Computer Science • 18 (1) 2017 http://dx.doi.org/10.7494/csci.2017.18.1.21

Wojciech Rząsa

PREDICTING PERFORMANCE IN A PaaS ENVIRONMENT: A CASE STUDY FOR A WEB APPLICATION

Abstract *This paper demonstrates how the combination of simulation and real-world experiments can be used to aid decisions concerning the performance of a distributed application. It presents a case study of performance analysis carried out for a commercial application implementing a web-based API server for mobile clients. The application was deployed on the Heroku cloud-based Platform as a Service (PaaS). The analysis described in this paper provided information required to choose the proper configuration of resources for the software. Simulation was used in the research to identify factors crucial to the performance of the application. This allowed for the preparation of basic experiments concentrating on these factors. Consequently, the basic parameters of resources crucial for the efficiency of the application could be benchmarked at insignificant cost and effort. This approach allows us to reliably aid decisions concerning resource configuration for an analyzed application. The simulation method used in this research is based on the formalism of Timed Colored Petri Nets, but the complexity of formal modeling is hidden from its users. Application developers are able to conveniently create a high-level model of their designs and perform simulations, while the reliability of the results is ensured by the formalism. The paper demonstrates the usefulness of the simulation method for analyzing real-world distributed systems.*

Keywords performance, simulation, distributed application, web, Petri nets, TCPN

Citation Computer Science 18 (1) 2017: 21–39

1. Introduction

In recent years, solutions based on distributed systems have become pervasive in computer science. They are used for more than scientific applications involving parallel high-performance computing and complex business systems. Nowadays, even relatively simple applications have a distributed nature due to the micro-service development model[1], mobile clients cooperating via centralized servers, and using other different distributed resources frequently based on cloud solutions (for example). In such an environment, the issue of distributed system performance becomes even more important, and it still remains a complex problem. In the case of business applications, sustaining satisfactory efficiency is necessary to sustain revenue; therefore, tools assisting application developers and maintainers in assessing the efficiency of distributed applications and allowing them to observe and analyze performance-related factors seems necessary.

This paper presents a case study of performance analysis carried out for a specific commercial web application deployed on a cloud-based infrastructure. It demonstrates the use of a simulation method in combination with simple benchmarks to reliably and conveniently predict the performance of the application (depending on the configuration of resources). The goal of the analysis was to aid in deciding which infrastructure should be used to run the software. The application – a web-based API server for mobile clients – was deployed for production use on a commercial Platform as a Service (PaaS) provided by Heroku[2] on a set of lightweight containers. The decision to be made concerned switching the application to another set of containers that were supposed to render the whole system more efficient.

The decision could be made on the basis of benchmarks run for the complete application; this was, however, a complex and expensive approach. Instead, simulations were used to identify the most crucial aspects that impact performance. Thereafter, simple benchmarks were carried out to only determine the values for important factors. Consequently, thanks to the simulation method, it was possible to reliably support decision which solution was best for performance of the real production application. It was feasible in short time and with insignificant cost.

The simulation method used in this research is meant for application developers and designed to be reliable and convenient for them. The reliability of the analysis is provided by the formalism of Timed Colored Petri Nets (TCPN) [9], which is the basis of the simulation. The formalism is, however, hidden from the developer who performs the analysis – the model provided by the developer is automatically translated to the formalism and simulated without user assistance. Consequently, a model of the analyzed system is created on a high level of abstraction, using notions of *nets*, computing *nodes*, *programs*, and *processes* that are natural for developers. The method allows us to describe important parameters of our resources. Crucial elements

[1] `http://martinfowler.com/articles/microservices.html`
[2] `http://heroku.com`

of application logic (e.g., CPU load generated by process, volume of communication, and the dependencies between them) can be easily and concisely described by means of event-driven programming using tools that are well-suited. Therefore, creating the model in this method – the necessary part of the analysis process – is facilitated and limited to reflecting the most important parts of the system in the most convenient ways. Moreover, parts of the models can be reused between subsequent works; thus, the analysis of similar systems becomes easier with time.

The paper is organized in the following manner. Section 2 describes related work. Section 3 briefly presents the architecture of the Heroku Platform as a Service (since it is crucial to understand the analyzed problem). Section 4 describes the actual problem of selecting the proper configuration of resources. Section 5 presents the model used to perform the analysis. Section 6 is a brief overview of the simulation method used in this research. Section 7 describes the simulations and experiments with results and conclusions concerning the analyzed application. Section 8 presents conclusions from the research.

2. Related work

Performance analysis of distributed systems is not a new problem. Usually, in scientific works, one of the formal methods is used, with Petri nets (PN) [14] being a frequently selected and convenient choice. Numerous flavors of this formalism (Real-Time Colored Petri nets [25] or Timed Colored Petri Nets [9], for example) allow for the convenient analysis of specific classes of problems or large systems that are hard to model with classical Petri nets. PNs can be used to perform formal analyses of system properties, with each PN extension having its own approaches [8, 9, 11, 25]. Petri net models are useful for more than formal analysis. A PN model is also suitable for a simulation that allows us to observe the behavior of modeled aspects of a system and also draw important conclusions concerning performance.

Petri nets with different extensions have been successfully used to analyze different types of systems, with web applications being one of them [15]. However, the analysis is frequently based on models created by a scientist using formalism and its primitives. Therefore, to perform such an analysis, it is necessary to be not only familiar but rather experienced with formalism. Consequently, the methods (however reliable) are not encouraging for application developers who usually have neither the time nor abilities to learn and use such methods in order to perform a kind of complex scientific work instead of application development.

