A Simulation System of THROWS Architecture with WS-SAGAS Transaction Model

Neila BEN LAKHAL[♠] Takashi KOBAYASHI[♦]
Haruo YOKOTA[♥]

With web services composition paradigm advent, 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. To check out our proposal viability and confidence degree, in this paper we propose a configuration of a system that simulates THROWS architecture for executing web services compositions specified as WS-SAGAS transactions.

1. Introduction

The advent of the web services composition paradigm gained a rapid uptake as it meets exactly the requirements of full interoperability. In fact, being based on ever-present protocols and on a set of widely recognized standards (e.g., HTTP, SOAP, UDDI, and WSDL) makes them developer-friendly and not involving a whole learning curve as it is for CORBA, COM/DCOM and so on. However these special features allowed the composition of web services concept to reach a high level of acceptance, in view of the interoperability notable enhancement, there is an actual appeal to address more properly the reliability and availability issues. Specifically, considering that the web services are in essence hosted by different providers, they might have not compliant characteristics (e.g., transactional supports, management model). As a result, the update of any of these services might induce critical unforeseen impacts on the overall composition consistency. Furthermore, there is a need to probe a more elaborated collaboration mode among the different involved web services when executing a composition to not suffer the reliability problems of a centralized collaboration.

Motivated by these concerns, we have proposed in a previous work WS-SAGAS a new transaction model[1][2] for web services compositions specification and THROWS (Transaction Hierarchy for Route Organization of Web Services), a distributed architecture for highly available execution of web services compositions[3].

In order to check out our proposal viability and confidence degree, in this paper we propose a configuration of a system that, first specifies web services compositions as WS-SAGAS transactions, and second simulate the execution of these web services compositions within THROWS architecture. The implementation is heavily based on java programming language and makes extensive use of a set of web services enabling technologies namely, WSDL, UDDI and SOAP.

In the reminder of this paper, we give an overview of WS-SAGAS and THROWS. After, we describe the proposed system configuration: the components and their interactions. We continue with sketching a case study and report on its execution. We end with concluding remarks and future work.

2. WS-SAGAS and THROWS Overview

WS-SAGAS transaction model specifies the web services compositions as a *hierarchy of arbitrary-nested transactions.* These transactions potential execution is provided with *retrial and compensation* mechanisms. WS-SAGAS inherits the arbitrary nesting level, the relaxed ACID properties, the compensation and the vitality degree features proposed in several advanced transaction models[4][5]. To overcome web services Statelessness, we proposed to enrich WS-SAGAS with the *state capturing.* In WS-SAGAS, we defined a *composition* as an *orchestration of elements* (E_i^v). An element has a *state* (S_i) and a *vitality degree* (noted V for *vital* and $\bar{V}$ for *not vital).* According to the considered nesting level, the same element could be either assimilated to an atomic element or to a composition of elements.

THROWS applies a peer-to-peer execution model where the composition execution control is distributed among *dynamically discovered engines.* An *engine* is attached to an involved web service and is allocated to an element. Once allocated, 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.

1. *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 the *CEP*.
2. *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. Implementation

We aim first at showing how the ideas proposed in WS-SAGAS and THROWS can be implemented using existing technologies namely WSDL, UDDI, SOAP and so on. Second, we endeavour to show to what extent these ideas can contribute considerably in reliability enhancement. For a better understanding of the implemented system, a case study is chosen.

3.1 Scenario

For homogeneity's sake, we kept using the same travel reservation scenario that we have described in our former work in [2]. We have chosen a process involving four activities: a trip information registration

[♠]Student Member Graduate School of Information Science and Engineering, Tokyo Institute of Technology neila@de.cs.titech.ac.jp
[♦]Regular Member Global Scientific Information and Computing Center tkobaya@cs.titech.ac.jp
[♥]Regular Member Tokyo Institute of Technology, Global Scientific Information and Computing Center yokota@cs.titech.ac.jp

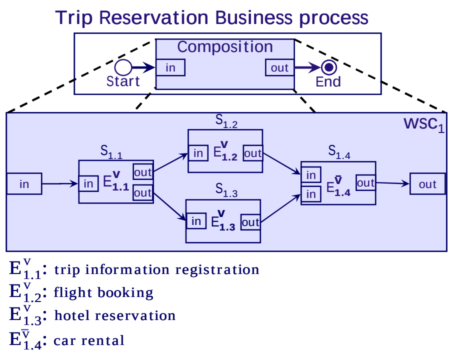

Figure.1 WS-SAGAS Specification

($E_{1.1}^{V}$) activity, a flight booking ($E_{1.2}^{V}$) activity, a hotel reservation ($E_{1.3}^{V}$) activity and finally a car rental ($E_{1.4}^{V}$) activity (process depicted as WS-SAGAS in Figure.1).

