Distributed and Resource-Aware Load Testing of WS-BPEL Compositions

Afef Jmal Maâlej[1], Mariam Lahami[1], Moez Krichen[1,2] and Mohamed Jmaïel[1,3]

[1]*ReDCAD Laboratory, National School of Engineers of Sfax, University of Sfax, B.P. 1173, 3038 Sfax, Tunisia*

[2]*Faculty of CSIT, Al-Baha University, Saudi Arabia*

[3]*Digital Research Center of Sfax, B.P. 275, Sakiet Ezzit, 3021 Sfax, Tunisia*

Keywords: Web Services Composition, Distributed Load Testing, Distributed Log Analysis, Performance Monitoring, Resource Awareness, Distributed Execution Environment, Tester Instance Placement.

Abstract: One important type of testing Web services compositions is load testing, as such applications solicit concurrent access by multiple users simultaneously. In this context, load testing of these applications seems an important task in order to detect problems under elevated loads. For this purpose, we propose a distributed and resource aware test architecture aiming to study the behavior of WS-BPEL compositions considering load conditions. The major contribution of this paper consists of (i) looking for the best node hosting the execution of each tester instance, then (ii) running a load test during which the composition under test is monitored and performance data are recorded and finally (iii) analyzing in a distributed manner the resulting test logs in order to identify problems under load. We also illustrate our approach by means of a case study in the healthcare domain considering the context of resource aware load testing.

1 INTRODUCTION

Nowadays, Web services compositions (particularly WS-BPEL[1] (Barreto et al., 2007) (or BPEL) compositions) offer different utilities to hundreds even thousands of users at the same time. An important challenge of testing these applications is load testing (Beizer, 1990), which is frequently performed in order to ensure that a system satisfies a particular performance requirement under a heavy load. In our context, load refers to the rate of incoming requests to a given system during a period of time.

In addition to conventional functional testing procedures, such as unit and integration testing, load testing is a required activity that reveals programming errors, which would not appear if the composition is executed with a small workload or for a short time. They emerge when the system is executed under a heavy load or over a long period of time. On the other hand, a given process may be correctly implemented but fails under some particular load conditions because of external causes (e.g., misconfiguration, hardware failures, buggy load generator, etc.) (Jiang et al., 2008). Hence, it is important to identify and remedy these different problems.

To handle challenges of load testing, we have proposed in a previous work (Maâlej and Krichen, 2015) an approach that combines functional and load testing of BPEL compositions. Indeed, our study is based on conformance testing concept which verifies that a system implementation performs according to its specified requirements. For more details, monitoring BPEL compositions behaviors during load testing was proposed in order to perform later an advanced analysis of test results. This step aims to identify both causes and natures of detected problems. For that, the execution context of the application under test is considered while periodically capturing, under load, some performance metrics of the system such as CPU usage, memory usage, etc.

However, recognizing problems under load is a challenging and time-consuming activity due to the large amount of generated data and the long running time of load tests. During this process, several risks may happen and undermine SUT[2] quality and may even cause software and hardware failures such as SUT delays, memory, CPU overload, node crash, etc. Such risks may also impact the tester itself, which can

[1]Web Services-Business Process Execution Language

[2]System Under Test

Jmal Maâlej, A., Lahami, M., Krichen, M. and Jmaïel, M.
Distributed and Resource-Aware Load Testing of WS-BPEL Compositions.
DOI: 10.5220/0006693400290038
In *Proceedings of the 20th International Conference on Enterprise Information Systems (ICEIS 2018)*, pages 29-38
ISBN: 978-989-758-298-1

produce faulty test results.

To overcome such problems, we extend our previous approach dealing with functional and load testing of BPEL compositions by distribution and resource awareness capabilities. Indeed, supporting test distribution over the network may alleviate considerably the test workload at runtime, especially when the SUT is running on a cluster of BPEL servers. Moreover, it is highly demanded to provide a resource-aware test system, that meets resource availability and fits connectivity constraints in order to have a high confidence in the validity of test results, as well as to reduce their associated burden and cost on the SUT. To show the feasibility of the proposed approach, a case study in the health care domain is introduced, its BPEL process is outlined and it is applied to the context of distributed and resource aware load testing.

The remainder of this paper is organized as follows. Section 2 is dedicated to describe our proposed testing approach for the study of BPEL compositions under load conditions. Then, we describe in Section 3 our resource-aware tester deployment solution. In Section 4, we illustrate our test solution by means of a case study in the healthcare domain. Section 5 contains discussions about some works addressing load testing issue, test distribution and test resource awareness. Finally, Section 6 provides a conclusion that summarizes the paper and discusses items for future work.

2 OUR APPROACH FOR THE STUDY OF WS-BPEL COMPOSITIONS UNDER LOAD

Our proposed approach is based on gray box testing, which is a strategy for software debugging where the tester has limited knowledge of the internal details of the program. Indeed, we simulate in our case the different partner services of the composition under test as we suppose that only the interactions between this latter and its partners are known. Furthermore, we rely on the online testing mode considering the fact that test cases are generated and executed simultaneously (Mikucionis et al., 2004). Moreover, we choose to distribute the testing architecture components on different nodes in order to realistically run an important number of multiple virtual clients.

