A Workload Model for Benchmarking BPEL Engines

George Din
Fraunhofer FOKUS
Berlin, Germany
din@fokus.fraunhofer.de

Klaus-Peter Eckert
Fraunhofer FOKUS
Berlin, Germany
eckert@fokus.fraunhofer.de

Ina Schieferdecker
Technical University of Berlin and,
Fraunhofer FOKUS, Berlin, Germany
schieferdecker@fokus.fraunhofer.de

Abstract

This paper describes a benchmarking workload model for Business Process Execution Language (BPEL) Engines for Web Services. The proposed model is based on simulation of real world traffic conditions by defining a set of requirements which best characterize the end-users. The performace characteristics are evaluated on top of collected measurements such as success/fail rate, response times or round-trip delays which are then used to identify problems for scalability or usability aspects under heavy load.

1 Introduction

A benchmark evaluates the performance of a system by monitoring the system while it is being exposed to a particular workload [10]. The activity to select and define a workload is called *workload characterization* [4] and it has the goal to produce models that are capable of describing and reproducing the behavior of a workload. There are different ways to characterize the workload. The simplest approach is based on collecting data for significant periods of time in the production environment [9]. These representative workloads are then used to determine system performance for what it is likely to be when run in production. One issue is that the empirical data is not complete or not available when creating new technologies [5]. In these situations, partial data can be collected from similar systems and may serve to create realistic workloads. Other approaches are based on modeling formalisms such as Markov chains [6] or Petri networks [8] to derive models for the workload which are then used to generate performance tests.

The Web Services Business Process Execution Language *WS-BPEL 2.0* [2], standardized by the OASIS consortium, is today's most commonly used language for the specification of orchestrated Web services. WS-BPEL is an XML based language that is used to describe the flow of information and control between Web services. BPEL processes are executed in so called BPEL engines that are often implemented as plug-ins to common Web/Application servers as Tomcat. These engines provide a complete run-time environment for concurrent BPEL processes that are triggered by respectively trigger themselves distributed Web services. This paper describes the interdependencies between the configuration of a BPEL engine and its performance, utilizing the open source engine *ActiveBPEL* [3].

This paper is structured as follows. After the introduction, Section 2 presents the benchmarking methodology. The tool built on top of this methodology is briefly described in Section 3. In Section 4 we apply the benchmarking concepts to evaluate the performance of the ActiveBPEL engine. The paper ends with conclusions and outlook section.

2 Benchmarking Methodology

A benchmark consists of three main elements: *test scenarios*, *benchmark tests* which instantiate the scenarios, and *benchmark test reports* generated out of execution traces.

Test Scenario. An individual interaction path is called a *test scenario* and it is described by its message flow between the talking entities. An example for a test scenario message flow is presented in Fig. 1. A message flow consists of an arbitrary number of transactions (at least one); each transaction describes the *request type*, its associated *response type* and a *maximal response time*. If, during the execution, the response time exceeds the maximal response time, or if the response does not match the expected response type, the transaction counts as *inadequately handled scenario* (IHS).

Performance Metrics. For each test scenario, *metrics* and *design objectives* are defined. Typical *metrics* include test scenario outcome, response times and message rates. The *design objectives* (DO) define the threshold values which are used to compute the metrics. They are of two types:

- *design objectives for delays* e.g. maximal time until a response is received

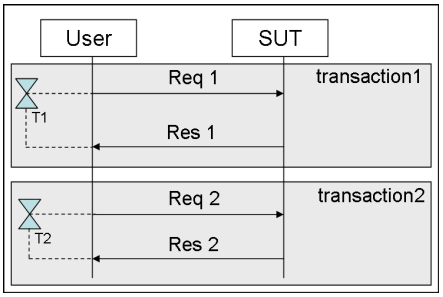

Figure 1. A Test Scenario Example

- *design objectives for error rates* e.g. threshold for allowed percentage of errors out of the total number of scenarios.

Design Objective Capacity. If during the benchmark execution, the rate of failed test scenarios goes above a predefined limit (for example 0.1%) then the SUT has reached its Design Objective Capacity (DOC). The DOC indicates the overload performance limit of the SUT and it represents the threshold for the accepted QoS. It is however, the output number which globally characterizes the performance of the tested system. The DOC can be used as capacity indicator for the overall performance of the system but also for comparison with other systems.

