A contract-based approach to describe and deploy non-functional adaptations in software architectures

Orlando Loques[1], Alexandre Sztajnberg[2], Romulo Curty[1] & Sidney Ansaloni[1]

[1]Instituto de Computação
Universidade Federal Fluminense (UFF)
Niterói - RJ - Brasil, CEP 24210-240

[2]Instituto de Matemática e Estatística
Universidade do Estado do Rio de Janeiro (UERJ)
Rio de Janeiro - RJ - Brasil CEP 20559-900

{loques, curty, ansaloni}@ic.uff.br, alexszt@uerj.br

Abstract

This paper presents a comprehensive approach to describe, deploy and adapt component-based applications having dynamic non-functional requirements. The approach is centered on high-level contracts associated to architectural descriptions, which allow the non-functional requirements to be handled separately during the system design process. This helps to achieve separation of concerns facilitating the reuse of modules that implement the application in other systems. Besides specifying non-functional requirements, contracts are used at runtime to guide configuration adaptations required to enforce these 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 allows designers to write a contract and to follow standard recipes to insert the extra code required to its enforcement in the supporting middleware.

Keywords: contracts, non-functional requirements, software architectures, dynamic configuration adaptation, separation of concerns, middleware

*Romulo Curty is supported by Petrobras. Orlando Loques is partially supported by CNPq (grant PDPG-TI 552137/2002). Alexandre Sztajnberg is partially supported by CNPq (grant PDPG-TI 552192/2002 and Faperj APQ1 E26/171430-02). The authors would like to thank the anonymous referees and the Editors for their critics and suggestions that have contributed to improve the paper presentation.

1 Introduction

The traditional notion of quality of service (QoS) was bound to communication system level concerns. However, a more recent view of QoS includes characteristics associated to application's non-functional requirements, such as availability, reliability, security, real-time, persistency, coordination and debugging support [27]. The specification, setup and adaptation concerns associated to non-functional requirements are generally embedded in the application programming modules in an ad-hoc manner, mixed with the application's specific code. This lack of modularity hinders software evolution and code reuse, also making difficult its verification and debugging. In this context, there is a growing interest for handling non-functional requirements in a specific abstraction level [4, 10, 20]. 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.

Non-functional requirements can be handled by reusable services provided by middleware infrastructures or native systems support. This makes it 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 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 guarantee that the non-functional requirements are met at runtime. Among the available techniques to specify non-functional constraints, we highlight the concept of contract as proposed in [13]. This kind of contract establishes a formal relationship among the parts that use or provide resources, where constraints and negotiation rules over the used resources are expressed.

In the previously described context, this work presents the CR-RIO framework (*Contractual Reflective -Reconfigurable Interconnectable Objects*) [10, 1] which includes concepts and mechanisms conceived to specify and support QoS contracts, associated to the architectural components of an application. The set of concepts included in the framework helps to achieve separation of concerns [18] 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 at runtime 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 for 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 two initial sections of this paper we describe the framework composed by the key elements of the approach, including an architecture description language with support to QoS contracts. Next, aiming to demonstrate the flexibility of the approach, we present three use-cases, and in the sequel we describe the components of the contract's support middleware. Concluding the article, we comment on some related proposals and provide some conclusions.

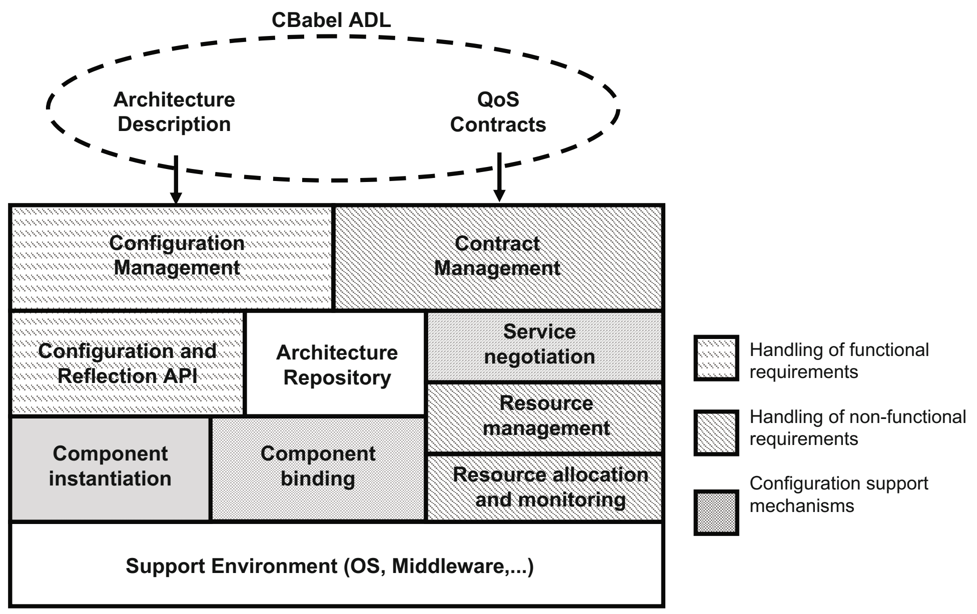
2 Framework Elements

The CR-RIO framework (Figure 1) integrates the software architecture paradigm centered on an architecture description language (ADL), with concepts such as reflection and dynamic adaptation capability [18], which are generally provided in an isolated fashion in other related proposals described in the literature [e.g., 23, 14, 28, 20, 16]. This integration facilitates the achievement of separation of concerns, software component reuse and dynamic adaptation capability of applications. CR-RIO includes the following elements:

a) 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 some non-functional aspects, such as coordination and distribution, and planned reconfigurations. A CBabel specification corresponds to a meta-description of an application that is available from a repository, and is used to deploy the architecture in a given operating environment. While the application is running this meta-description repository provides the basic information required to guide and manage architectural adaptations.

b) An architecture-oriented component model used to compose and implement the software configuration of the application: (i) Modules, which encapsulate the application's functional requirements; (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. With the help of ADL constructs, modules and connectors can be aggregated in order to define composite components, which can be reused in different application systems. This component-based model can be mapped to available implementation technologies; in our different experiments components were mapped to Unix processes, CORBA and Java objects.

