Automated Scalability Testing Framework for Web Service Choreographies

Pedro Morhy Borges Leal

Advisor: Fabio Kon

December 2, 2011

Contents

Student: Pedro Morhy Borges Leal
Advisor: Prof. Fabio Kon

1 Introduction

Service-Oriented Architecture (SOA) consists in an architectural model that uses web services as the building blocks of distributed applications (Hewitt, 2009). The use of web services to build distributed applications gives some benefits (Bean, 2009) such as:

Interoperability: A SOA application can be developed using multiple technologies.

Reusability: SOA applications are divided into services so they can be reusable.

Composability of services is one of the SOA principles and its goal is to deliver a service composed of the collaboration of a set of web services. Two main approaches have been proposed: Web Service Orchestration and Web Service Choreography. Orchestration is a centralized approach, its implementation is simple and straightforward. However, this centralized nature, in a large scale environment, can lead to scalability and fault-tolerance problems. Choreography is a kind of service composition proposed to solve the problems of web service orquestrations. It is a decentralized scalable approach with no single point of failure.

Nevertheless, few choreography standards have been proposed and, up to now, none of them have experienced wide adoption. Consequently, the choreographies implemented using *ad hoc* development processes. For this reason, choreography development, including testing cannot be performed properly. In this case, choreography scalability cannot be assessed, hindering the scalability that is actually achieved.

To take advantage of the choreography scalability, we propose a scalability testing framework to support the scalability verification of the implemented choreography. The choreography developer will be able to analyze which functionalities are not scaling well, and improve its architecture before enacting it in the production environment.

Our framework is part of the Rehearsal Framework, a Test-Driven Development Framework (TDD) for Web Service Choreographies (Besson *et al.*, 2011). It has a set of components that assist the developer in applying the TDD methodology for choreographies. It aims at facilitating choreography development and improving its adoption. TDD is a design technique that drives the development process through tests (Beck, 2002).

Rehearsal is part of the Baile Project[1] and the CHOReOS Project[2]. It has been developed along with the master student Felipe Besson at the Institute of Mathematics and Statistics from the University of São Paulo (IME-USP), who also supported the implementation of this course completion assignment.

1.1 Basic Concepts

In this sub-section, we will describe some basic concepts for this course completion assignment.

Web Service Web service is an interoperable way of providing a service through the Web. It is based on a set of standards defined by the World Wide Web Consortion. Among all, there are four standards that are the core of web services (Bean, 2009):

Extensible Markup Language (XML) is a machine-readable language used in web services to transport data.

[1] http://ccsl.ime.usp.br/baile/
[2] http://choreos.eu/

Web Service Description Language (WSDL) is the overall service interface language. It describes the information about the location of the service, its operations, and the expected inputs and outputs. The consumers requests it to collect information about the web service.

XML Schemas Definition Language (XSD) is used to validate XML files. It describes the structure that an XML file should have. In web services, it is used to define complex data types and to validate the XML messages exchanged between the services. It is referenced or encapsulated in a WSDL. The consumers request it to structure the data that it will be sent to the web service.

Simple Object Access Protocol (SOAP) is a protocol of message exchange. It has a set of rules for the structure of the exchanged messages. SOAP defines how the message with the data should be sent. The XML data that a consumer sends or receives is encapsulated by some informations such as delivery information and address.

There are several tools that assist the generation of web services by abstracting the standards defined above. In this project, the development of the distributed applications implemented to validate the framework used JAX-WS[3] to support the creation of the web services. It provides a Java API for generating web services.

Web Service Choreography Web Service Choreography is a service composition approach (Peltz, 2003). It has been proposed to be a decentralized implementation of the collaboration of services. It describes a flow of messages between a set of services in a global model, without a central point of control. This decentralized approach gives more scalability and fault-tolerance because there is not a central point of coordination. A choreography is composed of participants that describe their role in it. Each participant can have one or more services that implements it.

However, there is not a choreography standard widely adopted yet. In this project, the development of a web service choreography was made using BPMN2 to describe the flow of messages, and BPEL to describe the interaction of each participant.

BPMN2 (Business Process Model and Notation version 2) specifies a business process in a business process model with a graphical representation. It is used to describe a business process and the communication among business processes. We used BPMN in this project to document the choreography developed.

BPEL is an orchestration language, in which a central node coordinates all other services. A choreography can be seen as a set of orchestrations collaborating with each other. BPEL is an XML language that describes the interaction of a process with other web services, called partner links, based on their WSDLs. The developer can specify the interaction with the web services and the manipulation of the data exchanged in the interactions.

[3] http://jax-ws.java.net/

2 Scalability Testing

Currently on the Internet, we have high a variation of demand of the applications that run on it. Many of these applications cannot support this high variation properly because of bad implementations or bad architectures. These problems are related to the application scalability. Scalability testing aims at analyzing it and verifying which part of the systems are not scaling properly. First, we need to define what the term scalability means. Then, we must know how we analyze the application scalability to test it.

2.1 Definition of Scalability

A choreography, in a large scale environment, must be able to cope with different scenario sizes. Suppose that a specific choreography usually receives N requests per second but, in some specific times, receives $10N$ requests per second. It must tolerate this increase of access quickly.

To support this, we may increase the system capacity during this period of high demand. We want to have the same performance that we had previously. Therefore, we must know how many resources we should increase to achieve this goal. Each choreography will have its particular solution. This will depend on how the choreography is implemented. Choreographies with a bad architecture will need to increase its resources heavily to achieve the same performance.

For instance, in the example above, we could increase the resources in the same scale that the number of access increased to gain the same performance. However, this will not work properly if the choreography is not well implemented, i.e., the choreography may have a bottleneck that will not disappear if we add more resources. Thus, we also must pay attention to the implementation of the choreography. We gain scalability with the collaboration of hardware and software.

An application is scalable if it achieves the same performance when increasing the architecture capacity with the same proportion that the problem size increases (Quinn, 1994). To Law (1998), an application is scalable if the improvements in the execution capability express in direct proportion improvements in architectural capability. However, these definitions just say if an application is scalable or not. Consequently, we cannot compare the scalability of two applications by saying that one is more scalable than the other. To conclude how scalable an application is, we need to quantify it. Law quantifies simulation scalability in a mathematical model, based on two capabilities: execution and architectural.

2.2 Quantifying Scalability

The function that quantify the execution capability is based on two functions: the complexity size of the problem function, and the performance metric function.

The complexity size of the problem is based on a set of values that describes its state, which are called measures of interest. For instance, in a web service stress simulation, the number of requests per second is a measure of interest. If the number of requests per second increases, the complexity size of the problem increases too. Law defines the complexity size with the function $S(n_1..n_s)$, where n_1 to n_s is a set of s measures of interest. It is assumed that if one of these variables increase, the simulation size function will increase or at least stay the same, but will never decrease. There are different ways to define the complexity of a problem. It depends on the characteristics we want to analyze.

When we execute the application, a set of performance characteristics can be analyzed. Thus, we also have a function that defines the importance of the metric performances, $M(m_1..m_q)$, where m_1 to m_q is a set of q performance metrics. If a performance metric increases, it is assumed that the performance decreases. We can reverse the metrics that do not behave this way. With this definition, we conclude that a simulation capability is the relation of the functions S and M, defined as $C(S, M) = \frac{S}{M}$.

As said previously, to measure scalability we need also to quantify the architectural capability. The inputs that we need to define are the architecture characteristics, such as the CPU clock speed.

The $P(h_1..h_p)$ function describes it, where h_1 to h_p are the variables that represent the measures of the architecture.

With the definition of these two functions, we choose an initial architectural capability P_1, initial measures of interest S_1, and the metric performances M_1 we want to analyze. After the execution, a constant $k = \frac{C_1(S_1,M_1)}{P_1}$ is calculated. Let C_i and P_i be capabilities i times greater than C_1 and P_1, respectively. Therefore, Law defines scalability as the largest real-value j such that $\frac{C_i(S_i,M_i)}{P_i} \geq k$ for every i in the interval $[1, j]$. With this definition, we can are verify the interval where the ratio between the execution capability and the architectural capability stays at least equal as it was initially. If $j = \infty$, we say that the system is fully scalable.

This definition, however, has some limitations as pointed out by Law. One of them is the difficulty to define an architectural capability function that expresses the exact importance among each hardware characteristic. For example, a system with 512MB of RAM and a CPU of 100GHz may or may not have the same capability of one with 256MB and a CPU of 200GHz. Another one is the high dependency on the definition of the architectural and simulation capabilities in the scalability function. This means that, the same system can have different ranges of scalability depending on how we define these functions. The point here is that it is very difficult to quantify the P and C variables. Therefore, if we cannot quantify them properly, we will not be able to quantify the scalability of the system.

These restrictions show us that scalability is something relative, a system may have different ranges of scalability depending on the importance that we give to its elements. However, despite these limitations, we can adapt it to something more usable. We can use the scalability function to compare two systems with the same architectural and simulation capabilities. We can also limit the number of the capability functions input to one. Let s be the input of the complexity size of the problem, and m be the input of the performance metric for simulation capability function C_i and, h for the architectural capability function. Thus, we can evaluate if a system is more scalable then another based on these three variables. Law presents the final definition of scalability as the largest real-value j, such that $\frac{C_i(s,m)}{P_i(h)} \geq k$ for every i in the real-value interval $[1.0, j]$.

With this description, the author narrows the scalability comparison of two systems, regarding to a specific aspect of the simulation and architecture. For example, we chose s as the size of data sent per request to a web service, m as the percentage of used memory, and h as the total memory of the web service hardware. Then, we can compare if a system has more scalability than the other in terms of space utilization.