Benchmark Test. A *benchmark test* combines different test scenarios into a *traffic set*. Within a traffic set, each test scenario has an associated relative occurrence frequency which indicates how often a test scenario should be instantiated. The occurence frequency is defined as parameter, therefore, various configurations can be easily experimented. The execution of a benchmark test implies that the selected test scenarios are executed at the same time. Each started test scenario becomes a *scenario attempt*. The load rate applied to the SUT is called *Scenario Attempts per Second* (SAPS). When the frequency of IHS exceeds a predefined threshold, then the *design objective capacity* (DOC) has been exceeded.

Test Procedure. The benchmark execution consists of a sequence of several benchmark steps in order to measure the DOC of the tested system. After the execution, we validate whether the threshold for the DOC has been reached by investigating if the rate percentage of inadequate handled scenarios (IHS) goes above a threshold (i.e. 0.1%). We extend the execution trials until we find a load at which the error rate is below the threshold and another load at which the error rate exceeds the threshold. At this moment we decide to stop the test and consider the lower load value as the DOC of the tested system.

Test Parameters. The *benchmark test parameters* are used to control the behaviour of the test script. Such parameters have to be defined for any benchmark in order to allow the tester tune the load generation before the execution. The most important parameters are: the number of users, the amount by which the scenario arrival rate is increased, the number of steps in a benchmark test, the amount of time for a test to be executed with a given system load (a test step) before incrementing the load.

Benchmark Report. The *benchmark report* is generated after the execution of a benchmark test. The report contains a full description of the SUT configuration, the TS configuration, the process used to generate the system loads at each SUT reference point, and data series reporting the benchmark metrics as a function of time. Some of these graphs are presented in Section 4.

3 Benchmark Implementation

The prototype implementation is based on the TTCN-3 [7] language which is used to specify the behavior of the benchmark tests. The main reason for this selection is that TTCN-3 as a standardised test language has been increasingly accepted in the industry as test specification language. Additionally, various features offered by this language make it a suitable technology to implement benchmark tests.

In TTCN-3, the parallelism is realized by running in parallel a number of test components. The load is generated by a `SenderComponent`. Each call created by the load generator is associated to an `EventHandler`, which will handle all required transactions for that call. The number of EventHandlers is arbitrary and depends on the number of simulated users and on the performance of the hardware running the test system.

The behaviours of the test components are defined as functions. A function is used in benchmark tests to specify client activities within a test scenario. A simulated user may behave in different ways when interacting with the SUT, thus the test system may need different functions implementing different client behaviours.

The current implementation supports the execution of different types of test scenarios at the same time. The tool can run up to 10000 parallel users, each user acting as a separate entity. The DOs are evaluated offline on top of the execution measurements. The measurements are then used to derive performance metrics; the most important are: SAPS, transaction latency, IHS%.

4 Case Study: BPEL Engine

The Web Services Business Process Execution Language *WS-BPEL 2.0* [2] standardized by the OASIS consortium is today's most commonly used language for the specification of orchestrated Web services. A BPEL process can be designed using synchronous or asynchronous communication patterns. A synchronous process behaves like an

RPC server (receive-reply) while an asynchronous process follows a message based receive-send pattern utilizing the invoker's callback interface. The specification of the BPEL process comprises the definition of the information flow respectively the control flow between the process activities. These activities can be assignments to process variables or parallel respectively sequential invocations of other Web services.

Every time a BPEL process receives a triggering request message the run-time environment, which is called *BPEL engine*, has to decide if this message has to be routed to an existing instance of the process or if a new instance has to be created. This decision process is specified in the process itself. Every receive statement has a textual attribute that defines if a new instance should be created in case the process is triggered by a corresponding message. Additionally the concept of *correlations* is used to define correlated invocations of a process instance, based on the value of selected input and/or output parameters of the message. Depending on these correlations the BPEL engine decides if an incoming message should be dispatched to an existing process instance or if a new instance should be created.

