Evaluation of Large Scale Web Service Compositons

Thiago Rodrigues Colucci

Advisor: Fabio Kon

December 1, 2010

Contents

Student: Thiago Rodrigues Colucci
Advisor: Prof. Dr. Fabio Kon

1 Introduction

With the development of the Internet, several Web Applications (such as Gmail, Google Docs, and Amazon.com) became very popular. Those applications use many Web Services for decoupling its parts. With plenty of services to use, we need some technique to compose those Web Services into new ones with a higher level of abstraction. Web Service Orchestrations and Web Service Choreographies arose in this context as techniques to create new services using only description languages and pre-existing Web Services.

Some of the advantages of using Web Services compositions are:

- interoperability, because Web Services are platform independent
- scalability, since they are loosely-coupled and can be ran on a distributed environment
- reuse, as there are already several open Web Services available to use
- minimal development, as new services can be created by using simple declarative languages.

The growing importance of these composition techniques motivated this work. We will study the differences between Orchestrations and Choreographies and also evaluate synthetical compositions. These evaluations will assist the development of the *Baile project*, under development at the Institute of Mathematics and Statistics from the University of São Paulo (IME-USP) in partnership with Hewlett-Packard (HP) of the United States, as well as the *CHOReOS project*, in which IME-USP is in partnership with an European consortium.

Baile's main goal is to study the problems related to the development of choreographies in large scale environments, particularly in the context of Cloud Computing. The main objective of CHOReOS, for USP, is the development of a Service Oriented Middleware for the "Internet of the Future", an *Ultra-Large Scale internet*, heterogeneous and dynamic.

Also, by creating these compositions, this work will help with the study of problems for Baile and to show how scalable the current techniques of choreographies and orchestration are, later helping CHOReOS to understand the current state of the art of these Web Services composition techniques.

This work is divided in three parts: first we explain the technologies used and studied, then we describe how the activities were executed, and finally we show the produced code and results obtained.

2 Used Technologies

In this section we will describe and explain the main technologies used in this work.

2.1 Web Services

Web Services are a means to enable the interoperability among application components over the Internet. They are defined as a set of W3C standards, normally using HTTP as the transport layer and XML serialization for data interchange.

They are fundamental components of a web based system that uses the programming model called Service Oriented Architecture (SOA). In this model, services are self-describing and self-contained, leading to the development of distributed systems based on loosely-coupled distributed modules.

Interoperability. When using a Web Service, one must not concern about its platform, because it is not executed on the client's side, all the communication with it is done through HTTP, and the message's serialization is done using XML. These technologies are fully platform independent, therefore Web Services inherit the interoperability of their foundations.

Reusable application components. Applications often need common components such as currency conversion, weather reports, or even language translation. Web Services can offer those application components, making them easily interoperable and reusable. [W3Schools (2008)]

Connection of existing software. Web Services can help solve interoperability problems between legacy and new software by giving different applications a common and standard way to communicate and exchange data. With Web Services one can exchange data between different applications and different platforms. [W3Schools (2008)]

2.1.1 Web Services Foundations

The main technologies that compose a Web Service are:

Extensible Markup Language (XML), which is a set of rules for encoding documents in a machine-readable form. It is defined in the XML 1.0 Specification written by the W3C, and several other related specifications. It provides a language that can be used across different platforms and programming languages, and still express complex messages and functions. [W3C XML Working Group (2008)]

Hypertext Transfer Protocol (HTTP), which is "an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes, and headers. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred". [Fielding *et al.* (1999)]

Web Service Description Language (WSDL), which is "an XML format for describing services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow the description of endpoints and their messages regardless of what message formats or network protocols are used to communicate". [Christensen *et al.* (2001)]

Simple Object Access Protocol (SOAP), which is "a lightweight protocol for the exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses. SOAP can potentially be used in combination with a variety of other protocols; however, the only bindings in its definition describe how to use SOAP in combination with HTTP and HTTP Extension Framework. It does not define, for itself, any application semantics such as programming model or implementation specific semantics; rather it defines a simple mechanism for expressing application semantics by providing a modular packaging model and encoding mechanisms for encoding data within modules. This allows SOAP to be used in a large variety of systems ranging from messaging systems to RPC".[Box *et al.* (2000)]

2.2 Orchestrations

Orchestrations describe how Web Services can interact with each other at the message level, including the business logic and execution order of the interactions. These interactions may span applications and/or organizations, resulting in a long-lived, transactional, multi-step process model. In this model of service composition, there is a central node that controls the logic of the entire process. This node has the responsibility to manage the process, choosing which services will be invoked, how messages are exchanged between these services, and how to proceed in face of possible exceptions and failures. Figure 1 shows the message exchange between a client, a travel agency (the central node) and two Web Services.