The description of an application's architecture includes the signatures of the interfaces (set of ports) provided and required by the composed components. This allows CR-RIO's connectors to be context-reflective in the sense that, when plugged into modules, they can be automatically and dynamically (at configuration time) adapted to mediate interactions between ports using any specific signature. This kind of adaptation facilitates the reuse of connectors in different application systems. An example demonstrating the reuse of a connector encapsulating an implementation of the observer design pattern is available in [18].



CBabel ADL
Architecture Description
QoS Contracts
Configuration Management
Contract Management
Configuration and Reflection API
Architecture Repository
Service negotiation
Resource management
Component instantiation
Component binding
Resource allocation and monitoring
Support Environment (OS, Middleware,...)
Handling of functional requirements
Handling of non-functional requirements
Configuration support mechanisms

Figure 1: The CR-RIO framework

c) A simple software design methodology that stimulates the designer to follow a straightforward meta-level programming discipline [18], where functional requirements are concentrated in modules (base level entities) and non-functional requirements are encapsulated in connectors (meta-level entities). In this context, ports have a role similar to reification points used in meta-level programming approaches and, similarly, can be associated with different code boundary transfer mechanisms, e.g., method, procedure or messaging passing primitive invocations. It is worth to point out that some common non-functional requirements can be directly mapped into connectors, which are equivalent to meta-level components, and can be immediately configured in an application's architecture. For example, the access to real-time communication mechanisms, such as a real-time RMI [5], could be encapsulated into a connector and configured in different architectures.

d) The Configurator, a reflective element that provides services to 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 permits to instantiate, bind, stop and replace components of a running application dynamically changing its configuration. These operations are atomically performed and are causally connected, i. e., the applied changes are serially reflected in the architecture description repository. The architectural reflection API allows querying the configuration repository. This information can be used to guide configuration changes under certain conditions, for instance, in face of changes in the QoS support level.

e) Architectural contracts, a concept proposed to specify non-functional aspects, which provide a description where components of the architecture can express their static and dynamic non-functional requirements. This is achieved by defining the set of services (and their associated parameters) that can be used by an application, service negotiation rules and architectural adaptation policies for different operational contexts. In addition, the CR-RIO framework provides the required infrastructure to impose and manage the contracts during running time. More specifically, 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 contract supporting entities (please see Section 5).

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 [4]. Non-functional or QoS services are defined by restrictions associated to specific non-functional requirements of an application, and can admit some negotiation including the used resources. A contract regulating non-function requirements can describe, at design time, the use of the resources that the application will make, and acceptable variations regarding the availability of these resources, at runtime. Our proposal incorporates concepts from QML (QoS Markup Language) [13], which were reformulated for the context of software architecture descriptions [10]. A QoS contract includes the following elements:

a) QoS Categories are used by the designer to aggregate properties related to specific non-functional requirements; they are named and described separately from the 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    utilization: decreasing numeric %;
03    clockFrequency: increasing numeric MHz;
04    priority: increasing numeric;
05    memReq: increasing numeric Mbytes;
06  }
07
08  QoScategory Transport {
09    delay: decreasing numeric ms;
10    bandwidth: increasing numeric Mbps;
11    technology: enum {CDLS, GSM, WiFi};
12    slidingWindowSize: increasing numeric;
13    send-buf-size: increasing numeric;
14    recv-buf-size: increasing numeric;
15    MSS: increasing numeric;
16  }
17 ...
```

Figure 2: Processing and Transport QoS Categories

The *Processing* category (lines 1-6) can represent a processing resource where the *utilization* property express the required percentage of the CPU time (low values are preferred - *decreasing*), the *clockFrequency* property represents the processor's operating frequency (high values are preferred - *increasing*), *priority* represents a priority for processor utilization, and *memReq* represents the amount of memory required to run a process. The *Transport* category (lines 8-16) illustrates properties commonly associated to transport resources used by clients and servers components to communicate. For example, 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 a server. The specific use of these categories, and of the other elements of the language to be described next, will be illustrated through the examples presented in Section 4.

b) A QoS profile quantifies the specific set of properties of a QoS Category that are relevant in a given application. This quantification restricts each property according to its description, working as an instance of acceptable values for a given QoS Category. A component, a connection, or a part of an architecture, can define particular QoS profiles in order to constrain its operational context. For example, `transport.delay <= 10,` express the maximum communication delay required in a given context. The design decisions for defining a specific profile are based on the requirements of the application to be implemented by the architecture being described, and can be constrained by the available support to manage the used resources.

c) A set of services can be defined in a contract. A service defines a set of non-functional requirements that should be deployed in the architectural level; these non-functional requirements can be associated to either (i) the application's components or (ii) the interaction mechanism used by these components. In QoS terms, a service is differentiated from others by the desired or tolerated QoS characteristics required by the application, in a given operational context. A non-functional QoS constraint can be defined by associating a specific value of a property to an architecture declaration or associating a specific 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 sequence of services described in the clause, when a preferable service cannot be maintained anymore, the QoS supporting infrastructure tries to deploy a less preferable service. The supporting infrastructure can deploy a more preferable service again if the necessary resources are again available. When required, a designer can override the default policy and provide the specific code required to deploy any other desirable policy. Currently, we are investigating how to consider syntactically and semantically the available options.

4 Use-Cases

During our research we developed some prototype examples to evaluate and refine the approach. Here we present three of these experiments and raise discussions of some issues in their context. Additional implementation details are discussed in Section 5.1.

4.1 Communication Technology Adaptation

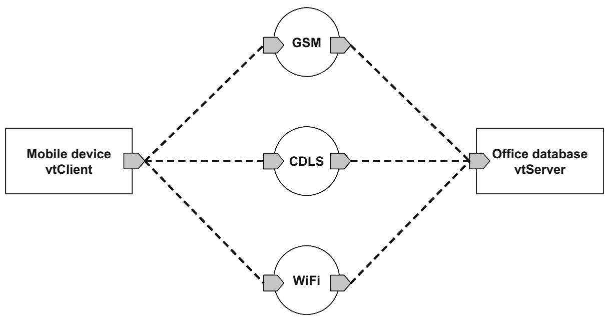
In [11] a simulated virtual terminal application was used to evaluate security and communication aspects in the context of a mobile network. Specifically, a first contract was used to specify an initial choice between secure and unsecure protocol options (*telnet* or *ssh*, and cipher types) and a second contract was used to specify communication channels that can be dynamically reconfigured during running time. The latter contract considers a mobile device that supports three communication channels each one using a different technology, as follows:

CDLS: in the range of a cordless base station, this channel operates over a regular wired telephone line; GSM: on the move, this channel operates over a cellular network; WiFi: when immerse in a wireless network, this channel uses the available communication protocols.

Figure 3.a presents the architecture description; note that the client-server connector does not appear explicitly in this configuration. Also, it is assumed that the server is already instantiated. Figure 3.b presents the application's overall architecture; the dashed lines represent the links that can be dynamically established, guided by the contract negotiation policy and depending on resource availability. The second contract for this use-case (Figure 4) defines three different transport services, each one associated to a specific communication channel. The support required for each service (defined in lines 2-5, 6-9 and 10-13, respectively) is encapsulated in a specific connector associated to the link construct (showed in lines 3-4, 7-8 and 11-12, respectively), as described by the associated profiles, which were defined directly using the specific `Transport.technology` property. For each available channel option there is an associated communication interface and a specific sensor that detects if the corresponding channel is available or not. During operation the supporting middleware tries to keep a communication link using the best among the available channels; to establish the link it just configures the corresponding connector. Other non-functional aspects could be considered in this architecture. For example, seamless communication could be achieved encapsulating the access to a reliable communication protocol (e.g., [30]) into a connector and configuring it in the application architecture.

