DEWS2004 7-B-4

WS-SAGAS のための THROWS アーキテクチャーの シミュレーション・システム

ベンラクハル・ネイラ[†]　　小林　隆志[††]　　横田　治夫[††,†]

[†] 東京工業大学 大学院 情報理工学研究科計算工学専攻　〒 152–8552 東京都目黒区大岡山 2–12–1
[††] 東京工業大学 学術国際情報センター　〒 152–8552 東京都目黒区大岡山 2–12–1
E-mail: †neila@de.cs.titech.ac.jp, ††{tkobaya,yokota}@cs.titech.ac.jp

あらまし　ウェブサービス統合の出現で、現在、分散システムでの有効性および信頼度を高めるための研究が注目を集めている。我々は、これまでに WS-SAGAS トランザクションモデルを提案してきた。さらに、統合ウェブサービスの高信頼な実行を実現ために THROWS アーキテクチャーも提案している。この論文では我々の提案する手法の適用性を調べるために開発したシミュレーション・システムについて報告する。

キーワード　ウェブサービス統合、分散アーキテクチャ, ワークフロー, 信頼性, トランザクション モデル.

A Simulation System of THROWS Architecture for WS-SAGAS

Neila BEN LAKHAL[†], Takashi KOBAYASHI[††], and Haruo YOKOTA[††,†]

[†] Department of Computer Science, Tokyo Institute of Technology
Oookayama 2–12–1, Meguro–ku, Tokyo, 152–8552 Japan
[††] Global Scientific Information & Computing Center, Tokyo Institute of Technology
Oookayama 2–12–1, Meguro–ku, Tokyo, 152–8552 Japan
E-mail: †neila@de.cs.titech.ac.jp, ††{tkobaya,yokota}@cs.titech.ac.jp

Abstract　With web services compositions emergence, the current research efforts are toward availability and reliability enhancement in distributed and loosely coupled systems. We proposed in a previous work WS-SAGAS transaction model and THROWS architecture for web services compositions reliable specification and execution. The purpose of this paper is to report on a simulation system that we implemented to check out our proposal applicability and confidence degree.

Key words　Web Services Compositions, Distributed Architecture, Workflow, Reliability, Transaction Model.

1. Introduction

Previous attempts in integrating distributed computing systems have yielded a large number of solutions(e.g., CORBA, COM/DCOM). Unfortunately, the majority of these solutions are exceedingly complex, vendors' dependant and requiring huge infrastructure investments. In addition, in such kind of system, ensuring reliable interoperation necessitates the use of complex and proprietary communication protocols and too many agreements on context sharing. As consequence, potential failures are increased and the recovery becomes tedious and time consuming.

To overcome these shortcomings, there is a present trend from tightly coupled systems toward loosely coupled systems.

Later on, the web services paradigm emerged and gained a rapid uptake because it meets exactly these requirements of interoperability. In fact, the web services are based on ever-present protocols and on a set of recognized standards (e.g., HTTP, XML, SOAP, UDDI, WSDL). These characteristics allowed the web services to reach a high level of acceptance and to trigger massive research efforts towards deploying business processes as an orchestration of web services compositions (e.g., the Open Grid Services Architecture (OGSA) [1]). However the resulting compositions interoperability is considerably enhanced, the reliability and availability issues are not yet well-addressed.

Specifically, considering that web services are in essence hosted by different providers, they might have not compliant characteristics(e.g., transactional supports, management mode). As a result, any update of a service might affect critically the overall composition consistency, reliability and availability. Moreover, regarding compositions execution model, up to now, the majority of the proposed work are executed with a centralized control, thereby such central controller will constitute a single point of failure.

Motivated by these concerns, we have proposed in a previous work *WS-SAGAS* a new transaction model [4] [5] for web services compositions specification and *THROWS (Transaction Hierarchy for Route Organization of Web Services)*, a *distributed* architecture for web services compositions reliable execution [6].

To check out the applicability of our proposals, we implemented a simulation system heavily based on java programming language and on a set of web services enabling technologies. In the reminder of this paper, we will give an overview of WS-SAGAS and THROWS, after that, we will report on this simulation system and describes its implementation. We will also sketch a case study and report on its execution. Finally, we will provide some concluding remarks.

2. WS-SAGAS and THROWS Overview

In a nutshell, the most prominent features of WS-SAGAS and THROWS are:

- WS-SAGAS specifies the web services compositions as a hierarchy of *arbitrary-nested* transactions. These transactions potential execution is provided with *retrial and compensation* mechanisms. The web services compositions depicted as WS-SAGAS can execute within THROWS architecture. In this case, the execution control is hierarchically delegated among *dynamically discovered engines* ranked in *CELs(Candidate Engines List)*. In addition, the progress of the compositions execution is continuously captured using the *CEP(Current Execution Progress)*.
- WS-SAGAS inherits the arbitrary nesting level, the relaxed ACID properties, the compensation and the vitality degree features proposed in several advanced transaction models [2] [7]. Since web services are originally stateless, when they are executed as components of a same composition, without the state concept, it would be difficult to know the whole composition execution progress. For that purpose, we proposed to enrich WS-SAGAS with *state capturing*.
- WS-SAGAS defines a *composition* as an orchestration of *elements* we noted E_i^v. An element has a *state* we noted S_i and a *vitality degree* we noted v for $vital$ and $\bar{v}$ for $notvital$. According to the nesting level, an element could be assimilated to an atomic element or to a composition specification.
- THROWS applies a peer-to-peer execution model where the composition execution control is distributed among dynamically discovered entities we called *engines*. An engine is attached to an involved web service and is allocated to an element. Once allocated to an element, the engine becomes in charge of the service invocation, the execution context communication, the execution context update, the execution forward and eventually the execution control delegation or completion. On each engine, the *Current Execution Progress (CEP)* and the *Candidate Engines List (CEL)* are stored.

