Applied Soft Computing 24 (2014) 1095–1104

Contents lists available at ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

Monitoring applications: An immune inspired algorithm for software-fault detection

Rui Ligeiro[a,b]

[a] INOV INESC – Instituto de Novas Tecnologias, Rua Alves Redol 9, 1000-029 Lisboa, Portugal
[b] CMAF – Instituto Investigação Interdisciplinar, Univ. Lisboa, Av. Gama Pinto 2, 1649-003 Lisboa, Portugal

ARTICLE INFO

Article history:
Received 24 February 2013
Received in revised form 11 May 2014
Accepted 11 August 2014
Available online 1 September 2014

Keywords:
Artificial immune system
Fault detection
Fault injection
Reinforcement learning
Monitoring
Metrics

ABSTRACT

Large-scale software systems are in general difficult to manage and monitor. In many cases, these systems display unexpected behavior, especially after being updated or when changes occur in their environment (operating system upgrades or hardware migrations, to name a few). Therefore, to handle a changing environment, it is desirable to base fault detection and performance monitoring on self-adaptive techniques.

Several studies have been carried out in the past which, inspired on the immune system, aim at solving complex technological problems. Among them, anomaly detection, pattern recognition, system security and data mining are problems that have been addressed in this framework.

There are similarities between the software fault detection problem and the identification of the pathogens that are found in natural immune systems. Being inspired by vaccination and negative and clonal selection observed in these systems, we developed an effective self-adaptive model to monitor software applications analyzing the metrics of system resources.

1. Introduction

Large-scale software systems are difficult to manage and monitor. These systems may display unexpected behavior, especially after being updated or when changes occur in their environment (operating system upgrades or hardware migrations, to name a few). Novel situations require robust defensive mechanisms, but monitoring can be rather time-consuming, requiring many efforts and tools [1–6] for the purpose.

A failure or malfunction occurs when the system behavior deviates from its initial specification, this being usually associated to the detection of a system error. In case of a failure in an application component, it must be detected quickly and, preferably, the overall system should be kept working, even with limitations. CPU usage, memory usage, load average and thread count are, among others, useful resource indicators of the efficiency and performance of a system. These metrics are highly correlated with the characteristics of the host where the software applications execute and, in general, when something wrong happens, some of the metrics reach values outside their usual ranges. Thus, the creation of an adaptive monitoring mechanism that ensures system fault detections based on resource metrics is a natural choice.

Despite all recent biological findings, lots of uncertainty still exists regarding how Nature works. However, in these last decades, biological systems have been used as a source of inspiration to solve complex technological problems, going far beyond the earlier boundaries of computer science. For example, analogies between the defending mechanisms of the immune system and anomaly detection in computer systems have been largely studied since 1994, after publications by Forrest et al. [8] and Kephart [9]. As a matter of fact, the vertebrate immune system has been the object of study by several authors [11–18], and as here, we give especial relevance to the insights most pertinent to the monitoring model, particularly the self/non-self discrimination, vaccination and some specific aspects of the adaptive immune response. The vertebrate immune system it is a complex system composed of a large collection of cells with several defense mechanisms that protect the body against diseases by recognizing, attacking and destroying pathogens. The system is divided into two inter-related branches: the *innate immune system* and the *adaptive immune system*. Roughly speaking, the innate immune system acts very quickly to the first signs of infection, being crucial to the initial inflammatory response by recognizing and signaling the adaptive immune response, whereas the adaptive immune system has the ability to change, improving the immune response during the lifetime of the organism. Note that the learning, memory and adaptation capabilities of the adaptive immune system emerge without any central control.

Lymphocytes, a special type of white blood cell with the function to recognize "non-self" antigens, are the most important agents of the adaptive immune system. B-cells and T-cells are the two main types of lymphocytes that together recognize and kill antigens. While B-cells produce and release large amounts of antibodies that attack pathogens, T-cells orchestrate the response of other cells as well as directly induce the death of cells that show signs of having been invaded by pathogens. The lymphocytes surface is covered with receptors that identify antigens by partial matching its shape. Consider, for instance, receptors and antigens as two pieces of LEGO, that even if they don't exactly join together, there are some complementary parts between each other. *Affinity* is the term used for the degree of recognition of antigens by lymphocyte receptors – stronger recognition corresponds to higher affinity and vice versa.

T-cells mature in the thymus gland, an organ located in the upper region of the chest to which T-cells travel after being created by the bone marrow in immature form. In this organ a process takes place called *negative selection*, responsible for eliminating T-cells capable of attacking the body's own cells. Nevertheless, this discrimination (*self/non-self*) can fail, resulting in the development of autoimmune diseases.

E-mail addresses: rui.ligeiro@inov.pt, rmligeiro@fc.ul.pt, rui.ligeiro@gmail.com

http://dx.doi.org/10.1016/j.asoc.2014.08.021
1568-4946/© 2014 Elsevier B.V. All rights reserved.

The reproduction of lymphocytes is based in a principle known as *clonal selection*. Once activated, B-cells produce and segregate antibodies, proliferating in a quantity proportional to the degree of affinity with the antigen. B-cells are also stimulated by T-cells (T-helper cells) to divide into offspring cells, which are very similar to their parent. This preferential proliferation of the most capable cells has clear similarities with Darwin's evolution principle. The immune system maintains a population of long-lived memory cells after clearance of infection and recruits newly generated B-cells into the memory. *Immune memory* enables the immune system to act quickly and efficiently in protecting the body in case it is infected by similar pathogens in the future. *Vaccination* follows the same principle. Summarizing, the essential features of the natural immune systems are distributed detection, self-organization, multi-layer structure, diversity, autonomy, imperfect detection, learning, memory and adaptability.

Several immune inspired algorithms have been developed and a relatively new research field, called Artificial Immune Systems (AIS), arose as a new computing paradigm. Nevertheless, AIS has not been used so far, to our knowledge, as a tool to monitor concrete software applications using resource usage metrics. Our model, for monitoring software applications, is based on metrics of the system resources and inspired in the natural mechanisms of the immune system, briefly discussed above.

There are similarities between the software fault detection problem and the identification of pathogens in natural immune systems. It should be mentioned that the natural immune system is merely used as a metaphor for anomaly detection and we are not trying to imitate all its features and detailed operation. Our inspiration comes mostly from the following three features:

1.1. *Self/non-self discrimination*

A healthy immune system is able to differentiate between the cells of its organism, know as *self*, and the foreign elements (antigens) that attack the organism, know as *non-self*. In the same way, an anomaly points toward a deviant behavior in relation to what is expected and characteristic of the system. Thus, inspired by the vertebrate adaptive immunity response, an algorithm is developed to distinguish common behavior of the host (*self*) from faults (*non-self*) in software applications.

1.2. *Vaccination*

The reason why we do not acquire some diseases more than once is because the immune system remembers pathogens. Vaccination is a good evidence that the immune system has memory. It consists in introducing into an organism some harmless organisms, which provoke an immune response against the foreign elements. As a consequence, immunological memory is induced which enables the immune system to act quickly and efficiently in protecting the body when it is actually infected by the real pathogens at some future time. Having this in mind, a kind of fault injection learning mechanism was created, as a part of the monitoring model. A fault injection is an application of an artificial malfunction, inserted into a particular monitored system with the purpose of simulating a specific error. Note that the monitored system is not actually affected by the fault. This process occurs in the interface between the monitoring model and the monitored application. It consists in intercepting metrics collected from the monitoring model into the monitored application and changing their values to outside the normal range.