Higher-level approaches to performance-related modeling were developed with grid-related projects [6], especially when a real grid infrastructure was not yet available. Most of these solutions, however, were designed to analyze algorithms meant for managing grid infrastructure [5, 16, 24, 26]. The few tools aimed at application analysis are either strongly connected with a specific programming model (e.g., MPI [3]) or seem not to be actively developed [1, 12]. Consequently, there is a lack of

a developer-centric approach that would enable a performance analysis of distributed systems.

Performance of cloud computing is an actively researched topic. Usually, it is analyzed with benchmarks. Garg et. al. [7] and Atas et. al. [2] describe a holistic approach to benchmarking platforms, with performance being one of the considered attributes. The main goal of these works is to provide a reliable solution to compare different platforms and select the one that fits a user's needs. Heroku used in this case study is one of the platforms analyzed in [2]. The cloud platforms are also assessed in terms of their usefulness for specific kinds of applications; e.g., for science [13, 27] or for multi-tenant web solutions [10]. There are also works aimed at measuring the performance of specific services depending on the configuration of specific cloud platforms [23]. Existing benchmarks are evaluated by considering their suitability for clouds and are extended to cover all aspects of the new environment [4, 10].

Compared to the mentioned research, the case study presented in this paper focused on two important aspects. First, the solution was specifically designed for application developers and maintainers, allowing them to concentrate on the aspects of their applications that are considered crucial and to aid them in assessing performance under different circumstances. Second, the goal was to minimize the need for performing benchmarks in order to simplify the solution while also minimizing costs and effort. As discussed further, performing benchmarks to test specific aspects of an application is costly and requires significant work. The case study carried out for a web application and described in this paper showed that the required information could be obtained faster and with less expense.

Different aspects of the simulation method used in this research [18] was presented in previous publications. The architecture and reliability of the method (based on TCPN) was discussed and tested in practice [20, 21]. The possibility of the more-precise modeling of TCP transmissions was verified [17], and a first approach to the high-level modeling of applications was published [22]. Currently, the usefulness of the method should be demonstrated not only for laboratory experiments but also real-world applications.

3. Heroku architecture

Heroku is a Platform as a Service designed for web applications. It enables the convenient deployment of applications implemented in most contemporary technologies. User applications are running inside lightweight containers called *dynos*. A single application can use one or more *dynos* of different types. The *dyno* type determines its computing capability, available RAM, and price. Heroku offers different types of *dynos*, from basic free configurations to advanced ones running on dedicated hosts. In case of changing requirements, applications can be scaled by changes in the number of exploited *dynos* or their types.

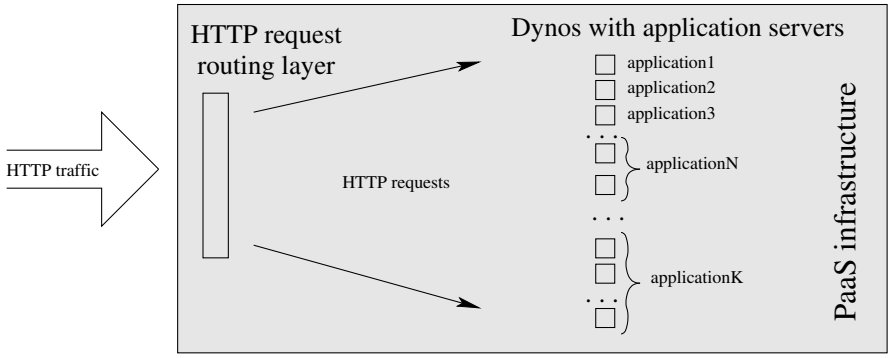

Figure 1. Simplified architecture of Heroku PaaS.

As stated in documentation[3], the HTTP requests sent to Heroku-hosted applications are received by a load-balancer, which forwards them to a set of *routers* called *routing mesh*. This forms an HTTP routing layer that is responsible for receiving requests and forwarding them to the proper *dynos* (Fig. 1). Obviously, the first criterion of selecting the correct *dyno* is to ensure that it belongs to the application for which the request is meant. However, if the application is served by more than one *dyno*, the HTTP router randomly selects the one that should serve the request. Certainly, it is not the solution that enables the most efficient use of resources [19].

4. Selecting proper dyno formation

The problem considered in this paper comes from the real need of aiding the selection of the correct *dyno formation* for a web application running on Heroku. The *dyno formation* defines the set of *dynos* used by an application – their number and types. Precisely, the problem consisted in making the proper decision about scaling *dyno* types and reducing their number for a specific application running in a production environment.

The software analyzed in this research was a business application custom-made for a commercial client. In one respect it was not a typical example of web software. It served as JSON API backend for clients running on mobile devices. Most of the client-server communication involved exchanging large chunks of JSON-encoded data and processing them, requiring significant CPU time. The computations performed for each request were significantly more complicated than usual in web applications. Consequently, CPU-intensive processing was a bottleneck in this case (unlike in typical web applications where database operations frequently cause performance issues).

[3] `https://devcenter.heroku.com/categories/heroku-architecture`

Table 1
Heroku dyno types (selected).

Dyno Type	Memory (RAM)	CPU Share	Compute	Price/dyno-month
standard-1x	512MB	1x	1x-4x	$25
standard-2x	1024MB	2x	4x-8x	$50

The application was implemented in Ruby on Rails[4] and deployed to a Unicorn[5] application server. There were several possible solutions when deploying a Ruby on Rails application, and they had different support for concurrent request processing. The Unicorn used for this application could manage multiple worker processes that concurrently served HTTP requests. An incoming request was either assigned to an idle worker or enqueued and served by the first available worker. Consequently, the time of the Unicorn workers was optimally exploited. The application was running in a production environment on resources provided by the Heroku platform.

