Customizing Component-Based Architectures by Contract

Orlando Loques[1] and Alexandre Sztajnberg[2]

[1]Instituto de Computação – Universidade Federal Fluminense (UFF)
Rua Passo da Pátria – Niterói – RJ – Brasil
loques@ic.uff.br

[2]Instituto de Matemática e Estatística – Universidade do Estado do Rio de Janeiro
Rua São Francisco Xavier, 524 / 6018-D – Maracanã – RJ - Brasil
alexszt@ime.uerj.br

Abstract. This paper presents an approach to describe, deploy and manage component-based applications having dynamic functional and non-functional requirements. The approach is centered on architectural descriptions and associated high-level contracts. Besides specifying non-functional (or QoS) requirements, these contracts are used to guide architecture customizations required to enforce the requirements. 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 approach allows designers to write a contract and to follow a standard recipe to insert the extra code required to its enforcement in the supporting middleware.

1 Introduction

The current software development technology offers a rich diversity of options to specify the interfaces and write the functional code of program components. Once built and made available, these components can be used to compose different applications, having specific non-functional requirements, that should be deployed in diverse operating environments. However, the specification of non-functional requirements and the implementation of the corresponding management strategies are, generally, embedded in the code of the components in an ad-hoc manner, mixed with the application's specific code. This lack of modularity makes component reuse difficult, also making difficult verification and debugging tasks. In this context, there is a growing interest for handling non-functional aspects in a specific abstraction level [2, 5, 11]. This approach would allow to single out the resources to be used and the specific mechanisms that will be required to support the non-functional aspects, and, if possible, turn automatic the configuration and management of those resources.

Besides requirements normally associated to communication system level performance, non-functional (sometimes called QoS) requirements (or aspects) include characteristics 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 approach makes feasible to design a software system based on its architectural description, which includes the functional components, the interactions among those components and also the non-functional requirements, which depend on the properties of the supporting infrastructure. 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 is to be available an environment that allows to deploy those requirements over the system resources even during running time.

Among the available techniques to specify non-functional constraints, we highlight the concept of contract [7]. A contract establishes a formal relationship between two or more 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 contract defining rules to replicate processing resources, in order to guarantee a maximum execution time constraint.

In the previous context, this work presents the CR-RIO framework (*Contractual Reflective - Reconfigurable Interconnectable Objects*) [5, 1] conceived to specify and support non-functional contracts, associated to the architectural description of an application. The approach helps to achieve separation of concerns [10] facilitating the reuse of components that implement the functional 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 running 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, unless of some explicit parts of code related to specific contract and resources classes.

In the rest of this paper, we initially describe the key elements of the framework including the architecture description language with support to contracts. Next, we present the supporting infrastructure and demonstrate the validity of the framework through an example. 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 [10], 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 (see Figure 1):

a) CBabel, an ADL used to describe the functional components of the application and the interconnection topology of those components. CBabel also caters for the description of non-functional aspects, such as coordination, distribution and different

types of QoS requirements. 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; these descriptions can be submitted to formal verification procedures [3].

b) An architecture-oriented component model, that allows programming the software configuration of the application; (i) Modules (or components), 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. This component model can be mapped to available implementation technologies; in our experiments components were mapped to Java and Corba objects.

c) A simple software design methodology that stimulates 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.

d) 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 persistent architecture description repository, where the two APIs reflect their operations. A specialized module can consult the architecture's description repository and decide to make adaptations, for instance, in face of changes in the QoS support level.

To specify non-functional aspects 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 running time. Regarding QoS aspects we propose an architectural pattern that simplifies the design and coding of the components of the supporting infrastructure, consistently establishing the relationship between the Configurator and the QoS contract supporting entities.