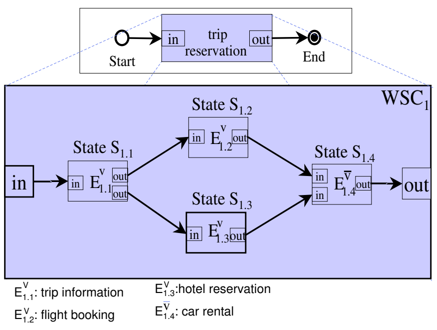


Figure 1 Compositions Specification with WS-SAGAS

(i) *CEP* keeps track of the web services compositions execution progress. When an element from a composition is executed by an engine, every change in that element's state is reflected on CEP.

(ii) *CEL* contains all the candidate engines potentially enabled to execute an element. Every engine, after finishing successfully the execution of the element it was allocated to, it is responsible of generating the CEL of its direct successor(s).

3. Simulation System Description

Implementing the simulation system aims at showing to what extent the ideas behind WS-SAGAS and THROWS are consistent with each other. Furthermore, it aims at checking their feasability using existing technologies namely WSDL, UDDI, SOAP and so forth.

3.1 Scenario

As a case study, for homogeneity's sake we kept using the same travel itinerary reservation scenario that we have described in our former work in [5]. The scenario that we have chosen involves a process with several activities including a trip information registration($E_{1.1}^v$), a flight booking($E_{1.2}^v$), a hotel reservation($E_{1.3}^v$) and finally a car rental($E_{1.4}^{\bar{v}}$). This scenario specification as WS-SAGAS is depicted in Figure 1. For easiness to implement, few adjustments to CEL and CEP contents were needed. These adjustments were necessary to add: *(i)* information needed for web services discovery (e.g. element description) *(ii)* information about the discovered web services(e.g. web service URI, parameters). The notation of CEP that we adopted is as follows (bold face components are newly added):

$$CEP(WSC_c)_{h-i} = \{(E_0, e_0, \mathbf{description}, S_0), ...(E_h, e_h, \mathbf{description}, S_h), (E_i, e_i, \mathbf{description}, S_i), [(E_j, e_j, \mathbf{description}, S_j); ...; (E_k, e_k, \mathbf{description}, S_k)], ...(E_n, e_n, \mathbf{description}, S_n)\}$$

- S_i (state) is exclusively one of the followings: $\{$*Waiting,Executing,Failed,Aborted,Committed,Compensated*$\}$

- The described $CEP(WSC_c)$ content is sent from e_h to e_i; e_h is responsible for E_h and e_i is responsible for E_i. E_0 is the first element and E_n is the last element of WSC_c.

- The notation $E_0 < [E_j; ...; E_k] < E_n$ is applied to specify the execution order of the elements of WSC_c. This means that E_0

is executed and committed first, then all the elements specified between "[]" and components of the same composition WSC_C, (i.e., E_j and E_k are included) are executed in parallel. Once they are committed, E_n is executed.

Similarly, the CEL for an element E_i is noted as follows:

$CEL(E_i) = \{(E_i, e_{i1}, \mathbf{wsdesc}, \mathbf{wsuri}, \mathbf{wsoperation}, \mathbf{wsport}, [\mathbf{wsparameter_1}, ...\mathbf{wsparameter_k}]), (E_i, e_{ip}, \mathbf{wsdesc}, \mathbf{wsuri}, \mathbf{wsoperation}, \mathbf{wsport}, [\mathbf{wsparameter_1}, ...\mathbf{wsparameter_k}])\}$

- $wsdesc$ the web service description;
- $wsoperation$ the web service operation name;
- $wsuri$ the web service Uniform Resource Locator;
- $wsport$ the web service binding port and
- $[wsparameter_1, ...wsparameter_k]$ parameters.

3. 2 Implementation Environment

3. 2. 1 Choices and Motivations

The simulation system implements a logically distributed version of THROWS architecture for executing web services compositions specified as WS-SAGAS. For the concurrent execution of engines, it makes extensive use of java threads and synchronization mechanisms. Our choice was mainly motivated by what follows:

(1) We are mainly interested in detection and recovery from semantic failures (i.e. web services and engines failures). A physically distributed system might suffer from the possible failures of the underlying infrastructure (e.g. message lost due to network failure, timeout). Our main focus is to show up how web services compositions specified as WS-SAGAS and executed in THROWS can overcome failures. We consider that recovery from other system-related failures are beyond the scope of the simulation system aims since, they have been widely addressed in other work. We also proposed tentative solutions in [6] that can apply as well.

(2) Regarding the web services, two choices are possible. The first choice is to set up a private UDDI registry and to publish the web service in it. The other choice is to publish in a public UDDI registry(e.g. Microsoft UDDI registry, IBM UDDI registry). We chose the first possibility because it is the more widely used and allows a better control [8].

3. 2. 2 Implementation Tools Description

The simulation system uses mainly java and a set of web services enabling technologies to implement an application with an entry point that can be eventually web-accessible.

We make also extensive use of the different APIs provided in Java Web Services Developer Pack (Java WSDP 1.2) [9]. Among the technologies that JWSDP comprises, we are using mainly Java API for XML Registries (JAXR) with Java WSDP Registry Server for the *Web Services Manager* and its various modules implementation. All the web services that we need for our simulation system are built and deployed in an XML registry that follows the UDDI specification. We are using JAXR to access this XML Registry. For building these web services, we are using Java API for XML-based RPC (JAX-RPC). The service invocation and context communication is done implicitly using SOAP messages over HTTP. Moreover, all the communication between the different modules uses the Soap with Attachments API for Java (SAAJ). Depending on the web services compositions execution stage, the exchanged SOAP messages encapsulate different kind of XML documents. These XML documents are parsed using JAXP and manipuled with JDOM and DOM.

3. 2. 3 Simulation System Components Description

The simulation system features four principles modules, the *Web Services Manager*, the *Engines*, the *User Conversation*, and the *WS-SAGAS Specification Generator*.

