A Middleware Based on Chemical Computing for Service Execution - Current Problems and Solutions.

Chen Wang

A Middleware Based on Chemical Computing for Service Execution

The Current Problems and Solutions

Chen WANG

IRISA/INRIA, Campus de Beaulieu, 35042, Rennes, France

chen.wang@inria.fr

1 Introduction

1.1 Middleware Overview

This report presents a middleware for service execution that is implemented based on the chemical computing. The global view of the middleware is shown in Figure **??**. As you can see, the middleware is an abstract layer working on the top of the implementation layer. Each service implementation in the real world defines a "chemical service" as its delegate in the middleware. The chemical service, often called as "abstract service", is responsible for all the activities on behave of the delegated service implementation, such as receiving the service invocation, return/forward the computing result, or even executing the workflow. The composition of the middleware and how the different components are collaborated will be discussed in the following sections.

This middleware is implemented by chemical programming that enables it with chemical semantics. All data or resources for the computation are represented by molecules. The computation is abstractly described as chemical reaction process. The *Chemical Middleware Layer* can be seen as a big/global chemical reaction container and the components inside can be regarded as chambers that are separated by membranes. The membrane limits the movement of a molecule within a certain chamber. As a result, each chamber (component) is an independent space for carrying out local chemical reactions. From the viewpoint of computer science, a chamber is an automated system. Inside a chamber, the chemical reaction is carried out in an automated and dynamical way without outside interference.

Each component is implemented by an HOCL program that creates a multi-set as a chamber. The multi-set extends the concept of set with multiplicity. An element can appear only once in a set while it can have more than one instance in a multi-set. Using chemical computing, the implementation of this middleware can be distributed. All these multi-sets can be implemented on different machines that spread over the internet. As

shown in Figure **??**, a component can be implemented on a server, grid, or even a personal computing devices such as laptops.

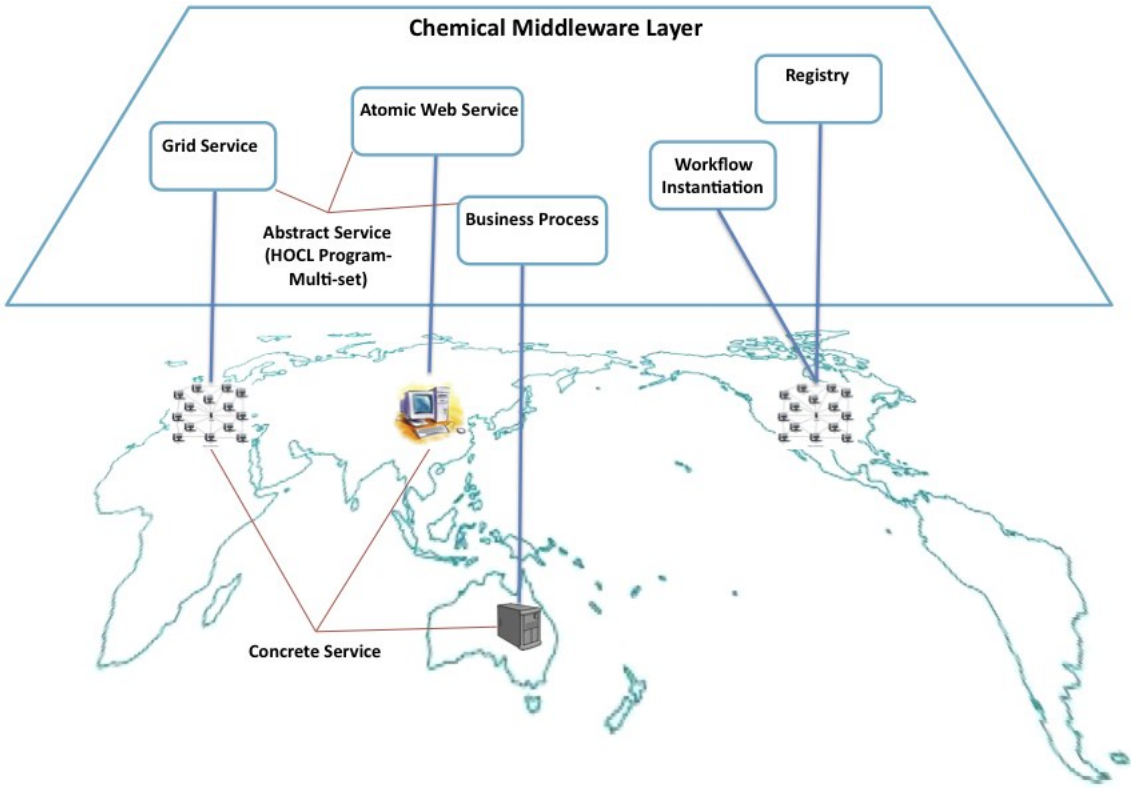

Fig. 1: The Global Schema of Chemical Middleware

A multi-set defines various elements for the computation. Generally speaking, there are two types of elements: data and rules. The data represents the computing resources; the rules defines the law of computation, which is similar to the chemical equation which reflects the law of nature. All the chambers are isolated but interconnected. Some rules are defined to send a set of molecules to the remote chamber (i.e.: when a certain type of element are produced or some conditions are held). This property help us to build the distributed implementation.

1.2 Terminology

Before going into the details, we are firstly going to regulate the terminologies, which will be used in the following part of this report.

- Web Service

Web service is a software system designed to support interoperable machine-to-machine interaction over computer networks. It can be seen as a black box providing certain functionalities to its clients through a set of interfaces. Generally speaking, Web services can be classified into two catalogs based on the different implementation methods: *Atomic Service* and *Composite Service*. The difference between the two is that *atomic service* implement the functionalities directly[1] based on certain programming language. By contrast, a *composite service* fulfills its promising functionalities by designing a workflow that groups and coordinates a set of Web services that are called *Partner Services*. A partner service can be either an atomic service or a composite service.

- Workflow

  A workflow consists of a sequence of connected steps. It is a depiction of a sequence of activities or tasks. Within the Web service domain, each activity is an service invocation. A workflow can be either abstract or concrete. The *Abstract Workflow* only specifies the functional requirement (sometimes together with non-functional requirements) for each activity but does not bind the activities to a certain service provider. By contrast, the *Concrete Workflow* specifies the endpoint reference for each task. In this sense, a concrete workflow is also called an *Execution Plan* because it can be executed by a workflow engine.

