# Foundations of Autonomic Computing Development

Sam Lightstone
*Senior Technical Staff Member, Development Manager,*
*DB2 Autonomic Computing Development*
*DB2 Universal Database*
*IBM Canada Ltd., 8200 Warden Avenue*
*Markham, Ontario, Canada L6G 1C7*
*light@ca.ibm.com*

## Abstract

*The complexity of modern middleware, and software solutions, is growing at an exponential rate. Only self-managing, or autonomic computing technology can reasonably stem the confusion this complexity brings to bear on human administrators. While much has been published on "architecture" and "function" for producing such systems, little has been written about the engineering of self-managing systems as a distinct paradigm. In this paper we suggest a straw-man for engineering of autonomic systems that is based on two essential tracks: a set of engineering principles that should guide the planning of autonomic systems and their interfaces and secondly a set of mathematical foundations upon which such systems can best be constructed. These foundational attributes are intended to guide the thinking of R&D organizations pursuing the development of autonomic computing capability. The role of architecture and standards is also discussed, highlighting their role in inter-component management.*

## Introduction

Autonomic Computing [1] has emerged as a paradigm for self managing IT systems to stem the tide of rapidly increasing administration costs in the face of rising IT system complexity [2][3][4][5]. The development of self managing systems posses special challenges to research and development teams. This article is based on part of a Keynote talk given at EASe'06 [20] and explores seven development principles for successful engineering of autonomic computing systems and key foundational techniques currently in use.

Little has been published concerning the unique challenges to development teams working on autonomic computing problems. Autonomic, zero admin, self-managing, embedded, invisible. These are all (somewhat) synonymous adjectives for systems that requires zero or little human administration. It's where the open server market is aggressively moving, particularly for middleware components. Self-managing middleware begins with an application view that drives all subsequent requirements for design, build, deployment, operations and change management. As TCO is dominated by human costs, autonomic computing is the primary path to dramatically reduced TCO.

The timeline of ownership for an IT system is illustrated in Figure 1. We describe the timeline of ownership as having five stages. The ownership process begins with an assessment of requirements for capital investment and capacity planning. In the next phase the system is designed. Thirdly the system is constructed, tested and tuned. Fourthly, the system goes into production and daily tuning and object administration takes place. Finally, in the last stage changes are made affecting the application or the database server directly.

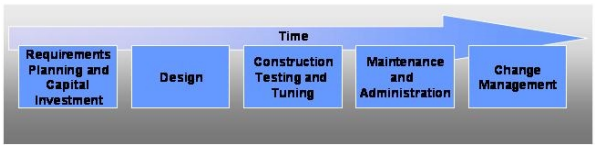


**Figure 1 Administrative timeline of ownership**

A brief analysis of the complexity of modern systems highlights the extent of the problem clearly. In

a brief examination of 3 popular middleware products produced by different companies, we observed the number of configuration/registry parameters ranged from 384 to 1200. These parameters were used to specify everything from memory configuration to connection and process limits. Many of the parameters have a dynamic range of potential values (for example, memory is often configurable in increments of 1, 4, or 1024 kilobytes) having thousands of possible configurable values. The most extreme simplification of the configuration space considers each configuration parameter as having a binary setting (ON/OFF or TRUE/FALSE etc). Using this gross oversimplification, the possible configurations for a product with 384 parameters is $10^{115}$ while a product having 1200 binary parameters would have $10^{361}$ configurations. These numbers are monstrous, far beyond the ability of human beings to assess. It's sobering to compare the complexity of possible configurations of middleware components to another metric: the latest estimates for the number of atoms in the universe. Consider for a moment that the known universe contains several hundred billion galaxies, each with roughly one hundred billions of stars. Astrophysicists estimate the upper bound on the number of atoms in the universe[1] to be $10^{81}$, many orders of magnitude less than the number of ways one can configure the three middleware products we have studied, even with dramatic oversimplifying assumptions.