2.3 Quantifying Scalability in Practice

The definition above, made us conclude that the function $\frac{C_i(S_i,M_i)}{P_i} = \frac{S_i}{M_i \times P_i}$ should be constant after each execution. Therefore, $\frac{S_i}{M_i}$ should vary in the same proportion as P_i. Since the execution capability is based on two functions, S as the complexity size of the problem and M as the performance metric, we can increase it in different ways:

1. Increasing S with the same proportion as the architecture capability. In this case, we expect M to be constant.

2. Fixing S. Then, we expect that M will decrease in the same proportion as the architecture capability.

3. Increasing S in a different proportion of the architecture capability. Then, we expect that M will decrease in a proportion that will make the $\frac{S}{M}$ function increase in the same proportion as the architecture capability.

If we use the first option, the performance metric of an application that has a good scalability should stay constant and one that has a bad scalability will increase. Thus, an application with good scalability will have a performance metric graph with a horizontal line (Figure 1) and an application with bad scalability will have an increasing line (Figure 2).

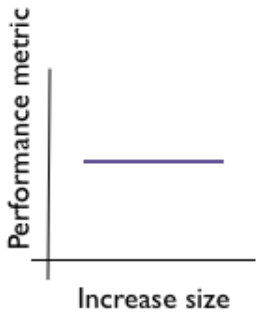

Figure 1: *Performance metric graph with a **good scalability***

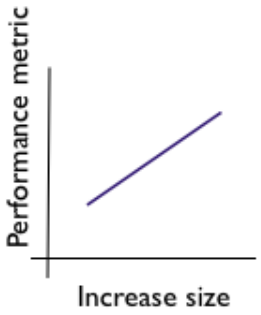

Figure 2: *Performance metric graph with a **bad scalability***

In this project, we will use the first option because we have control over the variables that define the complexity of the problem. Also, it is easier to visualize, in a performance metric graph, the difference between an application with good scalability and bad scalability.

2.4 Types of Scalability

With the definition presented in the Section 2.2, we conclude that there are different ways to examine the scalability of a system. Bondi (2000) presents four types of scalability that we can observe:

Load scalability. A system has more load scalability if it behaves properly with light, moderate, or heavy loads. In this case, the word *properly* is related to functional and non-functional aspects, such as resource contention, excess of delay, and ineffective memory consumption. Thus, to measure the load scalability of a system using the scalability function, one would use, for instance, number of request per second as the input for the simulation capability function, the number of processors as the input for the architectural capability, and fixing the response time average as a performance metric.

Space scalability. A system has a bad space scalability if its memory requirements grow largely when the number of supported items increases. For example, measuring the scalability based on the data size sent per request as an input for the simulation capability function, the RAM of the system hardware, and fixing the percentage of RAM used as a performance metric.

Space-time scalability. A system has more space-time scalability if it works properly, without large delays, when the number of its objects increases. If a search system takes more time to find an information because of the increasing of data storage, even if its hardware resources have grown too, we conclude that it does not have space-time scalability.

Structural scalability. A system is structurally scalable if it does not prevent the increasing of objects that it encompasses. For example, if a peer-to-peer system has a maximum number of nodes that it can support and another has not, the latter is more structural scalable than the former.

These types of scalability give us a direction of what we need to observe when comparing the scalability of the systems. It is important to note that they are, sometimes, dependent from each other. A system that needs space-time scalability may also need space scalability because a huge growth of memory might cause a delay to find information.

2.5 Related Work

Chen and Sun (2006) present the Scalability Testing and Analysis System (STAS), a tool that supports the users in running scalability analysis of algorithms and systems. STAS analyzes scalability

using the metric *isospeed-efficiency scalability*, also known as *isospeed-e*.

The architecture capability used in this metric is the sum of the supported speeds of all system nodes. Let h_i be the supported speed of a specific node i, then the architecture capability of the system is $P(n) = \sum_{i=1}^{n} h_i$, where n is the number of nodes.

A function that characterizes the complexity of the problem, defined by the user, combined with the execution time, which is the performance metric, results in the execution capability function $C = \frac{S}{T}$, where S is the value that represents the complexity of the problem and T is the execution time metric. Therefore the scalability function is defined as $\frac{C}{P}$.

STAS is composed of four components as shown in Figure 3: system characterization component, algorithm pre-analysis component, scalability tester component, and scalability analyzer component.

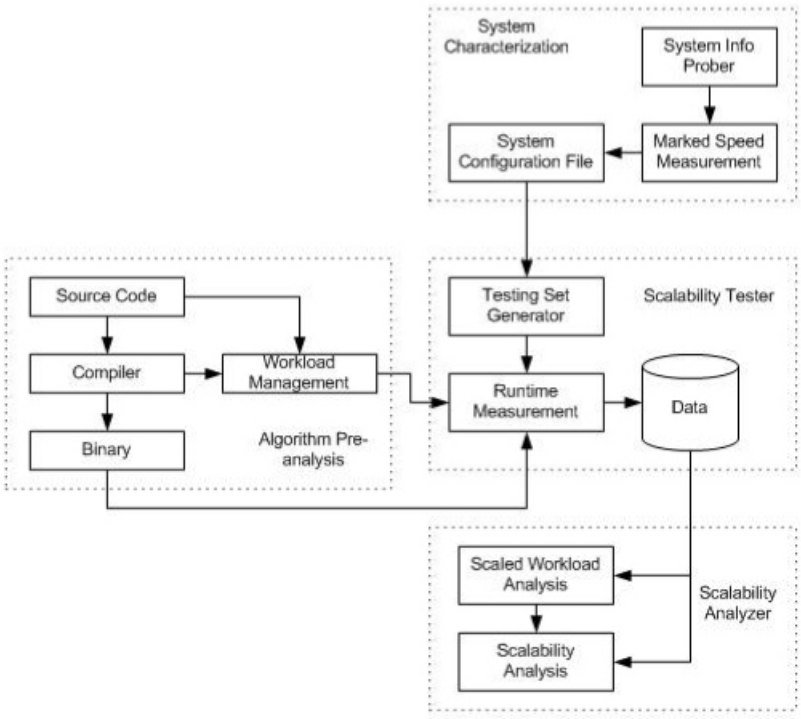

Figure 3: *STAS Architecture (Chen and Sun, 2006)*

The system characterization component is responsible for calculating the supported speed of each available node by measuring its speed in FLOPS. The algorithm pre-analysis component is responsible for calculating the problem complexity defined by the user.

The scalability tester component executes the application and calculates the execution time. It calculates C_1 by applying the function described above with the obtained values. After that, it executes the application doubling the number of nodes and the parameter of the complexity problem function. C_2 is obtained with the execution information. This process is done until it reaches the maximum number of nodes available. The results are compared by the scalability analyzer component that concludes how scalable the application is.

STAS supports only parallel applications, such as MPI, PVM, and HPF. Thus, it is not applicable for SOA applications, where the implementation of the services that integrate the system are heterogeneous. It restricts the choice of the scalability function variables. The tester, for instance, cannot test the scalaiblity based on the number of requests per second, since the complexity of the problem is only defined based on the complexity of the application algorithm. Our Scalability Testing Framework, described in the next section, aims at providing the freedom to choose the scalability test type desired.

de Almeida *et al.* (2008) presents a methodology and a framework for testing peer-to-peer (P2P) applications. The authors combine functional and non-functional tests because P2P applications are

volatile and have variations in its scale, which might affect the functionalities of the system. The framework is able to manipulate all nodes of the application and test it in different scenarios. It applies scalability testing by changing the scales of the application and validating their functionalities. However, each execution scenario must be specified, which can make the test specification cumbersome.

By using Java Annotations, the developer specifies the behavior and the tests of the nodes. These annotations give to the developer the freedom to manipulate and test the application in different scenarios, and scales.

3 A Framework for Scalability Testing

Our Scalability Testing Framework aims at assisting the scalability analyzer, the person who tests the scalability, to verify the scalability of an application. Based on the research presented in the last section, we conclude that the tester must apply the following steps to assess a system scalability.

1. Choose the variables of the problem complexity, the performance, and the architecture.
2. Define the functions of the complexity size of the problem, performance metric, and the architecture capability.
3. Choose initial values for these variables.
4. Execute the application with these initial values for obtaining the initial value of the performance metric.
5. Execute multiple times with the same process and collect the performance metric for each execution.
6. Analyze the performance metric.

Since we do not want to restrict which type of scalability the tester will analyze, we choose to develop a framework with which steps 1 to 3 could be made without limitations. To support the scalability analyzer, the Scalability Testing Framework automates steps 4 and 5, which are a repetitive process. It also assists the analysis of the obtained performance metric (Step 6).

The framework is developed in Java and it may be used with it or with other languages that run on the JVM (Java Virtual Machine) and can be used with Java Annotations, such as Scala.

The scalability analyzer needs to execute the same process multiple times by changing only the variables that define the complexity size of the problem and the architecture capability. To support that, the framework provides two Java Annotations *@ScalabilityTest* and *@Scale* that gives the ability to the scalability analyzer to execute steps 4 and 5. The tester must only specify a method that describes the execution and the variables that need to change for each execution.

```
1 @ScalabilityTest(scalabilityFunction=<ScalabilityFunction.class>, steps=<Integer>)
2 public List<Number> methodName(@Scale Number parameterScale, Object parameter) {
3   //execution process
4   return performanceMetrics;
5 }
```

Listing 1: *Scalability test method template*

A template of the method that is executed by the framework is shown in Listing 1. The annotation *@ScalabilityTest* has two parameters: *scalabilityFunction* and *steps*. The *steps* parameter defines how many times the method will be executed, we call each execution as a *step*.

