Stigmergic Interoperability for Autonomic Systems:
Managing Complex Interactions in Multi-Manager Scenarios

Thaddeus Eze
Computer Science Department
University of Chester, Cheshire, United Kingdom
t.eze@chester.ac.uk

Richard Anthony
Computing and Information Systems Department
University of Greenwich, London, United Kingdom
R.J.Anthony@gre.ac.uk

***Abstract* —The success of autonomic computing has led to its popular use in many application domains, leading to scenarios where multiple autonomic managers (AMs) coexist, but without adequate support for interoperability. This is evident, for example, in the increasing number of large datacentres with multiple managers which are independently designed. The increase in scale and size coupled with heterogeneity of services and platforms means that more AMs could be integrated to manage the arising complexity. This has led to the need for interoperability between AMs. Interoperability deals with how to manage multi-manager scenarios, to govern complex coexistence of managers and to arbitrate when conflicts arise. This paper presents an architecture-based stigmergic interoperability solution. The solution presented in this paper is based on the Trustworthy Autonomic Architecture (TAArch) and uses stigmergy (the means of indirect communication via the operating environment) to achieve indirect coordination among coexisting agents. Usually, in stigmergy-based coordination, agents may be aware of the existence of other agents. In the approach presented here in, agents (autonomic managers) do not need to be aware of the existence of others. Their design assumes that they are operating in 'isolation' and they simply respond to changes in the environment. Experimental results with a datacentre multi-manager scenario are used to analyse the proposed approach.**

Keywords – interoperability; stigmergy; autonomic system; multi-agent system; trustworthy architecture; trustability; validation; datacentre; dependability; stability; autonomic architecture

I. INTRODUCTION

Autonomic Computing has progressively grown to become a mainstream concept. Many mechanisms and techniques have been successfully explored and this success has led to multi-manager system scenarios where multiple AMs coexist and/or interact (directly or indirectly) within the same system. This is evident, for example, in the increasing availability of large datacentres with multiple [heterogeneous] managers which are independently designed [1, 2]. Coupled with heterogeneity of services and platforms, this leads to the possibility of integrating more AMs to achieve a particular goal, e.g., datacentre optimisation. This ultimately leads to conflicts ranging from cancellation or over-compensation effects at the simplest end of the spectrum, to system instability, and lack of predictability at the other end. There is therefore the need for interoperability between AMs, to facilitate multi-manager scenarios, govern complex interactions between managers and to arbitrate when conflicts arise. Although several works, e.g., [3-5] have identified interoperability as a key challenge for future autonomic systems, we do believe that the challenge is already imminent.

The challenge of multi-manager interactions can be understandably enormous. This stems from the fact that, for example, components (including AMs) can be multi-vendor supplied: upgrades in one manager could trigger unexpected events; increasing scale can introduce bottlenecks; one manager may be unaware of the existence of another; and managers, though tested and perfected in isolation, may not have been wired at design to coexist with other managers. A typical conflict example is illustrated with a multi-manager datacentre scenario: consider a datacentre with two independent AMs working together (unaware of each other) to optimise the datacentre – a Performance Manager (*PeM*) optimises resource provisioning to maintain service level achievement. It does this by dynamically (re)allocating resources and maintaining a pool of idle servers to ensure high responsiveness to high priority applications. A Power Manager (*PoM*) seeks to optimise power usage (a major cost overheads of datacentres [6]) by shutting down servers that have been idle for a certain length of time. Each manager performs brilliantly in isolation, but by coexisting, the success of one manager defeats the goal of another; one seeks to shut down a server that another seeks to keep alive. The activities of one manager affect the costs of provisioning (e.g., delay, scheduling, and power consumption etc.) for another in one way or the other.

This paper presents a stigmergic interoperability solution to multi-agent coordination. The proposed solution is architecture-based as we posit that interoperability support should be designed in and integral at the architectural level, and not be treated as an add-on. The TAArch [7], which includes mechanisms and instrumentation to explicitly support interoperability and trustworthiness is used. Multi-manager coordination is achieved using stigmergy concepts.

The Stigmergic Phenomenon [8] is achieving indirect coordination among coexisting agents by means of indirect communication via the environment. That is, using their environment for indirect communication, the agents are able to sense and adjust their actions and this way efficient coordination is achieved. So the stigmergic interoperability solution provides indirect coordination between AMs in a multi-manager scenario without the need for planning (or pre-knowledge of the existence of other AMs), control or direct communications between coexisting AMs. Section II discusses the proposed stigmergic solution while Section III provides a distinction between the proposed solution and those in related works. Section IV presents datacentre-based implementation and empirical analysis. Section V concludes the work.

II. STIGMERGIC INTEROPERABILITY

The stigmergic interoperability utilises the process of stigmergy to facilitate the coexistence of agents without individual agents necessarily being aware of the existence or wiring of each other. The basic principle is that a particular AM detects others by observing the effects of their management actions on its own operating environment and especially in terms of the control and use of resources. Agents are context-aware and autonomically react to environmental changes by retuning their behaviour as appropriate. Environmental changes, e.g., unexpected fluctuation, data spikes, policy violation (or alteration), external adjustment of parameters, process conflict etc. are considered *AgentActions*, which are assumed, by all agents, to mean conflicting actions by another agent. *AgentActions* can also be caused by other factors that are considered *'normal'* behavior (e.g., resource contention) of the system. As soon as *AgentAction* is detected, an agent starts retuning its behaviour until a steady state is reached. In this paper, AMs are agents that are designed using TAArch architecture. TAArch is centered around hierarchical control loops, with three main components, operating on different timescales (for short and longer term adaptations) allowing the AM to monitor its own performance, correctness, and effect on the controlled system. This enables it to detect any instability caused in the system. The three main components are; the *AutonomicController* (AC), which makes self-management (adaptation) decisions, the *ValidationCheck* (VC), which monitors performance and correctness of AC and the *DependabilityCheck* (DC), which monitors long term adaptation impact and effectiveness on system. AMs need to predict the effect on the system of their own management actions, and by detecting deviations from this can deduce the presence of another manager acting on the same resource set. See [7] for more details of TAArch.

In the proposed interoperability approach, Trend Analysis (TA) logic is implemented in the DC component to enable the AM to automatically detect conflicts and using Dead-Zone (DZ) logic, the AM is able to regulate its behaviour as appropriate. DZ logic is a mechanism to prevent AMs from unnecessary, inefficient and ineffective control brevity when the system is sufficiently close to its target state. It provides a natural and powerful framework for achieving dependable self-management in autonomic systems by enabling AMs to adapt only when it is safe and efficient to do so, within a defined safety margin [16].

