Performance characterization of black boxes with self-controlled load injection for simulation-based sizing

Ahmed Harbaoui*
France Telecom R&D
Grenoble, France

Bruno Dillenseger†
France Telecom R&D
Grenoble, France

Jean-Marc Vincent‡
Laboratoire LIG, Projet MESCAL
Grenoble, France

Abstract **Sizing and capacity planning are key issues that must be addressed by anyone wanting to ensure a distributed system will sustain an expected workload. Solutions typically consist in either benchmarking, or modelling and simulating the target system. However, full-scale benchmarking may be too costly and almost impossible, while the granularity of modelling is often limited by the huge complexity and the lack of information about the system. We propose a methodology that combines both solutions by first identifying a middle-grain model made of interconnected black boxes, and then to separately characterize the performance and resource consumption of these black boxes. We also propose a component-based supporting architecture, introducing control theory issues in a general approach to autonomic computing infrastructures.**

I. INTRODUCTION

"Many organisations expensively invest to build distributed systems applications and web services and pay a huge amount of money to maintain and keep the environment up-to-date. In most cases, the overall capacity planning and procurement is done without a defined methodology"[7].

This kind of situation is responsible for important loss of incomes, ranging from losing customers on an on-line purchase service to losing stock exchange transactions. Hence, it shows the utility of an infrastructure's capacity planning to support the associated load. In this context, our work comes from the problem of planning capacity of a distributed infrastructure to support a given load. While simulation techniques are developed in order to predict the performances, and to detect the bottlenecks and critical resources, the preliminary modelling phase of the system typically encounters opacity problems when a certain level of granularity is reached. Then, "Black boxes" appear, either because of a lack of information about their behaviour, or because of their great complexity. However, the modelling of the global system is impossible without a minimal model of these black boxes, including resources consumption. In this paper we deal with the problem of modelling parts of the system as black boxes. Some works studied methods for black boxes characterization. [6], [9] use an analytical model by considering the whole system as a one black box. We start in section 2 a discussion on the different approaches concerning the estimation and the determination of performance models. Then, we present the approach enabling the determination of parameters influencing our system. In the third section, we present the CLIF framework and this integration to our approach. Section 4 deals with problems of stability and saturation. The next section experiments our approach with a simple example. Finally, we give some ideas to study in future work.

II. MODELS DISCUSSION

Our goal is to generate black boxes models. These black boxes result from a lack of information concerning the behaviour and resource consumption, or a high level of complexity of some parts of the global system. Then, the generated models will be integrated in the global system model. With regard to this modelling problem, several approaches may be adapted. To begin, we present these approaches:

- *analytical modelling* consists in reducing the system in a mathematical model and analyzing it numerically. Several mathematical tools enable such a modelling: automata, Petri nets, probability approach (queuing network), etc.;
- *simulation* consists in establishing a simplified model for the system by using suitable software. This technique is commonly used to evaluate performance;
- with *traffic emulation*, direct measurements and analysis are carried out on the system. It gives a better understanding of the system's real behaviour. This kind of modelling does not need detailed information about the system. The model is generally built only by considering the outputs versus the inputs.

The software systems we want to qualify are distributed and complex. In general, they suffer from a lack of information describing their behaviours and interactions with their

environment. In addition, we cannot access their source code. All these reasons make direct modelling a hard and complex task and lead us to adopt a traffic emulation approach since it does not require such information.

III. METHODOLOGY

The traffic emulation approach gives the performance model by considering the system output as a function of the input load. Load is injected in the system in order to qualify its capacities and to extract performances and resources consumption before saturation.

A. Defining Black Boxes

This part consists in identifying the black boxes of a system. Depending on the system, one tries to divide it into as many black boxes as possible. When decomposition becomes too complex, the system must be kept fully. Otherwise, we define mutual interactions among the black boxes and other parts of system. In fact, interactions could be external invocations of other black boxes, system calls, access to resources, etc.