With the *scalabilityFunction* parameter, the scalability analyzer specifies how the method parameters with the annotation *@Scale* will increase for each step. The parameter specified is a class that extends the ScalabilityFunction class. The classes available are *LinearIncrease*, *ExponentialIncrease*, and *QuadraticIncrease*, which are described with more details below.

The framework collects the performance metric of each step as a list of values. After each execution, it calculates the mean and the standard deviation of these lists to plot them in a graph.

3.1 Example

Suppose that we developed a web service which is located in a cloud environment and we want to verify its *load scalability*. Therefore, we choose the number of requests per second to be the complexity size of the problem function, the number of nodes which the web service is running upon as an architecture capability, and the average response time as the performance metric. Code 2 shows an example of scalability test in this case.

```
1 @ScalabilityTest(scalabilityFunction=LinearIncrease.class, steps=5)
2 public List<Long> scalabilityTest(@Scale int requests, @Scale int numberNodes) {
3   List<Long> resposeTimes = new List<Long>();
4   instatiateNodes(numberNodes); // Change the number of nodes
5   responseTimes = makeRequestsPerSecond(rps);//Returns a list of response times
6   return responseTimes;
7 }
```

Listing 2: *Scalability test example*

Line 1 specifies that this method is a scalability test that must be executed 5 times and that the parameters with the *@Scale* annotation (Line 2) should increase linearly.

In Line 4, we call a method that will change the number of nodes where the web service is deployed. If we increase the number of nodes, we improve the architecture capability. Then, in Line 5, we call a function that makes a number of requests per second to the web service and returns the response time of each execution.

The framework will execute the scalability test with the call presented in the Listing 3. The object *wsScalabilityTest* is the one where the test is described, the string *"scalabilityTest"* is the name of the scalability test, and the *1000* and *1* integers are the initial values of the test parameters.

```
1 ScalabilityReport report;
2 report = ScalabilityTesting.run(wsScalabilityTest, "scalabilityTest", 1000, 1);
```

Listing 3: *Scalability test call*

The Sequence Diagram in Figure 4 shows how the example above will be executed by the framework. It will execute 5 times, increasing the test parameters linearly, collecting the results, and calculating the mean and the standard deviation for each execution step. After the execution, the framework returns a *ScalabilityReport* object, which contains the information about the execution.

Listing 4 shows how to use the report to plot a graph using the *ScalabilityReportChart*. Since it can plot more than one scalability test for comparisons, it receives a list of reports. The scalability analyzer can also specify the label for the performance metric axis, which is by default *"Performance metric"*. Figure 5 shows a graph illustration of the example above.

```
1 ScalabilityReportChart chart = new ScalabilityReportChart();
2 List<ScalabilityReport> reports = new ArrayList<ScalabilityReport>();
3 reports.add(report);
4 chart.createChart(reports, "average response time (ms)");
```

Listing 4: *Graph plotting code example*

3.2 Architecture

Our Scalability Testing Framework is composed of three main packages, the first related to the framework hot spots, where the annotations is located, the second used to interpret the annotations and execute the scalability tests, and the third to show the scalability testing report to the scalability analyzer. The class diagram is presented in Figure 6.

3.2.1 Framework Hot Spots: Annotations and Scalability Increase Functions

A hot spot of a framework is a place of the architecture where the developer can add or modified desired functionalities to adapt the application. In this case, for instance, the test specification is a content that the tester will add to execute the framework. Our Scalability Testing Framework has main principal hot spots. The first hot spots the framework provides are the two Java Annotations, *@ScalabilityTest* and *@Scale*.

The *@Scale* annotation is a parameter annotation. It must be located before the specification of the parameter and the parameter type. The parameters with this annotation will be increased by the framework in each execution step.

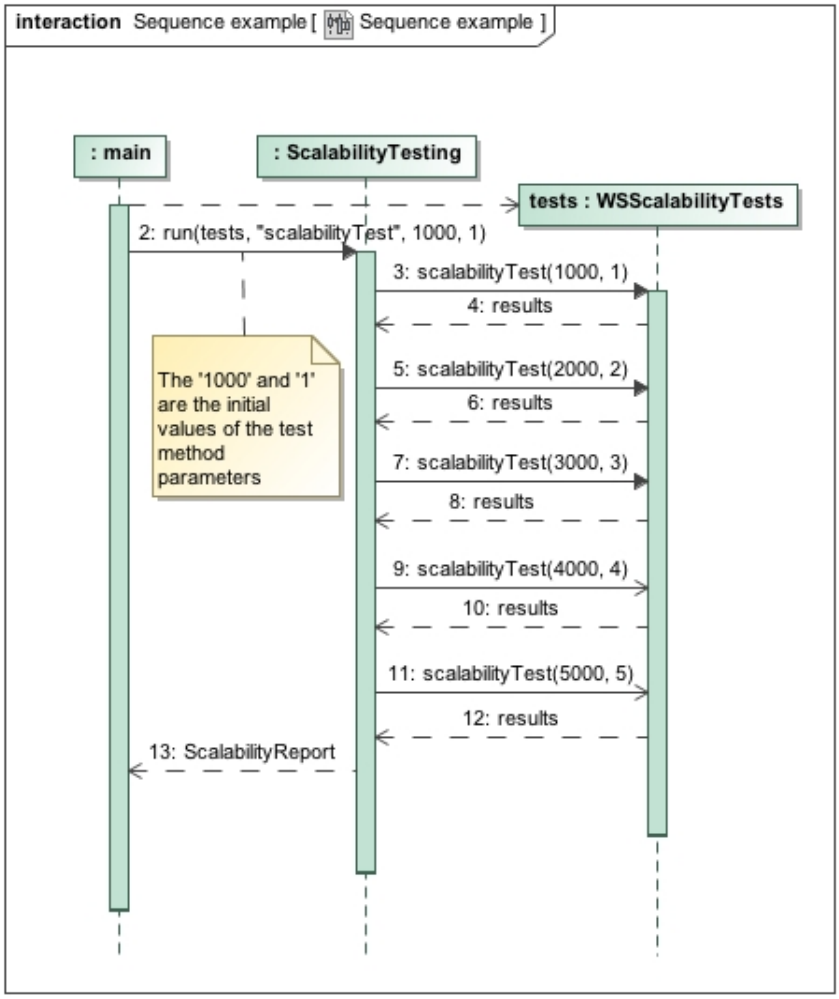

Figure 4: *Execution of the scalability test in example 3*

The *@ScalabilityTest* annotation is a method annotation, which means that it must be used for methods and it is located right above the method that describes a scalability test. It has two parameters: *steps* describes how many iterations the framework will perform to execute the test; *scalabilityFunction* describes how the method parameters with the *@Scale* annotation will increase in each step. It is defined by a class that extends the *ScalabilityFunction* class.

This is the second major hot spot of the framework. There are, already, three ScalabilityFunction classes implemented:

LinearIncrease increases the parameters linearly. Thus, a parameter with initial value x will be $2x$ in the second step and nx in step n.

ExponetialIncrease increases the parameters exponentially. The parameter with initial value x will be $x \times 2$ in the second step and $x \times 2^n$ in step n.

QuadraticIncrease increases the parameters quadratically. The parameter with initial value x will be $x \times 2$ in the second step and $x \times (n-1)^2$ in step n.

However, the scalability analyzer can develop other scalability function for a particular case if needed. The class must extend the *ScalabilityFunction* class which has a method that defines how a parameter will increase. It receives two integers with the last value used and the initial value and must return a new value.

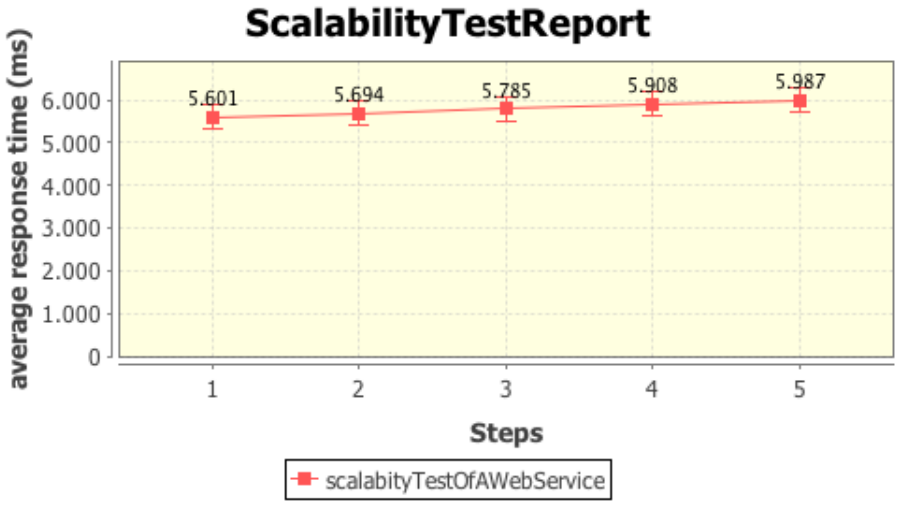

Figure 5: *Graph Illustration*

3.2.2 ScalabilityTest Interpreters

The *ScalabilityTesting* class is called by the scalability analyzer to execute the scalability tests. It executes the scalability test based on the parameters specified in the annotations *@ScalabilityTest* and *@Scale*. It uses the *ScalabilityTestMethod* class, which interprets the annotations of the scalability tests, to collect information about the test methods.

For each step, a *ScalabilityStepReport* is instantiated with the results. With the support of the JDK Commons Math package, it calculates the mean and the stardard deviation of the performance metrics.

After the execution, the *ScalabilityTesting* class instantiates a *ScalabilityReport* that has a list of *ScalaiblityStepReport*. This class can be used by the scalability analyzer to a plot a graph to analyze the results.