3.2 Implementation Tools Description

The implementation specifies the web services compositions as WS-SAGAS transactions and simulates their execution in a logically distributed version of THROWS architecture. We are using mainly java programming language and a set of web services enabling technologies: we make extensive use of the different APIs provided in Java Web Services Developer Pack (JWSDP) (JAXP, JAX-RPC, SAAJ, and JAXR):

– All the web services that we need for our 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;
– Communication among modules uses the Soap with Attachments API for Java (SAAJ);
– Depending on the composition execution stage, the exchanged SOAP messages encapsulate different kind of XML documents (parsed with JAXP and manipulated JDOM and DOM).

3.3 System Components Description

Our system features four key modules, *Web Services Managers, Engines, a User Conversation,* and *a WS-SAGAS Specification Generator*.

3.3.1 The Web Services Managers (WSM): The WSMs are not part from the original THROWS architecture as we define it. In fact, formerly THROWS was proposed to work with services already hosted on the web services providers' sides. Implementing such a system would require building a physically distributed system and indeed, dealing with the potential failures of the underlying infrastructure (e.g., message lost, timeout) from which it may suffer, is needed. This guided as to the following choices. First, building a logically distributed system and focusing mainly on detecting and recovery from semantic failures (i.e., web services/engines failures). Second, considering that the recovery from other system-related failures is beyond the scope of the present system aims. As a direct consequence, all the services need to be locally stored and published. Hence the Web Services Managers coming into play.

Each *WSM* is responsible for new web services definition, existing web services handling (e.g. update, delete, etc.), web services publishing, discovery, invocation, and finally web service failure information and communication. Each *WSM* consists of three modules:

– ***Services Builder:*** uses mainly JAX-RPC to define the new services (endpoint, clients and WSDL).
– ***Services Deployer (SD)****:* Deploy/undeploy the web services in/from the web container (we use Tomcat).
– ***Services Register (SR)****:* After deployment, the web services need to be published. The *SR* registers the web services in the UDDI registry. Besides, it is responsible of handling any received web service discovery query or web service invocation request. Actually it acts as an intermediate in-between the web service invoker and the invoked web service. In case the invoked web service fails, the *SR* have to generate a failure notification message so that the invoker gets informed about the failure and doesn't hang waiting for a response that would never come.

3.3.2 The User Conversation (UC): Is used as an entry point to input the potential user choices that will be considered as the web services compositions execution context (i.e., the necessary inputs for the web services execution and their outputs). These inputs are saved in the context, which is an *XML document.*

3.3.3 The Engines (e_i): This module is at the core of our implementation. Each engine contains three types of *Managers*: a *Context Manager*, a *CEL Manager*, and a *CEP Manager*. Likewise, each of these *Managers* encapsulates an *Updater* and a *Communicator*. The *Manager* supervises both of the *Updater* and the *Communicator*. The *Updater* is responsible for any update, modification, and information retrieval (e.g., element/engine/context selection, CEP/context update).The *Communicator* is responsible for any message exchange. It encapsulates received data within SOAP messages to send them or moreover, it extracts data from the received SOAP messages (e.g., Context/CEP propagation, services discovery/results communication, services invocation). We distinguish between external communication (between peer *Engines*) and internal communication (between the different *Managers* of the same *Engine*).We detail in what follows the different *Managers* descriptions and their functionalities:

– ***The Context Manager:*** is responsible for updating the context using a *Context Updater*. The update is performed on the base of the inputs received from the customer request or on the execution results of the attached web service to the *Engine*. Besides, a *Context Communicator* module propagates, when enquired, the context document to the web service. Actually, the context is encapsulated in a SOAP message and handed over to the *Web services Manager*, which will hand it to the web service.
– ***The CEL Manager:*** is in charge of generating the CEL as an *XML document*. As we already described it in [3], we remind that the CEL is necessary for the execution control delegation between the different *Engines*. As soon as an *Engine* finishes successfully the execution of its allocated element (from a WS-SAGAS specification) by executing its attached web service, it has to generate the CEL(s) of its direct successor(s) (in the same WS-SAGAS). For that purpose, based on

the element *description* (i.e. retrieved from the WS-SAGAS specification), the *CEL Communicator* is responsible for issuing a query to the *Web Services Manager* (in web services jargon, commonly known as web service discovery). The latter takes care of querying the UDDI Registries for web services with the desired functionalities. For each web service description returned in the query result, the *CEL Updater* allocates an *engineid* (coupled with the web service information) and adds it as a new candidate engine to the *CEL document*.

