A Model-Based Framework for Building Self-Adaptive Distributed Software

Ouanes Aissaoui[1], Abdelkrim Amirat[2] and Fadila Atil[1]
[1]LISCO Laboratory, Badji Mokhtar-Annaba University, P.O. Box 12, 23000 Annaba, Algeria
E-mail: aissaoui.ouenes@gmail.com, atil_fadila@yahoo.fr
[2]LiM Laboratory, University of Souk-Ahras, P.O. Box 1553, 41000 Souk-Ahras, Algeria
E-mail: abdelkrim.amirat@yahoo.com

Keywords: self-adaptive system, component-based system, dynamic reconfiguration, distributed reconfiguration, reliable reconfiguration

Received: August 16, 2013

Reconfiguration is widely used for evolving and adapting systems that cannot be shut down for update. However, in distributed systems, supporting reconfiguration is a challenging task since a reconfiguration consists of distributed reconfiguration actions that need to be coordinated and the application consistency must be preserved. To address this challenge, we propose a framework based on a reflexive three layer architecture model for the development of distributed dynamic and reliable component-based applications. The bottom layer of this model is the application layer. It contains the system's application-level functionality. The change management layer is the middle layer. It reacts to changes in state reported from the application layer. The uppermost layer is the self-adaptation layer that introduces the self-adaptation capabilities to the framework itself. It ensures the service continuity of the change management layer and manages the adaptation of this last to the changes which it carries out itself on the application layer. The framework is conceived especially for supporting the distributed reconfigurations. For that, it incorporates a negotiation and coordination mechanism for managing this type of reconfiguration. Moreover, it incorporates a separate system for ensuring the reliability of the application. The paper introduces a prototype implementation of the proposed framework and its empirical evaluation.

Povzetek: Članek predstavlja okolje za gradnjo samo-prilagodljivega porazdeljenega programja.

1 Introduction

Nowadays, more and more of distributed applications run more often in fluctuating environments such as mobile environments, clusters of machines and grids of processors. However, they must continue to run regardless of the conditions and provide high quality services. A solution to this problem is to provide mechanisms allowing the evolution or the change of an application during its running without stopping it [1, 16]. So, we talk about the dynamic adaptation of distributed applications which can be defined as the whole of the changes brought to a distributed application during its running [26].

Software reconfiguration [10] is strongly related to the domain of runtime software evolution and adaptation. In this domain, reconfiguration is used as a means for evolving and adapting software systems that cannot be shut down for update. Reconfiguration actions include component additions and removals; setting the parameter's value of a component; interfaces connections and disconnections; changes of the component state (started or stopped) and additions of new behaviours to component.

Several conditions must be checked by an adaptation operation where the most significant is the application consistency which can be summarized by the following points [19]:

- *Safety*: an adaptation operation badly made should not lead the adapted application to a crash.
- *Completeness*: At the end of a certain time, the adaptation must finish, and must at least introduce the changes necessary to the old version of the application.
- *Well-timedness*: it is necessary to launch the adaptation at the right time. The programmer must specify in advance the adaptation points.
- *Possibility of rollback*: even if we show the correctness of the adaptation, certain errors can escape from the rule. It is necessary to have some means that allow to cancel the adaptation and to roll back the application to its state known before the execution of the adaptation.

Therefore, the preservation of the application consistency is a very significant parameter to evaluate an approach for the dynamic adaptation.

Generally, the existing self-adaptive literature and research which has studied the dynamic adaptation of the distributed software systems provide solutions for adapting such systems, but the adaptation is not distributed (e.g. [29, 30, 31, 21, 9, 34]). In particular, the distribution of the adaptation system itself is rarely

considered. Also, parallel to the need of the dynamic adaptation of applications pose the problem of their reliabilities, which is an important attribute of the functioning safety [6]. In spite of the importance of the application reliability in the adaptation of applications, it was not taken into account in many works [31, 10, 9] particularly those treating the distributed reconfigurations. The works which studied this property in the dynamic adaptation did not reach the required level of coherence; it is only simple mechanisms generally based on the backward recovery technique (e.g. [20, 26]) which consists to roll back the application to a previously consistent state. Also, these mechanisms are incorporated in the components managing the adaptation of the application. So, the code responsible for the adaptation of the application is weaved with that which makes it reliable. Notice that this crosscutting of code prevents the evolution of the two mechanisms managing the reliability and adaptability.

After having identified this problem, we have concentrated on the reliable adaptation of the distributed component-based applications, which is a very topical. Our first objective is to provide a solution for the management of the distributed and coordinated dynamic adaptation. The second objective is to provide a separate solution for managing the fault tolerance of these applications in order to ensure their reliability which helps to lead to reliable reconfigurations, and the third objective is to facilitate the construction of this type of application studied by minimizing the time and the cost of the addition of the self-adaptation capabilities to it.

To achieve the first two objectives, we propose a reflexive three layer architecture model for the development of distributed dynamic and reliable applications. The bottom layer of this architecture model is the application layer which represents the software system. The change management layer is the middle layer. It reacts to changes in state reported from the application layer. The uppermost layer is the self-adaptation layer that manages the adaptation of the change management layer and ensures its service continuity.

In order to minimize the time and the cost of the addition of the self-adaptation capabilities to this type of software studied (distributed and dynamic) we propose a framework based on the proposed architecture model. This framework implements the two uppermost layers of the architecture model. As we deal in this work the distributed applications, we propose that each site must contain two parts; the first represents a sub-system of the application, i.e. components implementing the application's business logic whereas the second represents the proposed framework that controls and manages the adaptation of the first part. Notice that, the management of the adaptation is distributed. This decentralization guarantees the desired degree of fault tolerance required in certain situations.

The remainder of the paper is organized as follows. Section 2 presents the proposed three layer architecture model for building the self-adaptive systems. Section 3 details the design of the proposed framework according to the proposed architecture model. In Section 4, we give the implementation details for a prototype of our framework and we illustrate the validation plan. Section 5 analyses the related proposals found in the literature. Finally, Section 6 concludes the paper.

2 Overview of the proposed architecture model

In this work we propose firstly a three layer architecture model that is used to guide the development of the dynamic and reliable distributed software. Figure 1 summarizes this model.

2.1 Application layer

The bottom layer of the proposed model is the application layer. It consists of a set of components implementing the application's business logic. As we deal the distributed applications these components are distributed on several sites. We propose that each functional component must have a component of type «*ComponentController*» which controls it. This last plays two roles: (1) if the controlled component is active, the «*ComponentController*» intercepts and redirects the incoming calls of service (to the controlled component) to the component «*ApplicationController*» of the fault-tolerant system (see section 3.2). In the contrary case where the controlled component is in a reconfigurable state, i.e. at the time of adaptation, its controller intercepts and saves the incoming calls of service to it in a queue until the end of the launched adaptation operation.

2.2 Change management layer

The middle layer of the proposed architecture model is the change management layer. This layer reacts to changes in state reported from the application layer. For that, it consists of two separate systems; the first is the fault-tolerant system which manages the reliability of the application and the second is the adaptation system which reconfigures dynamically the application. We will present these two systems in detail in the next sections. This separation of the fault-tolerant system from the functional code of the application and the code charged to reconfigure it facilitates the evolution of the reliability mechanism and thus, the development to the developers or integrators of the application which will concentrate on the functional code of the application rather on the non-functional code charged to reconfigure it and make it reliable.

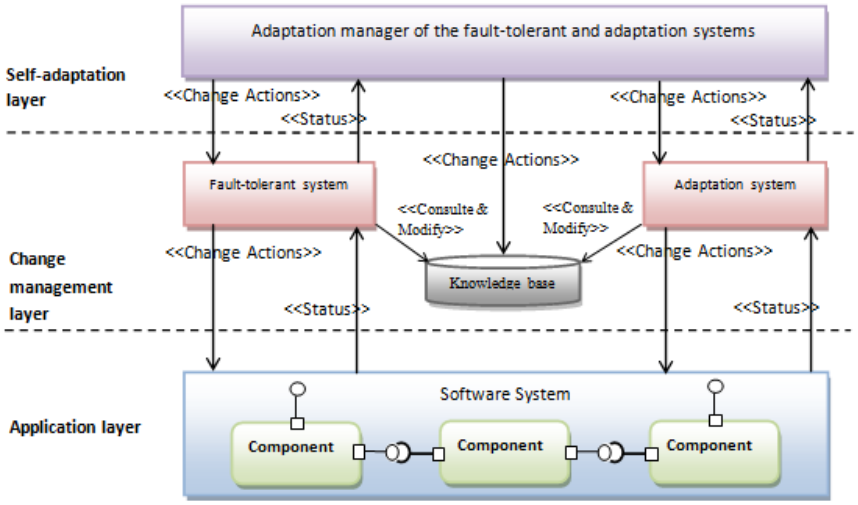

Figure 1: A three layer architecture model for self-adaptation.

2.3 Self-adaptation layer

The uppermost layer of the proposed architecture model is the self-adaptation layer. This layer introduces the self-adaptation capabilities to the framework itself. It controls and manages the change management layer for ensuring its service continuity and adapting its components to the changes that they carry out on the application in order to guarantee its correct operation and also its service continuity because certain changes in the application can lead to the appearance of faults in the execution of the system that manages these changes. For example, an operation of removal of a component in the application leads to the appearance of errors in the change management layer if the non-functional components managing the removed component have not adapted to this change.

