An Autonomic Computing Mechanism to Enable Self-Protection and Self-Healing

Luis M. Fernández-Carrasco, Hugo Terashima-Marín, Manuel Valenzuela-Rendón

{marcelo.fernandez, terashima, valenzuela}@itesm.mx
Tecnológico de Monterrey, Monterrey Campus
Center for Intelligent Computing and Robotics

Abstract. The use of computing systems is nowadays something that is taken for granted. One just needs to look around and will easily see that there is a computation process going on in almost every direction. Moreover, now such processes are not just restricted to personal computers but to devices such as cellular phones, PDAs, laptops, etc. Furthermore, these devices are not isolated units of processes but are interconnected and can send and receive information at any time, anywhere. The demands that now people and current business models place on computing systems go from running a simple application, where the hardware was not built specifically for such application, to a cooperative network where all constituents are using a variety of systems and commands. Managing all these networked devices as a whole in a robust, safe, secure and transparent manner demands a lot of resources and time.
This article presents the steps that have been taken in order to propose a new system architecture that ensures the well-being of an IT infrastructure by providing a self-protection and self-healing autonomic computing features. The model is a combination of a multiagent design and learning techniques. The main idea is to mold each component that a typical operating system manages as an agent, incorporate performance criterion evaluators to select the best candidate to perform a task and implement specialized agents to supervise and learn from threats and normal program executions in order to keep the system running (self-protection and self-healing).
The system was evaluated using a multiagent simulation. Consequently, a programming language was created, named HAL, which allowed the simulation of applications, both benign and harmful, running in the multiagent environment. Thus, the simulated prototype is very close to a computer that runs applications, allowing a proper evaluation of the proposed design.

1 Introduction

It is a fact that computers have *invaded* today's world. Everywhere one may look, it is almost sure that there is a computer executing a calculation or delivering a service. Moreover, all these *small* computers are performing various specific functions in information processing for industrial age devices like cars [1]. These are known as embedded computer systems and, although less visible, they are the ones that are being used the most providing easy access to information. Computers have *evolved* from single huge machines to modular oriented systems to personal computers networked with larger machines (i.e., servers) [2]. As a matter of fact, there has been an incredible progress in almost every aspect of computing in the last 20 years, microprocessors power up by a factor of 10^4, storage capacity by a factor of $45x10^3$, communication speeds by a factor of 10^6 [3].

Nevertheless, all this great improvement in performance has had a price. The code that governs all the mentioned systems has grown to millions and millions of lines and the correct functioning of these systems require the implementation of even more code, trying to foresee any possible complications or situations the system may face. As one may assume, this high demand of code also demands a large number of people. For instance, some operating environments weigh in at over 30 million lines of code created by over 4000 programmers [3].

As Figure 1 shows, the more technology progresses, the more complex it becomes to handle the computing systems people rely on. From Figure 1 one can see that complexity is something inevitable as science is always moving forward, trying to improve a previous development. This demand for progress is what makes complexity something that cannot be avoided, systems of components are built up to the point where they are still manageable and under control; at this point, they begin to fail and fall below everyday requirements. Computing systems are always evolving and, thus, in order to move system components up to the next level, more complex systems are built, and the cycle starts all over again. This cycle, consequently, is a threat to the well-being of the system (i.e., reliability is compromised).

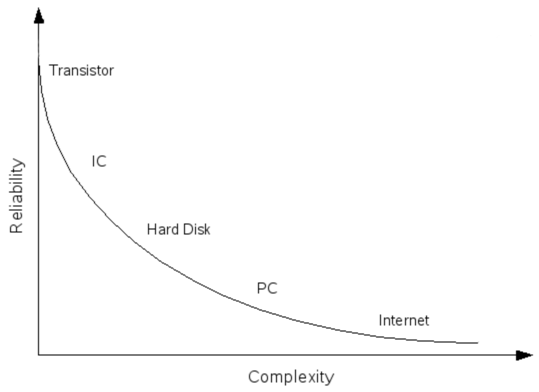

Fig. 1. Reliability versus Complexity

The following sections of this document present the efforts that were followed in order to provide an alternative to ensure the security of the system by employing an autonomic computing approach. In other words, by building a system that would be self-protecting and self-healing. The proposed solution relies on a multiagent system and it is tested on a simulated environment that was attacked by programs that were created using the newly design programming language HAL which allows the simulation of program executions, something similar to what happens in real IT infrastructures.