3.2.3 Scalability Tests Graph

The scalability analyzer can plot the result of one or more scalability tests with their *ScalabilityReport* using the *ScalabilityReportChart*. This class uses the *XYChart* class, that has the procedure to plot the graph with the support of the JFreeChart package[4] .

3.3 Implementation

The framework was developed in Java since it belongs to the Rehearsal Framework, which is in Java. We developed it using Test-Driven Development (TDD), described in the introduction, with the support of the JUnit Framework[5] to specify and run the tests.

The development used the Eclipse IDE[6] with the support of the EclEmma[7] and Eclipse Metrics[8] plugins. The first one is a Java code coverage tool to verify the coverage of the tests in the code. Our Scalability Testing Framework has 100% of test coverage. The second one calculates some metrics of the code, such as cohesion metrics, which helped us develop a code with quality.

The Scalability Testing Framework uses two third-party libraries:

[4]http://www.jfree.org/jfreechart/
[5]http://www.junit.org/
[6]http://www.eclipse.org/
[7]http://www.eclemma.org/
[8]http://eclipse-metrics.sourceforge.net/

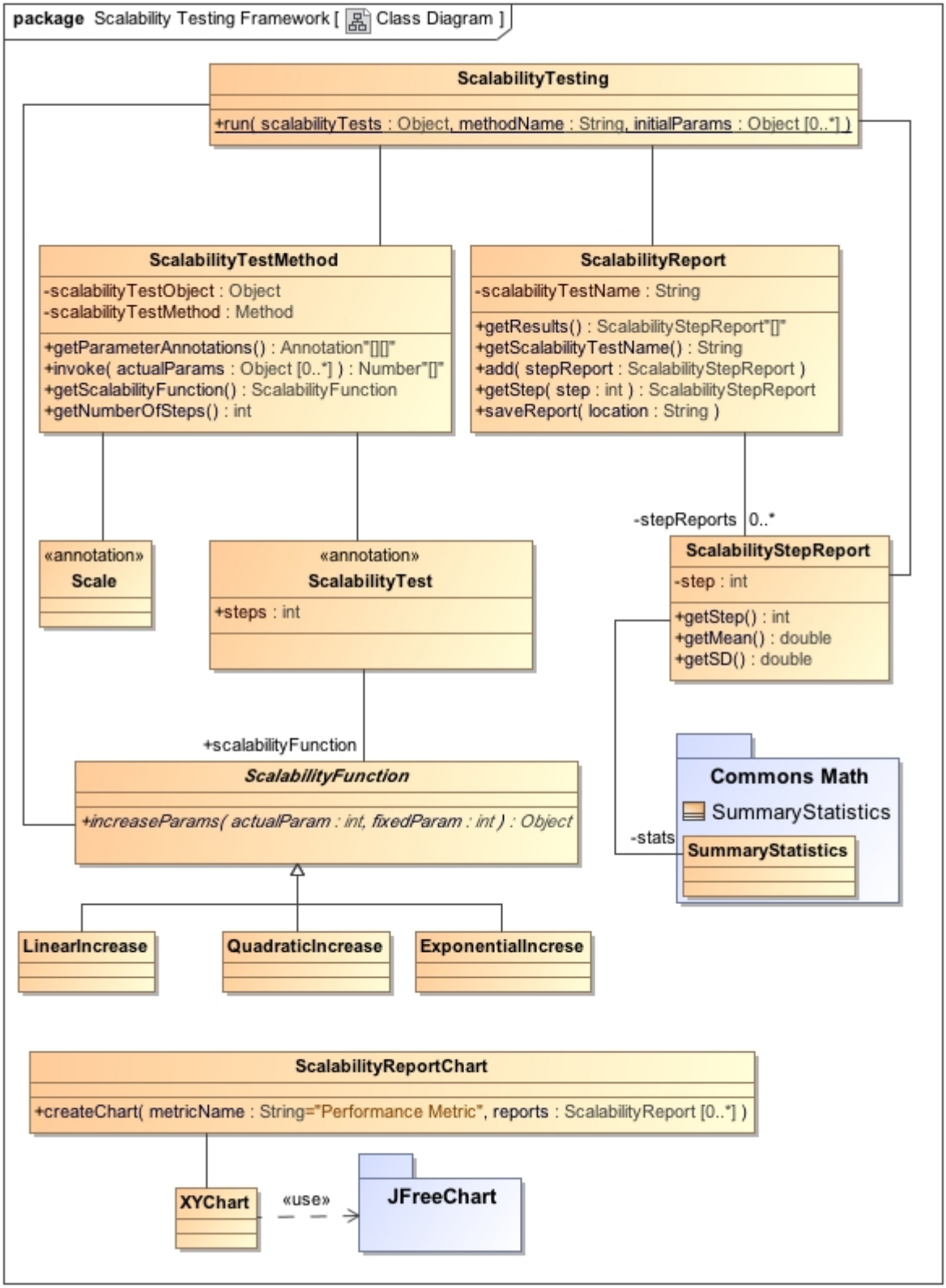

Figure 6: *Scalability Testing Framework Class Diagram*

JFreeChart[9] to plot the performance metric graphs. It depends on the JCommon library. Both are licensed under the terms of GNU Lesser General Public License (LGPL).

Math Commons[10] to calculate statistical values of the performance values returned by the scalability tests. It is licensed under the terms of the Apache License.

The project used the Strategy Design pattern (Gamma *et al.*, 1995) in the implementation of the functionality that increases the parameters in each step.

The *ScalabilityTesting* class, which executes the tests, collects a *ScalabilityFunction* class, the strategy class, and increases the test parameters without the knowledge of which strategy is being used. The concrete strategy classes can be the *LinearIncrease*, *ExponentialIncrease*, *QuadraticIncrease*, and another class developed by the scalability analyzer that extends the *ScalabilityFunciton* class. As we can observe in Figure 6, the structure of the classes described is equivalent to the Strategy Structure.

4 Validation

To validate the first version of the Scalability Testing Framework, we developed two distributed applications. The first one, a distributed matrix multiplication, is a simple application that can have two different scalability sizes with a small change in the implementation. Thus, we used it to compare the scalability of two similar applications and to validate if our framework was capable of showing the difference between them.

The second one is a choreography to be our test bed of the framework for this initial validation, and future research. It offers a service that buys a product list from a set of supermarkets looking for the best price of each item.

4.1 Distributed Matrix Multiplication Validation

The Distributed Matrix Multiplication service is a simple distributed application that multiplies two matrices. There is a central node that receives the request and coordinates the multiplication. It sends requests to other nodes to multiply parts of the matrices.

We used a simple matrix multiplication algorithm that is presented in Listing 5. The first *for* iterates over the lines of matrix A and the second one iterates over the columns of matrix B. Thus, for each line i of matrix A the algorithm multiplies all columns of matrix B to calculate each line i of matrix C. This algorithm involves $\theta(n^3)$ operations, since it has three *for*s of n iterations each one.

```
1  double[][] multiplyMatrix(double[][] A, double[][] B, int n) {
2    double[][] C = new double[n][n];
3
4    for(int i = 0; i<n; i++) {
5      for(int j = 0; j<n; j++) {
6        C[i][j] = 0.0;
7        for(int k = 0; k<n; k++)
8          C[i][j] = C[i][j] + A[i][k]*B[k][j];
9      }
10   }
11   return C;
12 }
```

Listing 5: *Matrix Multiplication Algorithm*

We can parallelize this algorithm by requesting each node of the system to multiply just a range of the matrix B columns. Thus, each node will return a range of the matrix C columns as we can see in Figure 7.

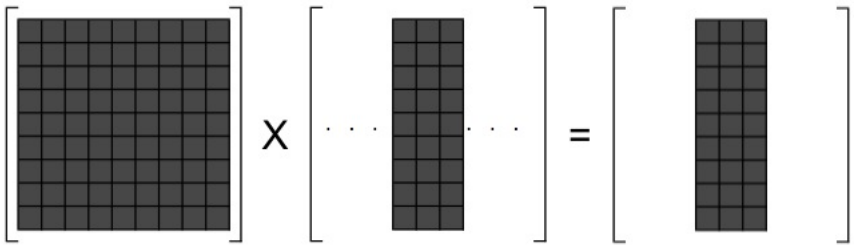

Figure 7: *Distributed Matrix Multiplication of a Node*

The resulting algorithm of each node is presented in Listing 6. Only the columns from *min* to *max* of matrix B are used to calculate the same columns of matrix C. If each node calculates its part in parallel, the algorithm will be executed Q times faster, where Q is the number of nodes. If they are not executed in parallel, the time will be the same of the first algorithm presented.

```
1  double[][] multiplyMatrix(double[][] A, double[][] B, int min, int max, int n) {
2    double[][] C = new double[n][n];
3
4    for(int i = 0; i<n; i++) {
5      for(int j = min; j<max; j++) {
6        C[i][j] = 0.0;
7        for(int k = 0; k<n; k++)
8          C[i][j] = C[i][j] + A[i][k]*B[k][j];
9      }
10   }
11   return C;
12 }
```

Listing 6: *Distributed Matrix Multiplication Algorithm*

We developed a distributed application with Java RMI for the communication between the service that coordinates the multiplication and the services that multiply parts of the matrices. Java RMI (Remote Method Invocation) allows the communication between objects of different JVMs [Oracle (1997)]. One object can manipulate another remote object in the same way it manipulates a local one. The JVM abstracts the lower layers of communication.