```
module VirtualTerminal {
  module {
    in port (char) Recv;
  } vtServer;
  module {
     out port (char) Send;
  } vtClient;
  instantiate vtClient
                 at clientNode;
  link vtClient to vtServer;
} vt;
start vt;
```

(a)



(b)

Figure 3: Mobile application basic architecture

The negotiation clause (lines 14-18) defines that the best service is the *wireless* transport service; if this service is not available, a *fixTel* service should be tried and so on. Transitions between services depend on their availability. Either when the currently used link fails, or a preferable service becomes available, a service adaptation can take place. Other adaptation policies could be used, e.g., based on a reduced-cost or bandwidth criterion.

The composition of contracts was also investigated in this use-case. Contracts regarding different non-

```
contract {
  service {
     link vtClient.Send to vtServer.Recv
     with Transport.technology: WiFi;
  } wireless;
  service {
     link vtClient.Send to vtServer.Recv
     with Transport.technology: CDLS;
  } fixTel;
 service {
     link vtClient.Send to vtServer.Recv
     with Transport.technology: GSM;
  } celTel;
  negotiation {
      wireless -> fixTel;
      fixTel   -> celTel;
      celTel   -> out_of_service;
  };
} vt;
```

Figure 4: Mobile device QoS contract

functional requirements (in the same or in different applications) are orthogonal when it is not necessary to combine states of their respective sets of services in the same negotiation chain, because they do not interfere with each other. Composing orthogonal contracts is immediate: their services, profiles and negotiation clauses would make independent parts of the composed contract; their negotiation clauses are literally combined. The unified negotiation clause combining the two contracts used in this use-case is presented in Figure 5.

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.

In this example, additional virtual terminal client instances can be instantiated during running time without any apparent conflict caused by the server's sharing. However, in order to deal with more general configurations we are investigating the description of individual contracts for clients and servers [25]. 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 for the support of dynamic architectures, this would allow managing conflicts through lower granularity interventions.

```
negotiation {
   secure   -> unsecure;
   unsecure -> out_of_service;

   wireless -> fixTel;
   fixTel   -> celTel;
   celTel   -> out_of_service;
}
```

Figure 5: Combining contracts

4.1.1 Client-Server Adaptation

Here we consider a server, which periodically receives service requests coming from one or more clients [19]. Each request has to be serviced and the corresponding reply should be issued within a given deadline. This basic architecture can be used in different application contexts and run on different support environments. For example, a simple configuration, with a single client, can run on a single processor, provided that enough processing power is available to the server to execute the required processing activities within the required time interval. A more demanding application, where requests come from many clients, 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. In this scenario, it is desirable to provide concepts and mechanisms to allow the basic architecture to be gracefully adapted in order to meet the requirements of each different application context. For example, in the simple client-server application a CPU reservation scheme would be enough to guarantee the processing power required for the server. For the demanding application, assuming that it is parallelizable, a solution would be to distribute the execution, for example using a replicated server 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.

4.1.2 Single Server Configuration

Figure 6 presents the CBabel description of the application's architecture, composed by a client (*client* - lines 3-5), a server (*server* - lines 6-8), and their interconnection topology. Interaction is performed through the client's *procDataSet* out port and the server's *procDataSet* in port (line 10). 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 can include communication, processing and replication aspects, the use of connectors in the architecture will be defined separately in a contract; the specific connector can be automatically selected by the contract support middleware.