2 Related Work

In the last years, autonomic computing has caught the attention of many important IT related companies that have started to work on this issue. There is the IBM Research Autonomic Computing Group, the ones that started this field [4]. Microsoft has also started its own search for autonomic computing, the Dynamic Systems Initiative, an approach to look for ways to have IT work closely with business in order to meet the demands of a rapidly changing and adaptable environment [5]. Planetary Computing is the answer HP Labs have for autonomic computing combining software development and hardware innovation [6]. Some other examples, not less important, are Dells Dynamic Computing Initiative, Hitachi's Harmonious Computing and Electronic Data Systems' (EDSs) Agile Enterprise [7]. Nowadays, more and more companies are actively doing research in autonomic computing and it is for sure that, in the near future, more will be involved in such endeavor [8].

Autonomic computing has not only attracted software companies but also researchers from academia and industry. Consequently, a lot of research has already taken place in this field. Some selected results and readings for self-healing are the ones presented by Agarwala and Schwan [9] who proposed a solution based on system monitoring at very low level; and Zheng et al. [10] who employed decision trees to detect failure in computing systems.

Representative work in the self-protection area is the work done by Jiang et al. [11] proposed multi-resolution abnormal trace detection using varied-length N-grams and automata; Lohman et al. [12] who make use of symptom detection to find software problems; and Qin et al. [13] who treat bugs as allergies in order to provide a method that would enable to survive software failures.

The difference between the solution this article presents and the ones mentioned lines above is that this document employs a multiagent approach which allows a decentralized solution to the protection and healing problem. The next section of this article presents in more detail the proposed solution.

3 Proposed Solution

Self-healing and self-protection are achieved in the proposed system by implementing a new kind of agents called guardians. These agents are in charge to *learn* what action or program executions can cause a problem to the system, prevent such attacks and *heal* the system once an attack has taken place.

The idea that is employed in this approach is *the more, the better* which basically means that if the execution pattern of a process is seen many times, it is considered to be a benign one. This is based on the fact that users only make use of a handful of applications and these are used almost constantly. These executions, then, are normal to the system, whereas, unseen execution patterns are most likely to be an anomaly. The way the *guardian* agents perform their tasks is described in the following lines

3.1 Protection by Learning: Using Online k-means

Given a set of P examples $\mathbf{x}_i$, the k-means algorithm computes k prototypes $m = m_k$ which minimize the average distance between each pattern and the closest prototype. The observations $\mathbf{x}_i$, for the present model, are 3-dimensional Euclidean vectors. The first two components of these vectors make use of a variation of what a *basic block* is using the idea proposed by Sherwood et. al. [14] to characterize programs[1]. However, these elements are not exactly basic blocks but a more relaxed concept. The first component is the number of calls to procedures or functions $P = \sum_i p_i$ inside a program execution. The second component is the number of calls of flow-control structures $S = \sum_j s_j$, i.e., *while*, *for*, etc. The last component is the averaged number of times, $C = \frac{\sum_k c_k}{N}$, that the program has been executed. In other words, the vector's components are $\mathbf{x}_t = \langle P_t, S_t, C_t \rangle$.

When the system starts, it is necessary to initialize $\mathbf{m}_i$. This is done considering that, in average, harmful program executions do not take place so frequently whereas programs that are valid and safe are executed more times. Based on this, appropriate values can be given to $\mathbf{m}_i = \langle P_i, S_i, C_i \rangle$ and the learning process can begin. In other words, $\mathbf{m}_i$ is initialized with a possible $\mathbf{x}_t$ entry. This process is supported by the work conducted by Bottou and Bengio [15].

3.2 Healing the System: Following High-Level Policies

The self-healing features are mainly set by high-level policies, this is so since it is the business environment that decides how a problem is solved, in other words, healed.

Basically, whenever a problem is seen, the *guardian* agent follows a set of rules in order to provide a solution to the situation. These rules are fired once, thanks to the guardian agent, the system is moved back to the last known healthy state:

– Stop the execution of the conflicting process.
– Try one more time the execution process of the invalid process.
– Ask for execution of tasks that are needed according to the problem.
– If a solution is not found, this process is banned from any future execution.

[1] A basic block is code that has one entry point (i.e., no code within it is the destination of a jump instruction), one exit point and no jump instructions contained within it.

4 Experimentation & Evaluation

Based on the proposed model, there are some things that have to be setup before starting any evaluation of the architecture. Accordingly, the following lines of this section present how these items were initialized.