Fig. 1 is a multi-manager datacentre example: it comprises a pool of resources S_i (live servers), a pool of shutdown servers $\check{S}_i$ (ready to be powered and restored to S_i as need be), a list of applications A_j, a pool of services $\mathcal{U}$ (a combination of applications and their provisioning servers), and two autonomic managers AM1 (performance manager *PeM*) and AM2 (a power manager *PoM*) that optimise the entire datacentre. A_j and S_i are, respectively, a collection of applications supported (as services) by the datacentre and a collection of servers available to the manager for provisioning available services according to requests. As service requests arrive, *PeM* dynamically populates $\mathcal{U}$ to service the requests following the scheduling algorithm discussed in Section IV (A). $\mathcal{U}$ is defined by:

$$\mathcal{U} = \begin{cases} A_1\text{: } (S_{11}, S_{12}, S_{13}, \ldots, S_{1i}) \\ A_2\text{: } (S_{21}, S_{22}, S_{23}, \ldots, S_{2i}) \\ \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \\ A_j\text{: } (S_{j1}, S_{j2}, S_{j3}, \ldots, S_{ji}) \end{cases} \quad (1)$$

Where A_j: (S_{j1} … S_{ji}) means that (S_{j1} … S_{ji}) servers are currently allocated to Application A_j and j is the number of application entries into $\mathcal{U}$. Servers are retrieved and redeployed across applications. All the servers i in S_i are up and running (constantly available as desired by *PeM*) waiting for (re)deployment. The primary performance goal of *PeM* is to minimise oscillation and maximise stability and efficiency (including just-in-time service delivery) while the secondary performance goal is to maximise throughput. The goal of *PoM*, on the other hand, is to optimize power consumption. This task is simply achieved by shutting down any server that has been idle for a threshold time *T*. As a result, the actions of *PoM* can negate the goal of *PeM* causing conflict in the system.

To manage interoperability between *PeM* and *PoM*, Fig. 1 shows the communications and control within the components of the AMs. The managers take performance decisions which are then validated by their respective VC (VCpom and VCpem) for correctness. VC ensures continuous self-validation of the AM's behaviour and configuration against the AM's goals and also reflects on the quality of the AM's adaptation behaviour. A control feedback (CF) is generated if validation fails and with this feedback, the manager adjusts its decisions. The DC takes a longer-term validation oversight of the managers' behaviour and either allows a manager to carry on with its actions (if the check passes) or generates a recalibration feedback (RF) otherwise. DC contains other subcomponents (K), to achieve e.g., interoperability, stability etc. The stability subcomponent is usually configured using DZ logic. The interoperability subcomponent, in this case example, is configured using TA logic (which identifies patterns within streams of information) with a combined effect of exponential smoothing technique. The details of the logic usage are explained in Section IV. Note that the designer of the manager can define as many DC subcomponents as necessary.

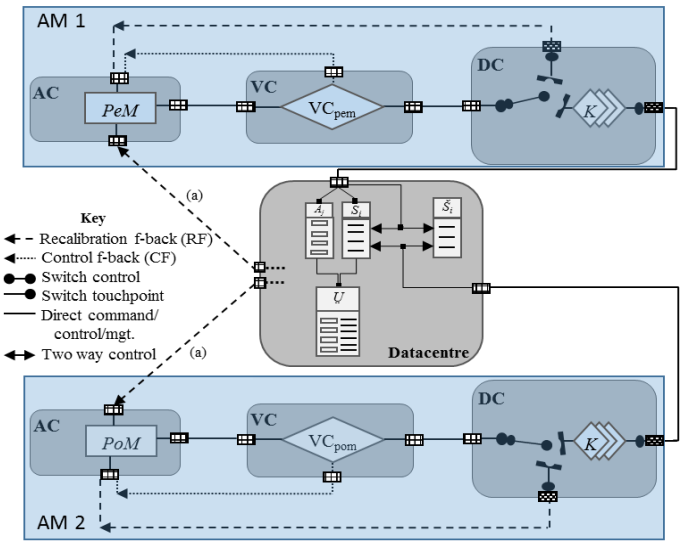

Figure 1: Stigmergic interoperability. AMs operate independently

The interoperability component learns and keeps track of the system's state following the historical decisions of the manager. If after a number of decision instances the manager senses a conflict with its decisions (based on expected versus actual system state), another RF is generated to retune the manager's decisions. For example, if after time *T*, *PoM* senses that the same set of servers it has shut down have repeatedly been restarted without it powering them, it concludes that another operation (probably a human, another AM, etc.) is not '*happy*' with *PoM*'s decisions. So, *PoM*'s DC generates a RF with an appropriate tuning parameter value (β) to throttle the size of *T* as shown in (2). By sensing the effects of its actions and dynamically throttling *T* within an acceptable boundary, *PoM* is able to coexist with any other AM with conflicting actions. Similarly, *PeM* can retune its behaviour, for example, if it senses that the set of servers it tries to keep running are constantly switched off. However, there are boundaries within which each AM's *cleverness* is limited, e.g., the size of *T* has a maximum limit. Notice that the two managers do not need to know any details or even the existence of each other. In real life, this is typical of two staff that share an office space but work at different times. If each returns on their next respective shift and finds the office rearranged, they will each adjust in their arrangement of the office until an accepted compromise structure is reached. This can be achieved without them ever meeting. DC provides extra capacity for a manager to dynamically throttle its behaviour to suit the goal of the system. This is in line with the stigmergic approach that enables coexisting AMs to achieve indirect coordination by means of indirect communication via the environment. That is, communicating indirectly using their environment, the AMs are able to sense the effects of each other's actions and adjust their own actions and thereby avoid conflict. So the stigmergic interoperability solution provides indirect coordination between AMs in a multi-manager scenario without the need for planning (or pre-knowledge of the existence of other AMs), control or direct communications between coexisting AMs. This provides efficient collaboration (as against competition) between coexisting AMs.

$$T_n = (T_{n-1} * \beta) \quad (2)$$

There are costs associated with the operations of a datacentre. These costs are affected in one way or the other by the actions of the managers. These and many other metrics are used to analyse the proposed solution in Section IV.

III. BACKGROUND

'Multi-agent systems' is a generic term referring to systems consisting of different sub-systems (e.g., AMs, agents) that cooperate (interact) with each other in order to achieve a common goal. The idea of a system with several components working together towards a common goal has been applied to an increasing number of domains including distributed systems, autonomic computing, supply chain, networks of networks and so on. Multi-agent coordination deals with the way the sub-systems interact with each other in the process of working together to achieve the common goal, and many techniques have been proposed. A detailed survey of multi-agent systems is presented in [9]. Multi-manager scenario, as described in this paper, is a situation requiring the cooperation of different AMs in the same system and this cooperation is referred to as interoperability. There are potential problems as a result of conflict-of-interest when these managers coexist. There is a growing concern that the lack of support for interoperability will become a barrier to progress for future systems. Several multi-agent coordination techniques have been proposed in the multi-agent systems community.