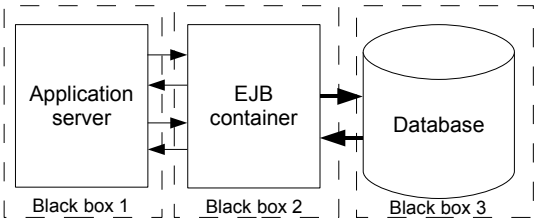

Figure 1. Example of J2EE application

Let us take the example of a J2EE web application, composed of an application server, an EJB container and a database. In such an architecture, an intuitive decomposition is possible that splits the system into three black boxes (see figure 1). The first one is dedicated to the application server, the second to the EJB container and the third to the database. This way, we obtain a more detailed and precise performance model.

B. Choosing the Performance Parameters

The parameters are the different characteristics that impact system performance. They depend on the type of target system and fixed goals. If we take the previous example - J2EE application - parameters could be: end-to-end response time, throughput in requests per second, number of customers per time unit, etc.

Given the important number of parameters that could influence the system performance and the huge amount of time needed for performances study, it looks more suitable to consider only relevant parameters which are directly linked to the aim of the study. If the choice looks difficult, a "factorial analysis" will enable to identify the actually important factors, through some experiments. In our J2EE example, we chose response time as the interesting performance factor.

C. Defining Workload and Instrumenting

Once the black boxes are identified, we define the load to apply through several uses cases and we execute the test. In our case (J2EE application), the load is defined through a number of typical usages consisting of interlaced sequences of requests and think times, and a parallel execution of a number of virtual users performing those usages.

However, since we want a good qualification of both the black box and the global model, it's necessary to apply a load that is as close as possible to the real load. In order to reach this goal, the testing platform may repeatedly replay pieces of real execution traces. Instrumentation deals with monitoring and measuring the use of resources (CPU, memory allocation and network occupation) by placing some probes in different parts of the system under test.

D. Modelling

Once we have collected performance measures associated to the applied loads, we will extract performance model based on these results. In order to model the system with queuing networks, we model each black box with a queue. Each queue is labeled by the performance characterization obtained in previous step. These queues could be represented in three different ways depending on the type of the black box. With *load-dependent resources*, queuing and service times depend on the load D.

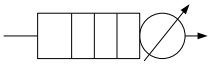

Figure 2. Queue for a load-dependent resource

The two other queue models are just particular cases of this model: *load-independent resources* represent resources where the service time does not depend on the load; *delay resources*' service time does not depend on the load and there is no queuing.

We have to identify the type of each black box (load-independent, load-dependent, etc.) according to the test results. The load test is executed on each black box. If our system is composed of several interacting black boxes, we define software-*plugs*. They replace interactions of the tested black box with other black boxes while conserving a constant value for performance parameters of interest. Then, one subtracts this constant from the value obtained from the test and hence we get the black box characterization. Let us

return to our example of the J2EE application, to characterize black box 1 which interacts with box 2. One develops a software-plug that replaces box 2 with constant response times for each invocation. At the end of the tests, one withdraws software-plug constant from the global response time to obtain the first black box one.

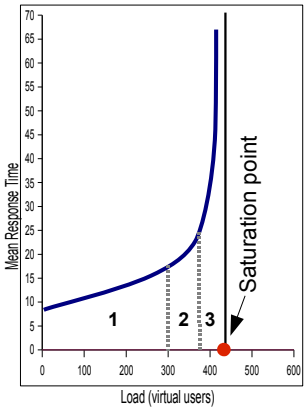

Figure 3. saturation point

After carrying out all tests, we draw response time as a function of the applied load. The result, as we expect, should be close to the one sketched in figure 3. In portion 1 in figure 3, response time linearly grows with load, which is a correct behavior for resource-shared processing. In portion 2, we observe the beginning of the effect of application contention. Approaching the saturation point, the system does not follow the imposed load any more, and its response time tends to infinity.

