ICAS 2012

The Eighth International Conference on Autonomic and Autonomous Systems

ISBN: 978-1-61208-187-8

March 25-20, 2012

St. Maarten, Netherlands Antilles

ICAS 2012 Editors

Freimut Bodendorf, University of Erlangen, Germany

Wendy Powley, Queen's University - Kingston, Canada

ICAS 2012

Foreword

The Eighth International Conference on Autonomic and Autonomous Systems (ICAS 2012), held between March 25-30, 2012 - St. Maarten, Netherlands Antilles, was a multi-track event covering related topics on theory and practice on systems automation, autonomous systems and autonomic computing.

The main tracks referred to the general concepts of systems automation, and methodologies and techniques for designing, implementing and deploying autonomous systems. Next tracks developed around design and deployment of context-aware networks, services and applications, and the design and management of self-behavioral networks and services. Also considered were monitoring, control, and management of autonomous self-aware and context-aware systems and topics dedicated to specific autonomous entities, namely, satellite systems, nomadic code systems, mobile networks, and robots. It has been recognized that modeling (in all forms this activity is known) is the fundamental for autonomous subsystems, as both managed and management entities must communicate and understand each other. Small-scale and large-scale virtualization and model-driven architecture, as well as management challenges in such architectures were considered. Autonomic features and autonomy requires a fundamental theory behind and solid control mechanisms. These topics give credit to specific advanced practical and theoretical aspects that allow subsystem to expose complex behavior. It was aimed to expose specific advancements on theory and tool in supporting advanced autonomous systems. Domain case studies (policy, mobility, survivability, privacy, etc.) and specific technology (wireless, wireline, optical, e-commerce, banking, etc.) case studies were targeted. A special track on mobile environments was indented to cover examples and aspects from mobile systems, networks, codes, and robotics.

Pervasive services and mobile computing are emerging as the next computing paradigm in which infrastructure and services are seamlessly available anywhere, anytime, and in any format. This move to a mobile and pervasive environment raises new opportunities and demands on the underlying systems. In particular, they need to be adaptive, self-adaptive, and context-aware.

Adaptive and self-management context-aware systems are difficult to create, they must be able to understand context information and dynamically change their behavior at runtime according to the context. Context information can include the user location, his preferences, his activities, the environmental conditions and the availability of computing and communication resources. Dynamic reconfiguration of the context aware systems can generate inconsistencies as well as integrity problems, and combinatorial explosion of possible variants of these systems with a high degree of variability can introduce great complexity.

Traditionally, user interface design is a knowledge-intensive task complying with specific domains, yet being user friendly. Besides operational requirements, design recommendations refer to standards of the application domain or corporate guidelines.

Commonly there is a set of general user interface guidelines; the challenge is due to a need for cross-team expertise. Required knowledge differs from one application domain to another, and the core knowledge is subject to constant changes and to individual perception and skills.

Passive approaches allow designers to initiate the search for information in a knowledge-database to make accessible the design information for designers during the design process. Active approaches, e.g., constraints and critics, have been also developed and tested. These mechanisms deliver information (critics) or restrict the design space (constraints) actively, according to the rules and guidelines. Active and passive approaches are usually combined to capture a useful user interface design.

All these points posed considerable technical challenges and make self-adaptable context-aware systems costly to implement. These technical challenges led the context-aware system developers to use improved and new concepts for specifying and modeling these systems to ensure quality and to reduce the development effort and costs.

SYSAT Advances in system automation
AUTSY Theory and Practice of Autonomous Systems
AWARE Design and Deployment of Context-awareness Networks, Services and Applications
AUTONOMIC Autonomic Computing: Design and Management of Self-behavioral Networks and Services
CLOUD Cloud computing and Virtualization
MCMAC Monitoring, Control, and Management of Autonomous Self-aware
CASES Automation in specialized mobile environments
ALCOC Algorithms and theory for control and computation
MODEL Modeling, virtualization, any-on-demand, MDA, SOA
SELF Self-adaptability and self-management of context-aware systems
KUI Knowledge-based user interface
AMMO Adaptive management and mobility

We welcomed technical papers presenting research and practical results, position papers addressing the pros and cons of specific proposals, such as those being discussed in the standard forums or in industry consortia, survey papers addressing the key problems and solutions on any of the above topics short papers on work in progress, and panel proposals.