2.1 Principle of Load Distribution

When testing the performance of an application, it can be beneficial to perform the tests under a typical load. This can be difficult if we are running our application in a development environment. One way to emulate an application running under load is through the use of load generator scripts. For more details, distributed testing is to be used when we reach the limits of a machine in terms of CPU[3], memory or network. In fact, it can be used within one machine (many VMs[4] on one machine). If we reach the limits of one reasonable VM in terms of CPU and memory, load distribution can be used across many machines (1 or many VMs on 1 or many machines). In order to realize remote load test distribution, a test manager is responsible to monitor the test execution and distribute the required load between the different load generators. These latters invoke concurrently the system under test as imposed by the test manager.

2.2 Load Testing Architecture

In this section, we describe a proposed distributed framework for behavior study of BPEL compositions under load conditions (Maâlej and Krichen, 2015). For simplicity reasons, we consider that our load testing architecture is composed, besides the SUT and the tester, of two load generators[5], as depicted in Figure 1.

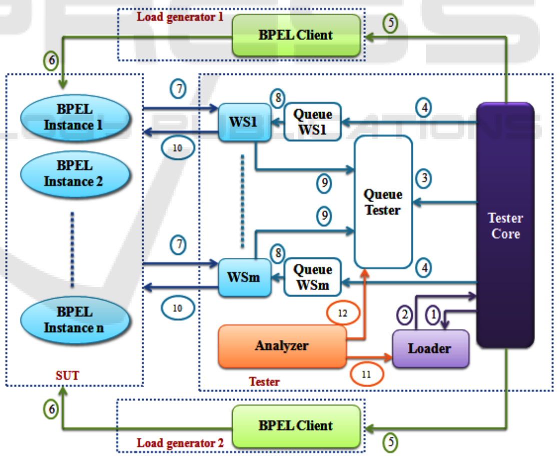

Figure 1: Load Testing Architecture.

As shown in Figure 1, the main components of our proposed architecture are:

- *The System under test (SUT)*: a new BPEL instance is created for each call of the composition under test. A BPEL instance is defined by a unique identifier. Each created instance invokes

[3]Central Processing Unit

[4]Virtual Machines

[5]More machines may be considered as load generators in order to distribute the load more efficiently.

its own partner services instances by communicating while exchanging messages.

- *The tester (Tester)*: it represents the system under test environment and consists of:
  - *The Web services (WS1, ..., WSm)*: these services correspond to simulated partners of the composition under test. For each call of the composition during load testing, new instances of partner services are created.
  - *The Queues (Queue WS1, ..., Queue WSm)*: these entities are simple text files through which partner services and the Tester Core exchange messages.
  - *The Loader*: it loads the SUT specification described in Timed Automata, besides the WSDL files of the composition under test and the WSDL files of each partner service. Moreover, it defines the types of input/output variables of the considered composition as well as of its partner services.
  - *The Tester Core*: it generates random input messages of the BPEL process under test. It communicates with the different partner services of the composition by sending them the types of input and output messages. In case of partner services which are involved in synchronous communications, the Tester Core sends information about their response times to the composed service. Finally, it distributes the load between the two generators. It orders each one to perform (more or less) half of concurrent calls to the composition under test, and passes in parameters the time between each two successive invocations besides the input variable(s) of the system.
  - *The test log (QueueTester)*: it stores the general information of the test (number of calls of the composition under test, the delay between the invocation of BPEL instances, etc.). Also it saves the identifiers of created instances, the invoked services, the received messages from the SUT, the time of their invocations and the verdict corresponding to checking of partner input messages types. This log will be consulted by the Analyzer to verify the functioning of the different BPEL instances and to diagnose the nature and cause of the detected problems.
  - *The test analyzer (Analyzer)*: this component is responsible for offline analysis of the test log QueueTester. It generates a final test report and identifies, as far as possible, the limitations of the tested composition under load conditions.
- *The load generators (BPEL Client)*: these entities meet the order of the Tester Core by performing concurrent invocations of the composed service. For that, they receive from the tester as test parameters the input(s) of the composition under test, the number of required process calls and the delay between each two successive invocations.

Besides, we highlight that load testing of BPEL compositions in our approach is accompanied by a module for the monitoring of the execution environment performances, aiming to supervise the whole system infrastructure during the test. Particularly, this module permits the selection, before starting test, of the interesting metrics to monitor, and then to display their evolution in real-time. In addition, the monitoring feature helps in establishing the correlation between performance problems and the detected errors by our solution.