- Workflow Instantiation

  An abstract workflow cannot be executed because there is no endpoint reference for partner services. As a result, before the execution, the instantiation process is necessary. Workflow instantiation is to find the appropriate partner service for each workflow activity. The selection criterion can be based on the non-functional requirement on the service composition level.

- Functional Requirement

  In software engineering, a functional requirement defines a function of a software system or its component. A function is described as a set of inputs, the behavior, and outputs. It may be calculations, technical details, data manipulation and processing as well as other specific functionality that define what a system is supposed to accomplish.

- Non-functional Requirement

  Non-functional requirements are often called qualities of a system. In general, functional requirements define what a system is supposed to do whereas non-functional requirements define how a system is supposed to be. Non-functional requirement is

[1] Here, "directly" means within the enterprise domain

always associated with several non-functional properties such as price, response time, availability, security and accuracy.

2 Middleware Architecture

As shown in Figure **??**, the middleware is composed of several kinds of components, such as *Abstract Web Service*, *Registry* and etc. All of these components are grouped into two levels: *Web Service Representation Level* and *Web Service Execution Level*.

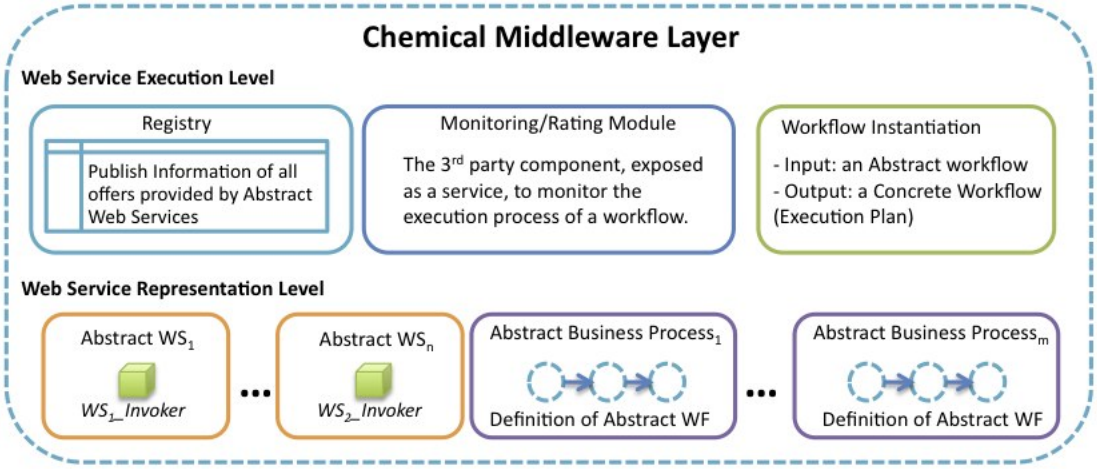

Fig. 2: The Chemical Middleware Components

Web Service Representation Level deals with the problem how to represent Web services in the middleware. In another word, its main task is to define Abstract Services[2]. Meanwhile, the higher level *Web Service Execution Level* takes charge of the execution of service composition, including workflow instantiation, service invocation etc.

2.1 Abstract Web Service

An *Abstract Service* is a description of functional interfaces related to a certain Web service rather than the relative implementation. From the viewpoint of this middleware, an abstract service can be seen as a representative of a concrete service. In contrast to the abstract service, a *Concrete Service* is also called a *Service Implementation*. Thus, in order to perform the real calculation, an abstract service has to group a set of concrete services[3] in order to forward the invocation requests.

[2] Here, abstract service means abstract atomic service and abstract business process

[3] For the current implementation, an abstract service can connect to multiple concrete services. But it is more logical that a chemical service connects only one concrete service. Because the abstract service is

The structure of an abstract service is shown in Figure **??**. The main module is the Connector. It groups a set of invokers to forward the invocation to the relative service implementation. Each invoker connects to a certain service implementations. An invoker is the chemical level reflection of a service implementation. So it declares the same interfaces as the relative implementation. Once a certain operation is invoked, the invoker gets the parameters and passes them directly to the service implementation. As shown in Figure **??**, it works as tubes that connect the chemical level to the implementation level. But moreover, an invoker monitors the non-functional performances for each invocation, such as response time and etc. All those monitored information are stocked in its built-in memory.

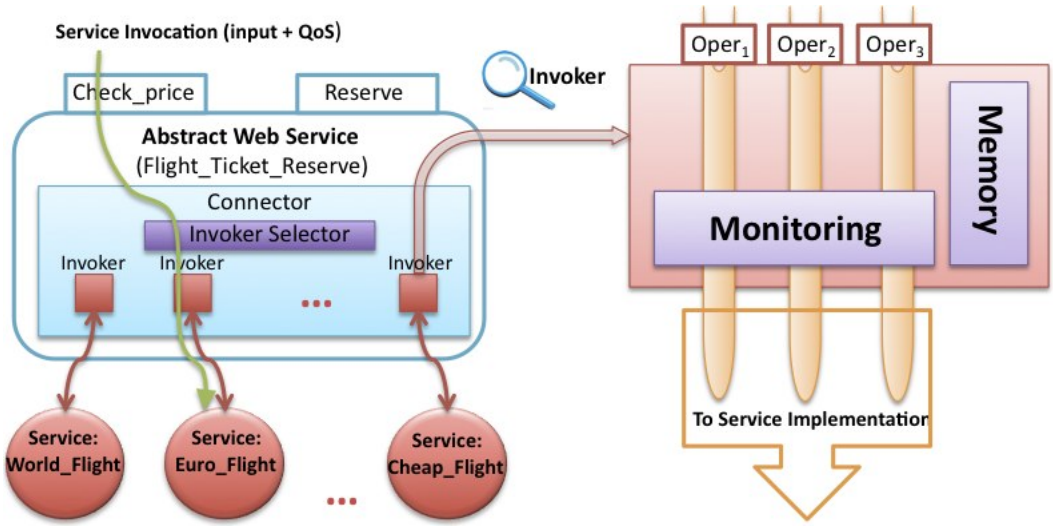

Fig. 3: The Structure of Abstract Service

Once an invocation arrives to the abstract Web service with parameters and QoS requirement, the abstract Web service will select and activate an invoker to forward the invocation. The selection process is done by the Invoker Selector. It manages both the historical and run-time QoS information of each invoker, for example, the average response time and the current charge of a service implementation (waiting queue and etc.). All these information is collected by the invoker after each invocation. Based on the QoS constraints specified in the invocation message, a qualified invoker is selected for passing the input parameters.