1.3. *Adaptive immune response*

The adaptive immune system is composed of a large collection of cells with no central control, which together have the ability to improve the immune response during the lifetime of the host. The system evolves based on the principles of mutation and selection, producing a number of lymphocytes proportional to the degree of binding (affinity) with the antigen. As stated above, in software systems, unexpected behavior requires robust adaptive defensive mechanisms capable of recognizing new faults. As proposed by de Castro and Von Zuben [10], one is here inspired by the clonal selection concept, together with the affinity maturation process of the immune response, to create an adaptive monitoring mechanism.

In this paper we show that not only the model performs well in detecting faults, but also fault injection and reinforcement learning substantially decrease the detection of false positives. The article begins by reviewing the most important aspects of software-fault monitoring, describes the faults that are simulated as well as the identification of all the metrics that are collected in the monitored system. In Section 3, we present a computational algorithm that, in addition to detecting anomalies, also identifies its type. After that, the results and discussion of the simulation are presented as well as the most relevant conclusions.

2. Preliminary knowledge

We advocate the use of monitoring as a major design principle to increase safety, reliability and dependability of software applications. Many tools have been proposed for runtime monitoring with the purpose of detecting, diagnosing and recovering from software faults. Nelly Delgado and colleagues described the taxonomy of software-fault monitoring systems and presented a state-of-the-art of the tools used to detect faults (for details, see [6] and references therein). Note that none of the tools referred in their study are based on metrics of the system resources together with immune system inspiration to distinguish the common behavior of the host (*self*) from faults (*non-self*).

In 1994 Forrest et al. applied to the problem of computer viruses (see [8] for details), is of paramount importance for the scientific community, because it unifies a wide variety of computational situations by treating them as the problem of distinguishing self from non-self. Later on, enhancements were made to the original version of the Negative Selection algorithm proposed by Forrest et al. ([20–23], to name a few), but the main features remained unchanged. Another valuable application of AIS-based algorithms for fault prediction is the study by Catal and Diri [35]. The authors analyzed the performance of several existing classifiers using a different kind of metrics: software metrics (method-level and class-level; see [7] for details on software fault prediction metrics). When compared to others, AIS algorithms present remarkable results in predicting faults, however no experimentation are presented in detecting faults in real time, like we do here. Two other relevant methods address the fault detection problem using AIS combined with other technics. One is an approach based on conventional fuzzy soft clustering and AIS for multiple sensor data fusion and fault detection [36], the other is a multi-objective AIS to optimize parameters of a Support Vector Machine (SVM) applied to fault diagnosis of induction motors and anomaly detection problems [37]. Our model mimics a system that has features very close to a real system in contrast to the mentioned works that although showing good results, seem to be not sufficiently mature to face the requirements and complex behavior of practical applications.

To the best of the author's knowledge, the only work presenting an evolutionary technique that also invokes a set of resource usage metrics for software faults detection is that by Wong et al. [24]. Due to this similarity, their approach will be discussed in more detail in the Section 4.

In general terms, despite of all the important studies carried out in software monitoring, we did not find one that detects faults using metrics together with artificial immune adaptation technics. Furthermore, most of them are based in disparate approaches and methodologies as the existence of an oracle, i.e., determining if the systems behavior under test is or not acceptable, or concern whether the design or implementation of the system meets the requirements (or specifications) or the instrumentation of a program code.

The main goal of runtime software-fault monitoring is to observe the software behavior in order to determine whether it complies with its intended purpose, in other words, to determine if it is consistent with a given specification [19]. Avizienis and Laprie [25] gave widely accepted definitions of systems fault, error and failure. In summary, a system *failure* occurs when the delivered service deviates from the required service because the system was erroneous: an *error* is that part of the system state that is liable to lead to a *fault* in the system. A fault is active when it causes an error and results in an incorrect state that may or may not lead to a failure. Some faults are, deliberately or not caused by humans, whereas others are trigged by natural phenomena without human participation. Both may cause a huge impact, affecting partially or even completely the integrity of the whole system.

Here, we do not need to treat differently faults, errors or failures. We simply use the term *fault*, considered as a malfunction that affects the proper functioning or full availability of a system, leading some particular resource metrics to reach values outside their usual ranges. Three faults were simulated to evaluate the performance of our model (which was developed in Java):

- *Denial-of-service (DoS)*: An attack attempting to make a machine or network unavailable to legitimate requests by saturating its resources. We created a JMeter script that generates 100 virtual users. These users make a series of HTTP requests simultaneously for 30 consecutive seconds to the monitored system.
- *Memory leak*: In Java it is usually associated to errors in the garbage collection. Occurs when an object is stored in memory but cannot be accessed by the running code. We implemented a component that progressively inserts 5×10^6 strings into a static list with global scope. Because this list is static, it prevents the garbage collector from cleaning the occupied memory that persistently grows until there is no space left in the Java Heap.
- *Thread explosion*: Cyclic creation of a big quantity of threads. We implemented a component that progressively creates 5000 running threads, each of them sleeps for a second and terminates immediately after that.

An important reason for choosing these faults is that they are relatively common (especially DoS), however, the most important motive is that all of them affect different groups of resource metrics thereby allowing a wide and general interpretation of the model success in detecting different types of faults.

The original definition of dependability is the ability to deliver a service that can justifiably be trusted [26,27]. Simulation-based fault injection is important to evaluate the dependability of computer systems and there are already several techniques and tools to inject faults (see for example [28] and references therein). Whereas engineers use fault injection to test fault-tolerant systems or components, we apply a fault injection mechanism for artificial learning purposes. As stated before the process consists in intercepting and changing the collected metrics to values outside the normal range, as described in detail in the next sections.

Our model exclusively monitors Java applications. Briefly, Java is an object-oriented programming language that provides platform independence by executing programs on a Java Virtual Machine (JVM) installed on the operating system. To deal with the complexity of today's distributed systems, the Java community designed a straightforward specification to address the management needs of applications written for the Java platform. This specification, called Java Management Extensions (JMX), was also designed for the instrumentation of resources, providing tools for building distributed managing and monitoring applications [29]. An MBean is an object that follows the design patterns conforming to the JMX specification. It can be used to provide information about the performance and resource consumption of the applications running on the virtual machine. JMX MBeans interfaces offer several metrics of systems resources. Among them we make use of the following:

- *CPU time*: Returns the amount of process CPU time consumed by the JVM [30].
- *Load average*: Returns the system load average for the last minute. The system load average is the sum of the number of runnable entities queued to the available processors and the number of runnable entities running on the available processors, averaged over a given period of time [30].
- *Heap memory*: Returns the current memory usage of the heap that is used for object allocation [30].
- *Cms old gen*: Returns the number of objects, copied in memory by the garbage collector from the young generation to the old generation [31].
- *Thread count*: Returns the current number of live threads including both daemon and non-daemon threads [30].