We used JAX-WS[11], a Java API for XML Web Services, to provide this service as a web service. This web service receives a request with the number of nodes that it must use, and the size of the two matrices to multiply. It coordinates the multiplication communicating with the Java RMI servers that multiply parts of the matrices. Figure 8 presents the structure of the application. They were deployed in the Amazon EC2[12], the Amazon cloud environment.

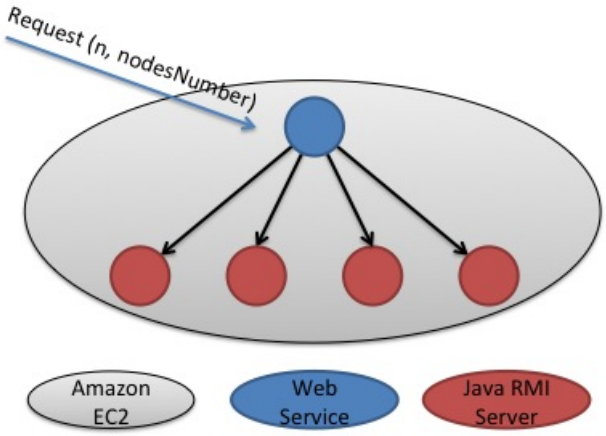

Figure 8: *Distributed Matrix Multiplication Structure*

Two coordinators were implemented with two approaches to validate our framework. In the first one, we simulated that the developer made a mistake by implementing the interation sequentially. It requests the multiplication for each node after the calculation of the last one. In the second implementation, the requests are made in parallel. Thus, all nodes multiply parts of the matrices in parallel.

4.1.1 Scalability Testing

To verify the scalability of the distributed matrix multiplication service, we need to follow the steps described in the beginning of Section 3. First, we needed to decide the variables that define the scalability function (Step 1):

[11] http://jax-ws.java.net/
[12] http://aws.amazon.com/ec2/

Problem complexity variable: Matrix size, n.

Performance variable: Response time, M.

Architecture variable: Number of nodes, Q.

Then, we needed to define how the functions that use these variables will be (Step 2):

Complexity size of the problem: n^3

Performance metric: M

Architecture capability: Q

Thus, the scalability function is $\frac{n^3}{MQ}$ and it is expected to be constant. Since we chose to increase the complexity size of the problem and the architecture capability with the same proportion, the response time of each execution should be constant.

After that, we chose the initial values for them (Step 3). For this experiment, the initial value of n and Q were 400 and 1, respectively.

Steps 4 to 6 were performed using the Scalability Testing Framework. Listing 7 shows a scalability test for the service that implements the algorithm in parallel. The first decision made was the number of steps we wanted to execute and how the variables would increase (Line 1). The number of steps chosen was 8 because of the limitation of nodes that Amazon EC2 provides to our student account. For this reason, too, we chose to increase the variables linearly to execute more steps. If we had chosen to increase exponentially, for instance, the framework would only execute 4 steps, because the number of nodes in each step would be: 1, 2, 4, and 8.

```
1  @ScalabilityTest(steps = 8, scalabilityFunction = LinearIncrease.class)
2  public ArrayList<Long> multOfTwoMatrices(@Scale int numberOfNodes)
3      throws Exception {
4    WSClient wsClient = new WSClient(
5        AMAZON_WS_URL + "multiplyMatrices?wsdl");
6    int n = getNBasedOnNumberOfNodes(numberOfNodes);
7    Item item = generateItem(n, numberOfNodes);
8    ArrayList<Long> results = new ArrayList<Long>();
9    for (int i = 0; i < REPETITIONS; i++) {
10     Long start = System.currentTimeMillis();
11     wsClient.request("multiplyMatrices", item);
12     Long end = System.currentTimeMillis();
13     results.add(end - start);
14   }
15   return results;
16 }
```

Listing 7: *Matrix Multiplication in Parallel*

To increase the complexity size of the problem linearly, we needed to choose n_i based on the number of nodes. Table 1 shows the chosen n values for each step. As we can see, the architecture capability and the complexity size of the problem stay with the same proportion. Therefore, we only pass the number of nodes as an argument because the value of n is based on it, using the function *getNBasedOnNumberOfNodes* (Line 6).

In line 4, we create a client dynamically, using a component of the Rehearsal Framework, to communicate with the web service. The message that is sent is created in line 7.

More than one request is made for each step to avoid values very different from the usual. The number of requests made are defined in the constant *REPETITIONS*, which is 30, in this case. For each step, a list of response times is returned. The framework is responsible for calculating their arithmetic mean and standard deviation.

The framework executes this method 8 times, increasing the parameter *numberOfNodes*. For each execution, it collects the results and calculates the mean and the standard deviation. The result is passed as a ScalabilityReport object. In Listing 8 , we request the execution of two scalability tests,

Q	n	n^3/Q
1	400	64×10^6
2	504	$\sim 64 \times 10^6$
3	577	$\sim 64 \times 10^6$
4	635	$\sim 64 \times 10^6$
5	684	$\sim 64 \times 10^6$
6	727	$\sim 64 \times 10^6$
7	765	$\sim 64 \times 10^6$
8	800	64×10^6

Table 1: *Relation between Q and n*

one for the service that executes the algorithm in parallel (Line 1), and another for the service that executes the algorithm sequentially (Line 2).

```
1 ScalabilityReport report1 = ScalabilityTesting.run(tests, "multOfTwoMatrices", 1);
2 ScalabilityReport report2 = ScalabilityTesting.run(tests, "multOfTwoMatricesSeq", 1);
3
4 ScalabilityReportChart chart = new ScalabilityReportChart();
5 List<ScalabilityReport> reports = new ArrayList<ScalabilityReport>();
6 reports.add(report1);
7 reports.add(report2);
8 chart.createChart(reports, "Execution time (ms)");
```

Listing 8: *Scalability Tests Call and Plot Results in a Graph Code*

The class that plots the scalability test graph is the *ScalabilityReportChart* (Line 4). It receives a list of reports and plot them in a graph. Thus, in line 5 to 7, we create a list of *ScalabilityReport*s to generate the graph (Line 8).

The graph created by the framework is presented in Figure 9. The blue line is the result for the service that implements the algorithm sequentially. It has a bad scalability, comparing to the red line which is the parallel approach. It is similar to the bad scalability graph described in Figure 2 while the red line is similar to the good scalability graph described in Figure 1.

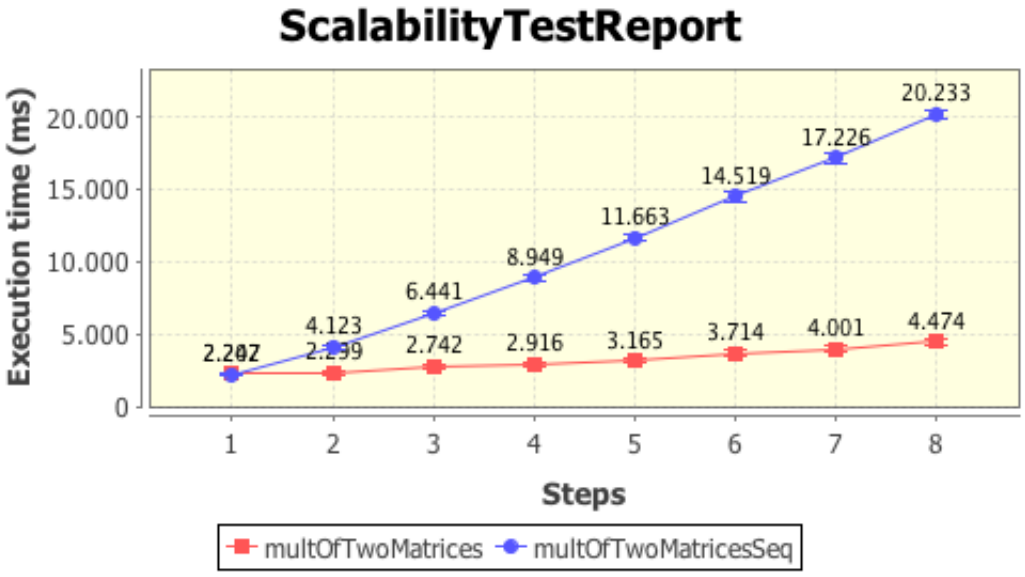

Figure 9: *Scalability Testing Results Graph*

However, if we only see the result of the service with the parallel approach in Figure 10, we can also say that it has a bad scalability because its execution time also increases. Nevertheless, we concluded in Section 2 that scalability is something relative. It is highly dependent on the chosen variables.

It is easier to verify the scalability of an application when we are comparing to another implementation. With the absence of another, the definition of some bounds can assist the scalability tester to analyze the scalability. For instance, in Figure 10, if we define that the service response time must be less then 5 seconds, then we conclude that the service is scalable until 8 nodes with matrices os size less than 800.

