A new architectural approach for dynamic adaptation of components-based software using multi agent system

Chouarfia Abdallah[1], Bouziane Hafida[1]

[1]*University of Sciences and Technology Oran Mohamed Boudiaf*
Computer Sciences Department
Oran, Algeria
chouarfia@univ-usto.dz, hafida_bouziane@univ-usto.dz

Abstract — Component-based development has become a commonly used technique for building complex software systems by assembling a set of existing components. In general adapting an application means stopping the application and restarting it after the adaptation. This approach is not adapted for a large classes of software systems in which continuous availability is a critical requirement, hence the need of adapting the application at run-time. In the paper we present an architecture based approach for dynamic adaptation in component-based software. We are interested in the dynamic adaptation independently of the nature of the system to be adapted. Also In the case, we use an agent based system to perform the adaptation. The agent system is guided by an architectural description of the adapted application. The adaptation mechanism is implemented in the connectors using the flexibility offered by the Java Scripting programming technique.
Key words: components, dynamic adaptation, multi agent system, architecture description, configuration

1. INTRODUCTION

The development of large software by assembling existing components is the objective of the components-based development (Szyperski, 1998). The permanent evolution of user's needs and the fast changing in the execution environments make the software adaptation a primordial task. In components-based development paradigm, an adaptation can address (Aksit and Choukair, 2003): *Architectural changes*, *geographical changes*, *Interface modification*, or *implementation adaptation*. The architectural changes consist of adding or removing components or modifying connections among them. The geographical changes correspond to the migration of components from a site to another one. The interface modification consists of changing the interface of a component to make it more compliant to the caller's expectations, while the implementation adaptation affects the internals implementation of components without changing interfaces.

Traditionally, an application is stopped to be adapted. This approach is not suitable for critical systems that have to be non-stop and highly available like bank, internet or telecommunication services. In these kinds of systems the adaptation must take place at run-time and the application should not be entirely stopped. Unfortunately, realize such adaptation is not trivial; there are several conditions and constraints to be verified, and many problems to overcome. Some important problems to be considered to make a dynamic update are (Aksit and Choukair, 2003):

- *Maintaining application consistency*: states of the components must not be affected by changes in the application architecture.
- *Preserving bindings of the components*: Bindings have to be preserved by redirecting the calls to new components and managing transient states.
- *Initializing new components*: New components must be initialized with adequate internal state according to the former component.
- *Preserving communication channels* by avoiding message loss, duplication or excessive delays.

In the paper, we describe our approach to achieve dynamic adaptation of components-based software applications. The idea is to introduce between two components a connector (unit of interaction) (Kell, 2007) that intercept and redirect inter components communications. The adaptation is made through the adaptation system by acting on connectors. The adaptation system is not integrated in the application, it is an independent system composed of software agents, whose role is to perform and supervise the adaptation operations. The agents system is guided by a knowledge base, which contains:

i) a set of rules that condition the adaptation launching,

ii) the architecture description of the adapted application, that enables the adaptation system to ensure the validity and the coherence of adaptation.

The paper is structured as follows: section 2 presents related works to the dynamic adaptation. Section 3 describes the proposed solution to achieve a dynamic update of components-based software applications. The implementation details and some measurements relative to our solution are given in section 4. Section 5 concludes and presents some perspectives.

2. RELATED WORKS

Several works dealt with dynamic adaptation problems, hence the emergence of several approaches.

In *the model driven approach*, the dynamic adaptation is based on a components model that designed to support this kind of adaptation. DCUP (*Dynamic Component Updating)* (Plasil et al., 1997) is an example of this approach. In DCUP the component is divided into two parts: *permanent part* and *replaceable part.* Adapting a component means replacing its replaceable part by a new version at run-time.