Among the available metrics provided by the MBeans interfaces, these are especially useful because, together, they provide information on resource consumption referring to all simulated faults.

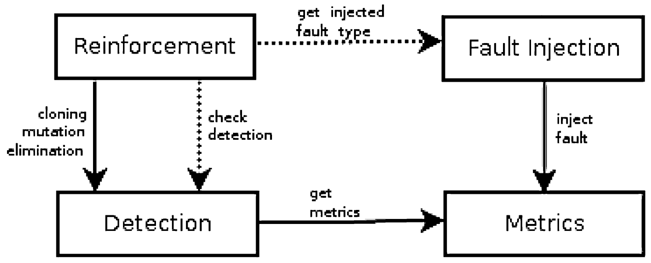

Fig. 1. Overview of fault injection reinforcement learning process.

Specifically, *denial-of-service* fault affects *CPU time* and *load average*, *memory leak* affects *heap memory* and *cms old gen* and finally *thread explosion* affects *thread count* and *CPU time*.

3. Algorithm

As stated before, our model combines mechanisms studied in Forrest's negative selection algorithm for protecting computer systems [8] and Castro's clonal selection algorithm (CLONALG) for solving complex machine learning tasks, like pattern recognition and multimodal optimization [10]. In our algorithm the negative selection feature is crucial in detecting faults, whereas the clonal selection feature promotes adaptation. The major difference between our algorithm and the ones mentioned before is the *fault injection reinforcement learning* process, which contributes to the improvement of the general performance of the model and drastically decreases the number of false positives. Reinforcement learning is learning by interacting with an environment in order to automatically determine the ideal behavior within a specific context. The learner is not told which actions to take, instead he must discover which actions yield the most reward by trying them [32]. Our algorithm uses reinforcement learning with the aim to improve the model accuracy in detecting faults. The procedure is autonomous and consists in injecting a fault into the monitored system by intercepting and changing the metrics values outside their usual ranges, and, after that, by checking whether the model detects it. As an example of a fault injection, consider the following metric values: CPU time = 20 and load average = 5 (the values of other metrics are irrelevant now). Injecting a denial-of-service fault might result in changing the metrics to the following values: CPU time = 90 and load average = 50. The learning process dynamics uses basically two types of feedback: (1) positive: cloning as positive reinforcement received by faults correctly detected; (2) negative: elimination or mutation as negative reinforcement received respectively by wrong detection of faults or no detection at all (see Fig. 1 for details about the fault injection reinforcement learning process).

The algorithm uses mutation and selection principles based on affinity measure between vectors, which is essential to the progressive adaptation of the monitoring model. Such behavior will lead to an increase on the overall affinity of the detector population. In addition, the affinity concept unifies a wide variety of combinations by allowing even a fault to be detected by matching only some complementary parts of the vectors. Besides, a method of approximate detection is of paramount importance for monitoring software applications because faults provoke different impact in their characteristic metrics. The algorithm has four phases. Note that in the first three phases one must ensure that the system is not affected by real faults because, if so, the faulty behavior would be incorrectly considered as typical.

The algorithm:

1. Self set generation
   1.1 periodically collect metrics from the target monitoring system (see Section 2 for details about the metrics chosen);

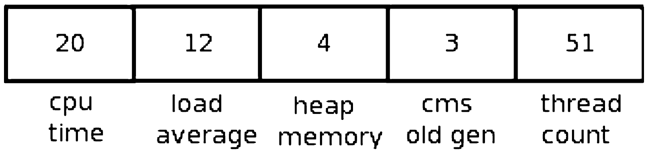

Fig. 2. Example of a vector signature containing all collected metrics.

   1.2 create a *vector* composed by all the metrics that together represent a *signature* of the system at that moment. Fig. 2 shows an example of a vector signature;
   1.3 insert the signature into the *self* set. If the number of *selfs* is enough, go to phase 2 or else go to step 1.1.
2. Detectors set generation
   2.1. generate a random signature vector representing an immature lymphocyte known hereafter as *detector*;
   2.2. calculate immature detector affinity measure with each vector signature of the self set (see Fig. 3). The affinity is calculated as the difference between signature's and detector's metrics. A signature metric *n* matches a detector metric *m*, if and only if $|n - m| < 3$. If the immature detector matches any self, reject it and go to step 2.1 or else proceed to the next step;
   2.3. insert the immature detector, which has passed the self filtration into the detectors set. If the number of detectors is enough, go to phase 3, or else go to step 2.1 again.
3. Learning process
   3.1. periodically inject faults by generating random values for metrics outside their usual ranges. Calculate the fake signature affinity with each detector of the detectors set. In case of a match, clone the activated detectors and slightly mutate the metrics;
   3.2. periodically check all the past detections and slightly mutate detectors without any detection until now;
   3.3. periodically collect metrics and calculate signature affinity with each detector of the detectors set. If a fault is detected without being injected (false positive) eliminate the detector(s) that wrongly detected it;
   3.4. if enough faults were injected, leave current learning process and proceed to phase 4 or else go to step 3.1.
4. Monitoring
   4.1. periodically collect metrics from the target monitoring system;
   4.2. create a vector composed with all the metrics that together represent a signature of the system at that moment;
   4.3. calculate signature affinity with each detector of the detectors set (the process is the same as depicted in Fig. 3). If the signature matches, signalize a fault by displaying an error message on the console identifying the type of the fault and its corresponding metric values. Go to step 4.1.

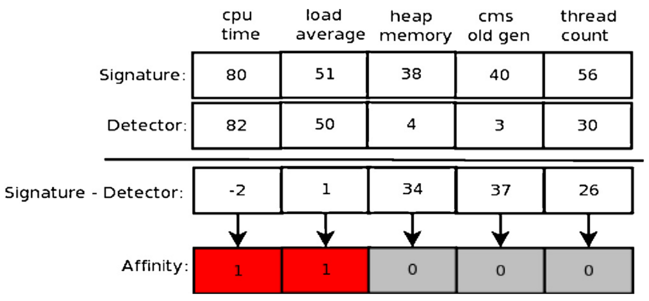

Fig. 3. Affinity measure match between a signature and a detector. A signature metric *n* matches a detector *m*, if and only if $|n - m| < 3$. Affinity vector is composed by 1's to represent position activation and 0's to non-active metric positions.

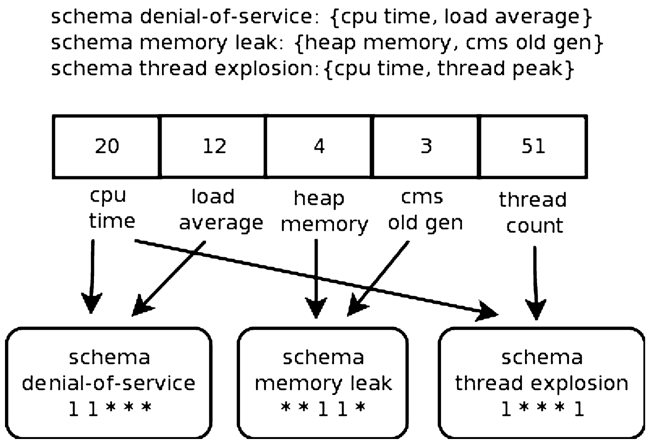
schema denial-of-service: {cpu time, load average}
schema memory leak: {heap memory, cms old gen}
schema thread explosion: {cpu time, thread peak}
20
12
4
3
51
cpu time
load average
heap memory
cms old gen
thread count
schema denial-of-service 1 1 * * *
schema memory leak * * 1 1 *
schema thread explosion 1 * * * 1

Fig. 4. A vector signature and the three "*fault detection schemas*".

