pdfauthor

A survey of Autonomic Computing — degrees, models and applications

MARKUS C. HUEBSCHER, JULIE A. MCCANN
Imperial College London

Autonomic Computing is a concept that brings together many fields of computing with the purpose of creating computing systems that self-manage. In its early days it was criticised as being a "hype topic" or a rebadging of some Multi Agent Systems work. In this survey, we hope to show that this was not indeed 'hype' and that, though it draws on much work already carried out by the Computer Science and Control communities, its innovation is strong and lies in its robust application to the specific self-management of computing systems. To this end, we first provide an introduction to the motivation and concepts of autonomic computing and describe some research that has been seen as seminal in influencing a large proportion of early work. Taking the components of an established reference model in turn, we discuss the works that have provided significant contributions to that area. We then look at larger scaled systems that compose autonomic systems illustrating the hierarchical nature of their architectures. Autonomicity is not a well defined subject and as such different systems adhere to different degrees of Autonomicity, therefore we cross-slice the body of work in terms of these degrees. From this we list the key applications of autonomic computing and discuss the research work that is missing and what we believe the community should be considering.

Categories and Subject Descriptors: A.1 [**INTRODUCTORY AND SURVEY**]:

General Terms: Design, Performance, Reliability.

Additional Key Words and Phrases: Autonomic computing, self-adaptive, self-healing systems

1. INTRODUCTION

Computing systems have reached a level of complexity where the human effort required to get the systems up and running and keeping them operational is getting out of hand. A similar problem was experienced in the 1920s in telephony. At that time, human operators were required to work the manual switchboards, and because usage of the telephone increased rapidly, there were serious concerns that there would not be enough trained operators to work the switchboards [Mainsah 2002]. Fortunately, automatic branch exchanges were introduced to eliminate the need for human intervention.

Autonomic computing seeks to improve computing systems with a similar aim of decreasing human involvement. The term "autonomic" comes from biology. In the human body, the autonomic nervous system takes care of unconscious reflexes, i.e. bodily functions that do not require our attention, e.g. bodily adjustments such as the size of the pupil, the digestive functions of the stomach and intestines, the rate and depth of respiration and dilatation or constriction of the blood vessels. Without the autonomic nervous system, we would be constantly busy consciously adapting our body to its needs and to the environment.

Autonomic computing attempts to intervene in computing systems in a similar fashion as its biological counterpart. The term autonomic computing was first used by IBM in 2001 to describe computing systems that are said to be self-managing

[Kephart and Chess 2003]. However, the concepts behind self-management were not entirely new to IBM's autonomic computing initiative. In the next section, we present a brief history of autonomic computing related projects and initiatives in chronological order. Throughout the paper we use the terms Autonomic Computing, Self-Managing Systems, and Self-Adaptive Systems interchangeably.

To explore this diverse field, we first provide an introduction to the motivation and concepts of autonomic computing in Section 2 and describe some research that has been seen as seminal in influencing a large proportion of early work in Section 3. Taking the components of the MAPE-K Loop reference model, which we describe in Section 4, we discuss the works that have provided significant contributions to that area, in Sections 4.1–4.5. We then look at larger scaled systems that compose autonomic systems illustrating the hierarchical nature of their architectures in section 5. Autonomicity is not a well defined subject and as such different systems adhere to different degrees of autonomicity, therefore we cross-slice the body of work and discuss these degrees in Section 6. From this, we list the key applications of autonomic computing in Section 7 and conclude with a discussion on what research work is lacking and what we believe the community should be considering in Section 8.

2. A BRIEF HISTORY

We now describe early work in autonomic computing that we believe have been influential to many other later projects in the autonomic research field.

Similarly to the birth of the Internet, one of the notable early self-managing projects was initiated by DARPA for a military application in 1997. The project was called the Situational Awareness System[1] (SAS), which was part of the broader Small Units Operations (SUO) programme. Its aim was to create personal communication and location devices for soldiers on the battlefield. Soldiers could enter status reports, e.g. discovery of enemy tanks, on their personal device, and this information would autonomously spread to all other soldiers, which could then call up the latest status report when entering an enemy area. Collected and transmitted data includes voice messages and data from unattended ground sensors and unmanned aerial vehicles. These personal devices have to be able to communicate with each other in difficult environmental conditions, possibly with enemy jamming equipment in operation, and must at the same time minimise enemy interception to this end [Kenyon 2001]. The latter point is addressed by using multi-hop ad-hoc routing, i.e. a device sends its data only to the nearest neighbours, which then forward the data to their own neighbours until finally all devices receive the data. This is a form of decentralised peer-to-peer mobile adaptive routing, which has proven to be a challenging self-management problem, especially because in this project the goal is keep latency below 200 milliseconds from the time a soldier begins speaking to the time the message is received. The former point is addressed by enabling the devices to transmit in a wide band of possible frequencies, 20–2,500MHz, with bandwidths ranging from 10bps to 4Mbps. For instance, when distance to next soldier is many miles, communication is only possible at low frequencies, which results in low bandwidth, which may still be enough to provide a brief but possibly crucial

[1]SAS home page: `http://www.darpa.mil/ato/programs/suosas.htm`

status report. Furthermore, there may be up to 10,000 soldiers on the battlefield, each with their own personal devices connected to the network.

Another DARPA project related to self-management is the DASADA[2] project started in 2000. The objective of the DASADA programme was to research and develop technology that would enable mission critical systems to meet high assurance, dependability, and adaptability requirements. Essentially, it deals with the complexity of large distributed software systems, a goal not dissimilar to IBM's autonomic computing initiative. Indeed, this project pioneered the architecture-driven approach to self-management (see Section 4.4.2), and more broadly the notion of probes and gauges for monitoring the system and an adaptation engine for optimising the system based on monitoring data [Garlan et al. 2001; Cobleigh et al. 2002; Kaiser et al. 2002; Gross et al. 2001; Wolf et al. 2000].

In 2001, IBM suggested the concept of *autonomic computing*. In their manifesto [Horn 2001], complex computing systems are compared to the human body, which is a complex system, but has an autonomic nervous system that takes care of most bodily functions, thus removing from our consciousness the task of coordinating all our bodily functions. IBM suggested that complex computing systems should also have autonomic properties, i.e. should be able to independently take care of the regular maintenance and optimization tasks, thus reducing the workload on the system administrators. IBM also distilled the four properties of a self-managing (i.e. autonomic) system: self-configuration, self-optimization, self-healing and self-protecting (these properties are described in more detail in Section 3).

The DARPA Self-Regenerative Systems programme started in 2004 is a project that aims to "develop technology for building military computing systems that provide critical functionality at all times, in spite of damage caused by unintentional errors or attacks" [Badger 2004]. There are four key aspects to this project. First, software is made resistant to error and attacks by generating a large number of versions that have similar functional behaviour, but sufficiently different implementation, such that any attack will not be able to affect a substantial fraction of the versions of the program. Second, modifications to the binary code can be performed, such as pushing randomly sized blocks onto the memory stack, that make it harder for attackers to exploit vulnerabilities, such as specific branching address locations, as the vulnerability location changes. Furthermore, a trust model is used to steer the computation away from resources likely to cause damage. Whenever a resource is used, the trust model is updated based on outcome. Third, a scalable wide-area intrusion-tolerant replication architecture is being worked on, which should provide accountability for authorised but malicious client updates. Fourth, technologies are being developed that supposedly allow a system to estimate the likelihood that a military system operator (an "insider") is malicious and prevent it from initiating an attack on the system.

Finally, we would like to mention an interesting project that started at NASA in 2005, the Autonomous NanoTechnology Swarm (ANTS). As an exemplary mission, they plan to launch into an asteroid belt a swarm of 1000 small spacecraft (so called "pico-class" spacecraft) from a stationary factory ship in order to explore the asteroid belt in detail. Because as much as 60–70% of the swarm is expected

[2]DASADA home page: `http://www.rl.af.mil/tech/programs/dasada/program-overview.html`

Table I. A brief chronology of influential self-management projects.

SAS Situational Awareness System	1997	DARPA	Decentralised self-adaptive (ad-hoc) wireless network of mobile nodes that adapt routing to the changing topology of nodes and adapt communication frequency and bandwidth to environmental and node topology conditions.
DASADA Dynamic Assembly for Systems Adaptability, Dependability, and Assurance	2000	DARPA	Introduction of gauges and probes in the architecture of software systems for monitoring the system. An adaptation engine then uses this monitored data to plan and trigger changes in the system, e.g. in order to optimise performance or counteract failure of a component.
AC Autonomic Computing	2001	IBM	Compares self-management to the human autonomic system, which autonomously performs unconscious biological tasks. Introduction of the four central self-management properties (self-configuring, self-optimising, self-healing and self-protecting).
SPS Self-Regenerative Systems	2003	DARPA	Self-healing (military) computing systems, that react to unintentional errors or attacks.
ANTS Autonomous NanoTechnology Swarm	2005	NASA	Architecture consisting of miniaturised, autonomous, reconfigurable components that form structures for deep-space and planetary exploration. Inspired by insect colonies.

to be lost as they enter the asteroid belt, the surviving craft must work together. This is done by forming small groups of worker craft with a coordinating ruler, that uses data gathered from workers to determine which asteroids are of interest and to issue instructions. Furthermore, messenger craft will coordinate communications between members of the swarm and with ground control. In fact, NASA has already previously used autonomic behaviour in its DS1 (Deep Space 1) mission and the Mars Pathfinder [Muscettola et al. 1998]. Indeed, NASA has a strong interest in autonomic computing, in particular in making its deep-space probes more autonomous. This is mainly because there is a long round-trip delay between a probe in deep space and mission control on Earth. So, as mission control cannot rapidly send new commands to a probe—which may need to quickly adapt to extraordinary situations—it is extremely critical to the success of an expensive space exploration mission that the probes be able to make certain critical decisions on their own.

3. SELF-MANAGEMENT PROPERTIES

The main properties of self-management as portrayed by IBM are self-configuration, self-optimisation, self-healing and self-protection. Here is a brief description of these properties (for more information, see [Kephart and Chess 2003; Bantz et al. 2003]):

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 and set itself up based on the needs of the platform and the user.

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 or quality of service.

Self-healing. An autonomic computing system detects and diagnoses problems. The kinds of problems that are detected can be interpreted broadly: they can be as low-level as bit-errors in a memory chip (hardware failure) or as high-level as an erroneous entry in a directory service (software problem) [Paulson 2002]. 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. That is, an autonomic system is said to be *reactive* to failures or early signs of a possible failure.

