Empirical analysis of self-optimization and fault tolerance

Anshula[1], Krishan Kumar[2]

[1]Assistant Professor, Computer Science Department, Kalindi College
[2]Assistant Professor, Department of Computer Science, Kalindi College, University of Delhi

Abstract: Autonomic Computing is a concept that prepares the systems with an intelligence power to adapt according to the environmental situations. The term is derived from human biology, where the autonomic nervous system monitors central nervous system. Likewise, it's the objective of autonomic computing to endow novel software solutions that can function in an autonomic way, without the need to depend upon complex and centralized management software and without the need of a human operator to take decisions. The autonomic computing architecture lays out a roadmap for the implementation of true Self-Managing software systems. This paper provides a thorough picture of autonomic computing systems, their characteristics, their architecture, issues, and challenges.

Keywords: Autonomic Computing, Self-Management, Self-Optimization, Fault-Tolerance.

I. INTRODUCTION

Autonomic computing (AC) is generally considered to be a term first used by IBM in 2001 to describe computing systems that are said to be self-managing. The past toward AC may be traced back to the work on artificial intelligence (AI), artificial neural networks, robotics, expert systems, intelligent systems, software agents, and cognitive informatics [1]. When reviewing the current state-of-the art in autonomic systems, the concept of self management usually groups into having four basic properties: self-configuration, self-optimization, self-healing and self-protection. Here is a brief description of these properties:

1. ***Self-configuration***: an autonomic computing system configures itself according to high-level goals, i.e. by specifying what is desired, not necessarily how to accomplish it. This can mean being able to install itself based on the needs of a given platform and the user.[1]
2. ***Self-optimization:*** an autonomic computing system optimises its use of resources. It may decide to initiate a change to the system *proactively* (as opposed to reactive behaviour) in an attempt to improve performance.
3. ***Self-healing:*** an autonomic computing system detects and diagnoses problems. What kinds of problems are detected can be interpreted broadly: they can be as low-level as a bit-error in a memory chip (hardware failure) or as high-level as an erroneous entry in a directory service (software problem). If possible, it should attempt to fix the problem, for example by switching to a redundant component or by downloading and installing software updates. However, it is important that as a result of the healing process the system is not further harmed, for example by the introduction of new bugs or the loss of vital system settings. Fault-tolerance is an important aspect of self-healing. Typically, an autonomic system is said to be reactive to failures or early signs of a possible failure.
4. ***Self-protection:*** an autonomic system protects itself from malicious attacks but also from end users who inadvertently make software changes, e.g. by deleting an important file. The system autonomously tunes itself to achieve security, privacy and data protection. Thus, security is an important aspect of self-protection, not just in software, but also in hardware (e.g. TCPA – The Trusted Computing Platform Alliance). A system may also be able to anticipate security breaches and prevent them from occurring in the first place. Self-management requires that a system monitor its components (internal knowledge) and its environment (external knowledge), so that it can adapt to changes that may occur, which may be known changes or unexpected changes where a certain amount of artificial intelligence may be required. However, there is no agreed definition of what an Autonomic system is, their evaluation and moreover comparison, is difficult.

The structure of this paper is as follows. Initially, in section 2 we started with architecture of Autonomic computing to attempt to build a map of the subject. To this end we provide an introduction to the concepts of Autonomic Computing and describe some research that is taking place in various fields of computing and some achievements that have already been made, section 3. We concentrate on research in the field of software engineering and describe projects that focus on adding

IJRRA

© 2016 IJRRA All Rights Reserved

autonomic behaviour to software systems. In section 4 we had discussed challenges in autonomic computing field and finally section 5 includes conclusion and future work.

II. ARCHITECTURE

Generic AC Architecture

The design of technical systems usually focuses on the intended functionality of the system and often obeys the "design follows function" principle. Consequently, a system is organized into components that implement the application specific functions. Such a system is then embedded into some runtime environment that deals with execution failures and captures exceptions. We believe that such a design cannot satisfy the requirements of AC systems and therefore introduce a generic architecture that introduces system components not at the level of application-specific functionalities, but at the level of functionalities derived from the key features of AC systems, see Figure 1[7].

