Self-adaptation of Fault Tolerance Requirements using Contracts

André Luiz B. Rodrigues\textsuperscript{1}, Leila N. Bezerra\textsuperscript{1}, Alexandre Sztajnberg\textsuperscript{1}, Orlando Loques\textsuperscript{2}

\textsuperscript{1}DICC/IME e PEL/FEN - Universidade do Estado do Rio de Janeiro, Brazil
\textsuperscript{2}Instituto de Computação - Universidade Federal Fluminense, Brazil

\{rblandre, leila\}@ime.uerj.br, alexszt@uerj.br, loques@ic.uff.br

Abstract

Fault tolerance is a constant concern in data centers where servers have to run with a minimal level of failures. Changes on the operating conditions or on server demands, and variations of the systems own failure rate have to be handled in such a way that SLAs are honored and services are not interrupted. We present an approach to handle fault tolerance requirements, based on component replication, which is supported by a context-aware infrastructure and guided by contracts that describe adaptation policies for each application. At run-time the infrastructure autonomically manages the deployment, the monitoring of resources, the maintenance of the fault tolerance requirements described in the contract, and reconfigures the application when necessary, to maintain compliance. An example with an Apache web server and replicated Tomcat servers is used to validate the approach.

1. Introduction

Distributed systems are subject to faults caused by malfunctions in the software or in the hardware infrastructure on which they are executed. The use of fault tolerance techniques allows for the recovery of these systems leading to the continuity of services \cite{10}. Fault tolerance is usually obtained through the redundancy of software and hardware elements, which can be underused since they are often statically allocated.

Some approaches to introduce fault tolerance use ad hoc solutions which mix the code responsible for meeting the functional requirements with the code for redundancy through the replication of components, and for the maintenance of the consistency of the replicas. The result is a highly coupled and non-reusable code. Other approaches eliminate this problem by using mechanisms embedded in the supporting infrastructure on which they are developed. Still, the configuration of these mechanisms is statically defined, forcing the pre-allocation of resources. Examples of systems that use this approach are the JEE and .NET.

Adaptive approaches to support fault tolerance requirements try to achieve a balance between robustness and an efficient use of resources. Differentiated replication and consistency techniques can be used in specific operation contexts while utilizing the necessary resources (processing resources or time, network bandwidth, etc.). The allocation of resources in this case is "green" in the sense that the redundant resources are used only when necessary.

The use of adaptive fault tolerance techniques has become attractive in data centers, where servers must run without perceivable interruption of the services they provide while maintaining a certain level of quality. Failures and changes in resource load should be mitigated so that SLAs (Service Level Agreement) are met. In this context it is desirable that the fault tolerance requirements and adjustment policies can be described in a high level of abstraction. It is also desirable that these requirements can be met with separation of concerns for each application, and managed autonomically during run-time.

In this article the use of architectural contracts is proposed to specify fault tolerance requirements, where profiles quantify properties such as replication type, the number of replicas, and the desired checkpointing interval. Additionally, a negotiation machine specifies the desired levels of quality and how these are imposed. During run-time a context-aware software infrastructure allows the deployment and maintenance of the requirements described in the contract in an autonomic form, dynamically reconfiguring the application and dynamically selecting/allocating the resources responsible for fault tolerance to keep the contracted requirements.

To validate this approach, contracts described in CBabel and the support infrastructure of CR-RIO were used to provide fault tolerance to a scenario containing
an Apache HTTP server integrated to a group of Tomcat servers [1]. The idea of the example application is to dynamically trigger an appropriate replication technique for each specific operating context, considering the response time of the replica set and the related fault rate.

Section 2 presents the CR-RIO components. Section 3 presents the integration of the support elements in CR-RIO and the specification of replication requirements through contracts. Section 4 then presents the Apache-Tomcat application and discusses some aspects of its implementation. In Section 5 related work are mentioned. In Section 6 conclusions and future work are presented.