Thus, every BPEL engine must be able to manage an unpredictable number of concurrent instances of BPEL processes that are associated to the different BPEL processes that have been deployed on the engine. Different threads have to be allocated and assigned to the external and internal events that may occur during the execution of a BPEL process. Appropriate policies for the management of the engine's message queues have to be established. Depending on the implementation of the chosen BPEL engine the configuration of these properties can be either hard coded or adjustable by the engine's administrator. The behaviour of the engine and especially its performance depends heavily on the chosen configuration parameters.

The analyzed *ActiveBPEL engine* running in a Tomcat environment uses the concept of *Work Managers* developed by BEA and IBM for the implementation of the engine's default behaviour. A work manager provides a simple API for application-server-supported concurrent execution of work items. This allows to schedule work items for concurrent execution. The work manager provides common "join" operations, such as waiting for any or all work items to complete. The *Timer and Work Manager for Application Servers* [1] specification provides an application-server-supported alternative to using language-level threading APIs. The administration tool for the ActiveBPEL engine allows to specify the minimum and maximum number of work manager threads that can be used by the engine. Additionally several timeout values for Web services responses or unresolved correlations can be specified [3].

Including the possible configuration attributes of a Web Server like Tomcat there are several ways to affect the behaviour of a BPEL engine. Because current BPEL engines offer mostly poor possibilities and related documentation for run-time configuration and especially the interdependencies of different attributes of an engine it is necessary to evaluate the effects of different configuration parameters utilizing appropriate test mechanisms.

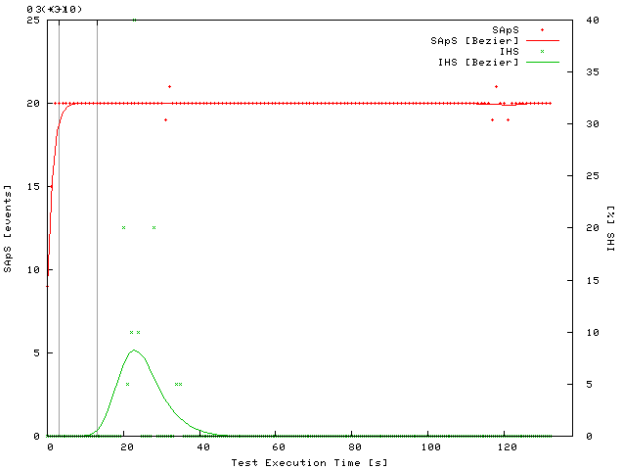

Figure 2. Example of a system load of 20 SAPS

For this purpose, we have used a simple BPEL process that consists of two steps. In the initialization phase a new process instance is created and returns its process identification to the caller. In the second phase the instance is called again using the process identification as a correlation. The process returns with a local time stamp and idles for a specified time before its termination to simulate the situation that several concurrent processes are running in the BPEL engine.

Figure 2 provides an example of a benchmark test with a system load of 20 calls per second applied to an ActiveBPEL engine installed on a Pentium 4 PC with 3.0 Ghz CPU and 768 Mb memory. We modified the default configuration of the ActiveBPEL engine by increasing the "Work Manager Thread Pool Max" to 1000 and by setting the size of the Java heap to 512 Mb. Additionally, we also increased the number of parallel threads in the Tomcat server to 1000 (the default value is 150). These modifications were necessary, since, for the default configuration, the ActiveBPEL is either running out of memory or it is running out of available threads. The test simulates 2000 users and applies the load for a duration of 2 minute. These values are selected arbitrary and can be configured as test parameters. The upper line shows the applied system load of 20 SAPS which means that the test system is initiating each second 20 BPEL processes. The lower line indicates the rate of errors which is drawn as a percentage out of the initiated calls. This line

shows an increase of the error rate up to 10% after the first 20 seconds but the SUT is capable of recovering from this situation and it can sustain the load without any further errors.