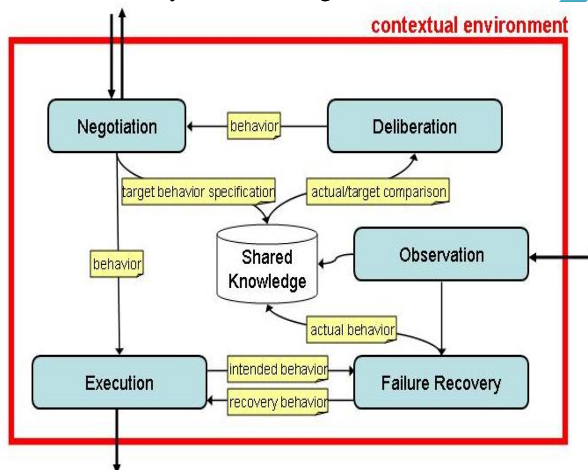

Figure 1: A generic AC architecture

Each AC system is situated in some environment or context. The interaction between the system and its environment occurs through three system components: *negotiation*, *execution*, and *observation[7]*.

The *negotiation* component has a *two-way* interaction with the environment that allows the system to receive requirements from the environment, negotiate the fulfillment of the requested requirements, make itself known to other systems, or communicate its own requirements to other AC systems it is aware of. The main purpose of this component is to receive and actively construct a *target behavior specification* based on its interaction with the environment. This target behavior specification is added to the shared knowledge of the system components. Our architecture highly abstracts from the knowledge contents and format, and the sharing mechanisms between the various AC system components. We only assert that the knowledge base contains a representation of the actual system behavior, the system itself and the environment as perceived by the system. When a new target behavior is added to the shared knowledge, which differs from the actual behavior, a *deliberation* process is triggered that will produce a new behavior. The *deliberation* process sends the new behavior to the *negotiation* component that decides whether this behavior should be executed. The decision can for example be based on whether new requirements have been received that make the behavior already obsolete. The *execution* component has a *one-way output* interaction with the environment to execute any behavior that was determined by the *deliberation* component and forwarded by the *negotiation* component. The *execution* component concentrates solely on executing the behavior in a specific environment, e.g. on expanding high-level action descriptions in sequences of lower-level system commands. The *observation* component has a *one-way input* interaction to receive status information from the environment. The component observes the effect of what the *execution* component is executing without knowing what was actually executed. It adds its observations to the shared knowledge and produces a representation of its observations for analysis by the *failure recovery* process. Limiting the interaction between the AC system and its environment helps to address the key factors of *self-protection* and *hidden complexity*. A system with a controlled interaction is less vulnerable to attacks and hides its internal complexity by exposing only clearly defined interfaces to its environment. The types of interaction we introduced (one-way, two-way) emphasize the predominant, not necessarily the only flow of information.

Two components that do not interact directly with the environment occur in this architecture: *deliberation* and *failure recovery*. As discussed briefly above, the *deliberation* component computes new behaviors for the AC system and encapsulates the "normal" application-specific functional components. It is responsible for fulfilling the key factors of *self-adaptivity* and *self-optimization*. Two major fields of AI will play a dominant role in the development of *deliberation* components: machine learning and AI planning. The *failure recovery* component adds *self-healing* and *self-protection* capability to the AC system. Interestingly, it does not interact directly with the environment, but interacts with the *execution* and *observation* components only. The reason for this design principle lies again in the need to reduce the complexity of the system and enhance its robustness at the same time. The *failure recovery* receives information about the intended behavior of the system from the *execution*, i.e., the *execution* component tells it, for example, what action or command it intends to execute next. This information is used by the *failure recovery* to build an internal expectation of what will happen next in the system

© 2016 IJRRA All Rights Reserved

environment. The *observation* component tells the *failure recovery* what it actually observed happening in the environment. As *execution* and *observation* are completely decoupled in this architecture, they cannot inadvertently influence each other.