We take here the opportunity to warmly thank all the members of the ICAS 2012 technical program committee as well as the numerous reviewers. The creation of such a broad and high quality conference program would not have been possible without their involvement. We also kindly thank all the authors that dedicated much of their time and efforts to contribute to ICAS 2012. We truly believe that, thanks to all these efforts, the final conference program consisted of top quality contributions.

We hope that ICAS 2012 was a successful international forum for the exchange of ideas and results between academia and industry and to promote further progress in autonomic and autonomous systems.

We are certain that the participants found the event useful and communications very open. The beautiful places of St. Maarten surely provided a pleasant environment during the conference and we hope you had a chance to visit the surroundings.

ICAS 2012 Chairs
Michael Bauer, The University of Western Ontario - London, Canada
Radu Calinescu, Aston University, UK
Michael Grottke, University of Erlangen-Nuremberg, Germany
Bruno Dillenseger, Orange Labs, France

ICAS 2012

Committee

ICAS Advisory Chairs

Michael Bauer, The University of Western Ontario - London, Canada
Radu Calinescu, Aston University, UK
Michael Grottke, University of Erlangen-Nuremberg, Germany
Bruno Dillenseger, Orange Labs, France

ICAS 2012 Technical Program Committee

Jemal H. Abawajy, Deakin University, Australia
Nouara Achour, USTHB University, Algeria
Carl Adams, University of Portsmouth, UK
Jérémie Albert, University of Bordeaux, France
Javier Alonso, Technical University of Catalonia, Spain
Razvan Andonie, Central Washington University - Ellensburg, USA
Richard Anthony, University of Greenwich, UK
Eva Ibarrola Armendariz, Escuela Técnica Superior de Ingeniería de Bilbao, Spain
Ismailcem Budak Arpinar, University of Georgia - Athens, USA
Mark J. Balas, University of Wyoming - Laramie, USA
Michael Bauer, The University of Western Ontario -London, Canada
Julita Bermejo-Alonso, Universidad Politécnica de Madrid, Spain
Philippe Besnard, IRIT - CNRS /Universite Paul Sabatier - Toulouse, France
Ateet Bhalla, NRI Institute of Information Science and Technology - Bhopal, India
Karsten Böhm, Fachhochschule Kufstein, Austria
Fabienne Boyer, University of Grenoble I, France
David W Bustard, University of Ulster, UK
Radu Calinescu, Aston University, UK
Paolo Campegiani, University of Roma Tor Vergata, Italy
Sara Casolari, Università di Modena e Reggio Emilia, Italy
Fernando Cerdan, Universidad Politecnica de Cartagena, Spain
Michal Certicky, Comenius University - Bratislava, Slovakia
Carlos Cetina, Technical Universidad San Jorge, Spain
Lei Chen, Sam Houston State University, USA
Philippe Codognet, University of Tokyo, Japan
Lorcan Coyle, Lero - University of Limerick, Ireland
Felix Cuadrado Latasa, University Polytechnic of Madrid, Spain
Noel De Palma, INRIA/SARDES - Grenoble, France
Marina De Vos, University of Bath, UK
Sotirios Diamantas, Pusan National University, South Korea
Tadashi Dohi, Hiroshima University, Japan
Carlos Duarte, University of Lisbon, Portugal
Xavier Dutreilh, Université Pierre et Marie Curie, France
Larbi Esmahi, Athabasca University, Canada

Table of Contents

Comparison of Bio-Inspired and Graph-Theoretic Algorithms for Design of Fault-Tolerant Networks

Matthias Becker, Waraphan Sarasureeporn and Helena Szczerbicka
FG Simulation and Modeling
Leibniz University Hannover
Welfengarten 1, 30167 Hannover, Germany
{xmb,sarasureeporn,hsz}@sim.uni-hannover.de

***Abstract*—Recently several approaches have been presented that exploit the ability of *Physarum polycephalum* to connect several food sources via a network of pipes in order to maintain an efficient food distribution inside the organism. These approaches use the mechanisms found in nature in order to solve a technical problem, namely the design of constructing fault-tolerant and efficient connection networks. These works comprise experiments with a real slime mold *Physarum polycephalum* as well as computer simulations based on a tubular model and an agent-based approach. In this work, we study the suitability of those bio-inspired approaches and compare their performance to a graph-theoretic algorithm for construction of fault-tolerant connection networks, the (k,t)-spanner algorithm. The graph-theoretic algorithm is able to construct graphs with a certain degree of fault tolerance as well as meet a given maximal path length between two arbitrary nodes. However the definition of fault tolerance in previous bio-inspired works differs to that used in graph theory. Thus in our contribution we analyze the bio-inspired approaches as well as the graph-theoretic approach for their efficiency of designing optimal fault-tolerant graphs. We demonstrate the usability of the graph-theoretic approach despite relying on a different definition of fault tolerance. We conclude that classical efficient computational algorithms from graph theory can be adapted and applied in the same field as the bio-inspired approaches for the problem of constructing efficient fault tolerant networks. They often provide an easier to use and more direct solution than bio-inspired approaches, that need more parameter tuning before getting satisfactory results.**