In *The reflexive approach*, an application has an abstract level (meta-level) that reify the real system. The adaptation is made first on the meta-level, after that, the changes are reflected on the executed applications thanks to the causal connection between the meta-level and the real system. An example of this system is DYVA (Ketfi,2004): a reflexive framework for dynamic reconfiguration of components-based applications. The framework is decomposed in two main parts: *The base-level* represents the concrete application that provides the expected functionalities and its execution environment and *the reconfiguration machine* that contains

i) the different operational modules responsible for achieving the reconfiguration,

ii) ii) the meta-level which represents the reification of the concrete application.

The architectural approach uses explicit description of the executed application through specific languages: the Architecture Description Language: ADLs (Medidovic and Taylor, 2000). An ADL describes an application in term of components, connectors and connection among them. The adaptation in this approach is verified and validated in the architectural level before to be applied on the application. This approach is used by (Qun et al., 2006); the adaptation takes advantage of both meta-architecture and the mobile agents. It uses an architectural model to guarantee the safety of the reconfiguration, while uses mobile agents to automate the adaptation process in a flexible way.

In *the flexible middleware approach*, the adaptation is delegated to the execution platform. In such system the dynamic adaptation is looked like a non factional properties offered by the middleware, like security and transactions management. In (Brinkschulte et al., 2005), the adaptation is based on the real time middleware OSA+ (Brinkschulte et al., 2002). The objective is to be able to reconfigure services during run-time, with a predictable and predefined blackout time (the time where the system does not react due to the reconfiguration).

The aspect oriented approach is based on the aspect oriented programming technique (Kiczales et al., 1997), in particular the dynamic aspect, which involve plug and unplug of aspects without stopping, and restarting a running system. DAOP (Pinto et al., 2003): Dynamic Aspect-Oriented Platform is an example of such system. DAOP provides a composition mechanism that plugs aspects into components dynamically at runtime. The composition between components and aspects is established during runtime interaction and is governed by a set of plug-compatibility rules in order to guarantee their correct interoperation.

Authors in (Oreizy et al., 1998) describe ArchStudio, a tool that implements an architecture-based approach to runtime software evolution. The approach is based on an explicit architectural model, which is deployed with the system and used as a basis for change. The connectors in the system are first class elements that have an important role to support run-time changes. An imperative language is used for modifying architectures. The tool supports adding, removing and replacing components and connectors, and changing the architectural topology. On the same axis, (Georgas and Richard, 2004) presents an architecture-centric approach to self-adaptive software applied to systems constructed using independent components interconnected through first-class connectors, both explicitly modelled using architectural descriptions. The architectural models are used as the basis for the decomposition. The architectural models representation is make using xADL 2.0 language (Dashofy et al., 2001): a highly extensible, XML-based ADL. An extension of this language is also used to define the structure of observations, responses, and adaptation policies.

In [Qun et al., 2006), the adaptation takes advantage of both meta-architecture and the mobile agents. It uses an architectural model to guarantee the safety of the reconfiguration, while uses mobile agents to automate the adaptation process in a flexible way.

(Huang et al., 2006) present an approach to recover software architecture from component based systems at runtime and changing the runtime systems via manipulating the recovered software architecture. As soon as software architecture is recovered, the runtime system can be observed, reasoned and adapted through its architecture views. The approach supports the addition, deletion and replacement of the components and connectors.

3. PROPOSED SYSTEM

The architecture of the proposed system (Belabed and Chouarfia, 2008) consists of two main parts: the Knowledge Base and the Multi Agent System SMA in the Fig 1:

3.1 The Knowledge Base: KB

The KB is composed of two parts: the adaptation policies, *a set of rules*, which defines the adaptation policy, and the *application architecture description*. The formalism used In the database is based on the logic of predicates with Prolog based implementation; the choice of this formalism is justified by:

- The important role that has the architecture description in the adaptation mechanisms involves a large number of inferences makes on the architecture description to guide the adaptation. Such mechanism is provided in Prolog.
- *Prolog* can be easily used as descriptive language. A predicate which presenting a fact is similar to an

XML tag, for example: < *tag* > *value* < *tag* >, can be written in P*rolog as tag (value)*. More, *Prolog* is used in several projects to represent more complex structures such as ontology (Samuel et al., 2006), which motivated us to use it as an architecture description language.