```
01  module Client_Server {
02     port procDataSet;
03     module Client {
04        out port procDataSet;
05     } client;
06     module Server {
07        in  port procDataSet;
08     } server;
09     instantiate client, server;
10     link client.procDataSet to
                         server.procDataSet;
11  } capture_images;
12  start capture_images under <contract_name>
```

Figure 6: CBabel description of the application's architecture

In this first context, we assume that the client and server components are deployed in the same node and that the client execution requirements are easily met. In this case, to meet the application's requirements, processing and storage resources have just to be reserved for the server module, as described in the *singleServer* contract presented in Figure 7. The *prioProc* service (lines 14-16) states that the instantiation of the server module at the *host1* node is associated to the *PProcMem* processing profile (lines 19-22). According to this contract, 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, as defined by *PProcMem*.

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

Figure 7: Single server contract description

4.1.3 Distributed Server Configuration

Now, the servers are replicated in order to distribute the processing load. To this end a *Replication* QoS category (Figure 8) is introduced. When this category is used, a special connector is selected to provide the services related to the group communication protocol, selected by the value of the *groupComm* property (line 22). The *numberOfReplicas* and *maxReplicas* properties (lines 19-20) 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 21) in the case of a contract that will dynamically handle the creation of replicas. The *distribPolicy* property (line 23) indicates a policy to be adopted for the distribution of replicas (in this example, driven by the best memory, CPU and transport delay values).

```
18  QoScategory Replication {
19    numberOfReplicas: increasing numeric;
20    maxReplicas: numeric;
21    replicaMaint: enum (add,
                        remove, maintain);
22    groupComm: enum (p2p, multicast,
                  broadcast, loadbalance);
23    distribPolicy: enum (bestMem, bestCpu,
                       bestTransp, optim);
24  }
```

Figure 8: Replication category

According to the *repServer* contract (presented in Figure 9) each replica will only be instantiated if the *PProcMem* and *Preplic* profiles properties are satisfied.

A number of five replicas was selected (line 25) and the distribution policy will try to optimize resources (line 26). For all the established client-replica interconnections a connector (*groupCon*) is used to provide a selected group communication mechanism. In this example, we consider that the computation of the service requested can be split among the replicated servers and a multicast protocol is selected (line 30). In a case where the service requests can be processed independently, a policy to distribute the load among the servers would be more suitable (*loadbalance*, line 22 in the *Replication* category); a heuristic aiming to optimize load balancing in this context is presented in [2]. Note that this use-case architecture can evolve to accommodate additional clients. For this, each new client has just to be started under the *repServer* contract in order to be automatically linked to the servers with the *PCom* profile.

```
13  contract {
14    service {
15      instantiate server
              with profile PProcMem, 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.memReq >= 200;
23  } PProcMem;
24  profile {
25    Replication.numOfReplicas = 5;
26    Replication.distribPolicy = optim;
27  } Preplic;
28  profile {
29    Transport.delay < 5;
30    Replication.groupComm = multicast;
31  } Pcom;
```

Figure 9: Contract for the distributed-replicated server configuration

Again, as the requests have to be processed at a given rate, the overall deadline within which the server task has to be performed should not be violated. In a distributed environment, where the message transport time to the server adds to the total preprocessing execution time, the overall deadline should include this parameter. In order to express this requirement, the contract includes a message transport time parameter (Figure 9, line 29); the latter aggregated with the previous processor reservation parameter will provide a trustful means to impose the application timing requirement at runtime.

In an application where the processing requirements can increase or decrease along the application running time, the number of replicated servers could be dynamically controlled, according to a strategy prescribed in a contract. For example, when the processing demand increases, the number of servers could be increased in order to reduce each one individual computation time, aiming to achieve an overall speed-up. Accordingly, the number of servers can be reduced in order to free system resources when the processing demands decreases (see [19] for more details).

We have also experimented with server replication techniques to achieve fault-tolerance [17]. In this context, diverse reliable message multicast protocols, based on a reliable group communication support, were implemented to ensure server's state consistency. These protocols are encapsulated into connectors, which can be configured to select a specific failure semantics more suitable to meet the requirements of a given application. These connectors also encapsulate the access to the group membership service, which was used to implement module failure detection and recovery mechanisms. In a recent work [1], we investigated the use of CR-RIO contracts to express this kind of non-functional requirement.

We note that components of our contract support middleware can encapsulate the access to different available resources and mechanisms, in order to obtain the information required to enforce the non-functional requirements. In our fault-tolerance experiment, the reliable multicast protocol and the group membership service provided a clear example. For the architectural contracts presented in this section, parameters such as CPU utilization, available storage capacity, network bandwidth, and resource discovery are required to assist the allocation of server replicas. This kind of support can be provided by available platforms such as the NWS framework [29].

4.2 Video on Demand (VoD) Application

The scenario of this 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. It is assumed that the clients can run on different platforms, from portable devices to workstations, in which the availability of resources required for video exhibition, such as available CPU capacity and communication bandwidth, can vary. In this context it is necessary to adapt 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; of course, they also can deploy the less demanding H.261 format. Clients of type (ii), with limited resources, can only exhibit the video in the H.261 format. Note that a type (i) client can degrade to type (ii), depending on the availability of the supporting resources. Figure 10 presents the CBabel description of the application's architecture, composed by a client (*player* - line 3) and a server (*videoSrv* - line 4), and their connection topology; communication is effected through the player's *request* port and the video-server's *provide* port (lines 6-8). Following the previous examples, as the non-functional restrictions include interaction aspects, the use of connectors in this architecture will be defined explicitly in a contract.

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

Figure 10: VoD application Architecture Description

The QoS categories for processing and transport used to specify the VoD application contract are those presented previously in Figure 2. In this example it is considered that the client has to run on a CPU with a minimum operating frequency of 700 MHz, and can require up to 50% of the available CPU time to exhibit videos in the MPEG-2 format. By its turn, the exhibition of videos in the H.261 format will demand a CPU with a minimum clock of 266 MHz and can require a maximum CPU usage of 30%.

For video transmission, the MPEG-2 format requires a bandwidth greater than 1.5 Mbps and a transport delay lower than 50 ms to sustain an acceptable video stream, while the H.261 format requires a minimum bandwidth of 56 Kbps and can tolerate transport 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.

The QoS contract of this example considers that two services can be used: the exhibition of the video (i) in the MPEG-2 format or (ii) in the H.261 format, according to the availability of resources at the specific client platform. In the client's node, to deploy any of these services, the resources to be handled are those related to the host's processing characteristics and to the client-server communication channel properties. Based on the previous requirements, the application's contract can be described as in Figure 11.

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 *videoSrv* ports are bound to the *network_01* QoS profile (lines 28-31), being the communication provided by a connector that encapsulates the required mechanisms (line 4). The mentioned profiles specify, respectively, the constraints to the *Processing* and *Transport* QoS Categories properties relevant to this contract. In this case, to create the player instance the *clockFrequency* of the node has to be at least 700 MHz and then the CPU utilization has to be less or equal than 50%.

The *H-261_video* service description follows a similar procedure. The *cpu_02* (lines 23-26) and *network_02* (lines 33-36) 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 codecs, 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 12-15) defines the priority order between the services. The *MPEG_video* service has to be preferably provided in

