System Architecture of an Autonomic Element

M. Muztaba Fuad[†] and Michael J. Oudshoorn
Department of Computer Science
Montana State University
Bozeman, MT 59717, USA
E-mail: {fuad, michael}@cs.montana.edu

Abstract

An autonomic element is the fundamental building block of any autonomic system. Although different aspects of autonomic computing are explored in isolation, the structural operation of an autonomic element has not been completely modeled. The standard definition for an autonomic element does not provide an architectural blueprint and several proprietary designs have been proposed that are not interoperable with each other. This paper presents an engineering perspective of building a domain independent autonomic element. We believe that architectural choices have a profound effect on the capabilities of any autonomic system and affect many of the design decisions during its implementation. Therefore, it is important to have a well defined model of the basic building block to develop autonomic systems. The architectural design presented is self regulating and uses standard object oriented primitives to make it easy to develop and implement.

1. Introduction

Autonomic computing is a new paradigm where computing systems possess the capability of self-management. As computing systems continue to evolve to meet ever changing needs, dynamic computing systems which are self-manageable give business and scientific organizations the ability to automate their business and scientific tasks. Unfortunately many programmers lack the skill of developing such self-managed systems due to the added programming complexity. It is tremendously beneficial to programmers if they do not have to worry about programming complexity of implementing different issues related to self management. A computing environment that hides the complexity of programming and addresses such self management issues automatically is certainly desirable. However, this sort of support should be provided without the need for programmer intervention or consideration other than the generation of policies to guide the execution process. The provision of all such self management features is an unresolved problem [1] and there are no truly autonomic computing systems in existence at this time.

Although every aspect of autonomic computing is a significant research challenge [2], we need to start designing and developing tools and methods towards the goal of autonomic computing [3]. While different aspects of autonomic computing have been explored in isolation, there is a lack of effective autonomic infrastructure that enables programmers to develop and integrate their programs using autonomic primitives. Although there is general agreement on the structure of the autonomic element, there are no general implementation details of the internal architecture of the autonomic component [3]. The lack of a common implementation of this basic autonomic component hinders the interactions among different self-managed technologies. Since most other research employs proprietary techniques to develop such components, it is difficult for programmers to incorporate autonomic properties in the design and development of such systems. The programmer is still more concerned with learning and implementing such proprietary techniques rather than concentrating on the actual problem in hand.

This paper presents an architectural design of the standard autonomic element. The authors believe that architectural choices have a profound effect on the capabilities of any autonomic system and affect many of the design decisions during its implementation. Having an open and flexible structure for the most basic component of any autonomic system alleviates programmers from the complexities associated with designing, developing, integrating and managing autonomic systems. Design goals of the proposed structure of the autonomic element are as follows:

1. It should be transparent to use and programmers should "program as usual" with minimal constraints.
2. It should employ open standards and common metaphors to develop the autonomic element.

[†] *Supported partially by Montana NASA Grant Consortium, Award No. MSGC-426080/07*

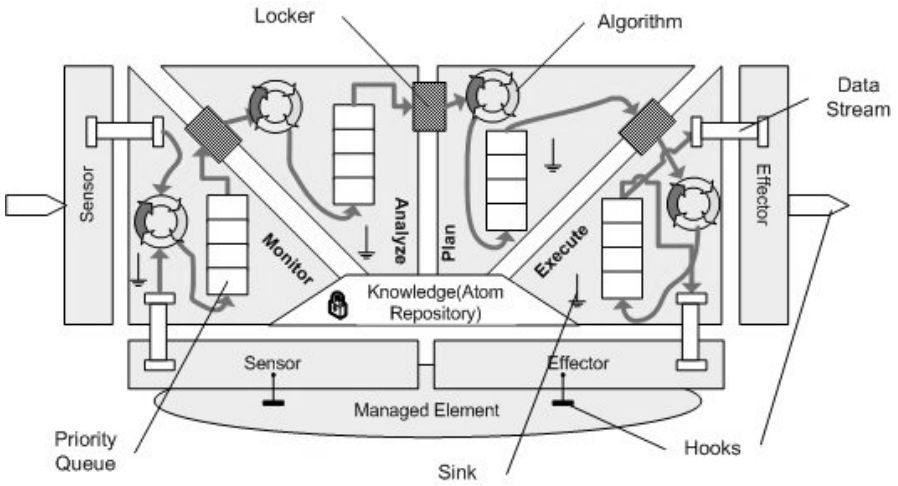

Figure 1. Internal architecture of the autonomic element.