***Index Terms*—slime mold, *Physarum polycephalum*, fault tolerant network, (k,t)-spanner**

I. MOTIVATION

Recently bio-inspired computing based on slime molds raised attention in scientific renowned journals [1]–[3] as well as in popular newspapers for the slime molds' ability to solve complex problems, despite being a brainless primitive lifeform [4].

Research groups use a real slime mold or different types of simulations of slime mold behavior for building networks or finding a short path through a maze. One group [1] conducted experiments with a real slime mold *Physarum polycephalum* and computer simulations based on a tube model in order to construct a fault tolerant and efficient transport network for the Tokyo rail system. The natural slime mold as well as the simulated slime mold generate networks that are similar to the existing rail system of Tokyo. While the quality of the solutions of the real slime mold shows considerable variations, the networks constructed by tubular simulations show a very regular structure in their quality that is correlated with one parameter. In another approach [5] an agent based simulation of *Physarum polycephalum* turned out to better approximate the characteristics of the real slime mold's networks.

However, existing classical algorithms for those problems have not been included in the evaluation of bio-inspired approaches in previous works.

Thus in this work we study, whether natural or simulated *Physarum polycephalum* is an efficient means for construction of fault tolerant networks at all, i.e., can networks of good quality be generated with reasonable computational effort. Although *Physarum polycephalum* simulations have also been used in the past for obtaining fault tolerant networks, the quality of such networks has only been compared to existing networks that have been historically grown and not been constructed efficiently from scratch. We demonstrate how algorithms from graph theory can be used for the design of fault-tolerant efficient networks, despite using different definitions of fault tolerance. We show that in many scenarios the graph-theoretic algorithm is a viable means to efficiently obtain reliable results without having the additional effort of parameter tuning which often is a serious disadvantage of bio-inspired algorithms.

In the following, we will describe the state of the art concerning simulation models of *Physarum polycephalum* that can be used for the construction of fault tolerant connection networks. We use the Tokyo railway network, which is the mostly used reference network in this context. We also describe the agent-based simulation model and the (k,t)-spanner algorithm from graph theory. A presentation and discussion of the resulting networks for all approaches concludes this work.

II. PHYSARUM POLYCEPHALUM

There are basically two kinds of slime molds (cellular and acellular) which are member of a category of eukaryotic organisms that typically have some fungal-like attributes and some animal-like attributes. In this work, we are interested in *Physarum polycephalum*, a slime mold visible to the unaided eye.

A. Biological Foundations

Physarum polycephalum is a yellow single-celled slime mold whose plasmodium is visible for the unaided human eye and can grow up to one square meter if the environmental conditions are ideal. Otherwise it usually has the size of a palm. Starting with spores of *Physarum polycephalum* mysamben with a single nucleus will be produced, which can reproduce by mitosis. Dependent on environmental conditions flagellates can evolve. If two flagellates of different sex meet they form a diploid zygote. This will grow to the final size plasmodium by division of the nucleus.

Summarizing, the large visible yellow slime mold is a large single cell with multiple nuclei. When food is used up in the area of the cell it enters the hunger phase. In this phase *Physarum polycephalum* optimizes its shape by maintaining thick pipes between food sources and by shrinking the contour where no food is available any more. This is the phase that is used by most computational slime mold inspired algorithms.

B. *Computational Applications of* Physarum polycephalum

New research showed that this kind of slime mold is able to construct efficient and fault tolerant connection networks between multiple food sources. This ability has been used (with real slime molds and simulations of *Physarum polycephalum*) on examples such as British and American motorways [6], [7] and for the Tokyo railway network [1]. *Physarum polycephalum* constructed networks that had similar properties as the existing networks designed by human engineers. Another application is the usage of *Physarum polycephalum* as light detector of a robot [8] and in wireless ad hoc networks [9].

C. *Simulation Models for* Physarum polycephalum

In the literature, several types of computer simulation models can be found.