- The existing tools facilitate the use and the integration of *Prolog* formalism with other languages (such the *java/Prolog* interface), this avoid us to reprogramming the necessary inferences mechanisms needed in our approach.

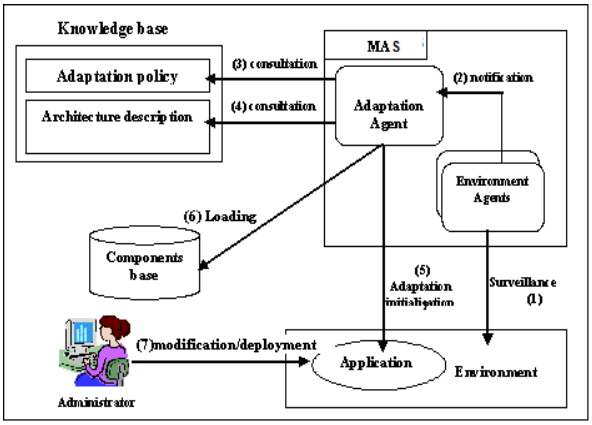

Fig 1: Proposed architecture

3.1.1 The adaptation policy

This base contains a set of rules that condition the adaptation triggering according to the values of certain environment variables. The rules are in the following form:

If <event> Then Action

An event can be a significant change in one of the environment variable like the used memory, network band-width,).The following example shows the form of rule that trigger an adaptation (replacement of component in the example) if the rate of the used memory exceeds the value "val".

Event ('t_Memo', Value):- Value > val, Replace (compo1, compo2).

The predicate **"**Event**"** has two parameters: the first specifies the type of context concerned by the event (the rate of the used memory); the second specifies the value of the event. The action "Replace (*compo1*, *compo2*)" will be triggered only if the condition "Value > val" is true.

3.1.2 Architecture description

This base contains:

- The detailed specification of each component of the base of components in term of provided and required interfaces and operations of each interface.
- The architecture description of the executed application (components and interactions).
- The Inferences rules, which used to deduce the correspondences and compatibilities between components.
- A set of rules is used to ensure the coherence and the validity of the adaptation.

The detailed specification of the proposed formalism is out the scope of this paper.

3.2 Multi Agent System MAS

Using MAS is justified by two main reasons:

- To adapt a distributed application requires using a tool which was acquired in MAS.
- The design of MAS is in mature phase, we can use an MAS platform qualified efficient and stable. In addition, the qualities of MAS such as communication, flexibility, scalability and mobility facilitate the tasks of adaptation.

The MAS contains two agents: an *adaptation agent A-A* and an *environment agents E-A* (Fig 2).

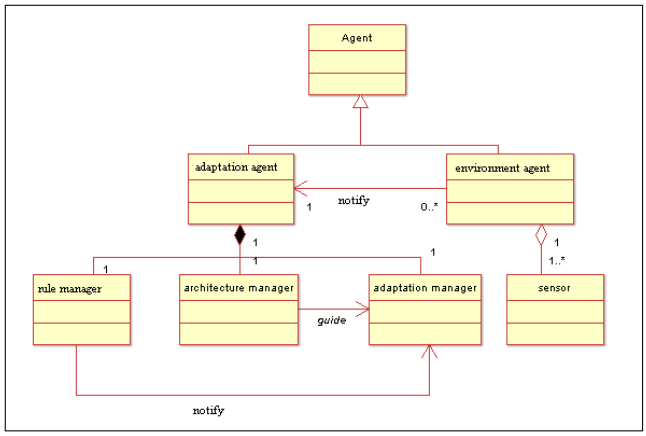

Fig 2: Multi Agent System class hierarchy

1. *Adaptation agent A-A* : The role of this agent is
   - Takes decisions about the adaptation triggering, according to the E-A notifications. The A-A uses the adaptation policy rules to accomplish this task.
   - Achieves the adaptation according to the actions deduced from the adaptation policy rules. The adaptation operations are guided by the architecture description base.
   - Modifies the architecture description base after each operation according to the realized changes on the application level.