3. It should be lightweight and should not consume system resources unnecessarily.

This paper presents a Java-based autonomic element design that greatly simplifies prototyping of self-management techniques. Although this design is based on Java, any programming language that supports standard object oriented metaphors can be easily used to develop the same model of autonomic element. The design allows different techniques to be compared and helps investigate the interaction among different techniques. The implementation is itself incorporated into a Java-based autonomic infrastructure [4, 5] to assist the rapid deployment of user applications and easy-to-use transparent, self-management of distributed systems.

2. Autonomic element architecture

Autonomic elements are the heart of any autonomic system. An autonomic element is described by two distinct parts. The *autonomic manager* provides the functional abilities of the element and the *managed element* is the entity that the autonomic manager is monitoring and controlling. All autonomic elements have a *control loop* that dictates the work flow of the different sub-components of the autonomic element. Figure 1 shows the design for the autonomic element.

We envision an autonomic element as a multi-threaded server (daemon) having multiple components (*monitor*, *analyze*, *plan* and *execute*) and interfaces (*sensors* and *effectors*) running concurrently for better CPU utilization. Having a sequential control loop can block the execution of the whole element when one component is blocked for any reason and therefore may have an adverse performance effect. Another alternative is to design the sub-components as threaded objects and schedule them according to the control loop. As Java does not support language level thread control (*stop*, *suspend* etc. are deprecated), it is not possible to write a scheduler for Java threads without sacrificing performance. Writing such a scheduler is a complicated programming task and makes the autonomic element bulky and consumes more CPU power for scheduling. Instead, this paper proposes each sub-component as a threaded object and delegates the scheduling decisions to the underlying JVM. The benefit of having a threaded model is that, algorithms which are responsible for different aspects of self-* can easily be incorporated as a child thread inside any of these parent threads denoting the four major components of the autonomic element. To have the same semantic functionality as the control loop, this design uses semaphores in strategic places to order the execution of the sub components and synchronizes the threads when accessing shared data. This way, the control loop in this design is non-blocking due to concurrent execution of the individual sub-components. This allows faster execution of autonomic behaviors but introduces the following challenges:

1. *Asynchronous execution*: Since each component is running asynchronously, it has to be guarantied that no internal messages are passed inadvertently to a different component.
2. *Deadlock and starvation*: Care must be taken during development of shared data structures and data components such that access to those shared resources are mutually exclusive.

There are threads for environment interfaces (sensors and effectors) and for the main components (monitor, analyze, etc.) of the autonomic element. Prior to describing the internal architecture of the components, this paper describes the data structures used to create the internal formation of the autonomic element:

- **Data atoms:** All internal communication is passed in a standard format, named *atoms*. Atoms normally travel between different components in the direction of the control flow. An atom is essentially a collection of XML tokens which can be interpreted by all the sub

Figure 2. Structure of a data atom.

components of the autonomic element. The main reason for selecting XML to represent internal data is that it gives flexibility for future improvement and extension. Since XML is machine agnostic, autonomic elements developed in other programming languages can be incorporated easily in an existing system. Figure 2 shows the internal structure of a data atom. Data atoms have the capability of chaining *n* atoms together to form a single atom. Atoms need to chain to each other when the occurrence of certain events requires a steady supply of atoms which have some form of common relationship. However, care needs to be taken with respect to the upper bound for *n*, as making an excessively sized data atom can slow down the system. The self optimization aspect of the autonomic element can dynamically change the value for *n* by monitoring the element over long runs.