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. Security is an important aspect of self-protection, not just in software, but also in hardware (e.g. TCPA[3]). A system may also be able to anticipate security breaches and prevent them from occurring in the first place. Thus, the autonomic system exhibits *proactive* features.

The self-X properties of autonomic systems are inspired by the properties of software (or hardware) agents, which Wooldridge and Jennings [1995] first identified as being:

Autonomy. Agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state.

Social ability. Agents interact with other agents (and possibly humans) via some kind of agent-communication language.

Reactivity. Agents perceive their environment, and respond in a timely fashion to changes that occur in it.

Pro-activeness. Agents do not simply act in response to their environment, they are able to exhibit goal-directed behaviour by taking the initiative.

There remains a debate as to what Self-Managing Systems actually are. For example, a query optimisor, resource manager or routing software in DBMSs, operating systems and networks, respectively, all allow those systems to self manage. However the Self-Managing systems' community are coming to an agreement that the term autonomic computing is not being used to describe these systems but those in which the query plan, resource management or routing decision changes to reflect the current environmental context; reflecting dynamism in the system. That is, the DBMS query plan changes as the query is running.

Self-Adaptive systems have contained some elements of the properties above for some time, especially to provide self-optimisation. Early examples of this can be seen in streaming media systems where the codec of the stream changes with network bandwidth fluctuations, the goal being to keep music or video playback as high

[3]TCPA - The Trusted Computing Platform Alliance. URL: `http://www.trustedcomputing.org/`

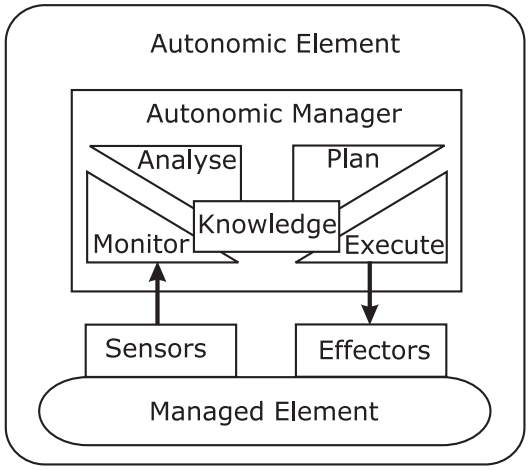

Fig. 1. IBM's MAPE-K (Monitor, Analyse, Plan, Execute, Knowledge) reference model for autonomic control loops.

a quality as possible, e.g. Kendra [McCann and Crane 1998] and Real Surestream [Lippman 1999]. However the autonomic community is more and more identifying a system as autonomic if it exhibits more than one of the self-management properties described earlier, e.g. Ganek and Friedrich [2006].

4. THE MAPE-K AUTONOMIC LOOP

To achieve autonomic computing, IBM has suggested a reference model for autonomic control loops [IBM 2003], which is sometimes called the MAPE-K (Monitor, Analyse, Plan, Execute, Knowledge) loop and is depicted in Figure 1. This model is being used more and more to communicate the architectural aspects of autonomic systems. Likewise it is a clear way to identify and classify much of the work that is being carried out in the field. Therefore this section introduces the MAPE-K loop in more detail and then taking each of its components in turn, describes the work that focuses its research on that component. We later take the same work and examine it from a degree of autonomicity point of view.

The MAPE-K autonomic loop is similar to, and probably inspired by, the generic agent model proposed by Russel and Norvig [2003], in which an intelligent agent perceives its environment through sensors, and uses these percepts to determine actions to execute on the environment.

In the MAPE-K autonomic loop, the *managed element* represents any software or hardware resource that is given autonomic behaviour by coupling it with an autonomic manager. Thus, the managed element can for example be a web server or database, a specific software component in an application (e.g. the query optimiser in a database), the operating system, a cluster of machines in a grid environment, a stack of hard drives, a wired or wireless network, a CPU, a printer, etc.

Sensors, often called probes or gauges, collect information about the managed element. For a web-server, that could include the response time to client requests, network and disk usage, CPU and memory utilisation. A considerable amount of

research is involved in monitoring servers [Roblee et al. 2005; Strickland et al. 2005; Xu et al. 2005; Sterritt et al. 2005; Diao et al. 2005].

Effectors carry out changes to the managed element. The change can be coarse-grained, e.g. adding or removing servers to a web server cluster [Garlan and Schmerl 2002a], or thin-grained, e.g. changing configuration parameters in a web server [Sterritt et al. 2005; Bigus et al. 2002].

4.1 Autonomic manager

The data collected by the sensors allows the *autonomic manager* to *monitor* the managed element and *execute* changes through effectors. The autonomic manager is a software component that ideally can be configured by human administrators using high-level goals and uses the monitored data from sensors and internal knowledge of the system to plan and execute, based on these high-level goals, the low-level actions that are necessary to achieve these goals. The internal knowledge of the system is often an architectural model of the managed element (see Section 4.4.2). The goals are usually expressed using event-condition-action (ECA) policies, goal policies or utility function policies [Kephart and Walsh 2004]. ECA policies take the form "when *event* occurs and *condition* holds, then execute *action*, e.g. "when 95% of web servers' response time exceeds 2s and there are available resources, then increase number of active web servers". They have been intensely studied for the management of distributed systems. A notable example is the PONDER policy language [Damianou et al. 1999]. A difficulty with ECA policies is that when a number of policies are specified, conflicts between policies can arise that are hard to detect. For example, when different tiers of a multi-tier system (e.g. web and application server tiers) require an increased amount of resources, but the available resources cannot fulfill the requests of all tiers, a conflict arises. In such a case, it is unclear how the system should react, and an additional conflict resolution mechanism is necessary, e.g. giving higher priority to the web server. As a result, a considerable amount of research on conflict resolution has arisen [Lupu and Sloman 1999; Kamoda et al. 2005; Ananthanarayanan et al. 2005]. However, a complication is that the conflict may only become apparent at runtime. Goal policies are more high level in that they specify criteria that characterise desirable states, but leave to the system the task of finding how to achieve that state. For example, we could specify that the response time of the web server should be under 2s, while that of the application server under 1s. The autonomic manager uses internal rules (i.e. knowledge) to add or remove resources as necessary to achieve the desirable state. Goal policies require planning on the part of autonomic manager and are thus more resource-intensive than ECA policies. However, they still suffer from the problem that all states are classified as either desirable or undesirable. Thus when a desirable state cannot be reached, the system does not know which among the undesirable states is least bad. Utility functions solve this problem by defining a quantitative level of desirability to each state. A utility function takes as input a number of parameters and outputs the desirability of this state. Thus, continuing our example, the utility function could take as input the response time for web and application servers and return the utility of each combination of web and application server response times. This way, when insufficient resources are available, the most desirable partition of available resources among web and application servers can be

found. The major problem with utility functions is that they can be extremely hard to define, as every aspect that influences the decision by the utility function must be quantified. Research is being carried out on using utility functions, particularly in automatic resource allocation [Walsh et al. 2004] or adaptation of data streaming to network conditions [Bhatti and Knight 1999].

Autonomic elements may cooperate to achieve a common goal [IBM 2003], e.g. servers in a cluster optimising the allocation of resources to applications to minimise the overall response time or execution time of the applications. Thus, autonomic managers may need to be aware not only of the condition of their own managed element, but also of their environment, in particular other autonomic elements in the network. This notion of cooperation of individual elements to achieve a common goal is a fundamental aspect of multi-agent systems research, and it is therefore not surprising that considerable research is investigating the application of multi-agent systems to implement cooperating autonomic elements, dealing in particular with the difficulties faced in multi-agent systems in guaranteeing that the behaviour emerging from the individual goals of each agent will truly result in the common goal being achieved [Kephart and Chess 2003; Jennings 2000; Gleizes et al. 2000; Kumar and Cohen 2000]. An alternative to multi-agent cooperation is a hierarchical structuring of autonomic elements [Wise et al. 2000].

As Autonomic Management solutions become more decentralised and less deterministic, we may begin to observe emergent features. There has been an initial interest in engineering this emergence or taking the bio-inspiration of autonomicity further than the original definition. The term *engineered emergence* can be described as the "purposeful design of interaction protocols so that a predictable, desired outcome or set of outcomes are achieved at a higher level" [Anthony 2006]. It is an approach to building systems that benefit from characteristics such as scale, robustness and stability, but do not require precise knowledge of lower-level activity or configuration. In such systems the solution emerges at the level of systems or applications while at lower levels the specific behaviour of individual components is unpredictable. Typically a small set of rules operate on limited amounts of locally available information concerning the components execution context and its local environment. This differs from traditional algorithmic design of distributed applications which has typically focused on strict protocols, message acknowledgments and event ordering. In such systems each message and event is considered important and randomness is generally undesirable, imposing sequenced or synchronised behaviour which is generally deterministic. However, natural biological systems are fundamentally non-deterministic and there are many examples of large-scale systems that are stable and robust at a global level; the most commonly cited examples being drawn from cellular systems and insect colonies [Anthony 2006].

4.2 Example implementations of the MAPE-K loop

4.2.1 *Autonomic Toolkit.* IBM has developed a prototype implementation of the MAPE-K loop called the Autonomic Management Engine, as part of its developerWorks Autonomic Computing Toolkit[4]. The Autonomic Computing Toolkit

[4]Autonomic Computing Toolkit Home page: `http://www-128.ibm.com/developerworks/autonomic/overview.html`

"provides a practical framework and reference implementation for incorporating autonomic capabilities into software systems" [Melcher and Mitchell 2004]. Thus, is it not a complete autonomic manager as such, but provides the foundations on which to build autonomic manager. It is implemented in Java, but can communicate with other applications via XML messages, e.g. for sensing by analysing the logs of a managed application. The architecture is applicable to all areas where the autonomic manager can be implemented at the software application level (as opposed to the operating system or hardware level). For instance, Melcher and Mitchell [2004] used it to create autonomic network service configuration, where network devices are modelled as managed resources and dedicated servers or gateways play the role of autonomic managers. However, they had to make a number of extensions to the toolkit. For instance, the toolkit only supported local communication between managed resource and autonomic manager, whereas in their case the two components are distributed in the network.

4.2.2 *ABLE.* Another toolkit well-known in the literature is IBM's ABLE toolkit [Bigus et al. 2002]. Autonomic management is provided in the form of a multi-agent architecture, i.e. each autonomic manager is implemented as an agent or set of agents, thus allowing different autonomic tasks to be separated and encapsulated into different agents. The toolkit is implemented in Java, and can be used to manage software applications such as web servers and databases.