The A-A achieves the above operations through three components: *the rules manager*, *the adaptation manager* and *the architecture manager*.

The rules manager is responsible for the manipulation of the adaptation policy rules, if a decision to adapt is taken, the rules manager informs the adaptation manager to perform the operation. This operation is guided by the architecture manager whose provide the necessaries inferences from the architecture description base. The architecture manager is

also responsible to reflect the changes made on the application on the architectural level. This operation ensures the matching between the executed application and the description of this application.

2. *The Environment Agents E-A*: The role is the control of the execution environment and the notification of the A-A if significant any changes appears in the environment variables (Belabed and Chouarfia, 2008).

3.3 Adaptation principles

The idea consists to associate for each component a connector, which implements its required interfaces. The inter components calls are done through these connectors.

The adaptation mechanism acts directly on connectors to achieve the adaptation (Fig 3). Each connector implements the necessaries mechanisms (for calls interception and redirection) that allow the adaptation agent to perform the update.

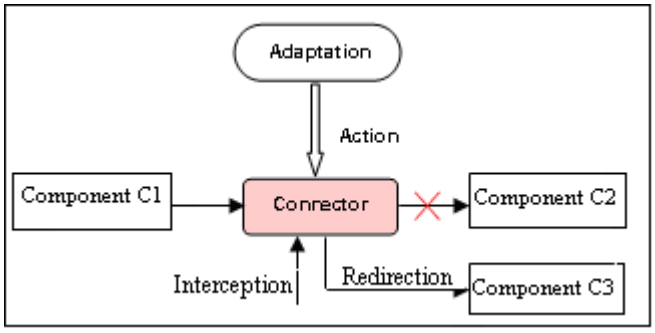

Fig 3: Adaptation principle.

Using Fig 3, we explain the structure of connectors in our approach. In the example, we assume that C1 component interacts with C2 component trough a connector in synchronous communication mode.

The C1 component requires interface I2, which the specification is:

Interface I1 {

Result1 M1 (param1.1, param1.2);

}

While "Result1" is the return of method M1, param1.1 and param1.2 are the input parameters.

The C2 component provides the interface I2 which specification is:

Interface I2 {

Result2 M2 (param2.1, param2.2);

}

The basic structure for the connector is as follows:

Connector implement I1 {
Result1 M1 (param1.1, param1.2)
{
Return execute ('Script');
}
}

The script code

```
// type casting of parameters call
Param31= castingToPramr2.1(param1.1) ;
Param32= castingToPramr2.2 (param1.2) ;
// call of C2 component's M2 method
Result3 = C2.M2 (Param3.1, Param3.2);
// Result type casting
Return cast_to_Resul (Resul3);
```

Listing 1: Script code

The body of method M1 of connector is implemented with a script that makes the call to method M2 of C2 component, with the necessary parameters and return types casting. The replacement of the C2 component is made by changing the executed script in the body of connector's method (M1 in the example).

The system is designed to support the addition, removal, replacement and migration (change of the deployment server) of components. The A-A operates according to an adaptation plan, this is a set of basic algorithms specific to each operation (ex: addition or removal operation).

Before any operation, the components directly implied in the adaptation operation must to be in a passive state (they not accept incoming calls), this is possible thanks to the connectors that can queue calls to a C component during adaptation.

A dynamic connection between two components C1 and C2 is to put in interaction between these two components through two methods (ports) M1 and M2. M1 is a required method for C1 and M2 is a provided method by C2, the connection is done by the A-A while indicating to the connector associated with C1 the name of the C2 component and the method to be called M2. A phase of mapping between the two methods is necessary before establishing connection. The correspondence concerns the parameters of call, their order and the type of return of each method. The mapping is done manually by the administrator of the application using the architecture description. The administrator must provide the methods of calculation (script code) for conversion between the types of the parameters of call and the type of return of each method. The methods are transmitted then to the connector responsible for the connection between the two components. To make conversions in an automatic way after connection, the administrator must also manage the semantic correspondence between the methods to be connected to avoid the semantic abuses use.