2.3 Automated Advanced Load Test Analysis Approach

Current industrial practices for checking the results of a load test mainly persist ad-hoc, including high-level checks. In addition, looking for functional problems in a load testing is a time-consuming and difficult task, due to the challenges such as no documented system behavior, monitoring overhead, time pressure and large volume of data. In particular, the ad-hoc logging mechanism is the most commonly used, as developers insert output statements (e.g. printf or System.out) into the source code for debugging reasons (James et al., 2010). Then most practitioners look for the functional problems under load using manual searches for specific keywords like *failure*, or *error* (Jiang, 2010). After that, load testing practitioners analyze the context of the matched log lines to determine whether they indicate functional problems or not. Depending on the length of a load test and the volume of generated data, it takes load testing practitioners several hours to perform these checks.

However, few research efforts are dedicated to the automated analysis of load testing results, usually due to the limited access to large scale systems for use as case studies. Automated and systematic load testing analysis becomes much needed, as many services have been offered online to an increasing number of users. Motivated by the importance and challenges of the load testing analysis, an automated approach was proposed in (Maâlej and Krichen, 2015) to detect functional and performance problems in a load test by analyzing the recorded execution logs and performance metrics. In fact, performed operations during load testing of BPEL compositions are stored in QueueTester. In order to recognize each BPEL instance which is responsible for a given action, each one starts with the identifier of its corresponding BPEL instance (BPEL-ID). At the end of test running, the Analyzer consults QueueTester.

Hence, our automated log analysis technique takes as input the stored log file (QueueTester) during load testing, and goes through three steps as shown in Figure 2:

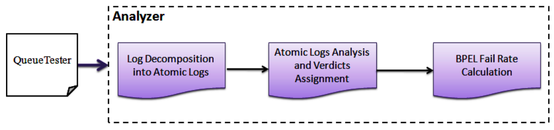
Analyzer
QueueTester
Log Decomposition into Atomic Logs
Atomic Logs Analysis and Verdicts Assignment
BPEL Fail Rate Calculation

Figure 2: Automated Log Analysis Technique.

- *Decomposition of QueueTester*: based on BPEL-ID, the Analyzer decomposes information into atomic test reports. Each report is named BPEL-ID and contains information about the instance which identifier is BPEL-ID.
- *Analysis of atomic logs*: the Analyzer consults the generated atomic test reports of the different BPEL instances. It verifies the observed executed actions of each instance by referring to the specified requirements in the model (Timed Automata). Finally, the Analyzer assigns corresponding verdicts to each instance and identifies detected problems.
- *Generation of final test report*: this step consists in producing a final test report recapitulating test results relatively to all instances and also describing both nature and cause of each observed FAIL verdict.

It is true that our load test analysis is automated. Yet, the analysis of the atomic logs is performed sequentially for each BPEL instance, which may be costly especially in term of time execution. Another limitation consists in using only one instance of tester and thus one analyzer for each test case. To solve this issue, we propose to use more than one instance of the tester which are deployed in distributed nodes and connected to the BPEL process under test. Each tester instance studies the behavior of the composition considering different load conditions (SUT inputs, number of required process calls, delay between each two successive invocations, etc.).

3 CONSTRAINED TESTER DEPLOYMENT

In this section, we deal with the assignment of tester instances to test nodes while fitting some resource and connectivity constraints. The main goal of this step is to distribute efficiently the load across virtual machines, computers, or even the cloud. This is crucial for the test system performance and for gaining confidence in test results.

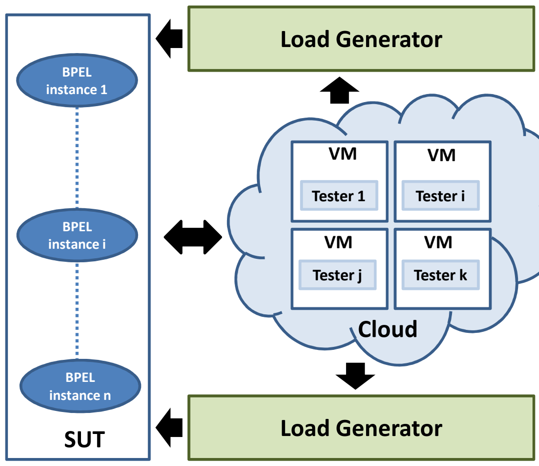

Figure 3: Distributed Load Testing Architecture.

Figure 3 illustrates the distributed load testing architecture in which several tester instances are created and connected to the BPEL process under test. These instances run in parallel in order to perform efficiently load testing.

Recall that each tester instance includes an analyzer component that takes as input the generated atomic test reports of the different BPEL instances, and generates as output a Local Verdict (LV). As outlined in Figure 4, we propose a new component called Test System Coordinator which is mainly charged with collecting the local verdicts generated from the analyzer instances and producing the Global Verdict (GV). As depicted in Algorithm 1, if all local verdicts are PASS, the global verdict will be PASS. If at least one local verdict is FAIL (respectively INCONCLUSIVE), the global verdict will be FAIL (respectively INCONCLUSIVE).

Once the distributed load testing architecture is elaborated, we have to assign efficiently its components (i.e.,its tester instances) to the execution nodes. To do so, we have defined two kinds of constraints that have to be respected in this stage: resources and connectivity constraints. They are detailed in the following subsections.