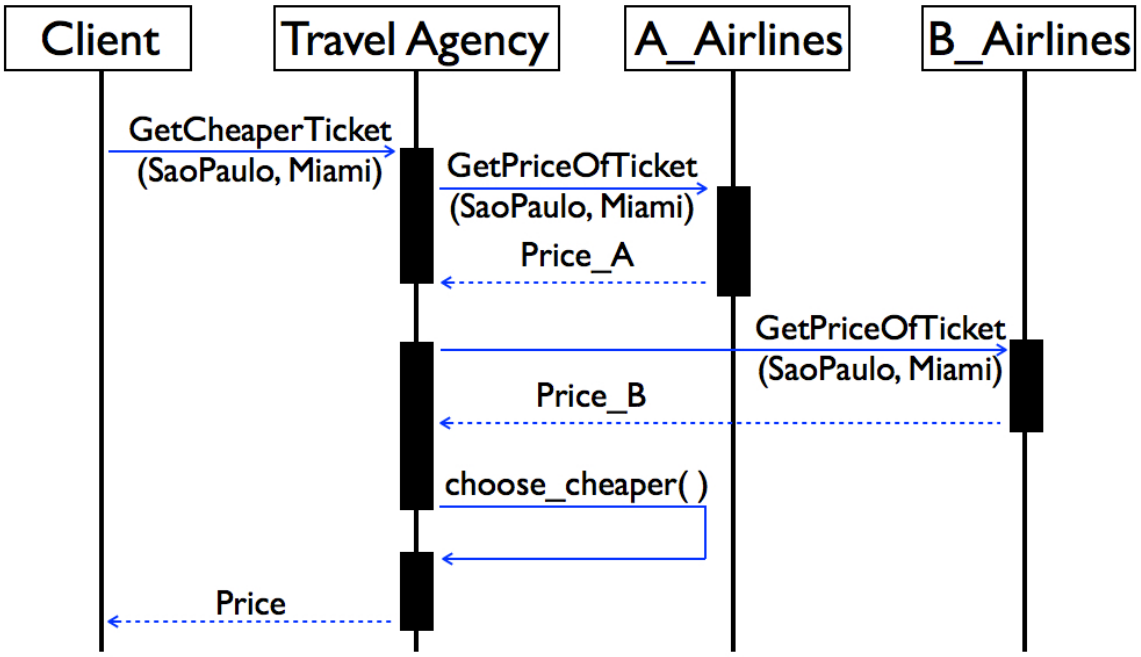


Figure 1: *Web Service Orchestration example flow*

While this could be achieved in many different ways, orchestrations are usually managed with standardized declarative languages that describe the whole process and which are, in turn, automatically transformed into executable code. Typically, one must first describe the orchestration through an XML document, then generate an executable code, and finally execute the process. A good analogy would be a *C* program, with only the *main* function (orchestration definition) that uses a lot of functions (operations) of included libraries (partner Web Services). The whole logic is embedded in this one description, the central node. The main difference is that instead of a procedural language, the orchestration is described in a higher level description language.

The industry is embracing orchestration as the way to specify, generate, and control business

processes built upon Web Services. The acceptance of the industry came from the cooperation effort among companies (such as IBM, Microsoft, Sun, BEA, and others interested in the development of the Web) to standardize composition techniques. From this effort, three candidate standards arose:

BPEL4WS stands for "Business Process Execution Language for Web Services"; this specification — called BPEL, for short — models the behavior of Web Services in a business process interaction. It provides an XML-based grammar for describing the control logic required to coordinate Web Services participating in a process flow. WSDL interfaces define the specific operations allowed and BPEL defines how to sequence them. In this sense, BPEL works as is a layer on top of WSDL. WSDL describes the public entry and exit points for every BPEL process; at the same time, WSDL data types describe the information that passes between process requests. Additionally, WSDL can reference external services that the BPEL process requires. A simple BPEL model is shown at Figure 2.

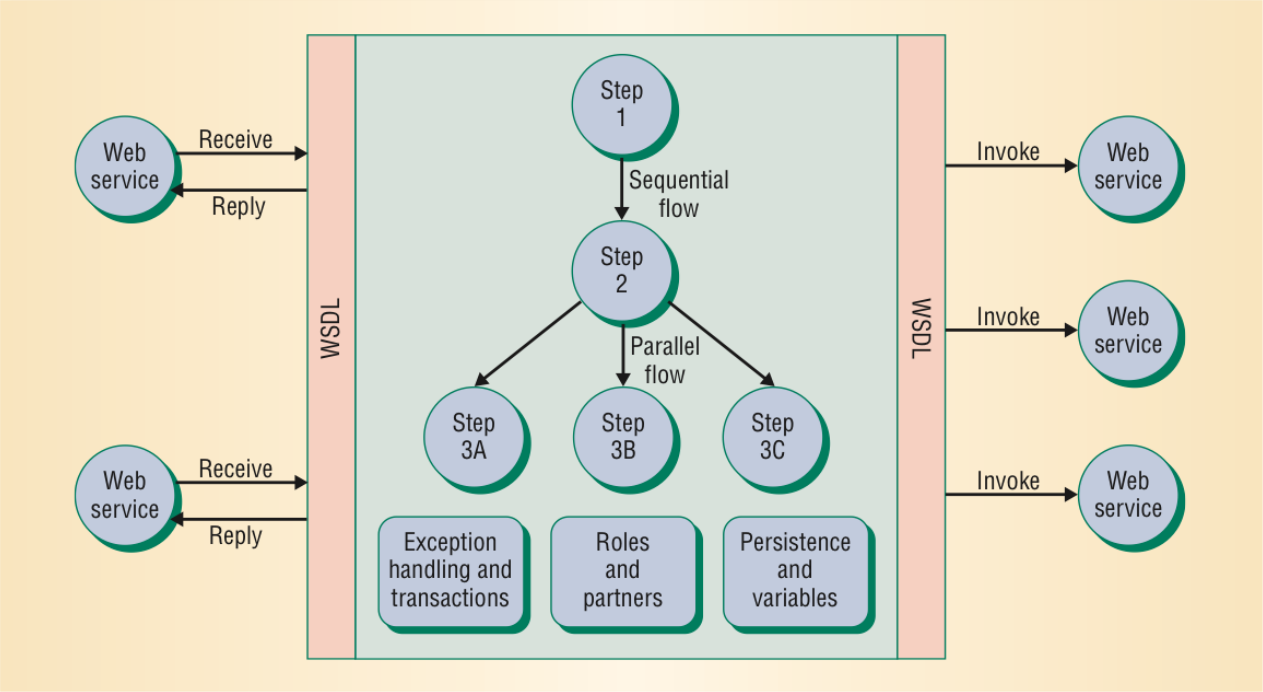


Figure 2: *BPEL structure [Peltz (2003)]*

BPML the "Business Process Modeling Language" was designed to be semantically complete according to the Pi-calculus formal representation of computational processes. In this sense, it is a superset of BPEL. But, because of its formality, companies such as IBM and Microsoft could not implement BPML into their existing workflow and integration engine implementations (BizTalk, Websphere etc.). Hence, they pushed for a simpler language.