Architecture based multi-agent coordination has been demonstrated before. In [10], a multi-agent coordination in multi-robot system, based on genetic programming (GP), is discussed. To coordinate a coorperative task between robots, Liu and Iba [10] propose an approach called Evolutionary Subsumption, which applies GP to Brooks' subsumption architecture [11]. Results show that this approach is more efficient in emergence of multi-robots complex behaviors compared to other (e.g., direct GP and artificial neural network) approaches. This supports the idea of our solution which is achieving *interoperability-by-design* – interoperability support designed in and integral at the architectural level.

Natural systems such as social insects which utilise stigmergy show remarkable flexibility, robustness and self-organisation. These characteristics are sort after in modern systems. Researchers have demonstrated this in multi-agent systems. O'Reilly and Ehlers [12] have demonstrated the utilisation of stigmergy by software agents to interact with each other and to collectively solve certain tasks. They presented a methodology of mimicking stigmergy into a software system and argue that many software projects are deemed failures due to the inability of the software systems to adapt to changing business environments. A multi-agent stigmergic coordination in manufacturing control system has been presented in [13]. Coordination among the agents in the manufacturing control system is a direct reflection of the pheromone-based stigmergy in ant colonies. In this approach, the control system consists of agents (e.g., resource, product and order) that distribute pheromones (e.g., agents' connections, location and general information) within the environment (e.g., cyber world) in which they reside. According to the authours, sharing such global information on a collective environment reduces design cycle, products' time-to-market, order lead times and also facilitates flexibility in manufacturing control systems. However, just as in similar approaches, the agents are logically (and in some approaches, physically) connected together, which in actual sense, indicates that the agents are aware of the existence of others. This is different from our approach in which the agents (AMs) do not need to be aware of the existence of other agents. The AMs' design assumes that they are operating in 'isolation' and simply respond to changes in the environment (as a result of *AgentAction*). See the office share example in Section II. Our goal is to facilitate correct behavior when the '*isolation*' assumption is broken. TA logic, for example, enables AMs to easily infer the presence of other AMs by the kind (or nature) of environmental changes experienced. In this approach, an external adjustment of some parameters in a system (e.g., by a human user), whether correctly or erroneously applied, is considered an *AgentAction* by other agents. One sophistication of the stigmergic interoperability approach is that, no matter the

conflict or disturbance, AMs are designed to react (e.g., self-retuning) within the boundaries of the system's stated goals. This is because the AMs are designed using TAArch.

Reference [14] presents a clear demonstration of the need for interoperability mechanisms. In [14], two independently-developed AMs were implemented: the first dealt with application resource management (specifically CPU usage optimisation) and the second, a power manager, dealt with modulating the operating frequency of the CPU to ensure that the power cap was not exceeded. It was shown that without a means to collaboratively interact, both managers throttled and sped up the CPU without recourse to one another, thereby failing to achieve their intended optimisations and potentially destabilising the system. This is a case of direct conflict, our solution deals with both direct and indirect conflicts. Direct conflicts occur where AMs attempt to manage the same explicit resource while indirect conflicts arise when AMs control different resources, but the management effects of one have an undesirable impact on the management function of the other [15]. This latter type of conflict, in our opinion, is the most frequent and problematic, as there are such a wide variety of unpredictable ways in which such conflicts can occur.

Reference [15] evaluates the nature and scope of the interoperability challenges for autonomic systems, identifies requirements for a universal solution and propose a service-based approach to interoperability to handle both direct and indirect conflicts in a multi-manager scenario. In this approach, an Interoperability Service (IS) interacts with AMs through a dedicated interface and is able to detect possible conflicts of management interests. New AMs register their capabilities and requirements (in terms of the kind of services they provide and what aspects of the system they intend to manage) with the IS and the IS grants management rights only if no other AM in its database is managing the same aspect of the system to which management right is requested. In this way, the IS manages all interoperability activities by granting or withholding management rights to different AMs as appropriate. One challenge with this approach is that if a new AM is more capable of managing (e.g., in terms of efficiency) an aspect of the system that an existing AM is already managing, the new AM will be denied management rights.

Interface based approaches inhibit scalability because they require reconfiguring of interfaces each time a new AM is added. Conversely, in our approach AMs do not need recoding each time a new AM is added. They autonomically retune (modulate) their behaviour as soon as they sense process conflicts.

The research community has made valuable progress towards multi-manager interoperability but this progress has not yet led to a standardised approach. Although the current state of research represents a significant step, an equally significant issue is that they do not tackle the problem of unintended or unexpected interactions that can occur when independently developed AMs co-exist. Further from that, and more realistically, AMs may not necessarily need to know about the existence of others as they are designed in isolation (probably by different vendors) and operate differently (for different goals) without recourse to one another. So, to have close-coupled interoperability (i.e., where specific actions in one AM react to, or complement those of another), the source code and detailed functional specifications of each AM must be available to all AMs. This is near impossible, and where it is possible requires a rewiring (or recoding) of each AM whenever a new AM is added. Hence, this work looks to the autonomic architecture to provide a dynamic solution. This work posits that to avoid introducing further complexity whilst solving the interoperability problem, the autonomic architecture should envision (and provide for) interoperability support from scratch. That is to say, the autonomic architecture should provide mechanisms to facilitate the co-existance of, and interoperability between, multiple AMs.

IV. IMPLEMENTATION AND EMPIRICAL ANALYSIS

This Section presents experimental analysis of the proposed interoperability solution using a datacenter resource request and allocation management scenario. The datacentre scenario used is the same as the one outlined in Section II. The essence of this analysis is not to investigate datacentres per-se but to examine the performance effects of the proposed interoperability solution in a multi-manager datacentre scenario using easy-to-assess examples. The analysis will investigate the performance of the multi-manager datacentre scenario with and without interoperability solution.

It is important, however, to point out that the proposed interoperability solution works well in a closed-world model but has some limitations in an open-world model and so may not be relied on to reach convergence. Convergence defines a point at which system is stable and has reached a steady state. In a closed system, there are a definite number of actors (in this case AMs) that influence the environment and the individual actions of each AM can be tracked as a trend. In this way, it is possible for each AM to detect persistent actions that conflict with its actions and be able to readjust behaviour. However, in an open system, there are indefinite number of actors that can influence the environment. An actor in this model can be a third party that interferes with the system and this interference could be a one-off instance or several instances from different actors. For example, the office share scenario in Section II is a closed-world model but it becomes an open-world model if a third party (say, different office cleaners) randomly contributes to the office (re)arrangement. So, there are certain specific situations where it would not reach convergence in an opens system, however, in the general case it could, especially where a new AM component is added to the system and remains for sufficient time for the initial disturbance to disperse.