Notice that, the proposed architecture is reflexive; the middle layer manages the bottom layer and the uppermost layer manages the middle layer. Also, this decomposition in three layers imposes a clear separation of concerns and facilitates the adaptation management as well as the evolution of the two mechanisms of fault tolerance and adaptation.

In order to facilitate the use of our architectural model we propose a framework implementing the two uppermost layers (self-adaptation and change management layers). So, the framework contains the two systems of adaptation and fault tolerance as well as the manager of these first two systems and which implements the self-adaptation layer. Therefore, an application developed according to our architecture model is made up of a set of functional and non functional components distributed on several sites. At each site we must find a sub-system (level of the application layer) which is a set of functional components representing the application's business logic plus an instance of the proposed framework, which is the responsible for the management of the application context (collection of data, analyses…) and the management of its change. So, the framework represents the hot subject of this paper. Figure 2 shows an overview of our solution for managing the distribution of the adaptation. For reasons of clearness, only two sites are represented.

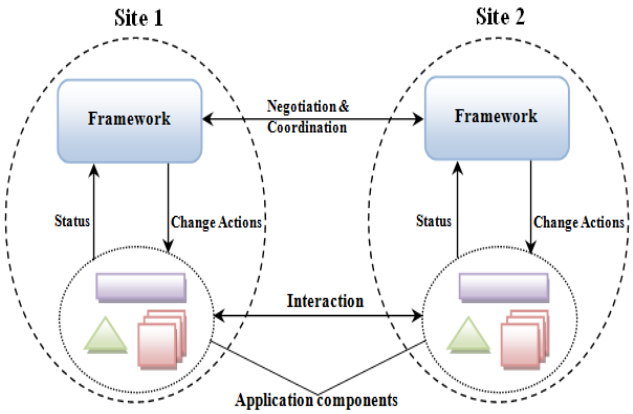

Figure 2: Overview of our solution for the management of distributed reconfigurations.

This organization makes the architecture of the self-adaptive applications developed according to our approach decentralized what avoids the problems of the centralized approaches [11].

In the next sections, we will present in detail the structure and the functioning of the different components of the two uppermost layers in the architecture model through the proposed framework.

3 Design and functioning of the proposed framework

This section describes in detail the various elements of the proposed framework thus that their functioning in order to perform the dynamic adaptation and preserve the consistency of the application. We present these elements according to their order of dependence.

3.1 Knowledge base

The knowledge base is a very important element in our framework since it plays a very significant role to provide reliable dynamic reconfigurations. For that, it is used by the different elements of the framework. It consists of three parts: (1) the description of software architecture, (2) the description of the adaptation policy and (3) the coherence rules. We propose the use of the logic of predicates with the language Prolog [32] for the description of these parts. This choice is justified by:

- Prolog is a language of knowledge representation.
- Prolog can be easily used for the description of the software architecture. We can write an XML tag (<tag>value</tag>) in Prolog as a fact as follows: tag (value).
- The representation of invariants (for the verification of the application consistency) by inference rules eliminates the programming of verification mechanisms of these invariants because this verification is performed by the inference engine of Prolog.
- The existence of Prolog interpreters developed in several languages, which facilitates the use of the prolog formalism.

3.1.1 Description of the software architecture

The description of the software architecture must contain:

- The detailed description of each application component.
- The specification of the component assembly.

3.1.1.1 Component description

component (‘id’, ‘name’).
component_state(‘comp_id’, ‘state’).

State: may be active or quiescent.

component_location(‘comp_id’, ‘ip_site’**).**
required_interface (‘comp_id’, ‘interface_id’).
provided_interface (‘comp_id’, ‘interface_id’).
interface (‘interface_id’, ‘name’).
include_operation (‘interface_id’,’operation _id’).
operation (‘id’, ‘name’, ‘list_param’, ’return_type’).
param (‘operation_id’, ‘name’, ‘type’, ‘value’).
component_property (‘comp_id’, ‘name’, ’value’).

3.1.1.2 Interaction between components

interaction (‘comp_id1’, ‘comp_id2’, ‘oper_id1’, ‘oper_id2’).

The ***interaction*** predicate specifies that the component *"comp_id1"* interacts with the component *"comp_id2"* where the operation *"oper_id1"* is required by the component *"comp_id1"* and the operation *"oper_id2"* is provided by the component *"comp_id2"*.

3.1.2 Application consistency

Parallel to the need of the dynamic reconfiguration of applications pose the problem of their reliabilities which is an important attribute of the functioning safety [6]. In fact, the modifications in a system can leave it in an incoherent state and thus challenge its reliable character. In order to guarantee the reliability of the system following a dynamic reconfiguration, we define the application consistency as the satisfaction of a set of constraints. These constraints are related to the definition of the architectural elements and their assembly and also to the state of the components.

We have used Prolog as a constraint language. So, we use the inference rules to express these constraints:

Example 1: Here is a rule to check if there are two components that have the same identifier:

haveSameID (Comp_name1, Comp_name2):-
Comp_name1 != Comp_name2,
component (Comp_id1, Comp_name1),
component (Comp_id2, Comp_name2),
Comp_id1=Comp_id2.

Notice that, the constraints vary from a component model to another and from an architectural style to another, for example there are models which authorizes the hierarchical structure and others not. The evaluation of these rules is made by the Prolog inference engine. The trigger of the evaluation of these rules is carried out by the two sub-components «*BehaviourChecking*» and «*StructureChecking*» of the component «*VerificationManager*» of the fault tolerant system (see section 3.2). Notice that, an operation of reconfiguration is valid only if the reconfigured system is consistent, i.e. if all the constraints in the knowledge base are satisfied.

3.1.3 Adaptation policy

One fundamental aspect in the software adaptation is the definition of the adaptation policy, i.e., the set of rules which guide the trigger of the adaptation according to the changes of the environment of the application and its components. These rules are in the form ECA, i.e. If (<Event> and <Condition>) then <Action>. The event part specifies the context change that triggers the invocation of the rules. The condition part tests if the context change is satisfied which causes the description of the adaptation (action) to be carried out.

We also propose the use of the inference rules to express the adaptation policy.

Example 2: Assume we have a software component that manages a cache memory. For this, it owns a property “*maxCache*” representing the maximum permitted memory space to save data into memory for faster processing. The following lines show an adaptation policy (described in Prolog) for a possible adaptation of this component.

rule1(Z):- **free_memory**(X), X> **2000**,
component_ property(*‘cacheHandler*, *’maxCache’*, Value),
Value<**10**, Z is "strategy1".

rule2(Z):- **free_memory**(X), X<**1000**,
component _property (*‘cacheHandler’*, *’maxCache’*, Value),
Value>**10**, Z is "strategy2".

strategy (‘strategy1’, “ [localhost] set_Value(‘*cacheHandler’*, *‘maxCache’*, 20) ”).

strategy (‘strategy2’, “ [localhost] set_Value(‘*cacheHandler*’, ‘*maxCache*’,5) ”).

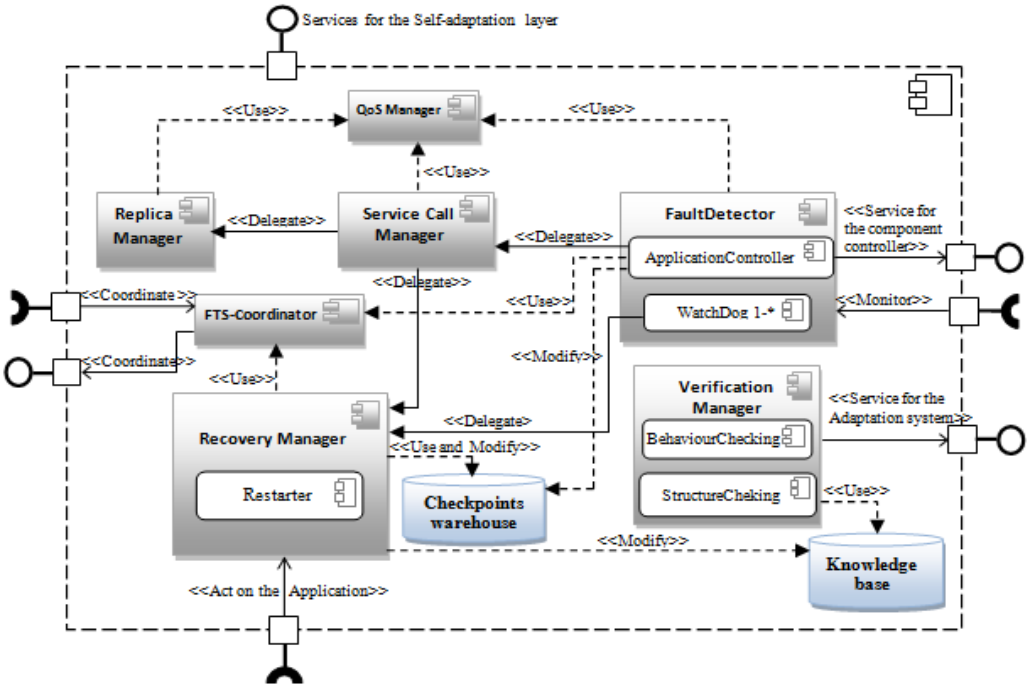

Figure 3: Overview of the fault-tolerant system.