As mentioned above, *denial-of-service* fault affects the *CPU time* and *load average* group of metrics, *memory leak* affects *heap memory* and *cms old gen* group of metrics and finally *thread explosion* affects *thread count* and *CPU time* group of metrics. We named these three groups as "*fault detection schema*". The term is based on John Holland's schema theorem for genetic algorithms [33]: a schema identifies a subset of strings with similarities at certain string positions. A "*fault detection schema*" is a binary string of length 5, where 1's represent the positions of the signature vector whose metrics must match the same detector positions for a fault be signalized. For other signature vector positions we use the asterisk to be a wildcard character whose bit could be either 0 or 1. This representation allows us to identify not only the occurrence of more than one fault at same time, but also its respective types. Fig. 3 already depicted this representation in the affinity vector result. Consider now Fig. 4 that illustrates a signature example and the three "*fault detection schemas*". Suppose a "*fault detection schema*" bit string as 11,000, which indicates a match in first (CPU time metric) and second (load average metric) positions of the vector, in this case a denial-of-service fault is signalized by our model.

Our monitored application is a web application developed in Java Enterprise Edition (Java EE) deployed in a Weblogic application server that communicates to a MySQL database storing a financial credit dataset downloaded from UCI Irvine Machine [34]. This application offers five different services: list all clients, create all clients, show client, update client and remove client.

Only very rarely the activity of the software system remains constant during the day. There are periods where accesses are higher, then lower, then higher again and so on. In many cases these periods tend to follow the same daily pattern. If these changes were not considered, it would be very difficult to monitor with a unique self set storing all the signatures collected in different access periods, which do not represent current typical behavior of the system. Hypothetically we have considered six 4-h periods of distinct accesses (see Fig. 5). This implies the four phases of the algorithm should be run for each of the six different periods considered.

4. Results

JMeter [1] is a desktop application designed to simulate concurrent load on a variety of services in order to test its behavior, analyze and measure overall performance. We used JMeter to perform the simulation of the six periods by sending simultaneous virtual user accesses into the monitored application services. For convenience in reducing the time of experimentation we simulated the six periods in 6 min (1 min each) that corresponds to a day. Table 1 lists a fixed quantity of simultaneous virtual users for the

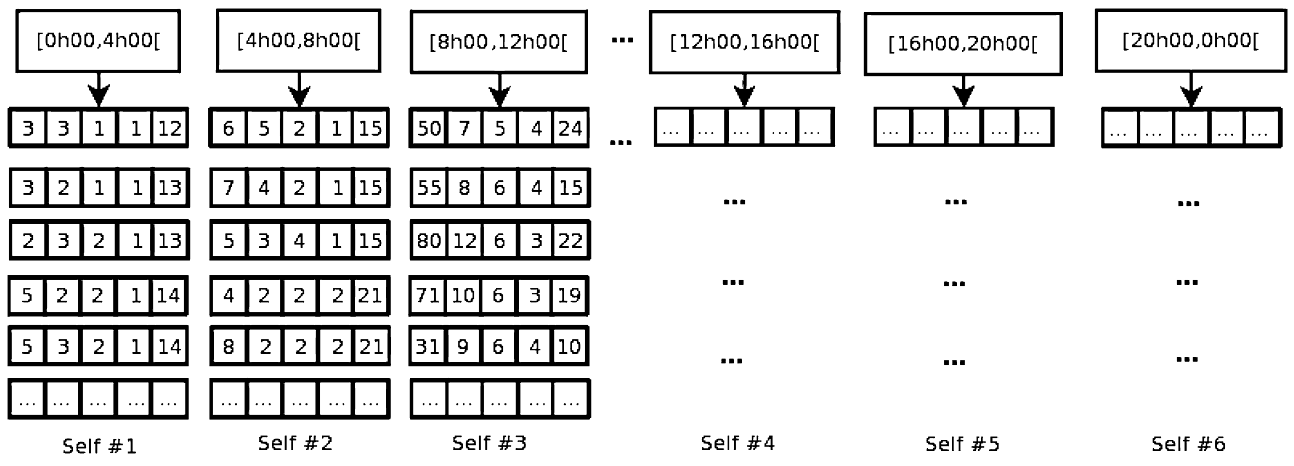

Fig. 5. Six periods and the corresponding six self sets.

Table 1
Quantity of simultaneous virtual users for the six different periods.

Period interval	Virtual users
[0h00,4h00[	1
[4h00,8h00[	5
[8h00,12h00[	25
[12h00,16h00[	15
[16h00,20h00[	10
[20h00,24h00[	20

Table 2
Reinforcement feedback actions of the learning process phase of the algorithm.

Action	Quantity (detectors)
Clones	13,303
Eliminations	1,968
Mutations	67,918
Total	*83,189*

six periods throughout the simulation. Take notice that this quantity of virtual users was considered appropriate taking into account the resources used (MacBook Pro; 2.26 GHz Intel Core 2 Duo; 5 GB RAM). However, in other circumstances it may not achieve the intended purpose.

The metrics were collected from the monitored application every 5 s that proved to be more than sufficient to guarantee proper monitoring. To correctly evaluate the impact of the learning process, initially our experiments were performed only with the negative selection feature, without the learning process, and later more experiments were carried out using also the third phase of the algorithm. In our experiments we set 100 as the amount of self elements and 500 as the amount of detectors for each of the six periods. We injected a total of 100 faults (denial-of-service, memory leak and thread explosion) for each period. These parameters were compiled from a pilot, developed to test the impact of the values variations as well as the computational load of the algorithm. We conclude that the performance is affected by gathering of five metrics of system resources that consume 2% of CPU load, as well as, by the increase of the amount of detectors in the set which consume approximately 1% of CPU load per 250 detectors. In all experiments, in the last phase of the algorithm (4. Monitoring) we simulated 90 faults (30 of denial-of-service, 30 of memory leak and 30 of thread explosion) into the monitored application.

Below, we present the results of a simulation that covers all the algorithm phases. At the end of this section we will present the results obtained without learning. The graphics of the metrics collected for the several selfs belonging the six periods are illustrated in Figs. 6 and A1–A5. These graphics display the activity of the first phase of the algorithm and, as expected, the number of virtual users directly influences metric values.

During the simulation, we send a total of 187,750 virtual user requests to services of the monitored application. These requests occurred during the different period intervals equally divided by the five services (37,550 requests each). The reinforcement feedback actions that took place during the third phase of the algorithm (3. Learning process) are presented in Table 2.

The simulation stayed at the last phase of the algorithm (4. Monitoring) during 2000 iterations. The results of a simulation with all the phases of the algorithm are listed in Table 3.