4.1 Defining the Learning Rate η

In order to achieve convergence in the online k-means algorithm, η has to be gradually decreased to zero. But this implies the *stability-plasticity* dilemma [16]: If η is decreased toward zero, the network becomes stable but adaptivity to novel patterns that may occur in time is lost because updates become too small. If η_i is kept large, $\mathbf{m}_i$ may oscillate.

Following the suggestions provided by Bottou and Bengio [15], the learning rate was set to be $\eta_i = \frac{1}{N_i}$ where N_i is the number of examples so far assigned to the prototype $\mathbf{m}_i$.

4.2 Defining Initialization Values for Prototypes $\mathbf{m}_i$

In order to initialize $\mathbf{m}_i$, it was considered that, in average, harmful program executions do not take place so frequently whereas programs that are valid and safe are executed more times and are used almost always. People make use of computers because the programs they run provide a service; the better the service, the more the program is used.

Following this idea and the suggestions proposed by Bottou and Bengio [15] $\mathbf{m}_i$ was initialized with $\mathbf{x}_t$ entries, assigning a very low value to C_i if the prototype was to represent harmful programs. Similarly, a higher value to C_i was assigned if the prototype was to represent safe program executions. The proposed values are listed in Table 1, where P is the number of calls to procedures or functions inside a program execution, S is the number of calls of flow-control structures, and the last component is the averaged number of times that the program has been executed.

Table 1. Initialization values for the $\mathbf{m}_i$ components

Prototype Components	**Safe Applications Prototype**	**Harmful Applications Prototype**
P	3	2
S	2	3
C	0.7	0.1

4.3 Determining Number of Agents

In order to test the proposed solution, the simulation of small data center was considered, consequently, one hundred computers were created trying to provide variability in the simulated data center by having different computer configurations.

4.4 Determining Number and Kind of HAL Programs

In order to test the robustness of the proposed architecture a number of 35 HAL programs were coded. The objective of these programs is to emulate those that a person would typically make use of. Of course, there is not a word processor coded as a HAL program, but all the created programs demand the use of memory, input/output devices (such as monitors and keyboard inputs), and the use of the processor, just like a real-life application.

Besides these *safe* programs, there was the need to code other ones that would *attack* the system. These programs basically attempt to destabilize the system by trying to erase memory segments (for instance, where the module manifests are stored), provide false or wrong data (e.g., trying to instantiate an array with negative size), and attack agents by sending ill-formed messages. The number of created HAL programs with these characteristics was also 35.

4.5 Self-Protection Results

This section presents the results obtained when evaluating the self-protection capabilities of the proposed architecture.

Figure 2 depicts the averaged learning curve for all three types of computers that were created for the system. As it can be seen Computer Type 3 learns faster the applications. This makes sense as this kind of computer agents have more processor units and processing capability. However, all three types of computers are able to learn and distinguish the programs quite fast. This is important as the system should be able to differentiate harmful programs from safe ones.

Consequently, it is important to see how well the proposed system classifies safe and harmful HAL program instances. This is shown graphically in Figure 3. The idea of this plot is to see how many safe programs were classified as *bad*, i.e., false positive instances, and how many harmful programs were classified as *safe*, i.e., false negative instances. The ideal world should show a very small number of those two; however, a large number of false positive events only could also be acceptable.

Figure 3 shows the false positive and negative averaged results for all three types of implemented computers (the top one chart is for computer type 1, the middle one is for computer agent 2, and the one at the bottom is for computer agent 3). As it can be seen, the number of false negative events is smaller that the number of false positive. This is more evident in the bottom chart where computer type three is able to improve its classification as it learns more about the characteristics of the HAL programs that are run.

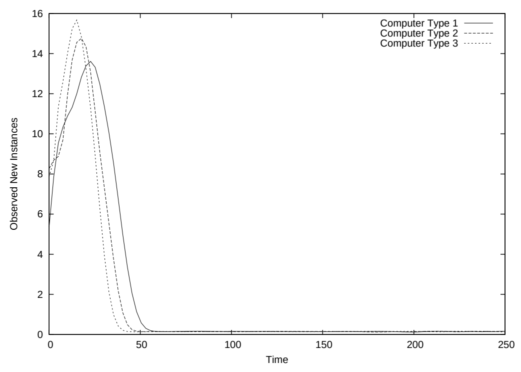

Fig. 2. New observed and learnt program instances

A common feature that all three computers show, according to Figure 3 is that they have more problems classifying the instances at the beginning of the experiment. This makes sense as the system is just starting to learn.