In the beginning of 2003, all the companies supporting BPML discontinued its support, migrating to BPEL4WS. BPEL4WS subsequently became a OASIS standard, renamed to "WS-BPEL". This change was made for it to be aligned with other Web Services standards of the organization. [OASIS (2010)]

Nowadays the main standard is still BPEL, since its definition, a lot of graphical tools, proprietary and open source, were created to aid in developing with it. But even though there are a lot of good tools for modeling the process (e.g., NetBeans, Eclipse, Petals studio, Orchestra, and many others), there are not so many good tools for its deployment.

2.2.1 Modeling Tools

There are several available tools for process modeling. In this work, we considered:

Eclipse is the Rich Client Platform behind one of the most popular Java IDE in the market, the JDT. Its plugin architecture allows it to become an IDE for anything. With the BPEL Designer plugin, Eclipse gains the ability to display and edit BPEL workflows with a graphical UI.

NetBeans which, like Eclipse, is originally a Java IDE that supports plugins. The Enterprise Edition version 6.5 brings a bundle that supports the development of Web Services and Orchestrations. This is another graphical designer for BPEL, but the code it produces is more easily readable and editable by human beings, so we used it to generate the base orchestration of the generator (described later). NetBeans has also a GUI based module to execute the process.

2.2.2 Service Engines

In order to execute the orchestrated process, one needs a service engine, that loads the services specifications and generates the corresponding code. In this work, we considered:

Apache ODE is a WS-BPEL-compliant Web Service orchestration engine. It organizes Web Service calls following a process description written in BPEL. Another way to describe it would be a Web Service-capable workflow engine. This engine is the most used open source engine for orchestrations available, but the Apache's license is not compatible with ours (GPL), so this was not the preferred tool for the project.

Orchestra is another open source BPEL engine. Its license is compatible with ours (GPL), but we could not find an easy way to deploy the services there.

PEtALS ESB is an Open Source Enterprise Service Bus for large SOA, Service-Oriented Architecture. [PetalsLink (2010)] An ESB is a tool that integrates existing services or applications (those that are exposed to the Bus as a service) using standard service languages, resulting in greater adaptability and automation. PEtALS ESB also has a module for executing BPEL. This was the the tool we chose as it enabled us to create synthetical services in each node of the Bus and execute the composition on the Amazon Elastic Compute Cloud (Amazon EC2).

2.3 Choreography

Web Service Choreography is a form of service composition in which there is no single node that controls the entire process, i.e., control is distributed throughout the system. In a choreography, the interaction protocol among several partner services is defined from a global perspective. At run-time, each participant in a choreography performs a role and interacts with its neighboring participants. As there is no single point of control, and therefore, no single point of failure, there is a better potential for building scalable systems than in the case of centralized orchestrations.

The name of the composition came from the analogy of the process with dancers. While in an orchestra there is the Conductor that sets the timing and rhythm, in a group of dancers there is no coordinator during the enactment; dancers only interact with their neighbors to perform their roles.

There have been a few attempts to create standards:

WS-CDL stands for "Web Services Choreography Description Language"; it is a W3C candidate recommendation. It is a language for describing how peer-to-peer participants collaborate. The language uses XML, and some aspects are inspired by pi-calculus.

WSCI is the "Web Service Choreography Interface". It is an XML-based interface description language that describes the flow of messages exchanged by a Web Service participating in choreographed interactions with other services. It describes the dynamic interface of the Web Service participating in a given message exchange by means of reusing the operations defined for a static interface (typically defined in WSDL). WSCI works in conjunction with the Web Service Description Language (WSDL). [BEA Systems *et al.* (2002)]

BPMN "Business Process Modeling Notation" is a graphical representation defined by the Object Management Group (OMG) for specifying business processes in a business process model. The new version (2.0), still in beta phase, has a set of construct blocks to specify choreography processes. This new version will also have execution semantics, which will enable the future development of tools to enact choreographies.

Although there have been some attempts by the W3C and OMG to specify standard languages to define choreographies, up to the moment there are no implementations that allow the enactment of choreographies specified in standard languages.

2.4 Cloud Computing

Recently, a new generation of middleware systems and hardware infrastructures were developed to cope with applications that requires large processing power. In this case, the motivation was end-user applications such as email, office suites, calendars, and many other e-commerce, social networks, and Web 2.0-style applications used daily by hundreds of millions of users. Large Internet-based companies such as Amazon and Google used virtualization technologies to develop the basis for what was later called Cloud Computing [Zhang *et al.* (2010)]. Google App Engine provides an execution environment for Web applications that can then be executed in the hardware infrastructure provided by Google; developers can write applications in Java or Python and use standard APIs for storage and communication. The Amazon Elastic Compute Cloud or EC2 [AMAZON (2010b)] provides a virtual computing environment in which developers can instantiate multiple virtual machines booting, from scratch, standard operating systems such as GNU/Linux or Windows. These sets of machines can then run any application developed for these operating systems. When necessary, the developer can also create an image from a current instance of his virtual machines set and then create new instances with the that image, shortening the configuration time for these new machines.

2.5 Conclusions

Orchestrations, choreographies, and cloud computing are increasingly important technologies in high-scale computing today. They offer both competing and complementary approaches to specific scalability and reuse problems.

Orchestrations and choreographies provide two different approaches to the problem of distributed services composition and process definition.

> Orchestration always represents control from one party's perspective. This differs from choreography, which is more collaborative and allows each involved party to describe its part in the interaction. Choreography tracks the message sequences among multiple parties and sources — typically the public message exchanges that occur between Web Services — rather than a specific business process that a single party executes. [Peltz (2003)]

A graphical view of the relationship described by Peltz is represented in Figure 3.