It is important to notice that faults simulated as one schema type can be wrongly detected as belonging also to others. As can be seen from Table 3 for the 30 denial-of-service faults simulated, although 29 of them are correctly detected, 28 were wrongly detected as thread explosion. No faults were simulated in the last 500 iterations of the last phase of the algorithm, however the model detected 3 faults (false positives), 2 thread explosions and 1 denial-of-service. Recall that virtual users keep doing requests that obviously affect resource metrics.

To understand the influence that the learning process has on the algorithm, a simulation without the third phase is presented next. The model without learning relies exclusively upon the negative selection feature, sufficient however to detect faults. In this case the algorithm jumps directly from the second to the fourth phase. Table 4 presents the results.

In these simulation 51 false positives were detected, 31 of which being the denial-of-service and 20 the thread explosion. Observe that there are much more faults detected of any type.

The work by Wong et al. [24] describes a evolutionary technique that uses genetic programming to automatically evolve an accurate utility function for a specific system, set of resource usage metrics, and precision recall reference. As in the case in question (as here), metrics are computed using sensor values that monitor a variety of system resources (e.g., memory usage, processor usage, thread count). There were five attacks and faults (denial of service, infinite loop, log file explosion, memory leak and recursion) on the Jigsaw web server[1] intended to either deny Jigsaw the ability to perform its normal hosting functions or add additional, malicious functionality to the running server. Table 5 presents the results of the fitness function scores for the evolved predicates as well as of both false positives (fpos) and false negatives (fneg). For further details see [24].

[1] http://www.w3.org/Jigsaw/.

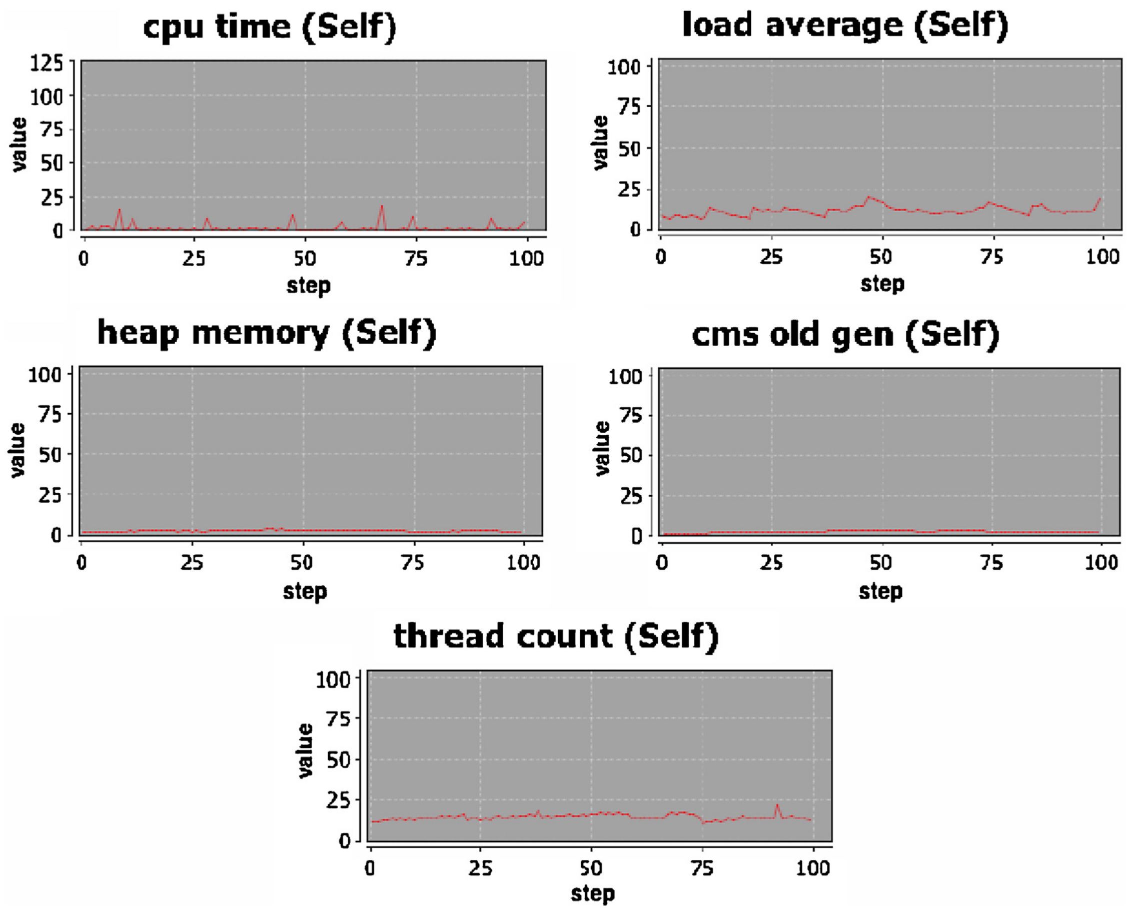

Fig. 6. Metrics of the "[0h00,4h00[" period interval (1 virtual user).

Table 3
Monitoring results of the simulated faults.

Fault simulated	Quantity	Detected	Detected	Detected
Schema		Denial-of-service	Memory leak	Thread explosion
Denial-of-service	30	29	0	28
Memory leak	30	2	30	0
Thread explosion	30	3	0	27
Total	*90*	*34*	*30*	*55*

Table 4
Monitoring results of the simulated faults without reinforcement learning.

Fault simulated	Quantity	Detected	Detected	Detected
Schema		Denial-of-service	Memory leak	Thread explosion
Denial-of-service	30	28	23	30
Memory leak	30	28	30	27
Thread explosion	30	24	2	25
Total	*90*	*80*	*55*	*82*

From the results presented above it is easy to conclude that, in terms of efficiency, what distinguishes these techniques is the higher quantity of wrong faults detected by our algorithm (we will explain the reason for this in the next section). Despite performing well, by reducing the problem to a single utility function the approach presented by Wong et al. has difficulties in determining the root cause and other details of the faults, and our approach has a clear advantage in this matter.

Table 5
$F\beta$ fitness function scores for the evolved predicates.

β Value	$F\beta$ score	fpos rate	fneg rate
0.5	0.996344	0.000%	1.835%
1	0.998327	0.000%	0.335%
2	0.998786	0.539%	0.016%

5. Discussion and conclusion

When comparing the graphics of Figs. 6 and A1–A5, one sees a pronounced difference between some metric values, while others do not seem to be influenced by the virtual user requests variation. The reason is that the services provided by the monitored application merely perform basic database CRUD (create, read, update, delete) operations, thereby not very demanding for all the resources used here.

The learning process contributed, through reinforcement rewarding actions, to the overall adaptation of the monitoring model. By examining Table 2, it is easy to infer that the learning process affected many detectors. The 67,918 mutations in detectors schemas values mean that there are a huge quantity of detectors that did not detect any fault. Mutations increase the probability of success in detecting future fault injections. Note that the detector might have multiple mutations and that the population is not constant because it grews after cloning and reduces after elimination.

There were 1968 eliminations, which indicate incorrect detection occurring in two ways: injection of a fault of one type, but detected as being of other type (from another schema) and detection of a fault where no fault injection occurred. The elimination measure, in general terms, gives an idea of the model accuracy and implies that the negative selection feature, by itself, scores lots of false positives detections. The reason is that those eliminations are done in detectors gathered during the second phase, that proved to be bad detectors. The elimination of these detectors naturally promotes the decrease of false positive detections.