```
contract {
  service {
    instantiate player at clientHost
                          with cpu_01;
    link player to videoSrv
                      by comTransport
                      with network_01;
  } MPEG_video;

  service {
    instantiate player at clientHost
                          with cpu_02;
    link player to videoSrv
                  by H-261.comTransport
                  with network_02;
  } H-261_video;

  negotiation {
    MPEG_video -> H-261_video;
    H-261_video -> out_of_service;
  }
} vod;

profile {
 Processing.clockFrequency >= 700;
 Processing.utilization <= 50;
} cpu_01;

profile {
 Processing.clockFrequency >= 266;
 Processing.utilization <= 70;
} cpu_02;

profile {
 Transport.delay <= 50;
 Transport.bandwidth >= 1.5;
} network_01;

profile {
 Transport.delay <= 200;
 Transport.bandwidth >= 0.056;
} network_02;
```

Figure 11: VoD application QoS Contract

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. Initially a negotiation is performed to establish the basic configuration. However, additional negotiations are plausible. For example, if the MPEG service is operational and either the *network_01* or the *cpu_01* profiles cannot be sustained; in either case the video service could be continued in a degraded mode using the H.261 encoding. Note that in a dynamic context, even if a CPU reservation mechanism is used to ensure the *cpu_01* profile, a contract service could be invalidated by the instantiation of another contract with higher priority.

5 Supporting Middleware

CR-RIO supporting middleware follows an architectural pattern composed by a set of components, namely: one Global Contract Manager (GCM), and Local Contract Managers (LCMs), Contractors and QoS Agents. The conceptual basis for this pattern is described in [10]; a more pragmatic view is presented in [1]. Here we present a brief description of CR-RIO middleware components. This middleware uses CBabel described architectures and QoS contracts, which are available as meta-level information, to instantiate an application and to manage its associated contract.

The GCM represents the main authority; it can fully interpret and manage contract descriptions and knows their service negotiation state machine. LCMs are distributed and were introduced mainly for practical reasons; they have a partial view of the active contracts. When a negotiation is initiated, the GCM identifies which service will be negotiated first and sends the related meta-level descriptions to each participating node LCM; this includes the associated QoS profiles. Each participating LCM is responsible for interpreting the local interests of a contract and for activating its associated Contractor and QoS Agents.

A service can be attended if all enclosed profiles are met; the LCMs involved in a negotiation gather and convey this information to the GCM. If a positive confirmation is received from all LCMs involved in a negotiation, the related service can be attended and the application can be instantiated with the required quality. If not, a new negotiation can be attempted in order to deploy the next possible service. If all services in a contract 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 concludes that a preferred service became available again. When the negotiation finishes the GCM uses the Configurator to deploy the components required for the selected service.

For each particular contract, a specific Contractor instance is created. Basically, it takes care of concerns associated to local profiles related to the contract's services. Contractors are built from a generic class, which is specialized for the profiles of a given service. More specifically: (i) When one or more property profiles are related to a local primitive service of the support system, it performs a request on behalf of this service (with the required parameters) to the corresponding QoS Agents; from then on, it can receive *out-of-spec* notifications from the QoS Agents, containing relevant values related to these profiles. (ii) When a property profile requirement can be implemented by a specific connector, it just interact with this connector that also acts like a QoS agent; for example, this happens in the case presented in Section 4.1. The content of an *out-of-spec* spec notification is compared against the associated profile and, in some cases, the Contractor can try to make (local) adjustments to the resource that provides the primitive service. For instance, in the VoD application, the priority of a streamer, encapsulated in the connector, could be raised in order to maintain a given frame generation rate; note that this requires an explicit intervention of the programmer. In a case where local adjustments are not possible, an *out-of-profile* notification is sent to the associated LCM.

QoS Agents are also specialized. They can just provide access to architectural level resources information or encapsulate the access to system level mechanisms, providing adequate interfaces to perform requests to local primitive services, initialize system resources and monitor the actual values of the required properties. According to the values to be monitored (that can include threshold limits), which were registered by a Contractor, a QoS Agent can issue an *out-of-spec* notification, which includes relevant resource-related information (that may be forwarded all the way to the GCM), allowing the respective Contractor to verify if a resource is no more available or if it does not meet the specification defined in the profile.

5.1 Implementation Details

The implementation of the contract of the application presented in Section 4.3 using the described architectural pattern is depicted in Figure 12. Each participant node has an instance of the *Local Contract Manager*, the specific *Contractor* for the VOD application and the *QoS Agent* associated to the resources to be controlled in each specific platform. 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; in this experiment the codification related to the video flow transport mechanisms were encapsulated in this *comTransport* connector, e.g, RTP and RTCP protocols. In a more network-aware context some kind of resource reservation mechanism could be also employed, e.g., the RSVP protocol.