Cloud computing offers automated techniques for distributed systems deployment and high availability with ease. Taking Amazon as an example, their services ensure that the number of Amazon EC2 instances one is using scales up seamlessly during demand spikes to maintain performance, and scales down automatically during demand lulls to minimize costs, using pre-defined rules based upon CPU, RAM, and Network usage. Also, it is trivial to, for example, build scripts to scale the service

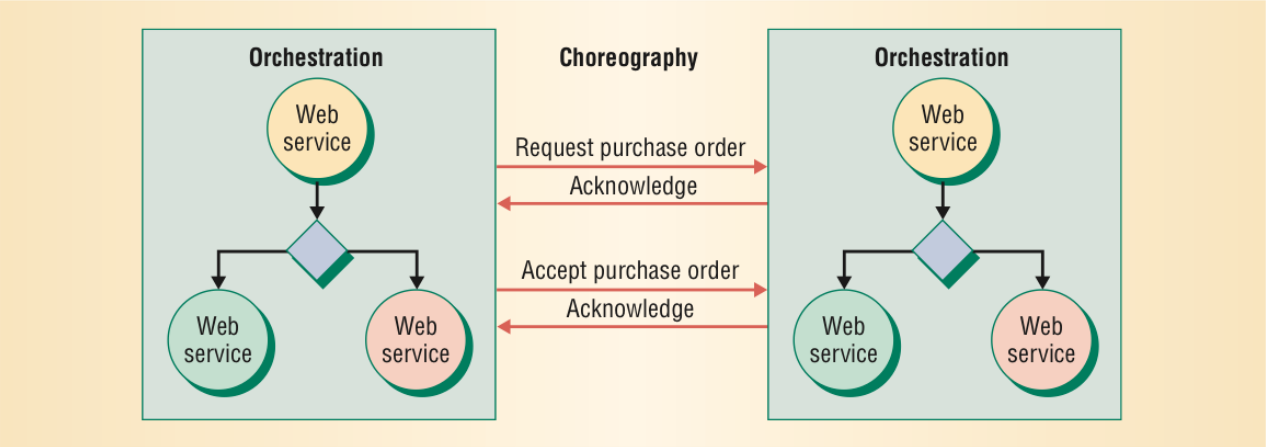


Figure 3: *Relationship between Orchestrations and Choreographies [Peltz (2003)]*

up at day time and later scale it down at night time, when there might be less people accessing the service. [AMAZON (2010a)]

3 Objectives and Activities

This work targets the evaluation of the scalability of the two kinds of composition explained above (Web Services Orchestration and Choreography). To make this assessment, we first needed a composition to evaluate. Thus, we designed a synthetical orchestration with the topology of a complete tree, in which there are two kinds of services: "Node" and "Leaf".

The root of the tree is a "Node" and each of its children is another synthetical orchestration. This means they are either "Nodes" (then, recursively, its children will be other synthetical orchestrations) or "Leaves" (these do not have children).

The tree is parametrized by the number of children per "Node" and by its depth (the distance from the root to each "Leaf"). The message received by the root (from a client) is propagated, in a sequential way, through its children until it reaches the leaves. When a "Leaf" receives the message, it sends the data back to its sender, which in turn sends the message back and so on, until it arrives at the root, which replies back to the client. A simple example with 3 children per node and depth of 2 is shown in Figure 4.