There were 13,303 detectors cloned. Cloning was initially carried out at a lower pace, gradually increasing as model learning. As the detector population grew promoting the fittest detectors, the cloned ones are also good candidates to be cloned in the future. Furthermore, there were more detectors signalizing the same fault and the cloning mechanism insured that the model converged to suitable detection intervals, contributing globally for better decisions. At the end, the population was higher because there were more clones than eliminations.

Of all the 90 simulated faults (see Table 3), 86 were correctly detected (29 of denial-of-service, 30 of memory leak and 27 of thread explosion). This is a very high detection rate (96%). However, there were too many faults signaled as being of one type when actually were from other type. For instance, the model detected only one less fault of thread explosion (28) than of denial-of-service (29) for injected faults of this last type. These results might cast doubt on the model accuracy, especially of the learning process. The fact is that these results occurred due to the peculiarities of the simulation JMeter scripts used in the experimentation. When simulating a denial-of-service, 100 virtual users' requests create a lot of threads, which are collaterally detected by the thread explosion schema.

Unquestionably, only three false detections by the model with learning is a good result. Lets us now pass to the results of the model without learning to better support our conclusion. Through the observation of the results presented in Table 4, it is easy to see (even taking into account the peculiarities of the JMeter scripts) that the model without learning has limitations in identifying the type of the simulated faults. A more serious limitation is the huge amount of false positives (51 in 500 iterations). Nevertheless, of all the 90 simulated faults 83 were correctly detected (28 denial-of-service, 30 memory leak and 25 thread explosion). This is a very high detection rate (92%) slightly below the results of the model with learning (96%).

The main goal of this research was to develop a computational model to monitor software applications developed in Java. We do consider it as a proof-of-concept whose insights might contribute to the development of monitoring tools and future studies in the AIS field. It was demonstrated that the algorithm shows good performance in detecting faults, as well as in identifying their type. Acting alone, the negative selection feature detects too many false positives, but the reinforcement learning process enables the overall model to become robust and efficient. Learning minimizes inconsistencies inherent in the randomness of the detectors during the negative selection. Without difficulty this model is adaptable to other kinds of monitored applications. The model is very flexible because it is based in performance indicators and consumption resources, which enables it to be generalized to other technologies.

Some of the results concerning the identification of the different types of faults were not as expected. They occurred due to peculiarities in the simulation JMeter scripts used in experimentation. A limitation of the algorithm is that during the first three phases one must ensure that the system is not affected by real faults because, if so, the faulty behavior would be incorrectly considered typical. After that, when entering the last phase, the algorithm would not readapt. Another shortcoming is the need of prior identification of the metrics affected by the faults.

For future research we plan to implement a feature that automatically readapts to changes in typical behavior of the monitored application, if they actually occur. Although we consider denial-of-service, load average and thread explosion sufficient to evaluate our model, we intend to experiment other types of faults to confirm the model performance and accuracy. Additionally, it will be of interest to integrate this model with others, especially with fault tolerance, fault prevention, fault removal and fault forecasting. Another important prospective work will be implementation of supervised learning mechanism that corrects false positive detections.

Appendix A.

See Figs. A1–A5.

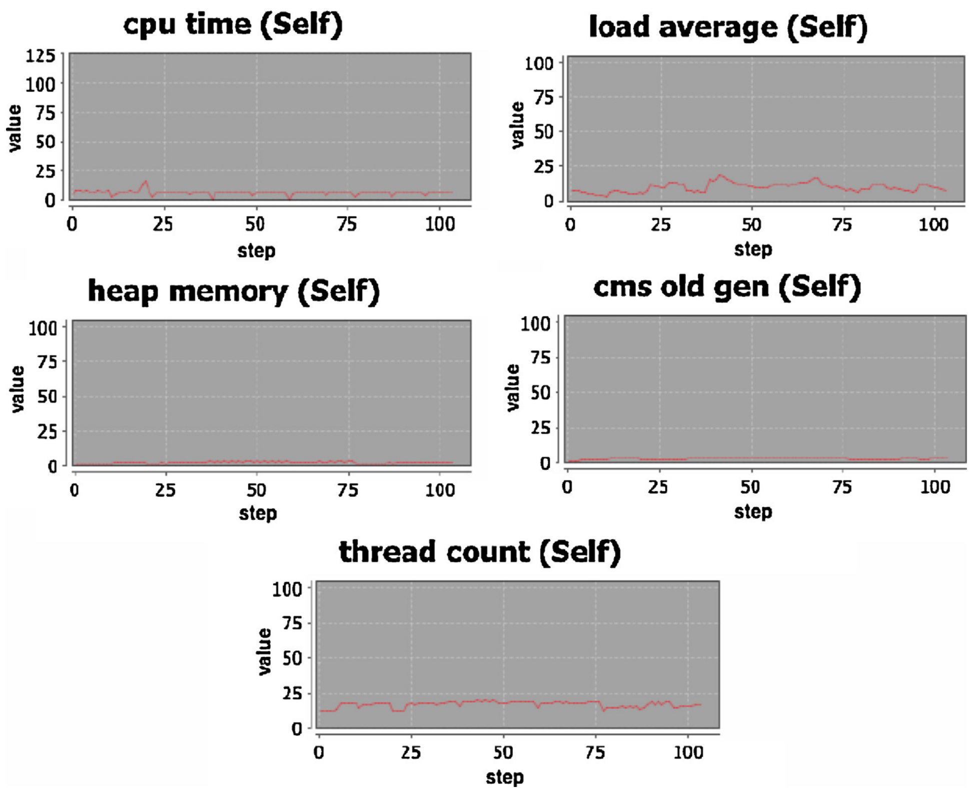

Fig. A1. Metrics of the “[4h00,8h00[” period interval (5 virtual users).