Each *Engine* dispatches its responsibilities to three categories of sub-modules, the *Manager*, the *Updater* and the *Communicator*. The *Manager* encapsulates and supervises both of the *Updater* and the *Communicator*. The *Updater* is responsible for any operation related to update, modification and information retrieval. In the other hand, the *Communicator* is responsible for any message exchange (sending and receiving). As we will detail it later on, we define three *Manager*s, the *CEP Manager*, the *Context Manager* and the *CEL Manager*. Moreover, there is another *Manager*, the *Web Services Manager*, but the latter is different from the *Manager*s encapsulated in the *Engine*, since it provides different functionalities we will detail in what comes hereafter.

(1) **Web Services Manager**: Consists of three modules, the *Services Builder* uses JAX-RPC and several other tools(e.g. ANT tool, wscompile, wsdeploy) to generate the web services endpoints, their clients and their WSDL documents. The *Services Deployer* deploys the built web services on a web container(we are using TOMCAT). The *Services Register* is responsible for service registration in the private UDDI registry, discovery and invocation.

(2) **User Conversation**: Is used as an entry point to input the potential user choices that will be considered as the web services compositions execution context. In the trip reservation, the inputs we are considering are destination, departure and return dates, and user name. These inputs are saved in the context, which is an XML document. This context is communicated in a SOAP message to the proper engine to be handled. In the implementation of this module can be we are using JSP(Java Server Pages).

(3) **Engines**: The number of instantiated engines depends from the route taken to execute the composition. Each *Engine* contains three modules, the *Context Manager*, the *CEL Manager* and the *CEP Manager*.

- The *Context Manager* is responsible for updating the context using the module *Context Updater*. The update is performed on the base of the inputs received from the customer request(inputs) or on the execution results of the web service. Besides, a *Context Communicator* module propagates the context to the web service.
- The *CEL Manager* is in charge of generating the CELs of the direct successor(s) of the current element. Based on the ele-

```
<?xml version="1.0" encoding="UTF-8" ?>
  <!DOCTYPE CEL (View Source for full doctype...)>
- <CEL>
  <candidateEngine elementid="1.3" engineid="1.31" wsdesc="JapanHotelService"
   wsuri="http://localhost:8080/japanhotel-jaxrpc/japanhotel"
   wsoperation="JapanReserveHotel" wsport="JapanHotelIF"
   wsparameter="String_1" />
  <candidateEngine elementid="1.3" engineid="1.32" wsdesc="HotelService"
   wsuri="http://localhost:8080/hotel-jaxrpc/hotel"
   wsoperation="ReserveHotel" wsport="HotelIF" wsparameter="String_1" />
  </CEL>
```

Figure 2 CEL Candidate Engines List (XML document)

ment description, the *CEL Communicator* is responsible for issuing a query to the *Web Services Manager* for web services with the desired functionalities. For each obtained web service, an $engineid$ is allocated and a candidate engine is selected from the CEL. The CEL is represented as an XML document. Its generation is under the responsibility of the *CEL Generator*. For instance, in the example of Figure 1, the CEL generated for element $E^v_{1.3}$ by the engine responsible of executing element $E^v_{1.1}$ is listed in Figure 2.

- The *CEP Manager* takes care of monitoring the CEP and depending on the web service execution state progress, the module *CEP Updater* is responsible for updating the state of the current element. The update from a state to another one is done on the reception of a particular SOAP messages. This message is received by the *CEP Communicator* and informs about the execution success or failure of the web service. Some other messages are also received from other engines and require as well the CEP update (e.g., requests for compensation, abortion). The CEP update and the element's state transitions are more detailed in [5] [6]. Figure 3 is the listing of the content of a synchronous SOAP message sent from engine $e_{1.11}$ to the newly allocated engine to execute the successor element. In this case the message is sent to two engines since element $E^v_{1.1}$ has as direct successors $E^v_{1.2}$ and $E^v_{1.3}$. Since element $E^v_{1.1}$ is successfully finished, its state in CEP is set to $Committed$ while all the other elements states are set to $Waiting$.

(4) **WS-SAGAS Specification Generator**: This module takes as an input the diagram representing the WS-SAGAS and gives as an output the context and the CEP described as XML documents. So far, we are still implementing this module. The simulation system uses CEP and context descriptions documents generated using directly DTD and XML documents.

4. Detailed Description of a Scenario

4.1 Outline

We will describe in what follows the execution of the trip reservation scenario within our simulation system. Before going in its details, we sketch out the scenario in a few brief sentences:

1. First, a customer request is accepted and is handed over to the server side. The request is processed and the results are sent back to the client to be checked by the customer;

2. The server-side application will go through several steps repeated up to the end of the composition execution completion. We divided these steps considering the module they will be handed over as follows(i.e. which module takes charge of which step):

3. The *Communicators* encapsulate received data within SOAP messages to send them or extract data from received SOAP messages. The steps that require to be handled like this are:

 a. Context and CEP propagation;

 b. Services discovery and discovery results communication;

 c. Services invocation and results communication.

4. The *Updaters* take care of any information retrieval, selection, or update. The steps handled are:

 a. Element, candidate engine and context selection;

 b. CEP and context update.