For an abstract service, the concrete services that it connects work as black-boxes. It connects to any services that can provide its promising functionalities but does not

generated automatically based on the description of the relative concrete service (i.e. WSDL file). As a result, this point should be discussed later

care how they are implemented. It can connect either an atomic service or a composite one. For example, as shown in Figure **??**, the World_Flight may be an atomic service, the Euro_Flight can be a composite service (that combines all the airline companies from Europe such as Air France and Terminal 1) while Cheap_Flight could be a cloud service.

2.2 Abstract Business Process

The distinguish between the abstract Web service and the abstract business process is that the former interacts with the implementation layer while the latter only interacts within the chemical layer. The abstract business process can be seen as 'composite service" in the middleware level. The detailed structure of an abstract business process is shown in Figure **??**. It is composed of several functional modules.

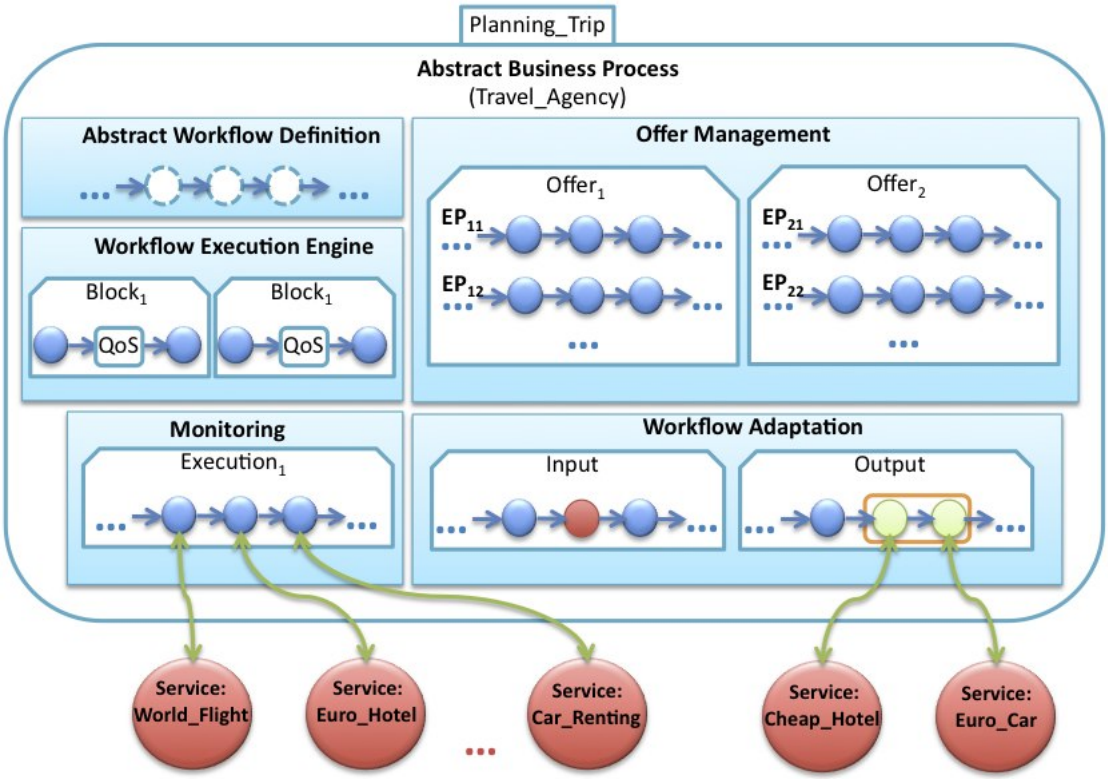

Fig. 4: The Structure of Abstract Business Process

Abstract Workflow Definition module defines the abstract workflow. This definition will be sent to the *Workflow Instantiation* component (introduced later on) for workflow instantiation process. As introduced, a service (or a functionality) can be delivered on dif-

ferent levels, so this abstract business process can publish multiple offers based on different global QoS constraints. For example, $Offer_1$ permits to deliver the "*Planning_Trip*" functionality within 10 seconds at the cost of 2 dollars; while $Offer_2$ promises to deliver the same functionality within 5 seconds at the cost of 5 dollars. For each offer, the abstract workflow should be instantiated at least once (in lucky case that there is no error) to get the feasible execution plan. The "*Offer Management*" module manages all the possible execution plans for all the published offers.

Once a feasible execution plan is found, the abstract business process can publish the relative offer in the Registry. In this way, this offer becomes discoverable by the other services during their instantiation process. Once its offer is selected, it has to wait to start the calculation until it receives the relative input. The Execution Engine is in charge of executing an execution plan. In our middleware system, the workflow execution is decentralized. The execution engine firstly cuts the execution plan to several blocks, and then sends these blocks to the relative partner services. On receiving these blocks, the partner services wait for their

Another important module is "*Monitoring*", which monitors and evaluates all the workflow executions. Once it detects the errors or deviations during the execution (for example, one of the partner service is crashed during the execution or the output format of a step does not conform with the input of its succeeding step), it will notify the Workflow Instantiation to adapt a part of execution plan. The adaptation is done in the run-time to avoid restarting the execution. From the client point of view, he can never perceive this action.