The dynamic disconnection of a component consists in making it passive, i.e. prohibit any incoming call towards this component.

3.3.1 Component Addition

Add a new component to an application corresponds to connecting each component port (interfaces operations) with the corresponding port of the application components. So an addition is a succession of connections. The addition is done as follow:

```
Add (component C) {

For each port m of c {

- Identify all component Ci and ports mi
to be connected with the port m of
component C;

- For each couple (Ci, mi) {

   make the correspondences with the
couple (C,m) ;

If not correspondence Then cancel
addition; }}

- establish the add at the architectural
level;

- verify the adaptation coherence after
addition (in architectural level)

If not coherence

     Then {cancel addition;

           Cancel the modifications on
architectural level;   }
```

Listing 2: Component addition code

3.3.2 Component Removal

A component is removed only if it does not refer to any component and no component refers to him, this condition can be verified at the architecture level when the component is not implied in any interaction rule. The suppression algorithm is defined as follow:

```
remove (component C) {
Check the absence of reference to or
from C;
      If OK Then remove C;
      else impossible to remove C; }
```

Listing 3: Component removal code

3.3.3 Component Replacement

Fig 4 shows the replacement of the *C*3 component by the *NewC3* component. In the example, the C3 component is in relation with two components *C1* and *C2* through provided interfaces and in relation with *C4* through a required interface. The adaptation consists then to replace the *C3* component by the *NewC3* component. Before starting the replacement operation the two components must be passed in correspondence phase.

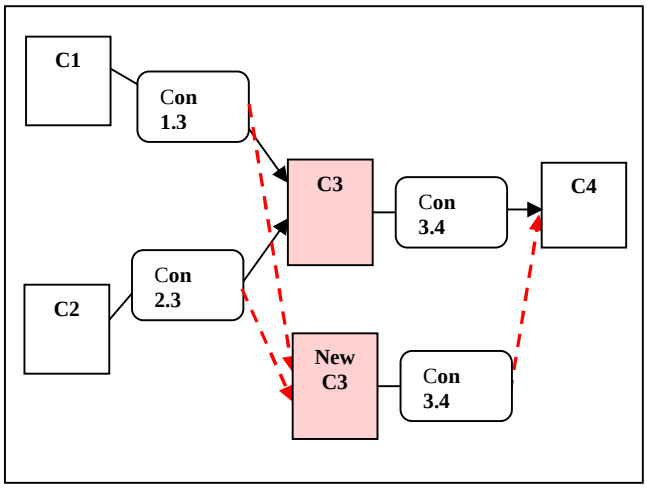

Fig 4: Component replacement.

After corresponding phase, the adaptation agent achieve the replacement according to the following adaptation plan:

1. Referring to the architecture description, the adaptation agent localises all connectors in relation (in provided interfaces) with the component to be replacing (« *con 1.3* » and « *con 2.3* »).
2. The adaptation agent puts *C3* in passive state by ordering to each localised connector to blocking all messages towards it.
3. The adaptation agent deploys the new component.
4. The adaptation agent sends the necessaries information to each connector to make the redirection of the calls towards the new component.
5. The adaptation agent connects the new component on provided interfaces with the *C4* component, this operation implies the deployment of the connector associated to the *NewC3* component.
6. If the component to be replaced is with state, the adaptation agent makes a state transfer between the two components.
7. After a time *t* corresponding to the maximum of the response times of the component to be replacing, the adaptation agent deactivates this last, activates the new component by unblocking the blocked messages and announces the end of the adaptation. This mechanism is used to make sure that the *C3* component has finished all in progress treatments before its suppression.