4.2.3 *Kinesthetics eXtreme.* Valetto and Kaiser have also worked on their own implementation (mainly in Java) of the complete autonomic loop, called Kinesthetics eXtreme (KX) [Kaiser et al. 2003; Parekh et al. 2003]. Their work was driven by the problem of adding autonomic properties to legacy systems, i.e. existing system which were not designed with autonomic properties in mind. Indeed, it is sometimes not possible to modify these systems, thus requiring the addition of autonomic properties to be completely decoupled, with autonomic monitoring sensors added on top of existing system APIs and monitoring functionality. Their work focusses more on the collection and processing of monitoring data from legacy systems and execution of adaptation and repairs rather than algorithms and policies for adaptation planning.

At the monitoring level, they propose two types of sensors: probes and gauges. Probes are system-specific sensors that extract data from a managed element. Probe data is then sent to gauges, which may filter, aggregate and process the probes' data before reporting it to higher-level components in the autonomic manager for adaptation planning [Gross et al. 2001; Valetto and Kaiser 2002]. For instance, gauges can determine the original failure that started a cascading problem by correlating the time when different probe events were fired and pattern-matching the attributes of probe events to determine the causality sequence. Probes and gauges may not be on the same machine, thus requiring network communication between. This can be considered a form of distributed sensor and makes sense when the managed element and its autonomic manager reside on different machines. In addition, while probes and effectors are often specialised to the specifics of the managed element's implementation (Java/J2EE, C++, .NET), the gauges and planning components must instead be specialised to the autonomic problem, e.g. optimising service availability

of a web-server cluster vs. managing a computational grid.

For adaptation planning, they have created a workflow engine called Workflakes [Valetto and Kaiser 2003]. While they have experimented with different approaches—such as implementing adaptation rules directly in Java code, using Little-JIL [Wise et al. 2000] formalism and the Acme ADL [Garlan and Schmerl 2002a] to create a working system for their case studies—the planning aspect of the autonomic loop has not been their main focus.

To effect adaptation, the autonomic manager (Workflakes) deploys Worklets to the managed system. Worklets are mobile software agents that are executed on the managed system by a Worklet Virtual Machine. Host-specific adaptor at each managed system translates Worklet adaptation commands into host-specific actions.

KX has been applied to various case studies, including an instant-messaging service [Valetto and Kaiser 2002], adaptive streaming of multimedia data [Kaiser and Valetto 2000], failure detection and recovery plus load balancing in a Geographical Information System called GeoWorlds [Kaiser et al. 2003].

4.2.4 *Self-management tightly coupled with application.* There are projects that propose an autonomic middleware framework that offers self-management properties to applications (in the role of managed element) built on top of these autonomic middleware frameworks. Examples of this approach include 2K (aka dynamicTao) [Kon et al. 2000], Polylith [Hofmeister and Purtilo 1993] and Conic [Magee and Sloman 1989].

There are also projects where the application is implemented following a certain process-coordination approach, e.g. encapsulating tasks in components and defining self-management and adaptation in terms of these components. An example of this approach are Containment Units/Little_JIL [Cobleigh et al. 2002; Wise et al. 2000].

The difference with the previously described projects such as KX is that here the application must be built and operate on top of the autonomic middleware or process-coordination platform, and thus the managed application cannot be a legacy system. The advantage though is that sensors and effectors can be reused across managed applications, and thus there is no need to create specific sensors and effectors for each application implementation.

4.3 Monitoring

We now consider the monitoring component of the MAPE-K loop.

Monitoring involves capturing properties of the environment (either physical or virtual, e.g. a network) that are of significance to the self-X properties of the system. The software or hardware components used to perform monitoring are called sensors. For instance, network latency and bandwidth measure the performance of web servers, while database indexing and query optimisation affect the response time of a DBMS, which can be monitored. The Autonomic Manager requires appropriate monitored data to recognise failure or sub-optimal performance of the Autonomic Element, and effect appropriate changes. The types of monitored properties, and the sensors used, will often be application-specific, just as effectors used to execute changes to the Managed Element are also application-specific.

We identify two types of monitoring in autonomic systems:

Passive monitoring. Passive monitoring of a computer system can be easily done

under Linux. For example, the `top` command returns information about CPU utilisation by each process. Another common command that returns memory and cpu utilisation statistics is `vmstat`. Furthermore, Linux provides the `/proc` directory, which is a pseudo-file system (this directory does not store actual files on a hard disk) containing runtime system information, e.g. system memory, CPU information, per-process information (such as memory utilisation), devices mounted, hardware configuration, etc. Similar passive monitoring tools exist for most operating systems, e.g. Windows 2000/XP.

Active monitoring. Active monitoring means engineering the software at some level, e.g. modifying and adding code to the implementation of the application or the operating system, to capture function or system calls. This can often be to some extent automated. For instance, ProbeMeister[5] can insert probes into the compiled Java bytecode.

More recent work has examined how decide which subset of the many performance metrics collected from an dynamic environment can be obtained from the many performance tools available to it (e.g. dproc). Interestingly they observe that a small subset of metrics provided 90% of their application classification accuracy [Zhang and Figueiredo 2006]. We are also beginning to observe a more dynamic approach to the monitoring of systems to facilitate autonomicity. For example, Agarwala et al. [2006] propose QMON, an autonomic monitor that adapts its monitoring frequency and data volumes so to minimise the overhead of continuous monitoring while maximising the utility of the performance data. Essentially, an autonomic monitor for autonomic systems

4.4 Planning

Let us know take a look at research that focuses on the planning aspect of the autonomic loop. In the broadest sense, planning involves taking into account the monitoring data from the sensors to produce a series of changes to be effected on the managed element. In the simplest case, we could define event-condition-action (ECA) rules that directly produce adaptation plans from specific event combinations, as already mentioned in Section 4.1. Examples of such policy languages and applications in autonomic computing include [Damianou et al. 2000; Lymberopoulos et al. 2003; Lobo et al. 1999; Agrawal et al. 2005; Batra et al. 2002; Lutfiyya et al. 2001; Ponnappan et al. 2002].

However, applying this approach in a stateless manner—i.e. where the autonomic manager keeps no information on state of the managed element, and relies solely on the current sensor data to decide whether to effect an adaptation plan—is very limited. Indeed, it is far better for the autonomic manager to keep information on the state of the managed element that can be updated progressively through sensor data and reasoned about.

Taking further the issue of state information that the autonomic manager should keep about the managed element, much research has examined a more complete approach to managing a system—called the architectural model-driven approach—in which some form of model of the entire managed system is created by the autonomic

[5]ProbeMeister Home page: `http://http://www.objs.com/ProbeMeister/`.

manager, usually called an *architectural model*, that reflects the managed system's behaviour, its requirements and possibly also its goal. The model may also represent some aspect of the operating environment in which the managed elements are deployed, where operating environment can be understood as any observable property (by the sensors) that can impact its execution, e.g. end-user input, hardware devices, network connection properties.

The model is updated through sensor data and used to reason about the managed system to plan adaptations. A great advantage of the architectural model-based approach to planning is that, under the assumption that the model correctly mirrors the managed system, the architectural model can be used to verify that system integrity is preserved when applying an adaptation, i.e. we can guarantee that the system will continue to operate correctly after the planned adaptation has been executed [Oreizy et al. 1999]. This is because changes are planned and applied to the model first, which will show the state of the system resulting from the adaptation, including any violations of constraints or requirements of the system present in the model. If the new state of the system is acceptable, the plan can then be effected onto the actual managed system, thus ensuring that the model and implementation are consistent with respect to each other.

The use of the architectural model-driven approach does not however necessarily eliminate ECA rules. Indeed, repair strategies of the architecture model may be specified as ECA rules, where an event is generated when the model is invalidated by sensor updates, and an appropriate rule specifies the actions necessary to return the model to a valid state, i.e. the adaptation plan.

In practice however, there is always a delay between the time when a change occurs in the managed system and this change is applied to the model. Indeed, if the delay is sufficiently high and the system changes frequently, an adaptation plan may be created and sent for execution under the belief that the actual system was in a particular state, e.g. a web server overloaded, when in fact the system has already changed in the meantime and does not require this adaptation anymore (or requires a different adaptation plan) [Garlan et al. 2001].

4.4.1 *Policy-based adaptation planning.* In policy-based adaptation, policies, which are essentially event-condition-action (ECA) rules, determine the actions to take when an event occurs and certain conditions are met [Sloman 1994]. These rules are written by system administrators and describe the adaptation plans of the system (see Section 4.1 for an example ECA rule.

Writing adaptation policies is fairly straightforward, but can become a tedious task for complex systems. Yet its simplicity remains its biggest strength. However, an issue with ECA rules is the problem of conflicts: an event might satisfy the conditions of two different ECA rules, yet each rule may dictate an action that conflicts with the other satisfying rule [Lupu and Sloman 1997]. Worse, these conflicts cannot always be detected at the time of writing the policies, some are only detected at run-time. This means that a human may be needed in the loop to solve policy conflicts when they arise.

4.4.2 *Architectural models.* All architectural models tend to share the same basic idea of the model being a network of components and connectors. The com-

ponents represent some unit of concurrent computing task, whereas the connectors represent the communication between components. Usually, there is no restriction as to the level of granularity of a component: it could be a complete web-server, an application on a web-server, or a component of an application.

The architectural model does not describe a precise configuration of components and connectors that the managed element must conform to. Instead, it sets a number of constraints and properties on the component and connectors, so that it can be determined when the managed element violates the model and needs adaptation.

Let us now continue our description of architectural model-based planning in the MAPE-K loop by taking a look at some of the most notable architectural description languages (ADLs), which can be used to specify an architectural model of a managed system.

4.4.2.1 *Darwin.* Let us start with Darwin, one of the first ADLs that was the result of seminal work by Magee et al. [1995]. In Darwin, the architectural model is a directed graph in which nodes represent component instances and arcs specify bindings between a service required by one component and the service provided by another. Further, the Allow object modelling notation [Jackson 2002] has been applied to Darwin components to be able to specify constraints on the components [Georgiadis et al. 2002]. For instance, consider the scenario where there are a number of server components offering services and a number of client components requiring services. Each service of a component is typed so that different services offered by a server or requested by a client can be distinguished and properly matched. In this scenario, the architectural model can guarantee that there are enough servers to service the clients. Should that not be the case, new server components must be started that offer the unavailable service types in order to return the model to a valid state.

In this approach each component keeps a copy of the architectural model. In other words, each component in the architectural model is an autonomic element with a managed element and an autonomic manager that holds the architectural model to the entire system. This approach avoids the presence of a central architectural model management service, which would otherwise introduce the problem of detecting and handling the failure of this central component. Where such a decentralised approach is taken, there is however the problem of keeping the architectural model up-to-date and consistent across all copies in the autonomic managers. This can be achieved with totally ordered atomic broadcasts, which works as long as no communication partitions occur between the components.