The first rule is triggered only if the memory available exceeds 2Go and the maximum value of the cache is less than 10MB. In this case, the rule returns the string *'strategy1'* which indicates that it is necessary to apply the adaptation strategy number 1. This strategy contains in its action plan a single reconfiguration action which involves increasing the cache value to 20MB. Note that this operation concerns only the local site “*Localhost*”. The second rule is the reverse of the first. It involves decreasing the cache value to 5MB if the memory available is less than 1Go and the max cache value exceed 10MB.

3.2 Fault-tolerant system

For the definition of the fault-tolerant system, we consider a set of constraints which are: (i) modularity and adaptability of the system, (ii) extensibility of the system, (iii) taking into account of the distributed nature of the application to make it reliable as we deal here the distributed software systems.

This system ensures the application service continuity which helps to lead to reliable reconfigurations. We think that this system is very important in the self-adaptive applications because an adaptation operation cannot be executed on a component if it is crashed or in an inconsistent state. Also, the preservation of the application consistency is an important condition in the adaptation of software systems as mentioned in the introduction. We have separated this system to the adaptation system and the application’s business logic in order to integrate more than one fault-tolerance technique for ensuring the application consistency and to facilitate the evolution of this system without influencing either the adaptation system or the application’s business logic.

In order that this system achieves its goal, it contains a component for the management of the service quality, a fault detection component, a recovery component, a component for the verification of the application consistency, a component for the management of the replicas of the functional components, a component for the execution of the call of service plus a component for the coordination of distributed checkpointing and distributed recovery. Figure 3 shows an overview of the fault-tolerant system.

3.2.1 Techniques used in the fault-tolerant system

The proposed fault-tolerant system is based on the following techniques: distributed checkpointing, active replication, distributed backward recovery and message store. Our objective is to use these techniques for providing a fault-tolerant system able to tolerate many types of the software faults.

3.2.1.1 Distributed checkpointing

A common method for ensuring the progress of a long-running application is to checkpoint its state periodically on stable storage [23]. The application can be rolled back and restarted from its last checkpoint which bounds the amount of lost work that must be recomputed [23]. As we deal in this work the distributed applications, the coordination for the distributed checkpointing is a very important operation. In a coordinated checkpointing, processes coordinate their checkpointing activity so that a globally consistent set of checkpoints is always

maintained in the system. For that, we have used in our fault-tolerant system the two-phase commit distributed checkpointing protocol presented in [23].

The algorithm of this protocol is composed of two phases. The next paragraph describes the mapping of this algorithm to the fault-tolerant system of the proposed framework.

The running of this algorithm starts if a service request is launched by a functional component of the application. In this case, the controller of this component intercepts this call and delegates the execution to the sub-component «*ApplicationController*» of the component «*FaultDetector*» in the fault-tolerant system (see figure 3). This last asks the coordinator of distributed checkpointing (sub-component of the component coordinator) to launch a coordination so necessary for saving a checkpoint. Notice that, the checkpointing is performed periodically around an interval of time indicated by the component «*QoS-Manager*». So, if the time is passed the application controller asks the coordinator for the checkpointing to start coordination for checkpointing. In this case, the coordinator according to his policy decides if the safeguard of a checkpoint requires coordination or not. If the two components (client and server) depend on other components installed on other sites the coordination process starts.

In the first phase, the coordinator identifies initially the participants (components installed on other sites and depend on one of the two components client and server) of this coordination operation by using the predicate "*interaction*" presented above. For that, the coordinator asks the question "***? interaction (Cp, 'C_id', _, _).***" for the two components server and client such as 'C_id1' must indicate the identifier of the component concerned of this question, i.e. the client or the server identifier. After, the coordinator broadcasts a checkpoint request message to all participants. Every participant, upon receiving this message, stops its execution, flushes its communication channels, takes a tentative checkpoint and replies "*yes*" to the coordinator, and awaits the coordinator's decision. If a participant rejects the request for any reasons, it replies "*No*". If all participants reply positively, the coordinator's decision is to commit the checkpoints. Otherwise, its decision is the cancellation of the checkpoints. The coordinator's final decision marks the end of the first phase. Note that, the waiting time of the reception of the participants' response by the coordinator is fixed. If the coordinator does not receive a response of a participant for this period of waiting it regards it as "*No*".

In phase II, the coordinator sends its decision to all participants. If its decision is "*Save the checkpoints*," every participant removes its old permanent checkpoint and makes the tentative checkpoint permanent. Otherwise, participants reject the tentative checkpoint previously taken. Finally, each participant resumes its execution. The table 1 presents an overview of this algorithm.

Table 1: Overview of the distributed checkpointing algorithm.

Coordinator	Participants

Begin
If (the coordination for checkpointing is necessary)
Begin
/* Begin Phase I */
determine the participants;
request participants to take tentative checkpoints;
await all replies;
if (all replies = "*Yes*")
decide←"*Save the tentative checkpoints*";
else
decide ←"*Remove the tentative checkpoints*";
/* Begin Phase II */
send the decision to all participants;
end-if
End.
	Begin
	/* Begin Phase I */
	receive the coordinator request ;
	if (accept request)
	begin
	suspend communication;
	take a tentative checkpoint;
	reply *"Yes"* ;
	end-if
	else reply *"No"* ;
	await the coordinator decision ;
	/* Begin Phase II */
	if (decision = *"Save the tentative checkpoints")*
	begin
	remove the old permanent checkpoint;
	make the tentative checkpoint permanent;
	end-if
	else discard the tentative checkpoint ;
	resume communication ;
	End.

3.2.1.2 Active replication

The highly available services can be achieved by replicating the server components and thereby introducing redundancy [24]. If one server fails, the service is still available since there are other servers that are able to process the incoming requests. The active replication also called the state machine approach is one of the techniques allowing achieving such software-based redundancy [24].

In the active replication technique, clients send request to all the servers and it receives the common response to all servers. So, all servers execute all requests and end up in the same final state. Thus, at any given time it is likely that there is at least one server that can accept and process the incoming requests. In the active replication the crash of any server is transparent to the client [24]. We have used this technique in order to tolerate the faults in value in the application.

In the replication technique the components duplicated are generally those that are the more used in the application and these components are generally subject of the dynamic adaptation. So, preserving the continuity of service of these components is a very important task.

The replication has as a consequence a faster recovery of the failed components because the replicas are active and ready to process the incoming requests [28]. We have implemented this technique in the

component *«ReplicasManager»* of the fault-tolerant system (see section 3.2.2).

When the replication technique does not guarantee the masking of faults for the reason of the software crash (for example a problem into a component requires the search of a coherent state to continue processing) or for the reason of hardware problems, the recovery will be the best solution [28].

3.2.1.3 Backward recovery

The backward recovery consists to roll back the application in the case of failure to a previously saved state in order that it continues processing normally [22]. For that, a set of checkpoints must be saved each time that it is necessary. One problem with this technique is that the recursive execution of the backward-recovery process on a component can lead to the domino effect, i.e. that the component could be in its initial state losing all the work performed before the failure [25]. Among the techniques which avoid the domino effect is the coordination of the checkpointing that we have integrated in the fault-tolerant system.

One of the problems which can be posed in the management of the adaptation of distributed systems is the assurance of the message transmission of one process to another. For example, if a message concerning a request for coordination of the execution of an adaptation operation sent by a participant to another is lost, this leads to the cancellation of the adaptation even if the answer of the participant at the other site is positive what prevents the adaptation of the application to the new situation. To overcome this problem, we have used the message store technique described in the next section.

3.2.1.4 Message store

The message store [24] is a technique used for ensuring the message transmission of one process to another. It is a technique used in the mailing systems. According to this technique, the sender does not send the message directly to its destination. It sends it to an intermediate node representing a message queue handler. This latter saves the sender's message in the queue and it takes care of sending it to its destination. The sender is relieved from any additional concerns of message sending. If the recipient is down at the time when the sender sends the message, the message queue handler waits until the server comes up. Moreover, in the case when the message queue handler fails, the sender message remains in the queue and it will be sent to the destination when the message queue has recovered.

We have implemented this technique in the adaptation system for ensuring the message transmission between the negotiators of the adaptation strategy and the coordinators of the reconfiguration execution which are deployed at the different sites (see section 3.3). Also, we have implemented this technique in the fault tolerant system in order to ensure the transmission of messages between all the coordinators of distributed checkpointing and recovery at the different sites.

As the fault-tolerant system is separated from the adaptation system and the application itself, and as the implementation of this system is based on the component paradigm it is easy to add other techniques or to reuse this system or also to evolve it without touching the application or its adaptation system.

3.2.2 Presentation of the fault-tolerant system components

In this section, we present in detail the components of the fault-tolerance system and their functioning.

The component «*VerificationManager*». This component is responsible for the verification of the application consistency. It performs the verification of the conformity of the application components to their component model and architecture style. For that, it has two sub-components *«StructureChecking»* and *«BehaviourChecking»*: the first allows making a structural verification of the application whereas the second allows the verification of the behaviour of the application components. These two sub-components trigger the verification of the coherence rules contained in the knowledge base as explained in the section 3.1. For the verification of the components behaviour we considered only the verification of the component properties.

The component *«FTS-Coordinator».* As we deal in this work the distributed applications, the coordination for the distributed checkpointing and for the backward recovery in the case of faults or crashes is very important. For that, the fault-tolerant system has a component *«FTS-Coordinator»* for such coordination. In order that this component reaches its goal, it is composed of two sub components *«CheckpointingCoordinator»* and «*RecoveryCoordinator*». The first allows the coordination for the distributed checkpointing whereas the second allows the coordination for the distributed recovery. The sub-component *«CheckpointingCoordinator*» implements the protocol of the distributed checkpointing described previously. The protocol of the component *«RecoveryCoordinator»* will be presented in the next sections.