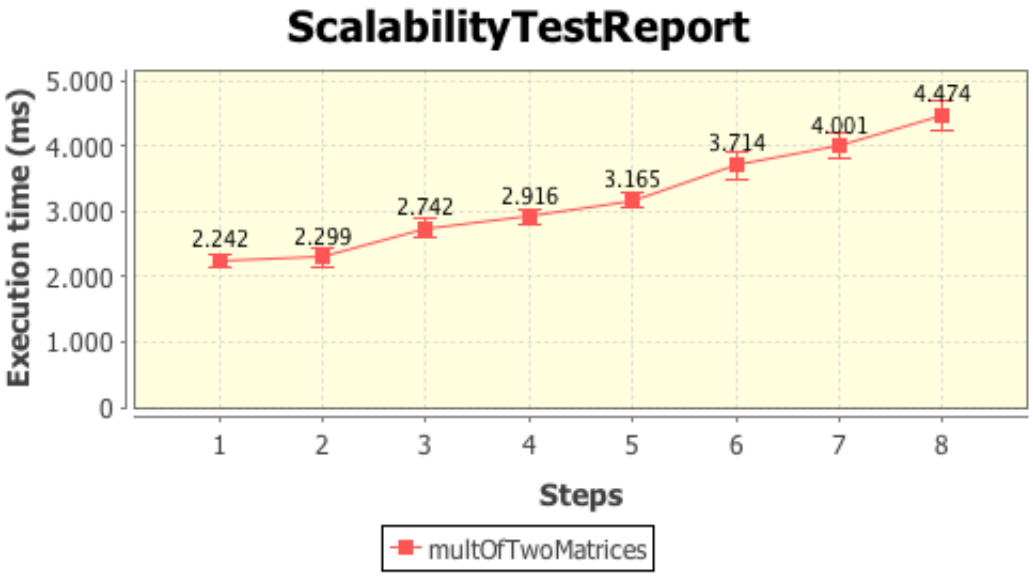

Figure 10: *Scalability Testing Result Graph for the Parallel Approach Service*

4.2 Choreography Validation

We developed a web service choreography to be our test bed for getting information and developing the Rehearsal Framework. Thus, we will use this choreography to collect information about how we are going to verify its scalability. In this section, we will explain how it works.

The choreography implements a service of a supermarket of the future. Based on a list of supermarkets, the system gets the lowest price possible of a desired product list. For each product, the choreography gets the price from the supermarket that offers the best choice. Therefore, we will purchase a list of products with the lowest possible price from a list of supermarkets.

4.2.1 Participants of the choreography

The choreography has five participants. Their roles and API are described bellow:

SMRegistry. The SMRegistry service stores a list of services that implements the supermarket role. Its methods are presented in Table 2.

SupermarketRole. The choreography can have multiple supermarkets, each supermarket must implement the Supermarket Role API, presented in Table 3. It must return the price of a product, and purchase a list of products.

Customer. The Customer service is the interface of the choreography. The user can use other service separately, however the complete functionalities of the choreography are accessible via this service. Its implementation is an orchestration that communicates with the SMRegistry service to get the list of Supermarket services registered. It communicates with all of them to get

Table 2: *SMRegistry API*

Method	Input	Output	Description
addSupermarket	endpoint : String	confirmation : String	Add a new supermarket endpoint to the supermarket list
getList		supermarkets: List<String>	Return a list of supermarket endpoints

Table 3: *Supermarket Role API*

Method	Input	Output	Description
searchForProduct	name: String	name: String, price: Double	Get the product price of this supermarket
registerSupermarket	endpoint: String	confirmation: String	Register the Supermarket endpoint in the RegistrySM service
purchase	id: String, personalDataType: data	confirmation: String	Make the purchase with the order ID and the user address

the minimum price possible of the product list. After returning the order, the user can send a requiset to the Customer to purchase and the delivery informations. Its API is presented in Table 4.

Table 4: *Customer API*

Method	Input	Output	Description
getPriceOfProductList	products: List<String>	order: Order	Return the minimum price of a product list and the id of the order
purchase	id: String, account: accountType	shipper: String	Make the purchase of the products requested with the operation above
getDeliveryData	shipper: String, orderID: String	delivery: String	Get information about the delivery of an order

Shipper. The Shipper service is responsible for storing information about the delivery of products to an address. The shipper receives information about an order from a Supermarket service and sends delivery information of an order to the Custumer Service. Table 5 shows its API.

4.2.2 The workflow

The choreography has three operations as described in the participant Customer of Section 4.2.1. Before purchasing a list of products, the user must send this list to the Customer to book them and verify the price. After that, the user can purchase them providing the *id* of the order. Finally, it is possible to verify the delivery status to know when the product will arrive. Another message flow that

Table 5: *Shipper API*

Method	Input	Output	Description
setDelivery	id: String, zipcode: String	confirmation: String	Receives the zipcode of an order
getDataAndTime	id: String	time: String	Returns the time that the order will be delivered

does not happen with these operations is the registering of a Supermarket service in the RegistrySM service. Figure 11 describes a global view of the conversation between the services using BPMN2[13].

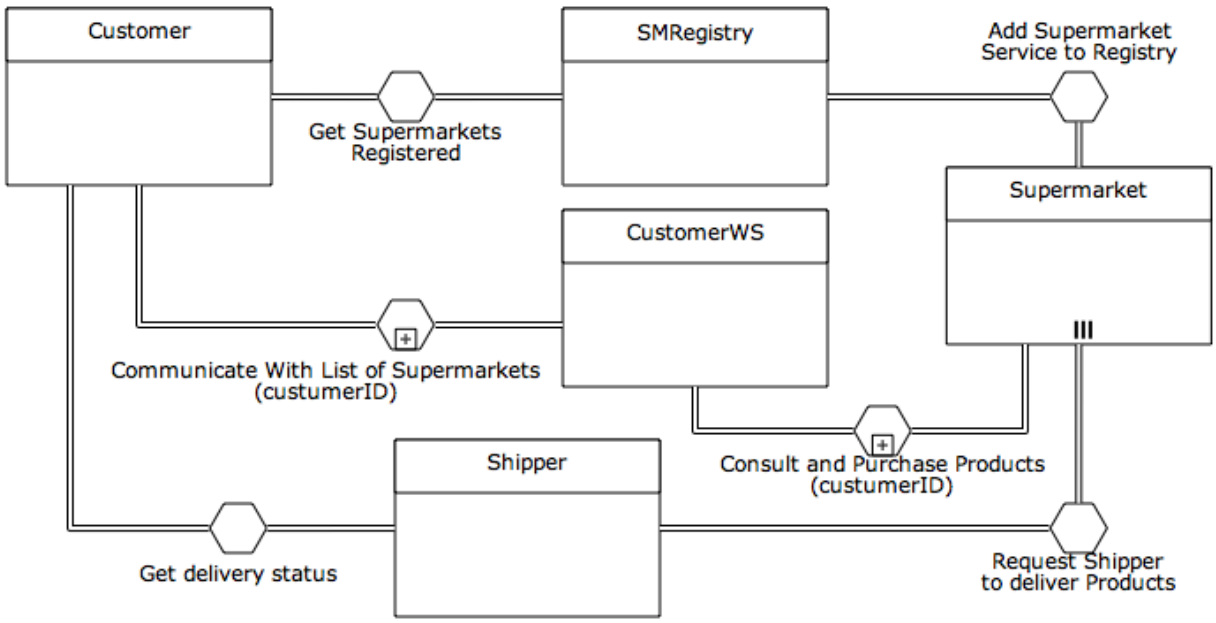

Figure 11: *Global conversation between the services of the Futuremart Choreography*

The *getPriceOfProductList* workflow

Figure 14 shows a BPMN2 Collaboration diagram of the getPriceOfProductList operation. Each lane represents a participant of the choreography. The getPriceOfProductList operation starts with the Customer participant receiving a request with a list of products. It requests a list of Supermarket services registered to the SMRegistry participant. The registry service collects, from its database, all service endpoints stored, and returns it to the Customer.

The Customer stores this list and the product list in its database. Then, it calculates the minimum price possible. The process that calculates the price requests, for each product, the product price to all Supermarkets.

The Supermarket lane represents all Supermarket services, the three little vertical lines inside it symbolize a multiplicity of participants. When each of them receives a request, it verifies in its database what is the price for the specific product to return to the Customer. If the price received is lower then the current lowest, it stores it with the endpoint of the Supermarket service that offers it.

[13] http://www.bpmn.org

After checking out and storing all the products price with their respective supermarkets, the Customer returns the price and an id that represents this order.

The *purchase* workflow

After ordering a list of products and getting the best price for it, the user can request the purchase. The Customer service receives the request with the order ID and the address of the user. Subsequently, the service starts a loop that purchases the products related to each Supermarket service.

When a Supermarket service receives a purchase request, it sends a message to the Shipper service to deliver the products. After that, it returns a message to the Customer with the Shipper service endpoint that will deliver the products.

The Shipper service stores the delivery information in its database for future verifications. Finally, the Customer service returns to the user a confirmation with the Shipper service endpoint that will deliver the products. Figure 15 describes the collaboration between the services that participates in this functionality.

The *getDeliveryData* workflow

The user can also verify the delivery status of an order. The Customer service has the getDeliveryData operation that receives an order ID, and the Shipper service endpoint. It sends a message to the Shipper requesting the status of an order. The latter collects the order information from its database and returns it. As we can see in Figure 16, the Customer simply returns these data to the user.

The *registerSupermarket* workflow

The registerSupermarket operation is not for the final user of the choreography, but for the user that administrates a Supermarket service. A service that implements the Supermarket role can register itself into the RegistrySM service, that has a list of all Supermarket services. Figure 13 describes this registration. A Supermarket service sends a message to register itself. The registrySM service receives this request, stores the service endpoint in its database, and returns a confirmation.

4.2.3 Scalability Testing

To validate this first version of our test bed choreography, we developed three service that implement the Supermarket participant, five service that implement the Customer participant, and one service for each other participants.

We also developed a web service that act as a registry of Customer services. It is possible to request the price of a product list using this service. It divides the received list into a number of blocks that corresponds to the number of services registered. Then, it requests for each service the price with part of the list in parallel aiming at decreasing the response time of the *getPriceOfProductList* operation.

Therefore, the variables chose for the scalability test were (Step 1):

Problem complexity variable: Number of products, n.

Performance variable: Response time, M.

Architecture variable: Number of Customer services registered, Q.

The functions that uses the variables to define the scalability function are (Step 2):

Complexity size of the problem: n

Performance metric: M

Architecture capability: Q

Thus, the scalability function is $\frac{n}{MQ}$. Since n and Q will increase with the same proportion, M should stay constant.