2. The framework for contracts

The CR-RIO framework (Contractual Reflective-Reconfigurable Interconnectable Objects), developed in our group, is centered on an architectural model and on the CBabel description language, to describe the functional architecture of applications and express their non-functional requirements through contracts [14]. Based on these elements, a support infrastructure (i) interprets the contracts and store them in a repository as meta-information associated to the application, (ii) provides mechanisms for reflection and dynamic adaptation, which enable adjusting the application's configuration as well as the supporting elements to meet the demands of the contracts, and (iii) provides elements consisting of a set of components forming reusable patterns to configure, monitor, and maintain them during operation.

2.1 Contracts

The functional configuration of an application is defined by the specification of the architectural components that perform its essential activities. Non-functional requirements are defined by the operational or quality restrictions and can accept some negotiation involving the used resources. A contract describes non-functional aspects of the application specifying the resources to be used during operation and acceptable variations in the availability of these resources:

**Categories.** Describe properties and non-functional aspects of specific components, features, or services including processor, memory, and communication features. Less measurable aspects such as price range ("expensive", "cheap"), fault tolerance, or the quality of a service ("good", "medium", "bad") can also be described. Each category is associated to components or to supporting services, which will allocate and monitor their resources and can use the available infrastructure in order to do it.

**Profiles.** Quantify or value the properties of a Category, constraining each property, working as an instance of acceptable values for a Category. Profiles can be defined with the desired granularity to indicate the acceptable quality level in the operation context of individual components or parts of the architecture.

**Architectural configurations or services.** Specify versions of the architecture that will define the possible quality levels or operational states of the application. Each configuration contains a description of architectural components associated to one or more profiles, specializing the basic architecture. The desired / tolerated quality level of a configuration is made different from another by the set of properties declared in the profiles. A configuration can only be deployed or maintained if all profiles associated to it are valid.

**Negotiation Clause.** Describes a policy defined by a state machine establishing an arbitrary order for the deployment of configurations. According to what is described in the clause, when a configuration with higher preference (high quality standards, for example) can no longer be maintained the contract management support will try to deploy a lower preference configuration (with lower quality or requiring fewer resources). The return or upgrade to a high-preference configuration can also be described, allowing a better quality setting to be (re)established if the resources necessary to it become available again.

2.2 Support infrastructure

The support infrastructure (Figure 1) consists of elements with well-defined roles in the deployment and reconfiguration of the applications and in the autonomic management of contracts.

**Configurator.** Element responsible for mapping the architectural descriptions (in CBabel) into actions that carry out the required settings in the native systems. It provides two APIs: configuration and architectural reflection; through which the configuration facilities are used. The configuration API allows initializing, connecting, stopping and replacing components to deploy and reconfigure the application. These operations are atomic and reflected in the persistent meta-level repository, which can be queried through the architectural reflection API.

**Contract Manager (CM).** Responsible for the deployment of differentiated services and for the management of the policies described in the contract. To deploy one of the configurations in the contract, the CM (i) sends the set of profiles of all the configurations to the Contractors, (ii) asks them to verify the constraints required by the profiles...
associated to the configuration to be deployed and (iii) waits for their notification. If all Contractors respond positively, the selected configuration can be deployed. Otherwise, another configuration should be selected according to the negotiation clause, and the deployment procedure is then restarted. The same occurs if during the operation of a given configuration the CM receives an invalid profile notification. Also, if none of the configurations described in the contract can be maintained/deployed the application is terminated. The CM can also start a new negotiation when the resources for the deployment of a preferred configuration become available (the profiles are valid), even if the current configuration is still valid.

To carry out the deployment of the architectural components of the selected configuration (recently negotiated with the Contractors) the CM uses the Configurator. The actual configuration steps to let the application leave one configuration and go for the new one are planned by the CM to be consistent. The approach uses transactions and exception handling with a nested sequence of reconfiguration commands requested to the Configurator based on the topology of both configurations [13].