– ***The CEP Manager:*** monitors the *CEP* and depending on the web service execution state progress; the *CEP Updater* updates the state of the current element in the *CEP document*. The update from a state to another one is done on receiving a particular SOAP message (e.g., service success, failure notification). A *CEP Communicator* is responsible for receiving the SOAP message, for extracting the service failure or success information from the message and finally for communicating the information to the *CEP Updater.* Some other SOAP messages are also received from other *Engines* that may entail as well the *CEP* update (e.g., requests for compensation, abortion). The *CEP document* update and the element's state transitions are detailed in [2] [3].

***3.3.4 The WS-SAGAS Specification Generator*:** It takes as an input the diagram depicting the WS-SAGAS and outputs the context and the CEP described as XML documents. So far, we are still implementing this module.

4. Description of Scenario

We will sketch in what follows the execution of the trip reservation scenario within our implemented system. A more detailed description is provided in [6]:

4.1 Customer Request Submission

A customer accesses the *User Conversation* and inputs his request (destination, departure date and return date and his name). Submitting the request entails saving the inputted values in a *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 from the context and communicated to the customer as a result for his request.

4.2 Customer Request Handling

A servlet running on the server-side discovers the new *Context Document* provided and handles it as follows:

***4.2.1 Element Selection*:** a current element is chosen (the first element from the WS-SAGAS specification, here $E_{1.1}^{V}$) from the *CEP Document*. The task of going through this *CEP Document* for selecting elements is attached to the *CEP Updater*. After an element ($E_{1.1}^{V}$) has been selected, a *CEL Document* needs to be generated.

***4.2.2 CEL Generation*:** The *CEL Updater* receives, as inputs from the *CEP Updater*, a *description* of an element (here $E_{1.1}^{v}$ description is provided) which is used to generate/create a query that will be submitted to the *Web Services Managers*.

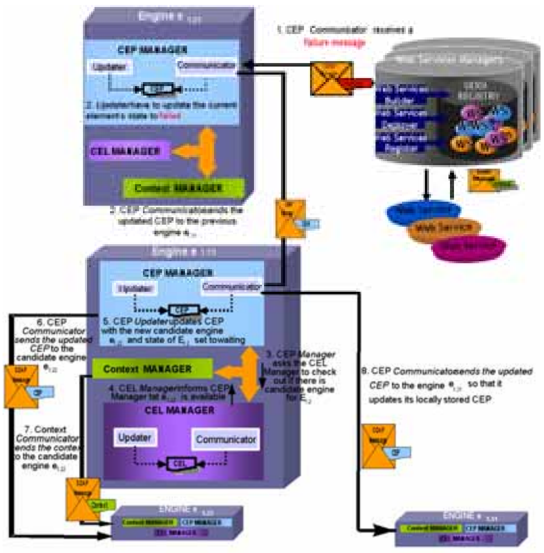

Figure. 2 Failure Handling with Execution Retrial

***4.2.3 Web Services Discovery*:** SOAP messages that contain the query results are sent back to the *CEL Manager.* They will be parsed for details about the services and used to generate the *CEL Document*.

4.2.4 Engine Selection: It is performed by the *CEL Updater.* An *Engine* is chosen and the *CEP Updater* is informed so that the *CEP Document* is updated with the new value of the *engineid* (the *engineid* of the element $E_{1.1}^{v}$ is changed from *null* to $e_{1.11}$). The Selected engine is labelled as already allocated (here *engineid* 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-labelled one will be chosen (e.g., the next to $e_{1.11}$).

***4.2.5 Control Delegation*:** A new thread *Engine* $e_{1.11}$ receives the *CEP* and *Context Documents* and stores them locally. A response is sent back to notify that the documents were received and that the element execution is launched.

***4.2.6 Context Selection*:** The *Context Updater* extracts from the *Context Document* the context necessary for the web service invocation (for $E_{1.1}^{v}$, it contains four inputs (*destination, departure date, return date, name*) with their values (received in the customer request)).

***4.2.7 Web Service Invocation*:** The *Context updater* provides this context to the *CEP Communicator.* The latter will invoke the web service client using this *Context*. 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:

- *Web service execution failure (Figure2.)*: If a fixed period of time is elapsed and the *CEP Communicator* doesn't receive any response, or it receives a message that informs about the web service failure from the *Web Services Manager,* the web service failure is deduced. The current thread *Engine* $e_{1.21}$ needs to deduce its own failure and to delegate the execution control to the previous engine thread. The *CEP Updater* updates the current element state to *Failed* and the *CEP Document* is communicated to the previous *Engine* thread. The locally stored *CEP Document* is updated ($E_{1.2}v$'s state set to *Failed*), the *engineid* set again to *null* and the *CEL Updater* is asked for selecting another candidate engine. If the *CEL Document* is empty, a backward recovery is performed: the *CEP Communicator* will send a compensation request to all committed elements and a compensation request to all elements that are still executing（e.g. $e_{1.31}$). Finally, a message is sent back to the *User conversation* notifying the customer of the failure to satisfy his request. If the *CEL Document* is not empty, a forward recovery is performed with execution retrial: another *Engine* is selected. In the *CEP Document*, $E_{1.2}^{v}$'s state will be modified from *Failed* to *Waiting* and the *engineid* will be set to $e_{1.22}$. After, the updated *CEP Document* is sent again to all the *Engines* that are already executing. The execution will be resumed with thread *Engine* $e_{1.22}$.