The component *«FaultDetector».* This component is responsible for: (1) the monitoring of the application components for detecting the faults which can appear in the application, and more precisely, the components' crashes and also (2) the reification of the calls of component services (i.e. the service request). For that, this component is composed of two types of component; components of type *«WatchDog»* and only one component of type «*ApplicationController*». The firsts are charge of the monitoring of the application components. They ping periodically the elements that they supervise for detecting the failed ones. If a component «*WatchDog*» detects that the component which it supervises is crashed, it calls the recovery function of the recovery manager for treating this fault.

The *«ApplicationController»* plays an important role in the fault-tolerant system. At the interception of a call

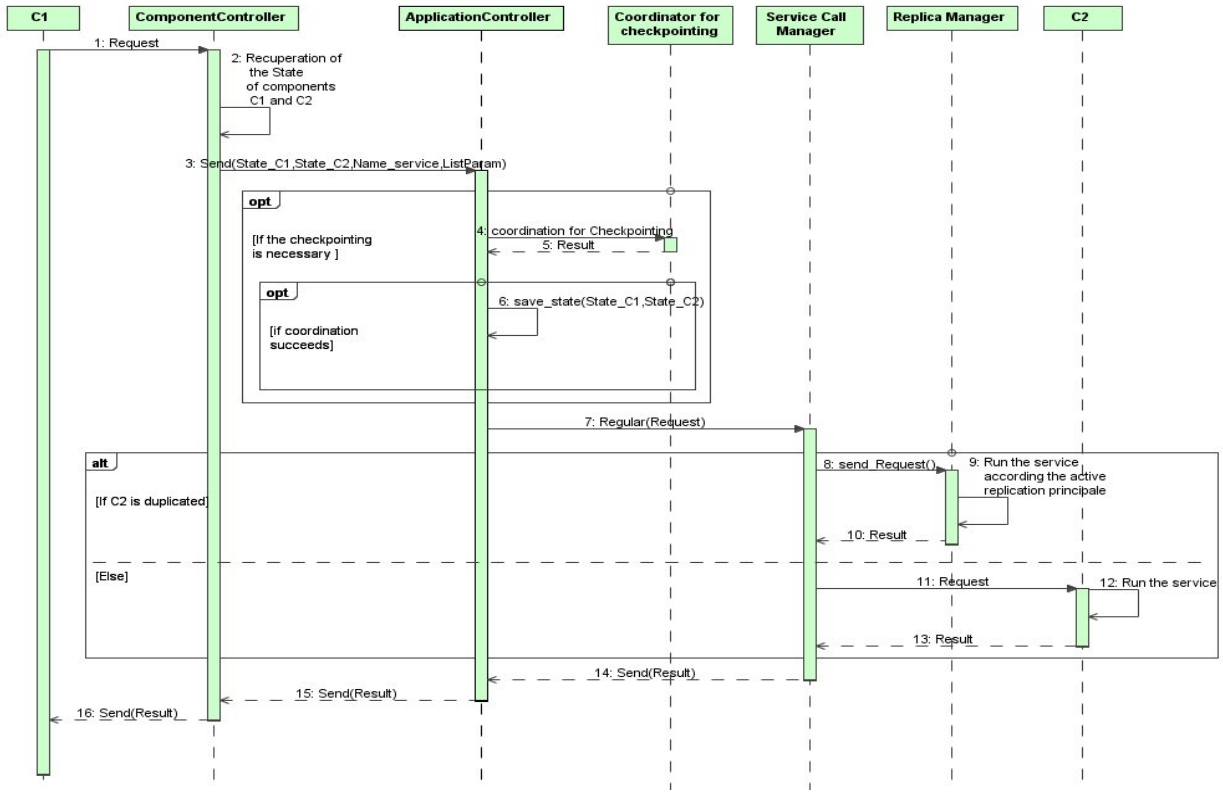

Figure 4: An execution without breakdown of a component C1 request by a component C2.

of service of one component by the corresponding controller, this last extracts the internal state of the two components client and server as well as the different parameters, and then it passes this information to the *«ApplicationController»*. This last, verifies if the checkpointing is necessary by checking if the time has passed compared to the last checkpointing such as this operation is performed periodically around an interval of time indicated by the component QoS-Manager. If the time is passed, the application controller asks the coordinator of distributed checkpointing to launch coordination so necessary for saving a checkpoint. After, it delegates the execution to the manager of the call of service. This last takes care of the processing of the service request.

The component *«ServiceCallManager»*. It is responsible for the management of the execution of the requests (i.e. calls of component services) submitted by the «*ApplicationController*». For the execution of these requests there are two cases: if the component server of the required service is duplicated the «*ServiceCallManager*» passes the execution of the request to the replica manager, which will manage the request processing according to the active replication technique. Otherwise, i.e. if the component server is not duplicated the manager of the execution of the call of service sends directly the request to this server and it awaits the reception of the execution result around a certain time indicated by the «*QoS-Manager*». If it does not receive a response after this waiting period it detects that the component provider of the required service is failed. In this case, it must call the recovery function of the recovery manager for tolerating the application to this fault.

The figure 4 presents a sequence diagram summarizing much more the functioning of the two components «*ApplicationController*» and «*ServiceCallManager*». This diagram explains the running of an execution without failure of a service request sent by a component C1 to a component C2.

The component *«ReplicasManager»*. It implements the active replication technique presented previously. This component is responsible for the execution of the service requests of the duplicated components. It executes the required service according to the active replication technique principle.

The component *«RecoveryManager»*. This component plays a very important role in the preservation of the application consistency. It treats the faults detected by the two components «*FaultDetector*» and «*ServiceCallManager*». The backward recovery technique is implemented in this component.

When a component «*WatchDog*» detects that the component which it supervises is failed or when the «*ServiceCallManager*» detects that the component provider of the required service is failed, it calls the recovery function of the recovery manager, which will carry out the backward-recovery of the failed component and also the components which depend on this last and which exist as well at the site of the failed component, or at the other sites. This distributed recovery is necessary

because we deal in this work the distributed applications, so, there is dependence between the distributed components which requires a recovery of all these components. Therefore, coordination for distributed recovery is necessary. For that, the component «*FTS-Coordinator*» has a sub-component «*RecoveryCoordinator*» that performs such coordination. This component has a specific protocol which we have proposed.

The basic idea behind a protocol for a distributed recovery is to ensure that all components depending on the failed component roll back to their previous coherent states. The set of the realized local recoveries must form a coherent global state of the application.

The algorithm starts with an initiation of a request for coordination to the recovery coordinator by the recovery manager for recovering all the components which are distributed on the other sites and which depend on the failed component (if they exist). In this case, the coordinator according to his policy decides if the recovery requires coordination or not. If the failed component depends on other components installed on other sites, the coordinator invites the participants (which are the recovery managers of the fault-tolerant systems of the instances of the proposed framework and which are installed on the other sites) to perform the rollback towards the last saved checkpoint for the components which depend on the failed component. If a participant rejects the request for any reasons, it replies *"No"*. Otherwise, the participant performs the recovery and it replies "*Yes*". If all the participants' reply "*Yes*", the communication stops and the coordinator announces the success of the recovery.

If there is one or more participant replying by "*No*", the coordinator will wait a certain time, then, it will send again a request for recovery to the participants who replied by "*No*" in order that they perform another time the recovery of the components which depend on the failed component. The basic idea behind this waiting before sending the recovery request again to the components replying by "*No*" is that these components can return to operate (reception of the requests) after this waiting period because they were for example in the process of running a reconfiguration operation or a critical operation (e.g. an operation on the database). The table 2 presents an overview of the proposed distributed recovery algorithm.

Notice that only one operation of recovery coordination can be carried out at the same time and this is for guaranteeing the application coherence.

The component «*QoS-Manager*». It is a component used for managing the service quality level in the application. This component allows to the user to change a set of parameter through a graphical interface in order to increase or decrease the level of the quality of the service in the application. These parameters are: the waiting time of the execution result of a request by the *«ServiceCallManager»*, the interval of time during which a checkpointing is performed, the interval of time during which a component *«WatchDog»* ping the component which it supervises, the interval of time during which the component *«CaptureContext»* supervises the application environment and the max number of replicas of each type of application's component.

Table 2: An overview of the distributed recovery algorithm.

Recovery coordinator	Participants

begin
request participants to perform the recovery of the components depending on the failed component ;
await all replies ;
if (all replies = "*Yes"*) stop the coordination ;
else
begin
await a certain time ;
request the participants who replied by *"No"* to perform the recovery ;
stop the coordination ;
end-else
 	
End.
	Begin
	if (accept request)
	begin
	perform the components recovery;
	reply "*Yes"*;
	end-if
	
	else
	reply "*No"*;
	
	End.

