Trustworthy Autonomic Architecture (TAArch): Implementation and Empirical Investigation

Thaddeus Eze and Richard Anthony

Autonomic Computing Research Group
School of Computing & Mathematical Sciences (CMS)
University of Greenwich, London, United Kingdom
{T.O.Eze and R.J.Anthony}@gre.ac.uk

Abstract — **This paper presents a new architecture for trustworthy autonomic systems. This trustworthy autonomic architecture is different from the traditional autonomic computing architecture and includes mechanisms and instrumentation to explicitly support run-time self-validation and trustworthiness. The state of practice does not lend itself robustly enough to support trustworthiness and system dependability. For example, despite validating system's decisions within a logical boundary set for the system, there's the possibility of overall erratic behaviour or inconsistency in the system emerging for example, at a different logical level or on a different time scale. So a more thorough and holistic approach, with a higher level of check, is required to convincingly address the dependability and trustworthy concerns. Validation alone does not always guarantee trustworthiness as each individual decision could be correct (validated) but overall system may not be consistent and thus not dependable. A robust approach requires that validation and trustworthiness are designed in and integral at the architectural level, and not treated as add-ons as they cannot be reliably retro-fitted to systems. This paper analyses the current state of practice in autonomic architecture, presents a different architectural approach for trustworthy autonomic systems, and uses a datacentre scenario as the basis for empirical analysis of behaviour and performance. Results show that the proposed trustworthy autonomic architecture has significant performance improvement over existing architectures and can be relied upon to operate (or manage) almost all level of datacentre scale and complexity.**

Keywords - trustworthy architecture; trustability; validation; datacentre; autonomic system; dependability; stability; autonomic architecture

I. INTRODUCTION

A robust autonomic architecture is a vital key to achieving dependable (or trustworthy) autonomic systems. We have made initial progress [1] in this direction to address the issue of autonomic trustworthiness through adequate run-time conformance testing as integral part of a trustworthy autonomic architecture (different from the traditional autonomic architecture). This work is an extension of the initial progress and the implementation (with empirical analysis) of the new trustworthy architecture. The traditional autonomic architecture as originally presented in the autonomic computing blueprint [2] has been widely accepted and deployed across an ever-widening spectrum of autonomic system (AS) design and implementations. Research results in the autonomic research community are based, predominantly, on the architecture's basic MAPE (monitor-analyse-plan-execute) control loop, e.g., [3][4]. Several implementation variations of this control loop, for example [5][6], have been promoted. While [5] breaks the MAPE components into two main groups with the Monitor/Analyze group handling reactive tasks and the Plan/Execute group responsible for proactive adaptation, [6] adopts a slightly different approach. In [6], the MAPE architecture is divided into global and local sub-architectures, which implement Analyze/Planning and Monitor/Execute components, respectively. Alternative approaches, e.g., the intelligent machine design (IMD) based approach [7] have also been proposed. However, research [8] shows that most approaches are MAPE [9] based. Despite progress made, the traditional autonomic architecture and its variations is not sophisticated enough to produce trustworthy ASs. A new approach with inbuilt mechanisms and instrumentation to support trustworthiness is required.

At the core of system trustworthiness is validation and this has to satisfy run-time requirements. In large systems with very wide behavioural space and many dimensions of freedom, it is close to impossible to comprehensively predict possible outcomes at design time. So it becomes highly complex to make sure or determine whether the autonomic manager's (AM's) decision(s) are in the overall interest and good of the system. There is a vital need, then, to dynamically validate the run-time decisions of the AM to avoid the system '*shooting itself in the foot*' through control brevity, i.e., either too loose or too tight control leading to unresponsive or unstable system respectively. The traditional autonomic architecture does not explicitly and integrally support run-time self-validation; a common practice is to treat validation and other needed capabilities as add-ons. Identifying such challenges, the traditional architecture has been extended (e.g., in [10]) to accommodate validation by integrating a *self-test* activity into the autonomic architecture. But the question is whether validation alone can guarantee trustworthiness.

The need for trustworthiness in the face of the peculiar nature of ASs, (e.g., context dynamism) comes with unique and complex challenges validation alone cannot sufficiently address. Take for instance; if a manager (AM) erratically changes its decision, it ends up introducing noise to the system rather than smoothly steering the system. In that instance, a typical validation check will *pass* each correct decision (following a particular logic or rule) but this could lead to oscillation in the system resulting in instability and inconsistent output, which could emerge at a different logical level or time scale. A typical example could be an AM that follows a set of rules to decide when to move a server to or

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

from a pool of servers; as long as the conditions of the rules are met, the AM will move servers around not minding the frequency of changes in the conditions. An erratic change of decision (high rate of moving servers around) will cause undesirable oscillations that ultimately detriment the system. What is required is a kind of intelligence that enables the manager to smartly carry out a change only when it is safe and efficient to do so – within a particular (defined) safety margin. A higher level of self-monitoring to achieve, for example, stability over longer time frames, is absent in the MAPE-orientated architectures. This is why autonomic systems need a different approach. The ultimate goal of the new approach is not just to achieve self-management but also to achieve consistency and reliability of results through self-management. These are the core values of the proposed architecture in this paper.

We look at the background of work towards AS trustworthy architecture in Section II. We present a new trustworthy autonomic architecture in Section III and present a datacentre-based implementation and empirical analysis of the new architecture in Section IV. Section V concludes the work.

II. BACKGROUND

The idea espoused in this work is that trustworthiness (and any other desired autonomic capability) should be conceived at design stage. This means that the autonomic architecture should be flexible (and yet robust) enough to provide instrumentations that allow designers to specify processes to achieve desired goals. It then follows that we need to rethink the autonomic architecture. In this section, we look at the current state of practice and efforts directed towards AS trustworthiness. We analyse few proposed trustworthy architectures and some isolated bits of work that could contribute to trustworthy autonomic computing. Trustworthiness requires a holistic approach, i.e., a long-term focus as against the near-term needs that merely address methods for building trust into existing systems. This means that trustworthiness needs to be designed into systems as integral properties.

Chan *et al.* [11] asks the critical question of "How can we trust an autonomic system to make the best decision?" and proposes a '*trust*' architecture to win the trust of autonomic system users. The proposal is to introduce trust into the system by assigning an "instantaneous trust index" (ITI) to each execution of a system's AM –where ITI could be computed, for example, by examining what fraction of AM suggested actions the user accepts unchanged, or by examining how extensive the changes that the user makes to the suggested actions are. The overall trust index, which reflects the system administration's level of trust in the AM, is computed as the function f(ITIi) where $i = 1, 2, 3, \ldots$ and ITIi are the individual ITIs for each AM execution. This is similar to the proposal in this work in the sense that it considers trust as architecture-based and also defines trust in the language of the user. However, this method will be overly complex (and may be out of control) in large systems with multiple AMs if the user is required to moderate every single AM suggested action. In such systems some of the AM's decisions are not transparent to the human user. Another effort that supports the idea that dependability should be conceived at design time and not retro-fitted to systems is the work in [12]. Hall and Rapanotti [12] propose an *Assurance-Driven Design* and posit that engineering design should include the detailing of a design for a solution that guarantees satisfaction of set requirements and the construction of arguments to assure users that the solution will provide the needed functionality and qualities. The key point here is that trustworthiness is all about securing the confidence of the user (that the system will do what it says) and the way to achieve this is by getting the design (architecture) right. This is the thrust of this work.

Shuaib *et al.* [7] propose a framework that will allow for proper certification of A-C systems. Central to this framework is an alternative autonomic architecture based on Intelligent Machine Design (IMD), which draws from the human autonomic nervous system.

Kikuchi *et al.* [13] proposes a policy verification and validation framework that is based on model checking to verify the validity of administrator's specified policies in a policy-based system. Because a known performing policy may lead to erroneous behaviour if the system (in any aspect) is changed slightly, the framework is based on checking the consistency of the policy and the system's defined model or characteristics. This is another important aspect of the proposed solution in this work –validation is done with reference to the system's defined goal.

A trustworthy autonomic grid computing architecture is presented in [14]. This is to be enabled through a proposed fifth self-* functionality, self-regulating: Self-regulating capability is able to derive policies from high-level policies and requirements at run-time to regulate self-managing behaviours. One concern here is that proposing a fifth autonomic functionality to regulate the self-CHOP functionalities as a solution to AS trustworthiness assumes that trustworthiness can be achieved when all four functionalities perform 'optimally'. This assumption is not entirely correct. The self-CHOP functionalities alone do not ensure trustworthiness in ASs. Take for example; the self-CHOP functionalities do not address *validation*, which is a key factor in AS trustworthiness. The self-CHOP (or sometimes referred to as self-*) stands for self-Configuring, self-Healing, self-Optimising, and self-Protecting. These are the characteristics or functional areas that define the capabilities of autonomic systems and will be referred to as autonomic functionalities in this paper.

Another idea is that trustworthiness is achieved when a system is able to provide accounts of its behaviour to the extent that the user can understand and trust. But these accounts must, amongst other things, satisfy three requirements: provide a representation of the policy guiding the accounting, some mechanism for validation and accounting for system's behaviour in response to user demands [15]. The system's actions are transparent to the user

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

and also allow the user (if required) the privilege of authorising or not authorising a particular process. This is a positive step (at least it provides the user a level of confidence and trust) but also important is a mechanism that ensures that any ‘authorised’ process does not lead to unreliable or misleading results. This is one aspect not considered by many research efforts. There are possibilities of erratic behaviour (which is not healthy to the system) despite the AM’s decisions being approved. One powerful way of addressing this challenge is by implementing a *dead-zone* (DZ) logic originally presented in [16]. A DZ, which is a simple mechanism to prevent unnecessary, inefficient and ineffective control brevity when the system is sufficiently close to its target value, is implemented in [16] using Tolerance-Range-Check (TRC) object. The TRC object encapsulates DZ logic and a three-way decision fork that flags which action (left, null or right) to take depending on the rules specified. The size of the DZ can be dynamically adjusted to suit changes in environmental volatility. A key use of this technique is to reduce oscillation and ensure stability in the face of high rate of adaptability despite process correctness. A mechanism to automatically monitor the stability of an autonomic component, in terms of the rate the component changes its decision (for example when close to a threshold tipping point), was presented in [17]. The *DecisionChangeInterval* property is implemented in the AGILE policy language [17] on decision making objects such as rules and utility functions. This allows the system to monitor itself and take action if it detects instability at a higher level than the actual decision making activity. This technique is used in the proposed solution herein.

Heo and Abdelzaher [18] present ‘AdaptGuard’, a software designed to guard adaptive systems from instability resulting from system disruptions. The software is able to infer and detect instability and then intervenes (to restore the system) without actually understanding the root cause of the problem *–root-cause-agnostic* recovery. Instability is another aspect addressed in the solution proposed in our work. Because AM control brevity could lead to instability despite process correctness, it is important to also consider this scenario. Hawthorne *et al.* [19] demonstrates Teleo-Reactive (T-R) programming approach to autonomic software systems and shows how T-R technique can be used to detect validation issues at design time and thus reducing the cost of validation issues.

Validation is central to achieving trustworthy autonomics and this has to meet run-time requirements. A generic self-test approach is presented in [10]. The authours of [10] extended the MAPE control loop to include a new function called *Test* (Figure 1). By this they define a new control loop comprising Monitor, Analyse, Decision, Test and Execute –MADTE activities. The MADTE loop works like the MAPE loop only that the Decision activity calls the Test activity to validate a chosen action should it determine to adapt a suggested behaviour. The Test activity carries out a test on the action and returns its result to the Decision activity, which then decides whether to implement, skip or choose another action. (An adaptation is favoured if *Test* indicates that it will lead to component’s better performance in terms of characteristics such as optimisation, robustness or security.) The process is repeated if the latter is the case. When an action is decided on, the decision activity passes it to the Execute activity for implementation. This is vital to run-time self-validation and is consistent with our proposed solution in this work in terms of designing validation into the system’s architecture. A feedback-based validation, which relies on a kind of secondary (mostly external) expertise feedback to validate the output of a system is presented in [20]. This is reactionary and has no contribution to the result of the system in the first place. Though this may suffice for some specific system’s needs, what is generally required for AS validation is run-time validation of decisions (or processes) that lead to system outputs.

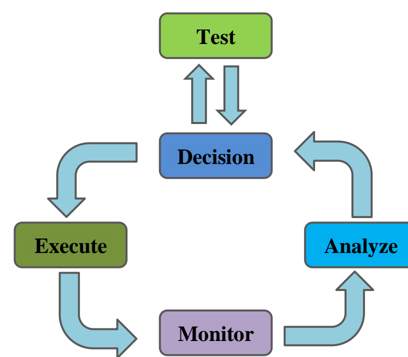


Figure 1: Control loop with a test function [10]

It should be noted that AS trustworthiness goes beyond secure computing. It is result orientated; not focusing on how a goal is achieved but the dependability of the output achieved. All systems, no matter how simple or complex, are designed to meet a need, but not all systems have security concerns. So trustworthiness is not all about security and validation. On the other hand, it is not about showing that a system or process works but also making sure that it does exactly what it is meant to do. This aspect is addressed in the proposed trustworthy autonomic architecture by a component that carries out a longer term assessment of the system’s actions. These are the evolving challenges and where work must be concentrated if we are to achieve certifiable autonomic systems.

A. *Autonomic architecture life-cycle*

We argue that trustworthiness cannot be reliably retrofitted into systems but must be designed into system architectures. We track autonomic architecture (leading to trustworthiness) pictorially in a number of progressive stages addressing it in an increasing level of detail and sophistication. Figure 2 provides a key to the symbols used.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

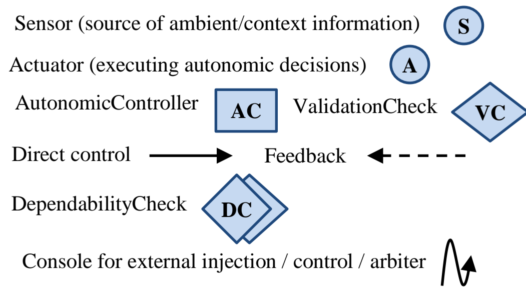


Figure 2: Pictographic key used for the architecture life-cycle.