Among the others, Heroku offered two kinds of *dynos*: *standard-1x* and *standard-2x* (described in Table 1). The *standard-2x dynos* had two times the RAM and were supposed to be twice as fast, meant to run double the application processes of the *standard-1x dynos*. They were also twice as expensive; thus, for the same price, one could buy twice the number of *standard-2x dynos* or half the number of *standard-1x dynos* and receive the same cumulative computing power.

The actual problem was making the following decision: *should the considered application be moved from the dyno formation consisting of six standard-1x dynos to the formation of three standard-2x dynos?* Obviously, experiments with the application deployed for production use were not desirable. Therefore, it should have been determined in advance not only whether the change would improve the efficiency of the system but also what gain in performance can be expected. An answer to the last question was necessary to assess if the improvement would justify the risks and costs connected with the changes.

The answer to such problem could be obtained by benchmarking the application on the two *dyno formations*. Properly carried out, such tests could give reliable answers to all questions concerning the performance of the application. There is wide variety of software and services that can be used to perform such benchmarks. For instance, the free JMeter[6] or Gatling[7] can be installed locally on one or more nodes. There are also cloud based solutions; e.g., BlazeMeter[8], Flood IO[9], or Blitz[10],

[4] `http://rubyonrails.org/`
[5] `http://unicorn.bogomips.org/`
[6] `http://jmeter.apache.org/`
[7] `http://gatling.io`
[8] `https://www.blazemeter.com`
[9] `http://www.flood.io`
[10] `https://www.blitz.io`

offering complete infrastructures that are able to generate the required load. While locally installed software can easily be used to perform basic benchmarks, the distributed solutions can stress-load applications from a number of different, geographically distributed locations. The use of such solutions is usually connected with some financial costs. This also requires one to spend some time configuring the test scenarios. This configuration can be done either with a GUI (as in JMeter) or using a scripting language (e.g., in Gatling). The software required to generate the workload is only the first step required to perform benchmarks. Testing also requires the separate deployment of the analyzed application, and this is also connected with the financial costs and additional work. Finally, to obtain measurements from a test configuration, monitoring software is needed. In the case of Heroku, software[11] can be easily integrated with an application, but it is saddled with further costs. Software and computer infrastructures required to perform benchmarks are available, and the financial costs can be limited by obtaining them for a short period of time. Providers of these services make an effort to facilitate the configuration of test scenarios. However, despite the conveniences and technical abilities, assembling all elements required for the benchmarks consumes noticeable time, effort, and financial costs.

The approach used in this research served to minimize effort as well as the expenses connected with analyzing a considered application. To achieve this, the first step of the analysis was based on modeling and simulation. This allowed identifying factors that were crucial to the performance of the application. Thereafter, basic experiments focused on these factors could be scheduled and performed with minimal cost and effort and provide the final results.

5. Model of web application and dyno formations

Analyzing the performance of the application required a model describing how the HTTP requests were routed inside the Heroku infrastructure and served by the application deployed on the considered *dyno formations*. The simulation method required the model to describe the resources used by the analyzed software and the basics of software functionality. Subsequent elements of the analyzed application had to be assigned to specific resources (nodes) and could communicate over the network, forming a topology corresponding to the modeled system.

5.1. Software components

In this case, the model should include the following software components:

- web client that generated HTTP requests for the application;
- heroku random HTTP router that was responsible for assigning incoming HTTP requests to a *dyno*;

[11] `http://newrelic.com`

- unicorn-like Ruby on Rails application server modeling the algorithm used by Unicorn to assign HTTP requests received by a *dyno* to a worker – web server responsible for processing the request;
- web server modeling a single worker of the application server that received HTTP traffic, processed the requests, and generated responses.

The simulation method enables us to create reusable parts of the application model. Therefore, significant parts of the model from previous work [19] could be reused in this research; to complete the components required for this analysis, only the Unicorn application server had to be modeled. Even this model was not created from scratch, since the model of *intelligent router* described in the previously mentioned paper was used as a basis for the Unicorn model. Consequently, the model of the application was completed in a short time and with insignificant effort.

The model of the Unicorn application server created for this research is presented in listing 1. This is an example of simple event-driven programming; thus, it enables us to concisely describe crucial aspects of the application activities.

The Unicorn router was modeled as a program that received a list of available HTTP workers (variable `servers`) as its parameter and maintained its request queue. It responded to two kinds of events. On the `:data_received` event, either a request from a client or response from an HTTP worker could be served. Incoming requests were put in a queue, and responses were sent to the appropriate clients. After a response was sent, the worker that served the corresponding request was added to the end of the list of idle servers. In both cases, a new `:process_request` event was registered to ensure that the router would try to serve a request from its queue. When the `:process_request` event occurred, the Unicorn router had pending requests in its queue and there were idle HTTP workers, then the request was forwarded to the first worker and the worker was removed from the list. Details concerning the parameters of the `send_data` command were removed from the listing 1 to improve clarity.

5.2. Dyno formation

The software components of the model were assigned to the *nodes* that were supposed to provide computing capabilities. CPU load caused by the web client and Heroku HTTP routers was not the scope of this research and was omitted in the model. Therefore, the configuration of the nodes hosting these components was not important. Similarly, the parameters of the networks connecting subsequent nodes were not crucial for the results, since the network was never identified as a bottleneck for the analyzed application.