In the contract negotiation phase, the middleware tries to establish the required profiles and the associated support required for them. If the GCM concludes that the negotiation was successful the service can be established. The next step is to start the application's functional components, in the context of the established support; this is performed using the services provided by the Configurator (described in Section 2).

All the presented case studies were implemented in Java. The *Java Media Framework* was used to implement the functional modules of the VoD application. Java objects were mapped to configurable components through the Configurator, which is implemented following the *Architecture Configurator* design pattern described in [7]. In particular, the structure of CR-RIO's connectors follows a standard pattern which implementation could be supported by a tool. Basically, at the invoker side, after being identified, each method invocation is serialized and, in the sequel, at the target object side a dynamic invocation of the concerned method is performed. We have not put much effort in optimizing our connector's implementation. However, we made a set of measurements [26] that show a small overhead when compared to the cost associated with the basic machinery required to perform dynamic method invocations in Java, which is considerably greater than the normal method invocation cost. The connector's performance can be optimized either using precompiled stubs (loosing the dynamic context reflective capability - see section 2-b) or using a specialized JVM, e.g, [22] (though loosing portability).

5.2 Additional remarks

Our presentation assumes a unique Contract Manager authority - the GCM, which is required in any system running diverse applications using shared resources, as is the case of large mission-critical embedded systems [9]. This centralized entity can introduce scaling and reliability issues, which have to be solved using appropriated replication techniques. In a more open environment, it would be feasible to use a fed-

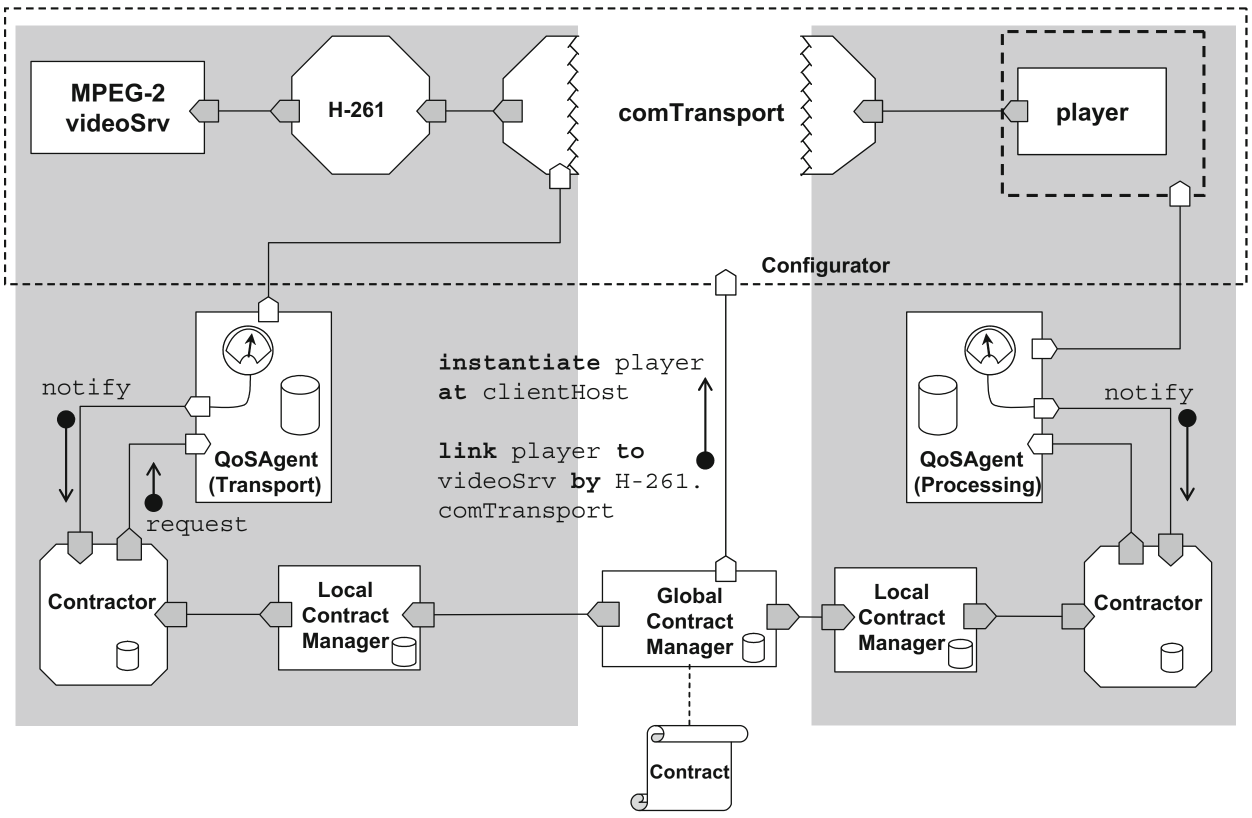


Figure 12: Mapping the VoD application contract in the architectural pattern

eration of GCMs, which would have to negotiate to implement the contract management functionality.

In our current implementation we did not consider the issue of failures affecting the configurable entities required to impose a given service for a contract. This could be tackled extending the GCM to treat as an atomic transaction the set of configuration actions required to impose a QoS service. Another related issue regards the application behavioral (and state) consistency during QoS service transitions. On this regard we do not think there is an application-independent solution; the specific application and the relative time span required to effect the configuration changes are among the factors to be considered. In some cases, consistency-related issues could be treated at the application realm, and the programmer should be aware of the intended use of the functional components within a given domain. In other cases, some system-level solution could be provided, e.g., to mask disruptions using a special communication protocol as mentioned in section 4.1. Available studies make very restricted assumptions to provide transparent solutions to the application consistency issue, e.g., [15]. Additional research is required to clarify the practical aspects of both failure and consistency issues.

Some other 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 can filter out the services with unattended profiles, and immediately select the next service to be activated. Another optimization could be applied when a set of services uses resources restricted to a given node. In this case the node's LCM can receive the information of all services and profiles related to the set and manage them locally. In some cases, the GCM can incorporate the functions of the other components, as the LCM, or services provided by existing middleware systems to optimize its implementation, as discussed in Section 4.2. In specific cases, it is even possible to collapse all the support to a single node, managing all the contract concerns locally; this would be suitable in the use-case presented in Section 4.1.

Some resources have specific and embedded re-adaptation policies and mechanisms, which are part of their intrinsic semantics, as happens with some specialized communication protocols. For these cases, resource re-adaptation can be locally managed by a Contractor, using the interface provided by the QoS Agents, as mentioned in the beginning of this section. As an interim solution, we are investigating how to treat this kind of concern in the contract level. However, in our research we found a clear need for explicit separation of resource status and availability monitoring functions from resource allocation and management functions. Due to historical reasons these two functions are embedded in most QoS aware protocol suites and can not be separately used. This makes difficult to a designer to take advantage of each of them to write a contract. We hope that this situation changes in the next generation systems.