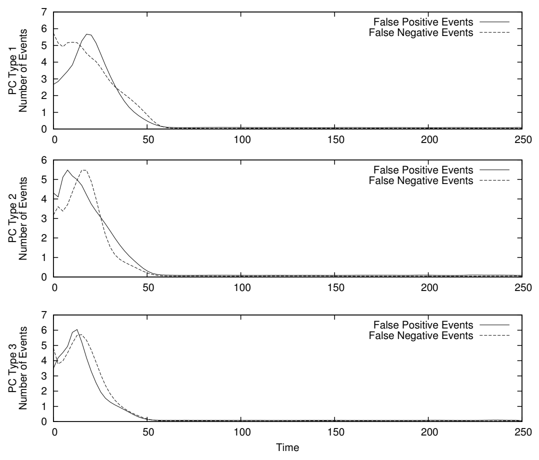

Fig. 3. Observed false positive and negative classifications

Another factor that was measured in order to see the self-protection capabilities of the proposed architecture was to evaluate the ratio between protection actions ($\mathcal{P}$) and discovered threats ($\mathcal{T}$); in other words, the result of $\frac{\mathcal{P}}{\mathcal{T}}$. Depending on the number that is obtained after that evaluation, on can determine how well the learning mechanism is protecting the system. The possible results and their meanings are:

- If $\frac{\mathcal{P}}{\mathcal{T}} = 1$ then every single threat is being tackled by the system, i.e., the system is constantly being protected.

– If $\frac{\mathcal{P}}{\mathcal{T}} > 1$ then more protection acts than needed are taking place, i.e., there are more false positive events.
– If $\frac{\mathcal{P}}{\mathcal{T}} < 1$ then more threats are being considered as safe events, i.e., there are more false negative instances. The system is not being properly protected.

The obtained results following this metric are presented in Figure 4 as an example of the obtained results. This figure shows the idea expressed above plus how distant the protection acts ($\mathcal{P}$) are to discover threats ($\mathcal{T}$). For instance, the top chart in Figure 4 informs that there is an almost perfect behavior since $\mathcal{P}$ events are overlapping the $\mathcal{T}$ ones, with some exceptions at the beginning of the experiment. However, this plot is better understood when one looks at the top chart which depicts the ratio $\frac{\mathcal{P}}{\mathcal{T}}$. As it can be seen, although there are moments when one notices false negative instances, the tendency is to have an almost perfect protection status.

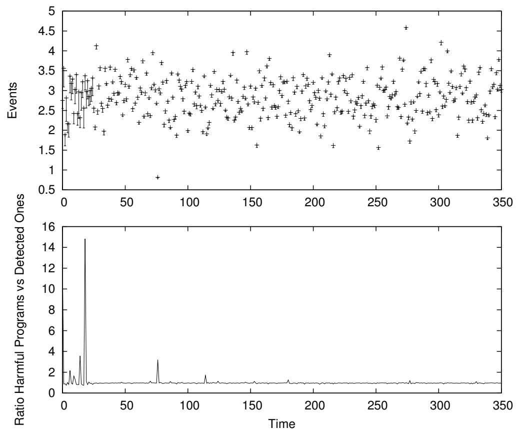

Fig. 4. Relation between threat and protection response for computer type one

4.6 Self-Healing Results

This section presents the last of the wanted self-$\star$ properties, self-healing. The idea here is to see if the system is able to recover by itself from events that have already damaged some part of their normal functioning. The way this is accomplished is by following a set of high-level set rules.

In order to measure this capability in the proposed system, three things were measured, the number of events that required some sort of healing, the number of those events that were successfully healed following the set guidelines and the number of the events that could not be healed. The obtained results are shown in Figure 5.

From this Figure 5 one can infer:

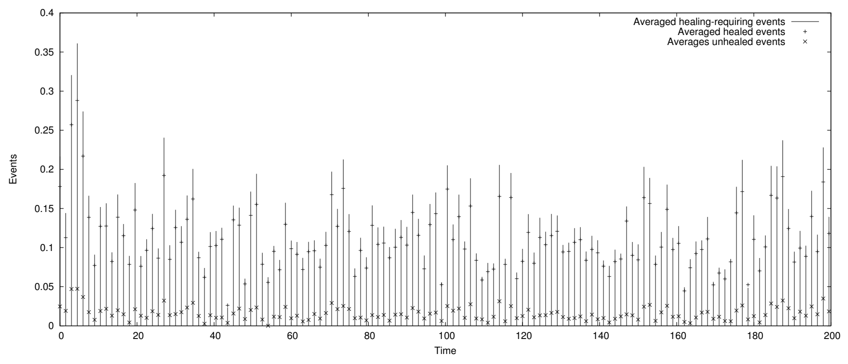

Fig. 5. Observed events where self-healing was needed.