In order to start the adaptation process, the whole execution plan will be sent to the *Workflow Adaptation* module. This module takes charge of replacing the bindings of a part of workflow in order that the following execution of workflow will still meeting the global QoS constraints. Take the example from Figure **??**, during the $Execution_1$ (this is the identity of the workflow execution), the monitoring detects that the second partner service (Euro_Hotel) is crashed, so it informs the Workflow Adaptation module to carry our the adaptation process. The whole workflow should be delivered within 10 seconds at the cost of 2 dollars, where the Euro_Hotel requires 3 seconds and 0.5 dollars and Car_Renting costs 2 seconds and 0.5 dollars. Since the Euro_Hotel service is crashed (due to some unpredictable reason such as server crash or network congestion), Workflow Adaptation module will consult the Registry to see whether it is possible to adapt a part of workflow in order to avoid the QoS violation. Finally, the Registry finds the service Cheap_Hotel and Euro_Car which requires 3.5 seconds 0.3 dollars and 1 seconds and 0.7 dollars. In this way, the two newly selected partner services cost only 4.5 seconds and 1 dollar which is better than the previous solution which costs 5 seconds and 1 dollar. In this sense, the rest of the workflow does not need to be adapted.

2.3 Registry

The *Registry* acts as a directory maintaining the information of all currently available offers. An *offer* specifies both functional and non-functional characteristics of a service delivery. It can be seen as a contract between a service requester and provider[4]. Take an example in our daily life: if a client wants to sign a mobile phone package (forfait), how he makes his choice among multiple offers (different packages from different operators)?

First of all, he should think about which kind of package he needs. For example, if he is a business man with a lot of phone calls to keep in touch with his clients, he prefers to choose some kinds of package with unlimited phone call hours. On the other hand, if he has plenty of travels, he might choose a package with unlimited 3G access for better entertainment during his trip (surfing, chatting etc.).

Next, there should be a little differences between the offers with the same/similar functional properties but from different operators. So the client has to compare them in order to make his choice. For example, for the package of 1 hour of phone calls and unlimited internet access, the offer from `Orange` is a little more expensive (39 euro/month), but it provides a twice rapid internet access. Compare to this offer, the one from SFR is cheaper, 37 euros/month, but only a normal speed; moreover, you can profit the SFR wifi freely on more than 300000 sites in France. All these criterions consist of both functional and non-functional properties.

For Web services, we have the same case. An offer has both functional and non-functional descriptions of a service delivery by a certain provider. A service provider (i.e. an enterprise) can join this middleware by publishing offer(s) to the *Registry* component. By doing so, when other services (the composite service) want to select adequate partner services, this service provider can be found and known. On the other hand, the service provider can leave this middleware freely. Before leaving, he has to inform the *Registry* component to erase all the offers that he has published. In this way, this service provider will never be found during the following instantiation process. If a service provider encountered a technical problem such as the server crash, there is a consistency problem in the middleware system: the offer is still validate while the provider has already gone. In this case, the *Registry* component has implemented some mechanisms to ensure the offers alive. Once it detects the absence of a service provider, it has to notify all the services in order to keep there execution plan (concrete workflow) validate. More details can be found in Adaptation part. A more in-depth discussion about the workflow adaptation can be found in Section **??**.

[4] but it is not true, an offer only defines the responsibility of a service provider such as functional and non-functional promise, but does not define the punishment in case that the offer is failed to deliver

2.4 Workflow Instantiation

Each request to the execution of a workflow is associated with a global QoS constraint specified by the requester (such as how much he can offer to purchase a service delivery with the response time less than 10s). Before the execution, the abstract workflow is forwarded to the *Workflow Instantiation* (WI) component. This component is responsible for building a possible *Instantiated Workflow* (IWF) by assigning each workflow activity to a suitable abstract service. The selection of adequate abstract services is based on all currently available offers in the Registry and the global QoS constraints specified by the client. Our system aims at searching a feasible IWF that can satisfy the client's requirement rather than the optimal one.

2.5 Global Monitoring

3 Implementation

3.1 Abstract Service

3.1.1 Offer Creation

Service providers can create offers to classify the services that it can provide. For a service provider, to create an offer is to create a triple: functional property, non-functional property and provider identification. The functional property is fixed/pre-defined, as specified in the WSDL file. Furthermore, the provider can be identified by its name and IP address (how to find it). As a result, to defined an offer is to define the non-functional properties (such as price, response time etc.).

For the *atomic service*, as shown in Figure **??**, the non-functional properties can be derived from the contract between this service provider and the utility provider, known as *low level contract*. The utility provider works on the lower level and provides some hardware support (such as hosting services) for a Web service. The Web service provider can define non-functional properties based on this low level contract. For example, the response time of a Web service depends on the disk read/write speed as well as internet access speed, the price is calculated based on the price of hosting.

3.1.2 Invoker

An invoker can be seen as chemical-level reflection of a service implementation. It is implemented by a JAVA object which provides a group of functionalities through a set of interfaces. From the chemical level, this object can be regarded as a molecule which can participate in the chemical reactions. Chemical rules use its interfaces to operate on the invokers, such as activating or selecting an invoker. Here some of the principal interfaces are reported:

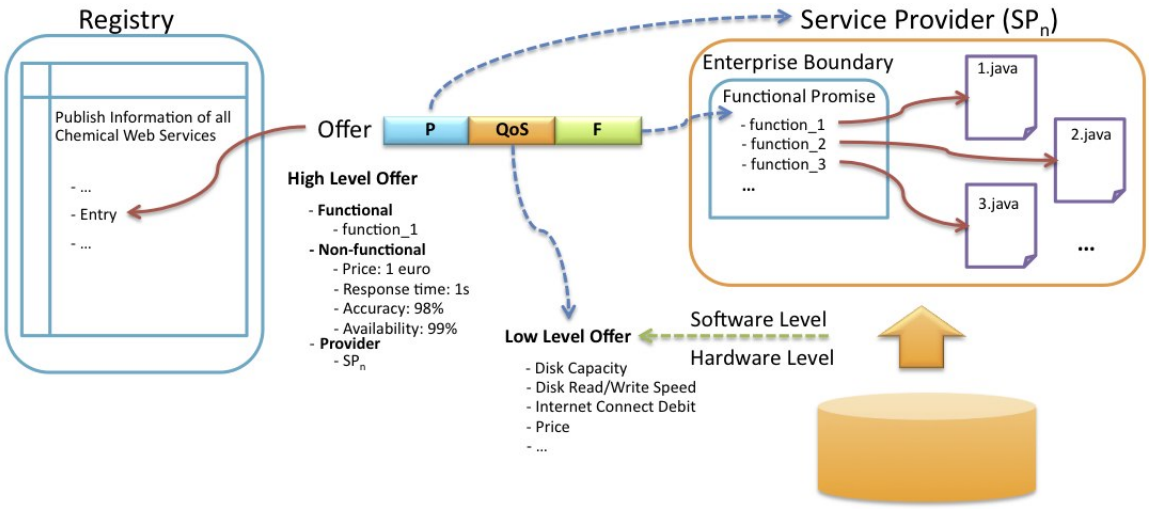

Fig. 5: Atomic Service: Define Non-functional Properties

- **isValid/setValid/setInvalid** To select an invoker, some interfaces are exposed to operate on its status. An invoker has two status: valid and invalid. *isValid* operation returns a boolean value indicating whether this invoker is active. *setValid* activates this invoker while *setInvalid* deactivates it.