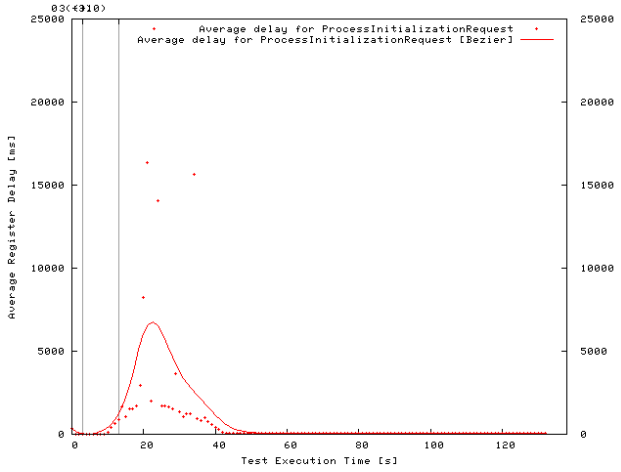

Figure 3. Average latency of process initialization

In the first phase we are able to measure and compare the time a BPEL engine needs to create a process, in the second phase we are able to measure the time the engine needs to activate the possibly dehydrated process. The SUT latency (i.e. response time) is reported separately in another type of graph, for each type of request. Figure 3 shows the average latency of process initialization operation corresponding to the benchmark test described above.

In the simulation scenarios we have used different workloads to check the performance and reliability of the BPEL engine depending on the number of parallel invocations and concurrent processes. In parallel we have used different numbers of the working threads and compared the performance of ActiveBPEL engine running on Tomcat.

We initially evaluated the ActiveBPEL SUT for short test durations, e.g. 2-5 minutes. During this test campaign we discovered the limit of 150 threads of the Tomcat web container and the thread pool limit of the BPEL engine. However, these are configuration parameters which can be optimized for better performance. At a later stage, we decided to increase the duration of the test runs to 30 minutes with the purpose to investigate whether the performance remains the same or not. Unfortunately, even for low system loads the SUT ran out of memory and, consequently, it cannot create processes anymore. This problem is detected at the test system side as an increase of the fail rate to 100% shortly after the "out of memory" exception occured. This problem is related to the number of parallel processes active at the same time. The ActiveBPEL engine keeps all these processes in memory, therefore, the longer the processes live, the faster the "out of memory" exception occurs.

5 Conclusions

This paper presented the design of a benchmark tool capable of evaluating the performance of BPEL engines for configurable workloads. The approach describes the concepts of test scenarios, test procedures, metrics and design objectives as a general method to describe workloads on a common basis.

The current experiments have been executed with a workload which consists of two operations only (but fairly enough to measure the response time of the BPEL engine at process creation and initialization). In future we intend to extend the workload with more operations which should occur after process creation. We expect that the performance of the BPEL engines will definitely suffer due to workloads with different parallel test scenarios. Adding more operations to the test scenarios will automatically also result into a longer duration of the processes lives which will cause that the BPEL engine will be able to create even less processes during the benchmark execution time.

References

[1] Timer and work manager for application servers. http://dev2dev.bea.com/wlplatform/commonj/twm.html.

[2] Web services business process execution language version 2.0.

[3] Activebpel development server user's guide, 2007.

[4] A. Avritzer, J. Kondek, D. Liu, and E. J. Weyuker. Software performance testing based on workload characterization. pages 17–24, Rome, Italy, 2002. ACM. ISBN 1-58113-563-7.

[5] S. Barber. Creating effective load models for performance testing with incomplete empirical data. In *Web Site Evolution, 2004. WSE 2004. Proceedings. Sixth IEEE International Workshop on*, pages 51– 59, 2004. ISBN 1550-4441.

[6] M. Beyer, W. Dulz, and F. Zhen. Automated ttcn-3 test case generation by means of uml sequence diagrams and markov chains. *ats*, 00:102, 2003. ISSN 1081-7735.

[7] ETSI. ETSI European Standard (ES) 201 873-1 V3.2.1 (2007-02) The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language, 2007. Sophia-Antipolis, France.

[8] S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo. Software performance modelling using pepa nets. pages 13–23, Redwood Shores, California, 2004. ACM.

[9] Y. Liu, I. Gorton, A. Liu, N. Jiang, and S. Chen. Designing a test suite for empirically-based middleware performance prediction. pages 123–130, Sydney, Australia, 2002. Australian Computer Society, Inc. ISBN 0-909925-88-7.

[10] A. J. Smith. Workloads (creation and use). *Commun. ACM*, 50:45–50, 2007.