- *The web service execution success:* The web service results of execution are received by the *CEP Communicator* and handed over to the *CEP Updater.* The latter will update the *CEP Document* by modifying the element state from *Executing* to *Committed*. After that, the *CEP Updater* will go through the *CEP Document* for the successors' of the current element and will later ask for generating the *CEL Documents* of these elements. Engines are selected and the execution is resumed.

4.3 Vitality Degree Consideration

In the above scenario, we described the execution of a vital element (e.g., $E_{1.1}^{v}$). When s vital element fails, its failure is critical and will cause the whole WS-SAGAS failure (as we described above $E_{1.1}^{v}$). If a not vial element (e.g., $E_{1.4}^{\bar{v}}$), while being executed, if it happens that the *CEP Communicator* receives a failure message, this implies that the element failure is to be ignored and the whole WS-SAGAS execution will go ahead($E_{1.4}^{\bar{v}}$ state set *Failed).*

5. Conclusion

In this paper, we have proposed a configuration of a system that first specifies web services compositions as WS-SAGAS transactions, and second simulate the execution of these compositions in THROWS 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 forward and backward recovery mechanisms.

So far, the implemented 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. Finally, other directions include the assessment of THROWS architecture performance and scalability.

[References]

[1] Neila BEN LAKHAL, Takashi KOBAYASHI and Haruo YOKOTA, "A Distributed architecture for reliable execution of web services," *Technical Report of IEICE,* DBWS2003 2B, DE2003-24, pp.97-102, 2003-DBS-131 (17), pp.129-136, 2003.7.
[2] Neila BEN LAKHAL, Takashi KOBAYASHI, and Haruo YOKOTA, "WS-SAGAS: A transaction model for reliable web-services-composition specification and execution", DBSJ Letters, Vol.2, No. 2, pp.17-20, 2003.10.
[3] Neila BEN LAKHAL, Takashi KOBAYASHI, and Haruo YOKOTA, "THROWS: an architecture for highly available distributed execution of web services compositions", in Proc. of RIDE WS-ECEG'2004, the 14th International Workshop on Research Issues on Data Engineering: Web Services for E-Commerce and E-Government Applications, pp.103-110, 2004.3
[4] A.k. Elmagarmid, ed., "Database transaction models for advanced applications," Morgan Kaufmann Publishers, San Mateo, California, 1992.
[5] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem, "Modeling long-running activities as nested sagas". *Data Eng. Bul., vol.14, no.1,* pp.14-18. March.1991.
[6] Neila BEN LAKHAL,, Takashi KOBAYASHI, and Haruo YOKOTA,"A Simulation system of THROWS Architecture for WS-SAGAS", in Proc. of 14th IEICE Data Engineering Workshop 2004, IEICE, 7-B-4, 2004.3

Neila BEN LAKHAL
She received a BS in computer science applied to management from High Institute of Management of Tunisia in 2000 and a M.E. in computer science from Tokyo Institute of Technology in 2004. She is currently a doctoral course student in the Graduate School of Information Science and Engineering of Tokyo Institute of Technology. She has been working on transaction processing and distributed workflow systems failure recovery.

Takashi KOBAYASHI
He received a B.E., M.E., and Dr.Eng. degrees in computer science from Tokyo Institute of Technology in 1997, 1999, and 2004. He is a research associate of Global Scientific Information and Computing Center, Tokyo Institute of Technology. His research interests include software patterns, software architecture and information retrieval. He is a member of IPSJ and JSSST.

Haruo YOKOTA
He received the B.E., M.E., and Dr.Eng. degrees from Tokyo Institute of Technology in 1980, 1982, and 1991, respectively. He joined Fujitsu Ltd. in 1982, and was a researcher at ICOT for the Japanese 5th Generation Computer Project from 1982 to 1986, and at Fujitsu Laboratories Ltd. from 1986 to 1992. From 1992 to 1998, he was an Associate Professor in Japan Advanced Institute of Science and Technology (JAIST). He is currently a Professor at Global Scientific Information and Computing Center in Tokyo Institute of Technology. His research interests include general research area of data engineering, information storage systems, and dependable computing. He is a chair of Technical Group of Data Engineering in IEICE and a member of IPSJ, JSAI, IEEE, IEEE-CS, ACM and ACM-SIGMOD.