4.4.2.2 *Acme/ABLE.* The Acme adaptation framework [Garlan and Schmerl 2002a; Garlan and Schmerl 2002b; Garlan et al. 2001] is a software architecture that uses an architectural model for monitoring and detecting the need for adaptation in a system. The components and connectors of their architectural model can be annotated with a property list and constraints for detecting the need for adaptation. A first-order predicate language (called Armani) is used in Acme to analyse the architectural model and detect violations in the executing system. An imperative language is then used to describe repair strategies, much like the policy based

approach. The difference lies in how the need for adaptation is detected and the appropriate adaptation rule selected. Whereas in policies it is explicitly described in the rules, with an architectural model the need for adaptation implicitly emerges when the running system violates constraints imposed by the architectural model.

[Garlan et al. 2002] also define architectural styles. These define a set of types of components, connectors, interfaces, and properties together with a set of rules that govern how elements of those types may by composed. Different styles can be defined for different classes of applications, e.g. web servers, and serve to reduce the overhead of creating an architectural model for a particular application. Indeed, when an architectural model conforms to a style, it is possible to reuse style-specific monitoring techniques (i.e. what raw data are extracted from monitoring the system and how these data map to high-level notions in the architectural model), style-specific repair rules (i.e. how to adapt to constraint violations) and style-specific adaptation operators (i.e. actions one can perform to repair the system).

An important aspect of Acme lies in the design decision whereby the task of planning adaptation in the autonomic loop is decoupled from the actual sensors and effectors used, which are dependent of the type of system being augmented with autonomicity.

4.4.2.3 *C2/xADL.* In C2/xADL [Oreizy et al. 1998; Dashofy et al. 2002b], an important contribution lies in starting with an old architectural model and a new one based on recent monitoring data, and then computing the difference between the two in order to create a repair plan. Given the architecture model of the system, the repair plan is analysed to ascertain that the change is valid (at least at the architectural description level). The repair plan is then executed on the running system without restarting it. This involves various stages: terminating a component that is to be replaced and suspending any components and connectors bordering the affected area; removing components and connectors and adding new ones as defined by the repair plan; resuming components and connectors bordering affected areas. This approach is based on a predefined set of rules. However, an alternative to being given old and new architecture models could be to describe an architecture that would best deal with a situation and determine the changes necessary to evolve the current architecture to reach the ideal one.

4.4.3 *Process-coordination approach.* Adaptation plans can also result from defining the coordination of processes executed in the managed elements. As an example, we would like to describe Little-JIL.

Little-JIL [Wise et al. 2000] is a coordination language for planning a task and coordinating the components that will execute specific sub-tasks in the plan. Thus, Little-JIL can be used to model the task the managed elements perform and the various components that can take care each sub-task, resulting in a tree-based graphical description of tasks and their sub-tasks. Should a particular component not be available for a sub-task, or the conditions prevent the component from successfully completing it, Little-JIL allows us to specify what alternatives are available to complete that sub-task. Thus, it can thus be used to model adaptation plans for the managed element in the autonomic manager. Little-JIL also models, among other things, which tasks must be executed before or after a task, which

tasks can run in parallel, what parameters (in a broad sense) need to be passed between tasks.

Little-JIL is not an architectural description language, but rather a process-coordination language (hence our decision to place it in its own section). However, Little-JIL has certain similarities with the architectural approach, in that Little-JIL allows us to define a description of a managed element's task that inherently contains the adaptation plan necessary when the managed element behaves abnormally. Instead in the policy-approach, adaptations plans must be explicitly defined, including how they are triggered by incoming sensor data.

4.5 Knowledge

The division between planning and the knowledge used to effect adaptation in the autonomic MAPE-K loop is quite fuzzy, as one would expect. The knowledge in an autonomic system can come from sources as diverse as the human expert (in static policy based systems [Bougaev 2005] to logs that accumulated data from probes charting the day-to-day operation of a system to observe its behaviour, which is used to train predictive models [Manoel et al. 2005; Shivam et al. 2006]. This section lists some of the main methods used to represent Knowledge in autonomic systems.

4.5.1 *Concept of Utility.* Utility is an abstract measure of 'usefulness' or benefit to, for example, a user. Typically a system's operation expresses its utility as a measure of things like the amount of resources available to the user (or user application programs), and the quality, reliability or accuracy of that resource etc. For example in an event processing system allocating hardware resources to users wishing to run transactions, the utility will be a function of allocated rate, allowable latency and number of consumers, e.g. [Bhola et al. 2006]. Another example is in a resource provisioning system where the utility is derived from the cost of redistribution of workloads once allocated or the power consumption as a portion of operating cost [Osogami et al. 2005; Sharma et al. 2003].

4.5.2 *Reinforcement learning.* Reinforcement learning is used to establish policies obtained from observing management actions. At its most basic it learns policies by trying actions in various system states and reviewing the consequences of each action [Sutton and Barto 1998]. The advantage of reinforcement learning is that it does not require an explicit model of the system being managed, hence its use in autonomic computing [Littman et al. 2004; Dowling et al. 2006]. However it suffers from poor scalability in trying to represent large state spaces, which also impacts on its time to train. To this end, a number of hybrid models have been proposed which either speed up training or introduce domain knowledge to reduce the state space, e.g. [Tesauro et al. 2006; Whiteson and Stone 2006].

4.5.3 *Bayesian Techniques.* As well as rule-based classification of policies to drive autonomicity, probabilistic techniques have been used throughout the self-management literature to provide a way to select from numbers of services or algorithms etc. For example, Guo [2003] shows how Bayesian Networks (BNs) are used in autonomic algorithm selection to find the best algorithm, whereas cost sensitive classification and feedback has been used to attribute costs to self-healing equations

to remedy failures [Littman et al. 2003].

5. MULTI-TIER SYSTEMS

An ultimate goal of autonomic computing is to automate management aspects of complex distributed systems, primarily e-commerce sites. These usually have a multi-tiered architecture comprised of a web-server tier (e.g. Apache), an application server tier (e.g. a J2EE Application Server) and a database tier (e.g. Oracle, DB2), which entails a high level of complexity. Different autonomic efforts focus on specific aspects of this architecture, but each tier will usually implement the entire MAPE-K loop independently. For instance, at the web-server tier Bigus et al. [2002] have used the ABLE multi-agent based autonomic toolkit (see Section 4.2.2) to auto-tune Apache web servers. Similar work on Apache auto-tuning was also carried out by Sterritt et al. [2005]. At the application server tier, Rutherford et al. [2002] have worked on adding reconfigurability at runtime in the Enterprise JavaBean component model, while Candea et al. [2003] added self-recovering to JBoss, an open-source J2EE application server. At the database tier, Markl et al. [2003] have created DB2 autonomic query optimiser, thus improving its performance.

Alternatively, Urgaonkar et al. [2005] worked on self-management at a coarser granularity, looking at entire multi-tier systems and creating an autonomic manager that determines, given performance loss of the multi-tier system, how available resource (server nodes) should be redistributed to the various tiers to improve total performance of the system. This dynamic provision, they argue, must take into account the unique properties of the different tiers, e.g. databases often cannot be replicated on-demand. Wildstrom et al. [2005] also look at allocating resources to the different tiers of a multi-tier applications, but consider this allocation problem within high-end servers. Another problem is deploying updates at a tier without seriously degrading responsiveness. This can be automated by gradually withdrawing servers, applying the necessary updates and reintroducing them into the tier [Valetto and Kaiser 2003].

There is a considerable amount research that investigates resource allocation more generally, dealing with the problem of allocating resources—mainly CPU, memory and I/O to disk and network—to multiple applications in a cluster of servers. Adaptation occurs in response to considerable changes in the monitored response time of the applications and sometimes also on the expected improvement of each application if resources are increased. This if usually done with a utility function that quantifies the desirability of an application's state as a function of the assigned resources [Bennani and Menasce 2005]. This takes into account that, at some point, given the current or expected load, adding resources will only have a minimal effect on some applications.

Another issue being researched is the interface between an autonomic system and a human system administrator (sysadmin). While with autonomic computing the human sysadmin will need to be less involved in repetitive tasks, he will still need to intervene during critical conditions, e.g. when a hardware fault occurs and hardware components need to be replaced, or when hardware or major software upgrades are necessary. During these critical conditions, it is crucial that the sysadmin be able to quickly gain a detailed view of a particular aspect of the system, understanding

what is going on and why it is going on, in order to quickly but correctly carry out repairs [Barrett et al. 2004]. Thus, autonomic computing does not eliminate the need for human administrators to be able gain access to the details of the inner workings of a system. Furthermore, increased autonomicity may only be introduced incrementally in a system, as the sysadmin learns to trust the autonomic managers.

6. DEGREES OF AUTONOMICITY

We have reviewed the literature from the point of view of the major contributors to the various components of the MAPE-K Loop reference model. However, to get a feel of cross-cutting concerns, we now examine how one can describe degrees of autonomicity and how that research matches those degrees.

IBM have proposed a set of Autonomic Computing Adoption Model Levels[6] that spans from Level 1: Basic, to Level 5: Autonomic. Briefly, Level 1 defines the state whereby system elements are managed by highly skilled staff who utilise monitoring tools and then make the require changes manually. IBM believes this is where most IT systems are today. Level 2 is known as Managed. This is where the system's monitoring tools collage information in an intelligent enough way to reduce the systems administration burden. Level 3 is entitled Predictive whereby more intelligent monitoring than Level 2 is carried out to recognise system behaviour patterns and suggest actions approved and carried out by IT staff. The Adaptive level is Level 4. Here the system uses the types of tools available to Level 3 system's staff but is more able to take action. Human interaction is minimised and it is expected that the performance is tweaked to meet service level agreements (SLAs[7]). Finally, the full autonomic level is Level 5, where systems and components are dynamically managed by business rules and policies, thus freeing up IT staff to focus on maintaining ever changing business needs. The problem with this model as a means to define autonomic computing is that it is too narrow (focusing on a specific type of traditional business support systems) and the differentiation between levels too vague to describe the diversity of self-management we have found in the work presented in this paper. Therefore we propose an alternative that we believe better represents the levels of self-management we have discovered.