- The number of events that required healing is rather low and one finds the most of such events at the beginning of the simulation (——).
- The number of healed instances is high when compared to the number of observed requiring-healing events (+).
- The number of unhealed events is rather low when compared to the number of observer requiring-healing instances (×).

5 Conclusions

As it can be inferred from the obtained results, the system is capable to protect itself by learning from the HAL program instances that each computer sees. Although in general there are some badly classified events at the beginning of the experimentation, as time goes by the system is capable to distinguish threats from safe programs. In other words, the proposed system shows the required self-$\star$ property.

Also, based on the observed results the systems is able to heal itself. However, there were some events that were not healed and the system was not able to determine how to proceed. Nevertheless, this is a rather good thing as these are the instances that would require more studying in order to provide a solution. Thus, learning from these events in order to be better prepared for future instances. It is this few number of times that IT personnel would be needed to look into and provide a solution. This means a reduction, in general terms, of human involvement as most of the administrative tasks have been already handled by the autonomic service this article proposes; thus, reducing IT troubleshooting.

Acknowledgments

This research was supported by Tecnológico de Monterrey (ITESM), Monterrey Campus, under the Research Chair CAT-144.

References

1. Bar-Yam, Y.: Unifying Principles in Complex Systems. In Roco, M.C., Bainbridge, W.S., eds.: Converging Technologies for Improving Human Performance. Kluwer Academic Publisher (2003) 380–409
2. ComputerHope.com: Computer History Line. `http://www.computerhope.com/history/` (2008) Web Page.
3. Horn, P.: Autonomic Computing: IBM's Perspective on the State of Information Technology. Manifesto (October 2001) IBM Research.
4. IBM Research: Autonomic Computing. `http://www.research.ibm.com/autonomic/` (2008) Web Page.
5. Microsoft Corporation: Dynamic Systems Initiative. `http://www.microsoft.com/business/dsi/` (2008) Web Page.
6. Patel, C.D., Bash, C.E., Belady, C., Stahl, L., Sullivan, D.: Computational Fluid Dynamics Modeling of High Compute Density Data Centers to Assure System Inlet Air Specifications. In: Proceedings of the Pacific Rim ASME International Electronic Packaging Technical Conference and Exhibition (IPACK 2001). (2001)
7. Müller, H.A., O'Brien, L., Klein, M., Wood, B.: Autonomic Computing. Technical Report CMU/SEI-2006-TN-006, Carnegie Mellon University (April 2006)
8. Ganek, A.: Autonomic Computing. `http://www.autonomiccomputing.org/` (2008) Web Page.
9. Agarwala, S., Schwan, K.: SysProf: Online Distributed Behavior Diagnosis through Fine-grain System Monitoring. In: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems, IEEE Computer Society (2006) 8–16
10. Zheng, A.X., Lloyd, J., Brewer, E.: Failure Diagnosis Using Decision Trees. In: Proceedings of the First International Conference on Autonomic Computing, IEEE Computer Society (2004) 36–43
11. Jiang, G., Chen, H., Ungureanu, C., Yoshihira, K.: Multi-resolution Abnormal Trace Detection Using Varied-length N-grams and Automata. In: Proceedings of the Second International Conference on Automatic Computing, IEEE Computer Society (2005) 111–122
12. Lohman, G., Champlin, J., Sohn, P.: Quickly Finding Known Software Problems via Automated Symptom Matching. In: Proceedings of the Second International Conference on Automatic Computing, IEEE Computer Society (2005) 101–110
13. Qin, F., Tucek, J., Sundaresan, J., Zhou, Y.: Rx: Treating Bugs as Allergies - A Safe Method to Survive Software Failures. In: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, ACM Press (2005) 235–248
14. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically Characterizing Large Scale Program Behavior. In: Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2002), ACM Press (2002) 45–57
15. Bottou, L., Bengio, Y.: Convergence Properties of the K-Means Algorithm. In: Advances in Neural Information Processing Systems 7, The MIT Press (1995) 585–592
16. Alpaydin, E.: Introduction to Machine Learning. The MIT Press (2004)