After this outline, in the following section we give a detailed description of the scenario execution with the different alternatives and the different treatments they entail.

```
REQUEST:
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<directory>C:/eclipse-SDK-2.1.1-
   win32/eclipse/workspace/THROWS4/E1.1/e1.11/receivedCEP.XML</directory>
<CEP><wsc nestinglevel="1" wscvitality="vital">
<element elementid="1.1" description="trip" engineid="NULL" state="committed"
   elementvitality="vital"><parallelwith elementnb="0" elementid= "NULL" /></element>
<element elementid="1.2" description="ticket" engineid="1.21" state="waiting"
   elementvitality="vital"><parallelwith elementnb="1" elementid="1.3" /> </element>
<element elementid="1.3" description="hotel" engineid="1.31" state="waiting"
   elementvitality="vital"><parallelwith elementnb="1" elementid="1.2" /> </element>
<element elementid="1.4" description="car" engineid="NULL" state="waiting"
   elementvitality="notvital"><parallelwith elementnb="0" elementid="NULL"
   /></element></wsc></CEP></SOAP-ENV:Body></SOAP-ENV:Envelope>
RESPONSE
<?xml version="1.0" encoding="UTF-8"?><soap-env:Envelope xmlns:soap-
   env="http://schemas.xmlsoap.org/soap/envelope/"><soap-env:Header/>
<soap-env:Body><Response>OK CEP is received correctly</Response>
</soap-env:Body></soap-env:Envelope>
```

Figure 3 Synchronous SOAP message for CEP Propagation

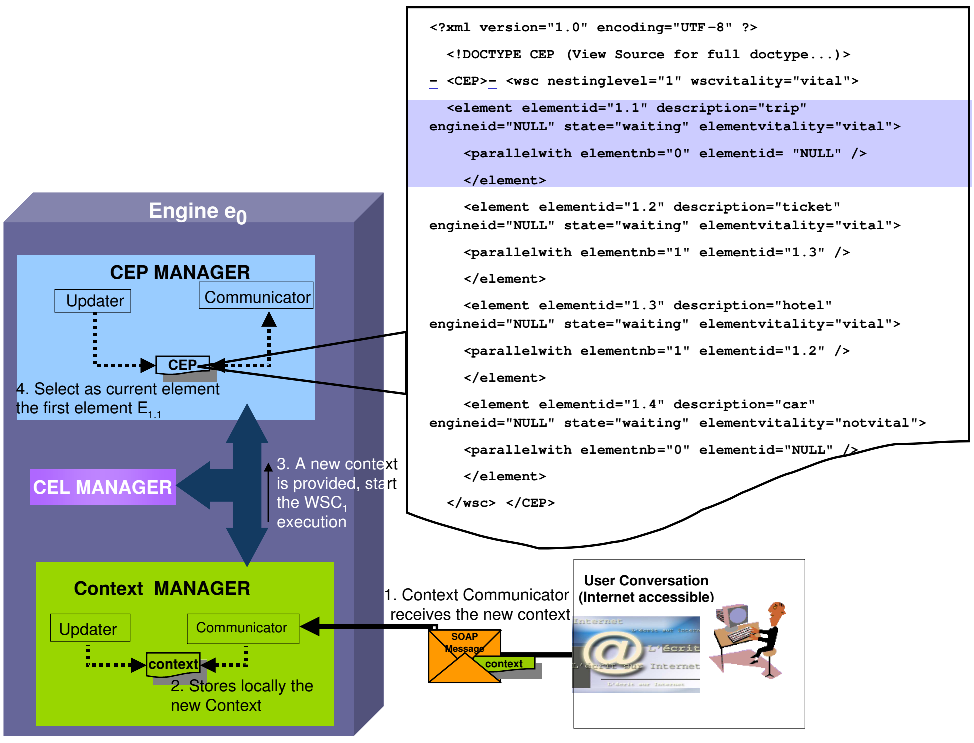
4.2 Customer Request Submission

A customer accesses the *User Conversation* interface and inputs his request*(destination, departure date, return date and his name)*. Submitting the request entails saving the inputted values in the *XML Context Document*. This *Context Document* will be updated and handled by the different *Engines* throughout the overall composition execution. By the end of its execution, the *Context Document* will contain information about the execution success(e.g. flight booking done, hotel ticket reserved, car reserved) or failure (e.g. no available flight). This information will be extracted using JDOM and JAXP from the context and communicated to the customer as a result for his request.

4.3 Customer Request Handling

A servlet running on the server-side will discover that a new *Context Document* was provided and will start handling it as follows:

1. *Element Selection:*


Engine e0
CEP MANAGER
Updater
Communicator
CEP
4. Select as current element the first element E1.1
CEL MANAGER
3. A new context is provided, start the WSC1 execution
Context MANAGER
Updater
Communicator
context
2. Stores locally the new Context
1. Context Communicator receives the new context
SOAP Message
context
User Conversation (Internet accessible)

```
<?xml version="1.0" encoding="UTF-8" ?>
  <!DOCTYPE CEP (View Source for full doctype...)>
- <CEP>- <wsc nestinglevel="1" wscvitality="vital">
  <element elementid="1.1" description="trip"
engineid="NULL" state="waiting" elementvitality="vital">
    <parallelwith elementnb="0" elementid= "NULL" />
    </element>
    <element elementid="1.2" description="ticket"
engineid="NULL" state="waiting" elementvitality="vital">
    <parallelwith elementnb="1" elementid="1.3" />
    </element>
    <element elementid="1.3" description="hotel"
engineid="NULL" state="waiting" elementvitality="vital">
    <parallelwith elementnb="1" elementid="1.2" />
    </element>
    <element elementid="1.4" description="car"
engineid="NULL" state="waiting" elementvitality="notvital">
    <parallelwith elementnb="0" elementid="NULL" />
    </element>
  </wsc> </CEP>
```

Figure 4 Context Reception and CEP Update

As depicted in Figure 4, a current element is chosen (the first element from the WS-SAGAS specification, here $E^v_{1.1}$) from the *CEP Document*. As we already specified, we are using JDOM for the document manipulation. The *CEP Document*, once being parsed, the first element(s) not yet executed is/are selected (i.e., both sequential and parallel elements execution are enabled). The function of going through this *CEP Document* for selecting elements is attached to the *CEP Updater* module. After an element, $E^v_{1.1}$, has been selected, a *CEL Document* needs to be generated. This is the responsibility of the *CEL Generator*.