The *failure recovery* analyzes the deviations between the intended and the independently observed changes occurring in the environment. For minimal deviations (that need to be precisely defined when implementing this architecture), it computes simple *recovery behaviors* that it sends to the *execution* component for immediate recovery. If greater deviations occur, it updates the shared knowledge with a new actual behavior. This will trigger an actual/target comparison and a new *deliberation* process that may lead to the replacement of the behavior in the *execution* component.

A particular AC system will be based on a sophisticated implementation of the generic architecture. In particular, the sharing of knowledge or information between the various components will usually distinguish between *globally* shared knowledge between all components and *locally* shared knowledge between only selected components. Furthermore, we can expect to see more than one instance of each component or complex components that are AC systems themselves. In particular, the *deliberation* component will probably involve a hierarchical decomposition into application specific functional components, which is already common in realistic application systems. *Self-configuring* AC systems can be expected to involve several deliberation components— specialized in computing system behaviors or computing new system configurations. We regard this architecture more in the sense of a general design principle that will always require refinements and even modifications when instantiated for a particular IT application.

III. PICTURE OF RELATED WORK DONE:

On March 8, 2001, Paul Horn presented importance of these systems by introducing Autonomic Computing Systems (ACSs) to the National Academy of Engineering at Harvard University [4]. Some benefits of autonomic computing include reduction of costs and errors, improvement of services and reduction of complexity. We are going to picture these issued in more depth in this paper. Many researchers have studied this subject since 2001. Their studies have been categorized as follows [2]:

- **Architecture and environment for ACSs:** S. White in, and R. Sterritt and D. Bustard in xADL 2.0 Homepage (http://www.isr.uci.edu/projects/xarchuci/) have described some general architecture for ACSs and their necessary elements called autonomic elements.
- **Studying criteria for evaluating ACSs:** J. A. McCann and M. C. Huebscher in [1] have proposed some metrics to evaluate ACSs like cost and adaptability. Some performance factors such as security and availability have been discussed by others.
- **ACS properties:** These are self-optimization [10], self configuration [9], self-healing, and self-protection. Of course, the IBM Group has stated a general schema for ACSs and their characteristics.
- **Evaluation ACS from software engineering vision:** P. Leaney, A. Mac Arthur, and J. Leaney [3] have established the role of autonomic computing in developing software projects.
- **Challenges in ACSs:** J. O. Kephart and many researches [4] have been done in this context.
- **AC Products:** Different projects and products have been developed in both by the industry and the academic. M. Salehie and L. Tahvildari have outlined some of these products in [10].

From another view, researches carried out in this field can be categorized in two groups as the follows:

- **Group 1:** Researches which describe technologies related to autonomic computing.
- **Group 2:** Researches which attempt to develop autonomic computing as a unified project. However, the lake of appropriate tools for managing the complexities in large scale distributed systems has encouraged researchers to designing and implementing ACSs features.

Table 1: Research Reviewed

Group	Domain	Main characteristics	Refs
Multi-agent systems			
Kuo-Ming, James, Norman	Framework for multi-agent systems	Communication middleware based on CORBA for monitoring and cooperation	[4]
Sterritt, Bustard	Autonomic components	Heartbeat or pulse monitor for monitoring	[6]
Georgiadis, Magee, Kramer	Architectural constraints for self-organising components	Self-organising components with a global view expressed as architecture description	[7]
Kumar, Cohen	Adaptive Agent Architecture	Broker agents used as to provide fault tolerance to overlying problem-solving agents.	[8]