A. Scheduling and Resource Allocation

Several scheduling algorithms that optimise the performance of datacentres have been proposed e.g., [17-18]. Our work, on the other hand, does not propose any new scheduling algorithm. It uses a simple resource allocation technique to model the behavior of AMs within the datacenter, and measures their performance in terms of the effectiveness of resource request and allocation management.

In the simulation, service (application) requests arrive and are queued. If there are enough resources to service a particular request then it is serviced otherwise it remains in the queue (or may eventually be dropped). The AM checks for resource

availability and deploys server(s) according to the capacity of the request. The capacities of application requests and servers are defined by the units million instructions per second (MIPS). In this paper, 'capacity' is stated in terms of MIPS, i.e., the extent of its processing requirement. When a server is deployed, it is placed in a queue for a time defined by the variable *ProvisioningTime*. This queue simulates the time (delay) it takes to load or configure a server with necessary application. Recall from Equation (1) that any server can be (re)configured for different applications and so servers are not pre-configured. Servers are then 'Provisioned' after spending *ProvisioningTime* in the queue. The provisioning pool is populated on demand, as requests arrive. As a result of the lag between provisioning time and the rate of request arrival or some unforeseen process disruptions, some provisioned servers do overshoot the total resource needed for the application, and are thereby left redundant in the queue. As requests are fully serviced (completed), servers are released into the server pool and redeployed. Note that service level achievement (SLA) is calculated based on accepted requests and not rejected or dropped requests. The essence of the request queue is to allow the AM to accept requests only when it has enough resources to service them. So the AM could reject or drop the requests based on '*insufficient resources*', i.e., *RequestedCapacity* > *AvailableCapacity*. This process is continuous and the AM manages the system to the level of its sophistication.

B. Experimental Design and Metrics

The experimental scenario is designed and implemented using the TAArch application (built in C#) which is available on request. The experiment simulates two instances of a datacenter scenario with each having two AMs – *PeM* and *PoM* optimising resource allocation and power management respectively. In the first instance, represented as *DatacentreNoInt*, the AMs co-exist without any form of interoperability solution. This means that both AMs perform their tasks within the boundaries of their individual autonomic framework without recourse to one another. In this case, *PeM* and *PoM* are represented as *PeM_NoInt* and *PoM_NoInt* respectively. In the second instance, *DatacentreInt*, the AMs co-exist with the proposed stigmergic interoperability solution. This means that both AMs, while performing their tasks within the boundaries of their individual autonomic framework, are sensitive to external interference. Here, external interference is also defined as an *AgentAction*; any action or effect that alters the AM's expected system state. In this case, *PeM* and *PoM* are represented by *PeM_Int* and *PoM_Int* respectively.

Note that this work investigates the performance of a multi-manager datacentre with (*DatacentreInt*) and without (*DatacentreNoInt*) interoperability solution. The scope of the experiment focuses on the performance of datacentre AMs in resource request and allocation management activities under varying workloads. Although some workload parameters are sourced from experimental results of other research [19, 20], the designed experiments allow for the tailoring of all parameters according to user preferences. Simulations are designed to model several options of real datacentre scenarios. So, depending on what aspect is being investigated, the user can vary the workloads according to specific requirements. The result of every simulation analysis is relative to workload and the specific application configuration.

Performance Metrics:

The performance metrics are specifically chosen to reflect the impact of the interoperability solution in a multi-manager datacentre.

SLA: Service level achievement is the ratio of provided service to requested service. It measures the system's level of success in meeting request needs. Note that requests and services are not time bound, so the time it takes to complete a request does not count in this regard. The metric is defined as:

$$SLA = \begin{cases} \text{(i)}: ProvisionedCapacity/RequestedCapacitym \\ \text{(ii)}: AvailableCapacity/RunningCapacity \end{cases} \quad (3)$$

Where *ProvisionedCapacity* is the total deployed server capacity (excluding those in queue and including those already reclaimed back to the pool) and *RequestedCapacity* is the total capacity of requests (including completed requests). *AvailableCapacity* is *ProvisionedCapacity* minus *ReclaimedCapacity* while *RunningCapacity* is the total capacity of requests (excluding completed requests). There are two definitions for *SLA* (3): (i) is more of a whole picture consideration, considering the entire capacity activities of the system while (ii), which is used in our experiment, takes a real time view of the system, tracking to the minute details of the system with delay, completed requests and reclaimed server effects all considered. The reference value for *SLA* is 1: values above 1 indicate overprovisioning while values under 1 indicate shortfall.

PowerCoefficient: This represents the average server power consumption. That is, the average power a server consumes at any point in time for being active (switched on and running). This is measured in kilowatt (kW). According to [19, 20], on average, individual servers consume about 3.195 MWh worth of power. This value is scaled and *PowerCoefficient* is pegged at 3195 kWh in the simulations. This is indicative of real systems although actual values can vary significantly owing to a lot of factors (e.g., cooling, processor, machine type etc.). TAArch Application allows for the tailoring of all parameters according to user preferences. The usage of this variable is limited to investigating the impact of interoperability actions in terms of power consumption.

PowerConsumption (PC): This metric represents the aggregated power consumption per unit time for all idle servers, i.e., servers that are running but not yet deployed. It is important to consider these servers as they can as well be switched and powered only when needed. Although this could impact on SLA, the tradeoff in **power savings** may be worthwhile, and is one of the dynamic aspects of such a system. So if we assume that each server, on average, consumes *PowerCoefficient* kilowatts worth of power per second, then PC is calculated as:

$$PC = PowerCoefficient * \#IdleServers \quad (4)$$

PC is calculated at every time interval defined by *RequestRate*. Individual AM PC is different from the *general* PC. For general PC, number of idle servers will be the total of server count in S_i and $\check{S}_i$ pools while for individual AM (*Int* or *NoInt*) PC number of idle servers will be the total of server count in S_i pool:

$$PC = PowerCoefficient * (Server.Count + ShutServer.Count);$$
$$PC_{Int} = PowerCoefficient * Server.Count;$$
$$PC_{NoInt} = PowerCoefficient * Server.Count;$$

Note that as a result of individual operations of the autonomic managers, *Server.Count* for *DatacentreNoInt* will usually be different from that of *DatacentreInt*.

PowerSavings (PS): *PS* is calculated as the difference between general power consumption and individual AM power consumption:

$$PS = PC - PC_{(Int\ or\ NoInt)}$$

So, e.g., the *PS* for *DatacentreNoInt* will be calculated as:

$$PS_{NoInt} = PC - PC_{NoInt}$$

As *PoM* intends to optimise power usage, which also entails saving power, the PS metric will be useful to analyse the impact of the manager's power management capability.

Instability: Instability is the number of servers moved per second between pools in the datacentre. Moving servers around frequently is inefficient and increases provisioning overheads. The cost effect can be enormous in terms of cooling, power, and scheduling costs etc. Instability in terms of irregular and high rate of server movement from one pool to another is a costly, unsafe (due to the introduction of variable delays) and undesirable occurrence in datacentres. This is a potential situation when you have two AMs optimising the same datacentre as in the case example here.