2. *CEL Generation:*

The *CEL Generator* receives, as inputs from the *CEP Updater*, a *description* of an element (here $E^v_{1.1}$ *description* is provided). The *description* is actually used to generate/create a query that will be submitted to the *Web Services Manager*. The *Web Services Manager* will run the *Register* module, which will query the UDDI Registry for web services potentially enabled to fulfill the element($E^v_{1.1}$) *description*. For instance, considering the element $E^v_{1.1}$, its *description* is "trip information".

3. *Web Services Discovery:*

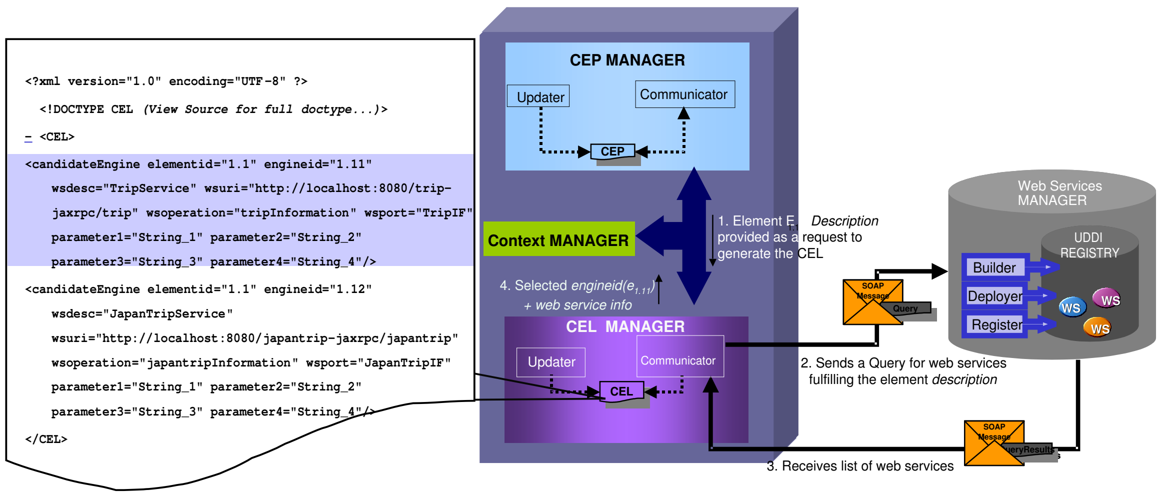
To ensure interoperable communication between the *Engine*(here considered the JAXR client) and the UDDI registry implementation, the SOAP messages that contain the query(and its corresponding results) are handled completely behind the scenes using SAAJ. The result of searching the UDDI registry for web services is all the organization(s) that contain(s) web services we are interested in (i.e. they have functionalities desired by the current element here $E^v_{1.1}$).

We remind here that web services are published in XML registries within organizations. An organization has a unique identification key, provided to it when it is published to the registry. To go back to our scenario, as depicted in Figure 5, when we query the UDDI Registry, the result is all the organizations with the *name* that contains the string "trip". For the web service, the element *Description* as we already mentioned is used. The retrieved information, as a result to the query, is parsed for details about the Organization(s) and the services it/they provide(s) and used to generate the *CEL Document*. To each web service, an engine is allocated, that is a new engine identification *engineid* is dynamically created and stored in the *CEL Document* coupled with the web service information.

In previous work [6], we proposed that the CEL would be ranked considering the QoS of the web services and the engines. In the actual situation, the engines reliability metrics are of minor status since we are in a logically distributed environment, so metrics such as web services invocation time and response time are insignificant. Furthermore, considering the web services, QoS measurement require to be more thoroughly addressed, for the simulation system we ranked them considering their order of appearance in the query results. Figure 5 contains the *CEL Document* generated.

4. *Engine Selection:*

The Engine selection is launched by the *CEL Generator* after terminating the *CEL Document* generation. This module will choose a candidate engine and inform the *CEP Updater* so that the *CEP Document* is updated with the new value of the *engineid*. The Selected engine is labeled as already allocated, here the chosen engine



```
<?xml version="1.0" encoding="UTF-8" ?>
  <!DOCTYPE CEL (View Source for full doctype...)>
- <CEL>
<candidateEngine elementid="1.1" engineid="1.11"
    wsdesc="TripService" wsuri="http://localhost:8080/trip-
    jaxrpc/trip" wsoperation="tripInformation" wsport="TripIF"
    parameter1="String_1" parameter2="String_2"
    parameter3="String_3" parameter4="String_4"/>
<candidateEngine elementid="1.1" engineid="1.12"
    wsdesc="JapanTripService"
    wsuri="http://localhost:8080/japantrip-jaxrpc/japantrip"
    wsoperation="japantripInformation" wsport="JapanTripIF"
    parameter1="String_1" parameter2="String_2"
    parameter3="String_3" parameter4="String_4"/>
</CEL>
```

CEP MANAGER
Updater
Communicator
CEP
Context MANAGER
1. Element $E_{1.1}$ *Description* provided as a request to generate the CEL
4. Selected *engineid($e_{1.11}$)* + *web service info*
CEL MANAGER
Updater
Communicator
CEL
SOAP Message
Query
2. Sends a Query for web services fulfilling the element *description*
Web Services MANAGER
UDDI REGISTRY
Builder
Deployer
Register
WS
WS
WS
SOAP Message
QueryResults
3. Receives list of web services

Figure 5 CEL Generation and Engine Selection

is $e_{1.11}$. The next time a candidate engine is to be chosen from the same *CEL Document*, the next ranked engine to the last-labeled one will be chosen (e.g. the next to $e_{1.11}$).

5. *CEP Update:*

The *CEP Updater* updates the *CEP Document*, the *engineid* of the element $E^v_{1.1}$ is changed from *null* to $e_{1.11}$.