Figure 3 illustrates the progression, in sophistication, of autonomic architectures and how close they have come to achieving trustworthiness. Although this may not be exhaustive as several variations and hybrids of the combinations may exist, it represents a series of discrete progressions in current approaches.

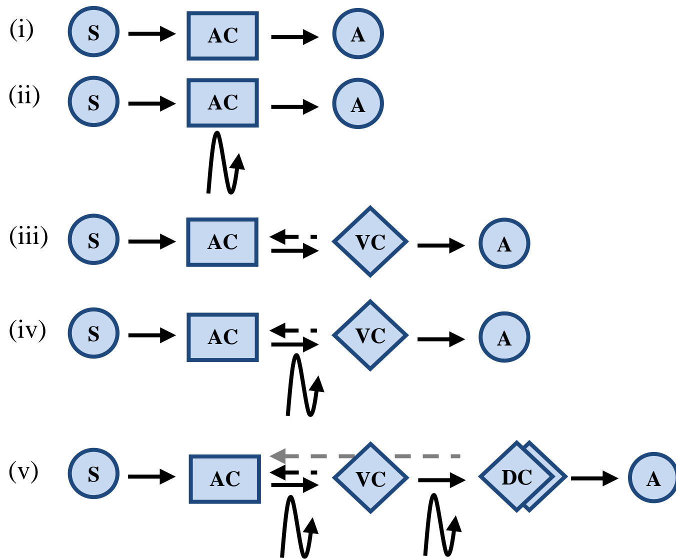


Figure 3: Pictorial representation of autonomic architecture life-cycles.

Two distinct levels of sophistication are identified: The first level represents the traditional autonomic architecture (Figure 3 (i) and (ii)) basically concerned with direct self-management of controlled/monitored system following some basic sense-manage-actuate logic defined in AC. For the prevailing context, AC is just a container of autonomic control logic, which could be based on MAPE or any other autonomic control logic. The original autonomic architecture proposed with the introduction of autonomic computing [2] falls within this level. This achieves basic self-management capability and has since been adapted by several researchers to offer more smartness and sophistication. To add a degree of trust and safeguard, an external interface for user control input is introduced in (ii). This chronicles such approaches that provide a console for external administrative interactions (e.g., real-time monitoring, tweaking, feedback, knowledgebase source, trust input, etc.) with the autonomic process. An example of level (ii) is work in [15], where the system's actions are transparent to the user and the user can moderate the behaviour of the system by allowing or disallowing system decided actions. The system has a console that offers the user the privilege of authorising or not authorising a particular process. Another example in this category is unmanned vehicles (UVs). In UVs there are provisions for activating auto piloting and manual piloting. The user can decide when to activate either or run a hybrid.

The second level (Figure 3 (iii) and (iv)) represents efforts towards addressing run-time validation. Instrumentations to enable systems check the conformity of management decisions are added. This includes such approaches that are capable of run-time self-validation of autonomic management decisions. The validation check is done by the VC component and the check results in either a *pass* (in which case the validated decision is actuated) or a *fail*. Where the check fails VC sends feedback to AC with notification of failure (e.g., policy violation) and new decision is generated. An additional layer of sophistication is introduced in Figure 3 (iv) with external touch-point for higher level of manageability control. This can be in the form of an outer control loop monitoring over a longer time frame an inner (shorter time frame) control loop. The work in [10] (explained in Section II), which is an extension of MAPE control to include a '*Test*' activity corresponds to level (iii) of Figure 3. The *Test* activity tests every suggested action (decision) by the plan activity. If the test fails the action is dropped and a new one is decided again. The work in [21] corresponds to level (iv) of Figure 3. The work in [10] is extended in [21] to include auxiliary test services components that facilitate manual test management and a detailed description of interactions between test managers and other components. Here test managers implement closed control loops on autonomic managers (such as autonomic managers implement on managed resources) to validate change requests generated by the autonomic managers.

At the level of current sophistication (state-of-the-art), there are techniques to provide run-time validation check (for behavioural and structural conformity), additional console for higher level (external) control, etc. Emerging and needed capabilities include techniques for managing oscillatory behaviour in autonomic systems. These are mainly implemented in isolation. What is required is a holistic framework that collates all these capabilities into a single autonomic unit. Policy autonomics is one of the most used autonomic solutions. Autonomic managers (AMs) follow rules to decide on actions. As long as policies are validated against set rules the AM adapts its behaviour accordingly. This may mean changing between states. And when the change becomes rapid (despite meeting validation requirements) it is capable of introducing oscillation, vibration and erratic behaviour (all in form of noise) into the system. This is more noticeable in highly sensitive systems. So a trustworthy autonomic architecture needs to provide a way of addressing these issues. Level (v) of Figure 3 falls within the next level of sophistication required to address the

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

identified issues and ensure dependability. This is at the core of the proposed solution presented in next the Section.

III. TRUSTWORTHY AUTONOMIC ARCHITECTURE

This section presents the new trustworthy autonomic architecture (TAArch). First, a general view of the architecture is presented and then followed by detailed explanation of its components. Figure 4 explains a trustworthy autonomic framework with three major components that embody self-management, self-validation and dependability. The architecture builds on the traditional autonomic architecture (denoted as the *AutonomicController* (AC) component). Other components include *ValidationCheck* (VC –which is integrated with the decision-making object of the controller to validate all *AutonomicController* decisions) and *DependabilityCheck* (DC) component, which guarantees stability and reliability after validation. The DC component works at a different time scale, thus overseas the finer-grained sequence of decisions made by the AC and VC.

The AC component (based on, e.g., MAPE logic, IMD framework, etc.) monitors the managed sub-system for context information and takes decision for action based on this information. The decided action is validated against the system's goal (described as policies) by the VC component before execution. If validation fails, (e.g., policy violation) it reports back to the AC otherwise the DC is called to ensure that outcome does not lead to, for example, instability in the system.

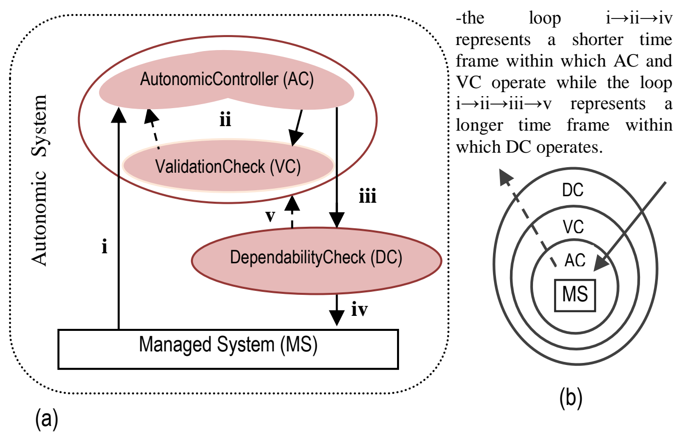


Figure 4: Trustworthy autonomic architecture

The *DependabilityCheck* component comprises of other sub-components, which makes it possible to be adapted to address different challenges. This feature makes the architecture generic and suitable to address even evolving autonomic capability requirements. For instance, in [22], the architecture is adapted to address interoperability challenges in complex interactions between AMs in multi-manager scenarios. *Predictive* component is one example of the *DependabilityCheck* sub-components that allows it to predict the outcome of the system based on the validated decision. The *DependabilityCheck* either prevents execution and sends feedback in form of some calibration parameters to the *AutonomicController* or calls the actuator to execute the validated decision.

A. Overview of the TAArch architecture components

This section presents the TAArch architecture in a number of progressive stages addressing it in an increasing level of detail. First, the self-management process is defined as a *Sense–Manage–Actuate* loop where *Sense* and *Actuate* define *Touchpoints* (the autonomic manager's interface with a managed system) and *Manage* is the embodiment of the actual autonomic self-management. Figure 5 is a detailed representation of the architectural framework.

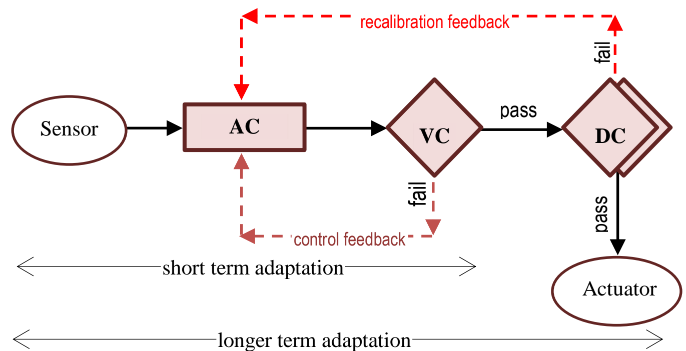


Figure 5: Detailed structure of the TAArch framework.

Traditionally, the *AutonomicController* (AC) senses context information, decides (following some rules) on what action to take and then executes the action. This is the basic routine of any AM and is at the core of most of the autonomic architectures in use today (Figure 3). At this level the autonomic unit matters but the content of the unit does not matter much, i.e., it does not matter what autonomic control logic (e.g., MAPE, IMD, etc.) that is employed so long as it provides the desired autonomic functionalities. This means that the AC component can be configured according to any autonomic control logic of choice making the framework generic as it is not tied to any one control logic. Basically, the AC component introduces some smartness into the system by intelligently controlling the decision-making of the system. Once an action is decided, following detailed analysis of context information, the decision is passed on for execution. This is at the level of sophistication defined by the autonomic architecture life-cycle level 1 (Figure 3 (i) and (ii)). So, the AC component of the TAArch framework provides designers the platform to express rules that govern target goal and policies that drive decisions on context information for system adaptation to achieve the target goal.

But, the nature of ASs raises one significant concern; input variables (context info) are dynamic and (most times) not predictable. Although rules and policies are carefully and robustly constructed, sensors (data sources) sometimes do inject rogue variables that are capable of thwarting process

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

and policy deliberations. In addition, the operating environment itself can have varying volatility –causing a controller to become unstable in some circumstances. Thus, a mechanism is needed to mitigate behavioural (e.g., contradiction between two policies, goal distortion, etc.) and structural (e.g., illegal structure not conforming to requirement, division by zero, etc.) anomalies. This is where the *ValidationCheck* (VC) component comes in. It should be noted that AC will always decide on action(s) no matter what the input variable is. Once the AC reaches a decision, it passes control to the VC, which then validates the decision and passes it on for execution. If the check fails, VC sends *control feedback* (CF) to AC while retaining previous passed decision. A control feedback is more of an inhibition command that controls what actions are and are not allowed by the manager. This can be configured according to deployment requirements. In a nutshell, the VC, while focusing on the goal of the system, deploys self-validation mechanisms to continuously perform self-validation of the manager's behaviour and configuration against its behavioural goals and also reflects on the quality of the manager's adaptation behaviour. Again, the nature and level of test is entirely user-defined. So, the VC is a higher level mechanism that oversees the AM to keep the system's goal on track. The ultimate concern here is to maintain system goal adhering to defined rules, i.e., adding a level of trust by ensuring that target goal is reached only within the boundaries of specified rules. It is then left for designers to define what constitute validation '*pass*' and validation '*fail*'. Actual component logic are application specific but some examples in literature include fuzzy logic [24], reinforcement learning [23], etc. This is at the level of sophistication defined by the autonomic architecture life-cycle level 2 (Figure 3 (iii) and (iv)).

But in real life we understand that despite the AM taking legitimate decisions within the boundaries of specified rules, it is still possible to have overall system behavioural inconsistencies. That is, a situation where each individual decision could be correct (by logic) and yet the overall behaviour is wrong. This kind of situation where the manager erratically (though legally) changes its mind, thereby injecting oscillation into the system, could be a major concern especially in large scale and sensitive systems. This is beyond the level of current consideration in the state of practice (Figure 3). Therefore, it is necessary to find a way of enabling the AM to avoid unnecessary and inefficient change of decisions that could lead to oscillation. This task is handled by the DC component. It allows the manager change its decision (i.e., adapt) only when it is necessary and safe to do so. Consider a simple example of a room temperature controller in which, it is necessary to track a dynamic goal –a target room temperature. The AM is configured to maintain the target temperature by automatically switching heating ON or OFF according to the logic in (1). A VC would allow any decision or action that complies with this basic logic.

$$\begin{array}{l} IF\ RoomTemp\ <\ TargetTemp\ THEN\ ON_Heating \\ IF\ RoomTemp\ >\ TargetTemp\ THEN\ OFF_Heating \end{array} \quad (1)$$

With the lag in adjusting the temperature the system may decide to switch ON or OFF heating at every slight tick of the gauge below or above target (when room temperature is sufficiently close to the target temperature). This may in turn cause oscillation, which can lead to undesirable effects. The effects are more pronounced in more sensitive and critical systems where such changes come at some cost. For example, a datacentre management system that erratically switches servers between pools at every slight fluctuation in demand load is cost ineffective. Actual component and sub-component logic are user-defined. One powerful logic example, as explained in Section II, for implementing the DC component is the dead-zone (DZ) logic [16]. DZ logic has been shown to offer a reliable means of achieving self-stabilisation, dependable systems and TAC.

The DC component may also implement other sub-components like Prediction, Learning, etc. This enables it to predict the outcome of the system and to decide whether it is safe to allow a particular decision or not. An example sub-component logic is Trend Analysis (TA) logic. TA logic identifies patterns within streams of information supplied directly from different sources (e.g., sensors). By identifying trends and patterns within a particular information, (e.g., spikes in signal strength, fluctuation in stock price, rising/falling trends etc.) the logic enables the AM to make more-informed control decisions and this has the potential of reducing the number of control adjustments and can improve overall efficiency and stability. Also, the analysis of recent trends enables a more accurate prediction of the future. With TA, managers can base decisions on a more-complete view of system behaviour. The usage and importance of TA are discussed in more detail in [16].