Since this paper focuses on self-managing systems we are precluding work that would conform to Levels 1 through to 3 and focus on what would be deemed by IBM as Adaptive and autonomic computing only. We have defined four elements of autonomicity (and not levels) which bring out the area of focus in the research and to what degree, architecturally, that focus has been applied. Our elements are listed below; the reader should not consider them definitive as this is impossible in such a varied and dynamically growing field:

Support. We describe this as work that focuses on one particular aspect or component of an architecture to help improve the performance of the complete architecture using autonomicity. For example, focus on resource management or network

[6]About IBM Autonomic Computing: `http://www-03.ibm.com/autonomic/about_get_model.html`

[7]A service level agreement (SLA) is a negotiated agreement between two parties, usually a service provider and a consumer. It establishes the quality of service (QoS) that the consumer can expect to receiver (which may be probabilistically bound), whereby a higher (QoS) usually comes at a higher expense. An SLA may also define the penalty if the agreed QoS is not met.

support or administration tools that contain self-management properties.

Core. This is where the self-management function is driving the core application itself. That is, if the application's focus is to deliver multi-media data over a network and the work describes an end-to-end solution including network management and display, audio, etc., then we describe the self-management as core. However, the work carried under this heading is not taking higher-level human based goals (e.g. business or SLAs) into account.

Autonomous. There is again where a full end-to-end solution is produced, but typically the solution concerns more emergent intelligence and agent-based technologies where the system self-adapts to the environment to overcome challenges that can produce failure, but it is not measuring its own performance and adapting how it carries out its task to better fit some sort of performative goals. Many research in the field of agent-based systems could come under this category, however in this paper we are only interested in applications of agent-based technology that drive the self-management of a computing system that performs more than one application. For example, Pathfinder is interesting to us because the 'operating system' must self-manage itself to carry out the tasks of the buggy in extreme terrains, and these tasks are varied and many, but an agent-based system whose sole focus is achieve a an application-specific goal is not driving the system in a self-management way, and therefore is not included in this paper.

Autonomic. At this level, work concerns a full architecture and describes the work in terms of that architecture. However, the interest is in higher-level human based goals, e.g. business goals or SLAs are taken into account. This level is the same as the Level 5: Autonomic level and it involves a system that reflects its own performance and adapts itself accordingly.

There is an argument for a further level of interest. A fifth level could examine systems that are both autonomic and allow the intelligence to drive the self-management to grow and refine itself. We would deem this fifth level as *Closed-Loop*. In many, if not all of the systems we have examined for this paper, the logic of what to do when has been more or less set statically and only updated by the parameters being fed into the loop. A closed loop system would also evolve the logic that drives the system depending on how well the 'old' logic did. This is akin to the work carried out in AI and is an area of research we can see becoming of more interest to the autonomic community.

We have compiled a survey table (see Tables II and III) in which we have placed most of the work mentioned in this article in the categories that we have described. Projects are categorised by the degree of autonomicity (horizontally) and the part of the MAPE-K loop focussed on (vertically). While some of the choices are highly subjective, we have tried to place them in a category that highlights the major contribution of a project.

7. EMERGING APPLICATION AREAS CURRENTLY DOMINATING AUTONOMIC COMPUTING

At the time of writing, it has been 5 years since IBM's manifesto, and we are now beginning to observe concentrations of research emerging in key applications areas. This section summarises some of this work.

Table II. Categorisation of described or mentioned autonomic work.

Degrees of Autonomicity MAPE-K focus	Support	Core	Autonomous	Autonomic
Full architecture	Rutherford et al. 2002; Candea et al. 2003; Bennani and Menasce 2005; Sharma et al. 2003	Badger 2004; Garlan and Schmerl 2002a	Kenyon 2001; Muscettola et al. 1998; Burrell et al. 2004; McCann et al. 2006; Wieselthier et al. 2002	Kephart and Chess 2003; Bantz et al. 2003; Ganek and Friedrich 2006; Melcher and Mitchell 2004; Kaiser et al. 2002;2003; Urgaonkar et al. 2005; Wildstrom et al. 2005; Handsen et al. 2006; Strowes et al. 2006; Huebscher and McCann 2005; Thomson et al. 2006
Autonomic management	Paulson 2002; Zhang and Figueiredo 2006; Markl et al. 2003	Bhatti and Knight 1999; McCann and Crane 1998; Bigus et al. 2002; Kon et al. 2000; Hofmeister and Purtilo 1993; Wolf et al. 2000	Chakeres and Belding-Royer 2004; Johnson et al. 2001; Perkins and Bhagwat 1994; Heinzelman et al. 2000; McCann et al. 2005	*All above contain some element of management*
Monitoring	Agarwala et al. 2006; Roblee et al. 2005; Strickland et al. 2005; Xu et al. 2005; Sterritt et al. 2005; Diao et al. 2005; Agarwala et al. 2006; Khargharia et al. 2006; Moore et al. 2006; Brodie et al. 2005	Garlan et al. 2001; Gross et al. 2001		*All above contain some element of management*
Planning models	Zenmyo et al. 2006		Jennings 2000; Gleizes et al. 2000; Kumar and Cohen 2000	

Table III. Continuation of Table II.

Degrees of Autonomicity MAPE-K focus	Support	Core	Autonomous	Autonomic
Policies	Batra et al. 2002; Lutfiyya et al. 2001; Ponnappan et al. 2002; Damianou et al. 2000; Manoel et al. 2005; Agrawal et al. 2005			
Architecture models	Oreizy et al. 1998;1999	Garlan and Schmerl 2002b; Magee et al. 1995; Georgiadis et al. 2002		Dashofy et al. 2002b;2002a; Wise et al. 2000
Knowledge	Walsh et al. 2004; Bougaev 2005; Manoel et al. 2005; Shivam et al. 2006; Osogami et al. 2005			Bhola et al. 2006; Dowling et al. 2006; Littman et al. 2003;2004

7.1 Power Management

Power management has been looked at from two systems' perspectives; the data centre and wireless sensors networks (WSN)—this section examines data centres only (WSNs are reviewed in Section 7.3). It has been estimated that power equipment, cooling equipment, and electricity together are responsible for 63% of the total cost of ownership of the physical IT infrastructure of a data centre[8]. Such statistics have motivated research in self-adaptive systems that not only optimise resource management in terms of performance metrics but in terms of the power that a given algorithm or service will consume on a given infrastructure. Earlier work focused on the processor's power consumption for individual server nodes [Kandasamy et al. 2004] or power allocation of server clusters [Femal and Freeh 2005]. However more recently we are seeing power models that take Memory, network, and IO consumption into account [Khargharia et al. 2006]. Heat management is related to this and is also being explored [Moore et al. 2006]. Such work has already shown that a 72% saving in power consumption can be achieved using autonomic systems management [Khargharia et al. 2006].

[8]Project on Operating Systems and Architectural Techniques for Power and Energy Conservation, Rutgers University. Home page: `http://www.cs.rutgers.edu/~ricardob/power.html`

7.2 Data Centres, Clusters and GRID Computing Systems

These kinds of systems can be grouped together as they are essentially wide-area high-performance heterogeneous distributed clusters of computers used to run anything from scientific to business applications for many differing users. This brings with it extra complexity in maintaining such a geographically distributed system (which can potentially span to world-wide). The allocation of resources is complicated by the fact that the system is expected to provide an agreed quality of service (QoS); which could be formalized in a Service Level Agreement (SLA). This typically means that the dynamic nature of the system's load, up-times and uncertainties etc are taken into account not only when initially allocating resources, but while the application is using those resources. Hence it is an excellent challenge for autonomic computing as maintaining a given QoS under such diverse dynamic conditions, using ad-hoc manual tuning and component replacement when failed, is unachievable.

The two key areas of research in here are dynamic resource management and systems administration. In many of the examples of the latter, a profile of the human operator behaviour and how this relates to the system's tuning constants or operator action, is obtained. Alternatively, the profile can be where a system's registry or regular log file is being tracked and the changes in the registry values (log updates) denote a reaction to a problem. Effectively from these profiles or logs a set of actions and responses are obtained then a set of policies can be derived to drive the autonomicity; i.e. when a policy rule is broken an action is taken and what action is determined by what was learned by the profile. Examples of such systems can be found in [Manoel et al. 2005; Brodie et al. 2005; Zenmyo et al. 2006].

The former, resource management, however has received the most attention. Typically the data centre or GRID is described as a Service-Oriented Architecture (SOA) and therefore QoS determines the resource selection options. Users of such systems typically pay for an agreed SLA specifying the price of an invocation of a given service. Resource management is further complicated by the fact that one must assume that the user and application requirements are as dynamic as the systems on which they have issued their tasks. Typically this has lead to a grading of service levels as Platinum, Gold, Silver and Bronze. However what these levels actually mean depends on the system in question and their users, and varies greatly throughout the literature [Agarwala et al. 2006]. Nevertheless the introduction of SLAs means that the resource allocation optimisation problem is extended to include cost [Bennani and Menasce 2005]. The aim to simplify the dynamic performance management (i.e. process migration) of GRID and larger Data Centre systems as well as ease the complexity of porting software round such environments, has provoked a renewed interest in systems' virtualization. That is, the GRID is a virtual machine [Handsen et al. 2006]. This also allows the payment calculation to be simplified.

Nimrod/G [Abramson et al. 2002] is an example of an architecture for grid-based resource management. Resource consumers define a *utility model* (see also Section 4.5.1) of their resource demands and preferences, while brokers (i.e. the autonomic managers) generate strategies for allocating resources based on this model. At the core of Nimrod/G is Nimrod, a declarative parametric modelling language for

expressing a parametric experiment [Buyya et al. 2000]. Nimrod is used to create a plan for a parametric computing task, which is submitted to the Nimrod runtime system where it is run on a remote grid resource by a Nimrod/G agent which senses its environment and manages its execution accordingly, thus ensuring that deadlines and budget limits are met.

Nimrod/G highlights how autonomic computing has also found inspiration from economic systems. Its use of the utility model assign a quantitative benefit for the allocation of a certain amount of a type of resource. Other economic models used in autonomic systems include the auction model [Waldspurger et al. 1992; Nisan et al. 1998], bid-based proportional resource sharing [Chun and Culler 1999], and the community, coalition and bartering models (e.g. Condor, SETI@Home and MojoNation).