6. *Control Delegation Preparation:*

The following step that comes after is preparing the necessary data for allocating effectively the engine so that it starts running. As a consequence, the *Context Document*, the *CEP Document* and a start signal are sent using SOAP messages. Meanwhile a new thread Engine $e_{1.11}$ is created, the received *CEP* and *Context Documents* are locally stored and a response is sent back to notify that the documents were received and that the element execution is launched.

7. *Control Delegation Done:*

As a first step after receiving the execution control, the module *CEP Updater* attached to $e_{1.11}$, will update in the *CEP Document* the state of the element $E^v_{1.1}$ from *Waiting* to *Executing*.

8. *Context Selection and Update:*

After that, the *Context Updater* will extract from the *Context Document* the context necessary for the web service invocation (i.e. the context document contains the context of all the WS-SAGAS's elements). The context of $E^v_{1.1}$ is selected. It contains actually the variables, their values and their type (input/inout/output).In the case of $E^v_{1.1}$, it contains four input (*destination, departure date, return date, name*) with their values(received in the customer request).

9. *Web Service Invocation:*

The *Context updater* provides this context to the *CEP Communicator*. The latter will invoke the web service client using this *Context*.

10. *Web Service Execution:*

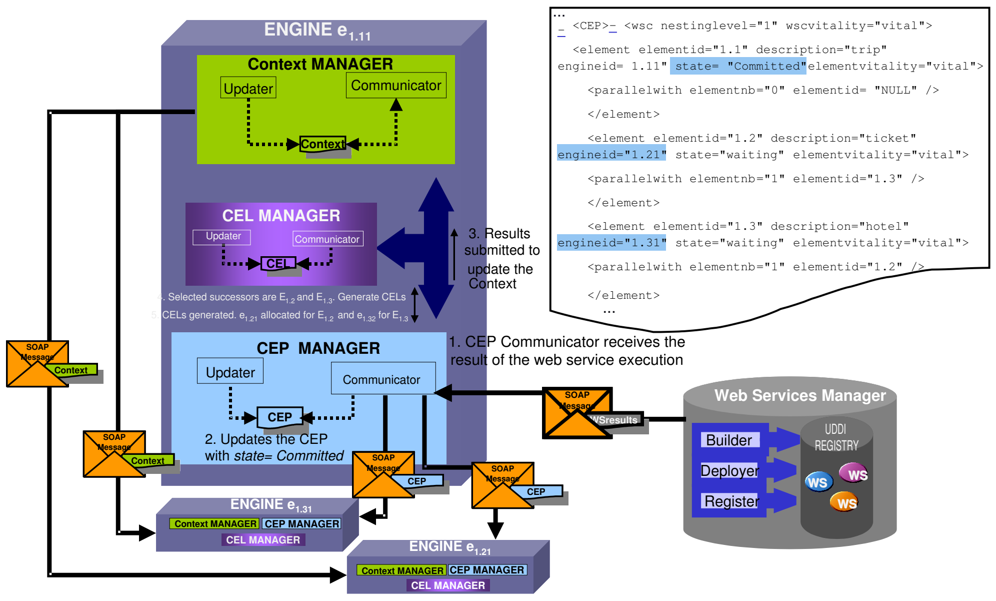
The JAX-RPC runtime is responsible for receiving this *Context* within the client call and for passing it to the web service endpoint. And when the web service finishes executing, it passes the results to the JAX-RPC runtime. Likewise, the latter will take care of handing over these results to the *CEP Communicator*. At this point, depending on the web service execution success or failure, two scenarios are most likely to take place:

a. **The web service execution fails**

If after a fixed time is elapsed, the *CEP Communicator* doesn't receive any response, or it receives a message that informs about the web service execution failure. As a result, the current thread *Engine* $e_{1.11}$ needs to deduce its own failure and to delegate the execution control to the previous engine thread (Figure 7). For that matter, *CEP Communicator* informs the *CEP Updater* about the failure. Afterwards, the latter will update the current element $E^v_{1.1}$'s state to $Failed$ and the former will communicate the *CEP Document* to the previous thread. Subsequently, the thread *Caller* is awakened and will confirm reception of the *CEP Document* with directly sending a response to thread *Engine* $e_{1.11}$ and suspending it. What comes after is that the locally stored *CEP Document* is updated (element $E^v_{1.1}$'s state set to $Failed$), the *engineid* set again to *null* and the *CEL Updater* is asked for selecting another candidate engine from the *CEL Document*. At this level, two cases are conceivable depending on the *CEL Document* content:

(i) If the *CEL Document* is empty: in this case, the *CEL Updater* will check if there is any other executed elements that need to be compensated. It will found that there is not. As a result, a message is sent back to the *User conversation* notifying the customer of the failure to satisfy his request.

(ii) If the *CEL Document* is not empty: the *CEP Updater* while going through it, it founds out another candidate engine available ranked next to the last labeled one. This engine is *Engine* $e_{1.12}$. Thus it comes that the *CEP Updater* is informed about this engine and will update the *CEP Document*: $E_{1.1}$'s state will be modified from $Failed$ to $Waiting$ and the *engineid* will be set to $e_{1.12}$. Afterwards, the newly updated CEP Document is sent again by the *CEP Communicator* to all the *Engines* that are already executing(in the present example, no other element is executing beside the cur-



```
...
_ <CEP>_ <wsc nestinglevel="1" wscvitality="vital">
  <element elementid="1.1" description="trip"
engineid= 1.11" state= "Committed"elementvitality="vital">
    <parallelwith elementnb="0" elementid= "NULL" />
    </element>
    <element elementid="1.2" description="ticket"
engineid="1.21" state="waiting" elementvitality="vital">
    <parallelwith elementnb="1" elementid="1.3" />
    </element>
    <element elementid="1.3" description="hotel"
engineid="1.31" state="waiting" elementvitality="vital">
    <parallelwith elementnb="1" elementid="1.2" />
    </element>
      ...
```

Figure 6   Control Delegation to other Engines

rent one $E^v_{1.1}$). The execution will be resumed as previously done with thread Engine $e_{1.11}$ but here with $e_{1.12}$.

b. **The web service execution succeeds**

In this case, as depicted in Figure 6, the web service sends back the result of its execution. The *CEP Communicator* will pass these results to the *CEP Updater*. The latter will update the *CEP Document* by modifying the element $E^v_{1.1}$'s state from $Executing$ to $Committed$. After that, the *CEP Updater* will go through *CEP Document* for the successors' of the current element $E^v_{1.1}$. It will find that there is two elements $E^v_{1.12}$ and $E^v_{1.13}$. The *CEP Updater* will ask the *CEL Generator* to generate the *CEL Documents* of these elements. The latter generates the *CEL Documents* and selects for the elements respectively the engines($e_{1.21}$ and $e_{1.31}$).