IV. SATURATION AND STABILITY OF SYSTEMS

All measurements should be done when the system reaches the limit just before saturation. However, if we wish to reach saturation, load injection should be done in such a manner that enables to be more and more close to this situation. First of all, one injects a minimum load and waits for the system to become stable. Then, one progressively increases the load to a higher level, waits for stability, and so on (see figure 5). This method could take a huge time depending on the system. This is why we propose in section V an infrastructure to automatically find the saturation point. To achieve this, the load injector is controlled through a feedback loop that observes the system response to the current load and makes the decision to increase or decrease the load with reference to the measured performances (cf. figure 4).

When looking for the saturation point, we must ensure that the system is stable during all the load ramp-up in order to get reliable and accurate results. The system is stable if its performance remains the same whenever the workload

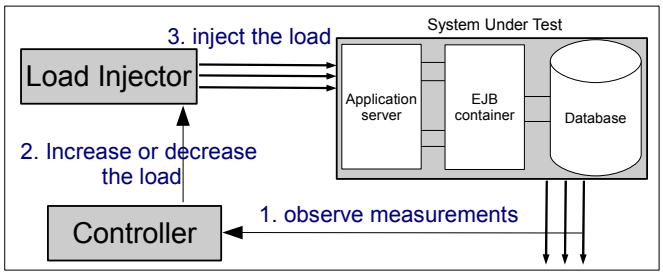

Figure 4. Load injection feedback loop

keeps the same. If the load ramp-up is too steep, it may be difficult to clearly identify the unstability area corresponding to the saturation point. For this reason, we have to maintain a constant load during a sufficient duration for the system to reach a stable state. Then, the duration as well as the load level for the following step depend on the response of the system to the current load level.

Stability criteria depend on the kind of system and the quality of service that must be provided. These criteria must be defined at the very beginning, just like the global performance parameters of interest. For example, in the case of a J2EE application, we may choose the maximum variation of response time as a stability criterion.

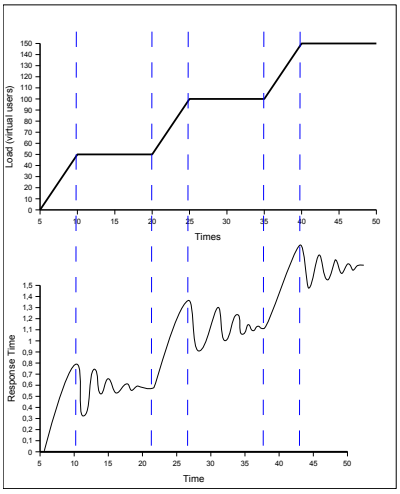

Figure 5. typical response time evolution under a step-by-step workload

The first graph of figure 5 shows the progressive level of load used to reach saturation. The applied load is a step-by-step function that enables to wait for stability once the load has been increased. If the stability condition is satisfied, we proceed with the next step (higher level load). Otherwise, we decrease the load until we obtain a stable situation. The second graph illustrates this load injection policy through a response time-based stability criterion. It sketches the vari-

ation of the system's response time as a function of the applied load. The system clearly reacts to the different load levels with an increase of the average response time and oscillations around the average that decrease while the load remains constant. When the response time oscillations are small enough to match the stability condition, we increase the load once again. This is repeated until the saturation point is reached.

V A component-based supporting infrastructure

A *An architectural approach to autonomic computing*

In order to experiment our methodology, we propose a practical software infrastructure that fits, on the one hand, genericity (our approach may be applied to any kind of black box), and, on the other hand, autonomy (self-regulated load injection). This is the reason why this work is carried out in the context of architectural research on autonomic computing. This approach has been proposed in [3], and is currently being developed in collaborative projects Selfware [2] and Selfman [8, 2].