3.2.3 The fault model

The use of the four techniques (active replication, message store, distributed backward recovery and distributed checkpointing) allowed us to propose a powerful fault-tolerant system for the proposed framework able to tolerate many types of failure. For the components crash, the proposed fault-tolerant system is able to detect them via the components «*WatchDog*». Each component in the application sends periodically a heartbeat message to its monitor «*WatchDog*» and this last periodically checks the heartbeat. If the heartbeat message from the supervised component is not received by a specified time, the component «*WatchDog*» assumes that the supervised component is hung. This problem will be treated by the recovery manager. The faults of type omission are treated in our approach via the message store technique which ensures the transfer of messages from an entity to another. The faults of type *"late timing"* are detected by the component *«ServiceCallManager»* such as each type of request has an interval of result waiting indicated by the component «*QoS-Manager*». If time passes and the manager of the calls of service has not received a response, it detects that there is a problem into the component provider of the service. This problem will be treated by the recovery manager as explained in the previous section. The faults in value require for their treatment the existence of several replicas. The active replication technique which we have incorporated in the fault-tolerant system allows treating this type of faults because a client request is sent to all the servers. If a response from a server is different to the majority of servers' response, this server has a fault of value. As we deal in this work the distributed

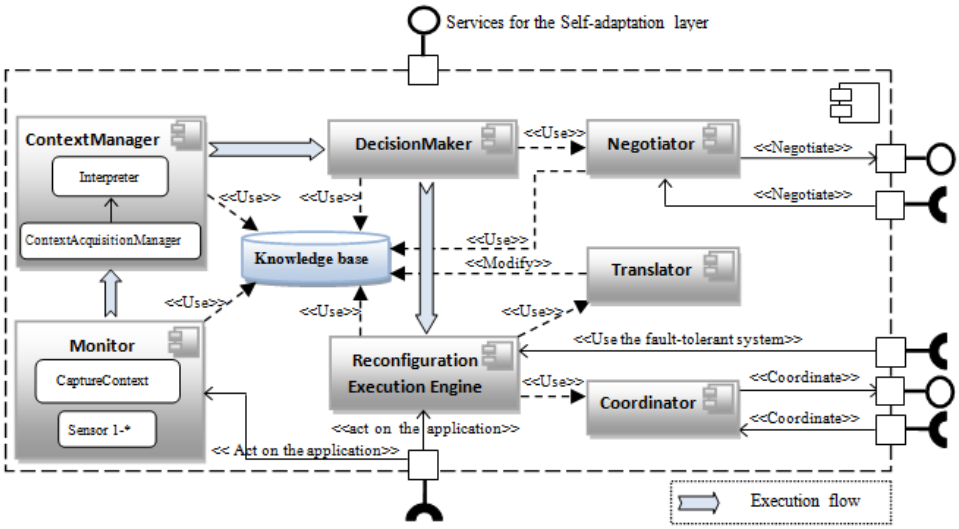

Figure 5: Overview of the adaptation system.

applications which are by nature complexes, the creation of several replicas of each functional component of the application for the treatment of the faults in value is impossible. For that, we propose to duplicate only the components the more used in the application and which are generally a subject of adaptation.

3.3 Adaptation system

This section presents in detail the adaptation system and its functioning for realizing distributed and reliable reconfigurations.

For the definition of the adaptation system, we consider a set of constraints which are: (i) taking into account of the distributed nature of the software to make it adaptable, (ii) reliability of the distributed reconfigurations, and (iii) flexibility and adaptability of the adaptation system.

The proposed adaptation system is designed according to the classical autonomic control loop MAPE-K (Monitoring, Analysis, Planning and Execution) [3], which is the most common approach for self-adaptive systems [5, 33].

So, in our adaptation system we have implemented the elements of this loop as separate components. The monitoring, analysis and adaptations are performed by the MAPE-K control loop. A significant part of the negotiation of adaptation strategy and coordination of reconfiguration execution were externalized from the control loop. Moreover, we have chosen to merge the analysis and plan components because a significant part of these components' logic is externalized from the components and stored in the knowledge base (Prolog script). Therefore, not leaving too much of analysis and planning to be performed within those two components.

Plus the set of components that implement the MAPE-K loop, the adaptation system contains a component *«Negotiator»* which negotiates an adaptation strategy with its similar at the other sites, a component «*Coordinator*» that coordinates the execution of reconfiguration actions, and a component «*Translator*» which executes the reconfiguration actions of the adaptation strategy on the architectural representation of the application. The figure 5 shows an overview of this system.

We propose the implementation of the whole cycle of the MAPE-K loop as a chain of responsibility pattern [13]. We have proposed to use this pattern because the processing is distributed on several objects (components of the adaptation system). When a component finishes its processing, it passes the execution to the next component. Moreover, it is easy to vary the components involved in the processing which makes the adaptation system more flexible.

3.3.1 Monitoring

The «*Monitor*» is the first component in the chain that comprises the control loop. It is responsible for periodically collecting information of the managed elements (i.e., sub-system of the application managed by the adaptation system) and of the execution of the application (CPU consumption, memory usage, bandwidth, service calls per minute). To achieve this goal, the monitor has a sub-component «*CaptureContext*» that collects information about the application execution plus a set of sub-components «*Sensor*» that collect information about the set of the application components at its site. These two sub-components of the monitor pass the collected information to the next object that is part of the execution chain, next to the context manager.

The «*ContextManager*» is the second component in the sequence of the responsibility chain. It interacts with the sensors associated with the execution environment and the application for collecting the information needed to characterize the execution context. For that, it has two sub-components «*ContextAcquisitionManager*» and «*Interpreter*». The context acquisition manager gathers the information collected by the sensors of the «*Monitor*»

and saves them in the knowledge base. After, it delegates the execution to the «*Interpreter*». This last, interprets data provided by the «*ContextAcquisitionManager*» in order to provide a significant contextual data. Notice that, the received data are separately interpreted for each type of measurement in order to provide a significant contextual data. If a suitable context change is detected the «*Interpreter*» notify the decision maker (see next section) of this change as this last subscribes to events near the context manager.

3.3.2 Analysis

The aim of the analysis phase is to see whether a reconfiguration action is required or not. For that, the decision maker component «*DecisionMaker*» is the third component in the sequence of the responsibility chain. It plays the role of the analysis and plan phases in the MAPE-K control loop. This component is responsible for taking decisions on adaptation. It provides in output an adaptation strategy that will be executed in the execution phase of the control loop.

3.3.2.1 Negotiation process

As we deal in this work the distributed reconfigurations, the negotiation is a significant step in the decision-making on adaptation. It is a cooperative process in which a group of adaptation systems reach an agreement on a comprehensive adaptation strategy. We define a global strategy as a set of strategies that the decision makers of the different adaptation systems choose during the negotiation process. Noting that, the negotiation must guarantee the independence in the decision-making of each «*DecisionMaker*» and ensure the global validity of a local decision.

The negotiation is started by the initiating decision maker. This last chooses an adaptation strategy. Then, it asks its negotiator to negotiate this chosen strategy. This negotiator proposes simultaneously to each participant the strategy that the initial decision maker has chosen. The negotiator of each participant receives the strategy and interprets its policy to reason on its applicability. It can then accept, refuse or propose a modification of the strategy; and then, it answers the initiating negotiator. When this last receives all answers, it thinks on the acceptances and/or the applicability of the modifications asked. When all the participants accept the strategy, the negotiation succeeds. Otherwise, it detects and solves the conflicts and can then, in its turn, propose a modification of the strategy. The negotiation process is stopped if one negotiator refuses a strategy or if a stop condition is checked. This condition is in connection to the authorized maximum time of negotiation or with the maximum number of negotiation cycles. If the negotiation succeeds, the initiating negotiator returns to the initiating decision maker the strategy resulting from the negotiation and sends to the negotiator of each participant the final strategy. If the strategy resulting from the negotiation is a new strategy, i.e. not exists in the adaptation policy, the decision maker adds it to the knowledge base and precisely to the adaptation policy part. This operation allows enriching the knowledge base with new adaptation rules in order to better adapt to the new changing situations. At the reception of this strategy, each participant (i.e. negotiator) asks to its decision maker to adopt the strategy resulting from the negotiation and delegates the execution to the next object in execution chain that is the reconfiguration execution engine. In the opposite case (i.e. negotiation failure), the initiating decision maker and participants are informed of the negotiation failure. Otherwise, the adaptation is cancelled and the loop cycle is stopped.

3.3.3 Execution

In order to increase the reliability of the reconfigurations executed by our framework we have used the transaction technique. This technique was originally used in the system managing databases [14]. Their use is widespread in all computer systems where there is a need to maintain the consistency of the information in spite of concurrency and the occurrence of failures. The transactions are thus a means to make systems fault-tolerant. A transaction consists to carry out a coherent computing operation consisting of several actions. The operation will be valid only if all its unit actions are carried out correctly. So, we speak about the commit. Otherwise, all data processed during the running of the operation must be returned to their initial state to cancel the transaction. So, we speak about the rollback.

We have used the transaction technique to define transactional reconfigurations.

According to the transactions principle each transaction is made up of a set of primitive operations. So, in our context an adaptation operation *Adop* is a transaction when its primitive operations are the primitive reconfiguration actions *Prac*. For example, the replacement operation of a component *C1* by another component *C2* is made up by the following primitive actions: stopping the component *C1*, creation of a new instance of the component *C2*, transfer of the *C1* state to the new instance of *C2* for preserving the application consistency and finally the start of the new instance of *C2*. The component replacement in our framework is carried out similarly with the work in [35].