- **getQoS/addQoS** An invocation can be associated with multiple QoS constraints. To simplify the description, it is assumed that only one generic QoS parameter (you can regard it as price or response time) is monitored by the monitoring module for each invocation. The monitored data is stored in the relative invoker. The historical information can be accessed by a pair of interfaces. *getQoS* returns the QoS value calculated based on the historical information. The calculation is based on a certain algorithm. *addQoS* records the monitored data from the *monitoring module* into its built-in memory. This information will be used for the next calculation of QoS value.

- **invoke** This operation forwards the invocation request to the corresponding service implementation. It requires two parameters: one is the operation name and the other is a set of parameters. These parameters are encapsulated into a SOAP message to invoke the corresponding operation.

3.2 Abstract Business Process

3.2.1 Offer Creation

For a composite service, the case is much more complex. A composite service defines an abstract workflow to coordinate partner services in order to carry out a functionality. As shown in Figure **??**, instead of implementing *function_1*, SP_n defines an *abstract workflow* with the functional requirements for each step. But the workflow is abstract. That is to say, service provider SP_n has only planed a workflow but he has no knowledge of the market. So it is hard for him to define the non-functional properties of an offer. Moreover, it is not assured that he can find all service providers for instantiating the workflow. As a result, it is more logical to instantiate a workflow before publishing an offer. (In [**?**], we propose to publish offers before the instantiation process)

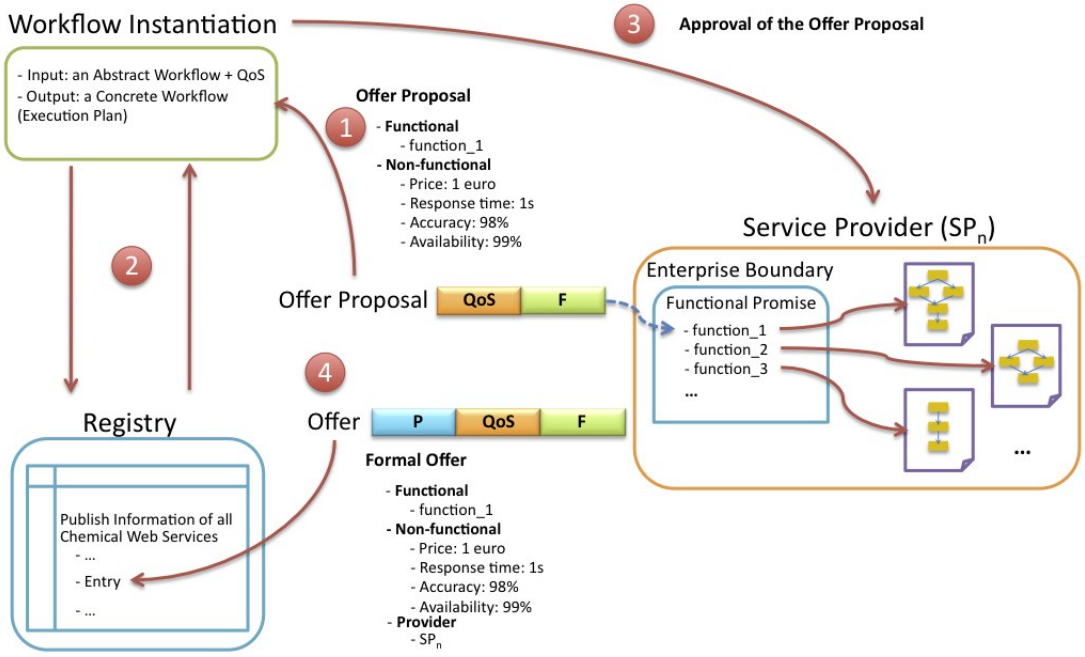

Fig. 6: Composite Service: Create Offers

So, to create an offer, at the first stage, the service provider creates a proposal offer and sends it to the *Workflow Instantiation* component. This proposal consists of the function requirements (abstract workflow description) as well as global non-functional requirement. The non-functional requirements are considered on the workflow level, because a client buys only the function at a certain price and under certain QoS constraints; but he does not care how the provider cut/divide them.

On receiving an offer proposal, the Workflow Instantiation component consults the Registry to get the live offer information that can provide the functional requirement. And then, it tries the different combinations to see whether a feasible solution exists. If a feasible solution is found, it sends back an approval message to the service provider. This message contains the concrete workflow information. Otherwise, it returns a rejecting message indicating that the proposed offer can not be completed based on the knowledge of all currently available offers. In this case, the service provider has to re-define the offer proposal and send it to the WI again until a feasible solution is found.

Because an offer stands for a promise, a service provider publishes an offer in the *Registry* means that he promises to be capable to provide the functional requirement F under the non-functional requirement QoS. That is why we propose to instantiate a workflow before publishing the offer.

3.2.2 Workflow Execution Engine

When an invoking message is received, the service provider has to start the execution of the workflow. Because the service provider has received an Execution Plan (concrete service, that associates each task with a certain service provider by endpoint reference), once an invocation arrives, he can execute this service plan. The service plan received during the instantiation process is set as the default execution plan. As proposed in [**?**], the execution of a workflow is distributed.

This process is shown in Figure **??**. The above is the abstract workflow while below is the concrete workflow (Execution Plan). Service Provider SP_n is a composite service. Once invoked, SP_n executes the plan calculated before, that is SP_n1, SP_n2 and SP_n3. Finally, the result will sends back to the next task SP_k. From this process, you can see that the execution of workflow defined by SP_n is decentralized, there is no centralized control, every partner service knows its preceding service and the succeeding service. The detail execution is described in [**?**].