**Figure 2 A spiral galaxy, with more than one hundred billion stars. The universe contains billions of galaxies**

Clearly, even with gross simplifying assumption, modern middleware has many orders of magnitude (trillions upon trillions) more possible configurations than the number of atoms in the universe. This observation leads to two conclusions:

1. Computer systems of reasonable complexity can never be perfectly optimized.
2. Given that such computer systems can never be perfectly optimized, self managing systems should strive for "adequate" and "near optimal" administration.

These observations give rise to the obvious question "how good is adequate and how near is near optimal?". We believe that in the case of autonomic system the answer to these questions is "as good as most skilled human administrators". This answer flows from the fact that the goal of autonomic computing is the reduction of Total Cost of Ownership (TCO) to reduce the exploding complexity of modern systems. The reduction in TCO provides the business case for the investment in autonomic computing, by avoiding a corresponding explosive growth in labor costs to administer the growing complexity. Since cost of incremental (not current) professional IT labor is the proposed saving it follows that in order to achieve this saving autonomic system must achieve results similar to the incremental human talent that would be their alternative. Much like the universe itself, the development of autonomic computing systems will depend on a broad set of conceptual domains. Autonomic system, by the nature of their vast complexity, cannot be developed using a narrow selection of computer science techniques, such as those provided from a single domain like artificial intelligence. Techniques form control theory, economics, social dynamics, and other fields have already shown their merit. The breadth and challenge of autonomic computing also exacerbates a number of classical pressures on development organizations, such as the difficulty in constraining an engineering R&D team to a solution that is sufficient to satisfy a business requirement, though far from complete.

Through the remainder of this paper we describe a foundation for the development of autonomic computing systems along two lines. First, we introduce a set of software engineering principles, and second present a short survey of foundational techniques that appear to be dominant in the development of autonomic computing capabilities.

## 2. A software engineering foundation

The following seven guiding principles should guide strategic thinking on TCO reduction, and represent the basic thesis of this paper. Each of the seven will be elaborated on below. There are of course many more than 7 design principles that designers and software

[1] Current estimates are in the range of $10^{63}$ to $10^{81}$. We have assumed a worst case here to strengthen the argument.

engineers ought to keep in mind during the development of autonomic computing components and systems. However, seven is in fact a nice round number, and just small enough for most of us to keep in mind. An exhaustive list, while perhaps more satisfying and academically rigorous would not be in fact practical. It's also worth bearing in mind that autonomic computing is to a large part a software oriented discipline which inherits the design goals and requirements of all good software (encapsulation, reliability, reuse, etc). The following are additional software engineering guidelines that appear to be particularly applicable to the development of successful autonomic computing projects.

### #1 Build what users need, not what's cool.

One of the major challenges to development teams building autonomic computing technology is surprisingly that autonomic systems are too much fun. This sounds ridiculous at first glance, and not particularly unique to autonomic computing systems. In fact almost everyone who has worked on autonomic systems (much like cybernetic systems) has faced the excitement and the pull to make the system a little more adaptive, and a little more intelligent. In many cases the added sophistication is not needed, and adds only to code complexity. Moreover, there are many cases within industrial software where entire features appear to have been implemented where a few heuristics would have sufficed.

### #2 Always give the user "an out". Features providing system automation must have an off switch.

There are two issues that motivate this rule. Firstly that despite the best technology, using feedback control systems and wonderful mathematical optimization, there will always be cases where the autonomic technology does not work perfectly. Poor control can occur because of either inadequate modeling or due to software defects. Either way, when the autonomic computing fails, the user must have an option to disable it. This is particularly true for mission critical systems.

The second consideration is one of trust. Trust in self-managing system is the subject of ongoing research. How we can build trust in autonomic technology is important because without trust, however good the technology is, it will no be used. In fact, without the option to disable the technology, many System Administrators and DBA managers will consciously decide not to purchase technology if it can't be trusted and can't be disabled. Peter Coffee wrote an article on this in 2004 speaking directly to this point [7]. He notes:

> *"But our confidence that vendors will do things right must increase before we let them do things without our knowledge or control....*
>
> *Building systems that scrutinize themselves, with mechanisms for graceful degradation or, ideally, for self-repair, is an idea whose time has clearly come. I'd rather have brains in the loop, but perhaps technology is on the verge of being as good—and, certainly, much less expensive."*