A crucial part of the infrastructure model was the group of *nodes* modeling the *dyno formation* actually used by the application. The *dynos* were modeled as individual *nodes* equipped in the CPUs corresponding to the *dyno* type. The number of these *nodes* reflected the number of *dynos* in the formation. Each *node* was running a Unicorn program from listing 1 with a specified number of HTTP workers.

Listing 1. Model of Unicorn HTTP router.

```
1  program :unicorn_router do |servers|
2    servers = servers.clone
3    request_queue = []
4    on_event :data_received do |data|
5      if data.type == :request
6        request_queue << data
7        register_event :process_request
8      elsif data.type == :response
9        servers << data.src
10       send_data to: data.content[:from], # ... <- DATA DETAILS
11       register_event :process_request
12     else
13       raise "Unknown data type #{data.type} received."
14     end
15   end
16   on_event :process_request do
17     unless servers.empty? or request_queue.empty?
18       data = request_queue.shift
19       server = servers.shift
20       send_data to: server, # ... <- DATA DETAILS
21       register_event :process_request unless request_queue.empty?
22     end
23   end
24 end
```

A this point of the modeling process, a serious difficulty was encountered. Heroku documentation did not provide a definite answer to the question of CPU configuration available for the particular standard *dynos*. As quoted in Table 1 the *standard-1x dyno* had 1x *CPU Share* and value 1x-4x in the *Compute* column, while the *standard-2x* had 2x *CPU Share* and value 4x-8x in the *Compute* column. This, however, was not sufficient to determine the actual number of available CPU cores, since the twice-bigger *CPU Share* could be provided either by double the CPU cores or by CPUs that are twice as fast. The operating system of a *standard-1x* dyno reported access four CPU cores, but the implementation of operating system containers managing the *dynos* could easily limit access to some of these CPUs. At this point, we also didn't have access to the *standard-2x* dyno and, thus, could not determine Heroku's method of scaling from *standard-1x* to *standard-2x*.

The method of scaling *dyno* sizes could be determined by tests on two types of *dynos*. However, at this point, we decided to first use a simulation to verify whether the scaling method significantly impacted application efficiency. Certainly, the twice-as-fast CPUs could contribute to faster request processing. However, considering Unicorn's efficient usage of numerous workers serving HTTP requests, one could suppose that an average HTTP request could also be efficiently served on a configuration with an increased number of CPU cores instead of an increased core speed.

Listing 2. Model of a *dyno* with horizontal scaling – *standard-1x dyno* has CPUs with speed factor 1 and *standard-2x* with speed factor 2.

```
1  node dyno_name do
2    4.times { cpu size }
3  end
```

Listing 3. Model of a *dyno* with horizontal scaling – *standard-1x dyno* has 4 CPUs and *standard-2x* 8 CPUs.

```
1  node dyno_name do
2    (4 * size).times { cpu 1 }
3  end
```

Consequently, two models of *dyno* formation were prepared. The models were parametrized by a `size` variable, with value 1 for *standard-1x* and value 2 for *standard-2x dynos*. The first model assumed vertical scaling from *standard-1x* to *standard-2x dynos*. In this model, each node representing a *dyno* was equipped in four CPUs with the speed factor set to the value of the `size` variable (listing 2). The second variant of the model assumed horizontal scaling from *standard-1x* to *standard-2x* dynos. In this model, each node had $4 * size$ CPUs with the speed factor set to 1 (listing 3).

5.3. Runtime parameters of the application

An important challenge in modeling a distributed application is connected with the assessment of application runtime parameters. The topology of the application components and the most important aspects of data processing can be obtained from the application design or implementation. The actual time required to serve an HTTP request by an application process is, however, hard to obtain this way. Obviously, each web application copes with a diversity of requests that require different processing times. Thus, the problem consists of determining the correct distribution of request service times for the actual application.

For the analyzed application, the existence of production deployment solved the problem. The application running in the production environment used performance monitoring tools to provide the necessary information to its maintainers. Data coming from the monitoring tool was used to provide the required histogram of request service times presented in Figure 2 and the request rate was set to 1000 requests per minute. Thus, the model could be used to assess application behavior under conditions as similar to real as possible (but for the arbitrarily selected configuration of resources).

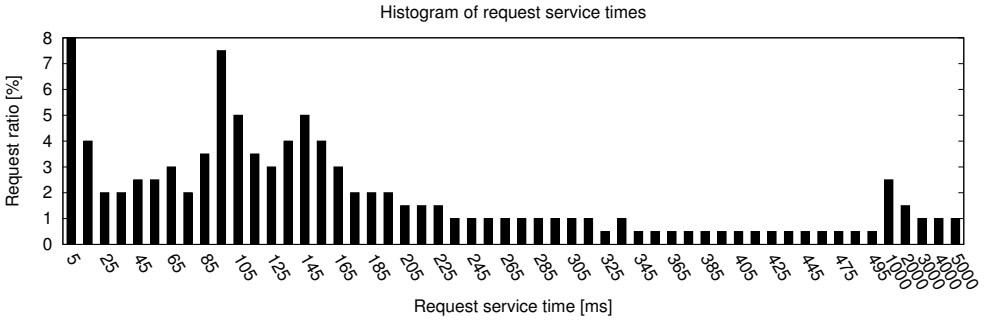

Figure 2. Histogram of HTTP request service times for the application obtained from monitoring of the production environment and provided as input for the simulation method.

6. Simulation method overview

Solutions applied to enable a simulation of the high-level model are complicated; a detailed description of the simulation method certainly exceeds the scope of this paper. Moreover, complete knowledge about the method is not required to comprehend the presented results and advantages of the approach. However, a brief description of the concepts was provided in order to facilitate understanding of the nature of the research described in this paper.