The initial values for the number of products (n) and the number of Customer services (Q) were 1 and 10, respectively. Listing 9 shows a scalability test using our Scalability Testing Framework. In Line 1, we specify how the framework must execute the scalability test. We also specified that the *numberOfCustomers* and *numberOfProducts* parameters should scale in each step.

```
1  @ScalabilityTest(steps = 5, scalabilityFunction = LinearIncrease.class)
2  public List<Long> futureMarketScalabilityTest(@Scale int numberOfCustomers,
3                    @Scale int numberOfProducts)
4                    throws Exception {
5    Item productList = createProductList(numberOfProducts);
6    changeNumberOfCustomers(numberOfCustomers);
7
8    List<Long> responseTimes = new ArrayList<Long>();
9    for(int i=0;i<REPETITIONS;i++) {
10     Long start = System.currentTimeMillis();
11     wsclient.request("getPriceOfProductList", productList);
12     Long end = System.currentTimeMillis();
13     responseTimes.add(end - start);
14   }
15
16   return responseTimes;
17 }
```

Listing 9: *Scalability test for the* getPriceOfProductList *operation*

In line 5, the message with the product list is created and, in line 6, the number of Customer services that participates in this choreography is updated. In line 9 to 14, we specified the sent of requests to the choreography operation. For each step, a number of requests are made and the response times is stored in the *responseTimes* list. The framework collects a list of response times and calculates the mean and the standard deviation, for each iteration.

The graph that represents the result for this scalability test is presented in Figure 12. Since there is only one implementation of the choreography, we could not compare to another implementation. However, we can analyze the derivative of the curve to conclude if this operation is scale enough based on our requirements. As we can see, the average response time of step 5 is four times larger than the first step. We can deduce that to obtained the same performance of step 1 in the step 5, we must increase the architecture capacity four times more than the framework had increased, which is normally not good.

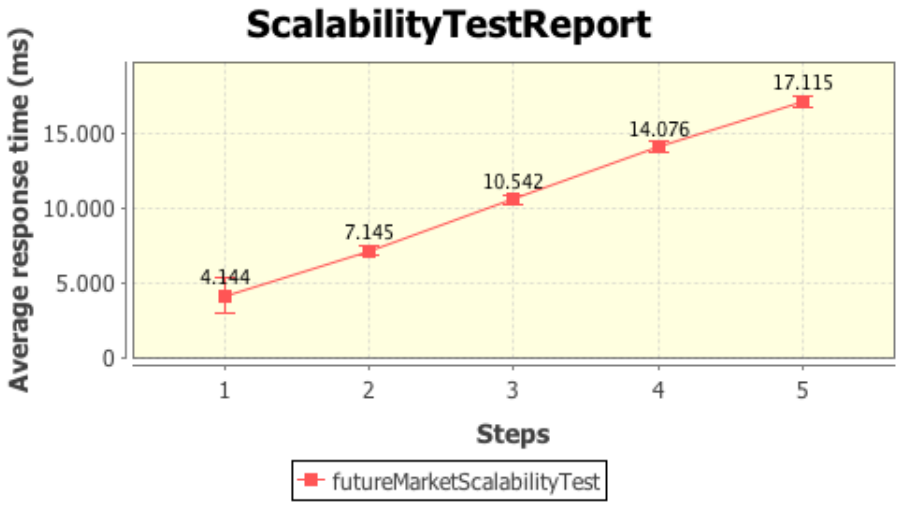

Figure 12: *Scalability Testing Results Graph*

Thus, the next step would be analyze the implementation of the choreography to find bottlenecks. Then, after changes, we would execute again the same test to examine if the modification had increased or not the choreography scalability.

5 Conclusions

In this technical report, we first defined scalability provide a function that quantifies it. To analyze the scalability of an application, using the function defined earlier, a set of steps was described. The first version of the Scalability Testing Framework was developed to support the scalability explorer to analyze the scalability of an application. It automates some of these steps, and assists the visualization of the results by plotting a graph with them.

We have carried two validations with it: the Distributed Matrix Multiplication and the Supermarket of the Future Choreography. In the first one, we compared two different approaches with different scalabilities. Using the framework, we saw the difference between them in the graph, and evaluated which approach was more scalable. Therefore, the framework proved to be a good tool for comparing two similar implementations, and verifying the impact of this difference in the scalability.

In the second validation, we analyzed the scalability of only one approach. Consequently, we could not compare to other approaches and evaluate which one was more scalable. We concluded that, to analyze the scalability with only one implementation, another steps should be done, such as defining an upper bound for the performance metric and calculating the derivative of the curve from the graph to analyze if this value is too high or not.

Since the scalability explorer defines the procedures of scalability testing in the framework specifying test methods, the application can pass through different kinds of scalability assessments. This is important on applications that have different variables that can affect its scalability, such as choreographies that are composed of different implementation layers.

5.1 Ongoing Work

A first experiment with choreographies has been carried out. However, we need to make further experiments with them to explore the different scalability types that are important to choreographies.

In addition, we should develop tools that manipulate cloud instances, where the choreography is usually deployed, to increase the architecture capability. This will facilitate the development of scalability tests for choreographies. The CHOReOS project is developing a series of tools to manipulate them, such as the Node Pool Manager[14], which manages cloud nodes. Nevertheless, these tools might need to be adapted to suit our Scalability Testing Framework.

In this first version, the framework does not make a conclusion of how much the choreography is scalable by itself. Therefore, our group intend to develop a heuristic to determine the scalability size of an application, based on a more profund scalability research and on experiments with choreographies.

[14] https://github.com/choreos/

A BPMN Collaboration Diagrams of the Future Market Choreography

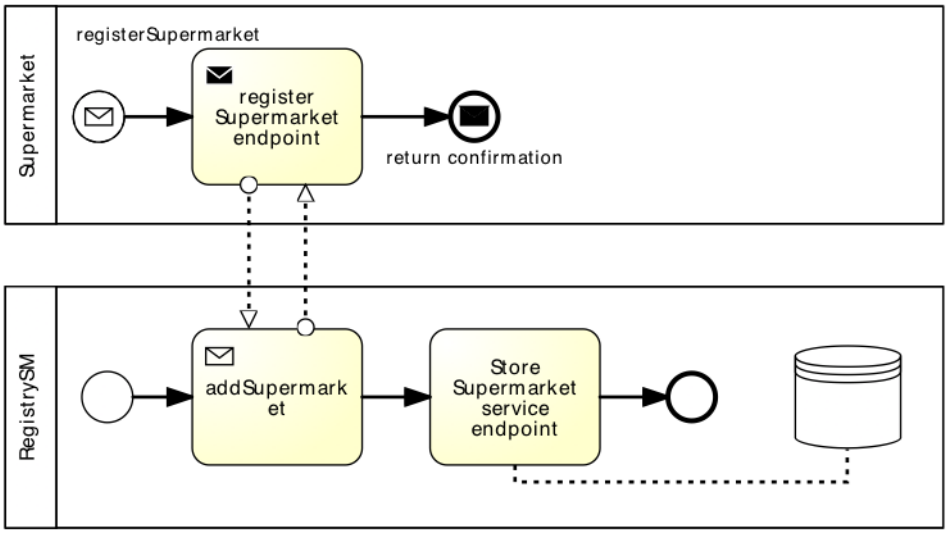

Figure 13: *BPMN Collaboration diagram of the choreography operation* registerSupermarket