3.1 Formalizing Resource Constraints

In general, load testing is seen as a resource consuming activity. Consequently, the consideration of resource allocation during test distribution should be

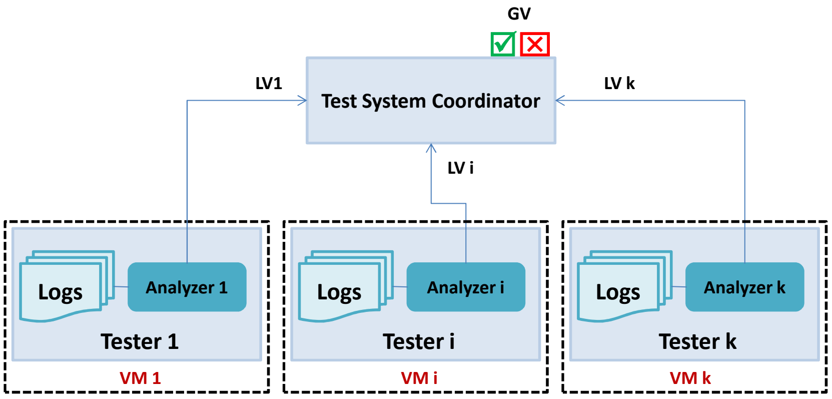

Figure 4: Distributed Load Test Analysis Architecture.

Algorithm 1: Generation of the Global Verdict.

Input: The array *LocalVerdicts*.
Output: The global verdict.

```
1:  BEGIN
2:  for $i = 0$ to $LocalVerdicts.size - 1$ do
3:    if ($LocalVerdicts[i]$==FAIL) then
4:      return FAIL
5:    end if
6:    if ($LocalVerdicts[i]$==INCONCLUSIVE) then
7:      return INCONCLUSIVE
8:    end if
9:  end for
10: return PASS
11: END
```

applied with the aim of decreasing test overhead and increase confidence in test results.

For each node in the test environment, three resources are monitored: the available memory, the current CPU load and the energy level (i.e., battery). The value of each resource can be directly captured on each node through the use of internal monitors. Formally, they are represented through three vectors: C that contains the provided CPU, R that provides the available RAM[6] and E that introduces the energy level.

For each tester instance, we introduce the memory size (i.e., the memory occupation needed by a tester during its execution), the CPU load and the energy consumption properties. We suppose that these values are provided by the test manager or computed after a preliminary test run. Similarly, they are formalized over three vectors: D_c that contains the required CPU, D_r that introduces the required RAM and D_e that contains the required energy by each tester.

As the proposed approach is resource aware, checking resource availability during test distribution is usually performed before starting the load testing process. Thus, the overall required resources by n tester instances must not exceed the available resources in m nodes. This rule is formalized through three constraints to fit as outlined by (1) where the two dimensional variable x_{ij} can be equal to 1 if the tester instance i is assigned to the node j, 0 otherwise.

$$\begin{cases} \sum_{i=1}^{n} x_{ij} dc_i \leq c_j & \forall j \in \{1, \cdots, m\} \\ \sum_{i=1}^{n} x_{ij} dr_i \leq r_j & \forall j \in \{1, \cdots, m\} \\ \sum_{i=1}^{n} x_{ij} de_i \leq e_j & \forall j \in \{1, \cdots, m\} \end{cases} \quad (1)$$

3.2 Formalizing Connectivity Constraints

Dynamic environments are characterized by frequent and unpredictable changes in connectivity caused by firewalls, non-routing networks, node mobility, etc. For this reason, we have to pay attention when assigning a tester instance to a host computer by finding at least one path in the network to communicate with the component under test.

$$x_{ij} = 0 \quad \forall j \in forbiddenNode(i) \quad (2)$$

where the $forbiddenNode(i)$ function returns a set of forbidden nodes for a test component i.

Finding a satisfying test placement solution is merely achieved by fitting the former constraints (1) and (2).

3.3 Optimizing the Tester Instance Placement Problem

Looking for an optimal test placement solution consists in identifying the best node to host the concerned

[6]Random Access Memory

tester in response with two criteria: its distance from the node under test and its link bandwidth capacity. To do so, we are asked to attribute a profit value p_{ij} for assigning the tester i to a node j. For this aim, a matrix $\mathcal{P}_{n\times m}$ is computed as follows:

$$p_{ij} = \begin{cases} ********0 & \text{if} \quad j \in forbiddenNode(i) \\ maxP - k \times step_p & \text{otherwise} \end{cases} \tag{3}$$

where $maxP$ is constant, $step_p = \frac{maxP}{m}$, k corresponds to the index of a node j in a *Rank Vector* that is computed for each node under test. This vector corresponds to a classification of the connected nodes according to both criteria: their distance far from the node under test and their link bandwidth capacities.