### #3 Features must be on by default in order for the majority of users to exploit them.

In a number of informal studies at IBM we have found that the major inhibitor to customer adoption of autononmic computing features is simply lack of awareness. More specifically, the vast majority of customer don't know features exist (any feature, not just the autonomic computing features), and discover them on as is needed basis. For every product there will always be a small subset of power users who have an extraordinarily broad awareness of product features. The power users are in fact the group most aware of autonomic computing features, and also the group least likely to use (and in fairness, least in need of) autonomic computing technology. This poses a serious threat to autonomic computing adoption designed for two audiences: one that desperately needs it and has never heard of it, and another that moderately needs it and doesn't want it. The challenge is not dissimilar to the introduction of the automatic transmission automobile. Fully automatic transmission was introduced in 1939 by Oldsmobile, however it was not until the mid 1950's that automatic transmission automobiles where dominant family cars, a lapse of roughly 15 years. Standard (manual) transmission cars remain popular today despite the higher repair costs and only slightly reduced fuel consumption. The adoption of automatic transmission was likely hindered by similar factors facing autonomic computing: the novices don't know about it and the enthusiast doesn't want it. The solution is to enable, where possible, autonomic computing features by default. This allows the normal user to reap the benefits of the technology without reading every aspect of the product documentation, while still giving the power user a deliberate choice.

### #4 Never force the user to make a choice that your developers could not.

Unfortunately, there are too many cases in the software world where a development team, unable to determine a reasonable setting for a variable that was crucial to system performance opted to make the setting a "user configurable parameter". This happens frequently when the correct value for a variable is "it depends". While development schedules may temporarily prevent the autonomic resolution of these kinds of "it depends" variables, over the long run eliminating these kinds of parameters is a key objective for autonomic systems. The reason is simple: the development team that designed and coded the system didn't know how to set the parameter it is certain that the vast majority of end users certainly won't. Foisting the problems of the development team onto unsuspecting users (in this case system administrators) is a losing strategy.

### #5 Autonomic computing technology must work in real world scenarios.

Another negative habit that has become rampant in the industry is the design and evaluation of autonomic solutions around benchmarking systems. Industry standard benchmarks are frequently used to assess the performance or recoverability of systems. The use of benchmarks is, in fact, a reasonable industry strategy. However, the vast majority of these benchmarks are extremely well behaved and non adaptive. Development teams often use benchmark systems to evaluate autonomic features because the benchmark system provides a well understood workload and performance baseline against which to pit the talents of a newly created autonomic feature. However, production systems are notoriously more complex and variable than benchmark systems. As a result, the success of autonomic technology with benchmark systems, while meaningful, is not adequate.

### #6 Never automatically undo or contradict the explicit choices of the administrator / application(s).

Autonomic systems typically execute a cycle of monitoring, analyzing, planning and execution. The analysis and planning will in many cases conclude that a system state change (modification of resource, or system design) should be made which contradicts or replaces the deliberate system choices of a human administrator or system designer. Ideally a perfect autonomic system would always recommend such changes where they were certain to improve on the human choices. In reality there are several serious considerations why overriding the deliberate choices of humans is a march of folly. First, because the quality of autonomic computing technology is not mature enough to ensure the quality of the decision of the autonomic computing system is superior to a deliberate human choice. Second, because the purpose of autonomic computing is the reduce TCO by obviating the incremental cost of administration caused by increasing system complexity. Once a human administrator has made a choice, however suboptimal, the system can be reasonably assumed to be in "acceptable" state in that dimension, and incremental improvement – even dramatic improvements – over the human design are not necessarily needed. Thirdly because computer systems by their nature are very good at optimizing problems within a constrained search space. The choices of human beings are often superior because of the ability of human beings to observe broader environments than any single component within a system can observe. Human beings understand the big picture where computers often do not. The ability of the administrator to understand the larger system means that if he/she has taken the time to manually intervene there are probably good reasons for that decision even if the autonomic components of the system can't detect them. Fourth, because even if the previous three reasons were wrong, it is unwise to tell your owner that he is doing a bad job. As a result, it follows that an important aspect of autonomic systems is that they distinguish between system changes made by human operators and systems changes made by the autonomic technology itself, so that changes performed by humans will not be overridden.