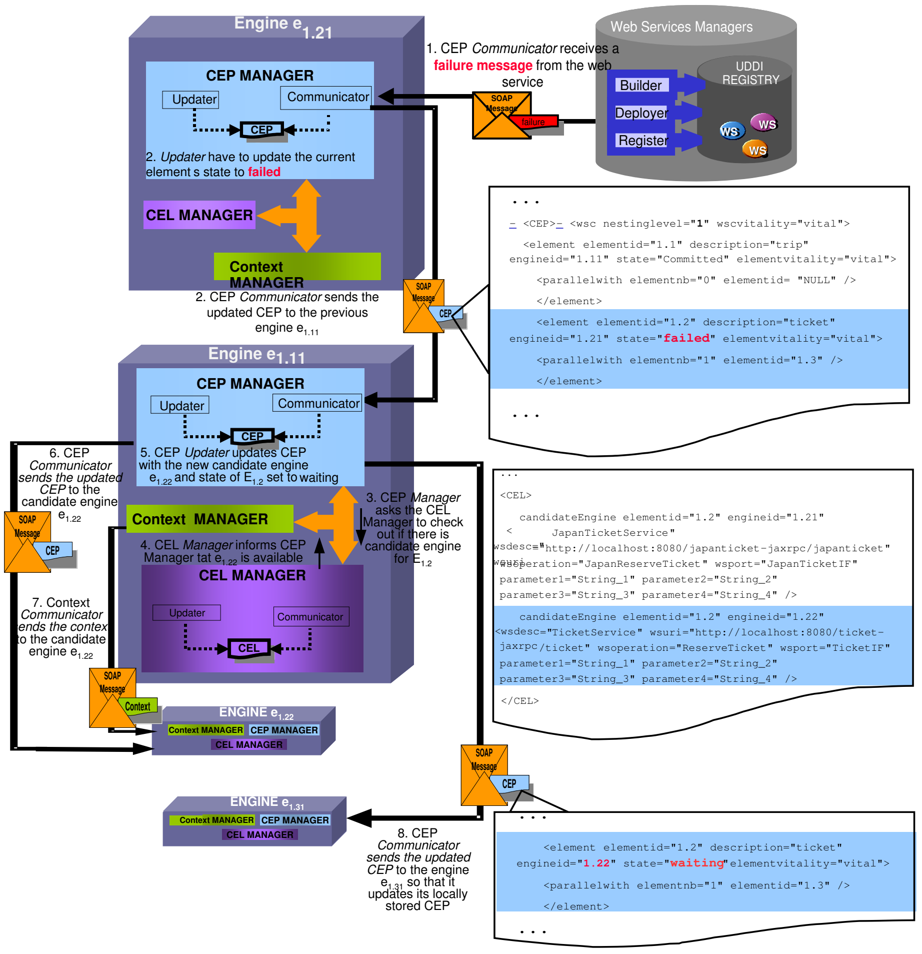
Afterwards, the *CEP Updater* updates the *CEP Document* with the new engines *engineid* allocated. Later, the *CEP Communicator* sends the *CEP Document* to both of the thread Engines $e_{1.21}$ and $e_{1.31}$. The execution process is the same as the above description expect that since $E^v_{1.2}$ and $E^v_{1.3}$ are to be executed in parallel. As we described in [6], the two engines $e_{1.21}$ and $e_{1.31}$ are to be executed simultaneously. For that, we divided the execution in phases, when every thread Engine finishes a phase, it informs the other parallel one. By the end of the execution of both of the elements $E^v_{1.2}$ and $E^v_{1.3}$, the engines $e_{1.21}$ and $e_{1.31}$ will generate the *CEL Document* of their successors(here element $E^{\bar{v}}_{1.4}$). They will also exchange between each other the newly-updated *CEP Document*. After that each *CEP Updater* will update its locally-stored *CEP Document*(i.e., the $e_{1.31}$ updates the element $E^v_{1.2}$ state to $Committed$ and the other way around). The Engines $e_{1.21}$ and $e_{1.31}$ agree on the candidate engine to execute the element $E^{\bar{v}}_{1.4}$ by merging their obtained *CEL Documents* and selecting an engine, $e_{1.41}$ to execute $E^{\bar{v}}_{1.4}$. Subsequently, engine $e_{1.41}$ suspends the Caller Engines here $e_{1.21}$ and $e_{1.31}$ and goes forward in its own execution.

4.4 Vitality Degree Consideration

Up to now, all the executed elements $E^v_{1.1}$, $E^v_{1.2}$ and $E^v_{1.3}$ were with a *vitality degree* attribute in the *CEP Documents* equal to "vital". For that propose, even if they come to fail, their failure is critical and will cause the whole WS-SAGAS failure as we described above for element $E^v_{1.1}$. Otherwise, if element $E^{\bar{v}}_{1.4}$, while being executed by engine $e_{1.41}$, if it happens that the *CEP Communicator* receives a negative message from the web service attached to it, this implies that this element $E^{\bar{v}}_{1.4}$ and contrary to the element $E^v_{1.1}$ failure, will be ignored and the whole WS-SAGAS execution will go ahead. The State of the element $E^{\bar{v}}_{1.4}$ is set by the *CEP Updater* to $Failed$. This Module will check for successors and will found out that the current element $E^{\bar{v}}_{1.4}$ was the last element (e.g. parsing the locally-stored *CEP Document* and looking for elements child from the same WSC returns an empty list). Hereafter, the success of the whole WS-SAGAS success is deduced, the *Context Updater* extracts the results from *Context Document* and informs the Customer of the reached results in a dynamically generated page.