As a result, the constrained tester instance approach generates the best deployment host for each tester instance involved in the load testing process by maximizing the total profit value while fitting the former resource and connectivity constraints. Thus, it is formalized as a variant of the Knapsack Problem, called *Multiple Multidimensional Knapsack Problem* (MMKP)(Jansen, 2009).

$$MMKP = \begin{cases} maximize \quad \mathcal{Z} = \sum_{i=1}^{n} \sum_{j=1}^{m} p_{ij} x_{ij} & (4) \\ subject \quad to \quad (1) \quad and \quad (2) & \\ \sum_{j=1}^{m} x_{ij} = 1 \quad \forall i \in \{1, \cdots, n\} & (5) \\ x_{ij} \in \{0,1\} \quad \forall i \in \{1, \cdots, n\} & \\ and \quad \forall j \in \{1, \cdots, m\} & \end{cases}$$

Constraint (4) corresponds to the objective function that maximizes tester instance profits while satisfying resource (1) and connectivity (2) constraints. Constraint (5) indicates that each tester instance has to be assigned to at most one node.

To solve such a problem, a well-known solver in the constraint programming area, namely Choco (Jussien et al., 2008), is used to compute either an optimal or a satisfying solution of the MMKP problem (Lahami et al., 2012).

3.4 Advantages and Limitations

Related to our previous work (Maâlej and Krichen, 2015) in which the test system is centralized on a single node, several problems may occur while dealing with load testing. In fact, we have noticed that not only the SUT, which is made up of several BPEL instances, can be affected by this heavy load but also the relevance of the obtained test results. In order to increase the performance of such test system and get confidence in its test results, testers and analyzers are distributed over several nodes. In this case, load testing process is performed in parallel.

Moreover, our proposal takes into consideration the connectivity to the SUT and also the availability of computing resources during load testing. Thus, our work provides a cost effective distribution strategy of testers, that improves the quality of testing process by scaling the performance through load distribution, and also by moving testers to better nodes offering sufficient resources required for the test execution.

Recall that the deployment of our distributed and resource aware test system besides BPEL instances is done on the cloud platform. It is worthy to note that public cloud providers like Amazon Web Services[7] and Google Cloud Platform[8] offer a cloud infrastructure made up essentially of availability zones and regions. A region is a specific geographical location in which public cloud service providersdata centers reside. Each region is further subdivided into availability zones. Several resources can live in a zone, such as instances or persistent disks. Therefore, we have to choose the VM instances in the same region to host the testers, the analyzers and the SUT. This is required in order to avoid the significant overhead that can be introduced when the SUT and the test system components are deployed in different regions on the cloud.

4 ILLUSTRATION THROUGH TRMCS CASE STUDY

This section is mainly dedicated to show the relevance of our approach through a case study in the healthcare domain. Subsections below details the adopted scenario, its business process and also its application in the context of resource aware load testing.

4.1 Case Study Description

Called Teleservices and Remote Medical Care System (TRMCS), this case study consists of providing monitoring and assistance to patients suffering from chronic health problems. Due to the new needs of modern medicine, it can be enhanced with more elaborated functionalities like the acquisition, the analysis and the storage of biomedical data.

The adopted business process is highlighted through a BPEL process in Figure 5. We assume that it composes of several Web services namely Storage Service (SS), Analysis Service (AnS), Alerting Service (AlS) and Maintenance Service (MS).

[7]https://aws.amazon.com/fr/

[8]https://cloud.google.com/

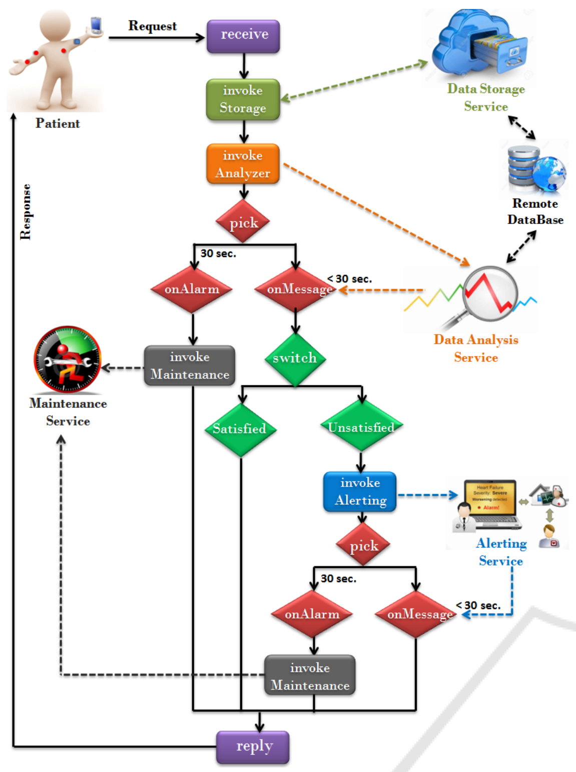

Figure 5: The TRMCS Process.