3.3.4 Component Migration

The component migration consists in moving it from an application server *S* to another server *NewS*. The new server must provide an ideal execution environment for the moved component, i.e. it must have all the resources whose component needs. The migration of a *C* component towards a *NewS* is made by the A-A according to the following algorithm:

```
Migrate (component C, server NewS) {
-  deploy a copy of C on the NewS;
- locate all connectors referring C;
- make C in passive state (no incoming
calls);
- send the address of the NewS to the
localised connectors to locate the
migrated component;
- If C is with state Then {
          - Waits a t time >= max
(response time of C);
          - make a state transfer
between C and its copy;
                        }
- enable all messages blocked on each
connector;
- remove C from the old server S;
}
```

Listing 4: Migration component code

3.4 Adaptation coherence

The adaptation coherence is ensured at the architectural level. Before establishing an adaptation at the applicative level, the architectural level checks the applicative level. The checking of coherence is done through a set of rules incorporated in the architecture description base. Any adaptation considered incoherent in this level is cancelled. The following section describes the solutions used in our approach to ensure a coherence and adaptation safety.

1. *Preserving communication channels***:** the communication channels are preserved through a mechanism implemented in each connector, this mechanism blocks the messages (put them in queue) during adaptation. The end of adaptation unblocks all messages on standby, and the application continues its execution without messages lost.
2. *Coherence of the interactions:* the coherence of the interactions between components is ensured through a manual phase of mapping between ports of components of the components to connect. This phase is performed using information provided for each port in architecture description base. This information allows to taking into account the semantics of use of each port and component, which avoids any conflict of call or use.
3. *Conflicts between adaptations:* if badly managed can lead to a total crash of application, in the fact that an adaptation can cancel the effect of another or introduce contradictory modifications. This type of inconsistency is avoided in our approach by allowing one adaptation at the same time, this can lead to degradation in the performances of the adaptation system, but it is a cost to be paid with the profit of a safe adaptation. If an adaptation is in progress and a new need for adaptation appears, this second is put in a queue until the end of the first adaptation. The adaptation queue is managed so as to eliminate a contradictory adaptations, for example, if an adaptation need due to an increase in one of environment parameters is in the queue, if another adaptation is needed with the reduction of the same parameter, the first adaptation is eliminated from the queue.
4. *The state transfer:* we are not introduce a specific solution in our approach, we adopt the solution proposed by (Ketfi, 2004), this solution is specific to the components written in java, it consists to pass each component before its deployment by an instrumentation phase using a byte codes manipulation tools such as Javassist [Chiba and Nishizawa, 2003) or BECEL (Dahm, 1999) . In this phase we introspect the component implementation then selecting the attributes which constitute the state of component, we add than the operations *getState()* and *setState()* in the component implementation. This mechanism increases the complexity of the operation of adaptation, for example, in component replacement it is also necessary to make a correspondence between the two components to see whether it is possible to make a state transfer, if the mapping fails, the replacement will be then impossible.

4. IMPLEMENTATION AND EVALUATION

A preliminary implementation of the system is done for the EJB (Enterprise Java Bean: Sun Micro Systems) component model. For implementing the interception and redirection mechanisms in the connectors, we have used the *reflexives* properties of Java language and the *Java scripting* programming technique (Bosanac, 2007). This technique consists to execute a script code in a java class. The script origin can be a file or other application. This technique enables to the adaptation agent to change connector's code dynamically. The script language used in the implementation is Groovy (Bordeie, (2007).

To evaluate the proposed system, the test application runs on a PC with PIV 1.7MHz, 256M SDRAM. It consists of a component client sending a string and component server receiving this string, printing it in screen and returning it back to the client.

The evaluation test is made by comparing two versions of the same application; one implements the adaptation mechanism on its connectors, the other without this mechanism.