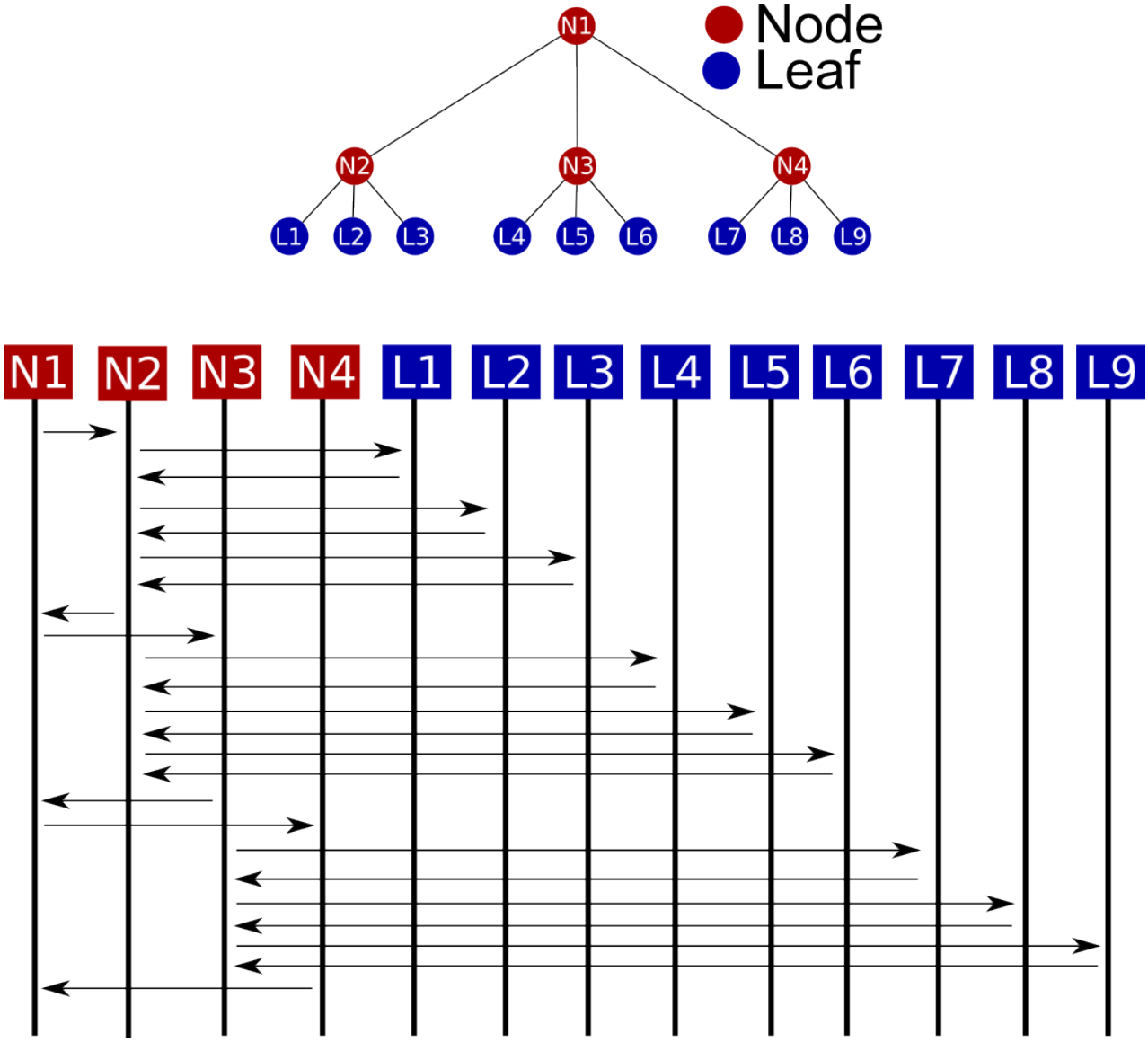


Figure 4: *Example of a composition with 3 children per node and depth of 2 and its flow of messages*

3.1 Composition Generator

To generate the tree structure described above, at first we developed "hardcoded" orchestrations using *Eclipse with BPEL Designer plugin*, but it alone could not execute the modeled process. Then we began a search of engines for orchestrations. One big requirement for the tools we were going to use was that they should be Open Source, GPL or LGPL, so that they could meet the required licenses of Baile and CHOReOS projects (GPL). The first service engine we came across was *Apache's ODE*. At first glance it worked well, but it could not compile the Eclipse generated processes and it is licensed under Apache License, incompatible with ours.

As ODE could not compile the generated orchestrations, we looked at the code generated by Eclipse looking for an issue, but it turned out that the code was unreadable for human beings. Given that we would use the generated code to create our generator, we continued the search for another development tool and another engine as well. We found *NetBeans Enterprise Edition version 6.5* that was known for its good usability and intuitive Graphical User Interface (GUI). We redesigned the Web Service orchestrations in NetBeans; the re-work was indeed easy to do and the generated code was human readable. The search for an engine was a little more complicated and we were already spending more time than planned on it, meanwhile we discovered how to make NetBeans execute orchestrations. So we postponed the search for an engine in order to develop the synthetical orchestrations.

Now that we had a fully working set of orchestrations, we needed a way to automate the whole process of preparing, generating and deploying the test environment. Thus, we developed a Ruby script (the "generator") to generate synthetical orchestrations using TDD. However, in this specific case, the test code had literal *strings* exactly as the tested code did. This similarity cast some doubt as to whether TDD was really helping the development or not. We have therefore decided not to use this technique at that time. When the development phase was over, we concluded that TDD *could* have helped us to find bugs and give us more confidence to make big refactorings.

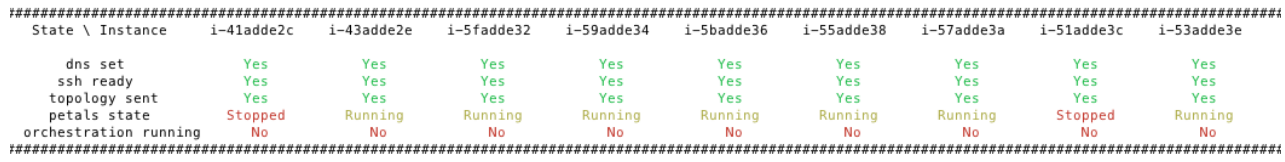
When ready, the generator was able to create orchestrations that were executable within Netbeans. The script generated the composition tree and also, for each node of the tree, a set of WSDL and BPEL files corresponding to the service of that node. Then we started to study Amazon EC2 and to improve the Ruby script to instantiate virtual machines on EC2 for each node of the tree. At this moment, we realized that the code was not suitable for these varying test scenarios and decided it was time to refactor.

One big discovery we made at that time was that NetBeans could not execute the orchestrations from a command-line shell, it required a GUI for the execution process, which made its use no longer feasible anymore due the fact that the only interface with Amazon EC2 instances is through *ssh*. We had to get back to the search for an orchestration engine. Meanwhile the generator was instantiating all the machines we would needed for the composition's deployment, but we also discovered that we had a quota in Amazon that limited us to use only 20 instances at a time.

Orchestra was the next tested engine. Even though we found this engine through good recommendations about it in forums, we could not execute any orchestration in it, and its documentation and support was very poor. The development was fully stagnant until we could find a usable engine. After a couple of weeks searching in more forums and papers, the solution came from the europeans in CHOReOS project. They suggested that we use PEtALS ESB, and this tool was excellent for us. Not only were we unable to find any other good open source engine — and the only we knew that worked, Apache's ODE, had an incompatible license — but also PEtALS ESB had the simplest install and setup process among all the servers studied; this was the most essential feature to build the automated process. We started the experiments on the standalone mode of PEtALS, which is an ESB with just one node.

PEtALS executed the generated orchestrations seamlessly on the standalone version. We were back to the development of the generator and deployer for Amazon EC2. After some adjustments, the generator was creating the tree, instantiating virtual machines on Amazon EC2, creating the compositions, configuring each instance to become a node of the PEtALS Bus, starting the PEtALS server in each node, and deploying the services on the ESB. This whole process is parallelized for each node, and the steps of the process can be followed in the shell's standard output, as shown in Figure

5.



State \ Instance	i-41adde2c	i-43adde2e	i-5fadde32	i-59adde34	i-5badde36	i-55adde38	i-57adde3a	i-51adde3c	i-53adde3e
dns set	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
ssh ready	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
topology sent	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
petals state	Stopped	Running	Running	Running	Running	Running	Running	Stopped	Running
orchestration running	No	No	No	No	No	No	No	No	No

Figure 5: *Example output from running the generator script for a tree with depth of 1 and 10 children per node*

When deployment is done, the generator outputs the URL of the process entry point (*Public DNS* of Root Node), on which port it is reachable, the path to the service, and the ID of this root node (to compose the SOAP messages).