This architecture avoids the Single Point Failure (SPF), but it has no central control, it is hard to monitor the workflow execution. When a composite service distributed the execution logic of a workflow and the initial input, the execution is launched automatically. But if a service provider is crashed during the execution or he leaves the middleware system before his turn of execution, no entity could know it and the execution of the whole workflow crashes. As a result, we need some the third party component to monitor the execution of a workflow.

The abstract composite service can be a good candidate as a “monitoring module”. Because in this architecture, after distribution of the workflow execution, the service provider SP_n becomes idle. So why not implement a monitoring module in each abstract business process component? In this way, the business process becomes a centralized entity, it can monitor the execution and even predict the necessity of adaptation. In this way, the execution is always decentralized while the control is centralized.

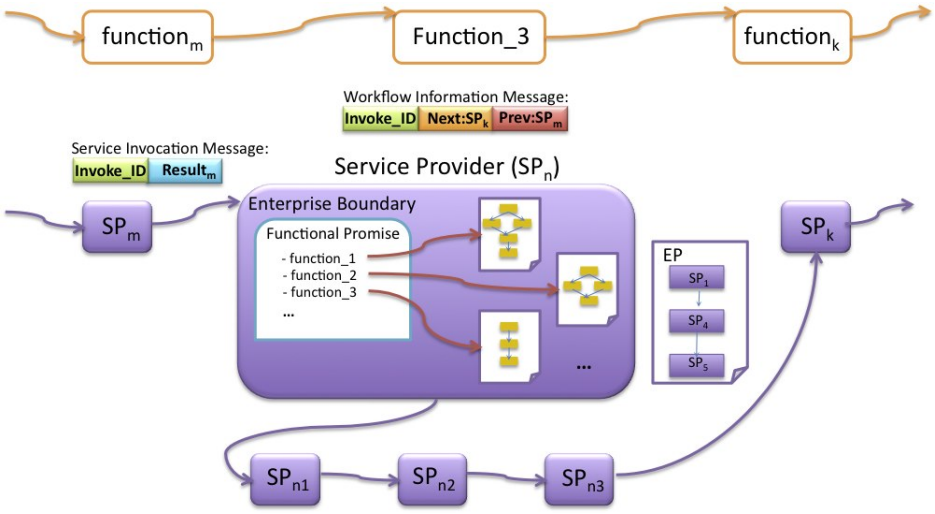

Fig. 7: Workflow Execution

3.2.3 Monitoring

Take the former scenario of "Planning_Trip", as shown in the Figure **??**, three services are bound to an execution of workflow: World_Flight, Euro_Hotel and Car_Renting. The blue ball represents the activity that is already finished, the green ball stands for the currently executing activity and the blue circle shows the activity to execute in the future. The Monitoring module has the definition of workflow, binding information as well as the offers of its partner services. In this way, it knows which service should be executed for which activity under which QoS constraints. For example, it knows for the activity A_2, the service Euro_Hotel will be executed at the price of 0.5 dollar and the expecting execution time is 3 seconds.

Suppose that at the moment t_1, service World_Flight finishes its calculate and sends the result to Euro_Hotel. Meanwhile, it has to send the execution-related information back to its client - Planning_Trip service. Planning_Trip service stores this information and set activity A_1 as "FINISHED", and then set A_2 as the currently executing activity. Next, it calculates the estimated time that activity A_2 should finish its calculation. In this example, the estimated time is $t_1 + 3$. Now, the Planning_Trip service has to wait Euro_Hotel service to calculate.

The system will evaluate the current executing activity periodically. Once it detects a QoS violation, for example, by comparing the estimated finish time with the current system time, it will mark the current activity as crashed (shown in red color) and then send the whole workflow execution (together with the execution state) to the Workflow Adaptation module in order to invoke the adaptation process. On the other side, if activity A_2 is

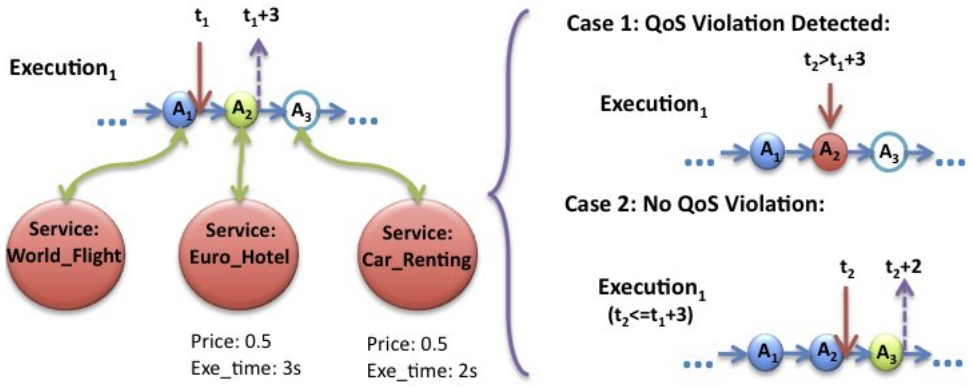

Fig. 8: Monitoring the Workflow Execution

accomplished in time, suppose that the accomplished time is t_2, which is earlier than $t_1 + 3$, the Monitoring component will mark the activity A_2 as "FINISHED" and A_3 as the currently executing activity. Meanwhile, like the last time, it will calculate the estimated finishing time of A_2 as $t_2 + 2$.

3.3 Registry

As introduced above, an offer is a contract between the requester and provider on both functional and non-functional aspects. As a result, an offer can be represented by a triple:

$$\text{Offer} = \{\text{F, QoS, P}\}$$

where F stands for the functional properties (analogy to 1 hour phone calls and unlimited internet), QoS specifies the non-functional properties such as price, the internet debit etc. and P identifies the service provider.