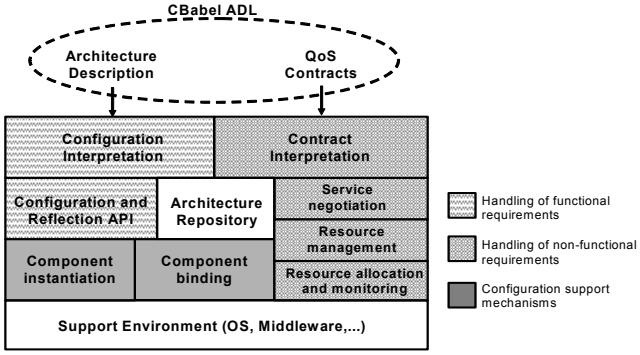

Fig. 1. The CR-RIO framework

3 The QoS Contract Language

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 [2]. 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. Our proposal incorporates concepts from the QML (QoS Markup Language) [7], which were reformulated for the context of software architecture descriptions [5]. A QoS contract includes the following elements:

a) QoS Categories are related to specific non-functional aspects and described separately from the functional components. For example, if processing and communication performance characteristics are critical to an application, associated QoS categories, *Processing* and *Transport*, could be described as in Figure 2.

```
01  QoScategory Processing {
02    cpuUse: decreasing numeric %;
03    cpuSlice: increasing numeric %;
04    priority: increasing numeric;
05    memAvaliable: increasing numeric Mbytes;
06    memReserv: increasing numeric Mbytes;
07  }
08  QoScategory Transport {
09    delay: decreasing numeric ms;
10    bandwidth: increasing numeric Mbps;
11    slidingWindowSize: increasing numeric;
12    MSS: increasing numeric;
13  }
```

Fig. 2. *Processing* and *Transport* QoS Categories

The *Processing* category (lines 1-7) represents processor and memory resources where the *cpuUse* property is the used percentage of the total CPU time (low values are preferred – *decreasing*), the *cpuSlice* property represents the time slice to be reserved / available to a given process (high values are preferred – *increasing*), *priority* represents a priority for its utilization, *memAvaliable* and *memReserv* represent, respectively the available memory in the node and the memory (to be) reserved for a process. The *Transport* category (lines 8-13) represents the information associated to transport resources used by clients and servers. The *bandwidth* property represents the available/required bandwidth for network connections and the *delay* property represents the transmission delay of one bit between two peer components. The use of those categories, and of the other elements of the language to be described next, is presented in Section 4.

b) 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.

c) 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.

d) 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.

3.1 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 9), composing a well-defined architectural pattern [5] 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 related configuration descriptions, 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 resource reservation and method requests monitoring.

If the GCM receives a positive confirmation from all LCM involved, the service being negotiated can be attended and the application can be instantiated 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.

QoS Agents encapsulate the access to system level mechanisms, providing adequate interfaces to perform resource requests, initialize local system services and monitor 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 [6]. In [1] it was presented a video on demand application, an application with fault tolerance requirements, and the application with timing requirements which will be detailed in the next subsections.

4.1 Data Acquisition-Processing Application

Let us consider a data acquisition system, which periodically receives data and image coming in batches from one or more sensors. The received image and data have to be processed and filtered before being stored in a data base. This basic architecture can be used in different application contexts and run on different support environments. For example, a simple application, with a single data source, can run on a single processor, provided that enough processing power is available to execute the required pre-processing activities within the required time interval for data acquisition. A complex application, where data comes from many geographically-distributed sensors, as well as where more complex and time consuming processing and filtering activities are performed, will require more processing power in order to meet the timing restrictions. Yet, a more complex application could have its processing requirements changing considerably along its running time; e.g., because an increase in the amount of input data triggered by the occurrence of an external event.

In such changing scenario, it is desirable to provide concepts and mechanisms to allow the basic architecture to be gracefully adapted in order to cater for the requirements of each different application context. For example, for the simple application a CPU reservation scheme would be enough to guarantee the processing power required for the application. For the complex application, assuming that it is parallelizable, a solution would be to distribute the execution, for example using a master-worker architecture. Such parallel architecture could be deployed on a grid of processors provided that some operational requirements are met in order to not hinder the application's performance; e. g., the allocated nodes should have enough resources and their message transport time to the master should be lower than a given limit.