Although all the phases have been completed, the connection from outside of the cloud was very unstable and unpredictable. This is likely what caused us to receive SOAP faults when sending messages to the compositions instantiated on the cloud. We contacted the developers of PEtALS for some help, and it turns out that they do not know how to solve this issue for now; they asked us to use another PEtALS tool to monitor the process, but we encountered another bug on this new tool. This discussion with the developers was happening while this document was being written. To be able to make tests involving more nodes, we started using a LAN (called "Revoada") inside IME-USP.

The tests proceeded in a smaller scale in "Revoada", since it has only eight computers in LAN, instead of the twenty possible virtual machines from Amazon EC2. But we came up with some interesting results, exposed on the next Section.

3.2 Message Sender

To evaluate the responses of the synthetical compositions, we developed another Ruby script to send messages of sizes s and with a frequency of f messages per second.

Given the fact that the connection with the composition may alter the results, we added a third parameter to define how many times the batch of messages would be sent, so when there is a lot of traffic on the network, any outlier will not be representative on the overall average. During execution, the script shows four numbers for each batch of messages sent; they represent (in order of appearance):

- **user** is the time spent (in seconds) with the execution of actions of the script, for example calling a class method
- **system** is the time spent (in seconds) with system calls, for example creating a *fork*
- **total** is the sum of *user* and *system* times
- **real** is how much "wall time" has passed (in seconds) since the task began until its end

An example of its execution is shown in Figure 6.

To minimize the latency problem, we ran the _"send_messages" script on one of the machines of the LAN.

```
                            user     system      total        real
sending msg 0           0.010000   0.000000   0.010000 (  1.228076)
sending msg 1           0.000000   0.000000   0.000000 (  1.227862)
sending msg 2           0.000000   0.010000   0.010000 (  1.229278)
sending msg 3           0.000000   0.000000   0.000000 (  1.226910)
sending msg 4           0.010000   0.000000   0.010000 (  1.228499)
sending msg 5           0.000000   0.010000   0.010000 (  1.228195)
sending msg 6           0.000000   0.000000   0.000000 (  1.228027)
sending msg 7           0.000000   0.000000   0.000000 (  1.228253)
sending msg 8           0.010000   0.000000   0.010000 (  1.228292)
sending msg 9           0.000000   0.010000   0.010000 (  1.228239)
> total:                0.030000   0.030000   0.060000 ( 12.281631)
> average:              0.003000   0.003000   0.006000 (  1.228163)
> variance:             0.000021   0.000021   0.000042 (  0.000000)
> standard deviation:   0.004583   0.004583   0.009165 (  0.000554)
                            user     system      total        real
```

Figure 6: *Example output from running the send message script with* $S = 1$, $F = 1$, $BatchSize = 10$

4 Produced Code and Results

We developed a software composed of two scripts:

- **generate_composition** receives two parameters:
  1. sets how many children (partners) each node will have;
  2. sets the depth of the tree
- **send_messages** receives 6 parameters:
  1. the host at which the service is;
  2. the port to reach the service;
  3. the service path from the root of the host;
  4. the size, in bytes, of each message that will be sent to the composition;
  5. the frequency (number of messages per second) with which the script will send the message to the composition;
  6. how many times the batch of messages will be sent

This system is licensed under GPL, version 2 or later.

4.1 Results

We managed to execute the composition on "Revoada", a LAN inside IME-USP, consisting of eight machines. With a maximum of eight nodes we were able to generate three designs of topologies, which we called:

- **Balanced Tree**, a complete balanced binary tree with depth of 2 (forcibly, by our infrastructural constraints, it has only 7 nodes);
- **Vertical Tree**, a tree with each node having just one child, and depth varying from 2 to 7 (using a minimum of 3 and a maximum of 8 nodes);
- **Horizontal Tree**, one root node with 2 to 7 Leaf nodes as children, depth fixed on 1 (using a minimum of 2 and a maximum of 8 nodes)

Graphical examples of these topologies are shown in Figure 7

To be "fair" in the comparison of the tests, we established that each of the trees would have 7 nodes. This means a "vertical tree" with depth of 6, the "balanced tree" described above, and a "horizontal tree" with 6 children.