We define the evolution of a component-based system by the transition system < C, Adop, → >:

- $C = \{C_0, C_1, C_2, \ldots\}$ *a set of configurations,*
- *Prac* $\in$ *{Instantiation/Destruction of component, Addition/Removing of component, modification of the component attribute value, modification of the life cycle of a component and adding of new behaviours}*
- *Adop is a set of Prac,*
- $\rightarrow \subseteq C \times Adop \times C$ *is the reconfiguration relation.*

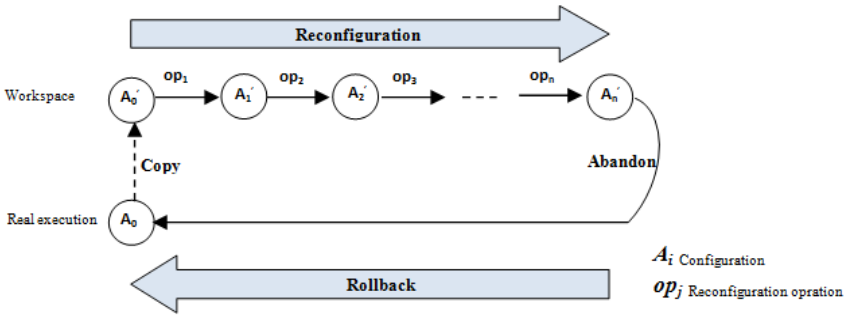

Figure 6: Abandon model of an adaptation operation.

3.3.3.1 Reconfiguration actions of the proposed framework

In our framework the dynamic reconfigurations are based on the following primitive actions:

- *instantiation/destruction of components*
- *addition/removal of components*
- *starting/stopping of components*
- *setting name of components*
- *setting the operating parameters of components or combinations of them.*

Thus, an adaptation strategy consists of a set of adaptation operations, where each operation is composed of at least only one primitive reconfiguration action.

The behaviour of each application component is generally statically encoded. However, changes in the application context, changes in use, changes in resource availability or the appearance of faults in the system, may require further abilities [10]. For this, it is very important to introduce dynamically the ability to add new behaviours to the application's components. The AOP (Aspect-Oriented Programming) and scripting languages are two techniques used for this end. In the AOP and with the runtime weaving, the binding between the logic code and aspect is done during the execution. The advantage of runtime weaving is that the relationship between the functional code and the aspects can be dynamically managed. Nevertheless, the use of the AOP for adding new capabilities to the system has one disadvantage is that the software system could be in an inconsistent and/or unstable state [10].

For the scripting languages, they allow the incremental programming, i.e. the possibility of running and developing simultaneously the scripts [8]. With these languages we can modify the code of a component without stopping it. Therefore, with this technique the addition of a new behaviour is much easier than the use of the AOP technique, but these languages are not powerful as the compiled ones. For that, the developer must find a compromise between the use of scripting languages and the AOP technique in order to improve the performance of the application. The implementation of the mechanism carrying out the addition of new behaviours to the application's components is left to the developer.

3.3.3.2 Quiescence management

Obtaining a reconfigurable state also called quiescent state [17, 18] is a very significant step in the reconfiguration process since it helps to ensure the application consistency in the case of reconfiguration. A reconfigurable state is a state from which a reconfiguration action can be performed without affecting the consistency of the application. For that, in our approach a reconfiguration action is carried out on a component if this last is in a reconfigurable state.

To search for a reconfigurable state, we have integrated in the proposed framework Wermelinger algorithm [18] which extend Kramer/Magee's algorithm proposed in [17]. Wermelinger proposes to block only connections between the components implied in the reconfiguration. An advantage of this algorithm is that interruption time is minimized while only affected connections must be blocked in contrast to whole components.

3.3.3.3 Algorithm of the reconfiguration execution engine

The reconfiguration execution engine is the fourth component in the execution chain. It undertakes the execution of the adaptation strategy proposed by the component «*DecisionMaker*». Firstly it (1) launches a search for a reconfigurable state before running the reconfiguration actions. Then, it (2) triggers the execution of the reconfiguration actions of the strategy. Notice that, we simulate firstly the execution of the reconfiguration on the architectural representation of the application. If no fault is detected, we execute the reconfiguration on the running application. Therefore, the effects of reconfigurations are not directly applied on the system which facilitates the cancellation of the reconfiguration in the case of its execution failure. This operation will be simply the removing of the work copy which has been used for the simulation of the reconfiguration. Figure 6 shows our abandon model of an adaptation operation.

As the reconfiguration of distributed application is a global reconfiguration process composed of distributed local reconfiguration processes, the proposed adaptation system incorporates a component for the coordination of the reconfigurations execution.

Following the running of each primitive reconfiguration action on the architectural representation the reconfiguration execution engine (3) carries out the verification of the consistency of the application structure and the verification of the validity of the behaviour of its components via the component «*VerificationManager*» of the fault-tolerant system. If a constraint is violated, the adaptation operation must be stopped for preserving the consistency of the application. In this case, the reconfiguration execution engine removes the copy of work used for the simulation of the reconfiguration. Moreover, this initiating reconfiguration execution engine notifies its coordinator of the execution failure of the primitive reconfiguration action in question. This last, notifies the other participants (coordinators of the reconfigurations execution deployed at the other sites) of this failure so that they can cancel the adaptation operation at their level in order to preserve the global consistency of the application.

In the opposite case, i.e. where the «*VerificationManager*» does not detect any error following the running of a primitive action of reconfiguration, the reconfiguration execution engine sends a message *"ApplyNextAction"* to the coordinator of the execution of reconfiguration actions. This last awaits the reception of all the participants' responses. Notice that, this waiting time of the participants' responses by the coordinator of the initiating reconfiguration execution engine is fixed. If this coordinator does not receive a message of a participant during this waiting period, it regards it as "*reconfiguration failure*". If one participant replies by "reconfiguration failure" the coordinator announces the failure of the execution of the reconfiguration. Otherwise, it asks for the participants to perform the next primitive reconfiguration action and the process is still repeated. If all the adaptation operations of the strategy are executed on the architectural representation of the application (copy of the work) without faults, the reconfiguration execution engine (4) runs these actions on the running system. Also, the copy of work used for simulating the execution of the reconfiguration is saved as the new architectural representation of the application. Notice that, this operation ensures the conformity of the architectural representation of the application to its system in running and it has many advantages. For example, it facilitates the comprehension of the software through its architecture, and thus its evolution because the architectural representation always conforms to the system. Finally, the reconfiguration execution engine (5) unblocks the connections blocked during the phase of searching for a reconfigurable state. The end of the execution of this operation determines the end of the control loop cycle. The running of the reconfiguration execution engine is summarized by algorithm 1.

Algorithm 1

$pact_j$ is a primitive reconfiguration action

```
1:  Begin
2:  SearchForReconfigurableState();
3:      For all pact_j ∈ startegy do
4:      RunPactOnArchRep(pact_j); /* execute the primitive reconfiguration action pact_j on the architectural representation via the component
5:                                   «Translator» */
6:      if not IsConsistentApplication () then
7:              SendMessageToCoordinator ("reconfiguration failure");
8:              RemoveWorkCopy(); // Removes the working copy used for the reconfiguration simulation
9:              BREAK; // to exit the loop "for"
10:     end_if
11:     else //case where the application is consistent
12:             SendMessageToCoordinator ("ApplyNextAction");
13:             response←coordinator.CoordinationDecision();
14:             if response != "ApplyNextAction" then
15:                 SendMessageToCoordinator("reconfiguration failure");
16:                 RemoveWorkCopy(); // Removes the copy of work used for the simulation of the reconfiguration
17:                 BREAK; // to exit the loop "for"
18:             end_if
19:     end_else
20:   end_for
21: if all actions pact_j in strategy are executed // If the reconfiguration is succeeds
22:     RunAllpactOnSystem(); // execute the primitive reconfiguration actions on the Running system
23:     SaveChanges(); // Save the performed changes on the architectural description
24: end_if
25: ReactivateConnections(); // unblock the blocked connections
26: End.
```

3.4 The adaptation manager of the fault-tolerant and adaptation systems

This manager implements the self-adaptation layer of the proposed architecture model. It manages the two systems of adaptation and fault-tolerance and looks after the adaptation of these two systems to the changes that they carry out themselves on the application components in order to ensure the correct operation of the global application. This manager allows (1) to replace the negotiator of adaptation strategy or the reconfiguration execution coordinator or also the recovery and checkpointing coordinator of the fault-tolerant system by other components if they crash in order to ensure a good management of the application changes. For that, this manager has a set of monitors of the type «*WatchDog*» which monitor these components. Also, it allows (2) in the case where an operation of removing of an application component is carried out, to stop and remove the two components: «*Sensor*» of the adaptation system and «*WatchDog*» of the fault-tolerant system that monitor this component to avoid the introduction of errors into the running of the application. This manager also carries out (3) the update of the two Prolog scripts representing the adaptation policy and the coherence rules by removing the facts representing the adaptation strategies and the coherence rules which are in relation with the removed components.

4 Implementation and validation plan

In this section, we give details and technical choices made to implement a prototype of the proposed framework. We also present the validation plan of this framework.

4.1 Background

For the implementation of the elements of the proposed framework, we have used the two component models EJB (Enterprise Java Beans) and ScriptCOM. So, the implementation of this framework is divided into two parts; one part implemented with EJB and the other with ScriptCOM. EJB [4] is an industrial model; we have used it because it is based on Java that is a powerful programming language that meets our implementation needs (support for AOP, support for native codes via JNI API, Java COM Bridge, support for the remote method invocation via RMI API, an API to access system information like SIGAR[1] and a support for multi-threading).