So after validation phase, the DC is called to check (based on specified rules) for dependability. DC avoids unnecessary and inefficient control inputs to maintain stability. If the check passes, control is passed to the Actuator otherwise a *recalibration feedback* (RF) is sent to AC. An example of RF is dynamically adjusting (or retuning) the DZ boundary width (explained later) as appropriate. The RF enables the manager to adjust its behaviour to maintain the level of required trust. So, while VC looks at the immediate actions, DC takes a longer term view of the manager's behaviour over a certain defined time interval. A particular aspect of concern, though, is that for dynamic systems the boundary definition of DZ may itself be context dependent (e.g., in some circumstances it may be appropriate to allow some level of changes, which under different circumstances may be considered destabilising). This concern is taken into consideration when defining such boundaries.

So the current state-of-the-art of autonomic architecture suffices for short term adaptation. To handle longer term frame adaptation, e.g., cases where continuous validation fails to guarantee stability and reliability, requires a robust autonomic approach. This robust autonomic approach is what the proposed TAArch offers. Consider the whole architecture as a nested control loop (Figure 4 (b)) with AC the core control loop while VC and DC are intermediate and outer

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

control loops, respectively. In summary, a system, no matter the context of deployment, is truly trustworthy when its actions are continuously validated (i.e., at run time) to satisfy set requirements (system goal) and results produced are dependable and not misleading.

IV. IMPLEMENTATION AND EMPIRICAL ANALYSIS

To demonstrate the feasibility and practicability of the new architecture, this section presents an implementation and simulation analysis of the TAArch architecture using a datacentre case example scenario. This analysis is a complex and robust implementation of TAArch demonstrated in a resource allocation scenario, which models basic datacentre resource allocation management. Although the demonstration uses a datacentre scenario, which though offers a way of efficiently managing complex datacentres, the application of TAArch can be widespread. In other words, although a datacentre is used to demonstrate the functionalities of the proposed architecture, it is not limited to this scenario. The datacentre model represents a very simple datacentre scenario where the simulation focuses on the efficiency and dependability of resource request and allocation management rather than other vast areas of datacentre, e.g., security, power, and cooling etc. So the purpose of the experiments is to demonstrate the applicability and performance of the proposed architecture and not to investigate datacentres themselves. However, the datacentre is chosen as implementation scenario because its many dimensions of complexity and large number of tuning parameters offer a rich domain in which to evaluate a wide range of techniques, tools and frameworks.

In this example, detailed experiments are designed to analyse three different systems based on three different autonomic architectures. The first system, comprising of only AC component, is based on the traditional architecture represented by level 1 (Figure 3 (i) and (ii)) of the autonomic architecture life-cycle. This system will be referred to as sysAC. The second system, comprising of both the AC and VC components, is based on the current level of practice represented by Figure 3 (iii) and (iv). This system will be referred to as sysVC. The third and TAArch-based system, referred to as sysDC, comprises of all three (AC, VC, and DC) components. This system falls within the representation of level (v) of Figure 3. The purpose of this implementation is to illustrate how powerful and robust the TAArch framework is when compared to existing frameworks.

A. Scheduling and Resource Allocation

Several research, e.g., [25][26][27], have proposed scheduling algorithms that optimise the performance of datacentres. In a utility function based approach, Das *et al.* [25] are able to quantify and manage trade-offs between competing goals such as performance and energy consumption. Their approach reduced datacentre power consumption by up to 14%. Other works that have resulted in improved performance and resource utilisation by proposing new scheduling algorithms include [26], which focuses on the allocation of virtual machines in datacentre nodes and [27], which uses a '*greedy resource allocation algorithm*' that allows distributing a web workload among different servers assigned to each service. Our work, on the other hand, does not propose any new scheduling algorithm for efficient utilisation of datacentre resources; however, it uses basic resource allocation technique to model the performance of datacentre autonomic managers in terms of the effectiveness of resource request and allocation management.

Let us consider the model of the datacenter used in this experimentation in detail, (in terms of scheduling and request services). The datacentre model 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 $\underset{..}{U}$ (a combination of applications and their provisioning servers), and an autonomic manager (performance manager *PeM*) that optimises the entire system. 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 (*PeM*) for provisioning (or scheduling) available services according to request. As service requests arrive, *PeM* dynamically populates $\underset{..}{U}$ to service the requests. $\underset{..}{U}$ is defined by equation (2):

$$\underset{..}{U} = \begin{cases} A_1\text{: } (S_{11}, S_{12}, S_{13}, \ldots, S_{1i}) \\ A_2\text{: } (S_{21}, S_{22}, S_{23}, \ldots, S_{2i}) \\ \ldots \ \ldots \ \ldots \ \ldots \ \ldots \ \ldots \\ A_n\text{: } (S_{n1}, S_{n2}, S_{n3}, \ldots, S_{ni}) \end{cases} \tag{2}$$

Where A_i: $(S_i \ldots S_n)$ means that ($S_i \ldots S_n$) servers are currently allocated to Application A_i and n is the number of application entries into $\underset{..}{U}$. (2) indicates that a server can be (re)deployed for different applications. All the servers i in S_i are up and running (constantly available –or so desired by *PeM*) waiting for (re)deployment. The primary performance goal of *PeM* is to minimise oscillation and maximise stability (including just-in-time service delivery to meet service level achievement target) while the secondary performance goal is to maximise throughput.

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 manager checks for resource availability and deploys server(s) according to the size of the request. The size of application requests and the capacity of servers are defined in million instructions per second (MIPS). In this report '*size*' and '*capacity*' are used interchangeably and mostly would refer to 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 (2) 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

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

constantly populated as requests arrive. Now as a result of the lag between provisioning time and the rate of request arrival or as a result of some unforeseen process disruptions, some servers do overshoot their provisioning time and thereby left redundant in the queue. This can be addressed by the manager, depending on configuration, to reduce the impact on the whole system. 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. Rejected or dropped requests are not considered in calculating SLA. The essence of the request queue is to allow the manager to accept requests only when it has enough resources to service them. Service contract is entered only when requests are accepted. So the manager could look at its capacity (in terms of available resources), compare that with the capacity requested and say '*sorry I haven't got enough resources*' and reject or drop the request. This whole process goes on and the manager manages the system to the level of its sophistication. This process is explained in Appendix A.

A basic system without any form of smartness can barely go far before the whole system is clogged due to inefficient and unstructured resource management. The level to which any autonomic manager can successfully and efficiently manage the process defined above depends on its level of sophistication. For us this largely depends on how each manager is wired (in terms of architecture) and not necessarily the scheduling algorithm or actual component logic used. For example, two managers, differently wired, may employ the same scheduling algorithm but achieve different results. Results here may be looked at in terms of, say, '*with such level of available resources how many requests were successfully serviced*'. These are the kind of considerations in the following experiments where three differently wired autonomic managers are analysed.

B. Experimental Design, Workload and Parameters

The experiments are designed and implemented using the TAArch application (Appendix A). This application is developed using the C# programming language. The scope of the experiments focuses on the performance of datacentre autonomic managers in resource request and allocation management activities under varying workloads. Although some workload parameters are sourced from experimental results of other research, e.g., [28][29][30], 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 is being investigated the user can design individual scenarios and set workloads according to specific requirements.

The result of every simulation analysis is relative to the set of workload or parameter set used, which configure the specific application instance. The parameter set used for the datacentre model analysis here are classified into *internal* and *external* variables. Internal variables are those variables that do not change during run-time, e.g., the capacity of a server. External variables, on the other hand, are those that can change in the cause of the simulation, e.g., the rate at which requests arrive. External variables are usually system generated and are always unpredictable. The experimental design has the capacity for heterogeneous workload representation. That means that even the internal variables can be reset before simulation begins thereby offering the possibility of scaling to high/low load to suit user preferences (see Appendix A). The range of value options for most of the variables reflects the experimental results of other research especially [28][29][30]. Note that the following variables are used with the C# application that has been designed to simulate the datacentre model and run the stated experiments.

- **Internal Variables**

Below is the list of internal variables used in this experiment. Some of the variables used are specific to this experiment while some are general datacentre variables.

- ***server.sCapacity***:

This represents the service capacity of each server and for the purposes of the experiments here all servers are assumed to be of equal capacity. Server capacity (size) is measured in MIPS.

- ***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. The margin 0.1 – 0.3 (representing 10 to 30%) is used because experiments show that it is the safest margin within which accurate results can be guaranteed. The datacentre is not completely settled below 10% and beyond 30% scenarios with low number of servers will yield inaccurate results. The higher the value of *RetrieveRequestParam* the earlier the start of request completion.

- ***RetrieveRate***:

Indicates rate at which requests are completed once simulation for service request completion is initiated. Value is relative to rate of request arrival – e.g., if value is 5, then it means service request completion is five times slower than rate of service request.

- ***BurstSize***:

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 managers react to it differently. Often times injecting a burst disorientates the system. The nature of this disruption is usually in the form of sudden burst 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

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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.

- ***ServerOnTime***:

Indicates how long it takes a server to power on. This is relative to the rate of request arrival -it is *ServerProvisioningTime* + 1.

- ***RequestRateParam***:

A constant used to adjust the possible range of request rate. The user of the TAArch Application (Appendix A) can set request rate according to preference but this preference may not be accommodated within the available rate range. For example, 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.

- **External Variables**

Below is the list of external variables used in this experiment. Recall that external variables, also known as dynamic variables, are those variables that are fed into the system during run-time either as system generated (dynamic sensitivity to contextual changes) or human input (through external touch-points). Some of the variables used are specific to this experiment while some are general datacentre variables.

- ***DZConst***:

DZConst is the tuning parameter the manager uses to dynamically adjust dead-zone boundaries. The dead-zone boundary is also known as DZWidth. Because this variable has significant effect on the system, it is suggested that the initial value be set at 1.5. The manager usually adjusts this value dynamically and there is also a provision to manually adjust the value during run time.

- ***AppSize***:

The application size variable represents the size or capacity of a service request (request for an application). In the experiments that follow, except otherwise changed, all applications are initially assumed to be of the same size. There are touch-points to dynamically change this value. The application size variable is measured in MIPS.

- ***RequestRate***:

This variable also referred to as rate of service request or rate of request arrival is the measure of the frequency of service request. This is in terms of the number of requests recorded per unit of time. In real systems, this can be calculated as an average for all services (applications) or for individual services. In [28], for example, *RequestRate* values are calculated for each service and are presented in *requests/day*. The experiments of this work take an average of *RequestRate* for all services and represent values as *requests/second*.

- ***BurstInterval***:

The burst interval variable defines the interval at which bursts are injected into the system during the simulation. This is specific to the experimental application and is dependent on what the user wants to investigate. Usually bursts are introduced once at a specific time or several at random times.

The experimental workload is flexible in that all variables can be scaled to suit user's workload (high or low) requirements. Every experiment has a detailed workload outline used as shown in the following experiments.

C. Manager Logic

Manager logic details the individual control logic employed by each of the managers in order to achieve the performance goal. This explains the logical composition of each manager. The three autonomic managers track the life-cycle of autonomic architecture as presented in Figure 3. *sysAC* represents the *AutonomicControler* level based manager while *sysVC* represents the *ValidationCheck* level based manager. *sysDC* represents the *DependabilityCheck* level based manager and this conforms to TAArch architecture.

The primary goal of the AM (also referred to as the performance manager –*PeM*), represented by each of *sysAC*, *sysVC*, and *sysDC*, is to ensure that the system remains stable under almost all perceivable operating and contextual circumstances and is capable of achieving desired and dependable results within such circumstances (i.e., over the expected range of contexts and environmental conditions and beyond). The secondary goal is to maximise throughput.

- ***sysAC***

This manager implements the basic autonomic control logic. Structurally based on Figure 3 (ii), the manager receives requests and allocates resources accordingly. The basic allocation logic here is to deploy a server whenever capacity offset (i.e., excess capacity of running servers –these are used to service new requests) is less than the current capacity of a single request. This is known as the *DecisionBoundary*. This is depicted, for example, as:

```
if (app1ACOffset < app1.appCapacity)
 {       <...deploy server...>   }
```

Where

```
app1ACOffset = app1ACAvailableCapacity -
         app1ACRunningCapacity;
```

sysAC has no additional intelligence. For example, decisions are not validated and the manager does not consider the rate at which system behaviour crosses the *DecisionBoundary*. As long as boundary conditions are met, the manager executes appropriate decisions.

- ***sysVC***

This manager shows a higher level of intelligence than *sysAC*. One aspect of validation here is to check the performance of the manager in terms of correctness. The

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

manager does not start a job that cannot be completed –i.e., at every *DecisionBoundary* the manager checks to make sure that it has enough resources to service a request. Where this is not the case (meaning the check has *failed*), the manager rejects the request and updates itself. The manager has a limit to which it can allow capacity deficit expressed as:

```
else if (app1VCOffset <= (0 - app1.appCapacity))
     {
         DroppedRequestCountVC += 1;
     }
```

So, in addition to the basic control and resource allocation logic of sysAC, sysVC carries out a validation of every allocation decision. Validation here is in terms of behavioural (e.g., starting a job only when there are enough capacity to complete it) and structural (e.g., avoiding initiating provisioning when server pool is empty i.e., `listViewServer.Items.Count = 0`) correctness.

sysVC is within the representation of current stages of autonomic architecture life-cycle presented in Section II as Figure 3 (iii) and (iv). Beyond the level of validation, *sysVC* exhibits no further intelligence.

- ***sysDC***

sysDC performs all the activities of the *sysAC* and *sysVC* managers with additional intelligence. The manager looks at the balance of cost over longer term and retunes its configuration to ensure a balanced performance. For example, the manager implements dead-zone (DZ) logic on decision boundaries. Firstly, the dead-zone boundaries (upper and lower bounds), for example, are calculated as:

```
App1.DZUpperBound = (app1.appCapacity +
(app1.appCapacity * DZ.DZConst));                    (3)

App1.DZLowerBound = (app1.appCapacity -
(app1.appCapacity * DZ.DZConst));
```

Note: the size of DZ boundary depends on the nature of the system and data being processed. For example, in fine-grained data instance, where small shifts from the target can easily tip decisions –sometimes leading to erratic behaviour, the DZ boundary is expected to be small and closely tracked to the target value. However, in other cases as in this experiment, the DZ boundary cannot be as closely tracked to the target value. Here the target value (*DecisionBoundary*) is defined by capacity Offset (see (7) later) and this is used by the AM to decide whether or not to deploy a server. And because Offset is populated in `serverCapacity` and depleted in `appCapacity` (i.e., the difference between available and requested capacity) any behaviour shift across the decision boundary (on either side of the boundary) is in excess of `appCapacity`. This means that fluctuations around the decision boundary are usually in multiples of `appCapacity` and to handle erratic behaviour around *DecisionBoundary* the AM will need to take `appCapacity` into consideration when calculating DZ boundaries. This explains the boundary size calculation of (3). Offset is positive when there is excess capacity than required and negative when there is a shortfall. Also, sample simulation results show that smaller sizes of dead-zone boundary have no effect on the system behaviour.