Moreover, considering that the processing requirements can increase or decrease along the application running time, the number of parallel workers can be dynamically configured. Thus, when the processing demand increases, the number of parallel workers could be increased in order to reduce each one individual computation time, aiming to achieve an overall speed-up. Accordingly, the number of workers can be reduced in order to free system resources when the processing demands decreases.

We highlight that components of our architectural contract support framework can encapsulate the access to different available resource management services, in order to obtain the information required to enforce the architectural adaptations. In a related work we used the contract approach to express and implement contracts related to multimedia distributed applications based on services provided by the OpenH323 framework [12]. For the architectural contracts presented in this paper we consider parameters such as CPU reservation / monitoring, CPU availability, network bandwidth, and resource discovery that can be provided by available platforms such as the WNS framework [14].

In the example presented in this section we demonstrate how our approach using software architecture and contract concepts can be used to: (a) describe the application's components and respective topology configuration; (b) describe the policies on resource usage required to comply with the processing constraints imposed by the application and (c) effectively deploy the application with the support of middleware components included in the framework.

4.2 Basic Configuration

The basic configuration of the application is depicted in Figure 3. A client module collects the data from the sensors and sends them to the server for pre-processing. As soon as the pre-processing procedure is finished the server signals the client, which then can send a new data sample to be processed. The interaction between the client and the server (or servers) modules is explicitly mediated by a connector that will help to implement the application contract.

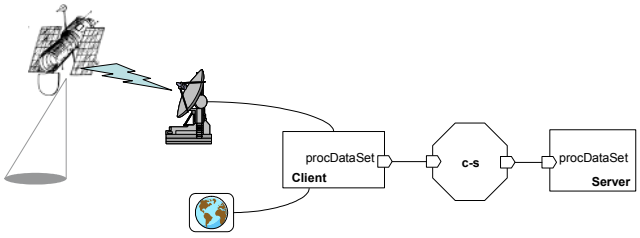

Fig. 3. Data Acquisition-Processing Application

Figure 4 presents the CBabel description of the application's architecture, composed by a client (*client* - line 3), a server (*server* – line 4), and their connection topology; interaction is performed through the client's *procDataSet* out port and the server's *procDataSet* in port (line 6). Note that this interconnection could be statically defined using a specific connector to mediate the client-server interaction,

encapsulating the required communication or interaction mechanism. However, as the non-functional requirements include communication, processing and replication aspects, the use of connectors in the architecture will be defined separately in a contract or automatically selected by the contract support middleware.

```
01  module Client_Server {
02    port procDataSet;
03    module Client {out port procDataSet;} client;
04    module Server {in  port procDataSet;} server;
05    instantiate client, server;
06    link client.procDataSet to server.procDataSet;
07  } capture_images;
08  start capture_images;
```

Fig. 4. CBabel description of the application's architecture

In an initial context we assume that the *client* and *server* components are deployed in the same node. In this case, to attend the application's requirements, processing and storage resources have just to be reserved for the *server* module. The QoS contract regarding such requirement is described in Figure 5. The *prioProc* service (lines 14-16) states that the instantiation of the *server* module at the *host1* node is associated to the *ProcMem* processing QoS profile (lines 19-22). In that case, the *server* module instantiation is conditioned to the availability of enough storage capability (at least 200 Mbytes) and of a processing slice of at least 0.25 (25%) of the processor's time.