Contractor. Coordinates the allocation and monitoring of the basic components (mechanisms, resources or services). A distributed application requires a Contractor in each node of the domain. It receives from the CM the set of profiles of all configurations. Periodically, each Contractor queries the Context Service for the set of monitored values of the properties of interest and compares them to the constraints described in the profiles. The Contractor notifies the CM that the current configuration is no longer valid if at least one of the profiles is violated. It also notifies the list of all valid profiles including those related to the current configuration.

Resource Agent (RA). Encapsulates the specific access to basic elements (resources, services, components, etc.), providing interfaces for management, and monitoring of values of required properties. The RAs are specialized and can access primitive services. The monitored values are sent to the Context Service.

Context Service (CS). Responsible for providing context information and hiding low level details used in communication with the (various) RAs. The application is only concerned with the necessary data and not with how it is obtained. Upon receiving a query from the Contractor, the CS verifies the context properties required for each of the resources (i.e., properties of interest described in the profiles). Then, the CS (as opposed to the application) communicates with each RA involved to obtain the individual context information. After collecting the information, the CS returns the consolidated result back to the Contractor.

![Figure 1. CR-RIO Infrastructure](image)

**Discovery Service** (DS). Consulted when an application does not know in advance the specific component to be used, but knows the type or required properties. For this, the class of the desired resource is informed as well as the context constraints that it must meet. For example, the application needs to access a Web server that has an average response time of at most 1 second. In this case, the DS must consult the CS to obtain the values of the context properties of pre-selected elements. The response of the DS is a list of references of all resource instances of the class required by the application which meet the imposed context constraints. One element of this list can then be selected.

It is worth noting that the contract support semantics consistently relies on the elements of the infrastructure. For example, if the architecture of the application statically specifies a particular TomCat module (for which the reference is previously known), the CM simply commands the Configurator to deploy and allocate the specific component. On the other hand, if the module is specified through dynamic allocation (for which the reference is not yet known), the CM calls the DS to discover and select a Tomcat component before requesting its allocation to the Configurator.

With the presented concepts and elements in mind we built our approach to fault tolerance.

### 3. Architecture, categories, and profiles

In our approach replication is considered a support service that can be used and referenced in a contract. Replication and fault properties in the architectural level are described by Categories. The **Replication**
category (Code 1) defines the properties of the replication service, inspired in [15], indicating what can be required of this service, and also what can be tracked, regardless of the used technique.

```java
1  category Replication { 
2    numberOfReplicas: numeric; 
3    checkpointInterval: numeric s; 
4    monitoringInterval: numeric s; 
5    timeoutInterval: numeric ms; 
6  };
7  profile
8    Replication.monitoringInterval = 20;
9    Replication.timeoutInterval = 200;
10 } ActiveCP;
11 profile
12    Replication.numberOfReplicas = 4;
13 } ActCNRepP;
```

**Code 1. Replication specification for Cyclic Active**

Properties: (i) the number of replicas, (ii) the interval for the checkpoint and for the trigger of the consistency protocol (Line 3), (iii) the monitoring interval for each replica (Line 4) and (iv) the limit time for each replica to answer when monitored (Line 5). With these properties the RM can identify that the number of replicas is out of specification. For example, the ActiveCP profile indicates that each replica should respond to the monitoring process every 20s, and faster than 200ms. A replica will be considered unavailable if it does not respond within this interval. The ActCNRepP profile, separated for modularity, indicates that the group should have 4 replicas.

```java
1  category Faults{ 
2    numberOfFaults: decreasing numeric; 
3    faultInterval: decreasing numeric s; 
4    stableInterval: increasing numeric s; 
5  };
6  profile
7    Faults.numberOfFaults=2;
8    Faults.faultInterval=15;
9    Faults.stableInterval=60;
10 } ActiveCFaults;
```

**Code 2. Faults specification for Cyclic Active**

The Faults category (Code 2) is proposed to specify the properties related to faults: (i) the number of faults tolerated before the configuration becomes invalid (Line 2); the interval during which the faults may occur (Line 3); and the minimal interval required for the group of replicas to be considered stable (Line 4). The stableInterval property, properly used in a contract, allows a less robust technique to be used since the number of failures is, for some time, below the specified. Moreover, it applies a certain delay to the control decision, avoiding instability due to transient conditions. For example, the ActiveCFaults profile (Lines 6-4) specifies that 2 faults are tolerated each 15s, and that the group is considered stable if it does not present faults for 60s.