As presented in [5], the basic idea of autonomic computing may be summarized as the principle of using computing power to automatically (autonomously) manage computing systems complexity. Our architectural approach to autonomic computing consists in relying on a uniform component-based representation of the target computing system, either in a native manner or a wrapper-based manner. Then, a feedback loop is introduced, with sensors at one end (observation), actuators at the other end (reaction/feedback control), and a decision element in between. The feedback loop relies on a communication middleware to handle observation events coming from the sensors, as well as reaction events coming from the decision elements to the actuators. More than just a plain transport service, this event middleware may also support aggregation, filtering and a variety of message delivery models (e.g. publish/subscribe, group communication). All elements in this architecture are uniformly represented and handled as components, using the Fractal component model [1].

B *CLIF Load Injection Framework*

Starting from this component-based and feedback loop-based architectural approach, we need to build a self-regulated load injection system. We need components that generate a workload on the System Under Test (*SUT*), and components that give feedback information about the resulting SUT performance (response time, throughput) and computing resource usage. Moreover, there should be a decision component that closes the feedback loop between observation and reaction, in order to dynamically and autonomously adapt the generated workload.

CLIF [4] provides a framework of Fractal components that meets these requirements. Main components are: *load injectors* for traffic generation and response times measurement, *probes* for monitoring the consumption of arbitrary computing resources, and a *supervisor* component which is bound to all injectors and probes and provides a central point of control and monitoring. While the typical CLIF usage consists in plugging a user interface on the supervisor, we are simply going to bind an autonomic *controller* component to the supervisor and discard the user interface. This is actually done by developing this controller component and slightly modifying an XML file describing the CLIF application, using a commonly called Architecture Description Language. As shown by figure 6, this controller component is bound to other components:

- a load injection policy component that computes the control feedback on the load injection system;
- a saturation policy component, that detects whether the SUT is saturated or not.

that computes the control feedback on the load injection system according to the observation of response times, resource usage and possible alarms. Both components rely on the observation of response times, resource usage and possible alarms.

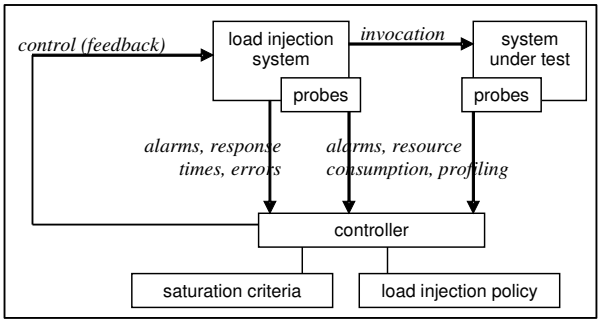

Figure 6. A CLIF assembly for self-regulated load injection

In order to vary the load level during the saturation look-up process, we use the classical virtual user concept supported by CLIF. A virtual user is a computer program that invokes the SUT in a similar way that a real user would do. Load testing consists in massively and concurrently running virtual users. Each CLIF load injector is actually an execution engine for such virtual users. Then, the workload regulation performed by the controller component simply consists in adjusting the number of virtual users run by the load injectors according to the observation. Here, it must be underlined that the load injection policy can be generic, since it may only handle the concept of virtual user whatever the

actual SUT is. Pure control theory-based algorithms may apply there. As far as the saturation policy is concerned, it may be defined in a generic manner also, but it may be chosen or parameterized in adequacy with the SUT. Simple, generic saturation detectors are: response time threshold, error or alarm occurrence, or request throughput stagnation.

VI An experiment

A Rationale

We propose a self-regulated load injection experiment based on our component-based architectural approach to autonomic computing, using the Fractal model and CLIF load testing framework. The target system under test is an Enterprise Service Bus (*ESB*), a kind of request broker used in Service Oriented Architectures to support mediation features such as accounting, routing, logging, security, management of service level agreement, etc. This ESB is the black box we want to characterize from the performance point of view. The system clients are emulated by virtual users running in CLIF load injectors and generating SOAP requests. Real services are replaced by software plugs, i.e. dummy services that reply to requests with a constant response time, whatever the incoming workload. Of course, the plugs' performance must be qualified before, to determine this response time and the correct operating range with regard to the incoming traffic throughput.