For a given patient suffering from chronic high blood pressure, measures like his arterial blood pressure and his heart-rate beats per minute are collected periodically (for instance three time per day). For the two collected measures, a request is sent to the TRMCS process. First, the *Storage Service* is invoked to save periodic reports in the medical database. Then, the *Analyzer Service* is charged with analyzing the monitored data in order to detect whether some thresholds are exceeded. This analysis is conditioned by a processing time. Indeed, the process should receive a response from the AnS before reaching 30 seconds. Otherwise, the process send a connection problem report to the Maintenance Service. In case of receiving the analysis response before reaching 30 seconds, two scenarios are studied. If thresholds are satisfied, a detailed reply is sent to the corresponding patient. Otherwise, the Alerting Service is invoked in order to send urgent notification to the medical staff (such as doctors, nurses, etc.).

Similar to the Analysis Service, the Alerting Service is constrained by a waiting time. If medical staff are notified before reaching 30 seconds, the final reply is sent to the corresponding patient. Otherwise, the Maintenance Service is invoked.

We suppose that the TRMCS application can be installed in different cities within a country. Thus, we should consider that several BPEL servers are required to handle multiple concurrent patient requests. From a technical point of view, we adopted Oracle BPEL Process Manager[9] as a solution for designing, deploying and managing the TRMCS BPEL process. We also opted for Oracle BPEL server. The major question to be tackled here is how to apply our proposal of load testing TRMCS BPEL composition in a cost effective manner and without introducing side effects?

4.2 Distributed and Resource Aware Load Testing of TRMCS Process

With the aim of checking the satisfaction of performance requirements under a heavy load of patients requests, we apply our distributed and resource aware load testing approach. For simplicity reasons, we focus at this stage on studying the load testing of a single BEPL server while the load is handled by several testers simulating concurrent patient requests. To do so, we consider a test environment made up of four nodes: a server node ($N1$) and three test nodes ($N2$,$N3$ and $N4$). As illustrated in Figure 6, we suppose that this environment has some connectivity problems. In fact, the node $N4$ is forbidden to host tester instances because no route is available to communicate with the BPEL instance under test.

To perform distributed load tests efficiently and without impacting test results, tester instances Ti have to be deployed in this test environment while fitting resources and connectivity constraints (e.g., memory and energy consumption, link bandwidth, etc.).

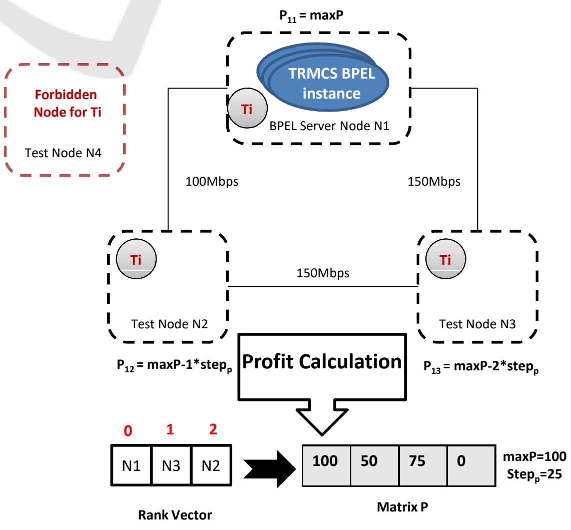

Figure 6: Illustrative Example.

[9]http://www.oracle.com/technetwork/middleware/bpel/

Thus, we look for a best placement solution of a given tester Ti. First of all, the node $N4$ is discarded from the tester placement process because the link with the BPEL server is broken. Consequently, the variable x_{i4} is equal to zero. Second, we compute the *Rank Vector* for the rest of connected test nodes and we deduce the profit matrix. We remark here that the profit p_{ij} is maximal if the tester Ti is assigned to the server node $N1$ because assigning a tester to its corresponding node under test and performing local tests reduces the network communication cost. This profit decreases with respect to the node index in the *Rank Vector*. For instance, $N3$ is considered a better target for Ti than $N2$ although they have the same distance far from the server node because the link bandwidth between $N3$ and $N1$ is greater than the link bandwidth between $N2$ and $N1$.

For example, in the case of four nodes and a given tester Ti (see Figure 6), the optimal solution of placement can be the test node $N1$. Consequently, the generated variable x_i is as follows:

$$x_i = \begin{pmatrix} 1, 0, 0, 0 \end{pmatrix}$$

5 RELATED WORKS

In the following, we discuss some existing works addressing load testing in general, test distribution and test resource awareness.

5.1 Existing Works on Load Testing

Load testing and performance monitoring become facilitated thanks to existing tools. In fact, load testing tools are used for software performance testing in order to create a workload on the system under test, and measure response times under this load. These tools are available from large commercial vendors such as Borland, HP Software, IBM Rational and Web Performance Suite, as well as Open source projects. Web sites. Krizanic et al (Krizanic et al., 2010) analyzed and compared several existing tools which facilitate load testing and performance monitoring, in order to find the most appropriate tools by criteria such as ease of use, supported features, and license. Selected tools were put in action in real environments, through several Web applications.

Despite the fact that commercial tools offer richer set of features and are in general easier to use, available open source tools proved to be quite sufficient to successfully perform given tasks. Their usage requires higher level of technical expertise but they are a lot more flexible and extendable.