First we have tested the adaptation mechanism influence on the application response time. The objective is to calculate the response time increase in the version which implements the adaptation mechanism according to the number of requests emitted by the client. The following results are obtained:

Table 1: Response time

Requests Numbers	Response time average : without adaptation mechanism	Response time average : with the adaptation mechanism	Increases rates
10	108,71 ms	138,80 ms	**27.67 %**
30	339,27 ms	407.71 ms	**20,17%**
50	526.70 ms	614.36 ms	**16.64 %**
100	1129,05 ms	1315.55 ms	**16.51 %**
200	2154.55 ms	2506.02 ms	**16.31 %**
300	3250.64 ms	3777.05 ms	**16.19 %**
500	5417.43 ms	6292.61 ms	**16.15%**
1000	11176.91 ms	13007.83 ms	**16.38 %**

According to the results, we can notice that the influence of the implementation of the adaptation mechanism is at least stable; it induces a response time increase of about 16 %. This increase is natural results of the use of an interpreted code on connectors instead a compiled code which is faster in term of execution time. We cannot say that is acceptable or not, that depends on the nature of application if it aims the dynamic adaptability more than performance or not. Thus there is always a compromise to make between the benefit in term of dynamic adaptability and the loss in term of performance.

The second evaluation has for objective the measurements of the adaptation duration, which is also the inactivity time, since the communications channels are blocked during adaptation. The adaptation duration is calculated as follow:

$\mathbf{T}_{adaptation} = \mathbf{T}_{rspt\text{-}wa} - \mathbf{T}_{rspt\text{-}na}$; with:

$\mathbf{T}_{adaptation}$ **:** adaptation duration.
$\mathbf{T}_{rspt\text{-}wa}$**:** response time including adaptation.
$\mathbf{T}_{rspt\text{-}na}$**:** response time without adaptation.

Table 2: Response time

Requests Numbers	Response time without adaptation	Response time with adaptation	Adaptation duration
500	7375.2 ms	10780,66 ms	**3405,46 ms**
600	8551,33 ms	11685,16 ms	**3133.83 ms**
700	9815,4 ms	13053.2 ms	**3237.8 ms**
800	11051 ms	14589.75 ms	**3538.75 ms**
Average			**3328.96 ms**

According to the results shown in table 2, we observe that the adaptation duration average is greater than 3 seconds. This time is very large compared to the response time of only one request which is approximately 14 ms (response time/a number of requests). This value presents an unacceptable inactivity rate, especially if we manipulate a highly critical or real time application. It's important to note that this experimentation is made without including the state transfer mechanism, which is not yet implemented in the system. Such mechanism increases more the adaptation duration.

The last experimentation aims to evaluate the ability of the system to preserve the communication channels during adaptations. For this reason, we have implemented on the client component a mechanism that enables to counting the number of the right responses obtained from the server component. We repeat the previous experience, which consists to make adaptation during requests execution.

Table 3: Altered messages rate.

Requests	Altered messages
500	**0%**
600	**0%**
700	**0%**
800	**0%**

The obtained results show that the rate of altered messages during an adaptation is 0%, thanks to the calls blocking/unblocking mechanism implemented on the connectors.

According to the previous results, we can say that the proposed approach is well adapted for the applications that haven't a frequent adaptation rate and which prefer performances losses to data loss.

5. CONCLUSION

In the paper we have presented an architecture based approach for dynamic adaptation in component-based software. The major advantage of the proposed system is the separation of the adaptation mechanism from the executed application. This makes the system independent from particular component models and platforms. The implementation for a specific component model (Enterprise Java Bean, CCM (Corba Component Model: OMG) or Dot Net: Microsoft) is made through little conveniences (in connectors' level) without changes in the main adaptation concepts. More advantages are the well communications channels preserving and the adaptation reliability.

However the evaluation of the proposed solution has revealed some limits, for example the long adaptation duration makes our solution adapted only for special kinds of applications. Others limits like the non support of simultaneous adaptations, leads us to plan the following prospects:

- The improvement of the adaptation system in terms of performances;
- To extend the MAS by introducing for example several adaptation agents and ensuring their collaboration to achieve more than adaptation at the same time.