With this simple experiment, we are just going to show how the looped load injection system is going to find the ESB saturation limit, in terms of maximum sustainable number of virtual users and request throughput, according to a given saturation criteria. The behavior of our virtual users consists in generating 20 requests during 20 seconds before exiting, with random think times between consecutive requests, which gives an average of 1 request per second per virtual user. The ESB is based on a dedicated hardware platform, which offers load percentage information through the SNMP protocol. We have defined a new CLIF probe to get this information.

The controller starts with one virtual user per load injector. Then, it proceeds through 20 seconds iterations, observing the ESB's average load percentage, comparing it to a given threshold (80% here), and deciding a new number of virtual users: increase that number when the threshold is passed, decrease when it is unreached. We see that we actually implement a control feedback function, with all the associated issues in terms of stability and reactiveness. This control feedback is rendered by the load injection and saturation policy, provided as simple algorithmic rules here, but this may be easily replaced in the architecture by arbitrarily complex and advanced computations relying on the observations from the load injectors and probes. For instance, the iterations duration shall not be constant but computed at runtime. Of course, more probes would be necessary, in the general case, not only for the sake of saturation lookup, but also to go further towards our final goal of full characterization for system simulation and sizing.

B Results

The results presented below have been produced with 4 load injectors and a controller distributed on 5 distinct computers (Intel bi-Xeon or AMD bi-Opteron, 2 or 3 GB RAM, Gb/s Ethernet, Linux kernel 2.6.15-1-686-smp). The ESB load probe is hosted on a 6th computer and simply gets information from the ESB platform's SNMP agent. The observation (see figure 7) shows promising results, particularly because this ESB platform had already been "manually" benchmarked with CLIF's common user interface on the same infrastructure, giving similar results. After 3-4 minutes, we see a rather quick and good stabilization of the number of virtual users around 400 and an ESB load around 80%. As expected, the request throughput is roughly following the number of virtual users (just a little smaller), with some sudden drops at time 270s and 390s, that can be explained by the occurrence of garbage collector on the load injectors. To be more accurate about this phenomenon, we should add CLIF's probes on the load injectors, and especially the JVM probe which detects occurrences of garbage collection. Garbage collection is the typical kind of phenomenon that must be taken into account to prevent unstability problems.

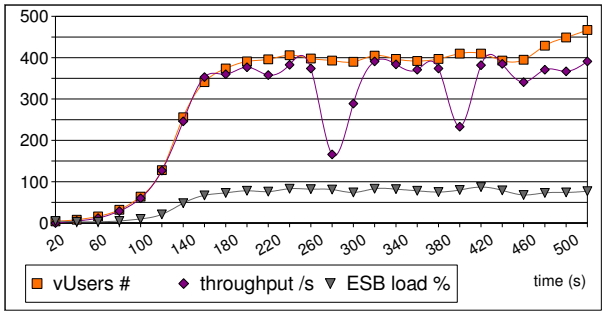

Figure 7. Automatic saturation of an ESB platform

VII. EXTENSIONS

This work is being applied to current R&D projects in France Telecom, where the characterization of black boxes performance and resource consumption is key to develop and to keep good working conditions for many infrastructures. Sizing and capacity planning are essential.

For example, in the case of Machine to Machine (M2M) services, a large number of machines (teller machines, detectors, cameras, boilers, etc.) exchange events and a variety of data. The M2M middleware also controls its own execution by observing resources usage. Such infrastructures are

typically overlay networks, that are widely distributed, generate huge amount of events and connect a great number of devices together. Breakdowns may be frequent in such systems, and the manual supervision and management of such big infrastructures is almost impossible. Here, autonomic computing (see section A) research becomes fundamental to support self-optimization, self-healing or self-configuration features.