7.3 Ubiquitous Computing

The advancement in electronics that enable the integration of complex components into smaller devices coupled with recent developments in wireless, mobile communications, have contributed to the emergence of a new class of wireless ad-hoc networks: Sensor Networks. These typically consist of a sensor board with a number of sensors of different modalities which, when combined with a microprocessor and a low-power radio transceiver, forms a smart network-enabled node. A sensor network may deploy a huge number of nodes depending on the nature of the application. Such applications include medical services, battlefield operations, crisis response, disaster relief, environmental monitoring, premises surveillance, robotics etc. Currently, such systems are described at two levels of granularity, smart dust and general ubiquitous computing; but both are closely related and suffer from similar constraints (ie small amounts of resources and power). Smart dust motes are tiny, as the name suggests and can scale down to $1mm^2$ and consist of microfabricated sensors, a photodetector, passive and active optical transmitters, a signal processing unit, a solar cell and a thick-film battery [Kahn et al. 1999]. Typical projected applications are environmental monitoring and defence applications, which run relatively unsophisticated programs and each node is, more or less, homogenous - therefore at the moment there is not much need for an autonomic system (just elementary autonomy, though this is likely to change). On the other hand, ubiquitous computing concerns the building of intelligent environments [Weiser 1991] from a number of, potentially, heterogeneous devices such as sensor nodes, PDAs, PCs etc. The sheer complexity of installing and maintaining such a system and keeping it running in a robust way, easily lends itself to autonomic computing. The application areas for this range from monitoring vineyards to looking after the elderly in the home [Burrell et al. 2004; McCann et al. 2006].

The key assumptions motivating autonomic Wireless Sensor Networks (WSN) research are that sensing and processing is cheap but communicating costs. That is, as the sensors are battery driven the main cost lies in driving the wireless equipment and not the sensors or CPU (though the actual devices themselves can be all relatively inexpensive to buy). Therefore placing redundant sensors on the node boards is an economical solution to help with robustness. Deciding on when to use which sensor to carry out a given task depends on the current context of the task or user. Herein lies the majority of autonomic WSN work, which can be divided

into middleware solutions or more lightweight 'emergent' solutions. The middleware solutions typically run on higher end nodes (i.e. 32bit microprocessor nodes, called motes, which were developed from a collaboration of University of California Berkeley and Intel Research Berkeley laboratory[9] and fit the same model of autonomicity as the MAPE-K loop, for example [Strowes et al. 2006; Huebscher and McCann 2005; Thomson et al. 2006]. Other ubiquitous computing architectures consist of lower resource and lower powered devices (e.g. nodes consisting of 4MHz 4MIPS CPU with 1kByte of SRAM [McCann et al. 2006]. Such devices cannot process vast amounts of self-knowledge and therefore more lightweight autonomicity is required. Here we see examples of utility functions being used to allow the system to discern between service options to provide a given level of service [McCann et al. 2006]. This latter piece of work embeds autonomicity into the core (operating system equivalent) of the sensor nodes thus providing for self-management from the lowest level in the software stack. Most of the emergent self-management has focused on sensor node ad-hoc networking, which we discuss below.

7.3.1 *WSN Routing.* Potentially, sensors can be positioned carefully (e.g. in aware buildings [Huebscher and McCann 2004]) or even scattered randomly (e.g. some defense applications or environmental monitoring, e.g., Glasweb[10]). Either way, routing data from one node to another has received much attention where the aim is to minimise the time to deliver a packet whilst minimizing the energy consumed in delivering it (incidentally energy consumption is proportional to the square distance between the communicating nodes for a two dimensional Euclidean space [Wieselthier et al. 2002]). Essentially there are three approaches to WSN routing: multi-hop, direct and a hybrid of both. Multi-hop routes data from the source to the sink via each of the nodes one at a time. It has the advantage of working out a near optimal route without knowing the network topology, and is therefore very flexible and resilient in that nodes can come and go and be mobile. Representative examples come from [Chakeres and Belding-Royer 2004; Johnson et al. 2001; Perkins and Bhagwat 1994]. Unfortunately, for this mechanism to work all nodes in the network must be listening for packets to relay them. Karl and Willig [2005] show that the listening while idle time's power consumption is almost the same as receiving a transmission, therefore alternative research into direct routing has been carried out [Intanagonwiwat et al. 2000]. However, this work has the disadvantage in that it assumes that the data source and sink are within reachable distance, which is not always the case therefore this has lead to a hybrid approach. Examples of hybrid approaches are [Heinzelman et al. 2000; McCann et al. 2005]. Essentially, a cluster of nodes elects a cluster-head to represent their 'region'; the cluster-heads then form a multi-hop network. More autonomic versions of these techniques elect cluster-heads dynamically depending on availability or to spread the cost of battery power usage across the network more fairly (e.g. nodes close to a popular sink typically are used more often and therefore their batteries become more depleted) [McCann et al. 2005].

[9]Motes home page at Crossbow (manufacturer): `http://www.xbow.com/Products/Wireless_Sensor_Networks.htm`

[10]University of Southampton. Glacsweb home page. `http://envisense.org/glacsweb/index.html`, 2006

8. DISCUSSION AND CONCLUSIONS

Inspired by biology, autonomic computing has evolved as a discipline to create software systems and applications that self-manage in a bid to overcome the complexities and inability to maintain current and emerging systems effectively. To this end autonomic endeavours cover the broad span of computing from the applications, middleware, to the systems software layers residing on hardware platforms as diverse as GRIDs to Smart Dust sensor nodes, and are already demonstrating their feasibility and value. This brings with it many open challenges which are receiving less attention at the point in time we write this paper. Some of these we discuss below.

State-flapping is a cross-cutting concern for autonomic computing. This is where oscillation occurs between states or policies that potentially diminishes the optimal operation of the element that is being managed. This is something that is emerging out of the implemented systems that we are seeing more off and there are many techniques we can borrow from other sciences (e.g. control theory etc.) that can help with dampening and de-sensitising the adaptivity mechanisms. However we do not believe that the likes of control theory is a panacea to solving all autonomic aspects as it is less able to deal with systems that exhibit discrete or continuous behaviours or states that can be time varying with less well defined inputs in an ever changing systems environment.

Another current challenge is how we can evaluate how well an autonomic system is performing, given that increased performance or power is not as relevant as say its ability to meet a given SLA. Of course evaluation depends on the application or system in question. However the optimal solution is no longer the aim but a sub optimal solution that maintains service levels or basic robustness etc. As these systems contain more emergent behaviour this emergence requires us to reason about it and reflect if the emergent behaviour is what we wish; and to what degree. We can see further metrics such as convergence and time to converge to a stable state (however this is defined) as becoming more important; but this is not really being addressed by the current autonomic community. Alternatively, there have been recent soundings to create a competition within the autonomic community to establish a representative Grand Challenge Application that can allow differing techniques to be compared and rated. This is similar to older endeavours like the information science community's TREC Challenge9. However the challenge to building such an application lies in the fact that not only software is described but management policies, SLAs and power/systems costs as well as more traditional metrics like performance, throughput etc must be defined.

Another challenge facing this community lies in the ability to carry out robust software engineering, not only to provide solidly built autonomic systems but those that can interoperate with each other. Current software engineering practice defines a system in a more or less pre-implementation state where requirements have been agreed a priori. Given that the autonomic system must change states at runtime and that some of those states may emerge and are much less deterministic, there is a great challenge to provide new guidelines, techniques and tools to help the autonomic system developer. Interoperating systems that already exist with or without autonomic components is also made more complex. This requires that

more work be done on not only the policies and intelligence that are required to meet a system's goals, but how we communicate the semantics and the outcomes of the behaviours and states that were actioned as a result of the activation of the policies, rules etc. As you can see from our Tables II and III, there is a fair amount of work on end-to-end architectures and their theoretical design, but very little of it has been implemented fully. Yet, ultimately, how useful a system is will be determined by end-to-end properties.

As observed in the work in Section 5, the MAPE-K loop components can be structured in a hierarchical fashion with many levels of abstraction. Consequently, we need software engineering methods and theories that better handle abstraction while being suitable in their ability to represent dynamicity in a "live" ever-changing system. Furthermore, programming design has traditionally assumed ridged and guaranteed interaction models; this needs rethought in an open adaptive system in favour of more negotiation based interaction. To this end, we are beginning to see influences from economic modelling and game theory in this regard.

Trust is an important issue at the many levels of abstraction of the autonomic system; from understanding how the system works as an individual artefact to interoperating it with other systems. For a given stimuli, the system's ability to adapt correctly and maintain expected behaviour sets contributes to the degree of trust the user, and other systems, have in it. However this must not stifle the system's ability to adapt to reach its goal—a complex trade-off therefore emerges. Adaptations at the component level to optimise the service at that level should not produce interactions that would cause undesirable behaviour with other components or levels of abstraction. As a consequence there remains a need to establish trust models that allow us to define these relationships and communicate trust amongst components. Furthermore, given that many trust models in existence today assume some form of centralised body of trust, the autonomic community need to focus on more distributed solutions.

In conclusion, though autonomic computing has become increasingly interesting and popular it remains a relatively immature topic. However, as more disciplines become involved, the established research they bring can be reutilised, adding to the maturity of the area. We believe that in the future the topic will become integrated into general computing and not be seen as the separate area it is today. This is much like the topic of 'Distributed Systems', which is so omnipresent in all computing that it no longer is considered separate. So too is the destiny of autonomic computing which will be naturally embedded in the design process where all systems architectures will have reflective and adaptive elements.

ACKNOWLEDGMENTS

Autonomic computing has changed a lot since we first initiated this survey paper. Therefore we would like to thank the Referees for their insight to ensure that the breadth of seminal work was covered. We would also like to thank our colleagues in the autonomic community who, perhaps unwittingly, contributed to some of our thoughts through enthusiastic conversations at workshops and conferences.

REFERENCES

Abramson, D., Buyya, R., and Giddy, J. 2002. A computational economy for grid computing

and its implementation in the Nimrod-G resource broker. *Future Gener. Comput. Syst. 18,* 8, 1061–1074.

Agarwala, S., Chen, Y., Milojicic, D., and Schwan, K. 2006. QMON: QoS- and Utility-aware monitoring in enterprise systems. In *Proceedings of the 3rd IEEE International Conference on Autonomic Computing (ICAC)*. Dublin, Ireland.

Agrawal, D., Calo, S., Giles, J., Lee, K.-W., and Verma, D. 2005. Policy management for networked systems and applications. In *Proceedings of the 9th IFIP/IEEE International Symposium on Integrated Network Management*. 455–468.

Ananthanarayanan, R., Mohania, M., and Gupta, A. 2005. Management of conflicting obligations in self-protecting policy-based systems. In *Proceedings of the Second International Conference on Autonomic Computing (ICAC)*.

Anthony, R. 2006. Emergent graph colouring. In *Engineering Emergence for Autonomic Systems (EEAS), First Annual International Workshop, at the third International Conference on Autonomic Computing (ICAC)*. 2–13.

Badger, L. 2004. Self-regenerative systems (SRS) program abstract.