Secondly, the zone areas are defined as follows (two zones are defined with one on either side of the *DecisionBoundary* –see Figures 8 and 9):

```
if (app1DCOffset < app1.appCapacity)
{
        App1.SystemBehaviour = "IsInDeployZone";
}
else
{
        App1.SystemBehaviour = "IsNotInDeployZone";
}
```

Then stability is maintained by persisting the behaviour (*DecisionBoundary* outcome) of the system across the zones as follows:

```
if (app1DCOffset >= app1.appCapacity)
{ App1.SystemBehaviour = "IsNotInDeployZone"; }

if   ((App1.SystemBehaviour  ==  "IsInDeployZone")  &&
      (app1DCOffset < App1.DZUpperBound))
{ App1.SystemBehaviour = "IsInDeployZone"; }
else
{ App1.SystemBehaviour = "IsNotInDeployZone"; }

if  ((App1.SystemBehaviour  ==  "IsNotInDeployZone")  &&
      (app1DCOffset > App1.DZLowerBound))
{ App1.SystemBehaviour = "IsNotInDeployZone"; }
else
{ App1.SystemBehaviour = "IsInDeployZone"; }
```

Thus, the *DecisionBoundary* in *sysAC* and *sysVC*, which is `(app1DCOffset < app1.appCapacity)` now becomes `(App1.SystemBehaviour == "IsInDeployZone")` in *sysDC*. The AM dynamically changes the `DZ.DZConst` value between three values of 1, 1.5 and 2. By doing this the manager is sensitive to its own behaviour and proactively regulates (retunes) its decision pattern to maintain stability and reliability.

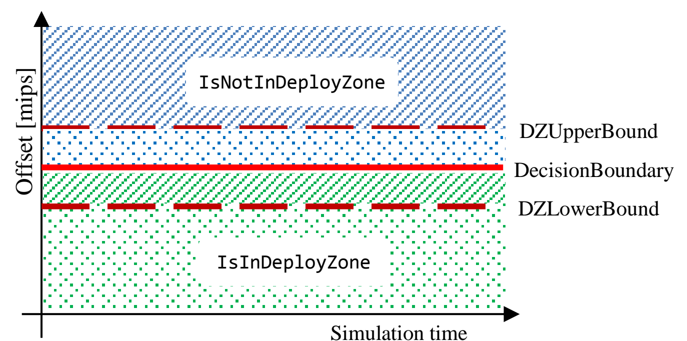


Figure 6: Dead-zone logic implemented by SysDC.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

In Figure 6, the area shaded in green represents the '*IsInDeployZone*', which means the manager should deploy a server while the area shaded in blue represents the '*IsNotInDeployZone*', which means the manager should not deploy a server. Likewise, the dotted shade pattern represents the '*IsInDeployZone*' while the diagonal shade pattern represents the '*IsNotInDeployZone*'. As shown, if, for example, the system behaviour falls within the '*IsNotInDeployZone*' area, the manager persists the action associated to this zone until system behaviour falls below the '*DZLowerBound*' boundary at which point the action associated to the '*IsInDeployZone*' area is activated. This way the AM is able to maintain reliability and efficiency. The AM also retunes its behaviour (as explained earlier) by adjusting *DZWidth* if fluctuation is not reduced to an acceptable level. Thus, three behaviour regions (in which different actions are activated) are defined; 'upper region' (*IsNotInDeployZone* with 'DO NOT DEPLOY SERVER' action), 'lower region' (*IsInDeployZone* with 'DEPLOY SERVER' action), and 'in DZ' (within the DZWidth with either of the two actions). It is important to note, as shown in Figure 6, that within the DZ boundary (i.e., the 'in DZ' region), either of the actions associated to '*IsInDeployZone*' and '*IsNotInDeployZone*' areas could be maintained depending on the 'current action' prior to deviation into the 'in DZ' region. So actions activated in the 'upper region' and 'lower region' are respectively persisted in the 'in DZ' region. This is further explained in Figure 7, which shows the resultant effect of the DZ logic in terms of what zone action is activated per time.

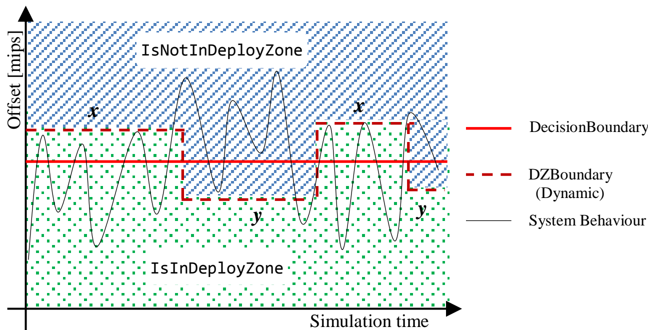


Figure 7: Illustration explaining actual performance effect of DZ logic.

Figure 7 explains what happens in Figure 6. As system behaviour fluctuates around decision boundary, the manager dynamically adjusts the DZ boundary to mitigate erratic adaptation. As shown, minor deviations across the DecisionBoundary do not result in decision (or action) change. In this case (Figure 7) actions for `IsInDeployZone` and `IsNotInDeployZone` are persisted at states *x* and *y* respectively despite system behaviour crossing the decision boundary at those state points.

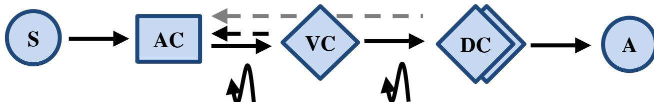


Figure 8: Structural representation of sysDC.

Figure 8 is a representation of the next level of sophistication in autonomic architecture life-cycle required to ensure dependability. This is presented in Section II as Figure 3 (v).

To illustrate the overall operation of the DZ logic, a simple numeric example is given: Let us consider a simple use-case example in which a room temperature controller is set to maintain temperature at 20^0C: The AM is configured to turn ON heating when room temperature falls below the target temperature (20^0C) and to turn OFF heating otherwise. If, for example, the room temperature keeps fluctuating between 19^0C and 21^0C the manager will as well fluctuate with its decisions (i.e., erratic behaviour of frequently turning heating ON and OFF). This situation is undesirable and can be enormously costly in crucial systems. To mitigate this situation, the manager can implement DZ logic with a DZLowerBound of 19^0C and DZUpperBound of 21^0C. This will allow the manager to turn off heating only when room the temperature rises above 21^0C and to turn on heating only when it falls below 19^0C. Putting this in the context of (20) means that, e.g.:

```
DZUpperBound = (20 + (20 * 0.05))
DZLowerBound = (20 - (20 * 0.05))
```

This will calm the erratic behaviour of the AM. However, if the erratic behaviour does not drop to an acceptable level the manager can further retune itself by increasing *DZConst* by multiples of 0.05 (e.g., *DZConst* += 0.05). If on the other hand the AM discovers that it is not making decisions frequently enough, (i.e., the room is getting too cold or too hot) it can retune its behaviour to increase its rate of decision-making by reducing the DZ boundaries (e.g., *DZConst* –= 0.05). So the AM retunes itself by dynamically adjusting the DZ boundaries using (*DZConst* ±= 0.05) as appropriate. It is important to note that the average of the DZ boundaries is equal to the target goal – e.g., the average of 19^0C and 21^0C is 20^0C, which is the target temperature.

D. Simulation Scenarios and Metrics

In the following simulations to analyse the performances of the three systems (*sysAC*, *sysVC* and *sysDC*), four simulation scenarios are used. The scenarios are presented in Table I. The user of the TAArch application can define further scenarios as required.

Table I: Resource allocation simulation scenarios

Scenario	Description	Metrics
Scenario 1	Basic simulation with uniform request rate and application size	SLA Delay cost Server deployment rate Optimum provisioning (Offset analysis)
Scenario 2	Basic simulation with uniform request rate and varying application sizes	
Scenario 3	Uniform application size with burst injected at a particular time in the simulation	
Scenario 4	Varying application sizes with inconsistent request rate	

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Scenario 1: In scenario 1, all parameters are kept constant except those (e.g., *DZConst*) that may need dynamic tuning by the manager as need arises. This scenario gives a default view of the behaviour of the managers under normal condition. Under this scenario of normal condition, it is expected that all managers will behave significantly closely.

Scenario 2: This scenario creates a condition where the managers will have to deal with irregular sizes of service request. This leads to contention between applications –huge applications will demand huge resources thereby starving smaller applications. Performance analysis here will include individual application analysis. Request rate is kept constant so that the effect of varying application sizes could be better analysed.

Scenario 3: In this scenario, request rate and application size are kept constant while burst is injected at a chosen time (*SimulationTime*) in the simulation. This is similar to Scenario 1 just that a sudden and unexpected disruption (burst) is injected into the system. This will measure the robustness of the AMs in adhering to the goal of the system. The impact of the burst is relative to the size of the burst (*BurstSize*).

Scenario 4: This is the most complex scenario with resource contention and two instances of burst injection. This scenario creates the combined effect of Scenarios 2 and 3 put together. Request sizes vary leading to resource contention and request rate is highly erratic. Inconsistent request rate can also lead to '*flooding*', which also is a kind of burst. *Flooding* is a situation where the system is inundated with requests at disproportionate rate.

All metrics are mathematically defined giving the reader a clear picture of the definition criteria should they wish to replicate this experiment.

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:

$$SLA = \begin{cases} \frac{ProvisionedCapacity}{RequestedCapacity} & \text{(i)} \\ \frac{AvailableCapacity}{RunningCapacity} & \text{(ii)} \end{cases} \quad (4)$$

Where *ProvisionedCapacity* is the total deployed server capacity (excluding those in queue and including those already reclaimed back to the pool) and Request*edCapacity* is the total size of request (including completed requests). *AvailableCapacity* is *ProvisionedCapacity* minus capacity of reclaimed servers (*ReclaimedCapacity*) while *RunningCapacity* is the total size of request (excluding completed requests). In (4), (i) is more of a whole picture consideration –considering the entire capacity activities of the system while (ii) 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 indicating 100%. Values above 1 indicate over-provisioning while values under 1 indicate shortfall. Optimum provisioning is achieved at close proximity to 1.

Delay cost: Delay cost can be calculated in many different ways as the cost can be influenced by many delay contributors. In this instance, delay cost is defined as the cost (in capacity) as a result of the delay experienced by the servers. This delay affects the completion time of service requests. This is mathematically represented as:

$$DelayCost = \frac{(DeployedCapacity - ProvisionedCapacity)}{DeployedCapacity} = \frac{ProvisioningCapacity}{DeployedCapacity} \quad (5)$$

ProvisioningCapacity is the capacity of servers in queue while *DeployedCapacity* is the total capacity of all deployed servers. The lower value of delay cost means the better performance of the system.

Deployment Rate: Server (re)deployment rate is the ratio of server deployment to service request. It measures the frequency at which managers deploy servers with regards to the nature of requests. This is mathematically represented as:

$$DeploymentRate = \frac{DeployedCapacity}{(RequestedCapacity - CompletedCapacity)} \quad (6)$$

The lower value of deployment rate means the better performance of the system translating to better maximisation of throughput.

Optimum provisioning: This metric is also an offset analysis. It indicates whether and when the manager is over or under provisioning. This is also known as efficiency calculation. Offset is calculated as:

$$Offset = AvailableCapacity - RunningCapacity \quad (7)$$

Under normal circumstances, average offset is not expected to fall below zero. The system is optimally provisioning when offset falls between zero and the average capacity of all applications. The closer to zero the offset value is, the better the performance of the system.

Note that, for all metrics, low or high values do not always necessarily translate to better performance. It is not usually realistic for the supposed better manager to always outperform the other managers. There are times when the manager underperforms and usually there may be a tradeoff of some kind that explains the situation.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

E. Experimental Results

Results are presented and analysed according to simulation scenarios. For precise results, ten different simulations of each Scenario are performed and results presented are based on average of these ten simulations. For each of the ten simulations, the parameters used are presented. It is important to note the workload and parameters used for individual simulations as results will largely depend on those.

Scenario 1: Basic simulation with uniform request rate and application size

Table II is a collection of major parameters used in this scenario. The number of requests and the distribution of those requests amongst applications differ with each AM as they are dynamically generated and unpredictable. This does not distort the results as analysis is based on system-wide performance and not on individual application performance.

Table II: Scenario 1 simulation parameters

Parameter		Value
# of servers		300
# of applications		4
Request rate		1 req/sec
Application capacity (MIPS)		20000
Server capacity (MIPS)		40000
Internal variables	RetrieveRate	5x
	RequestRateParam	10
	RetrieveRequestParam	0.2
	ServerProvisioningTime	3 (1.5 sec)
Managers (*sysAC*, *sysVC* & *sysDC*)		PeM
DZConst		1.5

In every simulation, there are 300 servers of 40000 MIPS capacity each. This means there is a total of initial 12000000 MIPS to share between requests for four applications (App1, App2, App3, and App4). Reclaimed servers are later added to this available capacity. If the total requested capacity is higher than the total provisioned capacity, the unused server list will be empty (leaving the manager with a deficit of outstanding requests without resources to service them) and the datacentre is overloaded. So the simulation stops whenever any manager runs out of resources (i.e., when the unused server list of any manager becomes empty). It is necessary to stop the simulation at this point because as soon as the unused server list of a particular manager becomes empty, the *RequestedCapacity* for that manager starts piling up while *AvailableCapacity* remains at zero, which leads to continuously increasing negative *Offset*. This will lead to inaccurate assessment of the three managers (recall that all three managers are compared concurrently and it is safer to do this while all three managers are active). Also, at this point, usually, other managers may have outstanding resources and this will mean better efficiency. Table III is a number distribution of requests and services for ten simulation runs of Scenario 1. The values shown are collected at the end of each simulation, for example, it can be seen that the manager of *sysAC* has no servers left in each of the simulations while *sysVC* has a couple and *sysDC* even more. Though *sysAC* and *sysVC* are able to service almost the same number of requests, *sysVC* has outstanding server capacity and could service more requests. However, the additional smartness of *sysVC* does not always translate to better performance as highlighted in Table III (this is an example of manager interference leading to overcompensation). *sysDC* clearly outperformed the others with an average of about 36 outstanding servers out of 300 initial servers. Figures 11-14 give a breakdown of the performances.