The Contractor is responsible for translating the requirements regarding the storage and processing resources described in the contract (in this case, `Processing.cpuSlice >= 0.25; Processing.memReserv >= 200;`), into parameters that can be passed to the *Processing* QoS Agent.

```
13  contract {
14    service {
15      instantiate server at host1 with profile PROCMEM;
16    } prioProc;
17    negotiation {prioProc -> out-of-service;};
18  } oneServer;
19  profile {
20    Processing.cpuSlice >= 0.25;
21    Processing.memReserv >= 200;
22  } ProcMem;
```

Fig. 5. *prioProc* contract description

In this first context, the requirements are static and if the *Global Contract Manager* receives a service violation notification, an out-of-service state is reached and no other service is attempted according the associated QoS contract (line 17). Thus, the application cannot execute given the lack of resources.

4.3 Second Configuration - Distributed Parallel Workers

In a second context the servers are replicated though a master-worker architecture in order to distribute the processing load, based on a slightly modified *Master-Slave* design pattern [4]. To this end a *Replication* QoS category (Figure 6) is introduced. When this category is used, a special connector is selected to provide the services related to group communication and maintenance, according to the value of the *groupComm* property (line 20). The *numberOfReplicas* and *maxReplicas* properties (lines 17-18) describe respectively the number of replicas to be deployed and the maximum number of replicas allowed. This last property can be used with *replicaMaint* (line 19) in the case of a contract that will handle dynamically creation of replicas. The *distribPolicy* property (line 21) indicates a policy to be adopted for the distribution of replicas (in this case, driven by the best memory, CPU or transport operating status, or an optimization of these parameters).

```
16  QoScategory Replication {
17    numberOfReplicas: increasing numeric;
18    maxReplicas: numeric;
19    replicaMaint: enum (add, remove, maintain);
20    groupComm: enum (p2p, multicast, broadcast);
21    distribPolicy: enum (bestMem, bestCpu, bestTransp, optim);
22  }
```

Fig. 6. *Replication* QoS category

Again, the preprocessing performed in each server should be concluded before a new data-set is produced by the client. Here, the communication system transport time becomes a relevant performance parameter. As the data-set has to be sampled at a given rate, the deadline within which the server task has to be performed is known beforehand. So, in a distributed environment, where the communication with the server adds to the total preprocessing execution time, the overall deadline should include this parameter. Thus, in order to express this fact, we consider in the contract a message transport time parameter (line 29, fig.7); the latter aggregated with the previous processor reservation parameter will provide a trustful means to impose the application timing requirement at run time. The corresponding contract is represented in Figure 7.

```
13  contract {
14    service {
15      instantiate server with profile ProcMem, Preplic;
16      link client to server with profile Pcom;
17    } repProc;
18    negotiation {repProc -> out-of-service;};
19  } repServer;
20  profile {
21    Processing.cpuSlice >= 0.25;
22    Processing.memReserv >= 200;
23  } ProcMem;
24  profile {
25    Replication.numOfReplicas = 5;
```

```
26    Replication.distribPolicy = optim;
27  } Preplic;
28  profile {
29    Transport.delay < 5;
30    Replication.groupComm = multicast;
31  } Pcom;
```

Fig. 7. QoS contract for the replication configuration

According to the *repProc* contract each replica will only be instantiated if the *ProcMem* and *Preplic* profiles properties are satisfied. The number of replicas and the distribution policy described in the *Preplic* profile (lines 24-27) are controlled by the GCM. A number of five replicas were selected (line 25) and the distribution policy will try to optimize resources (line 26). Additionally, it can be observed that replicating the *server* module in different processing nodes implies in creating instances of this module. This task is also initiated by the GCM as soon as it establishes the service, delegating the actual configuration of the instances to the *Configurator*. In this case, the GCM forwards a list of nodes were the replicated modules have to be created and the *Configurator* executes an instantiation batch such as:

```
instantiate Server as repl1 at node1;
link client.procDataSet to repl1.procDataSet by groupCon;
instantiate Server as repl2 at node2;
link client.procDataSet to repl2.procDataSet by groupCon;
…
```

The execution of this batch connects the *client* module to each replica of the *server* (*repl1*, *repl2*, …) by a connector composition (*groupCon*) that provides the group communication mechanisms (multicast, in this case – line 30). The *Configurator* dynamically manages the naming of the replicas and makes this information consistent for the GCM. For all the established client-replica interconnection this connector is used to provide the client-server interaction style and the group communication.