The profiles based on the Replication and Faults categories will be used together in a contract to specify the level of fault tolerance required. During runtime they will be used to evaluate whether this level is being respected or violated and should follow the basic architecture described in Section 2. It is necessary then to include the software components effectively responsible for managing these properties.

The elements to support the replication of components, recurrent in several proposals, such as [9], [15] were integrated into the CR-RIO's infrastructure: (a) a Replication Manager (RM), (b) the group of replicas and (c) Replication Controllers (R-CTL) for each individual replica.

In our solution (Figure 2) the role for the Replication Manager (RM) is encapsulated in a Contractor. Based on the Replication and Faults profiles, it controls the quality of service, assessing whether the replicas are "alive" and if the number of replicas or faults are within appropriate parameters. Observe that the RM performs its activities independently from the replication technique.

The interactions between a client module and the replicated modules are mediated by a group communication element. Since this is an interaction role¹, a group connector (GC) is employed to multicast the replicas, and an RA associated to this connector monitors the communication and the quality of the group communication.

Each module of the replica set has its interaction with the other modules of the application intercepted by a Replication Controller (R-CTL). This element supports the various maintenance strategies for the consistency of the replicas without interfering directly in the replicated modules. Each replication technique is associated to a specialized R-CTL. The R-CTL receives a profile containing the properties (interval and timeout) to be monitored. An associated RA pro-actively performs the tests and sends the results to the CS. Thus it is possible for the RM to query the CS to verify if each replica meets the replication profiles.

The R-CTL has autonomy to perform election procedures when necessary and behaves properly when elected as the primary (answering the requests, persisting the state, group consistency).

¹ Please note that modules represent functional components of the application and connectors represent the mediators of the interconnections between the modules (as so are considered non-functional elements). Connector chains can be interposed on the route of interaction between modules, allowing filtering, constraining or even distributing the interactions.
Once a replication technique is selected and the corresponding configuration is deployed, the set of R-CTL performs the procedures to achieve consistency after a reconfiguration and the RAs start to monitor the properties of interest as declared by the profiles. The RM regularly consults the CS, which presents the consolidated context information from the various RAs. The RM checks then if the time intervals and number of replicas are maintained within the profiles of the current configuration and whether they meet the profiles of other configurations. When receiving a notification from the RM the CM can either select another replication technique or maintain the current one according to the policy described in the negotiation clause of the contract.

4. Application example

To validate the presented approach, a scenario usually found in data centers was taken as an example: an HTTP Apache server and a set of Tomcat application servers. The use of a set of replicated servers may have the objective to improve the scalability of the service (or to decrease the response time) and to make the system more robust, mitigating failures through redundancy. In our example, the concern is related to the average response time of the group of Tomcat modules and fault tolerance:

(a) Under normal access load, with response time below 200ms, the passive replication technique will be used with only 2 Tomcat servers. The goal is to reduce the use of resources while the requests are still processed on time by the primary replica;

(b) If the access load increases and the response time increases to more than 200ms, up to 4s, the cyclic active replication technique will be used with 4 Tomcat servers. The policy is to increase the number of replicas to improve the throughput of treated requests and use a more robust replication technique, even if it requires more resources.

Figure 3 shows the general diagram for the architecture of the application. A request coming from a Web client is processed by the Apache module which identifies by the URL that this is dynamic content, which must be processed by a Tomcat module.
In a scenario without replication, the Apache-Tomcat communication is provided by two elements: Mod_JK and AJP, connectors available in their respective products. In our architecture this flow is intercepted by the MOD_JK_G connector, which implements the group communication.