An M2M overlay network is basically a set of nodes, performing arbitrary computations that produce events, connected together through an arbitrary network topology. The nodes typically perform arbitrary business computations that are unknown to the network operator. As a consequence, the queuing model and our approach applies quite well to the global M2M system, where nodes are black boxes. The nodes must be tested one by one with our self-regulated load injection platform in order to produce the necessary performance characterization. Then, we will be able to simulate the global system and provide M2M systems with support for sizing and capacity planning. Moreover, in the context of autonomic computing, it will be possible to evaluate self-reconfiguration decisions through simulation, before actually performing them, to prevent unexpected performance defects.

VIII. RELATED WORK

Two previous works in black boxes modelling could be used as references in this work. The first deals with black boxes modelling in a particular context which is storage environment and the second proposes a method to determine relevant and necessary parameters to estimate a performance model of black boxes.

In [9], the authors evaluate the most popular techniques used in black box modelling in storage environment and measure the precision of each technique to obtain the best of them. [6] tries to determine necessary properties to estimate performance model for black box when it is used in a feedback loop.

These papers use an analytical model by considering the system as a one black box unlike our method which decomposes the system in several black boxes and hence gives a more detailed model. If we have to do a factorial analysis to determine relevant parameters, we can use results of the second article which proved that the method of least squares does not give the best estimation any more when a control loop is used. Furthermore, a regression method can not be used since we are looking for performance before saturation which means we are not in the linear range.

IX. CONCLUSIONS AND FUTURE WORK

In this paper, we addressed the general issue of sizing and capacity planning of distributed systems, by proposing a combination of global system modelling and real testing of small, unknown elements (black boxes). The proposed methodology consists in characterizing the performance and computing resource consumption of the black boxes, by generating a variable workload on them and observing their behaviour, and to use these results as an input in the global system model. Then, this model will be used to predict the adequate sizing of the execution support as well as the expected performance. To achieve this prediction, we chose a queuing network model and a simulation-based approach.

We also presented a component-based software architecture to support the autonomous characterization of black boxes. Springing from architectural research for autonomic computing infrastructures, it relies on a load injection framework with a feedback control loop. We partly implemented end experimented this architecture in a real test case with an Enterprise Service Bus. The promising first results still require more research work in several directions, such as: identifying the black boxes, factorial analysis, saturation and stability criteria, control theory, and of course simulation to achieve our ultimate goal in terms of sizing and capacity planning. Our future work will be guided by this goal, in the context of Machine-to-Machine applications and related middleware.

References

[1] E. Bruneton and al. The fractal component model and its support in java. *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11-12), 2006.

[2] S. F. R. collaborative project. Déploiement, configuration et administration autonome de systmes rpartis. http://www.rntl.org/projet/resume2005/selfware.htm, 2005.

[3] T. Coupaye, F. Horn, and al. Principes généraux darchitecture logicielle pour la construction dapplications autonomiques ouvertes. *L'autonomie dans les réseaux (Traité IC2 série Réseaux et télécommunications)*, pages 1–34, September 2006.

[4] B. Dillenseger. Flexible, easy and powerful load injection with clif version 1.1. In *Fifth Annual ObjectWeb Conference*, http://objectwebcon06.objectweb.org/xwiki/bin/Main/DetailedSession, January 2006. ObjectWeb.

[5] IBM. An architectural blueprint for autonomic computing. http://www-03.ibm.com/autonomic/pdfs/AC Blueprint White Paper V7.pdf, June 2005.

[6] M. Karlsson and M. Covell. Dynamic black-box performance model estimation for self tuning regulators.

[7] D. A. Menasc and V. A. F. Almeida. *capacity planning for web services: Metrics, Models and Methods*. 2002.

[8] P. V. Roy and al. Self management of large-scale distributed systems by combining peer-to-peer networks and components. Technical Report TR-0018, CoreGRID, December 2005.

[9] L. Yin, S. Uttamchandani, and R. Katz. An empirical of black-box performance models for storage systems. *14th IEEE International Symposium on Modelling, Analysis and simulation of computer and Tlcommunication Systems (MASCOTS '06)*, October 2006.