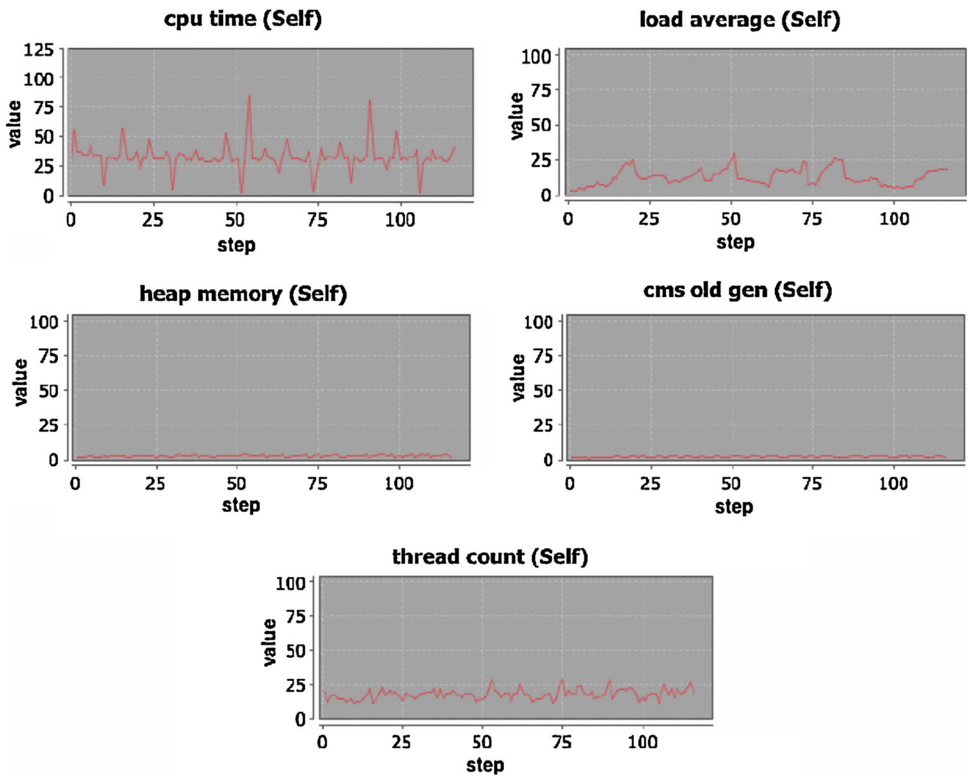

Fig. A2. Metrics of the “[8h00,12h00[” period interval (25 virtual users).

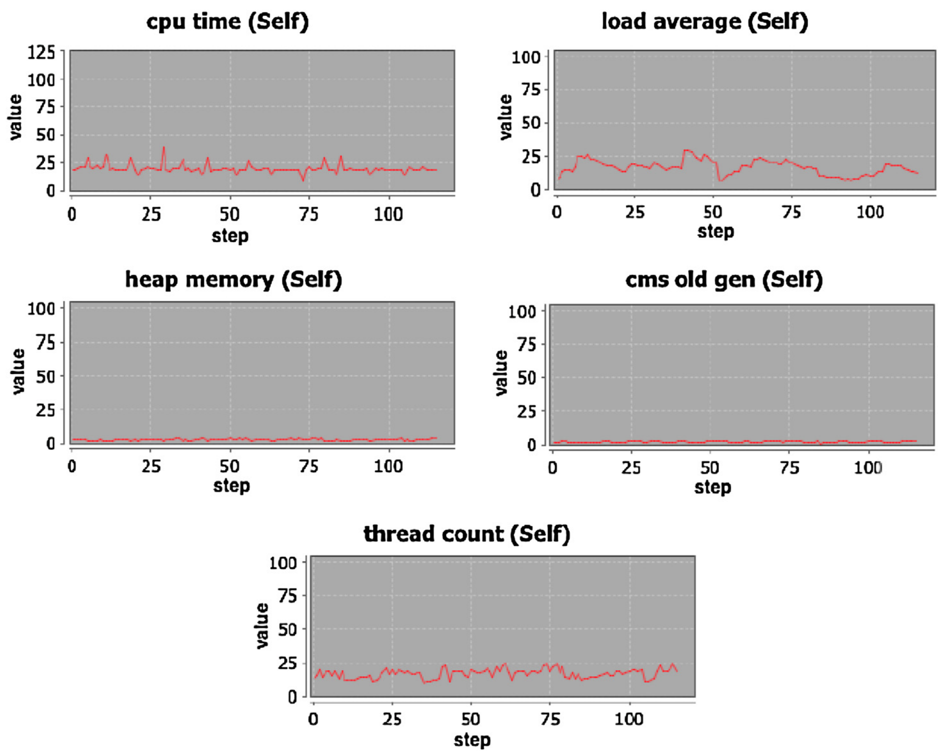

Fig. A3. Metrics of the "[12h00,16h00[" period interval (15 virtual users).

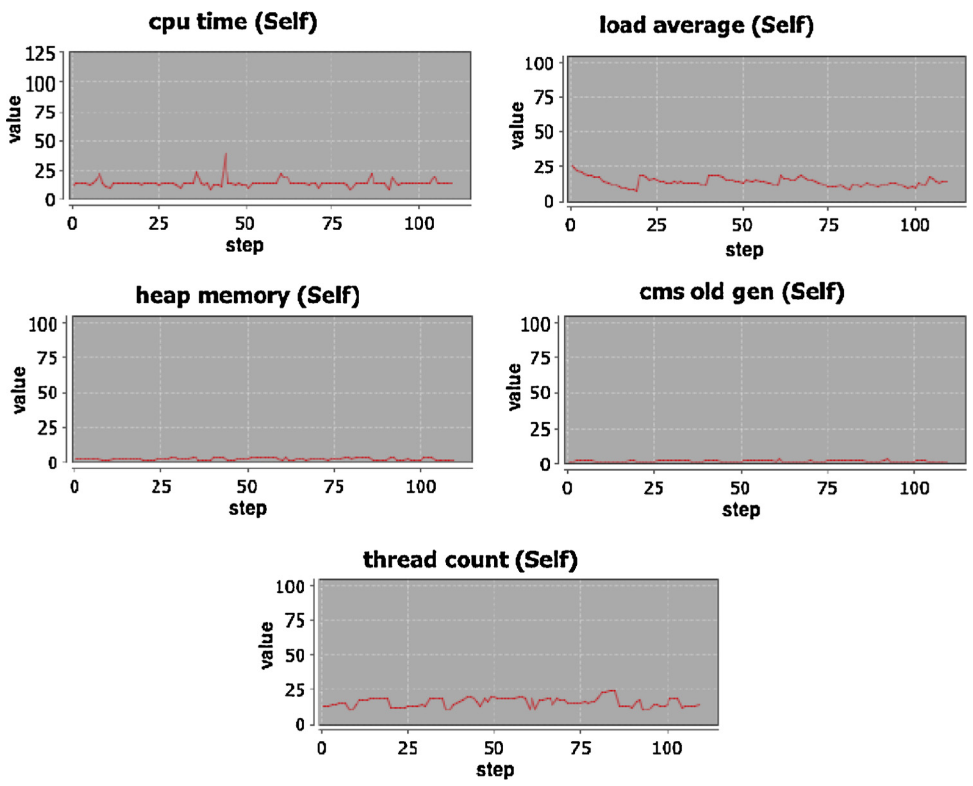

Fig. A4. Metrics of the "[16h00,20h00[" period interval (10 virtual users).

