Towards Making WSRF Based Web Services Strongly Mobile
Soumaya Marzouk
Pages 12 - 24

Towards Making WSRF Based Web Services Strongly Mobile

Soumaya Marzouk
ReDCAD Laboratory
National Engineers School of Sfax
BP 1173, 3038 Sfax Tunisia
Soumaya.Marzouk@redcad.org

Abstract: *Grid Services became a widely used technology for building service oriented applications. The Web Service Resource Framework (WSRF) is the current standard used for building such services as it describes the way to design and communicate with stateful services. However, deploying Web services on a grid environment raises many challenges such as adapting the Web service to the dynamic change of grid resources performance and availability. Therefore, mechanisms such as service mobility maybe very helpful for supporting autonomic properties of grid services.In this paper, we propose a solution for WSRF grid services strong mobility allowing services to migrate during their execution while keeping their execution state consistent. In addition, suspended services will resume their execution starting from the interruption point. To show the feasibility of our approach, we present a case study illustrating the application of our transformation on a WSRF Auction service. We provide then some evaluations performed on the Globus Toolkit (version 4.0).*

Keywords: *Grid Service, WSRF, Strong Mobility, self-Adaptivity.*

Received: August 30, 2008 | ***Revised:*** September 30, 2009 | ***Accepted:*** December 25, 2008

1. Introduction

Service-Oriented Architectures (SOA) provide integrity and interoperability for independent and loosely coupled services. Grid environments offer powerful infrastructures for executing heavy user applications. The integration of these two technologies requires the definition of new standards especially for describing stateful Web services. In this context, Web Service Resource Framework (WSRF) has been defined by OASIS (Organization for the Advancement of Structured Information Standards) as a collection of standards [15] ”Defining an open framework for modeling and accessing stateful resources using Web services”.

However, deploying a Web service on a Grid infrastructure raises new challenges. Indeed, grid environments have volatile resources and changing performances. This is due to the large number of resources distributed over virtual organizations. Indeed, resources can join and leave the system at any time during service execution. Moreover, resource performances can decrease suddenly as result of network problems or owner resource use. Therefore, the execution of Web services in such conditions requires supporting autonomic properties such as self-adapting, self-configuring and self-optimizing. Adding such properties is essential to handle unexpected environment changes and to ensure the correct execution, continuation and availability of the service while offering the expected quality of service (QoS).

In this context, several solutions were proposed in order to ensure the adaptation of Web services. These solutions are based either on service substitution [1, 3, 13] or on service weak migration [11, 10, 9] (i.e. migrating the Web service to a new node with its current state). However, these solutions do not ensure the consistency of the service state during the adaptation process. On the one hand, substitution cannot be applied for stateful services since it is completely unaware of the service state. On the other hand, the use of weak mobility for stateful services requires additional treatments to ensure their consistencies. In fact, weak mobility consists in

migrating the service with its state without taking into account the progress state (WSRF resources) of the running instructions. Thus, migration of running services involves the re-execution of the interrupted invocation which affects the consistency of its state. Therefore, using such solution for adapting statefull services requires waiting until the termination of the execution of all invocations before launching the adaptation process. Such a solution delays the migration and can involve service unavailability or state loss.

In this paper, we overcome these difficulties by proposing a transparent solution for WSRF service adaptation based on strong mobility of services. In fact, strong mobility consists in the migration of a running entity to a destination host where it execution will be resumed starting from the interruption point on the initial host. Our solution relies on service source code instrumentation and makes use of mechanisms for saving a consistent copy of the whole service execution state. The latter includes first the progress state of its interrupted invocations, second its resource values, third in-transit communications, and finally the information needed to re-establish its external dependencies. Thus our approach is applicable only when service source codes are available and not for legacy services. Using our approach, WSRF services can be migrated, at any execution time, by interrupting its execution, capturing its state and moving it to a new host. In the destination host, the service will be deployed, it will re-establish its state, and will resume its interrupted invocations starting from the interruption points. As a result, this solution guarantees instantaneous execution of the reconfiguration while ensuring the consistency of service states. Strong mobility of WSRF services may be initiated by the service it self or by any other entity when detecting that the performances of the hosting node is decreasing or when predicting a future failure. Thus, performance will be enhanced, especially for long running services, since, after migration, already executed parts don't need to be re-executed.

In the following of this paper, we first present related work in Section 2. Then we move to Section 3 which gives an overview of Web services, grid services as well as the WSRF standard. A detailed description of the proposed architecture for strong mobility of stateful grid services is described in Section 4. Thereafter, we suggest, in Section 5, a case study illustrating and evaluating the transformation of a WSRF Auction service into a strongly mobile one. Finally, we conclude and discuss future works in Section 6.

2. Related Work

To our knowledge, there is no work providing a solution for service strong mobility neither for Web services nor for grid services. However, we can mention some solutions dealing with weak mobility of WSRF services. These solutions suppose that migration does not occur during service invocation.

Messig et al. [10] propose a solution for WSRF service weak mobility. This solution maintains the state of every deployed service and adds a broker between the client and the service to ensure self-healing properties. To invoke a service, the client invokes a broker interface which, in turn, invokes the service and returns the result to the client. Before invocation, the broker checks the service to verify its availability. In case of a failure, a new service is deployed using the captured service state. Then, the broker will correctly invoke the service. Thus, migration can occur only before service invocation and never during service execution.

Reich et al. [11] suggest a decentralized solution for WSRF services weak mobility in a P2P environment. In this approach, migration is done to satisfy Service Level Agreements (SLA). Thus, services are deployed or un-deployed whenever a problem occurs. Unfortunately, this work does not provide any solution for managing migrations which are initiated during service execution.

Meehean et al. [9], in their part, presents a weak mobility solution for stateful services and proposes a checkpointing mechanism for saving the service state in an independent platform format. This solution is applied to the Condor scheduler and allows its migration after the termination of all invocations. This approach delays the service migration which may lead to service state loss in case of node crash or disconnection.