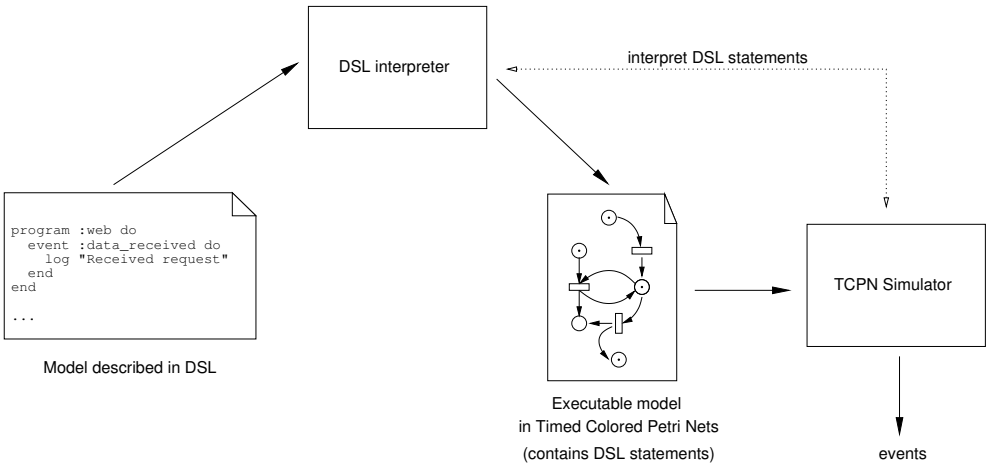

Figure 3. Concept of the simulation method.

The basic concepts of the method are illustrated in Figure 3. The DSL description of the high-level model presented in Section 5 is interpreted to generate an executable model expressed in the formalism of Timed Colored Petri Nets. The resulting TCPN

models the system that should be analyzed. Different parameters of the system in this model are still described with DSL statements provided in the high-level model. These statements are interpreted (evaluated) while the TCPN is being simulated. Thus, the values of specific parameters of the model can be determined on the basis of the model state that is changing during simulation. For instance, a block of code returning CPU processing time for a request from a remote process is evaluated only when the `:data_received` event for this request occurs and the parameters of the request are known.

The simulation produces stream of events corresponding to the activity of the modeled application (e.g., data reception, request processing, start of network transmission, obtaining CPU time). These events supplemented with additional information (e.g., occurrence time) can be interpreted to understand the behavior of the system and to compute performance indexes.

7. Combining simulations with experiments

7.1. Identifying performance factors

The main part of this research used simulations based on the model from Section 5 and run on a local workstation. The simulation enabled the observation of a large variety of events that occurred in the analyzed system. However, in order to reliably compare the results for different configurations, a collective efficiency index for the whole simulation was required. In this work, two values were used. The first was the average request service time computed for the whole experiment, and the second was Apdex[12], which better reflected a user's experience concerning efficiency. The value of $Apdex_t$ was computed using formula (1), where t is the request time threshold, S_t denotes the number of requests served in a time not longer than t (satisfactory service time), T_t is the number of requests with service time above t but not greater than $4t$ (tolerated service time), and F_t is the number of requests served in times exceeding $4t$ (frustrating service time). Apdex gives prominence to the requests served in a satisfactory time to the ones that cause impatience and (frequently) the resignation of a user. The value of threshold t should match the expected request service time for an application. In this case study, the use of Apdex was additionally justified by the fact that it was the main performance index used on Heroku. Consequently, it was the main indicator of efficiency observed for the production deployment of the analyzed application. Additionally, we observed the maximal queue length of the Unicorn application servers to assess its connection with efficiency of the whole application reported by the other indexes.

$$Apdex_t = \frac{S_t + \frac{T_t}{2}}{S_t + T_t + F_t} \tag{1}$$

The first simulations were performed with the assumption that *standard-1x dynos* were scaled to *standard-2x* vertically; thus, the model from listing 2 was used.

[12] `http://apdex.org`

Both performance indexes for this case showed that, if the *standard-2x dynos* were equipped in CPUs that could serve every request twice faster than the *standard-1x*, the performance of the whole application would significantly benefit from switching to the *standard-2x*-based *dyno* formation. It could be concluded from the Apdex values (computed for threshold time 0.3 s) presented in Figure 4 that, for the *standard-2x dynos*, they were as close to 1 as possible for this setup, while for the *standard-1x*-based *dyno* formation, it settled below 0.8. Values of the average request service time (Fig. 5) confirmed this conclusion. The average times for *standard-2x*-based *dyno* formation were twice as short. We could also notice that, for both *dyno* formations, adding more than 3-4 workers per *dyno* did not improve the performance of the application. This value corresponded to the number of CPUs assigned to each *dyno* in the simulations. All simulations were repeated 30 times, and the presented results are the average values of all executions.

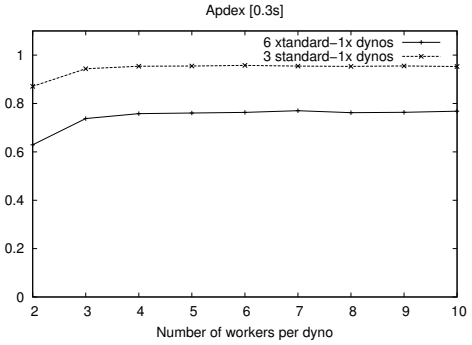

Figure 4. $\text{Apdex}_{0.3s}$ for vertically scaled *dynos*.