There are also different research works dealing with load and stress testing in various contexts. Firstly, Yang and Pollock (Yang and Pollock, 1996) proposed a technique to identify the load sensitive parts in sequential programs based on a static analysis of the code. They also illustrated some load sensitive programming errors, which may have no damaging effect under small loads or short executions, but cause a program to fail when it is executed under a heavy load or over a long period of time. In addition, Zhang and Cheung (Zhang and Cheung, 2002) described a procedure for automating stress test case generation in multimedia systems. For that, they identify test cases that can lead to the saturation of one kind of resource, namely CPU usage of a node in the distributed multimedia system. Furthermore, Grosso et al. (Grosso et al., 2005) proposed to combine static analysis and program slicing with evolutionary testing, in order to detect buffer overflow threats. For that purpose, the authors used of Genetic Algorithms in order to generate test cases. Garousi et al. (Garousi et al., 2006) presented a stress test methodology that aims at increasing chances of discovering faults related to distributed traffic in distributed systems. The technique uses as input a specified UML 2.0 model of a system, extended with timing information. Moreover, Jiang et al. (Jiang et al., 2008) and Jiang (Jiang, 2010) presented an approach that accesses the execution logs of an application to uncover its dominant behavior and signals deviations from the application basic behavior.

Comparing the previous works, we notice that load testing concerns various fields such as multimedia systems (Zhang and Cheung, 2002), network applications (Grosso et al., 2005), etc. Furthermore, all these solutions focus on the automatic generation of load test suites. Besides, most of the existing works aim to detect anomalies which are related to resource saturation or to performance issues as throughput, response time, etc. Besides, few research efforts, such as Jiang et al. (Jiang et al., 2008) and Jiang (Jiang, 2010), are devoted to the automated analysis of load testing results in order to uncover potential problems. Indeed, it is hard to detect problems in a load test due to the large amount of data which must be analyzed. We also notice that the identification of problem cause(s) (application, network or other) is not the main goal behind load testing, rather than studying performance of the application under test, this fact explains why few works address this issue. However, in our work, we are able to recognize if the detected problem under load is caused by implementation anomalies, network or other causes. Indeed, we defined and validated our approach based on interception of exchanged messages between the composition under test and its partner services. Thus it would be possible to monitor exchanged messages instantaneously, and to recognize what is the cause behind their loss or probably their reception delay, etc. Studying the existing works on load testing, we remark that the authors make use of one instance tester for both the generation and execution of load test cases. To the best of our knowledge, there is no related work that proposes to use multiple testers at the same time on the same SUT. Thus we do not evoke neither testers placement in different nodes nor their management.

5.2 Existing Works on Test Distribution

The test distribution over the network has been rarely addressed by load testing approaches. We have identified only two approaches that shed light on this issue.

In the first study (Bai et al., 2006), the authors introduce a light-weight framework for adaptive testing called *Multi Agent-based Service Testing* in which runtime tests are executed in a coordinated and distributed environment. This framework encompasses the main test activities including test generation, test planning and test execution. Notably, the last step defines a coordination architecture that facilitates mainly test agent deployment and distribution over the execution nodes and test case assignment to the adequate agents.

In the second study (Murphy et al., 2009), a distributed in vivo testing approach is introduced. This proposal defines the notion of *Perpetual Testing* which suggests the proceeding of software analysis and testing throughout the entire lifetime of an application: from the design phase until the in-service phase. The main contribution of this work consists in distributing the test load in order to attenuate the workload and improve the SUT performance by decreasing the number of tests to run.

Unlike these approaches, our work aims at defining a distributed test architecture that optimizes the current resources by instantiating testers in execution nodes while meeting resource availability and fitting connectivity constraints. This has an important impact on reducing load testing costs and avoiding overheads and burdens.

5.3 Existing Works on Test Resource Awareness

As discussed before, load testing is a resource-consuming activity. In fact, computational resources are used for generating tests if needed, instantiating tester instances charged with test execution and finally starting them and analyzing the obtained results. Notably, the bigger the number of test cases is, the more resources such as CPU load, memory consumption are used. Hence, we note that the intensive use of these computational resources during the test execution has an impact not only on the SUT but also on the test system itself. When such a situation is encountered, the test results can be wrong and can lead to an erroneous evaluation of the SUT responses.

To the best of our knowledge, this problem has been studied only by Merdes work (Merdes et al., 2006). Aiming at adapting the testing behavior to the given resource situation, it provides a resource-aware infrastructure that keeps track of the current resource states. To do this, a set of resource monitors are implemented to observe the respective values for processor load, main memory, battery charge, network bandwidth, etc. According to resource availability, the proposed framework is able to balance in an intelligent manner between testing and the core functionalities of the components. It provides in a novel way a number of test strategies for resource aware test management. Among these strategies, we can mention, for example, *Threshold Strategy* under which tests are performed only if the amount of used resources does not exceed thresholds.

Contrary to our distributed load testing architecture, this work supports a centralized test architecture.