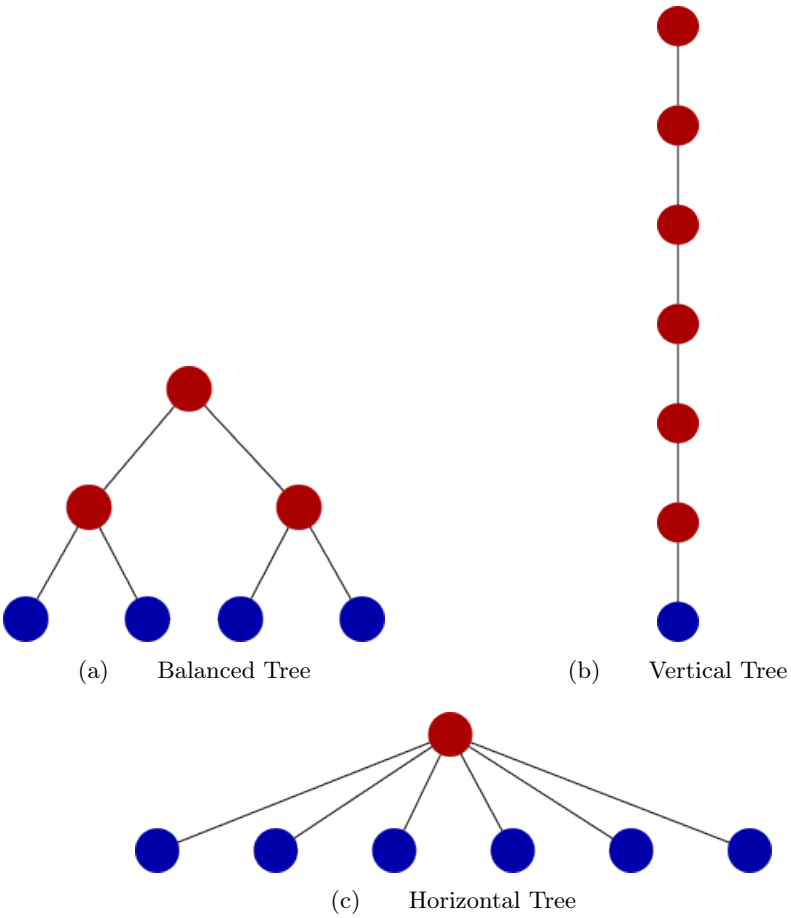


Figure 7: *Example trees for each topology available in "Revoada"*

For the scalability analysis we chose the method "2^k factorial" [1]. This method uses two levels for each factor of the experiment to determine the influence of them on the response time of the composition. Our controllable factors are the *size* of each message and the *frequency* to send the messages to the system, so we chose $1kB$ (one kilobyte) and $1MB$ (one megabyte) for the size and 10 (ten) and 100 (one hundred) messages per second for the frequency.

With these values, we ran the script ("send_messages") four times, one for each combination of the chosen values, for each of the three topologies. The results, simplified to show only the real time execution and its statistics, are shown next.

[1] This is a very well known method described in several educational texts, such as [Law and Kelton (1991)]

Balanced Tree

Size 1 kB, Frequency 10	Real Time	Size 1 kB, Frequency 100	Real Time
> total:	90.11 s	> total:	900.68 s
> average:	9.01 s	> average:	90.06 s
> variance:	0.11 s	> variance:	0.78 s
> standard deviation:	0.34 s	> standard deviation:	0.88 s

Size 1 MB, Frequency 10	Real Time	Size 1 MB, Frequency 100	Real Time
> total:	1869.89 s	> total:	18767.14 s
> average:	186.98 s	> average:	1876.71 s
> variance:	1.49 s	> variance:	4.55 s
> standard deviation:	1.22 s	> standard deviation:	2.13 s

Vertical Tree

Size 1 kB, Frequency 10	Real Time	Size 1 kB, Frequency 100	Real Time
> total:	133.23 s	> total:	1139.50 s
> average:	13.32 s	> average:	113.95 s
> variance:	7.95 s	> variance:	28.81 s
> standard deviation:	2.81 s	> standard deviation:	5.36 s

Size 1 MB, Frequency 10	Real Time	Size 1 MB, Frequency 100	Real Time
> total:	2763.81 s	> total:	20225.51 s
> average:	276.38 s	> average:	2022.55 s
> variance:	117.00 s	> variance:	17599.66 s
> standard deviation:	10.81 s	> standard deviation:	132.66 s

Horizontal Tree

Size 1 kB, Frequency 10	Real Time	Size 1 kB, Frequency 100	Real Time
> total:	180.03 s	> total:	1921.15 s
> average:	18.00 s	> average:	192.11 s
> variance:	0.20 s	> variance:	17.03 s
> standard deviation:	0.45 s	> standard deviation:	4.12 s

Size 1 MB, Frequency 10	Real Time	Size 1 MB, Frequency 100	Real Time
> total:	2500.01 s	> total:	21990.79 s
> average:	250.00 s	> average:	2199.07 s
> variance:	11.83 s	> variance:	86787.51 s
> standard deviation:	3.43 s	> standard deviation:	294.59 s

With the averages we are finally able to calculate the influence effect of each factor for each topology. We have done this with a predefined algebraic process. First let I be the mean case, A be the effect of frequency of messages and B be the effect of message size. Then let

$$x_A = \begin{cases} -1 & \text{if 10 messages per second} \\ 1 & \text{if 100 messages per second} \end{cases}$$

$$x_B = \begin{cases} -1 & \text{if 1kB message size} \\ 1 & \text{if 1MB message size} \end{cases}$$

Then we have the model: $y = q_I + q_A x_A + q_B x_B + q_A B x_A x_B$, where y is, in our case, the real time acquired from the experiment. The interpretation will be:

q_I is the Mean Real Time

q_A is the effect of Frequency of messages

q_B is the effect of message Size

$q_A B$ is the interaction between Frequency of messages and message Size

To solve the linear system generated by the substitution of y, x_A, and x_B, we used the "Sign Table Method" [Jain (2006)]. The tables used can be found next:

Table 1: *Sign Table to determine the influence effect of each factor in "Balanced Tree"*

I	A	B	AB	Real Time
1	-1	-1	1	9.01 s
1	1	-1	-1	90.06 s
1	-1	1	-1	186.98 s
1	1	1	1	1876.71 s
2162.78	1770.78	1964.62	1608.66	Total
540.69	442.69	491.15	402.16	Total/4

Table 2: *Sign Table to determine the influence effect of each factor in "Vertical Tree"*

I	A	B	AB	Real Time
1	-1	-1	1	13.32 s
1	1	-1	-1	28.81 s
1	-1	1	-1	276.38 s
1	1	1	1	2022.55 s
2341.06	1761.65	2256.79	1730.68	Total
585.26	440.41	564.19	432.67	Total/4

Table 3: *Sign Table to determine the influence effect of each factor in "Horizontal Tree"*

I	A	B	AB	Real Time
1	-1	-1	1	18.00 s
1	1	-1	-1	192.11 s
1	-1	1	-1	250.00 s
1	1	1	1	2199.07 s
2659.19	2123.19	2238.96	1774.96	Total
664.79	530.79	559.74	443.74	Total/4

Now that we have the effect of each variable, we need to discover what is the *variation explained* of each factor, so that we can realize the importance of each factor. There is another algebraic way to calculate this variation: $Total = 2^2 * q_A^2 + 2^2 * q_B^2 + 2^2 * q_A B^2$, where each element of the summation is the variation of the related factor. Finally we apply this calculations for each topology:

Balanced Tree	Total variation = 2395811.71 Variation of Frequency = 783920.85(32.72%) Variation of Size = 964937.26(40.28%) Variation of Intersection = 646953.59(27.00%)
Vertical Tree	Total variation = 3167847.20 Variation of Frequency = 1126984.03(35.57%) Variation of Size = 1253237.38(39.57%) Variation of Intersection = 787625.78(24.86%)
Horizontal Tree	Total variation = 2797959.71 Variation of Frequency = 775860.63(27.73%) Variation of Size = 1273284.26(45.51%) Variation of Intersection = 748814.81(26.76%)

5 Conclusions

At the end of the development phase we concluded that, with the test suite the TDD would have created, some of the refactorings would have been easier to do and we could have avoided a lot of small defects, such as forgetting to include a line in the server property file after one big refactor. But creating the orchestrations from scretch was probably the most difficult part of the development of this work. If we had a test framework and development methodology for the orchestrations development, we probably would not have spent the time we did trying to deploy simple orchestrations. Such a framework does not exists yet, this is another goal of the CHOReOS project.

The biggest contribution of this work, was to create the software that synthesize orchestrations. The Ruby script has some interesting *Domaing Specific Languages* (DSL), as shown in snippets 1 and 2. The later hides all the details of creating, managing, and joining the Process or Threads (according to the Ruby's version, 1.8 or 1.9). With these kind of helper methods, its easier to write the orchestration files (*BPEL*s, *WSDL*s, and *JBI* descriptors) that will be the foundations of the generated composition.

```
@graph.each_node do |node|
  response << "#{node.name.downcase}='http://localhost/#{node.type}Node#{node.id}'"
end
```

Listing 1: *Example of* DSL *(**each_node**) from Ruby script "generate_orchestration"*

```
@graph.each_node_parallel do |node|
  set_up_server_for node
  start_petals_on node
  generate_orchestration_of node
end
```

Listing 2: *Example of* DSL *(**each_node_parallel**) from Ruby script "generate_orchestration"*

After we finished the statistical analysis of the experiments, we concluded this:

1. All three topologies scales linearly according to the frequency it is submitted. For example, fixing the messages size to 1MB, when we submitted the balanced tree to a frequency of 10 messages per second, it took an average of 186.98 seconds for the response; with a frequency of 100 messages per second the system took about 10 times more to respond (1876.71 seconds)

2. All three topologies are more influenced by the size of messages than by the frequency. The case with the biggest difference was the "Horizontal Tree", where the size influences 45.51% of the total *variation*, while the frequency influence is only 27.73%.

3. In theory, if there was any significant difference between the average response times, it would be that the "Vertical Tree" should be slightly fast than the "Horizontal Tree", with "Balanced Tree" as the mean case. This difference would occur because of possibles overheads on the interpretation of PEtALS services orchestration and also because of the JVM's JIT Compiler, that would optimize the "Nodes", but not the "Leaves" (because the "Node" service is still active while waiting for its children reply, while the "Leaf" process has to be awoken from Memory every time it receives a message). The tests confirmed the suspects that the "Vertical Tree" was slightly faster than the "Horizontal Tree", but we did not expect that the "Balanced Tree" had the fastest execution of all. This specific case will be deeply studied later.

The future of this work will be to create new test cases, with more nodes, different topologies, and the invocation of children in parallel. We will also study more the "Balanced Tree" case, creating more experiments with more values for each parameter of "send_messages" script. Also, with the "framework" and the future tests, the understanding of the "Internet of the Future" scalability will be much easier to do.

References

AMAZON(2010a) AMAZON. Auto Scaling, 2010a. URL http://aws.amazon.com/autoscaling/. Last access on November 25, 2010. Referenced at page. 9

AMAZON(2010b) AMAZON. Amazon Elastic Compute Cloud (EC2), 2010b. URL http://aws.amazon.com/ec2. Last access on November 25, 2010. Referenced at page. 8

BEA Systems *et al.***(2002)** BEA Systems, Intalio, SAP, and Sun Microsystems. Web Service Choreography Interface, 2002. URL http://www.w3.org/TR/wsci/. Last access on November 25, 2010. Referenced at page. 8

Box *et al.***(2000)** Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol, 2000. URL http://www.w3.org/TR/2000/NOTE-SOAP-20000508/. Last access on November 25, 2010. Referenced at page. 5

Christensen *et al.***(2001)** Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Service Description Language, 2001. URL http://www.w3.org/TR/wsdl. Last access on November 25, 2010. Referenced at page. 4

Fielding *et al.***(1999)** R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol, 1999. URL http://www.ietf.org/rfc/rfc2616.txt. Last access on November 25, 2010. Referenced at page. 4

Jain(2006) Raj Jain. 2k Factorial Designs, 2006. URL http://www.cs.wustl.edu/~jain/cse567-06/k_172kd.htm. Last access on November 25, 2010. Referenced at page. 16

Law and Kelton(1991) Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis*. McGraw-Hill College, 2nd edição. ISBN 0070366985. Referenced at page. 14

OASIS(2010) OASIS. OASIS (Organization for the Advancement of Structured Information Standards, 2010. URL http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel. Last access on November 25, 2010. Referenced at page. 6

Peltz(2003) C. Peltz. Web Services Orchestration and Choreography. *Computer*. ISSN 0018-9162. Referenced at page. 6, 8, 9

PetalsLink(2010) PetalsLink. PEtALS ESB, the Open Source ESB, 2010. URL http://petals.ow2.org/. Last access on November 25, 2010. Referenced at page. 7

W3C XML Working Group(2008) W3C XML Working Group. Extensible Markup Language, 2008. URL http://www.w3.org/TR/xml/. Last access on November 25, 2010. Referenced at page. 4

W3Schools(2008) W3Schools. Web Service Tutorial, 2008. URL http://www.w3schools.com/webservices/. Last access on November 25, 2010. Referenced at page. 4

Zhang *et al.***(2010)** Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*. Referenced at page. 8