Throughout these solutions, we can notice that works dealing with migration of stateful services offer support only for weak migration. This approach leads, in most cases, first to state inconsistency, if migration occurs during service invocation, and second to a large overhead when migrating long running services. Our solution overcomes these shortcomings by offering strong mobility supports for WSRF services. It allows stateful Web services to migrate while ensuring that already executed instructions don't need to be re-executed after migration. Hence, we guarantee resource consistency after Web service migration. As well as a highly reduce of the reconfiguration overhead for long running services.

3. Background on Web & Grid Services

As defined by the W3C [14] " *A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable*

format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards".

In general, simple Web services are stateless. This means that a Web service does not keep a state from one invocation to another. However, grid applications, generally, require statefulness. Therefore, deploying Web services in a grid infrastructure has led to putting on a set of specifications (WSRF) [15] describing how to build, manage and communicate with a stateful Web service while keeping services and states completely separate.

A grid service can be defined as a Web service that conforms to a set of conventions, like service lifetime management, inspection, and notification of service state changes. All these conventions are specified within the Web Services Resources Framework (WSRF) which collects five different specifications: WS-Resources [4], WS-ResourceProperties [6], WS-ResourceLifetime [12], WS-ServiceGroup [8] and WSBaseFaults [7]. Besides, building a Grid service involves other specifications which are not a part of WSRF, but are closely related to it such as WS-Notification [5] (OASIS) and WSAddressing [2] (W3C).

The main WSRF specification is WS-Resources. The latter defines a stateful service as a couple of a Web service and a set of resources. The definition of a stateful resource may be referenced by the Web service WSDL to enable well formed queries against the state of a WS-Resource. So, the state of the WS-Resource can be queried and modified via Web service message exchanges. In addition, a WSRF service is addressed via the WS-Addressing specification. The latter provides mechanisms to address WSRF services via Endpoint References (EPR). In particular, an EPR is inserted into the SOAP header in order to convey end-to-end message characteristics including addressing for source and destination endpoints, message identity as well as the identifier (key) of the resource needed to execute an invocation.

4. Our Approach of WSRF Service Strong Mobility

WSRF service strong mobility should enable a running service to be interrupted, to be migrated to another host, and then correctly resumed starting from the interruption point. Hence, the whole state of the service should be captured, migrated, reloaded into the new host, then interrupted invocations should be resumed, and results should be returned to the client.

Our approach provides these functionalities while ensuring both instantaneous interruption of a service and its state consistency. The suggested solution relies mainly on service source code transformation to enable the capturing and the reestablishing of the service state. Moreover, additional services are deployed in order to manage the mobility process.

In the next subsections, we will detail our solution architecture allowing service strong mobility. We explain the WSRF service strong mobility process including the capture and the re-establishment of the service state. Finally, we present service transformations required to enable service migration.

4.1 Architecture Supporting WSRF Service Strong Mobility

Our architecture, allowing WSRF strong mobility, consists mainly in three entities, which are the mobile service, the WSIM, and the MMS.

The ***Mobile Service*** corresponds to the source code transformation result of the original WSRF service. It provides the same methods and functionalities as the original service, but has, in addition, the ability to capture and re-establish its state as it will be detailed in the next sections.

The ***Web Service Invocation Manager (WSIM)*** is deployed between the client and the mobile service. It plays the role of a virtual interface offering the same operations as the mobile service to be invoked. So, the client code will invoke the WSIM operation instead of the real service one. After that, the WSIM, in turn, invokes the corresponding mobile service method.
The main functionality of the WSIM is to handle service resumption after migration. Indeed, in such a case, the WSIM will transparently re-invoke the interrupted operation of the mobile service deployed in the new host, receives the response and returns it to the client. Moreover, the WSIM has also to manage in-transit invocations during service migration as well as service subscriptions and notifications.
It should be stressed that the WSIM is not supposed to be mobile. Thus, it has to be installed on a reliable node which can be the cluster coordinator in a grid cluster. This coordinator should be reliable and dedicated for grid management.

The ***Mobility Manager Service (MMS)*** is a WSRF service responsible for the service mobility execution by saving the service state (i.e. data required to resume the service execution after migration), in order to ensure correct resumption after migration. Moreover, the MMS ensures the consistency of the service state (i.e. the interruption of the service at this execution moment will not affect the service resource values) before its migrating to the destination host.

In Figure 1, we show the interaction between the client, the WSIM and the mobile service when no mobility occurs. First, the client invokes the WSIM in order to create a new resource (1) or simply to execute a service method (5). Thereafter, the WSIM invokes the real mobile service (6) and remains waiting for the service response (7). If the service invocation corresponds to a resource creation (2), the service returns the EPR of the new WS-Resource (3). In this case, the WSIM updates this EPR by putting its URL instead of the original service URL and returns it to the client (4). Otherwise, the WSIM responds to the client without changing the service response (8).

4.2 WSRF Service Strong Mobility Process

The main steps involved in the mobility process of a WSRF service are capturing and re-establishing its execution state.
In this section, we define a WSRF service state and we detail our approach which allows to capture and then to re-establish this state.

4.2.1. WSRF Service State

The state of a WSRF service represents a set of service information, required to ensure its instantaneous interruption, its migration and its correct resumption starting from the interruption point. In particular, a WSRF service state is constituted from (1) the set of its current resources, (2) the progress state of its interrupted invocations and in-transit communications, and (3) its external dependencies.

While resources are clearly defined in a WSRF service and can be simply saved and migrated towards other nodes, the progress states of invocations are internally controlled by the service and cannot be directly accessed. In fact, invocations progress state is entirely defined by the values of its input parameters, its local variables and the position of the next statement to be executed. Another part of the service state corresponds to in-transit communications. The latter represent messages and requests sent to the mobile service during its migration. In addition, a service state should also include a representation of its external dependencies. In our approach, we consider subscription/notification dependencies which are defined in the WSRF specification. Other types of dependencies like choreography and orchestration are out of the scope of this paper.

4.2.2. Capturing WSRF service State

When service mobility is launched, the WSRF service state is captured. To this purpose, (1) all executing invocations are interrupted and their progress states are saved within the local MMS, (2) the service resources are stored and sent, and (3) all in-transit communications are intercepted by the WSIM and stored.