Bantz, D. F., Bisdikian, C., Challener, D., Karidis, J. P., Mastrianni, S., Mohindra, A., Shea, D. G., and Vanover, M. 2003. Autonomic personal computing. *IBM Systems Journal 42,* 1, 165–176.

Barrett, R., Maglio, P. P., Kandogan, E., and Bailey, J. 2004. Usable autonomic computing systems: the administrator's perspective. In *Proceedings of the First International Conference on Autonomic Computing (ICAC)*. 18–26.

Batra, V. S., Bhattacharya, J., Chauhan, H., Gupta, A., Mohania, M., and Sharma, U. 2002. Policy driven data administration. In *Proceedings of the Third International Workshop on Policies for Distributed Systems and Networks*.

Bennani, M. N. and Menasce, D. A. 2005. Resource allocation for autonomic data centers using analytic performance models. In *Proceedings of the Second International Conference on Autonomic Computing (ICAC)*. 229–240.

Bhatti, S. N. and Knight, G. 1999. Enabling QoS adaptation decisions for Internet applications. *Computer Networks 31,* 7, 669–692.

Bhola, S., Astley, M., Saccone, R., and Ward, M. 2006. Utility-aware resource allocation in an event processing system. In *Proceedings of 3rd IEEE International Conference on Autonomic Computing (ICAC)*. Dublin, Ireland, 55–64.

Bigus, J. P., Schlosnagle, D. A., Pilgrim, J. R., III, W. N. M., and Diao, Y. 2002. ABLE: A toolkit for building multiagent autonomic systems. *IBM Systems Journal 41,* 3, 350–371.

Bougaev, A. A. 2005. Pattern recognition based tools enabling autonomic computing. In *Proceedings of 2nd IEEE International Conference on Autonomic Computing (ICAC)*. 313–314.

Brodie, M., Ma, S., Lohman, G., Syeda-Mahmood, T., Mignet, L., and Modani, N. 2005. Quickly finding known software problems via automated symptom matching. In *Proceedings of 2nd IEEE International Conference on Autonomic Computing (ICAC)*.

Burrell, J., Brooke, T., and Beckwith, R. 2004. Vineyard computing: Sensor networks in agricultural production. *Pervasive Computing, IEEE 3,* 1, 38–45.

Buyya, R., Abramson, D., and Giddy, J. 2000. Nimrod/G: An architecture of a resource management and scheduling system in a global computational grid. In *HPC Asia*. 283–289.

Candea, G., Kiciman, E., Zhang, S., Keyani, P., and Fox, A. 2003. JAGR: An autonomous self-recovering application server. In *Proceedings of the Autonomic Computing Workshop*. 168–177.

Chakeres, I. D. and Belding-Royer, E. M. 2004. Aodv routing protocol implementation design. In *ICDCSW '04: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - W7: EC (ICDCSW'04)*. IEEE Computer Society, Washington, DC, USA, 698–703.

Chun, B. and Culler, D. 1999. Market-based proportional resource sharing for clusters. Tech. rep., University of California, Berkeley.

Cobleigh, J. M., Osterweil, L. J., Wise, A., and Lerner, B. S. 2002. Containment units: a hierarchically composable architecture for adaptive systems. *SIGSOFT Softw. Eng. Notes 27,* 6, 159–165.

Damianou, N., Dulay, N., Lupu, E., and Sloman, M. 1999. Ponder: A language for specifying security and management policies for distributed systems. Tech. rep., Imperial College London.

Damianou, N., Dulay, N., Lupu, E., and Sloman, M. 2000. Ponder: A language for specifying security and management policies for distributed systems. Tech. rep., Imperial College London.

Dashofy, E. M., van der Hoek, A., and Taylor, R. 2002a. An infrastructure for the rapid development of XML-based architecture description languages. In *Proceedings Of the 24th International Conference on Software Engineering (ICSE), Orlando, Florida*.

Dashofy, E. M., van der Hoek, A., and Taylor, R. N. 2002b. Towards architecture-based self-healing systems. In *Proceedings of the first workshop on Self-healing systems*.

Diao, Y., Hellerstein, J. L., Parekh, S., Griffith, R., Kaiser, G., and Phung, D. 2005. Self-managing systems: A control theory foundation. In *Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS)*.

Dowling, J., Curran, E., Cunningham, R., and Cahill, V. 2006. Building autonomic systems using collaborative reinforcement learning. *Knowledge Engineering Review Journal Special issue on Autonomic Computing*. Cambridge University Press.

Femal, M. E. and Freeh, V. W. 2005. Boosting data center performance through non-uniform power allocation. In *Proceedings of the Second International Conference on Autonomic Computing (ICAC)*.

Ganek, A. and Friedrich, R. J. 2006. The road ahead—achieving wide-scale deployment of autonomic technologies. In *Chairing the Town hall meeting at the 3rd IEEE International Conference on Autonomic Computing*. Dublin, Ireland.

Garlan, D. and Schmerl, B. 2002a. Exploiting architectural design knowledge to support self-repairing systems. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*.

Garlan, D. and Schmerl, B. 2002b. Model-based adaptation for self-healing systems. In *Proceedings of the first workshop on Self-healing systems*.

Garlan, D., Schmerl, B., and Chang, J. 2001. Using gauges for architecture-based monitoring and adaptation. In *Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia*.

Garlan, D., Schmerl, B., Cheng, S.-W., Sousa, J. P., Spitznagel, B., and Steenkiste, P. 2002. Using architectural style as a basis for system self-repair. In *Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture*.

Georgiadis, I., Magee, J., and Kramer, J. 2002. Self-organising software architectures for distributed systems. In *Proceedings of the first workshop on Self-healing systems*.

Gleizes, M.-P., Link-Pezet, J., and Glize, P. 2000. An adaptive multi-agent tool for electronic commerce. In *Proeedings of the IEEE 9th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE)*.

Gross, P. N., Gupta, S., Kaiser, G. E., Kc, G. S., and Parekh, J. J. 2001. An active events model for systems monitoring. In *Proceedings of the Working Conference on Complex and Dynamic Systems Architectures*.

Guo, H. 2003. A bayesian approach for autonomic algorithm selection. In *Proceedings of the IJCAI workshop on AI and autonomic computing: developing a research agenda for self-managing computer systems*. Acapulco, Mexico.

Handsen, J. G., Christiansen, E., and Jul, E. 2006. The laundromat model for autonomic cluster computing. In *Proceedings of 3rd IEEE International Conference on Autonomic Computing (ICAC)*.

Heinzelman, W. R., Chandrakasan, A., and Balakrishnan, H. 2000. Energy-efficient communication protocol for wireless microsensor networks. In *HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8*. IEEE Computer Society, Washington, DC, USA, 8020.

Hofmeister, C. and Purtilo, J. M. 1993. Dynamic reconfiguration in distributed systems: Adapting software modules for replacement. In *Proceedings of the 13th International Conference on Distributed Computing Systems*. 101–110.

Horn, P. 2001. Autonomic Computing: IBM's perspective on the state of Information Technology. IBM Research.

Huebscher, M. C. and McCann, J. A. 2004. Adaptive middleware for context-aware applications in smart-homes. In *Proceedings of the 2nd Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC)*.

Huebscher, M. C. and McCann, J. A. 2005. Using real-time dependability in adaptive service selection. In *Proceedings of the International Conference on Autonomic and Autonomous Systems (ICAS)*. 76–81.

IBM. 2003. An architectural blueprint for autonomic computing. Tech. rep., IBM.

Intanagonwiwat, C., Govindan, R., and Estrin, D. 2000. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of MOBICOM*. 56–67.

Jackson, D. 2002. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology 11,* 2, 256–290.

Jennings, N. R. 2000. On agent-based software engineering. *Artificial Intelligence 117,* 2 (Mar.), 277–296.

Johnson, D. B., Maltz, D. A., and Broch, J. 2001. Dsr: The dynamic source routing protocol for multihop wireless ad hoc networks. In *Ad Hoc Networking*, C. Perkins, Ed. Addison-Wesley, Chapter 5, 139–172.

Kahn, J. M., Katz, R. H., and Pister, K. S. J. 1999. Next century challenges: mobile networking for "smart dust". In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*. ACM Press, New York, NY, USA, 271–278.

Kaiser, G., Gross, P., Kc, G., Parekh, J., and Valetto, G. 2002. An approach to autonomizing legacy systems. In *Proceedings of the Workshop on Self-Healing, Adaptive and Self-MANaged Systems*.

Kaiser, G., Parekh, J., Gross, P., and Valetto, G. 2003. Kinesthetics eXtreme: An external infrastructure for monitoring distributed legacy systems. In *Proceedings of the Autonomic Computing Workshop at the Fifth Annual International Workshop on Active Middleware Services (AMS)*.

Kaiser, G. and Valetto, G. 2000. Ravages of time: Synchronized multimedia for internet-wide process-centered software engineering environments. In *Proceedings of the 3rd ICSE Workshop on Software Engineering over the Internet*.

Kamoda, H., Yamaoka, M., Matsuda, S., Broda, K., and Sloman, M. 2005. Policy conflict analysis using free variable tableaux for access control in web services environments. In *Proceedings of the Policy Management for the Web Workshop at the 14th International World Wide Web Conference (WWW)*.

Kandasamy, N., Abdelwahed, S., and Hayes, J. P. 2004. Self-optimization in computer systems via on-line control: application to power management. In *Proceedings of the Second International Conference on Autonomic Computing (ICAC)*. 54–61.

Karl, H. and Willig, A. 2005. *Protocols and Architectures for Wireless Sensor Networks*. Wiley.

Kenyon, H. S. 2001. Battlefield cognizance tool points to future. *SIGNAL Magazine*.

Kephart, J. O. and Chess, D. M. 2003. The vision of autonomic computing. *Computer 36,* 1 (Jan.), 41–50.

Kephart, J. O. and Walsh, W. E. 2004. An artificial intelligence perspective on autonomic computing policies. In *Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks*. 3–12.

Khargharia, B., Hariri, S., and Yousif, M. 2006. Autonomic power and performance management for computing systems. In *Proceedings of 3rd IEEE International Conference on Autonomic Computing (ICAC)*. Dublin, Ireland, 145–154.

Kon, F., Campbell, R. H., Mickunas, M. D., Nahrstedt, K., and Ballesteros, F. J. 2000. 2K: A distributed operating system for dynamic heterogeneous environments. In *9th IEEE International Symposium on High Performance Distributed Computing*. 201–210.

Kumar, S. and Cohen, P. R. 2000. Towards a fault-tolerant multi-agent system architecture. In *Proceedings of the fourth international conference on Autonomous agents*.