- **Priority queue:** Each of the components inside the autonomic element has a buffer implemented as a priority queue. As data atoms are flowing across the control loop, they are stored temporarily inside the components for processing. Different components can create new data atoms after processing existing data atoms and can pass those to the next component. This design permits at most one atom to pass between components at any given time. This ensures a consistent flow of information among the components and ensures that individual components are not overwhelmed. Therefore, atoms are sorted and stored in the queue based on their priority and are processed according to the highest priority. Normally the size of the queue starts at a predefined size. However, over time it can grow or shrink depending on the flow of atoms in the system.
- **Locker:** This is a special data structure that resembles an operating system semaphore with added functionality. Lockers are placed between two sub-components in the autonomic element to transfer atoms between them. The main responsibility of the locker is to receive an atom when it is empty and notify the destination component of an incoming atom. There are two main reasons for passing atoms via the locker. Firstly, to avoid any deadlock situation between two concurrent components during data transfer. Secondly, to avoid wasting any processing time of the components unnecessarily. Instead of the components polling each other for atoms, the locker notifies the corresponding component if there is a new atom awaiting processing. Polling introduces synchronization and deadlock issues and a component is expected to spend most of its allotted schedule time polling for incoming atoms. Having all atoms transfer through the locker improves the response time of the components and does not block the receiver if the sender is processing slower than itself for any particular reason. All lockers fulfill following two conditions:
  1. The locker is always unidirectional. Therefore, only one component can *push* an atom in a locker and only the adjacent component can *pop* it. To loop an atom back to a previous component, it has to follow the control loop and appear in the queue of the destination component. It may be possible to have a bi-directional locker between components; however, care should be taken to avoid any deadlock situation. We believe that a bi-directional locker does not increase performance, rather complicates the whole design. In the current version of the design we are not considering this aspect, but with extended behavior modeling and verification, this could possibly be implemented.
  2. As with semaphores, the *push* and *pop* operations need to be atomic. Since Java does not provide any language level abstraction for atomic methods, we used the atomic reference object implementation in *java.util.concurrent.atomicReference* of the Java 1.5 API to implement those methods. So the methods have an atomic object reference, where the data atom can be pushed in or popped out atomically. The use of this class of variable offers higher performance than would be available by using standard synchronization techniques.
- **Data streams:** These are one-way dedicated communication channels between two entities. Only one thread can write into it and another can read from it. To avoid any deadlock situation, all read and write operations are forced to be atomic as described earlier. Data streams are used where no intermediate processing is required, such as in the sensors and effectors to provide rapid response.
- **Hooks:** This is the collection of data structures and methods that allow the sensors and effectors to actually interact with the environment or the managed element. The representation for such hooks may not be universal because of the diversity of applications and environments. The environmental hooks can be as simple as network level sockets with assigned ports for receiving incoming atoms and sending outgoing atoms. Hooks for managed elements may

force programming the managed elements through predefined interfaces such that at run time there are predefined methods that can be called by the sensors or effectors. However, this places extra complexity on the programmer to preserve self-managing aspects. The development of such hooks is domain specific and should be addressed on a per system/application basis. For our application domain [4, 5], we inject byte code segments at strategic places in the distributed Java objects which are treated as the managed element. This sort of code injection [5] is completely transparent to the programmer and performed during preprocessing and runtime and before deployment with an autonomic element.

- **Atom repository:** Any processed atoms that the system wants to store for future use are stored in the atom repository. This is identical to the knowledge part of the autonomic element. Along with storing atoms, it can also store rules and policies regarding different functional aspects of the autonomic element. Finding a good knowledge representation is a separate research challenge. This design used ACPL [7] to represent rules and policies as it provides a user friendly form of policy definition, policy management and different tools and an API to work with policies. Since ACPL is based on XML, this permits atoms to be seamlessly incorporated into the repository. Although multiple components can read simultaneously from the repository, only one component can write in it at any time. This is to ensure consistency of the data in the repository.