### #7 Minimize policy and keep it human

Architectures support policy, and functions consume policy, but neither functions nor architecture intrinsically need policy. Policy specification is in fact the specification by human administrators what the system could not glean on its own. Numerous policy grammars and specifications have been proposed over the past 30 years. The ultimate goal of autonomic systems should be to entirely eliminate the need for explicitly specified policy. However, that objective is more than a decade away. What we can say about the need for policy is therefore the following:

1. It is needed and will be needed for the next several years.

2. Policy should represent business objectives that can be described in relatively human terms, indicating what is expected of a system. Policy expression is not an excuse to inject configuration parameters and rules into an autonomic system.
3. In a world where policy remains, perhaps the most important attribute of policy is its standardization. Standardization more than anything else will lead to the easy combination of system components, and reusable components. Sadly today, "The nice thing about standards is that there are so many of them to choose from" (attributed to Andrew S. Tanenbaum).

## 3. Foundational Techniques

Autonomic computing is the term used for self-managing IT systems. Numerous publications over the past five years have shown how such self managing systems can draw on a wide range of disciplines. A short list, which research and development teams are encouraged to explore in the design of autonomic computing solutions includes: artificial intelligence, operations research, cybernetics, polycontexturality, second-order cybernetics, catastrophe theory, connectionism, control theory (digital, analog, closed/open loop), mathematical optimization, decision theory, game theory, information theory, semiotics, synergetics, sociosynergetics, and systems theory. However, the vast majority of development projects aimed at production use for the IT industry for storage systems, databases, web servers, network infrastructure, telephony, and self healing have focused on seven foundational approaches.

### 3.1 Autonomic computing as dependency management

Dependency management, modeled in various ways, but most commonly through mathematical graphs, has become a common domain in autonomic computing, particular for managing dependencies during system install [6]. The dependency management becomes particularly complex when installing numerous components onto an existing system with various different prerequisites and co-requisites.

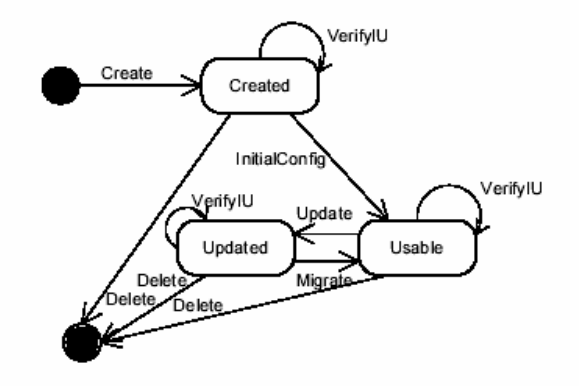


**Figure 3 State transition graph for installable units**

### 3.2 Autonomic computing as an expert system

Although the term expert system has a broad range of meanings, in this context we use the term specifically to refer to a system with a rule based or heuristic decision making infrastructure. Such systems typically use a knowledge base of known events or symptoms in combination with an inference engine. Numerous expert systems have been created for autonomic computing. A small expert system was shown to have significant value as a autonomic computing feature within the database domain to perform system configuration over several dozen tuning parameters [11]. There are several emerging examples as well for capacity planning to project the capital requirements of new systems based on expected users, throughput requirements and workload type.

Similarly, knowledge based expert systems have been proposed to detect and match problems within a complex solution stack (based on the assumption that the majority of problems encountered by customers are not unique). This is a particularly complex problem within a complex solution stack where each component within the stack has possibly been development by a distinct development team, and there is no certain correlation between the version levels of the various components when an error event occurs.