ScriptCOM [8] is a component model extension of COM (Component Object Model) [2] allowing the dynamic adaptation of the COM components. It allows the development of adaptable scripting components. Notice that, with the scripting languages we have the possibility of developing and running simultaneously the scripts which represents the component's implementation [8]. In this model, the component adaptation is carried out through three controllers which are: the interface controller, property controller and script controller. Moreover, as it is an extension of the COM model it benefited from the advantages of this latter (support of the distributed applications, independence of the programming languages, versioning...). We have used this model in order to facilitate the adaptation of the proposed framework. We think that dynamic inclusion and removal of adaptation management concerns allows the improvement of adaptation to the evolving needs without stopping the entire framework.

4.2 Framework implementation

We have designed a set of software components that implement the different elements of the proposed framework. The implementation of the components of type «*ComponentController*» and the different sensors as well the effectors (part performing the execution of the reconfiguration and backward recovery) are realized with EJB model. The coordination for reconfiguration execution and backward recovery and also negotiation parts are implemented with the two models EJB and ScriptCOM. The rest part of the framework is developed as a set of ScriptCOM components that we can add, remove or update at runtime. This is just one of possible implementations and particularly, this has been designed to provide self-adaptable capabilities to the framework.

For the implementation of the controllers «*ComponentController*» of the functional components of the application we choose the use of the aspect oriented programming. So, the implementation of each controller is based on an aspect. This aspect has a generic pointcut that intercepts all the incoming service calls to the controlled component and treats them as explained in the section 2.1.

For the knowledge base, i.e. the architectural representation of the application, adaptation policy and coherence rules description we have used the language Prolog as explained in the section 3.1. We have used the JPL[2] library which uses the SWI-Prolog foreign interface and the Java JNI interface providing a bidirectional interface between Java and Prolog that can be used to embed Prolog in Java as well as for embedding Java in Prolog. Also, we have used another interpreter Prolog[3] developed with the JavaScript language in order that it will be used with the part of the framework developed with ScriptCOM. The three elements of the knowledge base are contained in separate scripts which facilitate their modifications at runtime. We can then add, remove or change rules or facts in the knowledge base without stopping the framework.

Our framework is independent from particular component models. Therefore, elements of the application layer, i.e. components implementing the business logic of the application can be developed using any component models. The implementation for a

[1] https://support.hyperic.com/display/SIGAR/Home

[2] http://www.swi-prolog.org/packages/jpl/

[3] http://ioctl.org/logic/prolog-latest

specific component model is made with the least effort in the part developed using the model EJB and without changing the main adaptation concepts.

4.3 Validation plan

The objective of the validation in this paper is to test the influence of the proposed framework on the application response time and the adaptation time. These criteria are measured with randomly generated configurations which we have developed using the model EJB. The components of this application execute a few arithmetic operations and they are distributed on two sites. The evaluation test is made by comparing two versions of the same application; one incorporates the proposed framework, the other one without this framework. All the experiments were run on Intel Core 2 Duo CPU T5670 workstations with 1.0 GB DDR2 memory and Windows XP SP3 as the operating system.

The first evaluation consists to test the influence of the proposed framework on the application response time. This test is made by comparing the running of a certain number of requests on the two versions of the application (without and with the proposed framework). The Table 3 shows the response times before and after the incorporation of the framework.

We have calculated the response time increase in the version which implements the proposed framework and we found that the overhead for functional method calls is about 34% of the overall execution time.

Table 3: Increase rate of the response time.

Request Numbers	Response time average : without the Framework	Response time average : with the Framework	Increase rates of the response time
100	16.91 ms	22.87 ms	**35.21 %**
200	33.39 ms	44.48 ms	**33.20 %**
300	50.04 ms	67.31 ms	**34.51 %**
500	83.38 ms	111.18 ms	**33.33 %**
700	115.39 ms	156.1 ms	**35.25 %**
900	150.16 ms	205.67 ms	**36.96 %**
1000	169.83 ms	222.14 ms	**30.80 %**
Average			**34.18 %**

The second evaluation consists to measure the adaptation time, which is calculated as follows:

$$\mathbf{T}_{adaptation} = \mathbf{T}_{rspWAdap} - \mathbf{T}_{rspNAdap}$$

Where: $\mathbf{T}_{adaptation}$ is the adaptation time.

$\mathbf{T}_{rspWAdap}$**:** is the response time with adaptation.

$\mathbf{T}_{rspNAdap}$**:** is the response time with the proposed framework but without adaptation.

Table 4 shows the obtained results. The adaptation time average is approximately 430,95ms. Certainly, this figure is very large compared to the response time of one request, which is approximately 0,22ms (response time of an execution of one request with the framework).

Table 4: Adaptation time.

Request Numbers	Response time average : with the Framework but without Adaptation	Response time average : with the Framework and with Adaptation	Adaptation time
500	111.18 ms	531.39 ms	**420.21 ms**
700	156.1 ms	592.56 ms	**436.46 ms**
800	179.71 ms	639.13 ms	**459.42 ms**
900	205.67 ms	631.78 ms	**426.11 ms**
1000	222.14 ms	634.68 ms	**412.54 ms**
Average			**430.95 ms**

The obtained adaptation time is great, but it is acceptable because the adaptation is distributed (at two sites in this test application) which requires a negotiation of the adaptation strategy and a coordination for the execution of the reconfiguration actions and this influence the adaptation time. Moreover, as we have used the component model ScriptCOM for the development of one part of the proposed framework, this, influence the application response time and also the adaptation time because the implementation of the components of this model is based on the language Jscript[4], which is an interpreted language. So, the execution will be slower than the compiled versions. Notice that, we have used this model in order to facilitate the adaptation of the proposed framework.

Finally, we can say that the obtained results confirm that our framework is very suitable for developing distributed applications where we prefer the reliable dynamic adaptability more than performance.

5 Related work

The problem treated in this paper accosts the domain of research around the dynamic adaptation of the computing systems and in particular the distributed component-based systems.

In terms of model-based approaches Kramer and Magee [15] have proposed layered reference architecture for self-adaptive software. The bottom layer of this architecture is the component control layer which contains the system's application-level functionality. The change management layer is the middle layer. It manages the changes of components state or environment. For that, it contains a set of pre-compiled plans to deal with the different situations encountered by the system. The uppermost layer is the goal management layer which

[4] http://msdn.microsoft.com/fr-fr/library/hbxc2t98%28vs.85%29.aspx

generates new plans if none of the existing plans can address the current situation, or a new system goal is introduced. Also, we have proposed a three layer architecture model where the bottom layer is the application layer similar to the component control layer in the Kramer and Magee model's. Unlike this model, the change management layer of our model contains two systems managing separately the adaptation and the fault tolerance of the application layer. Moreover, to the difference of the uppermost layer of the Kramer and Magee model's, the uppermost layer of our model introduces the self-adaptation capabilities to the framework itself. It ensures the service continuity of the change management layer and manages the adaptation of this layer to the changes which it carries out itself on the application layer. Moreover, our architectural model is reflexive. In the Kramer and Magee model's, the distributed reconfigurations are possible through the decentralized architecture of the change management layer implementation proposed by the authors. From a reliability point of view, Kramer and Magee have expressed that a server failure is a predicted state change and the change management layer must include a procedure for dealing with the change. For that, they propose the use of the repairing strategy of the faults described by Garlan and Schmerl in [27] as a plan executed by the change management layer.

Several research activities [7, 9, 12] implement the autonomic control loop to dynamically reconfigure systems. For example, in [7] the authors use a component-based approach for modelling a framework that provides flexible monitoring and management tasks and allow introducing adaptation to component-based SOA applications. The framework implements the different phase of the autonomic control loop. The main purpose of the authors is to build a framework supporting heterogeneous components implementing the MAPE-k phases as SCA components. This framework supports the development of distributed applications, but it doesn't support to perform distributed reconfigurations while our framework is conceived especially for doing such type of reconfigurations.

A3 [10] is a framework for developing distributed systems that need adaptive features. A3 provides robust mechanisms of coordination that the components can use to share their own knowledge and knowledge of the system to which they belong. The framework itself is self-adaptable. A3 exploits the idea of *group* to organize a system in a set of independent partitions, and reduces the communication problem. From an adaptation point of view, A3 supports the distributed adaptations and it allows indeed interesting adaptations. This framework does not have any mechanism to preserve the reconfiguration reliability. It treats only the fault of type messages omission. Moreover, a reconfiguration action is executed at the system directly, i.e. without reaching a reconfigurable state before the execution of such action.

Huynh et al. [20] propose a platform supporting distributed reconfiguration of the component-based applications. This platform integrates a solution for the management of system states at reconfiguration time. The authors define different system states regarding reconfiguration and ways that the system will act accordingly. This platform allows to correct reconfiguration plans if a disconnection is detected during the reconfiguration in order to continue the reconfiguration if possible, or recover if the reconfiguration fails. It also allows the coordination of the distributed reconfiguration actions. In contrast, to this platform, our framework integrates a negotiation mechanism which allows the negotiation of the adaptation strategy before the coordination of its execution that is a very important point in the distributed reconfiguration process.