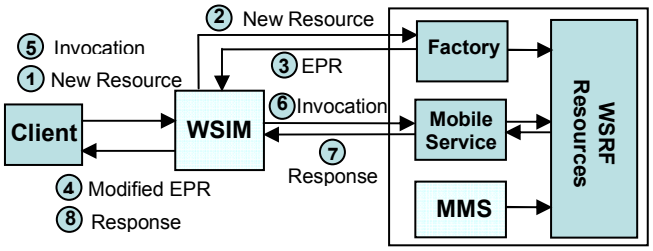

Figure 1. Architecture supporting WSRF Services Strong Mobility.

In our solution, interrupting and capturing invocations progress states are done by means of service source code transformation (see section 4.3). Thereby, when the mobility process is launched, all executing invocations are interrupted and their states are captured. The MMS is then called to create, for every interrupted invocation, a new resource storing its progress state. Thereafter, the MMS returns the EPR of the created resource. This EPR will be thrown to the WSIM, within a fault, informing it about the interruption of the execution.

Once all invocations are interrupted, the service resources become consistent and ready to be migrated. At this moment, service resources (as well as the MMS resources) are stored in an independent platform format (Resource Files) and sent to the destination host.

In addition, service mobility should preserve external dependencies. This issue concerns subscriptions and notifications of the mobile service.
Concerning subscriptions, service migration should save the set of couples (subscriber, topic) to continue notification sending after migration. We solve this issue through the WSIM. In fact, all subscription requests pass through the WSIM which stores them, and then requests clients' subscriptions to the mobile service.
Besides, for recovering in-transit notifications, which come to the service during its migration, we use also the WSIM (see Figure 2). In fact, the mobile service will subscribe to the notification producer through the WSIM. Thus, all incoming notifications go through it before reaching the mobile service. And, when the mobility process is launched, the mobile service will unsubscribe from the WSIM. Then, all in-transit notification will be stored in it and will not be transmitted to the mobile service.

4.2.3. Re-establishing WSRF service State

After receiving the service resources files, the mobile service, deployed on the new host, explores these files in order to re-create its resources. Then, it notifies the WSIM about the migration termination. Consequently, the re-establishment process begins.
First, the WSIM starts subscriptions reestablishment using the set of stored couples (subscriber, topic). Then, it launches invocation re-establishment by re-invoking interrupted ones using the initial client

request. This request is transformed by the addition of an input parameter representing the EPR of the corresponding interrupted invocation progress state, which is stored within the MMS. After that, thanks to source code transformation, every interrupted method resumes its execution starting from the interruption point (see section 4.3).

Simultaneously, the WSIM invokes the mobile service to convey in-transit service requests which have been sent during service migration. Moreover, it sends in-transit notification to the mobile service by invoking an added method called ”sendNotification”. The latter has the same code executed by the mobile service when receiving a notification.

The whole mobility process is illustrated in Figure 3. The latter shows interactions between different entities to enable the WSRF service migration. Hence, the mobility process is fully transparent, since the client invokes the service and receives the response without being aware about the service mobility process.

4.3 Source Code Transformation

In order to achieve the service transformation into a strongly mobile entity, we define transformation rules which will be applied for the service methods and WSDL. We will also detail the structure of the additional services which should be deployed to manage the mobility process.

4.3.1. WSRF Service Code Transformations

Every service method should be transformed in order to enable capturing and reestablishing its invocation state.
These transformations include position management, state capturing, and re-establishing.

Position management is necessary to maintain the position of the next instruction to be executed. This matter enables the resumption of the method execution starting from the interruption point.
In order to achieve this purpose, every service method is instrumented by adding a local variable *”position”* representing the position of the next instruction to be executed. Moreover, method instructions should be instrumented to update this position. Thereafter, we classify code instructions into two categories:

- (1) Simple instructions: they are elementary instructions, which include assignments, inputs/outputs instructions, etc.
- (2) Composed instructions: they are blocs of code containing loops or control structures.

(1) Simple instructions transformation serve to ensure execution state updating and execution resumption while preserving execution semantics. For that goal, after each instruction execution, the value of the next instruction position to be executed must be updated. Therefore, after each instruction of the transformed code, we propose to increment the position value of the next code instruction to be carried out.

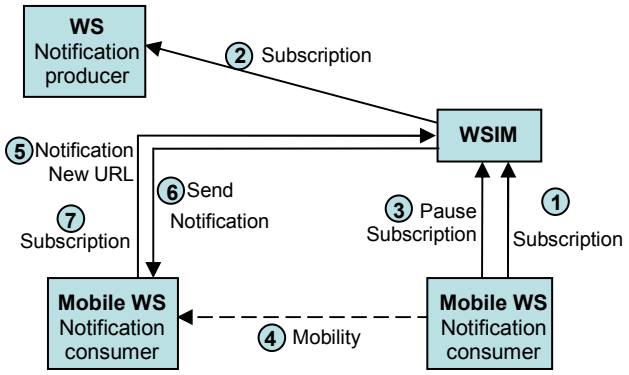

Figure 2. Handling in-transit Notification.