The presented use-cases showed us that the components of the CR-RIO middleware follow a recurrent architectural pattern in those different applications. In particular, in our experiments we observed that the interactions between the middleware components follow standard sequence diagrams, which facilitate implementations based on object-oriented principles. In addition, the behavior of these elements is parameterized by the contract of the specific application; at this level the manipulated information is symbolic. It appears that the GCM and the LCM overall codification can be generalized, being identified as frozen spots of the pattern. On the other hand, the QoS Agent and the Contractor were identified as hot spots and have to have specific implementations, although part of their code can be automatically generated from the related contracts. Each QoS Agent has dependencies related to the specific resource being managed. However, we note that a QoS Agent needs to be programmed once for a given resource; from then on, it can be reused in other applications that have operational requirements dependent on the same kind of resource. By its turn, the Contractor's implementation is dependent on the services and profiles to be imposed, which depend on the resources to be managed via QoS Agents. The Contractor can also contain the code implementing specific policies to perform local adaptations, as already discussed in this section. Finally, we note that the implementation of the interaction between a Contractor and a QoSAgent is based on the Observer design pattern, which can simplify the codification of these components.

6 Related Work

The proposal presented in [13], which was the main inspiration of our contract description language, is applied in the class-object design stage and does not discuss implementation level issues. Here we comment on some works directly related to our proposal. More detailed comparisons are available in [1].

The reflective middleware approach [16] allows for the provided services to be configured to comply with the non-functional properties of the applications. However, the proposal 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 with the application's program.

The Quality Objects (QuO) [20] provides a framework for the development of distributed applications with QoS requirements, based on CORBA. In QuO, the specification of such requirements is associated with 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 (similarly to the QuO approach), to the architectural level, in which the service provision can involve the reconfiguration of the application's topology.

Cazzola proposes the concept of using architectural reflection to adapt non-functional properties of base-level architectural configurations [8], describing a switching mechanism activated by logical rules for this purpose. The proposal described in [14] 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 similar to ours. CR-RIO complements both proposals providing an explicit methodology based on contracts and proposing extra mechanisms to deploy and manage these contracts.

[9] presents, in very general terms, the elements of a comprehensive pattern language for provisioning and managing quality-constrained services. We can relate many elements of the proposed pattern with similar elements in our framework. In particular its static application connector can be related to our architectural descriptions, and its dynamic connector can be related to the components of our contract support middleware. According to [9] the proposal requires access to the source code of the application and/or 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. Considering their similar aims, it would be very interesting to investigate further the correspondence between the two proposals.

As mentioned in Section 2(c), our approach is related to meta-level programming proposals described in the literature [24, 21, 12, 3]. Regarding this point, we observe that most of these proposals do not provide clear concepts and mechanisms for describing and supporting static and dynamic architectural adaptations, leading to ad-hoc solutions on this level of concern; more comparisons are available in [18]. Our contracts cater for this need by providing architectural level descriptions that act as specifications and can be used to guide implementations. In addition, the approach allows us to take advantage of formalisms for proving properties of the application's architecture [6].

It is interesting to note that many of the elements separately presented in the previously commented proposals have a counterpart in our proposal that tries to consider these elements comprehensively.

7 Conclusions

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 requirements of an application to be specified separately using high-level contracts expressed in an extended ADL. Being centered on an ADL-based configuration framework, the approach inherits the benefits mentioned in Section 6, among them the capability of reconfiguration, which facilitates to execute dynamic architectural adaptations on behalf of a contract. In addition, part of the codification, required to fulfill some non-functional requirements, can be encapsulated into connectors, which can be (re)configured during running time in order to cater for the impositions defined by the associated contract.

The approach has been partially evaluated through use-cases, which showed that the infrastructure required to enforce the contracts follows an architectural pattern that can be implemented by a standard set of components of a middleware. It was also verified that the code of these components could be automatically generated, except some localized pieces related to some specificity of the particular non-functional requirement under consideration. However, we should notice that the treatment of this kind of detail always has to be considered in any QoS-aware application. In this sense, our approach can help to identify the intervening hot spots and make the required adaptations more systematic.

It should be noticed that CBabel descriptions can be used to configure application systems based in different component implementation environments. This would also allow reusing QoS contracts in these environments, provided that the required supporting components are also made available. In this research, we found that the process of mapping architecture-level defined contracts to implementations could be better understood by exposing the internal structures of these components. We intend to follow this path in order to refine the framework and to identify useful optimizations for different implementation and application domains.