In long term, we plan to continue our experiments with others components models like CCM and DotNet, and study the possibilities to extend our adaptation solution to be supported by other kinds of applications like web services. Also others aspects such as what kind of system were this applied to? What did the architecture look like before adaptation? What extent did the adaptation reach? will be developed and evaluated.

REFERENCES

Aksit, M. and Choukair, Z. (2003). Dynamic, adaptive and reconfigurable systems overview and prospective vision. ICDCSW'03 0-7695-1921-0/03 IEEE Computer Society.

Belabed, A. and Chouarfia. A. (2008). Une approche orientée aspect pour l'adaptation dynamique des applications à base de composants. COSI'08. Tizi-Ouzou, Algérie.

Brinkschulte, U., Schneider, E. and Picioroag, F. (2005). Dynamic real-time reconfiguration in distributed systems: Timing issues and solutions. Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. (ISORC'05), 0-7695-2356-0/05.

Brinkschulte, U., Bechina, A., Picioroag, F. and Schneider, E. (2002). Distributed real-time computing for microcontrollers – The OSA+ Approach . ISORC, Washington D.C.

Bordeie, X. (2007). Aborder Groovy, langage de script pour Java, JDN Développeurs.

Boanac, D. (2007). Scripting in Java languages, Frameworks and patterns. Addison Wesley, ISBN 0321-32193-6.

Chiba, S. and Nishizawa, M. (2003). An easy to use toolkit for efficient Java bytecode translators. Proc of 2nd International Conference on Generative Programming and Component Engineering (GPCE'03), LNCS 2830, pp 364-376, Springer Verlag.

Dahm, M. (1999). Byte code engineering with the JavaClass API. Technical Report B-17-98, Institut fur Informatik, Freie Universitat Berlin January.

Dashofy, E.M., Hoek, A.v.d., and Richard, N.T. (2001). A highly-extensible, XML-based architecture description language. In Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001). Amsterdam.

Georgas, J.C., Richard N. T. (2004). Towards a knowledge-based approach to architectural adaptation management. *WOSS'04*, Newport Beach, CA, USA.

Huang, G., Mei, H., and Yang, F. (2006). Runtime recovery and manipulation of software architecture of component-based systems. Automated Software Eng. pp 257-281.

Kell, S. (2007). Rethinking software connectors. SYANCO'07 Dubrovnik, Croatia.

Ketfi, A. (2004). Une approche générique pour la reconfiguration dynamique des applications à base de composants logiciels. Thèse de doctorat de l'Université Joseph Fourier de Grenoble France.

Kiczales, G., Lamping, J. and Mendhekar, A. (1997). Aspect-oriented programming. In Proceedings of the ECOOP'97.

Medidovic, N. and Taylor, R.N. (2000). A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering, Vol. 26, N°.

Oreizy, P., Medidovic, N., Taylor, R.N. (1998). Architecture-based runtime software evolution. ICSE '98. Kyoto, Japan.

Pinto, M., Fuentes, L. and Troya, J.M. (2003). DAOP-ADL: An architecture description language for dynamic component and aspect-based development. GPCE 2003, LNCS 2830. (C) Springer-Verlag Berlin Heidelberg.

Plasil, F., Balek, D., Janecek, R. (1997). DCUP: Dynamic component updating in Java/CORBA Environment. Tech. Report No. 97/10, Dep. Of SW Engineering, Charles University.

Qun Yang, Xianchun Yang and Manwu Xu (2006) A mobile agent approach to dynamic architecture-based software adaptation. ACM SIGSOFT Software Engineering Notes, Vol. 31 N°. 3.

Samuel, K., Obrst, L., Stoutenburg, S. and Fox, K. (2006). Applying Prolog to semantic Web Ontologies & Rules moving toward description logic programs. ALPW.

Szyperski, C. (1998). Component software: beyond object oriented programming. Editor Addison- Wesley & ACM Press.