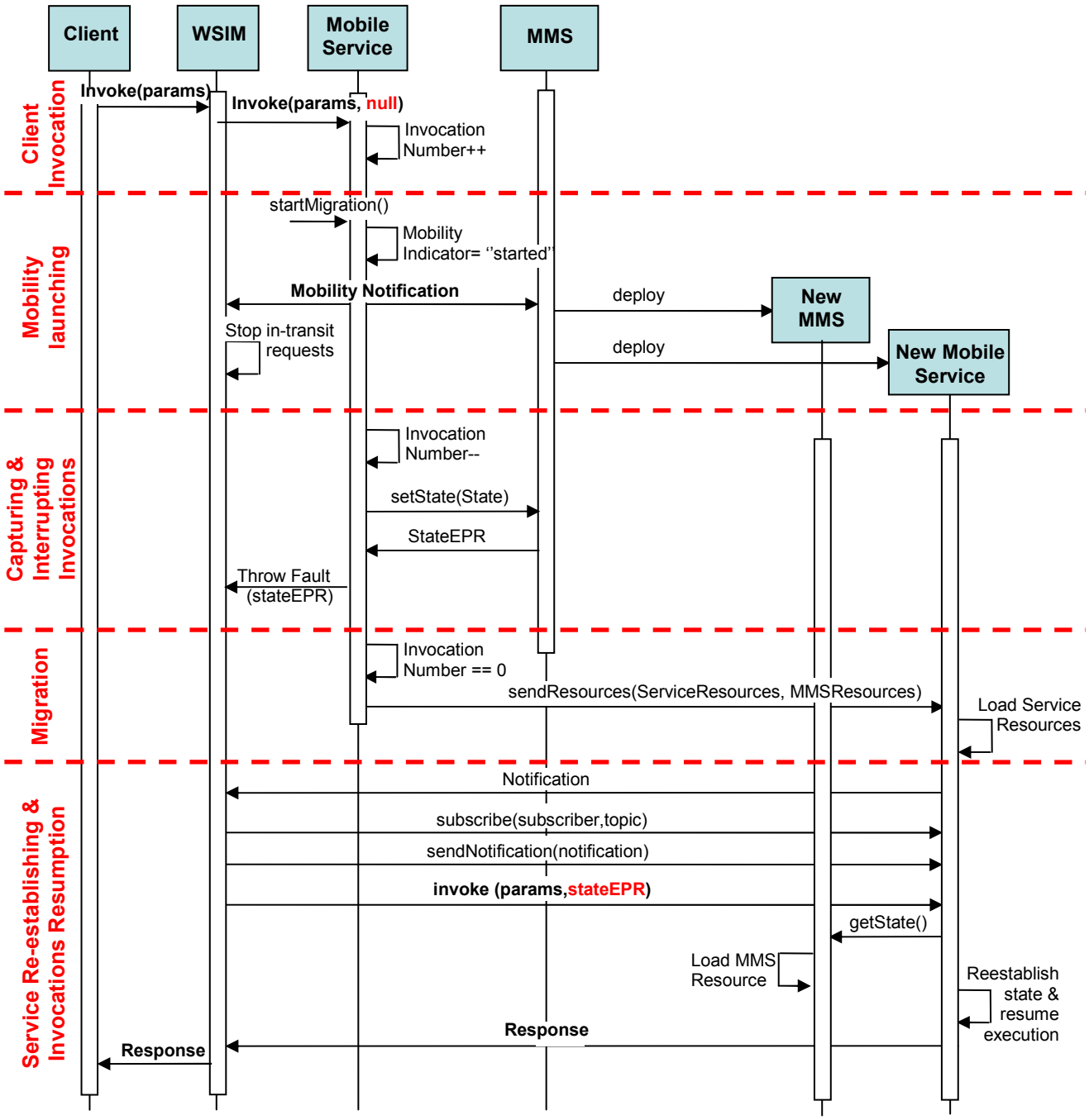

Figure 3. WSRF Service Strong Mobility Process.

In addition, to ensure resuming process execution after migration, every code instruction must be supervised by testing the value of its position. Consequently, instruction1 in the original method code will be replaced with:

```
if (position==current position) {
        instuction1;
        position++;
}
```

Besides, we must be sure that the execution interruption will not take place after the instruction execution and before the position update. Therefore, we propose to consider the transformation result of an instruction as an atomic operation which can't be interrupted by migration. The transformation of the instruction1 will be as follows:

```
lock_migration();
if ((position==current position) and
(MobilityIndicator ==”inactivated”))
{
        instuction1;
        position++;
}
unlock_migration();
```

“MobilityIndicator” refers to an added resource of the transformed service. This resource is set initially to “inactivated”. And when migration is required (by the mean of the invocation of a method, added to the service code, called ”startMigration”), this resource will be set to “started”. In this way, mobility can be

launched only at the beginning of an atomic operation.

Thus, these transformations applied for simple instructions, guarantee method execution state updating, as well as re-establishing after migration, while preserving its execution semantics.

(2) The difficulty which arises for the case of loops and control structures is the update of position of next instruction to be carried out.
In fact, the code transformation has to preserve the execution semantics, whatever the interruption position is, during loop or control structure execution.
Next, we will study the case of the following structures: "while(condition) Bloc;" (see Figure 4) and "if(condition) Bloc1; else Bloc2;" (see Figure 5).

```
while ((position >= inPc(Bloc_transformed)) && (position
<= outPc(Bloc_transformed))) {

    if (position == inPc(Bloc_transformed) && !condition)
    {
    // condition not verified
        position = outPc(Bloc_transformed)+1;
        break;
    }
    Bloc_transformed;
    if (position == outPc(Bloc_transformed))
        position = inPc(Bloc_transformed);
}
```

Figure 4. Transformation of a while loop.

```
if (((position >= inPc(Bloc1_transformed)) && (position
<= outPc(Bloc1_transformed))) ||
(position == Pc(If) && condition)) {

    Bloc1_transformed;
    if (position == outPc(Bloc1_transformed)) {
        // end of the bloc if: jump the bloc else.
        position = outPc(Bloc2_transformed) + 1;
    }
} // if the condition is not verified: enter to the bloc else
else {
    if (position== inPc(thisIf))
        position = outPc(Bloc1_transformed) + 1;
    }

if ((position>= inPc(Bloc2_transformed)) && (position <=
outPc(Bloc2_transformed))) {
    Bloc2_transformed;
}
```

Figure 5. Transformation of an if - else structure.

With:

- Bloc1_transformed represent the transformation result of Bloc1.
- inPc(Bloc_transformed)represent the first position in Bloc_transformed
- outPc(Bloc_transformed) represent the last position in Bloc_transformed.
- Pc(if) represent the position of the "if" instruction. Indeed, we attribute to the "if" instruction a position to ensure that the "if" condition will be evaluated only once.

The code given above preserves the initial semantics whatever the execution stop point in this code.

These transformations can be optimized. In fact, we propose to affect a position number for blocs containing more than one instruction. Thus, an instructions bloc, with the update instruction of its corresponding position, will form an atomic operation during which migration is not authorized. This makes it possible to reduce the size of the code added compared to the initial code, and consequently to reduce the execution time of the transformed method. This modification requires several rules for the choice of blocs.

First, blocs should not contain the headings of controls structures or of loops of the original code. This case can generate compilation errors, since it causes crossed loops.