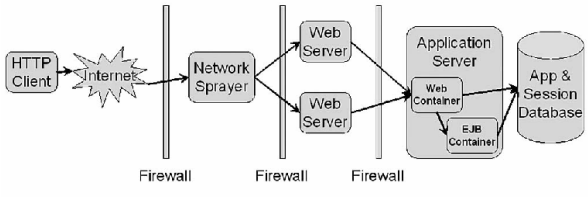


**Figure 4 Enterprise solution stack**

### 3.3 Autonomic computing as tradeoff elimination

The characterization of trade-off elimination as a discipline for autonomic computing development appears in a SIGMOD 2005 tutorial on self managing databases chaired by Chaudhuri and Weikum [4]. Chaudhuri et al give the example of cache replacement strategies, a classic self tuning problem. The two standard techniques are LRU and LFU. LRU drops the page that has been least recently used. LFU drops the page that has been least frequently used. Each algorithm has advantages. LFU is optimal for static access probabilities, but has no aging.

LRU is optimal if last access is indicative for next future access. Therefore, the choice of page replacement algorithms becomes tradeoff between recency vs. frequency, unless a compromise can be found that is "adequate" for both classes of use. An example of one such compromise strategy is LRU-k. LRU-k: drops the page with the oldest k-th last reference, and has been shown to have balanced page replacement for both recency and frequency.

$$\text{estimates } heat\ (p) = \frac{k}{now - t_k(p)} \quad (1)$$

Many problems in autonomic system management can best be approached as a search for a compromise that performs acceptably in the required environments rather than attempting to build an adaptive solution that toggles between multiple models by detecting the current environment; the later being naturally more error prone.

### 3.4 Autonomic computing as static optimization

Static optimization techniques have been widely used for autonomic computing technologies related to capacity planning, physical system design, and resource configuration. Standard techniques have included mathematical optimization, what-if analysis, and a slew of AI-related search schemes (greedy search, random search, dynamic programming with branch and bound, genetic algorithms, neural networks, simulated annealing, and many others). Notably, a number of impressive claims have been published in the database domain where static analysis has been used by several vendors in combination with mathematical modeling of resource consumption re-using the cost-based query compiler found in many modern RDBMSs to achieve dramatic performance gains [12][13].

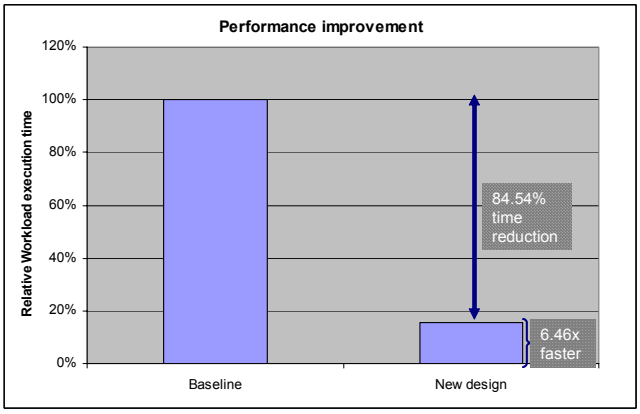


**Figure 5 Performance gains following static optimization of a database design, by Rao et al.**

### 3.5 Autonomic computing as online optimization

Open loop control and optimization remains a widely used paradigm, exploiting techniques from AI, control theory, queuing theory, mathematical optimization and cybernetics to achieve self tuning. In particular, numerous scheduling techniques form operations research and queuing theory are being actively used in production autonomic systems to achieve workload balance and prioritization particularly in the face of service level agreement (SLA) objectives [15] .

Queuing models have because very popular for modeling concurrency, response time, latency and throughput. Queuing models provide M/G/1 models with general service time distributions, multiple request (customer) classes, with priorities, service scheduling other than FIFO, GI/G/1 models and discrete-time models. Multiple queuing models are frequently coming into queuing networks used for modeling workflow [4].

### 3.6 Autonomic computing as feedback control loop

Control theory is a natural discipline to exploit in order to achieve self-managing systems. In fact numerous products have already used control theory techniques for this purpose. [8][9][10]. In particular the use of Proportional Integral (PI) controllers and Multi input/multi output (MIMO) controllers have been

reported in a number of self managing projects with good success.