Finally, the reader can observe that the three case studies presented in Section 4 exhibit the same architectural style. In principle, with some changes, their individual contracts could be combined into a hyper-contract representing a fully adaptable and distributed VoD system. We believe that our proposal can contribute towards transforming such a visionary goal into sound engineering practice.

References

[1] S. Ansaloni. An Architectural Pattern to Describe and Implement QoS Contracts. Masters Dissertation, Instituto de Computação, UFF, May 2003.

[2] A. M. Barroso, J. Leite and O. Loques. Treating Uncertainty in Distributed Scheduling. *Journal of Systems and Software*, Elsevier, Vol. 63/2 pp. 51-58, November 2002.

[3] L. Bergmans, and M. Aksit. Composing Crosscutting Concerns Using Composition Filters. *Communications of the ACM*, Vol. 44, N. 10, pp. 51-57, October 2001.

[4] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *IEEE Computer*, Vol. 32, N. 7, pp. 38-45, July 1999.

[5] A. Borg, and A. Wellings. A Real-Time RMI Framework for the RTSJ. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, Porto, Portugal, July 2003.

[6] C. Braga and A. Sztajnberg. Towards a Rewriting Semantics to a Software Architecture Description

Language. *Electronic Notes in Theoretical Computer Science*, Elsevier, Vol. 95, pp. 148-168, May 2004.

[7] Carvalho, S. T, Lisbôa, J. and Loques, O. A Design Pattern for Software Architecture Configuration. 2nd Latin American Conference on Pattern Languages of Programming, Itaipava, RJ, Brazil, August 2002.

[8] W. Cazzola, A. Savigni and A. Sosio. Architectural Reflection: Concepts, Design, and Evaluation. Technical Report RI-DSI 234-99, DSI, University degli Studi di Milano, May 1999.

[9] J. K. Cross and D. Schmidt. Quality Connector: A Pattern Language for Provisioning and Managing Quality-Constrained Services in Distributed Real-Time and Embedded Systems. *9th Conference on Pattern Language of Programs*, Monticello, Il, U.S.A., September 2002.

[10] R. C. Cerqueira. A Methodology to Describe and Implement Contracts for Services with Differentiated Quality in Distributed Architectures. Masters Dissertation, Instituto de Computação, UFF, 2002.

[11] R. C. Cerqueira, R., S. Ansaloni, O. Loques, and A. Sztajnberg. Deploying Non-Functional Aspects by Contract. *2nd Workshop on Reflective and Adaptive Middleware, Middleware2003 Companion*, pp.90-94, Rio de Janeiro, Brasil, June 2003.

[12] T. Elrad, A. Aksit, G. Kiczales, K. J. Lieberherr and H. Ossher. Discussing Aspects of AOP. *Communications of the ACM*, Vol. 44, N. 10, pp. 33-38, 2001.

[13] S. Frolund and J. Koistinen. Quality-of-Service Specifications in Distributed Object Systems. *IEE Distributed Systems Engineering*, N. 5, pp. 179-202, UK, 1998

[14] D. Garlan, B. R. Schmerl and J. Chang. Using Gauges for Architecture-Based Monitoring and Adaptation. *Working Conference on Complex and Dynamic Systems Architecture*, Brisbane, Austrlia, December 2001.

[15] I. Georgiadis, J. Magee and J. Kramer. Self-Organizing Software Architectures for Distributed Systems. *Proceedings of the first Workshop on Self-healing Systems (WOSS'02)*, pp. 33-38, Charleston, SC, U.S.A., November 2002.

[16] F. Kon, F. Costa, G. Blair and R. H. Campbell. The Case for Reflective Middleware. *Communications of the ACM*, Vol. 45, N. 6, pp. 33-38, June 2002.

[17] O. Loques, R. A. Botafogo and J. C. B. Leite. A Configuration Approach for Distributed Object-Oriented System Customization. In *Proceedings of the Third IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, pp. 185-189, Newport Beach, California, U.S.A., February 1997.

[18] O. Loques, A. Sztajnberg, J. Leite and M. Lobosco. On the Integration of Configuration and Meta-Level Programming Approaches. *In Reflection and Software Engineering*, LNCS, Vol. 1826, pp. 191-210, Springer-Verlag, Heidelberg, Germany, June 2000.

[19] O. Loques and A. Sztajnberg. Customizing Component-Based Architectures by Contract. *In 2nd International Working Conference on Component Deployment (CD 2004)*, LNCS, V. 3083, pp. 18-34, Springer-Verlag, Edinburgh, Scotland, UK, May 2004.

[20] J. P. Loyall, P. Rubel, M. Atighetchi, R. Schantz and J. Zinky. Emerging Patterns in Adaptive, Distributed Real-Time, Embedded Middleware. *9th Conference on Pattern Language of Programs*, Monticello, Il, U.S.A., September 2002.

[21] J. A. D. Pace and M. R. Campo. Analyzing the Role of Aspects in Software Design. *Communications of the ACM*, Vol. 44, N. 10, pp. 66-73, October 2001.

[22] A. Popovici, A. Frei, and G. Alonso. A Proactive Middleware Platform for Mobile Computing. In *Middleware 2003 Proceedings*, LNCS 2672, pp. 454-473, Springer-Verlag, Rio de Janeiro, Brasil, June 2003.

[23] F. Siqueira and V. Cahill. Quartz: A QoS Architecture for Open Systems. In*Proceedings of the 18o Simpósio Brasileiro de Redes de Computadores*, pp. 553-568, Belo Horizonte, MG, Brasil, May 2000.

[24] G. T. Sullivan. Aspect-Oriented Programming Using Reflection and Metaobject Protocols. *Communications of the ACM*, Vol. 44, N. 10, pp. 95-97, October 2001.

[25] A. Sztajnberg and O. Loques. Bringing QoS to the Architectural Level. *ECOOP 2000 Workshop on QoS on Distributed Object Systems*, Cannes France, June 2000.

[26] A. Sztajnberg and O. Loques. Software Connectors Performance Evaluation. In*Cadernos do IME - Série Informática*, Rio de Janeiro (UERJ), V. 14, pp. 7-18, 2003.

[27] A. Tripathi. Challenges Designing Next-Generation Middleware Systems. *Communications of the ACM*, pp. 39-42, Vol. 45, N. 6, June 2002.

[28] R. West and K. Schwan. Quality Events: A Flexible Mechanism for Quality of Service Management. *Seventh IEEE Real-Time Technology and Applications Symposium (RTAS '01)*, pp. 95-104, Taipei, Taiwan, May-June 2001.

[29] R. Wolski. Spring, T. Neil and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computer Systems*, Vol. 15, N. 5-6, pp. 757-768, 1999.

[30] V. Zandy and B. Miller. Reliable Network Connections. In *Proceedings of the 8th annual international conference on Mobile computing and networking (MOBICOM'02)*, pp.95-106, Atlanta, Georgia, September 2002.