C. Autonomic Manager Logic

AM logic details their individual control logic employed in order to achieve each one's performance goal. This explains the logical composition of each AM. There are two instances of each AM (*PeM_Int* and *PeM_NoInt*), i.e., with and without interoperability solution.

- **Performance Manager (PeM)**

PeM is directly responsible for dealing with application resource requests and allocation management. The AM receives requests and allocates resources according to the scheduling algorithm defined earlier. The first instance of this AM (*PeM_NoInt*) has no inbuilt interoperability solution.

- PeM_NoInt

As requests arrive, the AM checks for resource availability and deploys server(s) according to the capacity of the request. The server is placed in the provisioning pool which is constantly populated as requests arrive. The AM calculates an exponentially smoothed mean of the capacity of arriving requests in order to forecast the next expected request MIPS, i.e., it is used to predict requests:

```
smoothedAvgCapacityPeM_NoInt = (smoothingConstant *
avgAppCapacity) + ((1 - smoothingConstant) * oldMean);
```

With this forecast information, the AM constantly checks to ensure that the difference between the predicted MIPS and the available MIPS (idle server capacity ready for deployment) is not less than the equivalent of two servers. And if it is, the AM quickly checks and restores servers from the shutdown server pool ($\check{S}_i$). Procedure 1 is the algorithm that drives the server restoration process in the *PeM_NoInt* AM.

Procedure 1: Algorithm for checking and restoring servers

```
1: Calculate smoothedAvgCapacity
2: Calculate AvailableCapacity
3: Define a periodic Interval (PeM_NoIntTuningParam)
4: for every Interval
5:   if (AvailableCapacity – smoothedAvgCapacity)
                          < (ServerCapacity * 2)
6:   restore servers
7: next
```

This check ensures that, where possible, the AM maintains at least the capacity equivalent of two servers readily available for deployment (i.e., at least enough resources for current request and the next expected request). Checks are carried out at an interval defined by the *PeM_NoIntTuningParam* parameter. This ensures that the AM does not wait until the critical point before acting. So at every interval, the AM checks and restores servers on the $\check{S}_i$ pool.

- PeM_Int

The *PeM_Int* AM has an embedded interoperability solution based on the proposed interoperability solution (Fig. 1). In addition to all the functionalities of *PeM_NoInt* the *PeM_Int* AM performs further checks and retunes its behaviour. The AM tracks system state as it carries out checks at the specified interval defined by *PeM_IntTuningParam*. Each check is seen as an 'observation' instance and if on a periodic *IntObserve* observation the $\check{S}_i$ pool is not empty (signaling that the pool is being populated as it is being emptied by *PeM_Int*), the AM adjusts its checks interval (by increasing *PeM_IntTuningParam*) to reduce the rate at which it empties the $\check{S}_i$ pool (i.e., to be sympathetic to the other AM whose presence is implied, rather than to compete with it):

```
if ((serviceRequestCountPeM_Int - PeM_IntRefPoint) ==
                          PeM_IntTuningParam)
  {  PeM_IntObservationCount += 1;
     PeM_IntRefPoint = serviceRequestCountPeM_Int;
     if (PeM_IntObservationCount == IntObserve)
  {  PeM_IntObservationCount = 0;
     if (ShutServerCountInt != 0) //if Ši pool not empty
     { PeM_IntTuningParam += IntParamCount;
}       ... }     ...    }
```

Note: *PeM_IntTuningParam* parameter represents the initial time interval at which the *PeM_Int* AM checks to decide whether or not to power and restore servers that are down. Unlike *PeM_NoIntTuningParam*, it is dynamically adjusted by the *PeM_Int* AM. This parameter is measured in number of service requests.

A further internal set of observation iterations could be carried out. The tuning parameter is further adjusted if condition persists (i.e., persisted interference) after each further observation of the initial interval of observations. So, what happens here is that the AM powers on servers (restores servers from $\check{S}_i$) and keeps checking that there are enough reserves for prompt deployment. *PoM_Int* continues to shut servers down, which causes instability in the system as both AMs counter each other's actions. If *PeM_Int* senses that the restored servers are constantly put out-of-service, it relaxes its rate of re-powering the servers – this is because the whole essence is indirect collaboration rather than competition. In essence each AM has its own feedback loop but these are coupled indirectly by selected environmental parameters, facilitated by TAArch. If after a certain time (defined by the new check interval) the interference continues, the AM further relaxes the rate of its actions. This process is continuous, so adjustment is repeated until a stable condition is reached. This is demonstrated in detail in following experiments.

- **Power Manager (PoM)**

The power manager is directly responsible for power usage optimisation in the datacentre. The power optimisation method implemented by the AM is based on power conservation in which idle servers are shut down to conserve power. Other researchers have used different forms of power management. For example [21] discuses a power manager which optimises the power consumption of a server by adjusting its processor speed several times a second, and [22] discuses a power manager which is embedded in the firmware of a server and can use feedback control to precisely control the server's power consumption. While these are processor-level power management, the *PoM* AM conserves power by shutting idle servers and repowering them as need arises. This is sufficient to create conflicts with *PeM*, which seeks to keep as many servers running as possible in order to have enough capacity reserve *(and thus provides a suitable example on which to explore the stigmergic interoperability concepts)*. This form of power management technique is also used in [23] in which machines are turned on/off to conserve power.

- PoM_NoInt

Here, the AM checks and shuts down idle servers at a time interval defined by *PoM_NoIntTuningParam*. The idle servers are the same servers that *PeM_NoInt* considers as available resources. So in essence, when servers are shut down *AvailableCapacity* is depleted which in turn affects the performance of *PeM_NoInt*. So *PoM_NoInt* continues to check and shut down servers within a certain boundary. Procedure 2 is the algorithm that determines how *PoM_NoInt* checks and shuts down servers.

Procedure 2: Algorithm for checking and shutting down servers

```
1: int s = initial number of servers
2: Define a periodic Interval (PoM_NoIntTuningParam)
3: for every Interval
4:   int d = #AvailableServers //Servers.Count
5:  if (d > (s/5))
6:   Shut Sever[d-1]//shut the last server on Si pool
7:   Add Server[d-1] To Ši[]//add to Ši server pool
8: next
```

So what this means is that *PoM_NoInt* will continue to shut down idle servers as long as the number of servers in the S_i pool (available servers) is greater than one fifth of the total servers. (The DC component of *PoM_NoInt* is configured to stop shutting servers at (S_i count = (*server.sNumber* / 5)) because if the AM continues shutting servers beyond this point it will drag the entire datacentre to the brink of unresponsiveness which ultimately leads to under-provisioning and inefficiency.) This process continues regardless of the actions of the *PeM*. *PeM_NoInt* may at this point be restoring the servers to increase *AvailableCapacity* and this ultimately leads to high rate of server movement in the datacentre.