Second, the bloc size must be quite selected not to be, neither too large causing the delay of the migration operation, nor too small causing the increase of the size of the generated code compared to the original one.

We propose also another optimization, which consists in not applying transformations concerning loops and control structures in all cases. Indeed, if the code carried out by a loop or a control structure is simple (without nested structures), we propose to assign to this structure only one position number, and thus to authorize the migration only at the end of the execution of all the structure code. For the case of loops, this solution remains valid if the total number of instructions to be carried out by the loop is not very large. Otherwise, in general cases, we propose to allow the migration at the end of each iteration.

The second transformation applied to methods code allows its ***state capturing***. This transformation consists in adding a capture bloc to the end of each service method. This bloc will be executed only if the migration resource is set to "activated". Indeed, in such situation, all instructions, which occur after mobility launching, will not be executed since their entry condition will be not verified. Thus, all these instructions will be jumped until reaching the capture bloc which will be executed. This bloc will contacts the MMS to save the invocation progress state and reduces by one an added resource called invocation number.

The *"InvocationNumber"* resource counts the number of current not yet interrupted running invocations. Thus, when its value becomes zero, resources are consistent and can be saved and moved to the destination host.

Finally, the last transformation aims to enable the method ***state re-establishing***. In order to achieve this goal, a re-establishment bloc is added to the beginning of each method. This bloc will contact the MMS to load the captured state. The latter creates a

new resource using the received resource files storing the original host MMS resources. Thereafter, the re-establishment bloc uses the captured state to re-initialize the methods input parameters, local variables and position of the next instruction.
This bloc is executed only if the invocation corresponds to an interrupted one. Thus, an input parameter, representing the identifier of the interrupted invocation (EPR of the MMS resources saving it), is added to each method. In case of invocation executed for the first time, this identifier will be set to "null". Otherwise, this identifier will serve to get the correct execution state from the MMS.

Moreover, transformations done on the service code should be propagated to its ***WSDL*** description to ensure the correct service definition. Thus, the service WSDL should be updated by (1) adding the definition of the mobility and invocation number resources, (2) adding the definition of the "startMigration" and "sendNotification" methods, and by (2) transforming methods input parameters.

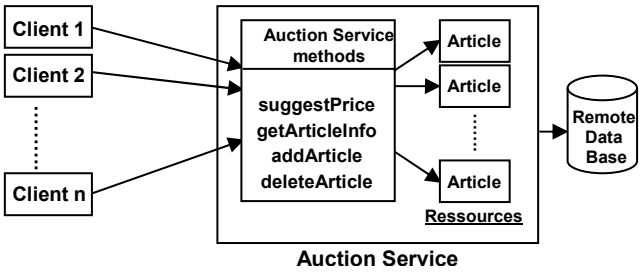

Figure 8. Auction service Architecture.

It should be noted that the transformation of the service WSDL is transparent to the client who can only see the WSIM's WSDL, which corresponds to the original service WSDL.

4.3.2. Additional Services

The **WSIM** service has to ensure service re-invocation in case of mobility. Thus, it offers to the client the same operation as the original service. And every method of the WSIM invokes its correspondent belonging to the mobile service. This invocation should be placed in a try/catch bloc in order to capture exceptions showing the service mobility. Moreover, service mobility may occur several times during the same invocation. For that goal, WSIM methods could handle several mobility exceptions. Figure 6 shows an example of a WSIM method corresponding to the *"operation"* method of the mobile service.
The **MMS** service has mainly to save the invocation progress state of each interrupted invocation. Forethat, we design the MMS as a WSRF service having the clients invocation progress state as resources. So the resource structure of the MMS, as presented in Figure 7, is constituted of the mobile service methods input parameters, its local variables, and the position of the next instruction.

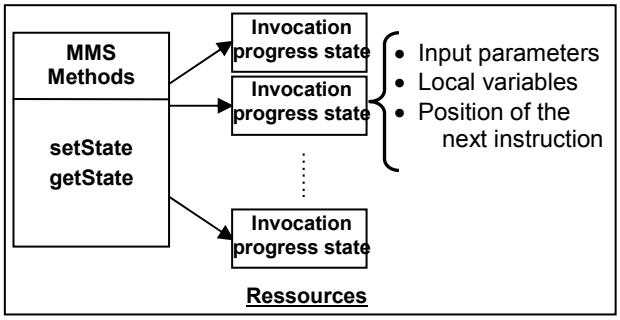

Figure 7. The MMS Structure.

5. Case Study: Auction Service

In this section, we introduce a case study showing the feasibility of our approach of WSRF services strong mobility. Therefore, we use a WSRF implementation of an Auction service to illustrate the service transformation and to evaluate the overhead induced by it. In the following, we describe the considered Auction service architecture according to WSRF. We describe how we transform it into a strongly mobile one. Finally, we measure the overhead.

5.1 Auction Service Description

We consider an Auction service exposing a set of articles for which clients may suggest prices in order to buy one of them. The Auction service accepts always the highest proposed price for each article. The architecture of this service according to WSRF is presented in Figure 8.

The Auction service maintains the list of available articles as resources. Every one contains the article identifier and characteristics, the maximum obtained price and the reference of the corresponding client. Furthermore, clients may interact with the service through a set of methods which are "suggestPrice", "getArticleInfo", "addArticle", "deleteArticle"...
In this case study, we focus on the "suggestPrice"

```
public operation(OperationParams inputParams)
{
    EndPointReference  ServiceEPR;
    EndPointReference MMSEPR=null;

    while(true)
    {
        try{
            ServiceEPR= inputParams.setEPR(EPR);
            //create reference (MS) to the mobile
            service using serviceEPR
            inputParams.setEPR(EPR);
            MS.operation(inputParams);
            break;

        catch(MobilityException me) {
            ServiceEPR=me.getNewServiceEPR();
            MMSEPR=me.getNewMMSEPR();
        }
}
```

Figure 6. WSIM Method Example.