© 2016 IJRRA All Rights Reserved

IJRRA

Bigus et al	ABLE agent toolkit	Framework for building multi-agent systems. Working on including autonomic agents.	[9]
Architecture design-based autonomic systems			
Garlan, Schmerl	Architecture model based adaptation for autonomic systems	Probes, gauges for monitoring running system, architecture manager implements adaptive behaviour, based on architecture-model of system.	[1] [3]
de Lemos, Fiadeiro	Architecture for fault tolerance in adaptive systems	Components considered as black-boxes.	[10]
Dashofy, van der Hoek, Taylor	Framework for architecture-based adaptive systems	xADL 2.0 architecture description language, c2.fw development framework.	[11]
Valetto, Kaiser	Adding autonomic behaviour to existing systems	Autonomic behaviour as a distributed multiagent infrastructure called Workflakes.	[12]
Hot swapping components			
G. S. Blair et al.	Reflective middleware	OpenORB, reflective middleware for self healing systems.	[16]
Rutherford et al.	Reconfiguration in EJB model	BARK tool as an extension to EJB to support component replacement	[13]
Whisnant, Kalbarczyk, Iyer	Model for reconfigurable software	Adaptivity through replacement of bindings between operations and invoked code blocks.	[14]
Appavoo et al	Hot-swapping at OS level	High-performance hot-swapping of fine-grained components in K42 OS.	[15]
Kon, Campbell et al.	Reflective middleware	Dynamic TAO, a middleware for dynamically reconfigurable software	[16]

IV. AUTONOMIC COMPUTING CHALLENGES:

Since autonomic computing is a new concept in large scale heterogeneous systems, there are different challenges and issues[2]. Some of them have been explained in the following:

A. Architecture Challenge: Relationships among AEs have a key role in implementing self-management. These relationships have a life cycle consisting of specification, location, negotiation, provision, operation, and termination stages. Each stage has its own challenges. Expressing the set of output services that an AE can perform and the set of input services that it requires in a standard form, as well as establishing the syntax and semantics of standard services for AEs, can be a challenge in specification. As an AE must dynamically locate input services that it needs and other elements that need its output services must dynamically locate this element with looking it up, AE reliability can be a research area in location stage. AEs also need protocols and strategies to establish rules of negotiation and to manage the flow of messages among the negotiators. One of challenges is for the designer to develop and analyze negotiation algorithms and protocols, then determine which negotiation algorithm can be effective. Automated provision can also be a research area for next stage. After agreement, the AMs of both AEs control the operation. If the agreement is violated, different solutions can be introduced. This can be a research area. Finally, after both AEs agree to terminate the negotiated agreement, the procedure should be clarified.

B. Learning and Optimization Theory: How can we transfer the management system knowledge from human experts to ACSs? The master idea is that by observing that how several human experts solve a problem on different systems and by using traces of their activities, a robust learning procedure can be created. This procedure can automatically perform the same task on a new system. Of course, facilitating the knowledge acquisition from the human experts and producing systems that include this knowledge can be a challenge. One of the reasons for the success of ACSs is their ability to manage themselves and react to changes. In short, in sophisticated autonomic systems, individual components that interact with each other, must adapt in a dynamic environment and learn to solve problems based on their past experiences. Optimization can be a challenge too, because in such systems, adaptation changes behavior of agents to reach optimization. The optimization is examined at AE level.

C. Conceptual Challenges: Conceptual research issues and challenges include (1) defining appropriate abstractions and models for specifying, understanding, controlling, and implementing autonomic behaviors; (2) adapting classical models and theories for machine learning, optimization and control to dynamic and multi agent system; (3) providing effective models for

© 2016 IJRRA All Rights Reserved

negotiation that autonomic elements can use to establish multilateral relationships among themselves; and (4) designing statistical models of large networked systems that will let autonomic elements or systems detect or predict overall problems from a stream of sensor data from individual devices.

D. Robustness: There are many meanings for robustness. Robustness has been served in various sciences and systems such as ecology, engineering, and social systems. We can interpret it as stability, reliability, survivability, and fault-tolerance, although it does not mean all of these. Robustness is the ability of a system to maintain its functions in an active state, and persist when changes occur in internal structure of the system or external environment. Some often mistake it with stability. Although both stability and robustness focus on persistence, robustness is broader than stability. It is possible that components of a system are not themselves robust, but interconnections among them make robustness at the system level. A robust system can perform multiple functionalities for resistance, without change in the structure. With the design of instructions that permit systems to preserve their identity even when they are disrupted, the robustness in systems can be increased. Robustness is one of grand scientific challenges which can be also examined in programming.