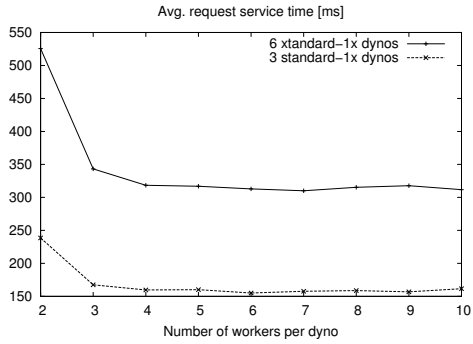

Figure 5. Average request service time for vertically scaled *dynos*.

The second set of simulations was run for the assumption of horizontal scaling. The *standard-2x dynos* were equipped in twice as many CPUs than the *standard-1x* ones, as in listing 3. From the results of this simulation presented in Figure 6 and 7, it could be noticed that no performance gain could be achieved by switching from a *standard-1x*-based to *standard-2x*-based *dyno* formation in this case. For such a small formation, the more-efficient load balancing offered by Unicorn for workers on particular *dynos* did not have a significant impact on performance (despite the inefficiency of Heroku's random routing). For the reasonable number of 4–5 workers per *dyno*, Apdex as well as the average request service time showed the same values for both *dyno* formations.

Besides the performance indexes, we also observed the values of the maximum queue length for the Unicorn application servers (Fig. 8 and 9). It appeared, however, that there was no performance gain for more than 3–4 workers per *dyno*, but 6–7 workers per *dyno* were required in order to ensure that requests were not queued at this level. This suggested two conclusions. First, it confirmed that the application in

these experiments was significantly loaded; thus, the previous results were not skewed by the fact that, at some point, the infrastructure was able to serve significantly more requests then it received. Second, since there was no performance gain for more than 4 *dynos*, the only advantage of having more workers was that the HTTP requests were dequeued from the Unicorn queue. Their processing was, however, immediately stopped at the operating system level, since there were not enough CPUs to serve them in parallel processes. Thus, it would be useless to derive performance conclusions concerning the application only on the basis of Unicorn queue length and aiming at configurations ensuring empty queues.

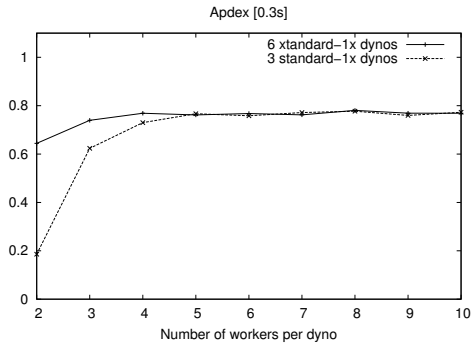

Figure 6. $\text{Apdex}_{0.3s}$ for horizontally scaled *dynos*.

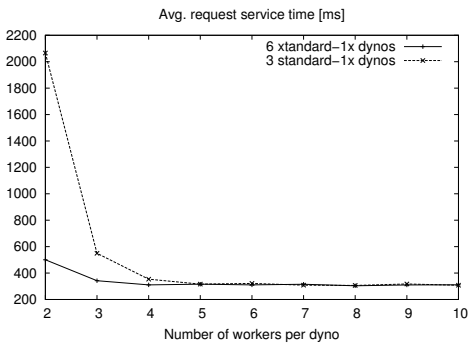

Figure 7. Average request service time for horizontally scaled *dynos*.

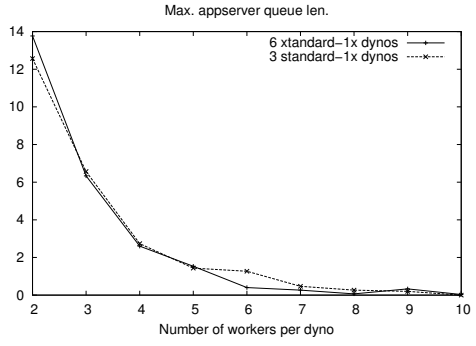

Figure 8. Maximum Unicorn queue length for vertically scaled *dynos*.

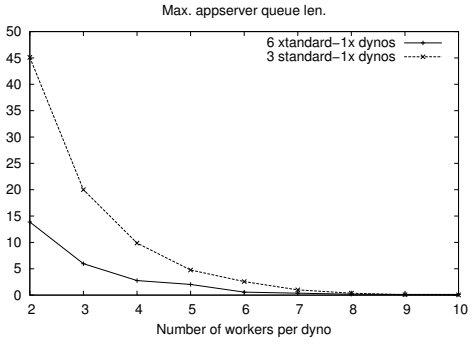

Figure 9. Maximum Unicorn queue length for horizontally scaled *dynos*.

7.2. Determining actual infrastructure parameters

An important conclusion at this stage of the research was that the unknown method of *dyno* scaling was crucial for the efficiency of the application. In order to make the correct decision about which *dyno* formation would be best for the analyzed system, it

was necessary to verify which method of scaling was chosen by Heroku. As it could not be found in Heroku's documentation, we performed basic experiments. One instance of each of the two types of *dynos* (*standard-1x* and *standard-2x*) were purchased. On each of them, two kinds of experiments were performed using a simple CPU-intensive task from listing 4. Real times and CPU times were measured and compared.

Running basic tests on Heroku *dynos* was not technically complicated. The source code of the required programs and shell scripts was committed to a git repository and pushed to the *dyno* filesystem. Thereafter using the Heroku client, proper commands were started. Time was measured either using the value returned by the function from listing 4 or using the operating system `time` command.