method which takes as input parameters: the new

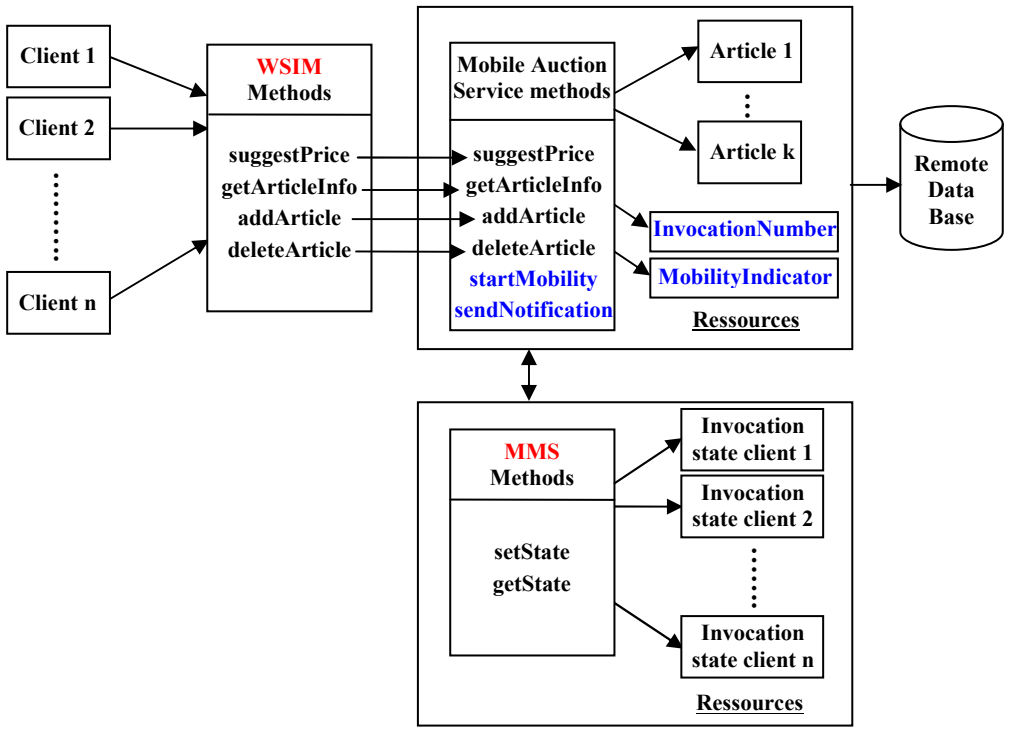

Figure 10. Architecture Supporting Auction Service Strong Mobility.

suggested price, the client identifier and his password. When invoked, this method updates the article price and the client reference. This update is constrained by two rules: The first one states that the suggested price must be greater than the current one, and the second rule ensures that the client exists in the remote database and is correctly authenticated. Thereafter, the request is logged in the remote database.

Another characteristic of the Auction service is it ability to produce notifications. So, client may subscribe to a given article in order to be notified whenever its price changes.

5.2 Transformation of the Auction Service into Strongly Mobile Service

Let's consider a scenario in which the host performances, where the Auction service is deployed, critically decrease. In this situation, there is a risk of possible host crash. So, the service has to be migrated in order to save its resource values. This action should be done instantaneously to avoid possible resource loss. In a weak mobility scenario, migration will lead to interrupting and re-executing the auction service invocations, which may cause resource inconsistency. Indeed, as shown in Figure 4, if the "suggestPrice" method execution is interrupted after the price update and before the client reference update, than the re-execution of the invocation will not satisfy the price constraints. As a consequence, all remaining instructions of the "suggestPrice" method will not be carried out. On the one hand, this matter will involve an inconsistency at the service resources level (as the client reference is not updated). And on the other hand, the information logged in the data base will be incomplete.

However, using strong mobility resolves completely this inconsistency, because interrupted invocations resume starting from their interruption points. The result of our transformation applied to the Auction service can be shown in Figure 10.

These transformations, allowing the generation of a mobile Auction service, include:

- Adding the WSIM which has the same methods as the original Auction service.
- Adding the MMS which provides methods to manage the Auction service invocation states, in addition to a set of resources storing the state of every interrupted invocation to be resumed in this host.
- Transforming the Auction service code by adding first the "MobilityIndicator" and "InvocationNumber" resources, second the "startMobility" and "sendNotification" methods, and finally by transforming every method of the service to enable capturing and re-establishing its execution state.

```
public SuggestPriceResponse suggestPrice (double
newPrice, String clientId, String clientPass) {

    String password;
    if ( newPrice > price) {
        // connect to the DB
        // get client password from DB
        if (clientpass.equals(password)) {
            price = newPrice;
            client = clientId;
            // log in DB (date, Article, price, client)
        }
    }
    return new SuggestPriceResponse();
}
```

Service Interruption

Figure 9. Initial Code of the *"suggestPrice"* method of the Auction service.

```
public SuggestPriceResponse suggestPrice (double newPrice, String clientId, String clientPass, EndPointReference EPR) {

        int position=1;
        String password;
        if (EPR!=null) {
                //get state to resume execution
                // MMS is a reference to the Mobility Manager using EPR
                State s = MMS.getState();
                newPrice = s.newPrice;
                clientId = s.clientId;
                clientPass = s.newClientPass;
                password = s.password;
                pos = s.pos;
        }

        InvocationNumber ++;
        if (( position<8 && position>=1) || (position==1 && newPrice > price)) {
                if(position==1)  position =2;
                if (!mobilityIndicator.equals("started")) {
                        // connect to the DB
                        position ++;
                }
                if (position ==3 && !mobilityIndicator.equals("started")) {
                        // get client password from DB
                        position ++;
                }
                if ((position >=5 &&  position  < 8) || (position ==4 && clientpass.equals(password))) {
                        if (position ==4)  position  = 5;
                        if (position  == 5 && !mobilityIndicator.equals("started")) {
                                price = newPrice;
                                position ++;
                        }
                        if (position ==6 && ! mobilityIndicator.equals("started")) {
                                client = clientId;
                                position ++;
                        }
                        if (position ==7 && ! mobilityIndicator.equals("started")) {
                                // log in DB (date, Article, price, client)
                                position ++;
                        }
                } else if (position ==4 && !clientpass.equals(password))  position  = 8;
        } else if (position ==1 && !(newPrice > price)) pos = 8;

        invocationNumber--;
        if (mobilityIndicator.equals("started")) {// save the state of the current invocation

                if (invocationNumber==0)
                mobilityIndicator="ready";
                State s = new State(newPrice, clientId, clientPass, password,  position);
                stateEPR = MMS.setState(s);
                // throw fault (stateEPR)
        }
        return new SuggestPriceResponse();
}
```

Figure 11. Transformation result of the *"suggestPrice"* method.