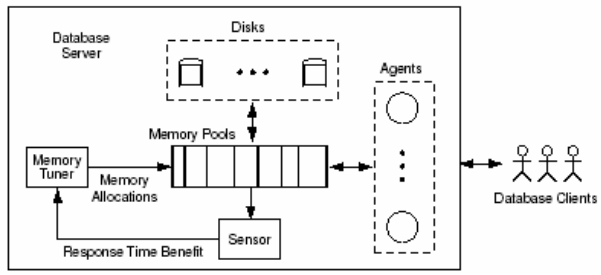


**Figure 6 Controller for memory tuning, using MIMO with feedback**

The closed loop control lends itself well to continuously adapting systems and a large body of literature and past history exists from mechanical and electrical engineering from which to draw on.

## 3.7 Autonomic computing as correlation modeling

Correlation modeling has already demonstrated its usefulness in both performance analysis and problem determination. IBM's eWLM workload management technology uses the ARM protocol to instrument and correlate the time spent by each transaction in each component of a solution stack.

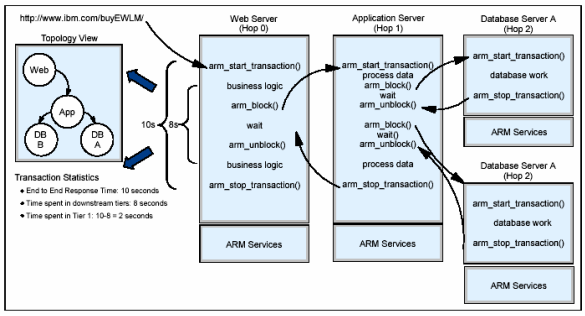


**Figure 7 ARM exploitation within IBM's eWLM (image courtesy of IBM)**

IBM's Log and Trace Analyzer performs event correlation between events in log files from multiple components which may reside on a physically distributed system. Analysis of the correlated events can help identify first point of failure.

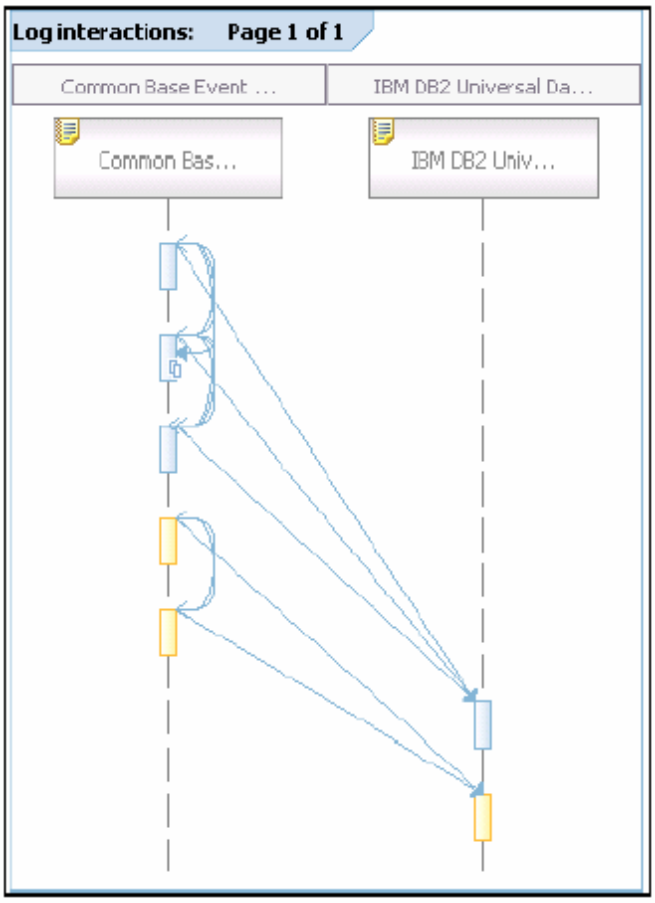


**Figure 8 Event correlation in IBM's Log and Trace Analyzer**

Correlation analysis systems are evolving as more profound problem determination engines, which is a critical step in the direction of truly self healing systems. [14][16][17]