3.3.1 Multi-set Based Implementation

The Registry is a directory for stocking offers. The straightforward way to implement the Registry component is to create a multi-set containing multiple offer triples as molecules, as shown below.

```
"REGISTRY":<
  "OFFERS":<
    1:"f1":"SP1":<"PRICE":1,"Response_time":1.5>,
    2:"f1":"SP3":<"PRICE":1.2,"Response_time":1.2>,
    3:"f1":"SP12":<"PRICE":0.8,"Response_time":2>,
    4:"f2":"SP2":<"PRICE":2,"Response_time":2>,
    ...
  >,
  selectOffers
>
```

In this way, to select the qualified offers for workflow instantiation process, the chemical rule `selectOffers` can be applied:

```
let selectOffers =
replace f1::String, f2::String, f3::String,
        "QoS":<"TOTAL_TIME":time::double, "TOTAL_PRICE":price::double>
        "OFFERS":<
            id1::int:f1::String:sp1::String:<
              "PRICE":p1::double, "Response_time"rt1::double
            >,
            id2::int:f2::String:sp2::String:<
              "PRICE":p2::double, "Response_time"rt2::double
            >,
            id3::int:f3::String:sp3::String:<
              "PRICE":p3::double, "Response_time"rt3::double
            >
         >
by  "OFFERS":<
       id1:f1:sp1:<"PRICE":p1, "Response_time"rt1>,
       id2:f2:sp2:<"PRICE":p2, "Response_time"rt2>,
       id3:f3:sp3:<"PRICE":p3, "Response_time"rt3>,
     >,
     "RESULT":<
       id1:id2:id3
     >
if (p1+p2+p3)<10 && (rt1+rt2+rt3)<5
```

This rule will get three offers, one for each workflow activity (see the example in the Figure **??**), calculates their total non-functional performance (here, the total price and response time is required). If the total non-functional requirement can satisfy the user's requirement (less than 10 dollar and within 5 seconds), this feasible solution is encapsulated in the result tuple.

3.3.2 Database Based Implementation

In this solution, all the offers are stocked in the database, the multi-set only defines a convertor that translates the user's requirement to the corresponding SQL queries. As shown in Figure **??**, the convertor receives the abstract workflow (often a part of abstract workflow description) and the global QoS constraints on this workflow fragment as input. Then, it converts the user's requirement to the corresponding SQL queries and search in the database for the appropriate offers for each workflow activity. Once the whole workflow can be instantiated within the global QoS constraints, it sends back the feasible execution plan to its client.

The convertor is implemented by a set of chemical rules. These rules replace the abstract workflow description with the relative SQL requirements. The SQL requirements can activate other rules to operate on an HDBC Java object. The HDBC (HOCL Database Connectivity API), quite like JDBC, is a standard for database-independent connectivity between the HOCL programming language and a wide range of SQL databases. An HDBC

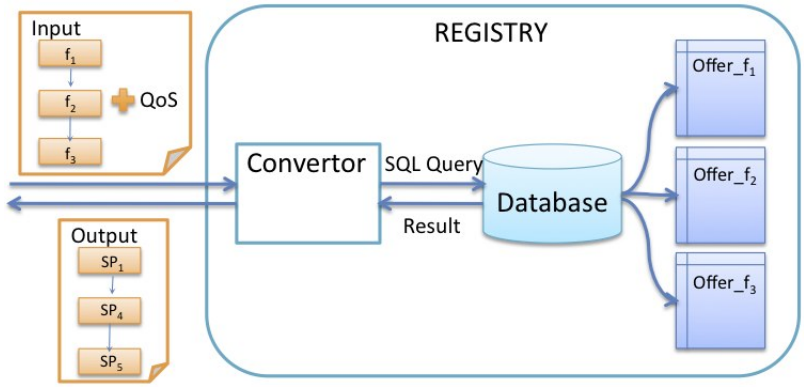

Fig. 9: Workflow Adaptation

object can be seen as a molecule involving in certain chemical reactions. Those reactions replace the SQL queries with the relative results by carrying out the selecting process in the database.

In the database, for each functionality, it maintains a table. Using the example shown in Figure **??**, the client wants to select offers for functionalities f_1, f_2 and f_3 with the following global QoS constraints: total price less that 10 dollars and the response time within 5 seconds. In this case, the convertor generates the following SQL query:

```
SELECT offer_id1, offer_id2, offer_id3
FROM Offer_f1, Offer_f2, Offer_f3
WHERE (Price1+Price2+Price3)<10 && (rt1+rt2+rt3)<5
```

And it will get the following result:

```
+------+------+------+
| ID1  | ID2  | ID3  |
+------+------+------+
|   21 |  106 |  208 |
|   67 |  153 |  284 |
|   67 |  199 |  208 |
|   21 |  199 |  284 |
|   67 |  199 |  284 |
+------+------+------+
5 rows in set (0.20 sec)
```

The Registry will send this information to its client. Figure **??** shows only one possible result, i.e.: offer 21 is provided by service provider SP_1, offer 106 is provided by SP_4 and offer 208 is provided by SP_5.

One of the big advantage to implement the Registry component based on the database system is that, the database stocks and organizes the data in a more efficient way. It can largely improve the system's performance. As you can see in the next section, the

run-time adaptation requires the rapid response. And in the reality, the number of offers in the system is really enormous. It requires the system to have the ability of fast data processing. Database is a good choice.

Furthermore, using multi-set based solution, there is a great possibility to miss a feasible solution. But database based solution solves this problem perfectly. As you can see in the last example, an offer can appear in multiple combinations.

4 Adaptation

4.1 Adaptation Overview

The execution environment changes from time to time. Resources can join and leave freely. When a service/resource is not available during the execution, the adaptation process has to be carried out to find an adequate substitute for the crashed service/resource. The adaptation is always carried out in run-time, without stopping the execution of a workflow.

Let's consider a general case, as shown in Figure **??**, suppose that the Service Provider for function$_m$ is crashed. As introduced, the workflow execution is carried out on chemical level, so each of the block represents an abstract service, in other words, a chemical program. This abstract service has three ways to carry out the real computation: 1) it invokes a service implementation; the service is implemented as a Web service. 2) it invokes a grid service; the grid service has the adaptation ability. 3) It defines another workflow and coordinates multiple services.