In [21], a transactional approach is proposed to ensure reliable reconfigurations in the context of component based systems and particularly in the Fractal component model. To ensure atomicity of reconfiguration transactions, operations performed in transactions must be cancelled if a fault occurs before the end of the reconfiguration. This operation of cancelation of the reconfiguration operations effect is carried out by the execution of the reverse action of each reconfiguration operation performed because the reconfiguration operations are carried out directly on the system. In contrast to this approach, we have proposed to carry out firstly the reconfiguration on the architectural representation of the application which facilitates the cancelation of this operation if there is a problem. From a reliability point of view, the authors propose the use of the integrity constraints to define the system consistency for guaranteeing the respect of these constraints at runtime.

6 Conclusion

In this paper, we have presented a framework for building distributed and dynamic component-based systems. The proposed framework is based on a reflexive three layer architecture model which we have proposed. The bottom layer of this model is the application layer. It contains the system's application-level functionality. The change management layer is the middle layer. It manages the changes of the bottom layer. The uppermost layer is the self-adaptation layer that introduces the self-adaptation capabilities to the framework itself. It ensures the service continuity of the change management layer and manages the adaptation of this last to the changes which it carries out itself on the application layer. The proposed framework implements the two uppermost layers of the proposed architecture model and it is based on a decentralised architecture. It incorporates two separate systems that manage the dynamic adaptation and fault tolerance of the application components and also, an adaptation manager implementing the self-adaptation layer in the architecture model. The proposed framework is designed especially to support the distributed reconfigurations. For that, it incorporates a robust coordination and negotiation mechanisms for managing this type of reconfiguration. The adaptation system of this framework is designed according to the classical autonomic control loop MAPE-K which allows a better

management of the adaptation. As the preservation of the application consistency is an important point in the dynamic reconfiguration, the framework incorporates a separate fault-tolerant system implements four fault tolerance techniques (distributed checkpointing, active replication, message store and distributed backward recovery) which makes it able to tolerate most of faults type. Also, as the adaptation operations in this framework are executed as transactions, this increases the reliability of these operations. A prototype of this framework has been implemented using two component models; EJB an industrial model and ScriptCOM a component model for developing adaptable components, which facilitates the adaptation of the proposed framework.

However, the evaluation of the proposed framework has revealed that the adaptation time is long, for that we plan to improve the adaptation system of the proposed framework in terms of performances.

In the long term, we want to study the possibilities to extend our solution to support dynamic adaptation of other kinds of applications like web services.

References

[1] R. N. Taylor, N. Medvidovic and P. Oreizy (2009). Architectural styles for runtime software adaptation. *In 3rd European Conference on Software Architecture (ECSA)*, pp. 171-180.

[2] Microsoft Corp, "Component Object Model" accessed on July 30, 2013. [Online]. Available : http://www.microsoft.com/COM

[3] IBM. An architectural blueprint for autonomic computing. Autonomic computing whitepaper, 4th edition. 2006.

[4] V. Matena and M. Hapner (1999). Enterprise Java Beans Specification v1.1- Final Release. *Sun Microsystems*.

[5] B. H. Cheng et al. (2009). Software Engineering for Self-Adaptive Systems: A Research Roadmap. *In B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, editors, Software Engineering for Self-Adaptive Systems*, volume 5525 of Lecture Notes in Computer Science, pp. 1-26. Springer.

[6] M. Léger, T. Ledoux and T. Coupaye (*2010)*. Reliable dynamic reconfigurations in a reflective component model. *In Proceedings of the 13th international conference on Component-Based Software Engineering, pp*. 74-92.

[7] F. B. Cristian Ruz and B. Sauvan (2011). Flexible adaptation loop for component-based SOA applications. *In Proceeding of the Seventh International Conference on Autonomic and Autonomous Systems*, pp. 29–36.

[8] O. Aissaoui and F. Atil (2012). ScriptCOM an Extension of COM for the Dynamic Adaptation. *In Proceedings of IEEE International Conference on Information Technology and e-Services (ICITeS'12)*, pp. 646-651.

[9] Y. Maurel, A. Diaconescu, and P. Lalanda (2010). Ceylon: A service-oriented framework for building autonomic managers. *In Proceedings of the Seventh IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems*, pp. 3 –11.

[10] L. Baresi and S. Guinea (2011). A3: self-adaptation capabilities through groups and coordination. *In Proceedings of the 4th India Software Engineering Conference*, ISEC'11, pp. 11-20.

[11] C. Tan and K. Mills (2005). Performance characterization of decentralized algorithms for replica selection in distributed object systems. *International Workshop on Software and Performance*, pp. 257-262.

[12] M. Zouari, M.T. Segarra, F. André (2010). A framework for distributed management of dynamic self-adaptation in heterogeneous environments. *In the 10th IEEE International Conference on Computer and Information Technology*, CIT 2010, pp. 265–272.

[13] E. Gamma , R. Helm, R. Johnson and J. Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software. *Addison-Wesley Longman Publishing Co., Inc*.

[14] J. Gray and A. Reuter (1992). Transaction Processing: Concepts and Techniques. *Morgan Kaufmann Publishers Inc.*, San Francisco, CA, USA.

[15] J. Kramer and J. Magee (2007). Self-Managed Systems: an Architectural Challenge. *Future of Software Engineering*, pp. 259-268.

[16] P. Oreizy, N. Medvidovic and R. N. Taylor (2008). Runtime software adaptation: framework, approaches, and styles. *In Companion of the 30th international conference on Software engineering (ICSE Companion '08)*. ACM, New York, NY, USA, pp. 899-910.

[17] J. Kramer and J. Magee (1990). The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, vol. 16, no. 11, pp. 1293–1306.

[18] M. Wermelinger (1997). A Hierarchic Architecture Model for Dynamic Reconfiguration. *In Proceedings of the Second International Workshop on Software Engineering for Parallel and Distributed Systems*, pp. 243–254.

[19] A. Ketfi, N. Belkhatir, P.Y. Cunin (2002). Adaptation Dynamique, concepts et experimentation. *In proceedings of the 15th International Conference on Software & Systems Engineering and their Applications ICSSEA02*, Paris, France.

[20] Ngoc-Tho Huynh, An Phung-Khac and Maria-Teresa Segarra (2011). Towards reliable distributed reconfiguration. *In Proceedings of the International Workshop on Adaptive and Reflective Middleware*, ARM 2011, pp. 36–41.

[21] M. Léger, T. Ledoux, and T. Coupaye (2007). Reliable dynamic reconfigurations in the fractal component model. *In Proceedings of the 6th*

international workshop on Adaptive and reflective middleware, ARM '07.

[22] E. J. Chikofsky and J. H. Cross II (1990). Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, vol. 7, no. 1, pp. 13-17.

[23] Anh Nyguyen-Tuong, Steve Chapin, Andrew Grimshaw, Charlie Viles (1998). Using Reflection for Flexibility and Extensibility in a Metacomputing Environment. University of Virginia, Technical Report CS-98-33.

[24] J. Koistinen (1997). Dimensions for Reliability Contracts in Distributed Object Systems. *Hewlett Packard Technical Report*, HPL-97-119.

[25] M. R. Lyu (2007). Software Reliability Engineering: A Roadmap. *In Future of Software Engineering*, *IEEE Computer Society*, pp. 153-170.

[26] O. Aissaoui, F. Atil and A. Amirat (2013). Towards a Generic Reconfigurable Framework for Self-adaptation of Distributed Component-Based Application. *In the book Modeling Approaches and Algorithms for Advanced Computer Applications, A. Amine et Al. (Eds), series SCI (Studies in Computational Intelligence)*, Vol. 488, Springer Ed., pp. 399-408.

[27] D. Garlan and B. Schmerl (2002). Model-based adaptation for self-healing systems. *In Proceedings of the first workshop on Self-healing systems*, ACM Press, Charleston, South Carolina, pp. 27-32.

[28] O. Aissaoui, A. Amirat, F. Atil (2012). An Adaptable and Generic Fault-Tolerant System for Distributed Applications. *In Proceedings of the International Conference on Advanced Computer Science Applications and Technologies (ACSAT)*, pp. 161-166.

[29] S.W. Cheng, A.C. Huang, D. Garlan, B. Schmerl and P. Steenkiste (2004). Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer, vol.* 37, no. 10, pp. 46–54.

[30] T. Batista, A. Joolia, and G. Coulson (2005). Managing Dynamic Reconfiguration in Component-Based Systems. In *EWSA'05: 2nd European Workshop on Software Architecture*, Pisa, Italy, pp. 1-17.

[31] H. Tajalli, J. Garcia, G. Edwards, and N. Medvidovic (2010). PLASMA: a Plan-based Layered Architecture for Software Model-driven Adaptation. *In Proceedings of the 25th IEEE/ACM Int'l Conf. Automated Software Eng, (ASE 10)*, IEEE CS Press, pp. 467-476.

[32] A. Colmerauer and P. Roussel (1992). The birth of Prolog. *In the second ACM SIGPLAN conference on History of programming languages*, pp. 37-52.

[33] Y. Brun, G.M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, M. Shaw (2009). Engineering Self-Adaptive Systems through Feedback Loops. *Software Engineering for Self-Adaptive Systems*, Springer-Verlag, Berlin, Heidelberg, pp. 48-70.

[34] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven (2006). Using architecture models for runtime adaptability. *Software IEEE* vol. 23, no. 2, pp. 62-70.

[35] J. Cano Romero and M. García-Valls (2013). Scheduling component replacement for timely execution in dynamic systems. *Software practice and experince*, doi: 10.1002/spe.2181