E. Middleware Challenges: The primary middleware level research challenge is providing the core services required to realize autonomic behaviors in a robust, reliable and scalable manner, in spite of the dynamism and uncertainty of the system and the application. These include discovery, messaging, security, privacy, trust, etc. Autonomic systems/applications will require autonomic elements to identify themselves, discover and verify the identities of other entities of interest, dynamically establish relationships with these entities, and to interact in a secure manner.

V. CONCLUSION

In this paper, a survey of autonomic computing systems and their importance was presented. As future researches, the following topics can be proposed in autonomic distributed computing domain:

1) Performance evaluation of applying the autonomic behavior in a distributed computing system model.

2) Designing an autonomic manager in multi-layer P2P form, so that autonomic behavior and management information as a knowledge base are stored in separated layers.

3) Studying languages which develop autonomic management behavior in a distributed computing environment.

4) Implementing a self-healing system in a virtual organization wherein some partners may fail.

VI. REFERENCES

[1] Julie A. McCann, Markus Huebscher "*Evaluation issues in Autonomic Computing "*Grid and Cooperative Computing - GCC 2004 Workshops, Lecture Notes in Computer Science Volume 3252, 2004, pp 597-608.

[2] Mohammad Reza Nami, Koen Bertels, A Survey of Autonomic Computing Systems, Third International Conference on Autonomic and Autonomous Systems, 2007. ICAS07, IEEE.

[3] Garlan D., B. Schmerl. *Model-based Adaptation for Self-Healing Systems*. Proceedings of the first workshop on Self-healing systems.

[4] J. O. Kephart. Research challenges of autonomic computing. In *Proceedings of the 27th International Conference on Software Engineering*, pages 15–22, May 2005

[5] Kuo-Ming C., James A., Norman P. *A Framework for Intelligent Agents within Effective Concurrent Design*. The Sixth International Conference on Computer Supported Cooperative Work in Design.

[6] Sterritt R., Bustard D.. *Towards an Autonomic Computing Environment*. University of Ulster, Northern Ireland.

[7] Georgiadis I., Magee J., Kramer J.. *Self-Organising Software Architectures for Distributed Systems*. ACM, Proceedings of the first workshop on Self-healing systems, November 2002.

[8] Kumar S.,. Cohen P. R. *Towards a Fault-Tolerant Multi-Agent System Architecture*.

[9] Bigus J. P. et al. *ABLE: A toolkit for building multiagent autonomic systems.* IBM Systems Journal, Vol. 41.

[10] de Lemos R., Fiadeiro J. L.. *An Architectural Support for Self-Adaptive Software for Treating Faults.*

[11] Dashofy E. M., van der Hoek A., Taylor R.N.. *An Infrastructure for the Rapid Development of XML-based Architecture Description Languages*. Proceedings Of the 24th International Conference on Software Engineering (ICSE2002), Orlando, Florida, May 2002.

[12] Valetto G., Kaiser G.. *Combining Mobile Agents and Process-based Coordination to Achieve Software Adaptation.* Columbia University.

[13] Rutherford M. J., Anderson K., Carzaniga A., Heimbigner D., Wolf A. L., *Reconfiguration in the Enterprise Javabean Component Model.* Proceedings of the IFIP/ACM Working Conference on Component Deployment, Berlin.

[14] Whisnant K., Kalbarczyk Z. T., Iyer R. K., *A system model for dynamically reconfigurable software*. IBM Systems Journal, Vol. 42.

[15] Appavoo J. et al. *Enabling autonomic behaviour in systems software with hot swapping*. IBM Systems Journal, Vol. 42.

[16] Kon F., Campbell R. H., Mickunas M. D.,. Nahrstedt, K Ballesteros F. J.. *2K: A Distributed Operating System for Dynamic Heterogeneous*

© 2016 IJRRA All Rights Reserved

IJRRA

Environments. IEEE, Proceedings of The Ninth International Symposium on High-Performance Distributed Computing, 1-4 Aug. 2000, Pages 201-208.
[17] Garlan D., Schmerl B., Chan J.. *Using Gauges for Architecture-Based Monitoring and Adaptation* Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia.
[18] Wooldridge M., *An Introduction to MultiAgent Systems.* John Wiley & Sons Ltd.

IJRRA