Lippman, A. 1999. Video coding for multiple target audiences. In *Proceedings of the IS&T/SPIE Conference on Visual Communications and Image Processing*, K. Aizawa, R. L. Stevenson, and Y.-Q. Zhang, Eds. 780–784.

Littman, M., Nguyen, T., and Hirsh, H. 2003. A model of cost-sensitive fault mediation. In *Proceedings of the IJCAI workshop on AI and autonomic computing: developing a research agenda for self-managing computer systems*. Acapulco, Mexico.

Littman, M. L., Ravi, N., Fenson, E., and Howard, R. 2004. Reinforcement learning for autonomic network repair. In *ICAC*. 284–285.

Lobo, J., Bhatia, R., and Naqvi, S. 1999. A policy description language. In *AAAI '99/IAAI '99: Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence*. American Association for Artificial Intelligence, Menlo Park, CA, USA, 291–298.

Lupu, E. and Sloman, M. 1997. Conflict analysis for management policies. In *Proceedings of the Fifth IFIP/IEEE International Symposium on Integrated Network Management*. 430–443.

Lupu, E. and Sloman, M. 1999. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering - Special Issue on Inconsistency Management 25,* 6, 852–869.

Lutfiyya, H., Molenkamp, G., Katchabaw, M., and Bauer, M. A. 2001. Issues in managing soft qos requirements in distributed systems using a policy-based framework. In *POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks*. Springer-Verlag, London, UK, 185–201.

Lymberopoulos, L., Lupu, E., and Sloman, M. 2003. An adaptive policy-based framework for network services management. In *Journal of Network and Systems Management*. Vol. 11. Special Issue on Policy-based Management.

Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. 1995. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*. Springer-Verlag, London, UK, 137–153.

Magee, J. and Sloman, M. 1989. Constructing distributed systems in conic. *IEEE Trans. Softw. Eng. 15,* 6, 663–675.

Mainsah, E. 2002. Autonomic computing: the next era of computing. *Electronics & Communication Engineering Journal 14,* 1 (Feb.), 2–3.

Manoel, E., Nielsen, M. J., Salahshour, A., and Sampath, S. 2005. *Problem Determination Using Self-Managing Autonomic Technology*. IBM Redbooks. ISBN 073849111X.

Markl, V., Lohman, G. M., and Raman, V. 2003. LEO: An autonomic query optimizer for DB2. *IBM Systems Journal 42,* 1, 98–106.

McCann, J. A. and Crane, J. S. 1998. Kendra: Internet distribution & delivery system an introductory paper. In *Proc. SCS EuroMedia Conference, Leicester, UK*, A. Verbraeck and M. Al-Akaidi, Eds. Society for Computer Simulation International, 134–140.

McCann, J. A., Huebscher, M., and A., H. 2006. Context as autonomic intelligence in a ubiquitous computing environment. *International Journal of Internet Protocol Technology (IJIPT) special edition on Autonomic Computing,*.

McCann, J. A., Navarra, A., and Papadopoulos, A. 2005. Connectionless Probabilistic (CoP) routing: an efficient protocol for mobile wireless ad-hoc sensor networks. In *Proceedings of 24th IEEE international performance, computing and communications conference,*. Phoenix, AZ, USA.

Melcher, B. and Mitchell, B. 2004. Towards an autonomic framework: Self-configuring network services and developing autonomic applications. *Intel Technology Journal 8,* 4 (Nov.), 279–290.

Moore, J., Chase, J., and Ranganathan, P. 2006. Weatherman: Automated, online, and predictive thermal mapping and management for data centers. In *Proceedings of the Second International Conference on Autonomic Computing (ICAC)*. Dublin, Ireland, 155–164.

Muscettola, N., Nayak, P. P., Pell, B., and Williams, B. C. 1998. Remote agent: to boldly go where no AI system has gone before. *Artificial Intelligence 103,* 1-2, 5–47.

Nisan, N., London, S., Regev, O., and Camiel, N. 1998. Globally distributed computation over the Internet — the POPCORN project. In *ICDCS '98: Proceedings of the The 18th International Conference on Distributed Computing Systems*. IEEE Computer Society, Washington, DC, USA, 592.

Oreizy, P., Gorlick, M., Taylor, R., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., and Wolf, A. 1999. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems 14,* 3, 54–62.

Oreizy, P., Medvidovic, N., and Taylor, R. N. 1998. Architecture-based runtime software evolution. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*. IEEE Computer Society, Washington, DC, USA, 177–186.

Osogami, T., Harchol-Balter, M., and Scheller-Wolf, A. 2005. Analysis of cycle stealing with switching times and thresholds. *Perform. Eval. 61,* 4, 347–369.

Parekh, J., Kaiser, G., Gross, P., and Valetto, G. 2003. Retrofitting autonomic capabilities onto legacy systems. Tech. Rep. CUCS-026-03, Columbia University.

Paulson, L. D. 2002. Computer system, heal thyself. *Computer 35,* 8 (Aug.), 20–22.

Perkins, C. E. and Bhagwat, P. 1994. Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers. In *SIGCOMM '94: Proceedings of the conference on Communications architectures, protocols and applications*. ACM Press, New York, NY, USA, 234–244.

Ponnappan, A., Yang, L., Pillai, R., and Braun, P. 2002. A policy based qos management system for the intserv/diffserv based internet. In *Proceedings of the Third International Workshop on Policies for Distributed Systems and Networks*. 159–168.

Roblee, C., Berk, V., and Cybenko, G. 2005. Implementing large-scale autonomic server monitoring using process query systems. In *Proceedings of the Second International Conference on Autonomic Computing (ICAC)*. 123–133.

Russel, S. and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice Hall.

Rutherford, M. J., Anderson, K., Carzaniga, A., Heimbigner, D., and Wolf, A. L. 2002. Reconfiguration in the Enterprise Javabean component model. In *Proceedings of the IFIP/ACM Working Conference on Component Deployment, Berlin, Germany*. 67–81.

Sharma, V., Thomas, A., Abdelzaher, T., Skadron, K., and Lu, Z. 2003. Power-aware qos management in web servers. In *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium*. IEEE Computer Society, Washington, DC, USA, 63.

Shivam, P., Babu, S., and Chase, J. 2006. Learning application models for utility resource planning. In *Proceedings of 3rd IEEE International Conference on Autonomic Computing (ICAC)*. Dublin, Ireland, 255–264.

Sloman, M. 1994. Policy driven management for distributed systems. *Journal of Network and Systems Management 2,* 4, 333–360.

Sterritt, R., Smyth, B., and Bradley, M. 2005. PACT: Personal autonomic computing tools. In *Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS)*. IEEE Computer Society, Washington, DC, USA, 519–527.

Strickland, J. W., Freeh, V. W., Ma, X., and Vazhkudai, S. S. 2005. Governor: Autonomic throttling for aggressive idle resource scavenging. In *Proceedings of the Second International Conference on Autonomic Computing (ICAC)*.

Strowes, S., Badr, N., Dulay, N., Heeps, S., Lupu, E., Sloman, M., and Sventek, J. 2006. An Event Service Supporting Autonomic Management of Ubiquitous Systems for e-Health. In *Proceedings of Intl. Workshop on Distributed Event-Based Systems*.

Sutton, R. S. and Barto, A. G. 1998. *Reinforcement learning: An Introduction*. MIT Press, Cambridge, MA, USA.

Tesauro, G., Das, R., Jong, N., and Bennani, M. 2006. A hybrid reinforcement learning approach to autonomic resource allocation. In *Proceedings of 3rd IEEE International Conference on Autonomic Computing (ICAC)*. Dublin, Ireland, 65–73.

Thomson, G., Stevenson, G., Terzis, S., and Nixon, P. 2006. A self-managing infrastructure for ad-hoc situation determination. In *4th International Conference on Smart Homes and Health Telematics (ICOST2006)*. IOS Press, Belfast, Northern Ireland, UK.

Urgaonkar, B., Shenoy, P., Chandra, A., and Goyal, P. 2005. Dynamic provisioning of multi-tier internet applications. In *Proceedings of the Second International Conference on Autonomic Computing (ICAC)*. 217–228.

Valetto, G. and Kaiser, G. 2002. A case study in software adaptation. In *Proceedings of the first workshop on Self-healing systems*. 73–78.

Valetto, G. and Kaiser, G. 2003. Using process technology to control and coordinate software adaptation. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society, Washington, DC, USA, 262–272.

Waldspurger, C. A., Hogg, T., Huberman, B. A., Kephart, J. O., and Stornetta, W. 1992. Spawn: a distributed computational economy. *IEEE Transactions on Software Engineering 18*, 103–117.

Walsh, W. E., Tesauro, G., Kephart, J. O., and Das, R. 2004. Utility functions in autonomic systems. In *Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*. 70–77.

Weiser, M. 1991. The computer for the 21st century. *Scientific American 265,* 3 (Sept.), 94–104.

Whiteson, S. and Stone, P. 2006. Evolutionary function approximation for reinforcement learning. *Journal of Machine Learning Research 7*, 877–917.

Wieselthier, J. E., Nguyen, G. D., and Ephremides, A. 2002. Energy-efficient broadcast and multicast trees in wireless networks. *Mob. Netw. Appl. 7,* 6, 481–492.

Wildstrom, J., Stone, P., Witchel, E., Mooney, R., and Dahlin, M. 2005. Towards self-configuring hardware for distributed computer systems. In *Proceedings of the Second International Conference on Autonomic Computing (ICAC)*. 241–249.

Wise, A., Cass, A. G., Lerner, B. S., Call, E. K. M., Osterweil, L. J., and Jr., S. M. S. 2000. Using Little-JIL to coordinate agents in software engineering. In *Automated Software Engineering Conference (ASE 2000)*.

Wolf, A., Heimbigner, D., Knight, J., Devanbu, P., Gertz, M., and Carzaniga, A. 2000. Bend, don't break: Using reconfiguration to achieve survivability. In *Proceedings of the Third Information Survivability Workshop*.

Wooldridge, M. and Jennings, N. 1995. Intelligent agents: Theory and practice. *Knowledge Engineering Review 10,* 2, 115–152.

Xu, J., Adabala, S., and Fortes, J. A. B. 2005. Towards autonomic virtual applications in the in-vigo system. In *Proceedings of the Second International Conference on Autonomic Computing (ICAC)*.

Zenmyo, T., Yoshida, H., and Kimura, T. 2006. A self-healing technique based on encapsulated operation knowledge. In *Proceedings of 3rd IEEE International Conference on Autonomic Computing (ICAC)*. Dublin, Ireland, 25–32.

Zhang, J. and Figueiredo, R. 2006. Autonomic feature selection for application classification. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*.