The functionalities of different components of the autonomic element and its structure is now considered in more detail:

- **Sensors and Effectors**: There are two sensors in every autonomic element. One is responsible for interfacing with the environment and other is necessary to interface with the managed element. The common responsibility is to acquire runtime information and incoming messages from the managed element or environment respectively. The sensors are not heavy weight processes as they only check the validity and format of any incoming message and forward them to the monitor component. On the other hand, effectors receive data from the *execute* component and verify its format and then either forward it to the environment or invoke appropriate methods inside the managed element to effect its execution.
- **Sub-components:** The four basic components in the autonomic element are implemented using a technique described in [8]. Explicitly, the functional part (algorithm) of the element is implemented through a common interface so that at runtime it can be updated, if necessary, to fulfill user policy and goals. So the functional code resides inside a class (*task.java*) which implements a common interface named (*taskInterface.java)*. At runtime, the old *task* class is replaced by any newer version of its implementation. This allows *adaptive composition* of the component by adding, deleting or modifying the algorithm of each component at run-time. Usually each components takes atoms from its input source (locker or sensors) and processes it and puts it in the priority queue. At each scheduling cycle, it checks whether there is anything in the queue which needs further processing or needs to be passed to the next stage of execution. If the corresponding locker or effector is free, it then pushes the top most data atom to the corresponding destination. Data atoms may not propagate all the way to the last component if it is internal to the autonomic element or there is no further processing available or necessary for a particular data atom. Therefore all the sub-components have data sink points, which is basically an object nullifier. The garbage collector is called to free up the memory of those nullified objects. Since the Java garbage collector blocks all threads during garbage collection, calling it frequently decreases the performance of the autonomic element. So different heuristics (such as if number of nullified objects reached a maximum threshold or the queue size reached a maximum threshold) are used to decide when to call the garbage collector.

3. Existing work

There have been several autonomic system projects that describe autonomic infrastructure and define the architectural aspects of an autonomic element. However these research projects address the issue of designing an autonomic element with different objectives than ours. The goal of this research is to make the autonomic elements simple to program and use and which in turn is universal enough that it can be used in most cases. Following is the list of research works and systems that describe architectural aspect of the autonomic element:

a. [9] has the same objective as this paper, however, they did not provide an architectural design.
b. [10] provides some good ideas on how to design autonomic elements and identifies the issues that should be addressed. However, they did not provide any architectural design.
c. [10, 11] presents a prototype autonomic system without defining the internal workings of the autonomic elements.
d. [12] proposed design using proprietary techniques that add additional complexity and a steep learning curve for the programmer.
e. [13] provides tools and an API for monitoring, analysis, planning, and executing autonomic

applications. Although these tools provide some inter-dependent functionality for an autonomic environment, they however, do not address the development of autonomic elements.

f. Existing research on multi-agent systems is a rich source of good ideas about system-level architectures and software engineering practices [15, 16] for autonomic computing. System such as Unity [16] uses multi-agent paradigms for developing autonomic systems. However, these approaches lack common agreement on the exact design of most agents. The autonomic computing community could certainly benefit from their experiences.

4. Conclusions and future work

Autonomic elements have been implemented following the proposed design and are being incorporated into a distributed autonomic system [5]. Initial analysis of runtime behavior and profiling shows that the design is sound and can work smoothly without causing any deadlock or producing extensive overhead.

A testbed environment is under development where simulation of the working of such autonomic element in a real environment can be performed. That will allow us to see how components interact with each other and how much time they really spend for their own management. Future works include the implementation of the same design with other programming languages and incorporation in a system where autonomic elements from different programming language are in play. Once a stable and robust implementation is achieved, producing an API for the autonomic element would certainly help other programmers to design autonomic element in a consistent and standard form.