This configuration is robust but still static. If any of the processing or transport properties of any replica is out of specification the respective LCM is notified by the QoS Agent, which forwards this notification to the GCM. As no other service is provided in the contract, the application is terminated.

4.4 Third Configuration - Dynamic Processing Requirements

Finally, in a third context, it is assumed that the processing requirements change dynamically, either increasing or decreasing. Thus, we add to the contract specification three new profiles (maintReplica, addReplica, removeReplica) which indirectly capture this behavior, allowing to optimize the number of processors processing the application, and also cater for the processing time deadline. These profiles include upper and lower bounds to the execution time, which are used to control the number of worker replicas. The final contract is presented in Figure 8.

```
13 contract {
14    service {
15      instantiate server with profile maintReplica, ProcMem;
16      link client to server with profile Pcom;
17    } Smaint; // basically the same service as repProc
18    service {
19      instantiate server with profile addReplica,ProcMem,Pmax;
20      link client to server with profile Pcom;
21    } Sadd;
22    service {
23      remove server with profile removeReplica;
24    } Sremove;
25
26    negotiation {
27      Smaint -> ((Sremove -> Sremove) || (Sadd -> Sadd));
28      Sremove -> Smaint;
29      Sadd -> Smaint;
30      Sadd -> out-of-service;
31      Smaint -> out-of-service;
32    };
33  } dynRepServer;
34
35  profile {
36      Replication.maxReplica = 10;
37  } Pmax;
38  profile {
39    Replication.Maint = maint;
40    Processing.execution_time >= 500 ms <= 600 ms;
41  } maintainReplica;
42  profile {
43    Replication.Maint = add;
44    Processing.execution_time > 600 ms;
45  } addReplica;
46  profile {
47    Replication.Maint = remove;
48    Processing.execution_time < 500 ms;
49  } removeReplica;
```

Fig. 8. QoS contract for the dynamic replication configuration

In the *dynRepServer* contract three services are described. The *Smaint* service (lines 14-17) is the preferred one, where the execution time meets the application requirements and no replicas need to be created (profile *maintReplica* – lines 38-41). If the execution time (`execution_time` property was added to the *Processing* category) is greater than the upper bound, the *Smaint* service is discontinued and the *Sadd* service (lines 18-21) is tried. In this case, the *addReplica* profile is imposed and one or more replicas are created (line 43), but the number of replicas is limited by the *Pmax* profile `Replication.maxReplica = 10` property. If this limit is reached no more replicas can be created and the service cannot be provided. On the other way, if the execution time gets bellow the lower bound, the *Sremove* service (lines 22-24) is

deployed in order to release resources, removing one or more replicas. The calculation of the actual number of replicas to be added or removed can be performed by the GCM using some heuristic based on the information regarded to resource availability collected from the LCMs.

According to the negotiation clause, where the switching modes for the services are described, when the *Sadd* or *Sremove* services are effective they are renegotiated while the measured execution time is out of the required range (i.e, < 500 or > 600). When this value fits again in the preferred range, the establishment of the *Smaint* service is again negotiated. Similarly, if any property of the involved profiles is invalidated during operation, a new negotiation can be initiated. In the worst case, when the *Sadd* (or Smaint) service is selected, and no configuration of replicas can fulfill the contract profiles, an *out-of-service* state is reached and the application is terminated. In the next section we discuss how the described configurations could be deployed using our framework.