Upon receiving a request from the Apache the MOD_JK_G connector broadcasts the request for the R-CTL connectors on each Tomcat replica. They perform the appropriate consistency procedures for the established replication technique, and forward the request as appropriate. For example, in the case of the cyclic active replication, the R-CTL holding the token forwards the request to the processing of the corresponding Tomcat module via AJP connector. The others will drop the request. In the case of passive replication, the R-CTL of the primary replica would pass the request to the Tomcat module, and send the state to the secondary replicas according to the checkPointInterval property.

After processing the request, the Tomcat module returns the response to its R-CTL that puts it on its way back.

4.1. Architecture and contract

Once outlined with the fault tolerance policy, the architecture of the application is described (Code 3). The module classes are listed: Apache, which will act as a client, and Tomcat, which will be replicated (Line 1). The same happens with the specific connectors used in the application architecture (Line 2).

```
1 module Apache, TomCat;
2 connector Mod_JK_G, AJP, CTLRp,
   CTLRs, CTLRa;
3 module{
4   group TCGroup; // TomCat replicas
5   instantiate Apache as ap, Tomcat as tc;
6   join tc to TCGroup;
7   link ap to TCGroup by Mod_JK;
8 } webApp;
9 } start webApp under webContract;
```

**Code 3. Architecture description of the example**

A reference to the group of Tomcat replicas is created (Line 4), and references to the instances of the modules are declared (Line 5). Connector instances are created automatically. The module tc is included in the TCGroup group (initially the group has only one element). Finally, the topology is described by connecting the ap module to the elements of the TCGroup group through the Mod_JK_G connector. Line 9 states that the module webApp should be initiated under the webContract contract. Due to limited space, the semantics description was not detailed. It is worth noting that a CBabel description is declarative. Actions are performed during deployment, to load the application according to this description.

The next step is the description of the contract that specifies in the architectural level the replication and fault policies previously discussed. Each policy is the seed for a different architecture configuration:

- passServ, for the passive replication with hot standby where a replica is elected as the primary and only this one processes the requests. The state of the secondary replicas are updated on every checkpoint;
- actCServ, for the cyclic active replication, round-robin style, where several replicas periodically assume as the primary, circulating a token. In this case there is no status update based on checkpointing.

In each configuration (Code 4), passServ (Lines 2-8) and actCServ (Lines 9-15), architectural structures are specialized to incorporate the replication architectural elements, and to associate them to the appropriate profiles. In the passServ configuration the TCGroup group will be constrained by the PassNRepP and PassiveFaults profiles (Line 3) and will only be considered valid if all of their properties are valid. An array of Tomcat modules is structured with the desired number of replicas, and each selected instance is constrained by the PassiveP profile (Lines 4-5). Then, the module is incorporated to the group (Line 6). Finally, the Apache module, ap, is connected to the group TCGroup by a composition of connectors: Mod_JK_G, CTLRp, a specialized R-CTL for passive replication, and the AJP connector to adapt the Tomcat module interface. This composition is associated with the comPassP profile (details not presented), which constrains the response time (Line 7).

The select* construction indicates that the specific module will be dynamically selected using the Discovery Service (see Section 2). It is parameterized by the class of the module (TomCat, in the case) and by the profiles associated to the instantiate statement. In our case, the profile specifies that the selected replica must have the timeoutInterval parameter validated (Lines 5 and 14).

The select* construction indicates that the DS will be continuously monitored, and if a more capable instance of the requested class is available, the current module will be replaced. So it is possible to perform located and atomic repairs in the configuration without the need for the intervention of the RM [5].