- PoM_Int

On the other hand, the embedded interoperability solution enables *PoM_Int* to sense conflicts and then readjusts its behaviour. The same method as in *PeM_Int* is used here. For example, the AM keeps count of servers in the $\check{S}_i$ pool (*listViewShutServer.Items.Count*) as it shuts and repowers servers and if on a periodic tenth check the server count does not match expected count (signifying *AgentAction*), the AM adjusts the tuning parameter:

```
if ((serviceRequestCountPeM_Int - PoM_IntRefPoint) ==
                PoM_IntTunningParam) //
{    PoM_IntObservationCount += 1;
     PoM_IntRefPoint = serviceRequestCountPeM_Int;
     if (PoM_IntObservationCount == 10)
       { PoM_IntObservationCount = 0;
         if (listViewShutServer.Items.Count <
                                  PoM_IntCheckPoint)
     {// if on a 10th observation S_i.Count doesn't match
      expected count
        PoM_IntTunningParam += 1; //adjusting parameter
     }  }
 //below is same as defined by Procedure 2 algorithm
 int d = listViewServer.Items.Count;
 if (d > (server.sNumber / 5)) // unsafe to shut servers
 {   listViewShutServer.Items.Add(listViewServer.Items[d-
1].Text);
  listViewServer.Items.Remove(listViewServer.Items[d-1]);
  PoM_IntCheckPoint = listViewShutServer.Items.Count;} }
```

The AM keeps adjusting the tuning parameter (*PoM_IntTuningParam*) until it senses stability in the datacentre. The observation process operates continuously, so whenever a new conflict arises the adjustment behavior begins again, to find a new compromise.

D. Experimental scenario and results analyses

To analyse the performance effects of the proposed interoperability solution on the datacentre case example, a scenario of varying application capacity with inconsistent request rate is used. This scenario replicates a situation where there is resource contention (as a result of hugely varied request capacities) and the possibility of abrupt and inefficient server deployment (as a result of inconsistent request rate, e.g., burst injection). This condition is perfect for testing the robustness of the interoperability solution. The effect of resource contention and irregular (sometimes erratic) request rate is usually rapid and frequent movement of servers between the various pools which the AMs will struggle to contend with. This is made worse when there is conflict between the AMs, with one restoring servers and another powering them down, which leads to more server movement. The robustness of the interoperability solution is tested by its level of sensitivity to this situation. This simulation can be replicated using the TAArch Application. Table I is a collection of the major parameters used in this simulation.

TABLE I. SIMULATION PARAMETERS

Parameter		Value
# of servers		400
# of applications		2
App capacity (MIPS)	App1	30000
	App2	15000
Request rate (initial)		1 req/sec
Server capacity (MIPS)		40000
Internal variables	RetrieveRate	5x
	RequestRateParam	10
	RetrieveRequestParam	0.2
	BurstSize	2500ms
	ServerProvisioningTime	3 (1.5 sec)
Managers (for *NoInt* and *Int*)		PeM & PoM
DZConst (initial)		1.5

- ***RetrieveRate***: Indicates rate at which requests are completed once simulation for service request completion is initiated. Value is relative to request rate – e.g., if value is 5, then it means service request completion is five times slower than rate of service request.
- ***RequestRateParam***: A constant used to adjust the possible range of request rate. The user of the TAArch Application can set request rate according to preference but this preference may not be accommodated within the available rate range. E.g., if the least available rate is 1 request/second and the user wishes to use 2 requests/second, the *RequestRateParam* parameter can be used to extend the available range. A higher value increases the range for a lower rate of request arrival.
- ***RetrieveRequestParam***: Tuning parameter indicating when to start shutting services (this simulates service request completion) – at which point some running requests are closed as completed. This value is measured as percentage of number of servers in use and has been restricted to value between 0.1 and 0.3 (representing 10% to 30%) because experiments show that it is the safest margin within which accurate results can be guaranteed. The datacentre is not completely settled below 10%, that is, the data generated below this point is insufficient for adequate analysis. Also, scenarios with few servers will yield inaccurate results beyond 30% mark. The higher the value of *RetrieveRequestParam* the earlier services start shutting ('shutting services' simulates service request completion).
- ***Burtstsize***: Indicates how long the user wants the burst (injected disturbance) to last. This value is measured in milliseconds. Burst is a disturbance introduced by the user to cause disruption in the system. This alters the smooth running of the system and AMs react to it differently. The nature of the disruption is in the form of sudden spike or significant shift in the rate of service request.
- ***ServerProvisioningTime***: Indicates how long it takes to load or configure a server with an application. This is relative to the rate of request arrival – it is measured as half the rate of request arrival e.g., the value of 3 will translate to 1.5 of rate of request arrival.
- ***DZConst***: The tuning parameter the AM uses to dynamically adjust dead-zone width (DZWidth). This variable has a significant effect on the system, and it was found experimentally that the initial value should be set at 1.5. The AM usually adjusts this value dynamically and there is also a provision to manually adjust the value during run time.

- Results

The results are based on the average of ten simulation runs. In every simulation run, there are 400 servers of 40000 MIPS capacity each to be shared amongst two applications (App1 and App2). This means there is a total of initial 16000000 MIPS to share between requests for App1 with 30000 MIPS and App2 with 15000 MIPS. Table 2 shows a distribution of requests and services for ten simulation runs.

TABLE II. HIGH LEVEL PERFORMANCE ANALYSIS OVER TEN SIMULATION RUNS

Runs	unused server		serviced request		queued request		deployed server	
	Int	*NoInt*	*Int*	*NoInt*	*Int*	*NoInt*	*Int*	*NoInt*
1	0	0	585	610	116	91	439	439
2	26	0	586	597	99	88	416	446
3	0	0	635	639	105	101	464	453
4	0	0	586	587	89	88	441	441
5	0	0	600	615	112	97	445	427
6	3	0	602	597	92	97	434	439
7	0	0	629	660	145	114	442	443
8	19	0	593	598	103	98	423	447
9	23	0	603	614	104	93	409	444
10	6	0	602	603	95	94	436	437
avg	**7.7**	**0**	**602.1**	**612**	**106**	**96.1**	**434.9**	**441.6**

Table II shows slight differences in performance optimisation between when interoperability solution is implemented (*Int*) and when it is not (*NoInt*). In terms of resource per service efficiency, for example, *NoInt* performed slightly better than *Int* with the ratio of 0.7216 : 0.7223. This is because of increased delay experienced in *Int* as a result of delayed (queued) requests following the burst. The tradeoff for *Int*, in this case, is an increased number of unused servers which means that more requests would be serviced in the long run. This relationship is reflected in the general performance

optimisation analysis (e.g., scheduling and costs, *SLA* etc.), in which both *Int* and *NoInt* outperformed each other intermittently. The *SLA* analysis (Fig. 2) corroborates this position and also shows that both datacentres gradually stabilised to optimal provisioning after a short time of under-provisioning. The tradeoff for *Int*'s slight low *SLA* performance is increased power savings as shown in Fig. 4. (Tradeoff between *SLA* and power savings has been discussed in [23]). However, there is significant performance difference in terms of power optimization analyses. Recall that the actions of the performance manager have enormous impact on the power manager whereas the performance manager, to some extent, mitigates the effects of the power manager's actions.