4.5 Implementation Details

Each participant node (Figure 9) has instances of the *LCM*, of the specific *Contractor* for the application and *of the QoS Agents* associated to the resources to be controlled in each specific platform. The *groupCon* connector only takes part of the configuration when the replication services are deployed. As the first step, the GCM retrieves the application's contract (for the explanation, we consider the third configuration only) and creates instances of the LCM in the nodes where the application components are to be instantiated. Next, it selects the preferred service (*Smaint*) to be used and initializes a negotiation activity, sending to the LCMs the information related to this service, including the associated QoS profiles (*ProcMem, Pcom* and *maintReplica*). Each LCM instantiates (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.

In the server node, the LCM identifies the processing resources that have to be managed (`instantiate` that creates an instance of the *server* – QoS contract, line 15). Also, based on the `link` primitive that interconnects the *client* module to the *server* module (QoS contract, line 16), the LCM in the client's node identifies the need of a group communication connector and makes the necessary arrangements to manage the transport resources. 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 associated to a server node is responsible for reserving and monitoring the CPU time slice (*cpuSlice*) and memory (*memReserv*) for the server *module*. Also, observe that in addition to monitor the communication *delay* the client-server communication channel could optionally use some kind of resource reservation (e.g., the RSVP protocol) put in effect through the *Transport* QoS Agent. 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. If all involved LCMs did return a

positive confirmation, the GCM concludes that the negotiation was successful and that the *Smaint* service can be established.

In steady state, if a significant change in the monitored values is detected, the QoS Agents notifies the registered Contractors. 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.

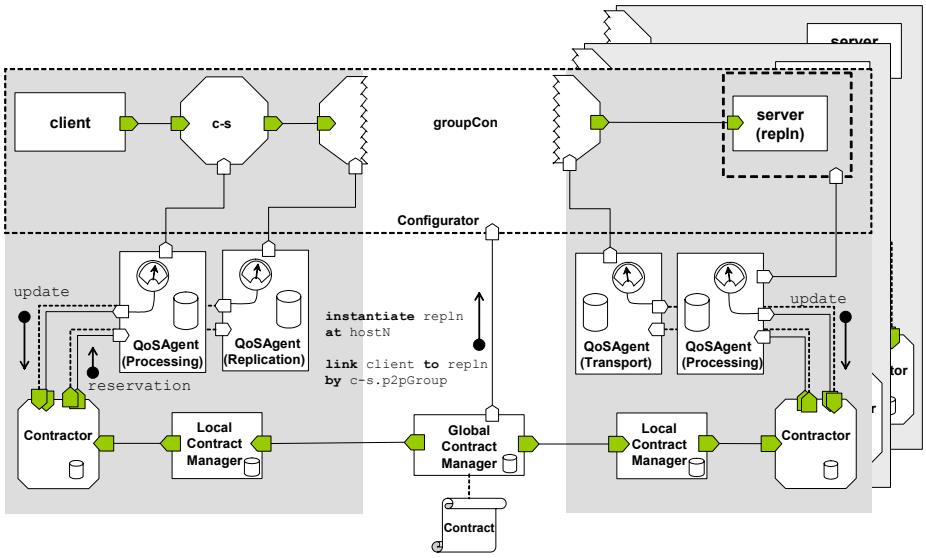

Fig. 9. Mapping the application contract in the architectural pattern

To exemplify an operation, let's suppose that while the *Smaint* service is operational the *Processing* QoS Agent in the client node observes that the measured *execution_time* value rises beyond the upper bound defined by the *maintReplica* profile (> 600). The *Processing* QoS Agent notifies the Contractor triggering a new negotiation. The server's Contractor verifies that a property is out of the *ProcMem* profile specification and sends the respective LCM an *out-of-profile* notification. This information is then propagated to the GCM, along with an *out-of-service* notification. Then the GCM selects the *Sadd* service and starts the actions required to create a new replica.

The described infrastructure can be adapted to different support environments, currently we are working in a prototype using the WNS framework [14]. Many optimizations are also 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. Another point of interest is having resource re-adaptation locally managed by a Contractor, using the interface provided by the QoS Agents.