For example, in the case of an impending failure in a node, monitored by the timeoutInterval, the reference of a new module can be discovered, replacing the current one. This avoids the sequence of invalid profile and service notifications, and the firing of another negotiation and deployment procedure.
contract{
  configuration{
    group TCGroup with PassNRepP, PassiveFaults;
    for (i=0; i < PassiveP.Replication.numberOfReplicas; i++) {
      instantiate tc[i]= select*(TomCat) with PassiveP;
      join tc[i] to TCGroup;
    }
    link ap to TCGroup by Mod_JK_G > CTLRp > AJP with commPassP;
  } passServ;
  configuration{
    group TCGroup with ActNRepP, ActiveCFaults;
    for (i=0; i < ActiveCP.Replication.numberOfReplicas; i++ {
      instantiate tc[i] = select*(TomCat) with ActiveCP,;
      join tc to TCGroup;
    }
    link ap to TCGroup by Mod_JK_G > CTLRa > AJP with commActCP;
  } actCServ;
  negotiation{
    not passServ -> (actCServ || out-of-service);
    actCServ -> passServ;
  }
} webContract;

Code 4. Contract for the example deployment

The negotiation clause effectively maps the fault tolerance requirements in a state machine, which determines the deployment policy, the priority, and the possible transitions between the configurations previously described.

Once the application is in operation, this policy is managed autonomously. The system will only suffer manual intervention once none of the configurations specified in the contract can be established or maintained. The order of the negotiation rules (Lines 16-18) determines their priority and the configuration in the left part is the one to be deployed and monitored. The service with higher priority (Line 17) is described in the passServ configuration. If this one can not be established or maintained because one of the profiles has been violated, the CM will try to deploy the actCServ configuration. In other words, if the number of faults increases, a service with a more robust replica configuration will be used. If any of the configurations can not be deployed a special service, out-of-service, is deployed, indicating that the application can not run with the required quality. On the other hand in the rule on line 18, there is no “not” condition. This means that if the current configuration is the one from the actCServ service (i.e., the profiles for this service are valid), and the passServ service can also be deployed (i.e., the profiles of this service are also valid), then the transition to it is unconditional. This way it is possible to express the requirement to return the system to a service with a replication technique that requires fewer resources, with fewer replicas.

4.2 Deployment

To evaluate the webContract contract, we developed some application specific components and integrated them to the previously developed infrastructure [5], [19]. Java classes were developed for the group connector MOD_JK_G, for the R-CTL connectors, and we customized RA abstract classes for the RAs for the Replication and Faults categories.

The original communication between Apache and Tomcat is done through the Mod_JK connector [1], which forwards the requests to the AJP connector, a standard in the Tomcat server. In our solution, the MOD_JK_G connector is put in the way to receive the Apache requests from Mod_JK and carry out the group communication, forwarding these requests to the group of R-CTLs. It was also necessary to address the Mod_JK and AJP specific protocols within the code. Figure 4 shows the simplified interaction diagram for this composition of elements. The 1.2.1.1.1 interaction is exactly the 1:n communication and the corresponding return is n:1.

The group communication was implemented with the JGroups package [2], through the RPCDispatcher class, which provides a mechanism for dynamic invocation on the client and procedure remote call to the servers (a little more complex than a group RPC).
Note in the following call that the second parameter “sendBytes” is the name of the remote method to be invoked in each replica, “buffer” contains the data sent by Apache, “class” is a vector with the types of parameters in “buffer” (used to recreate the invocation using reflection on the remote side).

```
RspList rsp_list = disp.callRemoteMethods((Vector) null, "sendBytes", 
     new Object[]{buffer}, 
     new Class[]{byte[].class}, 
     GroupRequest.GET_ALL, 0L);
```

The “server” side (ServerConnector in Figure 4) encapsulates the functionality of the R-CTL connectors, implementing the specific characteristics of each replication technique, and the interactions with the AJP connector. It was necessary to adapt this interface to the scheme of reusing open connections between Apache and Tomcat. For this, JNI package threads were used. The support provided by JNI for threads and for non-blocking I/O calls is more scalable. To implement the R-CTL specializations for each replication technique the strategy pattern [7] was chosen instead of separate classes.