Figure 11 presents the transformed code of the "suggestPrice" method. This code is written according to the Globus toolkit (version 4) implementation of the WSRF API, but it is simplified for clarity reasons.
Using this transformation allows the mobility of the Auction service while ensuring the integrity of its states stored in the service resources and also in the database.

5.3 Implementation & Evaluation

At the present, we are implementing our WSRF strong mobility approach on Globus toolkit (version 4.0), in order to build a framework allowing automatic transformation of WSRF service into strongly mobile one.

We have already done the transformation of the Auction service into a strong mobile WSRF service. Indeed, we applied the source code transformation rules enabling capturing and re-establishing the mobile service state.

In addition, we implemented the necessary additional services (i.e. the WSIM and the MMS). In order to ensure resources persistence, we exploited the Persistence API of Globus 4.0. Thus, the implementation of the Auction and the MMS service resources extend this API. Besides, for each service we implement "store" and "load" methods to enable managing resource persistence. Thus, MMS resources are loaded when the re-establishing bloc of the corresponding invocation is executed. Whereas, the Auction service resources are loaded at migration end. In odder to achieve this purpose, we add to the mobile auction service a method which read the received resources file in order to extract and create the migrated auction service resources. This method is invoked after the auction service migration termination.

We provide, in this section, evaluations of the strongly mobile Auction service prototype. This evaluation aims to estimate the overhead of the service code transformation and migration. To reach this goal, we deploy the mobile Auction service on a computer having 3GHz processor and 512 Mo of memory. And we compare the execution time of the transformed ”suggestPrice” method to the original one.

Figure 12 shows the evaluation of the overhead of this transformation when no mobility occurs. The noticed overhead represents the time needed to execute the added code. Besides, this additional time becomes higher when the number of clients whose are invoking the service method increases. However, this overhead stills relatively low compared to the execution time of original service method.

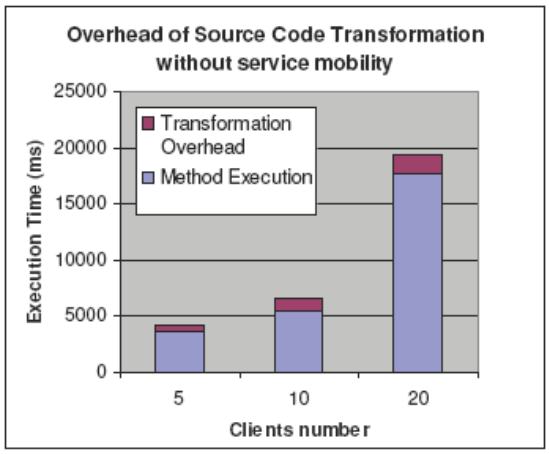

Figure 12. Evaluation of the overhead inferred by our transformation of the ”suggestPrice” method.

Figure 13 illustrates the overhead of the auction service mobility. This overhead could be divided into three types: transformation, capturing and re-establishing overhead. At this step, we do not consider the time needed to transfer the service to a distant host (i.e. all the strong mobility steps are done in the same host).

The capturing overhead corresponds to the time elapsed between the mobility order and the migration initiation. It includes the time needed to store the Auction and the MMS resources in the resource files. As presented in Figure 13, this time is very small compared to the *suggestPrice* execution time. This observation proves the efficiency of our solution since the fast capturing of the service reduces the probability of the hosting node failure before service migration.

The re-establishing step has a higher overhead and it corresponds to the time elapsed between the service transfer beginning and the service resumption. This time includes initializing the Auction and the MMS resources by using the received resource files. This overhead is sensibly high. However, this step is done on the new service host after file transfer termination which ensures the service state consistency. Moreover, this overhead is not influenced by the service code but only by the Auction and MMS resource number and size. So, for long running services, this overhead will have less impact on the overall execution.

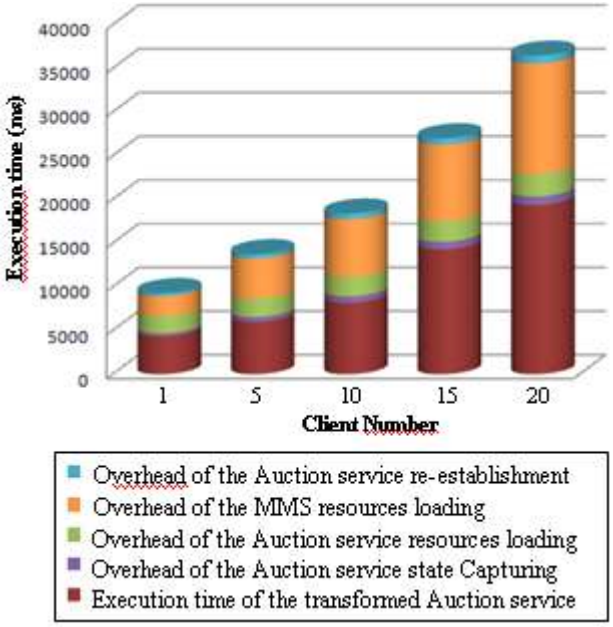

Figure 13. Evaluation of the overhead inferred by the Auction service capturing and re-establishing on the ”suggestPrice” method.

Figure 14 represents the evaluation of the service transfer to the destination host using the GlobusFTP service. This overhead is very large. It reaches 1200 second for 20 executing invocations using one resource.