This would be suitable for resources that have embedded re-adaptation policies and mechanisms. For example, considering the *Processing.cpuSlice* property, the Contractor could try to raise the priority of the local server *process* to maintain this property within the profile specification. We are investigating how to specify this kind of concern at the contract level.

5 Related Works and Conclusions

The reflective middleware approach [9] allows for the provided services to be configured to meet the non-functional properties of the applications. However, the approach does not provide clear abstractions and mechanisms to help the use of such requirements in the design of the architectural level of an application. This leads to the middleware services to be used in an ad-hoc fashion, usually through pieces of code intertwined to the application's program. The proposal described in [8] 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 detailed comparisons are available in [1].

Our approach helps to achieve separation of concerns and component reuse by allowing non-functional aspects of an application to be specified separately using high-level contracts expressed in an extended ADL. Part of the codification, 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 a contract follows an architectural pattern that is implemented by a standard set of components. 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 case studies that showed that the code of the 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 make the required 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 [6]. Contracts regarding different non-functional aspects can be orthogonal and cause no interference with each other. Contract conflicts can be handled applying a suitable decision policy; already assigned resources could then be retaken in order to satisfy the preferred contracts. We are also investigating the specification of individual contracts for clients and servers [13]. Besides providing the flexibility to support more dynamic architectures, this would allow to manage contract composition conflicts through lower granularity interventions.

Acknowledgments. Sidney Ansaloni and Romulo Curty, M.Sc. students, provided the examples and valuable insights that helped us to present this text. Orlando Loques and Alexandre Sztajnberg are partially supported by CNPq (grants PDPG-TI 552137/2002 e 552192/2002, respectively). Alexandre Sztajnberg is also partially supported by Faperj (grant E-26/171.430/2002).

References

1. Ansaloni, S., “An Architectural Pattern to Describe and Implement Qos Contracts”, Masters Dissertation, Instituto de Computação, UFF, May, 2003.
2. Beugnard, A., Jézéquel, J.-M., Plouzeau, N., Watkins, D., “Making Components Contract Aware”, IEEE Computer, 32(7), July, 1999.
3. Braga, C. e Sztajnberg, A., “Towards a Rewriting Semantics to a Software Architecture Description Language”, 6th Workshop on Formal Methods, Campina Grande, Brasil, October, 2003.
4. Buschman, F., et alli, “Pattern-Oriented Software Architecture – a System of Patterns (POSA1)”, John Willey and sons, Chichester, ISBN 0-471-95869-7, UK, 1996.
5. Curty, R., "A Methodology to Describe and Implement Contracts for Services with Differentiated Quality in Distributed Architectures ", Masters Dissertation, Instituto de Computação, UFF, 2002.
6. Curty, R., Ansaloni, S., Loques, O.G. e 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.
7. Frolund, S. e Koistinen, J., "Quality-of-Service Specifications in Distributed Object Systems", Distributed Systems Engineering, IEE, No. 5, pp. 179-202, UK, 1998.
8. Garlan, D., Schmerl, B. R. and Chang, J., “Using Gauges for Architecture-Based Monitoring and Adaptation”, Working Conference on Complex and Dynamic Systems Architecture, December, 2001.
9. Kon, F. et alli, “The Case for Adaptive Middleware”, Communications of the ACM, pp. 33-38, Vol. 45, No. 6, June, 2002.
10. 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.
11. 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.
12. “Open H323”, Quicknet Technologies, http://www.OpenH323.org, 2004.
13. Sztajnberg, A. and Loques, O., “Bringing QoS to the Architectural Level”, ECOOP 2000 Workshop on QoS on Distributed Object Systems, Cannes France, June, 2000.
14. Wolski, R., Spring, Neil T. and Hayes, J., “The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing”, Future Generation Computer Systems", vol. 15, No. 5-6, pp. 757-768, 1999.