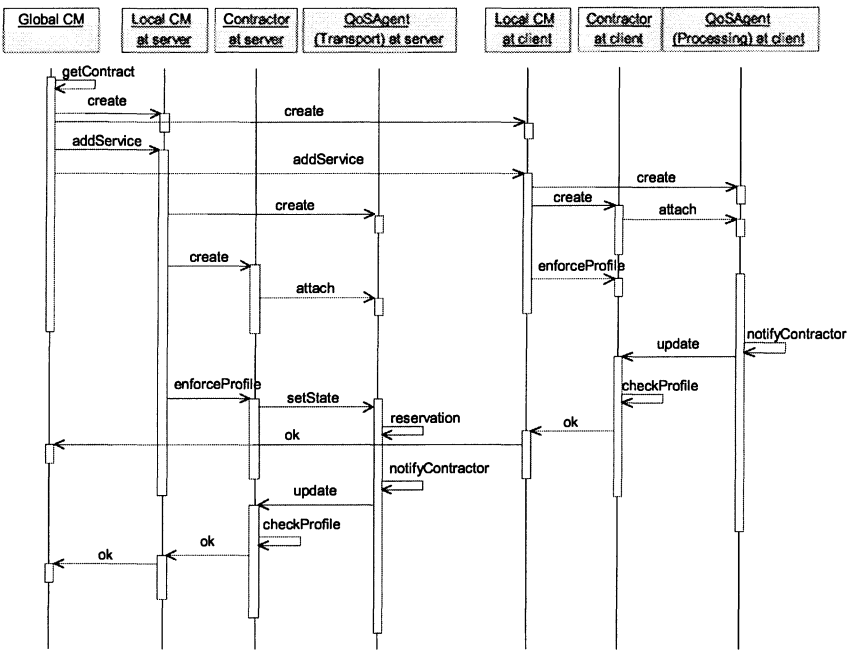

Figure 5. Establishing the MPEG_video service

After the initial phase, if the required QoS profiles were imposed, a Contractor notifies the success to its associated LCM that, by its turn, forwards a corresponding notification to the GCM. In the example, if all involved LCMs did return a positive confirmation, the GCM concludes that the negotiation was successful and that the *MPEG_video* service can be established. The next step is to instantiate the application's functional components in the context of the reserved resources and, then, to initialize its execution. This step is performed by the *Configurator* (Section 2) based on the *Architecture Configurator* design pattern[10]; see details in[5]. If during the negotiation any Contractor has a resource demand denied, or if it verifies that a QoS Agent notified an out of range value, an *out-of-profile* notification is sent to the LCM that, by its turn, sends an *out-of-service* notification to the GCM. In consequence the GCM selects the next service to be attempted, in this case the *H-261_video*, and a new negotiation cycle begins.

In steady state, if a significant change in the monitored values is detected, the QoS Agent notifies the registered Contractors invoking the *update()* method. If the reported values do not violate the active QoS profiles, nothing

has to be done. If there is a violation, the Contractor can try to locally readapt the resource in order to keep the service; for instance, passing new parameters to the QoS Agent. If it is not possible to readapt, the Contractor sends an *out-of-profile* notification to the LCM and, in the sequence, another service can be negotiated. To exemplify the situation let's suppose that while the *MPEG_video* is operational, new processes are admitted to the client's node, diminishing the available processing power to the player. This would be captured by the *Processing* QoS Agent observing the increase of the value of the *utilization* property. Let's consider that the measured value overcomes the limit of 50% defined by the *cpu_01* profile, but is still lower than the 70% limit defined by the *cpu_02* profile.

The *Processing* QoS Agent notifies the Contractor triggering a new negotiation. The client's Contractor verifies that the property is out of the *cpu_01* profile specification and sends the respective LCM an *out-of-profile* notification. This information is then propagated to the GCM through an *out-of-service* notification. Then the GCM selects the H-261_video to be negotiated and sends the respective information as parameters invoking to the involved LCMs. Each LCM discontinues the current service and the procedures to impose the new service, bound by the *cpu_02* and *network_02* QoS profiles are performed (similarly as in the case to deploy the initial service). Several optimizations are feasible. For instance, when a Contractor sends an *out-of-profile* notification this could be followed by the set of QoS profiles that could be attended at that moment. Receiving this composed information the GCM could select the next service to be negotiated, immediately discarding the services with associated profiles out of the set. We are investigating the use of an event support service, with composition capability, to implement this optimization. A second optimization could be applied when a set of services is restricted to a given node. In this case the LCM of the given node could receive the information of all services and profiles related to the set and manage them locally.