Table III: High level performance analysis of managers over ten simulation runs of Scenario 1

Sim	unused server			serviced request			deployed server		
	AC	VC	DC	AC	VC	DC	AC	VC	DC
1	0	2	35	578	577	555	307	307	268
2	0	3	27	594	594	574	310	299	278
3	0	0	36	600	590	574	309	305	268
4	0	0	34	593	585	566	309	313	274
5	0	0	30	609	586	587	312	303	273
6	0	0	38	597	586	576	308	309	268
7	0	0	36	613	605	587	314	304	268
8	0	15	39	591	590	565	307	287	263
9	0	6	33	582	582	566	304	302	271
10	0	8	48	569	567	542	310	298	255
avg	**0**	**3.4**	**35.6**	**592.6**	**586.2**	**569.2**	**309**	**302.7**	**268.6**

The difference between requested capacity and provisioned capacity (or in real time analysis, running capacity and available capacity) is known as *Offset*. Where offset is close to zero, the difference with respect to running and available MIPS is low and the AM is therefore very efficient. When offset is much greater than or much less than zero, the AM is over-provisioning or under-provisioning respectively and is very inefficient. The AMs are designed to have a window of 'optimum provisioning' defined by the interval ($0 \leq Offset \leq AvgAppCapacity$), which means that the AM are configured to maintain *AvailableCapacity* of up to average *appCapacity* for just-in-time provisioning. However, AM efficiency is defined by its ability to maintain *Offset* as close as possible to zero. Figure 9 shows the efficiency analysis of the three managers in terms of maximising resources. This is in terms average performances of the three AMs over ten simulation runs. This means that the same scenario was run for ten times and then the average result was calculated. This gives a clearer picture and more accurate analysis of manager performance.

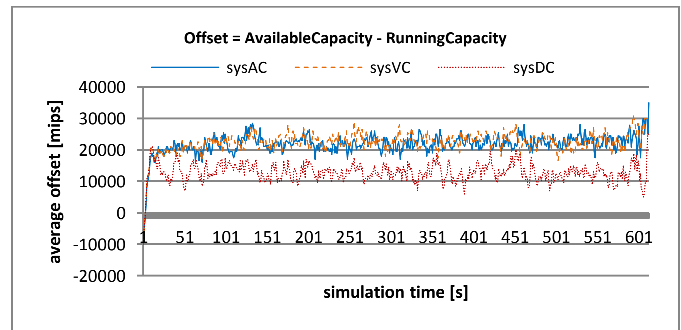


Figure 9: Manager efficiency analysis for scenario 1.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Figure 9 shows that, in terms of efficiency, *sysAC* performed significantly similar to *sysVC* with a couple of instances where *sysAC* also performed better than *sysVC*. This is as a result of over compensation introduced by the extra level of smartness in *sysVC*. The validation check of *sysVC* gives it an advantage over *sysAC* but it sometimes leads to over compensation. For example, though *sysVC* checks to ensure resource availability against resource requests, it is not adequately sensitive to erratic request fluctuation. High level of erratic request fluctuation disorientates *sysVC* (as can be seen in later scenarios where burst is injected) but this effect is naturally and dynamically handled by *sysDC*. *sysDC* takes a longer term look at the self-management effect on the datacentre and retunes its self-management behaviour. The rate at which the managers change decision, (which can indicate erratic behaviour) is indicated by the gap between the crests and troughs of the graph in Figure 9. Smaller gap indicates erratic change of decision while bigger gap indicates more persisted decision. As seen, *sysDC* has significantly more persisted decisions and this allows it to more adequately track resource availability against resource requests, which leads to more efficient performance as can be seen. Recall that optimum provisioning is defined by the ($0 \leq Offset \leq AvgAppCapacity$) interval, which in this case is between 0 and 20000 MIPS. *sysDC* clearly falls within this range, though a bit towards the 20000 border. This means that while *sysAC* and *sysVC* try to maintain *AvailableCapacity* of up to 20000 MIPS for just-in-time provisioning, *sysDC* efficiently depletes this reserve to maximise resources while at the same time maintaining the same level of performance and even better compared to the other two. This is evidently seen in the following deployment rate, SLA, and cost metrics analyses.

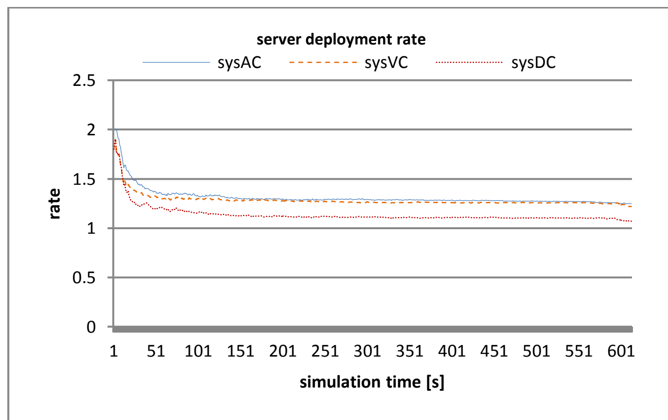


Figure 10: Server deployment rate analysis for scenario 1.

Figure 10 shows the rate at which the three AMs deploy servers as requests arrive. With the same request rate, the AMs deployed servers differently. While *sysAC* deployed the most servers, *sysDC* deployed the least servers. This explains why *sysAC* easily runs out of servers followed by *sysVC* while *sysDC* still retains a couple of unused servers (Table III). Interestingly, this does not negatively affect the performance of *sysDC* and when *sysDC* underperforms in one aspect there is usually compensation (say tradeoff) in another aspect. The lower server deployment rate of *sysDC* resulted in lower SLA value of *sysDC* (when compared to *sysAC* and *sysVC* –Figure 11) but this only keeps the value very close to the optimum value of 1, which also indicates high efficiency.

Figure 11 depicts the service levels of the three AMs with the zoomed-in inset revealing the gaps between their performances. As expected, following from the result trend above, *sysAC* and *sysVC* performed quite similarly with each outperforming the other in some places. *sysDC* on the other hand, keeps SLA as close as possible to the target goal of 1 (a perfect system would keep SLA at 1). *sysDC* has the ability to dynamically scale down unnecessary and inefficient provisioning by proactively throttling oscillation. This capability also leads to cost savings as shown in Figure 12.

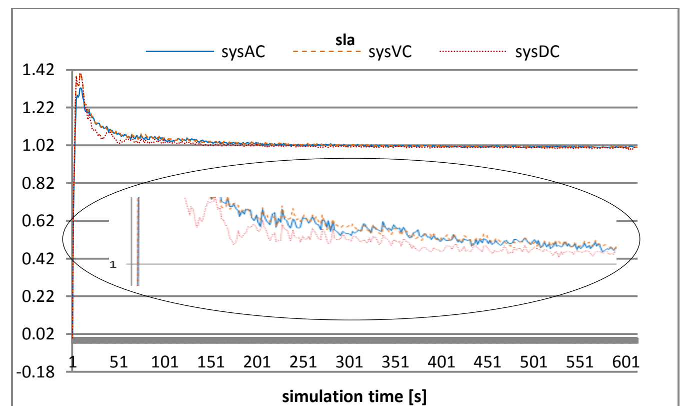


Figure 11: Service level achievement (SLA) analysis for scenario 1.

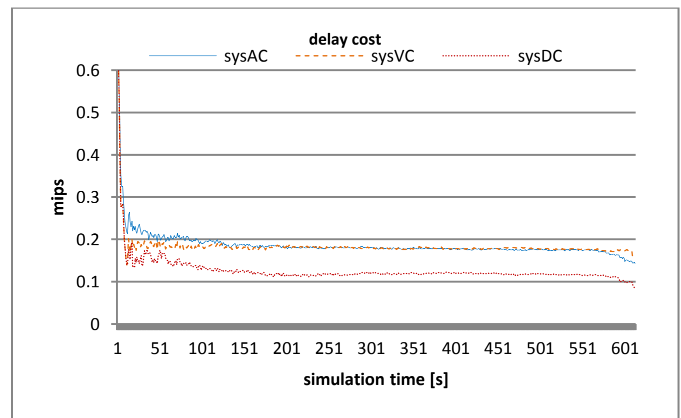


Figure 12: Delay cost analysis for Scenario 1.

The high level of deployment rate (i.e., deploying more MIPS than required) for *sysAC* and *sysVC* (Figure 10) leads to high cost (in terms of excess MIPS) of servicing individual requests. Also this means that the rate at which servers enter the provisioning queue is much higher than the rate they leave the queue. This results in an increasing number of redundant servers in the queue, which contributes to delay cost (Figure 12). Also, the number of redundant servers for *sysDC* is doubled by that of *sysAC* and *sysVC*.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

The results analyses of Scenario 1 indicate that the proposed TAArch (represented by *sysDC*) has significant performance improvement over existing architectures. This assertion is further tested in the following scenarios.

Scenario 2: Basic simulation with uniform request rate and varying application sizes

Table IV is a collection of the major parameters used in this scenario.

Table IV: Scenario 2 simulation parameters

Parameter		Value
# of servers		300
# of applications		2
App capacity (MIPS)	App1	30000
	App2	5000
Request rate		1 req/sec
Server capacity (MIPS)		40000
Internal variables	RetrieveRate	5x
	RequestRateParam	10
	RetrieveRequestParam	0.2
	ServerProvisioningTime	3 (1.5 sec)
Managers (*sysAC*, *sysVC* & *sysDC*)		PeM
DZConst		1.5

In this scenario, there are 300 servers of 40000 capacity each to be shared amongst two applications (App1 and App2). This means there is a total of initial 12000000 MIPS to share between requests for App1 with 30000 MIPS and App2 with 5000 MIPS. The capacity gap between the two applications is so wide that it may naturally lead to contention with App1 demanding more resources than App2. In this kind of situation where it is easy to underserve one application because of the contention, it is left for the datacentre autonomic managers to decide how best to efficiently allocate resources. Results show that while *sysAC* maintained a proportionate resource allocation (in terms of applications) for the two applications, *sysVC* and *sysDC* prioritised provisioning for App1 with much higher MIPS request. One disadvantage of proportionate provisioning is that it treats requests according to applications (in this case two applications) and not according to capacity (in this case 30000 versus 5000). When this happens, the high capacity application (App1) will be heavily under-provisioned while the low capacity application (App2) will be adequately provisioned (and sometimes over-provisioned) compared to the level of provisioning for App1 as shown in Figure 14 (a) for *sysAC* Offset analysis. Also this amounts to inefficiency and explains why *sysAC* easily exhausts its resources as shown in Table V. Table V shows the results of requests distribution amongst the three managers.

The 'dropped/queued request' analysis shows that in prioritising App1, *sysVC* and *sysDC* dropped more of App2 requests while *sysAC*, which does not drop any application, struggled to cope with the capacity imbalance. For a clearer picture Figure 13 shows how *sysVC* and *sysDC* prioritised App1 over App2.

Table V: High level performance analysis of managers over ten simulation runs of Scenario 2

Sim.	unused server			serviced request			deployed server		
	AC	VC	DC	AC	VC	DC	AC	VC	DC
1	0	118	127	423	242	231	399	227	207
2	0	113	125	465	263	251	422	233	213
3	0	132	145	450	234	225	418	211	191
4	0	120	113	447	248	254	411	211	223
5	0	124	122	440	246	243	405	218	218
6	0	100	120	451	259	250	413	237	221
7	0	108	127	470	265	253	420	239	208
8	0	96	114	434	262	258	404	236	228
9	0	102	116	458	261	257	413	241	222
10	0	107	112	428	250	249	394	225	219
avg	**0**	**112**	**122.1**	**446.6**	**253**	**247.1**	**409.9**	**227.8**	**215**

As can be seen in Figure 13, there is a consistent trend of high rate of dropped App2 requests. This means that more resources were allocated to App1 and thereby starving App2. As this continued, it led to more App2 being dropped as there were limited resources per time to service App2 requests. Also noticeable is the smoothness of provisioning for App1 compared to the bumpiness of provisioning for App2 –this is further explained in the Offset analysis that follows.

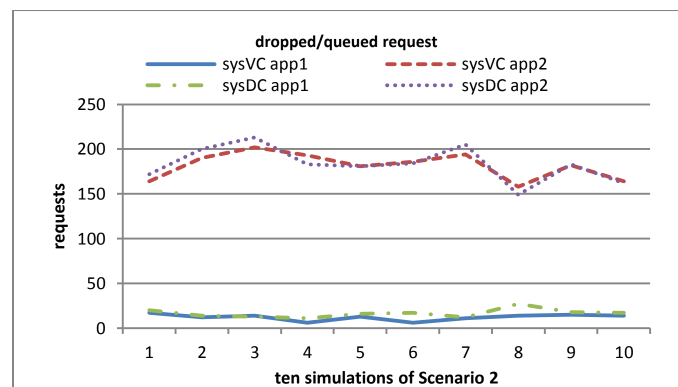


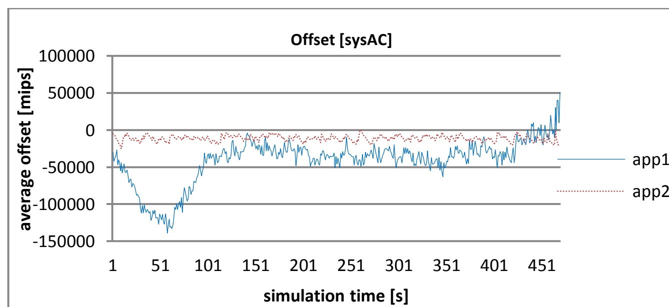
Figure 13: Dropped/queued request analysis for Scenario 2.

sysAC on the other hand did not drop any request and trying to evenly joggle resources between the highly imbalanced MIPS requests for the two applications meant that more resources per time than necessary are used. This explains why *sysAC* exhausted its resources quite early in the simulation while the other managers have hundreds of servers still unused (Table V). Figure 14 (a) shows that while App2 is about adequately provisioned, App1 is heavily under-provisioned. This is because *sysAC* evenly provisioned for the two applications thereby starving App1, which has very high MIPS requests. So by accepting all requests despite low resource availability *sysAC* under-provisioned for App1 far more than it did for App2 because of the large size of App1 requests. There is no check in *sysAC* to ensure resource availability before requests are accepted.