## 3.8 Autonomic computing as security

Security has become, to a large degree its own mathematical discipline, though there clearly crossover. In the domain of autonomic computing systems, self-protection technology is largely based on security technology for both static and dynamic security intrusions and attacks. Formost in the foundational knowledge of self-protecting systems is a deep understanding of the broad classifications of intrusions and attacks. Excellent sources include the U.S. DoD Trusted Computer System Evaluation Criteria (the Orange Book) and the more recent Common Criteria. The SANS Institute (http://www.sans.org/) maintains a list of "The Twenty Most Critical Internet Security Vulnerabilities", through a consensus of a group of

experts in the field. The Center for Internet Security (http://www.cisecurity.org/) has also developed a tool for testing computer systems against the SANS Top 20 list, and also a set of documents and tools for benchmarking the security of different computing platforms. The ISO (International Organization for Standardization) 7498-2 is a widely referenced document associated with the design of IT security solutions. Its purpose is to extend the applicability of the seven-layer OSI (Open Systems Interconnection) system model to include secure communication between systems.

Self protecting technology is divided into two domains, namely static defenses and dynamic defenses. The former exploit encryption, antivirus/antiworm and firewall technology. This includes the detection of self-encrypting viruses and worms (polymorphic viruses) [19]. Dynamic defenses protect against real time intrusion and attacks (both denial of service (DoS) and distributed denial of service (DDoS)). Dynamic defenses against attacks are largely based on router and traffic analysis. Intrusion detection schemes are more complex and use a variety of pattern matching and routing analysis techniques [18].

## 4. The role of autonomic computing architecture

Several architectures have been proposed for autonomic computing in recent years [3]. In reality no single architecture will satisfy all domains. The architectural requirements for real-time embedded systems with extreme scalability and latency objectives force architectural tradeoffs that would never be acceptable in enterprise management system where availability, serviceability and component heterogeneity. What is known about architecture is simply this: it's good to have one for whatever system you are building, and it's even better for there to be an open standard for architectures within a domain. Architectures allow developments teams two degrees of freedom. First they allow for improved component reuse (the Philosopher's Stone of software development). Second, architectures allow groups of developers to collaboratively develop software more easily. In particular if the architecture is mature (reused) the development team can proceed rapidly with confidence that the architecture they are using is tried and true and unlikely to suddenly collapse from sudden forgotten failings (such as design scalability/concurrency). Therefore, the industry would be wise to move to an open architecture for autonomic computing for enterprise management systems. That autonomic computing architecture may be very different from an architecture we might build for a real-time transaction processing system, or an embedded operating system.

## 5. Conclusion

The development of autonomic systems and middle-ware pose special challenges (some unique others less so). We have show the administration of modern IT systems to have combinatorial complexity far exceeding the number of atoms in the universe, and therefore far beyond the ability of human administrators to fully optimize. A software engineering paradigm based on 7 specific guidelines for autonomic computing, such as the focus on building "sufficiently" self managing systems, and always providing an off switch. Autonomic systems should be based on a wide range of techniques, and a strong mathematical foundation exists for many of these. These include many well known foundational techniques form AI, control theory, operations research, and other domains. In this paper we surveyed some of the major techniques that have had the most significant traction in production oriented autonomic computing systems. By applying the software engineering principles described here and drawing from the core foundational mathematics that underlie the most successful autonomic computing projects software development teams can improve their chances for building autonomic technology that is both high value and accepted and trusted by system and database administrators.

## Acknowledgements

This article is based on part of a keynote talk given at the 3rd IEEE International Workshop on the Engineering of Autonomic and Autonomous Systems (EASe'06), March 29, 2006, Postdam, Germany. http://www.ulster.ac.uk/ease.
DOI: 10.1109/EASE.2006.18

## References

[1] P. Horn, "Autonomic Computing: IBM's Perspective on the State of Information Technology", http://www.ibm.com/research/autonomic, International Business Machines, Armonk, NY, 2001

[2] S. Lightstone, A. Storm, C. Garcia-Arellano, M. Carroll, J. Colaco , Y. Diao, M. Surendra "Self tuning memory management in a relational database system" Fourth Annual Workshop on Systems and Storage Technology, December