A prototype of the VoD application is presented in[5]. The Java *Media Framework* was used to implement the functional modules. Some classes related to the video flow were encapsulated in connectors, e.g., those implementing RTP and the H.261 codec. The QoS architectural pattern was implemented as a set of classes integrated to the CR-RIO framework.

It was possible to identify that the implementations of the GCM and the LCM, directly related the application contract are reusable. The behavior of these elements is parameterized by the QoS contract of the specific application; in this level the manipulated information are symbolic. Each QoS Agent has dependencies related to the resource being managed. However, once implemented, an Agent can be reused in other applications that have operational requirements dependent on the same kind of resource.

The Contractor, by its turn, represents the *hot spot* of the pattern. Its implementation is dependent on the services and profiles to be imposed, and also dependent on the own resources to be managed via QoS Agents. The Contractor can also contain the code implementing specific policies to perform local adaptations, as discussed in the end of the last section.

5. RELATED WORKS

The reflective middleware approach[11] allows for the provided services to be configured to comply with the non-functional properties of the applications. However, the approach does not provide clear abstractions and mechanisms to help the use of such features in the design of the architectural level of an application. This leads to the middleware services being used in an *ad hoc* fashion, usually through pieces of code intertwined to the application's program. The Quality Connector pattern provides a methodology for the re-allocation of resources in response to context changes in the execution environment[12]. However, it requires access to the source code of every application and/or to the infrastructure's components in order to instrument them. Our approach, that includes configuration-programming mechanisms, is more transparent regarding the access to the source code of the application. The Quality Objects (QuO)[3] provides a framework for the development of distributed applications with QoS requirements, based on CORBA. In QuO, the specification of such requirements is associated to method invocations, through a contract description language, allowing only adaptations at this level. Our proposal considers services with differentiated quality in diverse levels, from the interface (or connection) level, in which services are encapsulated into connectors (similar to the QuO approach), to the architectural level, in which the service provision can involve the reconfiguration of the application's topology. The proposal described in[13] includes basic mechanisms to collect status information associated to non-functional services. It also suggests an approach to manage non-functional requirements in the architectural level, in a way quite similar to ours. CR-RIO complements this proposal providing an explicit methodology based on contracts and proposing extra mechanisms to deploy and manage these contracts. More details are available in[5].

6. CONCLUSION

We presented a unified approach to specify, deploy and manage applications having non-functional requirements. The approach helps to

achieve separation of concerns and software reuse by allowing non-functional aspects of an application, such as QoS requirements, to be specified separately using high-level contracts expressed in an extended ADL. Being centered on an ADL-based configuration middleware the framework inherits all its well-known benefits, among them the capability of reconfiguration, which facilitates to execute dynamic architectural adaptations on behalf of a contract. Part of the coding, related to a non-functional requirement, can be encapsulated in connectors, which can be (re)configured during running time in order to cater for the impositions defined by the associated contract. The infrastructure required to enforce the contracts follows an architectural pattern that is implemented by a standard set of components of the middleware. In this pattern, each component performs a well-defined role in the support of the contract. We think that making these structures explicit and available to designers, the task of mapping architecture-level defined contracts to implementations can be simplified. The approach has been evaluated through several case studies that showed that the code of these supporting components can be automatically generated, excepting some localized pieces related to specificities of the particular QoS requirement under consideration. However, we should notice that the treatment of low-level details always has to be considered in any QoS aware application. Our approach can help to identify the intervening hot spots and to make adaptations more rapidly.

In our proposal, the composition of contracts can be specified combining in a unique clause the negotiation clauses of the involved contracts[9]. Contracts regarding different non-functional aspects (in the same or in different applications) can be orthogonal and cause no interference with each other; in this case, composing those contracts is immediate. In the general case, the composition process can lead to conflicts on the use of shared scarce resources. Conflicts can be handled applying a suitable decision policy to the set of involved contracts; already assigned resources could then be retaken in order to satisfy the preferred contracts.