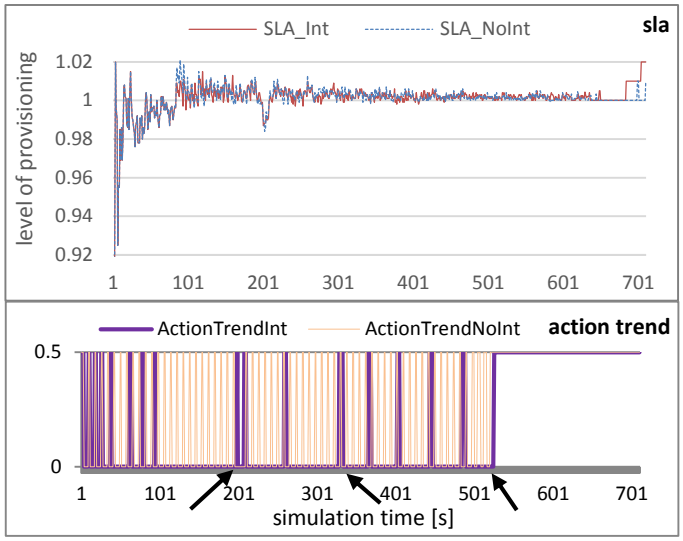

Fig. 2. The analysis of both datacentres' *SLA* and behaviour patterns (*ActionTrend*). Burst was introduced at 200s. The reference point value for SLA is 1 (indicating 100% – optimal provisioning): values above 1 indicate overprovisioning while values under 1 indicate shortfall. For *ActionTrend*, the level 0.5 was chosen arbitrarily as its numerical value is not significant – it is just used to indicate behaviour patterns (in terms of tuning and retuning actions) of AMs in both datacentres in the face of conflict. *DatacentreInt* achieved steady state at 320s and conflicts stopped at about 520s. These are indicated by the arrows.

Action trends in Fig. 2 reveal that *DatacentreNoInt* shows high level of instability in terms of inefficient movement of servers between logical pools in the datacentre. The behaviour trend in *NoInt* remained constant from start to finish and only experienced a minor jump when burst was injected. As shown, a burst of service requests was injected into the system at 200s and in both scenarios the datacentres recovered quite quickly. Behaviour pattern in *DatacentreInt* (*Int*) reveals a level of dynamic self-tuning of behaviour. As conflicts arise (coupled with the underlying conditions of resource contention and erratic requests), AMs of *DatacentreInt* retune their actions until the system is stable and has reached a steady state. This steady state is achieved from about 320s mark at which point both AMs begin to efficiently coexist with minimal conflict. At this point both AMs (*PeM_Int* and *PoM_Int*) have successfully adjusted their actions to mitigate existing interferences and at about 520s (when conflicts have stopped), AMs stopped adjusting their behaviours.

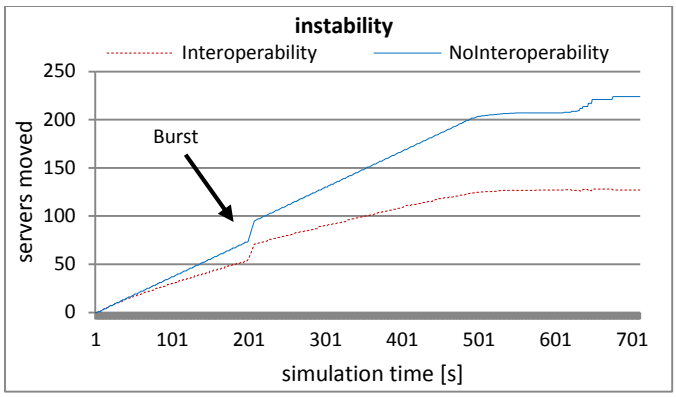

Fig. 3. Instability analysis with burst injected at 200s

Again the instability analysis (Fig. 3) shows that the proposed interoperability solution is still capable of autonomically addressing conflicts between coexisting AMs in complex situations. Despite the complex conditions of the experimental scenario, there is still significant reduction in the rate of server movement in *DatacentreInt*. Also, we can see that the server movement in *DatacentreInt* tails off at about 520s while it continues to significantly fluctuate in *DatacentreNoInt*. The rate of instability increase is a resultant effect of the burst. The movement of servers has some power cost implications as analysed by Fig. 4. Results show that without any form of interoperability control, as in *DatacentreNoInt*, *PeM_NoInt*, under the underlying conditions, almost completely impedes the actions of *PoM_NoInt* rendering its power management effect almost negligible. However, in the case of interoperability control, as in *DatacentreInt*, AMs are able to dynamically adjust their actions so as to gradually reduce or remove conflicts. This is why there is visibly clear difference in the power consumption of both datacentres.

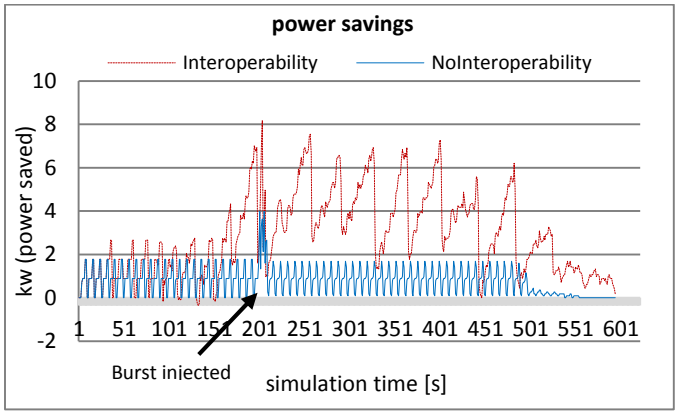

Fig. 4. Power savings analysis

Fig. 4 is a clear view of performances in terms of power optimisation and the impact of the interoperability solution. The sudden jump in power savings at 200s can be attributed to the fact that when the burst was injected, AMs temporarily paused the movement of servers and majority of requests are either queued or dropped, reducing the number of servers being deployed. The prevailing conditions of the scenario (resource contention and irregular request rate) added another twist to the conflicts experienced by the AMs. Under these conditions,

DatacentreNoInt struggled to achieve its goal as it experienced significant drop in performance while *DatacentreInt* was more robust in achieving its general performance goal.