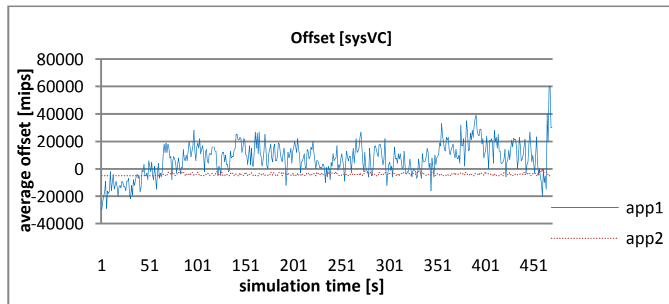
In Figure 14, App2 offset is maintained at (0 ≥ -18000 MIPS) by *sysAC*, (-1666 ≥ -5000 MIPS) by *sysVC* and (0 ≥ -5000 MIPS) by *sysDC*. Also, App1 offset ranges between (50000 and -139000 MIPS) for *sysAC*, (60000 and -30000 MIPS) for *sysVC* and (30000 and -30000 MIPS) for *sysDC*.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

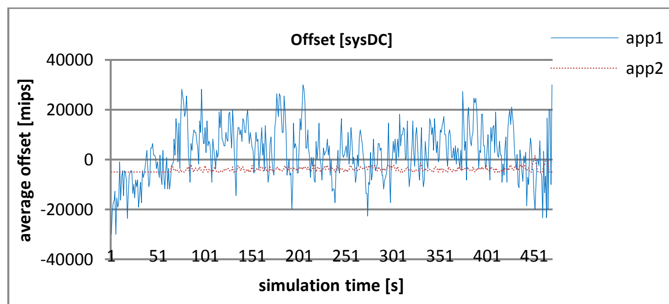
This shows that while *sysAC* treats requests according to applications (i.e., by trying to evenly provision for both applications), *sysVC* and *sysDC* are sensitive to the individual size of requests. As a result, by taking on all requests and attempting an even distribution of resources for both applications, *sysAC* heavily under-provisions for App1 and this also affected its performance for App2. *sysVC* and *sysDC* on the other hand, maintained more balanced resource allocation for both applications in terms of request capacity with *sysDC* showing higher efficiency than *sysVC*. Note that a positive Offset above the optimal provisioning mark amounts to over-provisioning while a negative Offset amounts to under-provisioning. Recall that optimal provisioning mark is defined by the interval ($0 \leq Offset \leq AvgAppCapacity$), which in this case is (0 ≤ Offset ≤ ((30000 + 5000)/2)) –that is, between 0 and 17500 MIPS.



(a) *sysAC* Offset analysis for App1 and App2. App2 is about adequately provisioned (i.e., Offset ≈ 0) while App1 is heavily under-provisioned



(b) *sysVC* Offset analysis for App1 and App2. App2 is about adequately provisioned while App1 over-provisioned (well above the optimal provisioning mark, which is defined by $0 \leq Offset \leq AvgAppCapacity$)



(c) *sysDC* Offset analysis for App1 and App2. App2 is about adequately provisioned while App1 is slightly over-provisioned (slightly above the optimal provisioning mark, which is defined by $0 \leq Offset \leq AvgAppCapacity$)

Figure 14: Individual Offset analysis for scenario 2.

Figure 15 shows the average manager efficiency analysis for all three systems. On the average *sysAC* did not stand up to the complex provisioning condition of Scenario 2 as did the other systems. Figure 15 shows that *sysAC* could not efficiently cope with the level of resource contention experienced between App1 and App2. *sysVC* and *sysDC* show almost the same level of autonomic sophistication however, *sysDC* is shown to be more efficient. Although both systems have the same least under-provisioning value of -17500 MIPS, *sysVC* recorded a maximum over-provisioning value of 27500 MIPS (well above the optimal provisioning mark of 17500) while *sysDC* recorded a maximum positive Offset value of 13500 MIPS (below the optimal provisioning mark). This indicates that *sysDC* is efficiently more sophisticated in handling complex resource allocation scenario that would ordinarily prove difficult for traditional autonomic managers (*sysAC* and *sysVC*) to handle. E.g., this increased efficiency arises from the fact that the *DependabilityCheck* sub-component of *sysDC* enables it to go beyond dropping requests if there are insufficient resources to deploying resources only when it is necessary and efficient to do so.

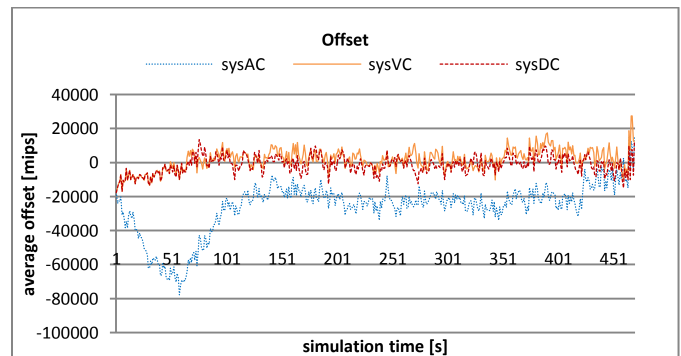


Figure 15: Manager efficiency analysis for scenario 2.

The results analysis of Scenario 2 is a further corroboration of the assertion that the TAArch architecture (represented by *sysDC*) has significant performance improvement over existing architectures. There are two more simulation scenarios to further test this assertion.

Scenario 3: Uniform application size with burst injected at a particular time in the simulation

In this scenario, request rate and application size are kept constant while burst is injected at a particular time (200s) in the simulation. This is similar to Scenario 1 just that a sudden and unexpected disruption is injected into the system. This simulation will measure the robustness of the AMs in adhering to the goal of the system. Another important factor to look at is how long it takes the AMs to recover from the disruption caused by the burst. The impact of the burst is relative to the size of the burst, (which in this case is 2500 ms). Table VI is a collection of major parameters used.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Table VI: Scenario 3 simulation parameters

Parameter		Value
# of servers		300
# of applications		4
Request rate		1 req/sec
Application capacity (MIPS)		20000
Server capacity (MIPS)		40000
Internal variables	RetrieveRate	5x
	RequestRateParam	10
	RetrieveRequestParam	0.2
	BurstSize	2500ms
	ServerProvisioningTime	3 (1.5 sec)
Managers (*sysAC*, *sysVC* & *sysDC*)		PeM
DZConst		1.5

In every simulation, there are 300 servers of 40000 MIPS each. This means there is a total of initial 12000000 MIPS to share between four applications (App1, App2, App3, and App4). Reclaimed servers are subsequently added to this available capacity. The managers receive requests and allocate resources accordingly as long as *AvailableCapacity* is not zero. The reliability of a manager will be measured by its ability to remain efficient under almost all perceivable operating circumstances. Table VII is a number-distribution of requests and services for ten simulation runs of Scenario 3.

Table VII: High level performance analysis of managers over ten simulation runs of Scenario 3

	unused server			serviced request			deployed server		
	AC	VC	DC	AC	VC	DC	AC	VC	DC
1	0	68	89	453	417	407	306	240	211
2	0	55	74	564	431	418	309	253	230
3	0	61	90	467	430	415	309	248	216
4	0	63	86	481	439	423	307	242	220
5	0	59	79	482	447	431	312	255	232
6	0	57	87	462	426	412	304	246	214
7	0	69	93	444	408	391	307	235	219
8	0	67	94	455	420	404	302	238	209
9	0	63	95	463	424	408	305	248	213
10	0	58	80	453	420	410	304	247	226
avg	**0**	**62**	**86.7**	**472.4**	**426.2**	**411.9**	**306.5**	**245.2**	**219**

On the average, from Table VII, *sysAC* had initiated about 46.2 requests (924000 MIPS) more than *sysVC* and about 60.5 requests (1210000 MIPS) more than *sysDC* but has no extra capacity left to proceed beyond this point. However, *sysVC* and *sysDC* both have about 2480000 MIPS and 3468000 MIPS extra capacity respectively. This means that, under normal circumstances, both systems (*sysVC* and *sysDC*) could conveniently provision for about additional 124 and 173.4 requests respectively. Clearly, *sysDC* is seen to have outperformed the other systems. This is principally because the dead-zone logic of *sysDC* helps it to significantly reduce the number of activated decision boundaries. This means that decisions are not erratically taken, which leads to high efficiency and reliability. Figures 17 – 20 give a breakdown of the performances.

Figure 16 shows how all three managers reacted to the disruption injected at 200s. While *sysVC* and *sysDC* were able to recover after about 9s each (with *sysDC* a bit less than that), it took *sysAC* about 90s to recover. We can also see that *sysDC* reasonably maintained provisioning within the optimal provisioning mark, which in this case is between 0 and 20000 MIPS. There is also a noticeable trend that suggests an extra level of autonomic sophistication in *sysDC* which is also a sign of reliability. Notice that within pre disruption and post disruption recovery both *sysAC* and *sysVC* maintained their level of performances (which nonetheless is averagely about 5000 MIPS above the optimal provisioning mark) while *sysDC*, within the same time frame, switched between two levels of performance as shown by the solid black line. This is the effect of dynamic (re)tuning of the *DZWidth* by *sysDC*. This capability enables *sysDC* to systematically track the system's goal (in this case maintaining reliability and efficiency within the optimal provisioning mark) by dynamically retuning its decision boundary. As shown in Figure 16, before the disruption *sysDC* maintained a steady and continuous level of efficiency by keeping *DZConst* at 1.5 but as soon as the disruption sets in it quickly retunes itself and reduced the *DZConst* to 1. At this point the manager stopped accepting further requests (as the datacentre is now receiving torrential streams of requests) but the initial shock (caused by the lag between when the disruption started and when the manager shuts its door) meant that a few resources were released to mitigate the effect of the situation. This will instantly start pushing up *Offset* until the datacentre normalises and then as shown *sysDC* retunes its decision boundary by returning *DZConst* back to 1.5. So while *sysAC* is heavily affected by a disruption of this magnitude and *sysVC* shows a remarkable level of robustness, *sysDC* shows a longer term ability to sensitively throttle its behaviour to efficiently and reliably track the goal of the entire system.

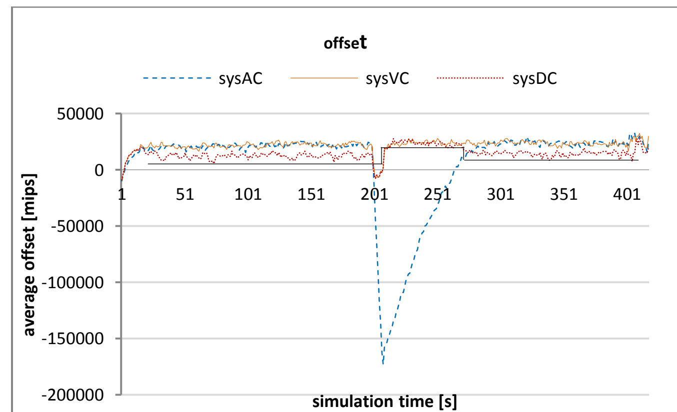


Figure 16: Manager efficiency analysis for scenario 3. The black solid line indicates sysDC's dynamic tuning of dead-zone boundary. The manager started with a *DZConst* of 1.5 (left lower part of the line) then changed to *DZConst* of 1 (high part) and then back to *DZConst* of 1.5.

Figure 17 shows that while *sysVC* and *sysDC* responded to the disruption by rejecting requests as soon as they were overwhelmed thereby pushing down their server deployment rate, *sysAC* responded by deploying even more servers to meet the current service demand. Despite deploying more servers *sysAC* still could not meet up with demand rate, which ultimately affected its SLA achievement (Figure 18). This is because the provisioning rate, (which is dependent on *ProvioningTime*) could not keep up with the rate at which

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

servers are deployed. As a result *sysAC* had more servers (almost tripling that of *sysDC*) overshooting their *ProvisioningTime* thereby getting redundant and pushing up delay cost as well.

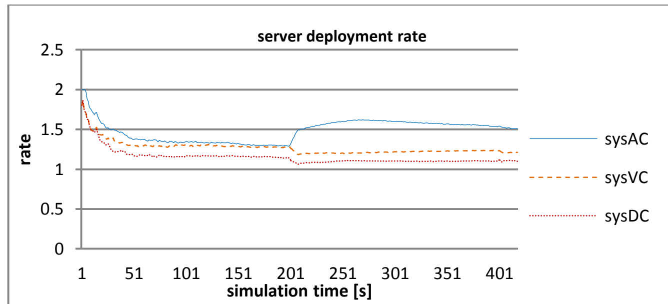


Figure 17: Server deployment rate analysis for scenario 3.

As the datacentre settles (after the disruption) *sysAC* starts normalising the rate of server deployment but because there is already a huge backlog of requests (about 173000 MIPS as in Figure 16) it takes *sysAC* a long time to recover. This also contributes to why it quickly exhausts its resources. *sysVC* and *sysDC* on the other hand, with a small backlog of about 7500 MIPS, need not deploy more resources than the ordinary (Figure 17) but gradually absolves the backlog allowing them to quickly recover.