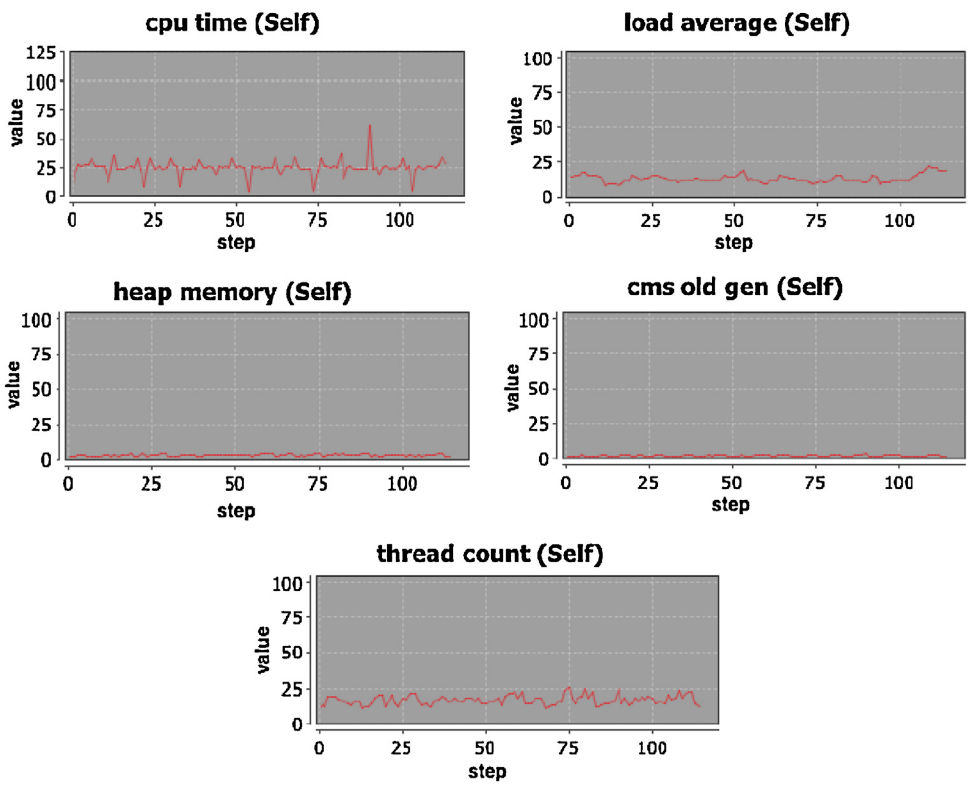

Fig. A5. Metrics of the "[20h00,24h00[" period interval (20 virtual users).

References

[1] Jmeter, http://jakarta.apache.org/jmeter/
[2] Cacti, http://www.cacti.net/
[3] Nagios, http://www.nagios.org/
[4] JBoss RHQ, http://www.jboss.org/jopr/
[5] Hyperic, http://www.hyperic.com/
[6] N. Delgado, A.Q. Gates, S. Roach, A taxonomy and catalog of runtime software-fault monitoring tools, IEEE Trans. Softw. Eng. 30 (2004) 859–872.
[7] D. Radjenovic, M. Herico, R. Torkar, A. Zivkovic, Software fault prediction metrics: a systematic literature review, Inf. Softw. Technol. 55 (2013) 1397–1418.
[8] S. Forrest, S. Perelson, L. Allen, R. Cherukuri, Self-nonself discrimination in a computer, in: IEEE Symp. Res. Secur. Priv., Los Alamitos, CA, 1994.
[9] J. Kephart, A biologically inspired immune system for computers, in: Proceedings of Artificial Life IV: The Fourth International Workshop on the Synthesis and Simulation of Living Systems, MIT Press, 1994, pp. 130–139.
[10] L.N. de Castro, F.J. Von Zuben, The clonal selection algorithm with engineering applications, in: GECCO'00 – Workshop Proceedings, 2000, pp. 36–37.
[11] L.N. de Castro, J. Timmis, Artificial Immune Systems as a Novel Soft Computing Paradigm Soft Computing, vol. 7, Springer-Verlag, 2003, pp. 526–544.
[12] L.N. de Castro, Fundamentals of Natural Computing: Basic Concepts, Algorithms, and Applications, Chapman & Hall/CRC Computer & Information Science Series, 2006.
[13] M. Mitchell, Complexity: A Guided Tour, Oxford University Press, New York, 2009.
[14] D. Floreano, C. Mattiusi, Bio-Inspired Artificial Intelligence: Theories, Methods, and Technologies, The MIT Press, Cambridge, MA, 2008.
[15] A. Perelson, G. Weisbuch, Immunology for physicists, Rev. Mod. Phys. 69 (4) (1997) 1219–1267.
[16] H. Pagels, The Dreams of Reason: The Computer and the Rise of the Sciences of Complexity, Simon and Schuster, New York, 1988.
[17] A. Somayaji, S. Hofmeyr, S. Forrest, Principles of a computer immune system, in: Proceedings of the Second New Security Paradigms Workshop, 1997.
[18] P.J.C. Branco, J.A. Dente, R.V. Mendes, Using immunology principles for fault detection, IEEE Trans. Ind. Electron. 50 (2) (2003).
[19] D. Peters, Automated Testing of Real-Time Systems; Technical Report, Memorial University of Newfoundland, 1999.
[20] P. D'haeseleer, S. Forrest, P. Helman, An immunological approach to change detection: algorithms, analysis and implications, in: Proc. of the IEEE Symposium on Computer Security and Privacy, 1996.
[21] D. Dasgupta, S. Forrest, Novelty-detection in time series data using ideas from immunology, in: Proc. International Conference on Intelligent Systems, Reno, Nevada, 1996.
[22] A. Hofmeyr, S. Forrest, Architecture for an artificial immune system, Evol. Comput. 8 (4) (2000) 443–473.
[23] Z. Ji, D. Dasgupta, Revisiting negative selection algorithms, Evol. Comput. 15 (2) (2007) 223–251.
[24] S. Wong, M. Aaron, J. Segall, K. Lynch, S. Mancoridis, Reverse engineering utility functions using genetic programming to detect anomalous behavior in software, in: Proc. 17th Working Conference on Reverse Engineering, 2010.
[25] A. Avizienis, J. Laprie, Dependable computing: from concepts to design diversity, Proceedings of the IEEE 74 (May (5)) (1986) 629–638.
[26] A. Avizienis, J. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Transactions on dependable and secure computing 1 (1) (2004).
[27] A. Avizienis, J. Laprie, B. Randell, C. Landwehr, Fundamental Concepts of Dependability; Tech. Rep. N01145, LAAS-CNRS, 2001.
[28] M. Hsueh, T. Tsai, R. Iyer, Fault injection techniques and tools, IEEE Comp. 30 (4) (1997) 75–82.
[29] S.J. Perry, Java Management Extensions, O'Reilly Media, Sebastopol, CA, 2002.
[30] Server Administration Guide, Oracle Utilities Meter Data Management, Version 2.0.0 (OUAF 4.0.2), E18183-01, 2010.
[31] Memory Management in the Java HotSpot^{TM} Virtual Machine, Sun Microsystems, Santa Clara, CA, 2006.
[32] R. Sutton, A. Barto, Reinforcement Learning: An Introduction, The MIT Press, Cambridge, MA/London, England, 1998.
[33] J. Holland, Adaptation in Natural and Artificial Systems;, University of Michigan Press, Cambridge, MA/London, England, 1975, Reprinted by The MIT Press (1992).
[34] UCI Machine Learning Repository, http://archive.ics.uci.edu/ml/
[35] C. Catal, B. Diri, Investigating the effect of dataset size metrics sets and feature selection techniques on software fault prediction problem, Inf. Sci. 179 (8) (2009) 1040–1058.
[36] M. Jaradat, R. Langari, A hybrid intelligent system for fault detection and sensor fusion, Appl. Soft Comput. 9 (2009) 415–422.
[37] I. Aydin, M. Karakose, E. Akin, A multi-objective artificial immune algorithm for parameter optimization in support vector machine, Appl. Soft Comput. 11 (1) (2011) 120–129.