11, 2005, IBM Research Lab, Haifa University campus, Mount Carmel, Haifa, Israel.
[3] Steve R. White, James E. Hanson, Ian Whalley, David M. Chess, Jeffrey O. Kephart: An Architectural Approach to Autonomic Computing. ICAC 2004: 2-9
[4] Surajit Chaudhuri, Gerhard Weikum: Foundations of automated database tuning. SIGMOD Conference 2005: 964-965
[5] S. Lightstone, B. Schiefer, D. Zilio, J. Kleewein "Autonomic Computing for Relational Databases: the ten year vision", IEEE Workshop on Autonomic Computing Principles and Architectures (AUCOPA' 2003), Banff AB, Aug. 2003.
[6] Christine Draper, Randy George, Marcello Vitaletti: Installable Unit Deployment Descriptor for Autonomic Solution Management. DEXA Workshops 2004: 742-746
[7] P. Coffee, "Autonomic hinges on truest, eWeek.com, June 7 2004 .
http://www.eweek.com/article2/0,1895,1607786,00.asp
[8] J.L. Hellerstein, Y. Diao, S. Parekh, D.M. Tilbury: Feedback Control of Computing Systems, Wiley 2004
[9] S. Parekh, K. Rose, Y. Diao, V. Chang, J. Hellerstein, S. Lightstone, M. Huras: Throttling Utilities in the IBM DB2 Universal Database Server, American Control Conference, 2004
[10] Yixin Diao, Joseph L. Hellerstein, Sujay Parekh, Rean Griffith, Gail Kaiser, Dan Phung, "Self-Managing Systems: A Control Theory Foundation," ecbs, pp. 441-448, 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05), 2005. doi: 10.1109/ECBS.2005.60
[11] E. Kwan, S. Lightstone, B. Schiefer, A Storm, L Wu. "Automatic Configuration for IBM DB2 Universal Database: Compressing years of performance tuning experience into seconds of execution", 10th Conference on Database Systems for Business, Technology, and the Web (BTW 2003),February 26 - 28, 2003, University of Leipzig, Germany.
[12] J. Rao, S. Lightstone , G. Lohman, D. Zilio , A. Storm, C. Garcia-Arellano, S. Fadden. "DB2 Design Advisor: integrated automated physical database design", VLDB 2004, Toronto, Canada
[13] Nicolas Bruno and Surajit Chaudhuri. Automatic Physical Database Tuning: A Relaxation-based Approach. SIGMOD 2005.
[14] Sterritt R, (Dec 2002) "Towards Autonomic Computing: Effective Event Management", Proceedings of 27th Annual IEEE/NASA Software Engineering Workshop (SEW), Maryland, USA, December 3-5 2002, IEEE Computer Society, Pages 40-47
[15] Melissa J. Buco, Rong N. Chang, Laura Z. Luan, Christopher Ward, Joel L. Wolf, Philip S. Yu: Utility computing SLA management based upon business objectives. IBM Systems Journal 43(1): 159-178 (2004)
[16] Mark Brodie, Sheng Ma, Leonid Rachevsky, Jon Champlin: Automated Problem Determination using Call-Stack Matching. J. Network Syst. Manage. 13(2): (2005)
[17] Mark Brodie, Sheng Ma, Guy Lohman, Tanveer Syeda-Mahmood, Laurent Mignet, Natwar Modani, Mark Wilding, Jon Champlin, Peter Sohn, "Quickly Finding Known Software Problems via Automated Symptom Matching", ICAC 2004.
[18] James J. Whitmore: A method for designing secure solutions. IBM Systems Journal 40(3): 747-768 (2001)
[19] "The History of Computer Viruses", ViruScanSoftware, http://www.virus-scan-software.com
[20] S. Lightstone, "The need for ease: development principles for successful autonomic computing projects," *ease*, pp. 5-8, Third IEEE International Workshop on Engineering of Autonomic & Autonomous Systems (EASE'06), 2006. doi**:** 10.1109/EASE.2006.16