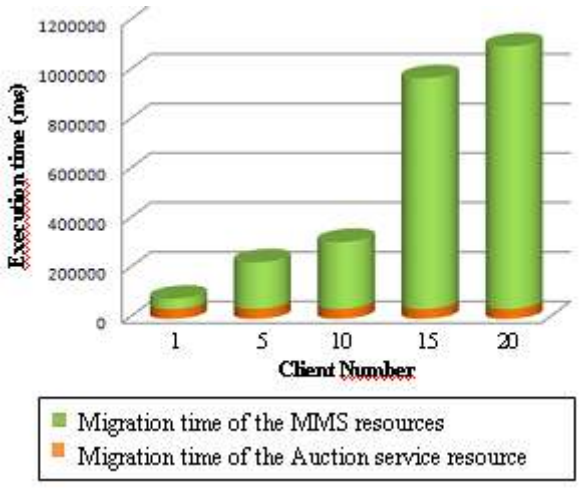

Figure 14. Evaluation of the overhead inferred by the Auction service transfer on the ”suggestPrice” method.

In such a case the overall size of the transferred files is 912 bites and the available bandwidth is 100 Mbits. This huge overhead is caused by the GlobusFTP service which has low performances due to heavy controls required by the RFT[1] service. So this result may sensibly be enhanced when using simpler transfer techniques such as typical FTP or sockets.

[1] Reliable File Transfer: a Globus service ensuring the consistency of file transfer, even in case of interruption. In such case, the transfer will be resumed starting from the interruption point.

6. Conclusion and future work

In this paper, we presented a solution for WSRF services strong mobility. Our solution proposes an architecture allowing WSRF service to migrate transparently while saving the unaccomplished invocations states, the service resources (WS-Resources) and the WSRF service notification dependencies.

Our architecture ensures the automatic un-deployment, migration and deployment of WSRF services. This approach is used to ensure service migration to a more reliable host, in response to node performance degradation or to failure prediction. Thus, migration may be performed at any time even if the service is executing client requests. In these cases, invocations are suspended transparently to the client and resumed at the new service host starting from the interruption point. Thereby, resource consistency is ensured and the high cost of migrating services while executing long running invocation will be sensibly reduced.

At the present, we are implementing the automatization of our transformation targeting Globus toolkit (version 4.0) based WSRF services. In the future, we plan to use this solution to implement a self-adaptive environment for deploying and managing WSRF services.

References

[1] R. Ben Halima, M. Jmaiel, and K. Drira. A QoS-driven reconfiguration management system extending Web services with selfhealing properties. *In Proceedings of the 16th IEEE International Workshops on Enabling Technologies Workshop on Information Systems et Web Services (WTICE)*, pages 339-344, Paris, France, June 2007. IEEE Computer Society.

[2] A. Bosworth, D. Box, E. Christensen, F. Curbera, D. Ferguson, J. Frey, C. Kaler, D. Langworthy, F. Leymann, S. Lucco, S. Millet, N. Mukhi, M. Nottingham, D. Orchard, J. Shewchuk, E. Sindambiwe, T. Storey, and S. Weerawarana. Web Services Addressing (WS-Addressing). W3C , 2004. Available at http://www.w3.org/submission/2004/subm-ws-addressing-20040810.

[3] O. Ezenwoye and M. Sadjadi. TRAP/BPEL: A framework for dynamic adaptation of composite services. *In Proceedings of the International Conference on Web Information Systems and Technologies (WEBIST 2007),* Barcelona, Spain, March 2007.

[4] I. Foster, J. Frey, S. Graham, S. Tuecke, K. Czajkowski, D. Ferguson, F. Leymann, M. Nally, T. Storey, and S. Weerawaranna. Modeling Stateful Resources with Web Services. Globus Alliance, 2004.

[5] S. Graham, D. Hull, and B. Murray. Web Services Base Notification 1.3 (WS-BaseNotification). OASIS Standard, 2006. Available at http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-os.pdf.

[6] S. Graham and J. Treadwell. Web Services Resource Properties 1.2 (WS-ResourceProperties). OASIS Standard, 2006. Available at http://docs.oasis-open.org/wsrf/wsrf-ws_resource_properties-1.2-specos.pdf.

[7] L. Liu and S. Meder. Web Services Base Faults 1.2 (WSBaseFaults). OASIS Standard, 2006. Available at http://docs.oasisopen.org/wsrf/wsrf-ws-basefaults-1.2-spec-os.pdf.

[8] T. Maguire, D. Snelling, and T. Banks. Web Services Service Group 1.2 (WS-ServiceGroup). OASIS Standard, 2006. Available at http://docs.oasis-open.org/wsrf/wsrf-ws_service_group-1.2-spec-os.pdf.

[9] J. Meehean and M. Livny. A service migration case study: Migrating the Condor schedd. *In Midwest Instruction and Computing Symposium*, April 2005.

[10] M. Messig and A. Goscinski. Self Healing and Self Configuration in a WSRF Grid Environment. *In Michael Hobbs, Andrzej M. Goscinski, and Wanlei Zhou, editors, ICA3PP*, volume 3719 of Lecture Notes in Computer Science, pages 149–158. Springer, 2005.

[11] C. Reich, M. Banholzer, R. Buyya, and K. Bubendorfer. Engineering an Autonomic Container for WSRF-Based Web Services. *adcom*, 0:277–282, 2007.

[12] L. Srinivasan and T. Banks. Web Services Resource Lifetime 1.2 (WS-ResourceLifetime). OASIS Standard, 2006. Available at http://docs.oasis-open.org/wsrf/wsrf-ws resource lifetime-1.2-spec-os.pdf.

[13] Y. Taher, D. Benslimane, M.-C. Fauvet, and Z. Maamar. Towards an Approach for Web Services Substitution. In Proceedings of the 10th International Database Engineering & Applications Symposium (IDEAS'2006), Delhi, India, 2006

[14] W3C, http://www.w3.org/TR/ws-arch, 25 December 2008.

[15] OASIS, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf, 25 December 2008.

Soumaya Marzouk, received her Engineering degree in Computer Science in 2005, and her Master in Computer science in 2006 from the National school of engineers of Sfax in Tunisia. Since 2007, she is working towards his PhD at the ReDCAD research unit in the Department of Computer Science in the National school of engineers of Sfax. Her research is focused on self-adaptation of distributed applications, in particular she is interested on software mobility as an adaptation action applied for different application such as Web services and orchestrated Web services.