Currently, we are investigating the specification of individual contracts for clients and servers[14]. This intends to allow each client to specify what it requires and each server to specify what it is committed to provide. This capability would permit to make decisions regarded to a component instantiation taking into account the availability of resources at its instantiation time. Besides providing the flexibility required to the support of dynamic architectures, this would allow managing conflicts through lower granularity interventions. We are also working towards giving a formal semantics to the QoS contracts, using Rewriting Logic, in the same line as presented in[7] for the CBabel ADL. With the results of that experience we

plan to produce a set of guide-lines to allow the formal verification of the QoS contracts in the architectural level.

ACKNOWLEDGMENTS

Orlando Loques is partially supported by CNPq (grant PDPG-TI 552137/2002) and Alexandre Sztajnberg is partially supported by CNPq (grant PDPG-TI 552192/2002 and Faperj APQ1 E26/171430-02).

REFERENCES

1. Beugnard, A., Jézéquel, J.-M., Plouzeau, N., Watkins, D., "Making Components Contract Aware", IEEE Computer, 32(7), July, 1999.
2. Cerqueira, R. C., "A Methodology to Describe and Implement Contracts for Services with Differentiated Quality in Distributed Architectures ", Masters Dissertation, IC/UFF, 2002.
3. Loyall, J. P., Rubel, P., Atighetchi, M., Schantz, R., Zinky, J. "Emerging Patterns in Adaptive, Distributed Real-Time, Embedded Middleware", 9th Conference on Pattern Language of Programs, Monticello, Il., September, 2002.
4. Frolund, S. and Koistinen, J., "Quality-of-Service Specifications in Distributed Object Systems", Distributed Systems Engineering, IEE, No. 5, pp. 179-202, UK, 1998.
5. Ansaloni, S., "An Architectural Pattern to Describe and Implement Qos Contracts", Masters Dissertation, Instituto de Computação, UFF, May, 2003.
6. Loques, O., Sztajnberg, A., Leite, J., Lobosco, M., "On the Integration of Configuration and Meta-Level Programming Approaches", in Reflection and Software Engineering V. 1826, LNCS, pp. 191-210, Springer-Verlag, Heidelberg, Germany, June, 2000.
7. Braga, C. and Sztajnberg, A., "Towards a Rewriting Semantics to a Software Architecture Description Language", 6th Workshop on Formal Methods, Brasil, October, 2003.
8. Borg, A. and Wellings, A., "A Real-Time RMI Framework for the RTSJ", Proceedings of the15th Euromicro Conference on Real-Time Systems, Porto, Portugal, July, 2003.
9. Cerqueira , R. C., Ansaloni, S., Loques, O.G. and Sztajnberg, A., "Deploying Non-Functional Aspects by Contract", 2nd Workshop on Reflective and Adaptive Middleware, Middleware2003 Companion, pp.90-94, Rio de Janeiro, Brasil, June, 2003.
10. Carvalho, S. T, Lisbôa, J. and Loques, O, "A Design Pattern for Software Architecture Configuration", 2nd Latin American Conference on Pattern Languages of Programming, RJ, Brasil, August, 2002.
11. Kon, F. et alii, "The Case for Adaptive Middleware", Communications of the ACM, pp. 33-38, Vol. 45, No. 6, June, 2002.
12. Cross J.K. and Schmidt, D., "Quality Connector: A Pattern Language for Provisioning and Managing Quality-Constrained Services in Distributed Real-Time and Embedded Systems", 9th Conf. on Pattern Language of Programs, Monticello, Illinois, Sep., 2002.
13. Garlan, D., Schmerl, B. R. and Chang, J., "Using Gauges for Architecture-Based Monitoring and Adaptation", Work. Conference on Complex and Dynamic Systems Architecture, December, 2001.
14. Sztajnberg, A. and Loques, O., "Bringing QoS to the Architectural Level", ECOOP 2000 Workshop on QoS on Distributed Object Systems, Cannes.