The results above suggest that it is counterproductive to run a multi-manager datacentre without any form of interoperability solution. We have seen that conflicts between coexisting AMs can defeat the AMs' set goals and also lead to spiraling overhead cost. This often leads to unsatisfactory results, especially in complex operating conditions. Results, however, have shown that the proposed interoperability solution is sufficiently sophisticated to efficiently handle conflicts between pairs of coexisting AMs and shows promising signs of yielding satisfactory results under a wide range of operating circumstances (assuming closed-systems). So, we conclude that stigmergic interoperability is a promising approach to calm instability arising from complex interactions in multi-manager datacentres and other similarly complex autonomic systems.

V. CONCLUSION

The success of autonomic computing has inevitably led to situations where multiple autonomic managers (AMs) need to coexist and/or interact directly or indirectly within the same system. In this paper we have provided motivation for interoperability solutions for multi-manager autonomic systems. We have provided example scenarios where such solutions are needed and can be evaluated.

We suggest that support for interoperability should be an integral part of the autonomic system. We have proposed a solution based on stigmergy, using environmental variables and architectural support to facilitate indirect interaction between the feedback loops of separate AMs, operating independently, without explicit knowledge of each other's presence or operation.

A stigmergic interoperability mechanism, which is based on our earlier published TAArch architecture, has been presented. We have shown how our approach to interoperability provides implicit automatic coordination between AMs in a multi-manager scenario without the need for design-time or run-time planning or knowledge of the run-time AM population / mix. The approach provides efficient collaboration (as against competition) between coexisting AMs. The stigmergic interoperability solution builds on the *Stigmergic Phenomenon*. The AMs are designed to sense their environment and dynamically adjust (retune) their behaviour as soon as they notice process conflicts. The experimental analyses of multi-manager datacentre scenarios show that the proposed stigmergic interoperability solution achieves over 42% performance improvement (see instability analysis in Fig. 3) in a complex (conflict prone) coexistence of AMs.

REFERENCES

[1] R. Nou and J. Torres, "Heterogeneous QoS Resource Manager with Prediction," The Fifth International Conference on Autonomic and Autonomous Systems (ICAS), Karlsruhe, Germany, 2009.

[2] V. Ramachandran, M. Gupta, M. Sethi, and S. Chowdhury, "Determining Configuration Parameter Dependencies via Analysis of Configuration Data from Multi-tiered Enterprise Applications," The sixth International Conference on Autonomic Computing (ICAC), Barcelona, Spain, 2009.

[3] C. Kennedy, "Decentralised metacognition in context-aware autonomic systems: some key challenges," The twenty fourth American Institute of Aeronautics and Astronautics (AIAA) Workshop on Metacognition for Robust Social Systems, Atlanta, Georgia, USA, 2010.

[4] M. Salehie and L. Tahvildari, "Autonomic computing: Emerging trends and open problems," The 2005 Workshop on the Design and Evolution of Autonomic Application Software (DEAS), New York, USA, 2005.

[5] R. Quitadamo and F. Zambonelli, "Autonomic communication services: a new challenge for software agents," Journal of Autonomous Agents and Multi-Agent Systems, Springer, 17 (3), pp. 457–475, 2008.

[6] G. Schulz, "The Green and Virtual Data Center," CRC Press, 2009.

[7] T. Eze and R. Anthony, "Trustworthy Autonomic Architecture (TAArch): Implementation and Empirical Investigation," International Journal on Advances in Intelligent Systems (IntSys), IARIA, 7 (1 & 2), pp. 279 – 301, 2014.

[8] M. Dorigo, E. Bonabeau, and G. Theraulaz, "Ant algorithms and stigmergy," Future Generation Computer Systems, 16 (8), pp. 851-871, 2000

[9] P. Stone and M. Veloso, "Multiagent Systems: A Survey from a Machine Learning Perspective," In Autonomous Robots, Springer, 8 (3), pp. 345-383, 2000.

[10] H. Liu and H. Iba, "Multi-agent Learning of Heterogeneous Robots by Evolutionary Subsumption," In Lecture Notes in Computer Science (LNCS), Springer, 2724, pp. 1715-1728, 2003.

[11] R. Brooks, "Robust Layered Control System for a Mobile Robot," IEEE Journal of Robotics and Automation, 2 (1), pp. 14-23, 1986.

[12] G. O'Reilly, and E. Ehlers "Synthesizing Stigmergy for Multi Agent Systems," In Lecture Notes in Computer Science (LNCS), Springer, 4088, pp. 34-45, 2006.

[13] K. Hadeli, P. Valckenaers, M. Kollingbaum, and H. Brussel, "Multi-agent Coordination and Control using Stigmergy," In Lecture Notes in Computer Science (LNCS), Springer, 2977, pp. 105-123, 2004.

[14] J. Kephart, H. Chan, R. Das, and D. Levine, "Coordinating multiple autonomic managers to achieve specified power-performance tradeoffs," In Proceedings of the fourth International Conference on Autonomic Computing (ICAC), Florida, USA, 2007.

[15] R. Anthony, M. Pelc, and H Shauib, "The Interoperability Challenge for Autonomic Computing," The third International Conference on Emerging Network Intelligence (EMERGING), Lisbon, Portugal, 2011.

[16] T. Eze and R. Anthony, "Dead-Zone Logic in Autonomic Systems," IEEE Conference on Evolving and Adaptive Intelligent Systems (EAIS), Linz, Austria, 2014.

[17] J. Perez, C. Germain-Renaud, B. Kegl, and C. Loomis, "Utility-based Reinforcement Learning for Reactive Grids," The fifth International Conference on Autonomic Computing (ICAC), Illinois, USA, 2008.

[18] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif, "On the Use of Fuzzy Modeling in Virtualized Data Center Management," The fourth International Conference on Autonomic Computing (ICAC), Florida, USA, 2007.

[19] J. Berral, R. Gavalda, and J. Torres, ""Living in Barcelona" Li-BCN Workload 2010," Technical Report LiBCN10, Barcelona Supercomputing Centre, Barcelona, Spain, 2010.

[20] M. Pretorius, M. Ghassemian, and C. Ierotheou, "An investigation into energy efficiency of data centre virtualization," International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, Fukuoka, Japan, 2010.

[21] V. Durani, "IBM BladeCenter Systems Up to 30 Percent More Energy Efficient Than Comparable HP Blades," IBM Press Release, Nov. 16, 2006.

[22] X. Wang, C. Lefurgy, and M. Ware, "Managing peak system-level power with feedback control," Research Report RC23835, IBM, 2005.

[23] J. Berral, I. Goiri, R. Nou, F. Julià, J. Guitart, R. Gavaldà, and J. Torres, "Towards energy-aware scheduling in data centers using machine learning," The 1st International Conference on Energy-Efficient Computing and Networking (e-Energy), New York, USA, 2010.