Current research on autonomic computing suffers significantly from the lack of a common definition of the basic autonomic entities. Defining and developing the basic autonomic entities and making it publicly available would greatly simplify prototyping of different self-management techniques and will allow these techniques to be compared in a common base and will allow investigation into their interaction. One of the basic building blocks of any autonomic system is the autonomic element. This paper presents a self regulating design of an autonomic element and describes how the different subcomponents should interact and process information. The goal is to provide a design which is acceptable in most domains and which can be implemented in any domain with minor modifications.

5. References

[1] M. Salehie and L. Tahvildari, "Autonomic Computing: Emerging Trends and Open Problems", *SIGSOFT Software Engineering Notes*, Vol. 30, No. 4. pp. 1-7, July 2005.

[2] J. O. Kephart, "Research Challenges of Autonomic Computing", *The 27th Int. Conf. on Software Engineering*, USA, pp. 15-22, 2005.

[3] J. O. Kephart and D.M. Chess, "The Vision of Autonomic Computing", *IEEE Computer*, Vol. 36, No. 1, pp.41–52, 2003.

[4] M. J. Oudshoorn, M. M. Fuad and D. Deb, "Towards an Automatics Distribution System - issues and challenges", *The Int. Conf. on Parallel and Distributed Computing and Networks*, Austria, pp. 399-404, 2005.

[5] M. M. Fuad and M. J. Oudshoorn, "An Autonomic Architecture for Legacy Systems", *The Third IEEE International Workshop on Engineering of Autonomic Systems (EASe 06)*, Maryland, USA, April 24-28, pp. 79-88, 2006.

[6] M. M. Fuad, D. Deb and M. J. Oudshoorn, "Adding Self-Healing Capabilities into Legacy Object Oriented Applications", *Int. Conf. on Autonomic and Autonomous Systems*, IEEE Press, California, USA, pp. 51-56, July 18-22, 2006.

[7] D. Agrawal, K. W. Lee and J. Lobo, "Policy-Based Management of Networked Computing Systems", *IEEE Communications Magazine*, Vol. 43, No. 10, pp. 69-75, 2005.

[8] A. Orso, A. Rao, M. Harrold. "A Technique for Dynamic Updating of Java Software", *18th IEEE Int. Conf. on Software Maintenance*, pp. 649-658, 2002.

[9] M. Jarrett and R. Seviora, "Constructing an Autonomic Computing Infrastructure Using Cougaar", *The Third IEEE International Workshop on Engineering of Autonomic Systems,* Maryland, USA, April 24-28, pp. 119-128, 2006.

[10] H. Liu and M. Parashar, "Accord: a programming framework for autonomic applications", *IEEE Transactions on Systems, Man and Cybernetics*, Special Issue on Engineering Autonomic Systems, Editors: R. Sterritt and T. Bapty, IEEE Press, pp. 341-352, 2005.

[11] M. Parashar, H. Liu, Z. Li, et. al., "AutoMate: Enabling Autonomic Grid Applications", *The Journal of Networks, Software Tools, and Applications, Special Issue on Autonomic Computing*, Kluwer Publishers, Vol. 9, No. 1, 2006.

[12] X. Dong; S. Hariri, L. Xue, et. al., "AUTONOMIA: An Autonomic Computing Environment", *The IEEE Int. Performance, Computing, and Communications Conf.*, pp. 61-68, April, 2003.

[13] W. David, W. Levine et al., "A Toolkit for Autonomic Computing", *IBM Developer works Live*, 2005.

[14] N. R. Jennings, "On agent-based software engineering", *Artificial Intelligence*, Vol. 117, No. 2, pp. 277–296, 2000.

[15] N. R. Jennings, "Building complex, distributed systems: the case for an agent-based approach", *Communications of the ACM*, Vol. 44, No. 4, pp.35–41, 2001.

[16] D. M. Chess, A. Segal, I. Whalley, et. al., "Unity: Experiences with a Prototype Autonomic Computing System", *First Int. Conf. on Autonomic Computing*, pp. 140-147, 2004.