5. Conclusions

In this paper, we have reported on our implemented simulation system of WS-SAGAS transaction model for THROWS distributed architecture. The implementation allowed us to check that the ideas that we have previously proposed in THROWS and WS-SAGAS were feasible using existing technologies namely SOAP, WSDL and UDDI. Moreover, it allowed us to verify that these ideas enhanced notably the reliability through: applying a transactional specification of web services compositions, running them in a distributed environment with a decentralized control and dynamic control delegation, and finally providing them with proper failure detection,



Engine e1.21
CEP MANAGER
Updater
Communicator
CEP
2. Updater have to update the current element s state to failed
CEL MANAGER
Context MANAGER
1. CEP Communicator receives a failure message from the web service
SOAP Message
failure
Web Services Managers
Builder
Deployer
Register
UDDI REGISTRY
WS
2. CEP Communicator sends the updated CEP to the previous engine e1.11
Engine e1.11
5. CEP Updater updates CEP with the new candidate engine e1.22 and state of E1.2 set to waiting
3. CEP Manager asks the CEL Manager to check out if there is candidate engine for E1.2
4. CEL Manager informs CEP Manager tat e1.22 is available
6. CEP Communicator sends the updated CEP to the candidate engine e1.22
7. Context Communicator ends the context to the candidate engine e1.22
Context
ENGINE e1.22
ENGINE e1.31
8. CEP Communicator sends the updated CEP to the engine e1.31 so that it updates its locally stored CEP

```
. . .
– <CEP>– <wsc nestinglevel="1" wscvitality="vital">
  <element elementid="1.1" description="trip"
engineid="1.11" state="Committed" elementvitality="vital">
    <parallelwith elementnb="0" elementid= "NULL" />
    </element>
    <element elementid="1.2" description="ticket"
engineid="1.21" state="failed" elementvitality="vital">
    <parallelwith elementnb="1" elementid="1.3" />
    </element>
. . .
```

```
...
<CEL>
   candidateEngine elementid="1.2" engineid="1.21"
 <      JapanTicketService"
wsdesc="http://localhost:8080/japanticket-jaxrpc/japanticket"
wsuri wsoperation="JapanReserveTicket" wsport="JapanTicketIF"
parameter1="String_1" parameter2="String_2"
parameter3="String_3" parameter4="String_4" />
   candidateEngine elementid="1.2" engineid="1.22"
<wsdesc="TicketService" wsuri="http://localhost:8080/ticket-
jaxrpc/ticket" wsoperation="ReserveTicket" wsport="TicketIF"
parameter1="String_1" parameter2="String_2"
parameter3="String_3" parameter4="String_4" />
</CEL>
```

```
. . .
    <element elementid="1.2" description="ticket"
engineid="1.22" state="waiting"elementvitality="vital">
    <parallelwith elementnb="1" elementid="1.3" />
    </element>
. . .
```

Figure 7 Failure Handling and Forward Recovery

forward recovery and backward recovery mechanisms.

So far, the simulation system featured mainly three kinds of modules, the *Communicators* dedicated to controlling all messages exchange, the *Updaters* responsible for any information retrieval or update, and finally the *Managers* dedicated to supervising the *Updater* and *Communicator* they encapsulate. To allow the parallel execution, we made extensive use of java threads.

On-going work includes finalizing the implementation. In addition, since there is many toolkits presently available for web services development, we will check the applicability of some of them. Finally, other directions we provision to address include the assessment of THROWS architecture performance and scalability.

Acknowledgment

A part of this research was supported by CREST of JST(Japan Science and Technology Agency), MEXT of the Japanese Government, a Grant-in-Aid for Scientific Research on Priority Areas(#15017233) and the 21st Century COE Program "Framework for Systematization and Application of Large-scale Knowledge Resources".

References

[1] I. Foster, C. Kesselman, and S. Tuecke. "The anatomy of the Grid: enabling scalable virtual organizations". International J. Supercomputer Applications, 15(3), 2001.

[2] A.k.Elmagarmid, ed., "Database transaction models for advanced applications", Morgan Kaufmann Publishers, California, 1992.

[3] Open Grid Services Architecture (OGSA), accessed from http://www.globus.org/ogsa/, 2003.

[4] N. Ben Lakhal, T. Kobayashi and H. Yokota, "Distributed architecture for reliable execution of web services", Technical Report of IEICE, DBWS2003 2B, DE2003-24, pp.97-102, 2003-BS-131 (17), pp.129-136, 2003.7.

[5] N. Ben Lakhal, T. Kobayashi and H. Yokota, "WS-SAGAS: transaction model for reliable web-services-composition specification and execution",DBSJ letters Vol.2, No.2. 2003.

[6] N. Ben Lakhal, T. Kobayashi and H. Yokota, "THROWS: An Architecture for Highly Available Distributed Execution of Web Services Compositions", to appear in the proc. of 14th International Workshop on Research Issues on Data Engineering Web Services for E-Commerce and E-Government Applications (RIDE'04), March 28-29, 2004, Boston, USA.

[7] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem, "Modeling long-running activities as nested sagas". Data Engineering Bulletin, vol.14, no.1, pp. 14-18. March.1991.

[8] IBM, "ALL about UDDI", http://www-106.ibm.com/ developerworks/speakers/colan/, April2002.

[9] Sun, "Java Web Services Developer Pack JWSDP", http://java.sun.com/ webservices/ webservicespack.html, Nov.2003.