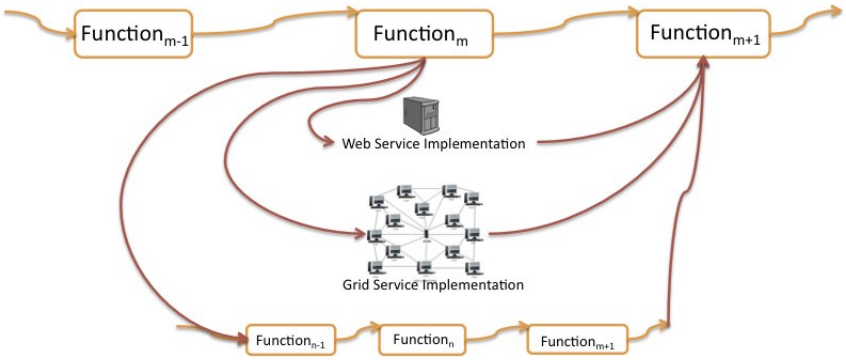

Fig. 10: Workflow Adaptation

From this example, we can see that the adaptation can be carried out on two levels: abstract service level and workflow level. The former is the case 2) and 3). Because it connects to the grid service or another workflow, both of them has adaptation ability. So, the adaptation can be taken place at the abstract service level, which we also called "local adaptation". This means we do not to modify the workflow or adapt the node

Function$_m$. While in the case 1), or the local adaptation is failed to find out the solution, the adaptation can be performed on the workflow level. In this case, the former service provider for Function$_m$ has to be replaced by selecting an adequate substitute. In contrast to the "local adaptation", it is also called "global adaptation". This case is much more complex since the modification of a partner service can affect the global non-functional constraints. As a result, the global adaptation is always together with the adaptation of a part of workflow.

4.2 Low Level Adaptation

Low level adaptation is carried out within the domain of an service provider. It will not affect the whole workflow. Take the example shown in Figure **??**, if Function$_m$ is crashed, it has some adaptation ability, so there is no need to modify the workflow WF$_1$. In this case, the adapting service can be seen as a black box, it adapts some of its modules, but finally, it can still meeting the functional and non-functional requirements as it promised (the offer that it has published).

This low level adaptation is based on the adaptation ability of the implementation of an abstract service. As introduced, this can happen in two cases: first, the implementation is based on grid service (cloud service). As you know, the grid/cloud has certain adaptation abilities. This part can be connected with the work of André. The second case is that it defines a workflow, this service provider can carry out the global adaptation of this workflow. For example, it can replace Function$_n$ with another service provider to enable the whole workflow can be delivered within the non-functional constraints that it promised. As for how to realize the global adaptation, we are going to introduce in the following section.

4.3 High Level Adaptation

Once the low level adaptation is not succeeded, for example, the global adaptation is failed to perform (no adequate candidate can be found) or the Function$_m$ connect an atomic Web service implementation, the high level adaptation has to be performed. As shown in Figure **??**, at the very beginning, the composite service receives an invocation under the following constraints: \$10 budget and 5 seconds execution time. At time 2, SP$_n$ is crashed. At that moment, the \$2 budget and 3 seconds are left. So it is expected to adapt the following workflow by select again a set of adequate partner services (SP$_l$, .., SP$_j$). Because the replace of SP$_n$ make leads the overall non-functional constraints violation. As a result, replace one service is always accompanied by the substitution of a set of element. The adaptation should consider also the cost of adaptation process itself (in this example, an instantiation process is promised to be achieved within 0.5 seconds at the cost of \$0.5).

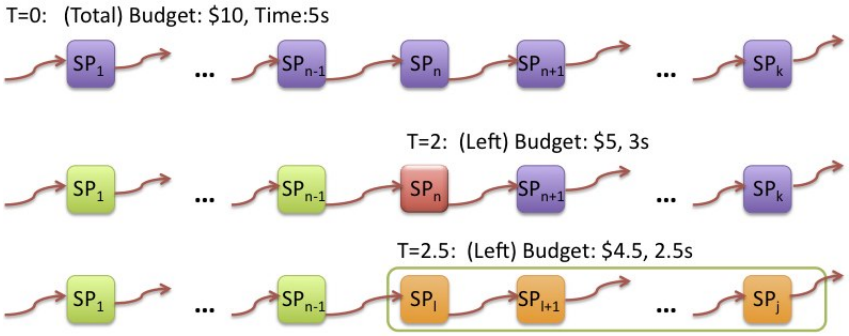

Fig. 11: Global Workflow Adaptation

4.3.1 Problem

The biggest problem we encountered are that chemical programming has the run-time adaptation ability but a very low efficiency. Using chemical programming, if a partner service is crashed due to some unknown reasons, we can use a chemical rule to replace it with another candidate partner service. Each component can be seen as a molecule that participates in certain reactions (reactions for adaptation). But the process to find an adequate substitute is costly. As you know, in the reality, for a certain functionality, you can find hundreds or even thousands providers. They provide the same function with different non-functional constraints, such as price, response time etc. As a result, it becomes a challenge that how to select an substitute service provider or a set of service providers to ensure that the execution of the overall workflow can still satisfy the requester's global requirement.

4.3.2 Proposed Solutions

- Regard the Workflow Instantiation As a Service

  The Workflow Instantiation Component is also a service that can publish its offers into the *Registry*. In this case, when a composite service needs to find a possible instantiated workflow, it can just buy the offer published by the Workflow Instantiation service.

- Pre-calculate Enough Execution Plans Before the Execution

  The alternative solution is to pre-calculate enough Execution Plans before the execution.

  This way is much efficient than the first solution once some unpredictable errors occurs during the execution; As every coin has two sides, if the crash or absence of a service provider happens rarely, it becomes costly.

4.4 Scientific Workflow

The scientific workflow

References

[1] Héctor Fernandez, Thierry Priol, and Cédric Tedeschi. Decentralized approach for execution of compositeweb services using the chemical paradigm. *IEEE International Conference on Web Services (ICWS)*, 2010.

[2] Chen Wang and Jean-Louis Pazat. Using chemical metaphor to express workflow and orchestration. *The 10th IEEE International Conference on Computer and Information Technology*, June 2010.

[3] Chen Wang, Jean-Louis Pazat, Maurizio GIORDANO, and Claudia DI NAPOLI. A chemical based middleware for workflow instantiation and execution. *ServiceWave*, 2010.

[4] Chen Wang and Thierry Priol. *HOCL Programming Guide*. INRIA Rennes, September 2009.