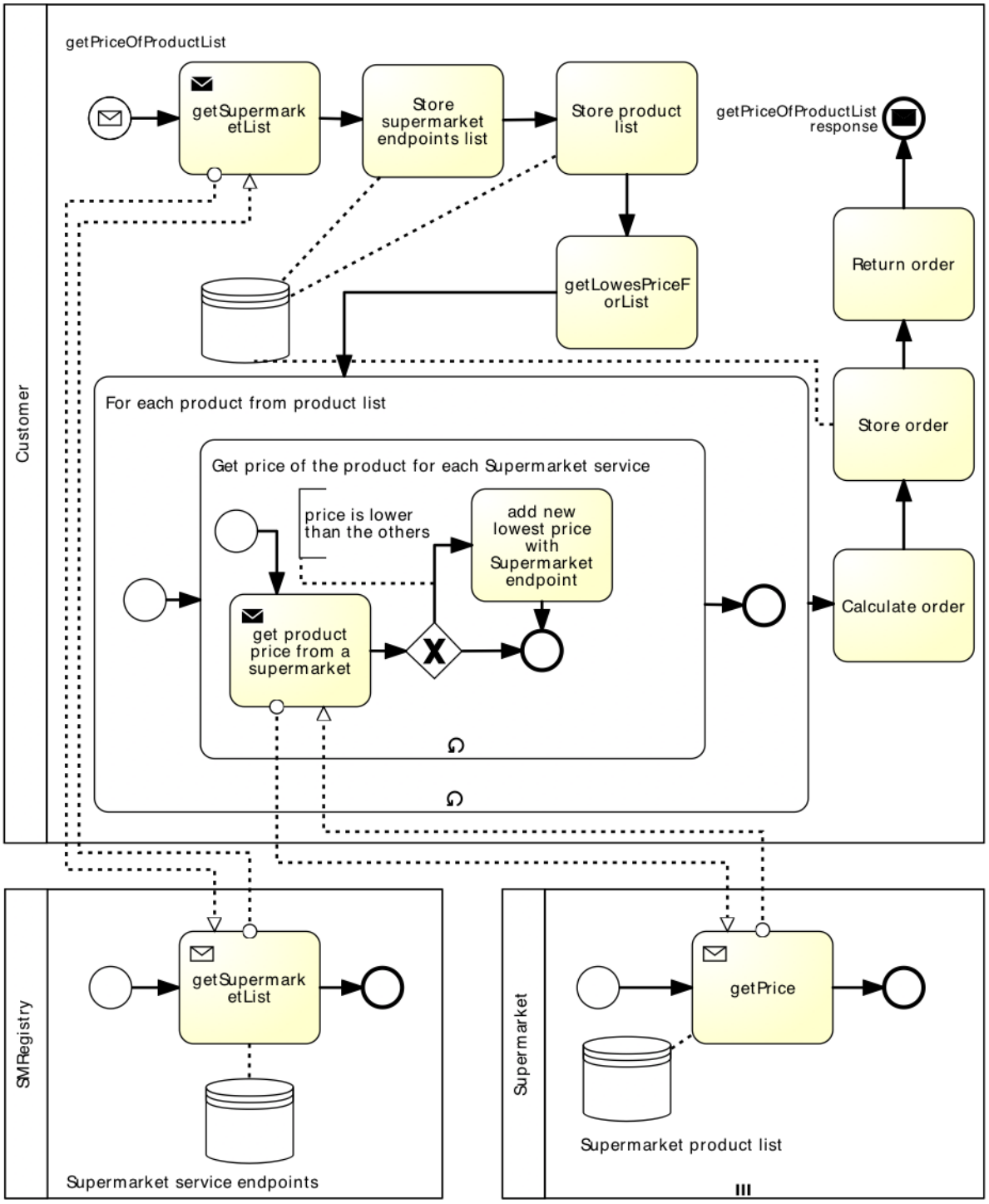

Figure 14: *BPMN Collaboration diagram of the choreography operation* getPriceOfProductList

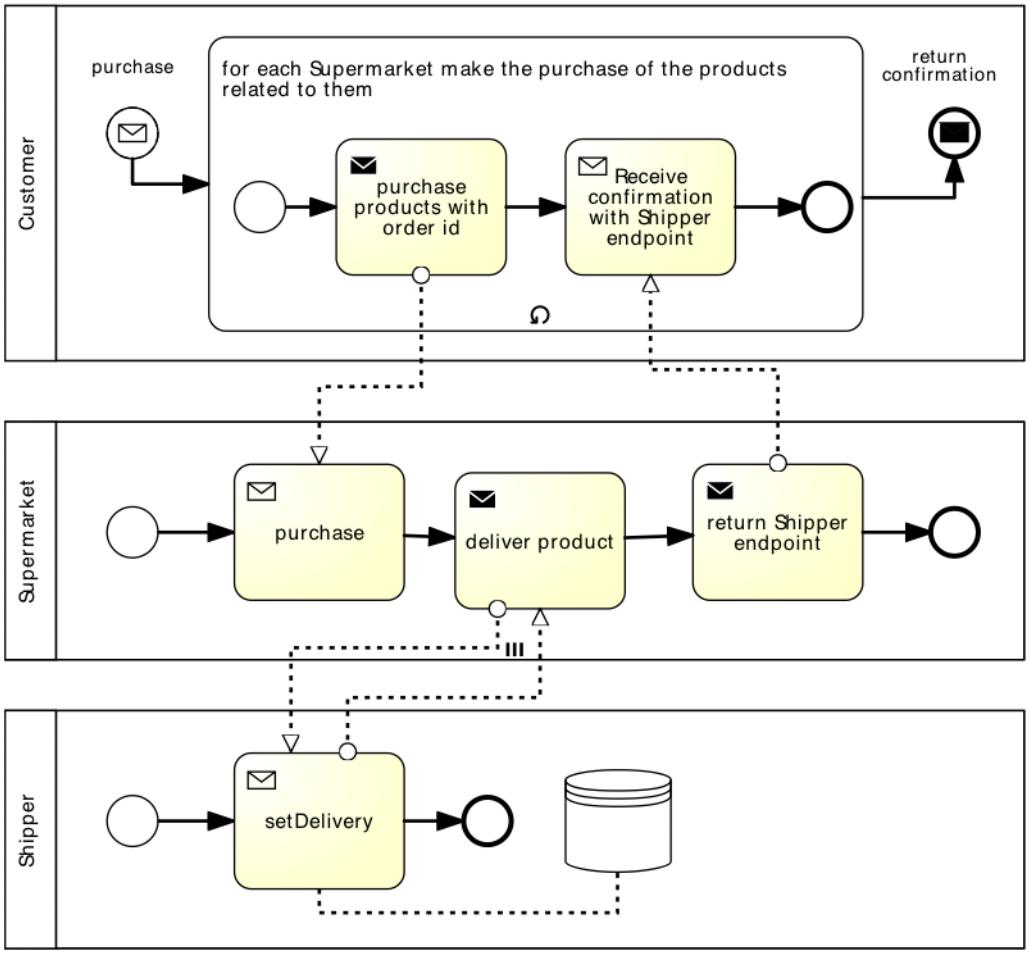

Figure 15: *BPMN Collaboration diagram of the choreography operation* purchase

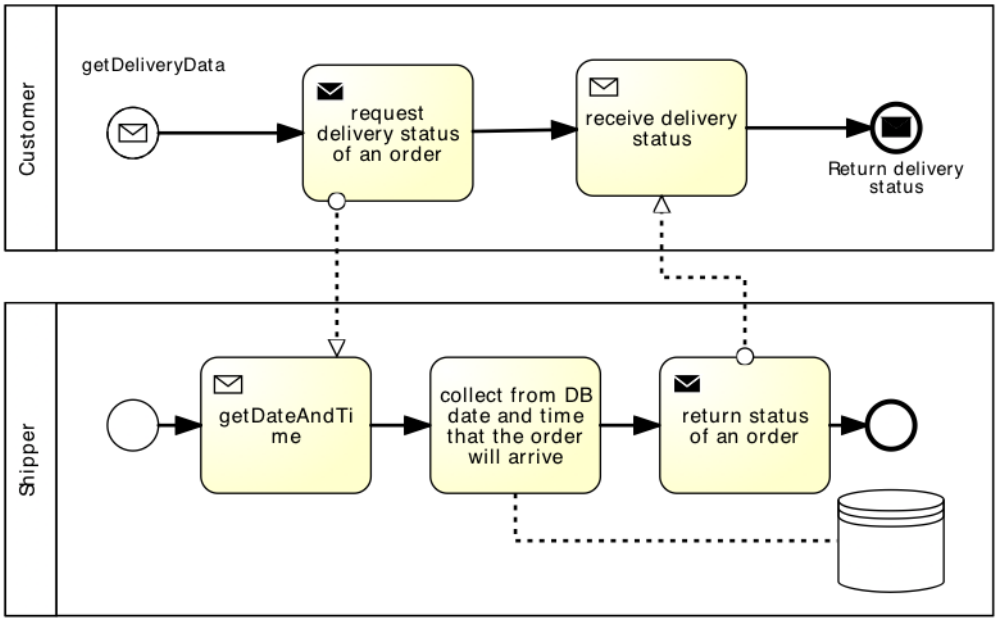

Figure 16: *BPMN Collaboration diagram of the choreography operation* getDeliveryData

References

Bean(2009) James Bean. *SOA and Web Services Interface Design: Principles, Techniques, and Standards*. Elsevier. Referenced at page. 3

Beck(2002) K. Beck. *Test-Driven Development: By Example*. Addison-Wesley Professional. Referenced at page. 3

Besson *et al.***(2011)** Felipe M. Besson, Pedro M.B. Leal, Fabio Kon, Alfredo Goldman, and Dejan Milojicic. Towards automated testing of web service choreographies. In *Proceeding of the 6th international workshop on Automation of software test*, AST '11, pages 109–110, New York, NY, USA. ACM. ISBN 978-1-4503-0592-1. doi: http://doi.acm.org/10.1145/1982595.1982621. URL http://doi.acm.org/10.1145/1982595.1982621. Referenced at page. 3

Bondi(2000) André Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*, WOSP '00, pages 195–203, New York, NY, USA. ACM. ISBN 1-58113-195-X. doi: http://doi.acm.org/10.1145/350391.350432. URL http://doi.acm.org/10.1145/350391.350432. Referenced at page. 7

Chen and Sun(2006) Yong Chen and Xian-He Sun. Stas: A scalability testing and analysis system. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1 –10. doi: 10.1109/CLUSTR.2006.311882. Referenced at page. 7, 8

de Almeida *et al.***(2008)** Eduardo Cunha de Almeida, Gerson Sunye, Yves Le Traon, and Patrick Valduriez. A framework for testing peer-to-peer systems. *Software Reliability Engineering, International Symposium on*, 0:167–176. ISSN 1071-9458. doi: http://doi.ieeecomputersociety.org/10.1109/ISSRE.2008.42. Referenced at page. 8

Gamma *et al.***(1995)** Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, volume 206. Addison Wesley. Referenced at page. 15

Hewitt(2009) Eben Hewitt. *Java Soa Cookbook*. O'Reilly Media, 1st edição. Referenced at page. 3

Law(1998) D.R. Law. Scalable means more than more: a unifying definition of simulation scalability. In *Simulation Conference Proceedings, 1998. Winter*, volume 1, pages 781 –788 vol.1. doi: 10.1109/WSC.1998.745064. Referenced at page. 5

Oracle(1997) Oracle. Trail: RMI, 1997. URL http://docs.oracle.com/javase/tutorial/rmi/index.html. Last access on November 23, 2011. Referenced at page. 17

Peltz(2003) C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46 – 52. ISSN 0018-9162. doi: 10.1109/MC.2003.1236471. Referenced at page. 4

Quinn(1994) M. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, 2nd edição. Referenced at page. 5