Some performance tests were carried out with JMeter [1], which allows stressing the Apache server. In a preliminary test, JMeter was set to simulate 10 simultaneous users requesting 8Kbytes size documents to the two Tomcat replicas running on the same machine, with active replication. The time measured in the test was ~4s, versus ~400ms on the test with only one Tomcat instance and without the replication infrastructure (order of magnitude similar to that found in [6]). In addition, to complete and refine the implementation, tests will be executed on distributed scenarios, as the example requests. This will also help detecting limitations in the approach.

5. Related Work

Application servers such as JEE and .NET provide replication mechanisms, but do not allow the configuration of the used technique. The configuration of the JBossCache infrastructure [11] to replicate the cache objects is ad hoc, for instance. Considering the Apache-Tomcat, it is possible to configure load balancing using the Mod_JK connector, but adaptive replication techniques are not supported either.

An adaptive approach to fault tolerance in replicated services is also explored by [12] in a similar work. The main concerns of the authors are the architecture and the performance of the supporting middleware. The specifications are however, ad hoc and embedded in the code of the services. In [4] a self-replication mechanism is designed based on multi-agent systems. However fault-tolerance is not considered. In our example the number of replicas configured guided by the contract.

The organization of the elements for fault tolerance in our approach is based on [15] an extension for the FT-CORBA standard from OMG. However, FT-CORBA is also not adaptive. Once defined the requirements for fault tolerance can not be changed. In [16] an adaptive infrastructure for fault tolerance is presented called GroupPac, a free implementation of the FT-CORBA standard. Using this infrastructure it is possible to create programs based on CORBA that change fault tolerance properties according to rules within the program. But, these rules are programmed in an ad-hoc manner. In our approach fault tolerance is specified in a high level and integrated in a supporting infrastructure that can be used in various applications.

The management of adaptive applications such as those applied in our work requires a supporting infrastructure that includes (i) a form for specifying...
quality and adaptation policies; (ii) mechanisms for configuring, deploying, and adapting the application’s components and (iii) mechanisms for discovering and monitoring components and resources [5]. Although this infrastructure is not the focus of this paper, the discussion is worthy. Frameworks for managing and supporting distributed applications [17, 20] usually deal with dynamic requirements but, in general, do not support resource discovery or handle it as manually programmable hotspots.

Recent proposals offer convenient services for ubiquitous and pervasive applications. For instance, Rainbow [8] allows the specification of elements to be monitored and quality requirements of an application to be guaranteed by adaptation strategies. The CR-RIO is comparable to Rainbow in some points, but adopts an ADL based on modules, connectors, ports and a contract governing dynamic configuration. This approach paves the way to a formal description of the ADL, facilitating formal verification. Rainbow allows more flexible reconfiguration strategies to be specified than does CR-RIO, but this turns formal verification more difficult to apply. Besides, CR-RIO and CBabel are being developed in our group easing the assembling of the prototypes.

6. Conclusions and future work

In this work we presented an approach that provides a way of specifying fault tolerance policies in a high-level of abstraction, using contracts and a reusable software infrastructure to deploy and maintain the specified policy. The infrastructure provides context-aware services. A contribution from our approach is the synthesis of dynamic fault-tolerance requirements into contracts, and the binding between the semantics of contracts and the corresponding actions in the supporting infrastructure. Moreover, this semantics facilitates formal verification on the fault tolerance specifications before deploying the application [3].

Other non-functional requirements, such as request rate, can be considered in a contract. This could allow load balancing of the replicas or manage power-aware servers maintaining the replicas with an acceptable energy cost [18].

Our approach provides autonomic capabilities of self-configuration and self-optimization in response to the operation context [5]. Specifically, the best replication technique for the current context is deployed dynamically, keeping the application within the required quality.

Acknowledgement. The authors would like to thank FAPERJ and CNPq for the support.

References