6 CONCLUSION & FUTURE WORK

In this paper, we firstly described our contribution for the study of BPEL compositions behaviors under various load conditions. Then, we explained the principle of test logs analysis phase. Indeed, test results are exhaustively analyzed and advanced information are provided by our tester.

In order to better the system performance and responsiveness, we also proposed in this work a tester deployment solution that considers many distributed tester instances. Thus, we extended a previous tool with test distribution and resource awareness capabilities. To do so, we adopted a distributed test architecture in which several BPEL instances are running in a given execution node and several tester instances as well as load generators are running in several test nodes. With the aim of avoiding the network burden, the placement of tester instances was performed with respect to resource availability and network connectivity. As illustrative example, a case study in the healthcare domain is introduced and applied in the context of resource aware load testing.

A promising future work would be to support the scalability issue by opting, as an example, for the load balancing concept. In fact, implementing clustering mechanisms presents an interesting technique to improve the performance of distributed BPEL servers deploying the same SUT. Thus, incoming BPEL client requests should be equally distributed among the servers to achieve quick response. Indeed, the principle of load balancing is to realize a distribution of tasks to some machines in an intelligent way.

REFERENCES

Bai, X., Dai, G., Xu, D., and Tsai, W. (2006). A Multi-Agent Based Framework for Collaborative Testing on Web Services. In *Proceedings of the 4th IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, and the 2nd International Workshop on Collaborative Computing, Integration, and Assurance (SEUS-WCCIA'06)*, pages 205–210.

Barreto, C., Bullard, V., Erl, T., Evdemon, J., Jordan, D., Kand, K., Knig, D., Moser, S., Stout, R., Ten-Hove, R., Trickovic, I., van der Rijn, D., and Yiu, A. (2007). *Web Services Business Process Execution Language Version 2.0 Primer*. OASIS.

Beizer, B. (1990). *Software Testing Techniques (2Nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA.

Garousi, V., Briand, L. C., and Labiche, Y. (2006). Traffic-aware stress testing of distributed systems based on UML models. In *Proceedings of ICSE'06*, pages 391–400, Shanghai, China. ACM.

Grosso, C. D., Antoniol, G., Penta, M. D., Galinier, P., and Merlo, E. (2005). Improving network applications security: a new heuristic to generate stress testing data. In *Proceedings of GECCO'05*, pages 1037–1043, Washington DC, USA. ACM.

James, H. H., Douglas, C. S., James, R. E., and Aniruddha, S. G. (2010). Tools for continuously evaluating distributed system qualities. *IEEE Software*, 27(4):65–71.

Jansen, K. (2009). Parametrized Approximation Scheme for The Multiple Knapsack Problem. In *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'09)*, pages 665–674.

Jiang, Z. M. (2010). Automated analysis of load testing results. In *Proceedings of ISSTA'10*, pages 143–146, Trento, Italy. ACM.

Jiang, Z. M., Hassan, A. E., Hamann, G., and Flora, P. (2008). Automatic identification of load testing problems. In *Proceedings of ICSM'08*, pages 307–316, Beijing, China. IEEE.

Jussien, N., Rochart, G., and Lorca, X. (2008). Choco: an Open Source Java Constraint Programming Library. In *Proceeding of the Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, pages 1–10.

Krizanic, J., Grguric, A., Mosmondor, M., and Lazarevski, P. (2010). Load testing and performance monitoring tools in use with ajax based web applications. In *33rd International Convention on Information and Communication Technology, Electronics and Microelectronics*, pages 428–434, Opatija, Croatia. IEEE.

Lahami, M., Krichen, M., Bouchakwa, M., and Jmaiel, M. (2012). Using knapsack problem model to design a resource aware test architecture for adaptable and distributed systems. In *ICTSS*, pages 103–118.

Maâlej, A. J. and Krichen, M. (2015). Study on the limitations of WS-BPEL compositions under load conditions. *Comput. J.*, 58(3):385–402.

Merdes, M., Malaka, R., Suliman, D., Paech, B., Brenner, D., and Atkinson, C. (2006). Ubiquitous RATs: How Resource-Aware Run-Time Tests Can Improve Ubiquitous Software Systems. In *Proceedings of the 6th International Workshop on Software Engineering and Middleware (SEM'06)*, pages 55–62.

Mikucionis, M., Larsen, K. G., and Nielsen, B. (2004). T-uppaal: Online model-based testing of real-time systems. In *Proceedings of ASE'04*, pages 396–397, Linz, Austria. IEEE Computer Society.

Murphy, C., Kaiser, G., Vo, I., and Chu, M. (2009). Quality Assurance of Software Applications Using the In Vivo Testing Approach. In *Proceedings of the 2nd International Conference on Software Testing Verification and Validation (ICST'09)*, pages 111–120.

Yang, C. D. and Pollock, L. L. (1996). Towards a structural load testing tool. *SIGSOFT Softw. Eng. Notes*, 21(3):201–208.

Zhang, J. and Cheung, S. C. (2002). Automated test case generation for the stress testing of multimedia systems. *Softw., Pract. Exper.*, 32(15):1411–1435.