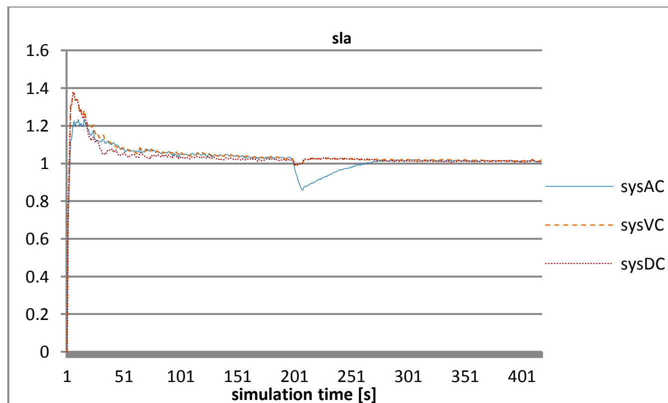


Figure 18: Service level achievement (SLA) analysis for scenario 3.

High level of deployment rate (inefficient deployment of more MIPS than necessary) also leads to high cost (in terms of excess MIPS) of servicing individual requests. This means that the rate at which servers enter the provisioning queue is much higher than the rate they leave the queue. The rate for *sysAC* almost doubles that of *sysVC* and almost triples that of *sysDC*. This leads to increasing number of redundant servers in the queue, which contributes to delay cost.

The results analysis of Scenario 3 shows that it is absolutely inefficient and unreliable to run a datacentre with a manager based on *sysAC*. While *sysVC* based AMs are more robust, their robustness is limited in terms of the extent of sensitivity to system's goal under unfamiliar circumstances in which *sysDC* based AMs are more sophisticated and dynamically reliable. This further corroborates the assertion that the TAArch architecture (*sysDC*) has significant performance improvement over existing architectures.

Scenario 4: Varying application sizes with inconsistent request rate

This is the most complex scenario with a combined effect of Scenarios 2 and 3 put together. The complexity presented by this scenario (i.e., a combined effect of resource contention and two injected disruptions) allows us to further test the robustness of these systems by stretching their capabilities to extremes. Table VIII is a collection of the major parameters used in this scenario. As in previous scenarios, results presented are based on average of ten different simulation runs.

Table VIII: Scenario 4 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	1500ms
	ServerProvisioningTime	3 (1.5 sec)
Managers (*sysAC*, *sysVC* & *sysDC*)		PeM
DZConst (initial)		1.5

In every simulation of this scenario, there are 400 servers of 40000 MIPS 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 IX is a number distribution of requests and services for ten simulation runs of Scenario 4.

Table IX: High level performance analysis of managers over ten simulation runs of Scenario 4

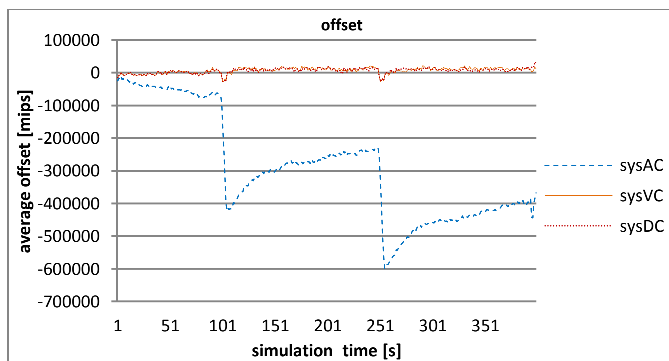
	unused server			serviced request			deployed server		
	AC	VC	DC	AC	VC	DC	AC	VC	DC
1	0	109	120	474	395	394	435	339	316
2	0	124	133	465	387	382	433	325	303
3	0	123	125	471	400	397	443	330	314
4	0	112	114	473	395	400	439	343	321
5	0	114	130	476	398	402	440	335	304
6	0	118	124	473	393	398	439	331	308
7	0	115	117	468	393	394	437	336	320
8	0	113	122	468	398	396	435	330	307
9	0	113	116	476	395	401	444	342	322
10	0	110	115	476	398	394	446	337	323
avg	**0**	**115**	**122**	**472**	**395**	**393**	**439**	**335**	**314**

Results reveal that *sysAC* is not adequately robust in such complex situations as in Scenario 4. The system is heavily inefficient in handling this type of situation (Figure 19 (a)). Its algorithm, which maintains proportionate provisioning with respect to number of applications as against

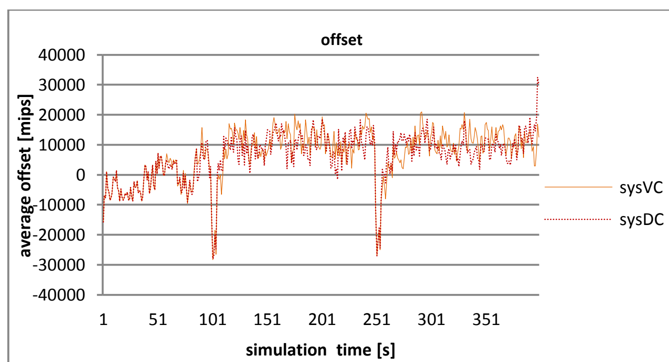
2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

capacity of requests, was disorientated by the level of contention and disruption experienced.

As shown in Figure 19 the first burst was injected at 100s while the second was injected at 250s. *sysAC* is limited in its ability to handle complex situations and so cannot be relied upon to operate large scale and complex datacentres. *sysVC* and *sysDC* both have a wide range of operability in complex situations. However, a closer look at *sysVC* and *sysDC* in this scenario reveals a unique change in expected (as observed in previous results) trend. The highlighted bits of Table IX show that *sysDC* dropped fewer requests than *sysVC* and thereby initiating more requests. Under normal circumstances, as observed in previous scenarios, *sysVC* usually would drop fewer requests than *sysDC*. In this situation the level of disturbance (as a result of resource contention and erratic request disorder) in the datacentre led to instability in *sysVC*, which caused it to over react by inefficiently dropping requests. This instability reveals a weakness in design because in real-life datacentres such disturbances (like sudden request spikes) do occur and managers are expected to adequately stabilise the entire system under such circumstances. *sysDC* on the other hand, with the capability of a longer term view of the entire system, was able to take on more requests.



(a) Manager efficiency analysis of all three systems



(b) Manager efficiency analysis for *sysVC* and *sysDC*.

Figure 19: Manager efficiency analysis for Scenario 4. Bursts affect all managers at 100s and 250s time frames

However, this achievement is with associated tradeoff in delay cost (Figure 20). This shows that *sysDC* is more sensitive to the relationship between requested MIPS and available MIPS. For example, in a situation where *sysVC* dropped a number of requests following a fixed decision boundary (when there is lack of immediate available resources to handle incoming requests), *sysDC* used a dynamic decision boundary to accommodate more requests allowing it to efficiently use up its available resources. By taking on more requests, *sysDC* trades off delay cost, which is not so much of importance but at the same time improves scheduling efficiency, which is of more importance. Interestingly, the efficiency level is not affected –Figure 19 (b) shows that there is no significant difference in efficiency performance of both *sysVC* and *sysDC*. So what we have is a situation where, on the average, *sysDC* utilised significantly fewer resources (313.8 : 334.8 servers) to serve slightly higher amount of requests (395.8 : 395.2 requests) as *sysVC* (Table IX) resulting in improved efficiency (Figure 19 (b)) for *sysDC* and approximately same level of SLA (Figure 21) and delay cost (Figure 20) achievement for both *sysVC* and *sysDC*.

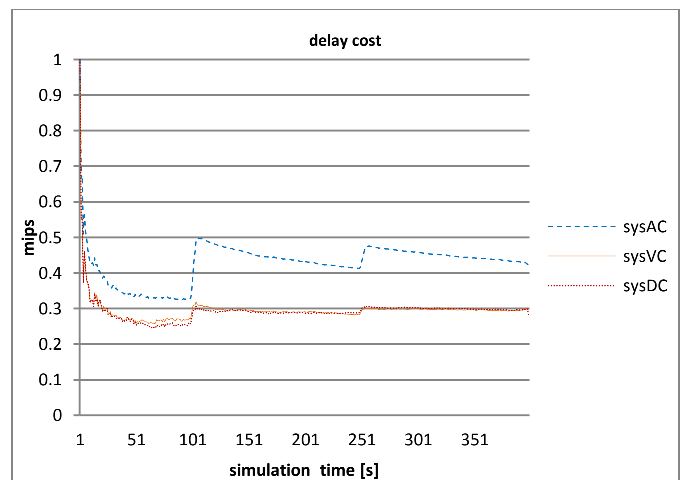


Figure 20: Cost analysis for Scenario 4.

There is consistent corroboration of the fact that *sysAC* is limited in the range of its operational scope when it comes to complex situations. Scenario 4 results show that it is highly expensive, inefficient and unreliable to operate complex datacentres with autonomic managers based on *sysAC*. However, *sysAC* based managers may suffice for simple and basic datacentres. On the other hand, *sysDC* has shown consistent reliability in all tested scenarios. The level of robustness exhibited in this scenario by *sysDC* is a clear indication that it is not a hard-wired one-directional self-managing system. For example, in this scenario we have seen that *sysDC* does not only act when *sysVC* is taking more actions than necessary but also when it is taking fewer actions than necessary. So it can be said that *sysDC* is capable of reducing inefficient adaptation (e.g., when *sysVC*'s decisions are erratic) as well as increasing adaptation when it is necessary and efficient to (e.g., when *sysVC* is not making decisions frequently enough). This capability of increased adaptation is shown in Table IX and illustrated in Figures 20

to 22 –*sysDC* is able to maximise resources while achieving the same level of performance as *sysVC*.

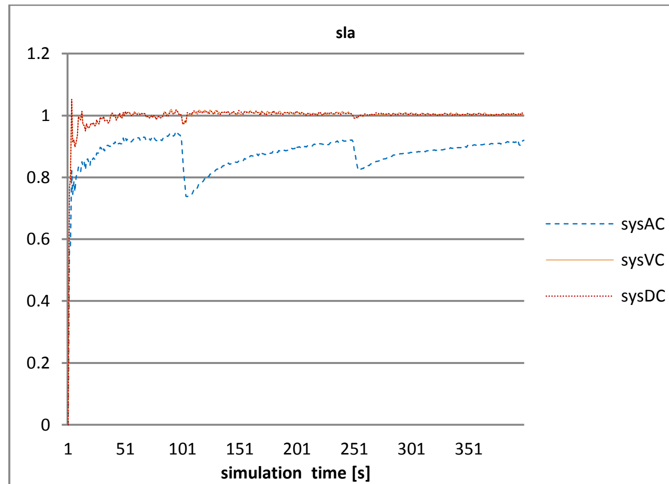


Figure 21: Service level achievement (SLA) analysis for scenario 4.

From the results of the four experimental scenarios presented above we can conclude that *sysAC* has a narrow envelope of operational conditions in which it is both self-managing and returns satisfactory behaviour. On the other hand, *sysVC* tends towards a wider operational envelope with increased efficiency and satisfactory behaviour, but once the limits of that envelope are reached the efficiency and reliability of the system drops. In moderate operational complexities *sysVC* performs adequately efficient but fluctuates rapidly and may need human input to override some decisions that lead to instability in the case of highly erratic and complex situation, which for example *sysDC* can deal with autonomically. Results have shown that *sysDC* is sufficiently sophisticated to operate efficiently and yield satisfactory results under almost all perceivable operating circumstances. So we can now confidently conclude that the proposed trustworthy autonomic architecture (represented by *sysDC*) has significant performance improvement over existing architectures and can be relied upon to operate (or manage) almost all level of datacentre scale and complexity.

Generally, the combination of DC and VC (VC + DC) leads to significant performance improvement over VC. However, the extent of this improvement is application and context dependent. Results show that there are circumstances in which performance improvement is evident from VC + DC as well as circumstances in which improvement is not evident. Complex applications with the possibility of unexpected behaviour patterns, e.g., large scale datacentres with complex algorithms, will usually experience improvement with VC + DC. Also, applications that are sensitive to fluctuating environmental inputs (i.e., depend on volatile environmental information for decision-making), for example, auto stock trading systems are expected to see greater benefit from VC + DC. On the other hand, there are applications that are not expected to see any benefit. Example includes small scale datacentres with predefined request rate and request capacity.

V. CONCLUSION

This paper has presented a new trustworthy autonomic architecture (TAArch). Different from the traditional autonomic solutions, TAArch consists of inbuilt mechanisms and instrumentation to support run-time self-validation and trustworthiness. The architecture guarantees self-monitoring over short time and longer time frames. At the core of the architecture are three components, the *AutonomicController*, *ValidationCheck* and *DependabilityCheck*, which allow developers to specify controls and processes to improve system trustability. We have presented a case example scenario to demonstrate the workings of the proposed approach. The empirical analysis case example scenario is an implementation of a datacentre resource request and allocation management designed to analyse the performance of the proposed TAArch architecture over existing autonomic architectures. Results show that TAArch is sufficiently sophisticated to operate efficiently and yield satisfactory results under almost all perceivable operating circumstances. Analyses also show that the proposed architecture achieves over 42% performance improvement (in terms of reliability) in a complex operating circumstance. It is also safe to conclude that the proposed trustworthy autonomic architecture has significant performance improvement over existing architectures and can be relied upon to operate (or manage) almost all level of datacentre scale and complexity.

The importance of trustworthiness in computing, in general, has been echoed in the Computing Research Association's 'four grand challenges in trustworthy computing' [31] and Microsoft's white paper on Trustworthy Computing (TC) [32]. The Committee on Information Systems Trustworthiness in [33] defines a trustworthy system as one which does what people expect it to do – and nothing more – despite any form of disruption. Although this definition has been the driving force for achieving trustworthiness both in autonomic and non-autonomic systems, the peculiarity of context dynamism in autonomic computing places unique and different challenges on trustworthiness for autonomic systems. Validation for example, which is an essential requirement for trustworthiness, can be design-time based for non-autonomic systems but must be run-time based for autonomic systems. Despite the different challenges, it is generally accepted that trustworthiness is a non-negotiable priority for computing systems. For autonomic systems, the primary concern is not how a system operates to achieve a result but how dependable is that result from the user's perspective. For complete reliance on autonomic computing systems, the human user will need a level of trust and confidence that these systems will satisfy specified requirements and will not fail. It is not sufficient that systems are performing within requirement boundaries, outputs must also be seen to be reliable and dependable. This is necessary for self-managing systems in order to mitigate the threat of losing control and confidence [34]. We posit that such capabilities need to be built in as integral part of the autonomic architecture and not treated as add-ons.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

The traditional MAPE-based autonomic architecture as originally presented in [2] has been widely accepted and autonomic research efforts are predominantly based on this architecture's control loop. We must admit that a good research success has been achieved using the MAPE-based architecture. However, we suppose, like others, e.g., [7][10], that this architecture is vague and thus cannot lead to the full goal of autonomic computing. For example, the MAPE-based architecture does not explicitly and integrally support run-time self-validation, which is a prerequisite for trustworthiness.