Listing 4. CPU-intensive task used to test *dyno* scaling method.

```
1  def cpu_intensive_task(n)
2    start = Time.now
3    (1..n).reduce(:*)
4    Time.now - start
5  end
```

First, a single CPU intensive task (listing 4) was run on each type of *dyno*. Run times for both *dyno* types were comparable, suggesting that the *dynos* were not scaled vertically. Thus, a set of 16 tasks was run on each *dyno* type, and the total time was measured. The time for *standard-2x dyno* was about twice as short; therefore, it could be concluded that the *dynos* were scaled horizontally – the *standard-2x* had twice as many CPUs than the *standard-1x*.

Another useful conclusion that was derived from these simple tests came from observations of real-time and user/system time reported by the Unix `time` command. The experiments showed that the real time for the *standard-1x* was about one half of the user and system time, while for the *standard-2x*, it was about one quarter of the user and system time. Thus, it could be concluded that, unlike in our assumptions, the *standard-1x dynos* were equipped in two CPUs and the *standard-2x* in four CPUs.

Results of the experiment showed that the *dynos* were scaled horizontally from the *standard-1x* to *standard-2x* types. Therefore, on the basis of simulations for this case, it could be concluded that *switching an application from six* standard-1x *to three* standard-2x dynos *would not result in a performance gain and, thus, was not justified.*

7.3. Summary of the results

The first part of the research based on the simulation allowed us to verify how the performance of the analyzed system depended on the *dyno* formation. It was determined that, to make the proper decision concerning the choice of *dyno* formation, it was crucial to know how the *dynos* were scaled from *standard-1x* to *standard-2x*. Additionally, the HTTP request queue length of the application servers was analyzed.

Conclusions from the simulations were used to decide which experiments should be carried out to verify crucial aspects of application performance. Consequently, it was possible to plan very basic tests that consumed insignificant time and generated minor costs. All actual experiments were prepared and carried out in a few hours and cost about $0.09.

8. Conclusions

This paper presented a performance analysis case study for a web application deployed in the Heroku PaaS environment. It was demonstrated how the simulation method described in [18] can be connected with real-world experiments to aid decisions concerning the performance of distributed applications. The modeling process required to carry out simulations was facilitated by reusing fragments of previously defined models from previous research [19] and, thus, did not require significant effort. Similarly, simulations of the analyzed case could be quickly performed on a PC.

Results of simulations allowed us to develop the most efficient experiments to determine factors crucial for the performance of an application. Consequently, the experiments could also be performed without unnecessary costs and effort. The combination of modeling, simulation, and real-world experiments aided one's decision concerning the proper choice of *dyno* formation for a distributed application.

The simulation method used in this research proved to be a convenient instrument. It allowed us not only to save time and effort that would be necessary to perform benchmarks of the real application, but also to reduce the costs of the required real-world experiments.

Nowadays, while even comparably simple applications are frequently deployed to a PaaS based on a cloud solutions and, thus, based on distributed infrastructure, the distributed nature of computer systems has become pervasive. Especially, that currently deployed applications frequently integrate third-party distributed services to easily provide their functionality to the clients. Since the performance of applications is (and most probably will remain) a crucial issue, it seems that there is a strong need to provide developers with a method that allows them to easily identify performance issues in their products without complex scientific work but by using primitives natural for them. The usefulness of such a method was demonstrated in this paper on the example of a real-world application.

References

[1] Adve V., Bagrodia R., Browne J., Deelman E., Dube A., Houstis E., Rice J., Sakellariou R., Sundaram-Stukel D., Teller P., Vernon M.: POEMS: end-to-end performance design of large parallel adaptive computational systems. *Software Engineering, IEEE Transactions on*, vol. 26(11), pp. 1027–1048, 2000.

[2] Ataş G., Gungor V.C.: Performance evaluation of cloud computing platforms using statistical methods. *Computers & Electrical Engineering*, vol. 40(5),

pp. 1636–1649, 2014, http://www.sciencedirect.com/science/article/pii/S0045790614000718.

[3] Badia R.M., Escalé F., Gabriel E., Gimenez J., Keller R., Labarta J., Müller M.S.: Performance Prediction in a Grid Environment. In: F. Fernández Rivera, M. Bubak, A. Gómez Tato, R. Doallo, eds., *Grid Computing*, *Lecture Notes in Computer Science*, vol. 2970, pp. 257–264, Springer, Berlin, Heidelberg, 2004, http://dx.doi.org/10.1007/978-3-540-24689-3_32.

[4] Binnig C., Kossmann D., Kraska T., Loesing S.: How is the Weather Tomorrow?: Towards a Benchmark for the Cloud. In: *Proceedings of the Second International Workshop on Testing Database Systems*, DBTest '09, pp. 9:1–9:6, ACM, New York, NY, USA, 2009, http://doi.acm.org/10.1145/1594156.1594168.

[5] Buyya R., Murshed M.: GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *Concurrency and Computation: Practice and Experience (CCPE)*, vol. 14(13), pp. 1175–1220, 2002.

[6] Foster I., Kesselman C., Tuecke S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, vol. 15(3), pp. 200–222, 2001, http://dx.doi.org/10.1177/109434200101500302.

[7] Garg S.K., Versteeg S., Buyya R.: A framework for ranking of cloud computing services. *Future Generation Computer Systems*, vol. 29(4), pp. 1012–1023, 2013, http://www.sciencedirect.com/science/article/pii/S0167739X12001422, special Section: Utility and Cloud Computing.

[8] Gniewek L.: Coverability Graph of Fuzzy Interpreted Petri Net. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 2014.