REFERENCES

[1] T. Eze, R. Anthony, C. Walshaw, and A. Soper, "A New Architecture for Trustworthy Autonomic Systems," in *Proceedings of the Fourth International Conference on Emerging Network Intelligence: (EMERGING) 2012*, pp. 62-68 Barcelona, Spain.

[2] IBM Autonomic Computing White Paper, An architectural blueprint for autonomic computing, 3rd edition, June 2005. Available via http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf last viewed 18th December 2013.

[3] M. Huebscher and J. McCann, "A survey of autonomic computing—degrees, models, and applications," *ACM Computer Survey (CSUR)*, Volume 40, Issue 3, August 2008, Article 7.

[4] C. Reich, K. Bubendorfer, and R. Buyya, "An autonomic peer-to-peer architecture for hosting stateful web services", in *Proceedings of the Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID),* pp. 250-257, 2008.

[5] P. de Grandis and G. Valetto, "Elicitation and utilization of utility functions for the self-assessment of autonomic applications," *in Proceedings of the sixth International Conference on Autonomic Computing (ICAC), 2009*, Barcelona, Spain.

[6] C. Dorn, D. Schall, and S. Dustdar, "A model and algorithm for self-adaptation in service-oriented systems," in *Proceedings of the seventh IEEE European Conference on Web Services* (ECOWS), 2009, pp. 161 – 170, Eindhoven, The Netherlands.

[7] H. Shuaib, R. Anthony, and M. Pelc, "A Framework for Certifying Autonomic Computing Systems", in *Proceedings of the Seventh International Conference on Autonomic and Autonomous Systems: (ICAS) 2011*, Venice, Italy.

[8] T. Eze, R. Anthony, C. Walshaw, and A. Soper, "Autonomic Computing in the First Decade: Trends and Direction," in *Proceedings of the Eighth International Conference on Autonomic and Autonomous Systems (ICAS)* 2012, St. Maarten, Netherlands Antilles.

[9] J. Kephart and D. Chess, "The Vision of Autonomic Computing," Computer, vol. 36, no. 1, pp. 41-50, 2003.

[10] A. Diniz, V. Torres, and C. José, "A Self-adaptive Process that Incorporates a Self-test Activity," *Monografias em Ciência da Computação*, number 32/09, 2009, Rio – Brazil.

[11] H. Chan, A. Segal, B. Arnold, and I. Whalley, "How Can We Trust an Autonomic System to Make the Best Decision?" in *Proceedings of the second International Conference on Autonomic Computing (ICAC), 2005*, Seattle, USA.

[12] J. Hall and L. Rapanotti, "Assurance-driven design in Problem Oriented Engineering," in *International Journal On Advances in Intelligent Systems (IntSys)*, volume 2, number 1, 2009, pp. 26-37.

[13] S. Kikuchi, S. Tsuchiya, M. Adachi, and T. Katsuyama, "Constraint Verification for Concurrent System Management Workflows Sharing Resources," in *Proceedings of the third International Conference on Autonomic and Autonomous Systems (ICAS), 2007*, Athens, Greece.

[14] X. Li, H. Kang, P. Harrington, and J. Thomas, "Autonomic and trusted computing paradigms," in *Lecture Notes in Computer Science*, Volume 4158, 2006, pp. 143-152.

[15] S. Anderson, M. Hartswood, R. Procter, M. Rouncefield, R. Slack, J. Soutter, and A. Voss, "Making Autonomic Computing Systems Accountable," in *Proceedings of the fourteenth International Workshop on Database and Expert Systems Applications (DEXA), 2003*

[16] R. Anthony, "Policy-based autonomic computing with integral support for self-stabilisation," *International Journal of Autonomic Computing*, Vol. 1, No. 1, 2009, pp. 1–33.

[17] R. Anthony, "Policy-centric Integration and Dynamic Composition of Autonomic Computing Techniques," in *Proceedings of the fourth International Conference on Autonomic Computing (ICAC), 2007*, Florida, USA.

[18] J. Heo and T. Abdelzaher, "AdaptGuard: Guarding Adaptive Systems from Instability," in *Proceedings of the sixth International Conference on Autonomic Computing (ICAC), 2009*, Barcelona, Spain.

[19] J. Hawthorne, R. Anthony, and M. Petridis, "Improving the Development Process for Teleo-Reactive Programming Through Advanced Composition," in *Proceedings of the Seventh International Conference on Autonomic and Autonomous Systems (ICAS) 2011*, Venice, Italy.

[20] D. Richards, M. Taylor, and P. Busch, "Expertise Recommendation: A triangulated approach," in *International Journal On Advances in Intelligent Systems (IntSys)*, volume 2, number 1, 2009, pp. 12-25.

[21] T. King, A. Ramirez, R. Cruz, and P. Clarke, "An Integrated Self-Testing Framework for Autonomic Computing Systems," *Journal of computers*, vol. 2, no. 9, 2007, pp. 37-49.

[22] T. Eze, R. Anthony, C Walshaw, and A. Soper, "A Methodology for Evaluating Complex Interactions between Multiple Autonomic Managers", in *Proceedings of the Ninth International Conference on Autonomic and Autonomous Systems* (ICAS), 2013, Lisbon, Portugal.

[23] H. Li and S. Venugopal, "Using Reinforcement Learning for Controlling an Elastic Web Application Hosting Platform," in *Proceedings of the eighth International Conference on Autonomic Computing (ICAC)*, *2011*, Karlsruhe, Germany.

[24] T. Yu, R. Lai, M. Lin and B. Kao, "A Fuzzy Constraint-Directed Autonomous Learning to Support Agent Negotiation," in *Proceedings of the third International Conference on Autonomic and Autonomous Systems (ICAS), 2007*, Athens, Greece.

[25] R. Das, J. Kephart, J. Lenchner, and H. Hamann, "Utility-function-driven energy-efficient cooling in data centers," *in Proceeding of the Seventh International Conference on Autonomic Computing (ICAC)*, 2010, New York, USA.

[26] I. Goiri, J. Fit´o, F. Juli`a, R. Nou, J. Berral, J. Guitart and J. Torres, "Multifaceted Resource Management for Dealing with Heterogeneous Workloads in Virtualized Data Centers*," in Proceedings of Eleventh IEEE/ACM*

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

International Conference on Grid Computing (GRID), 2010, Brussels, Belgium.

[27] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle, “Managing Energy and Server Resources in Hosting Centers,” ACM SIGOPS Operating Systems Review, vol. 35, no. 5, pp. 103–116, 2001.

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

[29] M. Pretorius, M. Ghassemian, and C. Ierotheou, “An investigation into energy efficiency of data centre virtualisation,” *in Proceedings of International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, 2010, Fukuoka, Japan.

[30] M. Pretorius, M. Ghassemian, and C. Ierotheou, “Virtualisation –Securing a Greener Tomorrow with Yesteryear’s Technology,” *in Proceeding of the Twelfth IFIP/IEEE International Symposium on Integrated Network Management* (IM 2011), 2011, Dublin, Ireland.

[31] Computing Research Association, “Four Grand Challenges in Trustworthy Computing,” in *Proceedings of Second Conferences on Grand Research Challenges in Computer Science and Engineering*, November 16–19, 2003.

[32] C. Mundie, P. Vries, P. Haynes, and M. Corwine, “Trustworthy Computing: Microsoft White Paper,” *Microsoft Corporation*, October 2002.

[33] F. Schneider, “Trust in Cyberspace,” *Committee on Information Systems Trustworthiness, National Academy Press*, 1998, Washington, D.C.

[34] L. Yang and J. Ma, Introduction to the Journal of Autonomic and Trusted Computing. American Scientific Publishers www.aspbs.com/joatc.html 26/08/13 last viewed 28/05/2014

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

APPENDIX A: TAArch Application

The simulations of this paper are performed using the TAArch Application. To understand the workings of the application let us consider Figure A, which is a screen shot of a basic resource allocation simulation with 75 servers (x) and 4 applications (ix). The user selects the number of servers and applications and this will populate the S_i and A_j pools respectively (labels x and ix). The application supports two experiments ('Normal Simulation' and 'Interoperability', which is not covered here) as shown (iii) and in this case the 'Normal Simulation' option is selected, which will automatically check the *PeM* autonomic manager option (vi). Then the actual manager is selected, which in this case is the [AC+VC+DC] option representing all three managers. As shown (vii) the DZWidth can be manually controlled by the user or dynamically tuned by the system depending on which option is selected. Before the simulation starts it is possible to set the internal variables through (xiv) to user preferences. The possibility of changing the internal variables is deactivated (as shown by xiv) as soon as the simulation starts. Change of server capacity is also deactivated (i) as soon as simulation starts. Meanwhile, application size (i), which is an external variable, can be changed at any time in the simulation. Once all parameters are set the simulation can be started by clicking 'Run Simulation'. For the purpose of this example the shutdown server pool $\check{S}_i$ is not used (xi) –it is only used for the 'Interoperability' simulation.

Once the simulation starts, the manager starts populating the U pool (xiii). The view of this pool shows current and live updates of process status. 'Available capacity' shows running capacity available to serve individual application request while 'Run'g requests' are the total running individual request capacity. 'Offset' is the difference between running request capacity and available capacity. 'Server_ID' shows the collection of servers currently providing services for individual application request. Depending on the number of servers in use, some of the allocated servers may no longer be visible in the U pool but can be viewed from the respective individual pool (xii). The provisioning servers, that is, servers that are been configured in the queue can be viewed through (ii). Individual results for the managers are displayed in (iv) and (v). Also as stated, data displayed below (viii) and in (ii) are for AC (*sysAC*). Although there is provision for live graphing of results through the 'Show Graph' button, complete result values can be exported to Excel Sheet through the 'Export Results' button (vii).

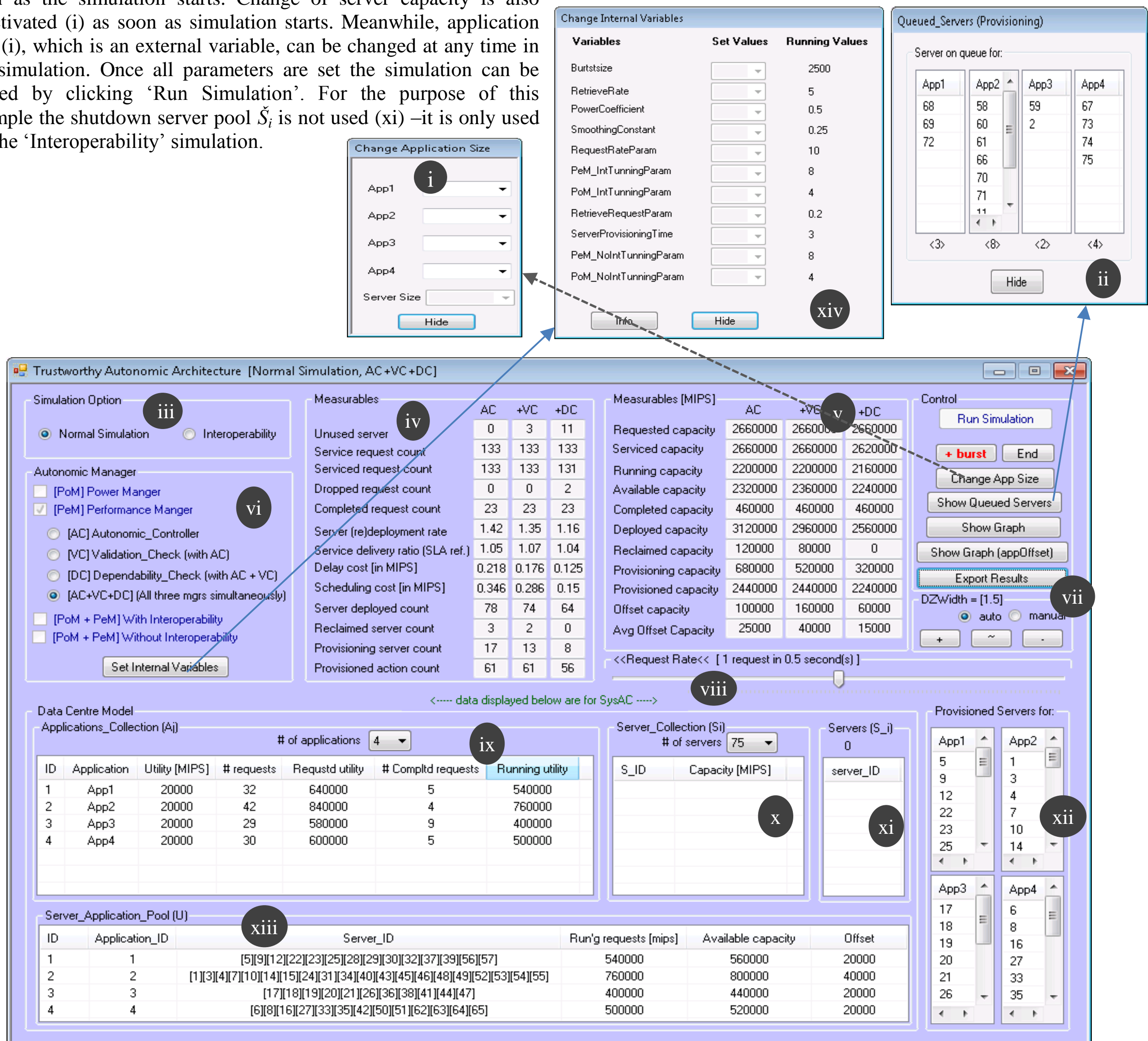


Figure A: Simulation screen shot showing TAArch application front end.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org