[9] Jensen K., Kristensen L.: *Coloured Petri Nets. Modeling and Validation of Concurrent Systems*. Springer, Berlin, Heidelberg, 2009.

[10] Krebs R., Wert A., Kounev S.: *Web Engineering: 13th International Conference, ICWE 2013, Aalborg, Denmark, July 8–12, 2013. Proceedings*, chap. Multi-tenancy Performance Benchmark for Web Application Platforms, pp. 424–438. Springer, Berlin, Heidelberg, 2013, http://dx.doi.org/10.1007/978-3-642-39200-9_36.

[11] Murata T.: Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, vol. 77(4), pp. 541–580, 1989.

[12] Nadeem F., Yousaf M., Ali M.: Grid Performance Prediction: Requirements, Framework, and Models. In: *Emerging Technologies, 2006. ICET '06. International Conference on*, pp. 695–702, 2006.

[13] Ostermann S., Iosup A., Yigitbasi N., Prodan R., Fahringer T., Epema D.: *Cloud Computing: First International Conference, CloudComp 2009 Munich, Germany, October 19–21, 2009 Revised Selected Papers*, chap. A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing, pp. 115–131. Springer, Berlin, Heidelberg, 2010, http://dx.doi.org/10.1007/978-3-642-12636-9_9.

[14] Petri C.: *Kommunikation mit Automaten*. Ph.D. thesis, Darmstadt University of Technology, Germany, 1962.

[15] Rak T., Samolej S.: Distributed Internet Systems Modeling Using TCPNs. In: *International Multiconference on Computer Science and Information Technology*, pp. 559–566, 2008, http://dx.doi.org/10.1109/IMCSIT.2008.4747298.

[16] Ranganathan K., Foster I.: Decoupling computation and data scheduling in distributed data-intensive applications. In: *Proceedings 11th IEEE International Symposium on High Performance Distributed Computing*, pp. 352–358, IEEE Computer Society, 2002, http://dx.doi.org/10.1109/hpdc.2002.1029935.

[17] Rząsa W.: Combining Timed Colored Petri Nets and Real TCP Implementation to Reliably Simulate Distributed Applications. In: A. Kwiecień, P. Gaj, P. Stera, eds., *Computer Networks*, *Communications in Computer and Information Science*, vol. 39, pp. 79–86, Springer, Berlin, Heidelberg, 2009, http://dx.doi.org/10.1007/978-3-642-02671-3_9.

[18] Rząsa W.: *Timed Colored Petri Net Based Estimation of Efficiency of the Grid Applications*. Ph.D. thesis, AGH University of Science and Technology, Kraków, Poland, 2011.

[19] Rząsa W.: Simulation-Based Analysis of a Platform as a Service Infrastructure Performance from a User Perspective. In: P. Gaj, A. Kwiecień, P. Stera, eds., *Computer Networks*, *Communications in Computer and Information Science*, vol. 522, pp. 182–192, Springer International Publishing, 2015, http://dx.doi.org/10.1007/978-3-319-19419-6_17.

[20] Rząsa W., Bubak M.: Application of Petri Nets to Evaluation of Grid Applications Efficiency. In: *Proceedings of CGW'10*, pp. 194–201, 2011.

[21] Rząsa W., Bubak M.: Simulation Method Supporting Development of Parallel Applications for Grids. In: *Proceedings of CGW'10*, pp. 194–201, 2011.

[22] Rząsa W., Bubak M., Nawarecki E.: High-Level Model for Performance Evaluation of Distributed Applications. In: J. Balicki, H. Krawczyk, E. Nawarecki, eds., *Grid and Volunteer Computing*, pp. 7–23, Gdańsk University of Technology Faculty of Electronics, Telecomunication and Informatics Press, Gdańsk, 2012.

[23] Stantchev V.: Performance Evaluation of Cloud Computing Offerings. In: *Advanced Engineering Computing and Applications in Sciences, 2009. ADVCOMP '09. Third International Conference on*, pp. 187–192, 2009.

[24] Sulistio A., Cibej U., Venugopal S., Robic B., Buyya R.: A Toolkit for Modelling and Simulating Data Grids: An Extension to GridSim. *Concurrency and Computation: Practice and Experience*, vol. 20(13), pp. 1591–1609, 2008, http://dx.doi.org/10.1002/cpe.v20:13.

[25] Szpyrka M.: Analysis of RTCP-nets with Reachability Graphs. *Fundamenta Informaticae*, vol. 74(2), pp. 375–390, 2006.

[26] Takefusa A., Tatebe O., Matsuoka S., Morita Y.: Performance Analysis of Scheduling and Replication Algorithms on Grid Datafarm Architecture for High-Energy Physics Applications. In: *12th International Symposium on High-Performance*

Distributed Computing (HPDC-12 2003), 22–24 June 2003, Seattle, WA, USA, pp. 34–47, 2003, http://dx.doi.org/10.1109/HPDC.2003.1210014.

[27] Tudoran R., Costan A., Antoniu G., Bougé L.: A Performance Evaluation of Azure and Nimbus Clouds for Scientific Applications. In: *Proceedings of the 2Nd International Workshop on Cloud Computing Platforms*, CloudCP '12, pp. 4:1–4:6, ACM, New York, NY, USA, 2012, http://doi.acm.org/10.1145/2168697.2168701.

Affiliations

Wojciech Rząsa
Rzeszow University of Technology, al. Powstańców Warszawy 12, 35-959 Rzeszów, Poland, http://prz.edu.pl, e-mail: wrzasa@prz.edu.pl

Received: 16.11.2015
Revised: 6.03.2016
Accepted: 13.04.2016