In [10], the hunger-phase of *Physarum polycephalum* is modeled by a mesh network of tubes that can enlarge or shrink. This model is close to the natural mechanisms, where nutrition is streamed through the slime mold, so that nutrition is spread throughout the whole cell, from food sources to areas with less food. During that process the streaming channels of the slime mold enlarge, where more nutrition has to be moved, and channels shrink or disappear where little nutrition is present or is to be transported. The simulation model includes differential equations of the pressure and the movement of the fluid with time, and the changing of the size of pipes dependent on the moving fluid.

In [11], an agent based model is used which basically is a cellular automaton. Each place in the two dimensional matrix can be visited by an agent. An agent has three sensors in its front view, front right, front left and front middle. Parameters are sensor angle and sensor range. Two actions are possible: move to another cell and/or leave a trace. This approach models the distribution of the nutrition inside the *Physarum polycephalum* cell in a more abstract way, physically and quantitatively not very close to the natural mechanisms. However the phenomena of building fault tolerant short networks is captured by this model very well. The agents can be interpreted as moving nutrition that is not explicitly channeled. However channel-like streams will build up implicitly by the rules governing the agents' behavior. Furthermore the agent based approach allows fast simulations/calculations and it is not necessary to deal with costly calculation and solution of differential equations.

The approach in [12] models the expansion and shrinkage phase similarly to dilation and erosion as know from computer graphics. This algorithm is especially designed for path finding in a maze.

Subsequently, we will show how to obtain connection networks using the agent-based simulation of *Physarum polycephalum*. The resulting networks (see also [5]) will be presented and their quality will be compared to that of the networks obtained by tubular simulations and with experiments done with real *Physarum polycephalum* (see [1]). Then, it will be demonstrated how algorithms from graph theory [13] can be adapted for application in our context.

III. Network Design by Agent Based Simulation of *Physarum polycephalum*

We used our simulator, that is based on the model in [11], in order to construct a number of different fault tolerant connection networks and compared the characteristics of the found solutions with the existing real railway network and with a manually constructed fault tolerant graph (by human expertise, resulting from enhancing the Minimum Spanning Tree (MST) where fault tolerance has been introduced by manually adding some edges).

Our simulator is based on a cellular automaton. The playground is a two dimensional matrix, on which agents are placed, that observe their environment and that can move around and/or leave a chemical, dependent on the environmental conditions. The chemical is steering the movement of agents. The chemical is emitted by food sources and spreads spatially, by the same time vanishing through evaporation. Agents can reinforce this signal by emitting the chemical themselves. The value of the chemicals concentration is stored as attribute of each cell of the matrix.

The main rules of the agent's behavior according to [11] (slightly different rules can be found in [14], [15]) are:

Movement:

```
Step 1: 'Attempt to move forward in current direction'
Step 2:
.....IF ('move forward successful')
..........THEN 'Deposit trail in new location'
..........ELSE 'Choose random new orientation'
```

Given that each agent faces into a certain direction and can sense the matrix for the concentration of the chemical in front direction (F), front right (FL) and front left (FL), the agent maintains its direction, rotates a certain angle (RA) to the left or right, or randomly:

```
Step 1: 'Sample trail map sensor values F, FL, FR'
```

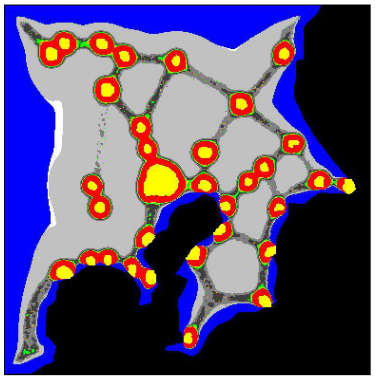


Fig. 1. Screenshot of simulation

```
Step 2:
IF (F > FL) AND (F > FR)
.....'Stay facing same direction'
ELSEIF (F < FL) AND (F < FR)
.....'Rotate randomly left or right by RA'
ELSEIF (FL < FR)
.....'Rotate right by RA'
ELSEIF (FR < FL)
.....'Rotate left by RA'
ELSE
.....'Continue facing same direction'
```

The simulation starts with a matrix which can be represented as bitmap where the different colors stand for the various states of a cell (cf. Fig. 1): The background representing empty cells is white. A food source is yellow. Yellow food sources represent the nodes of a graph (cities) which have to be connected by the slime mold (connections representing railways). Black and blue is forbidden terrain, red to gray is the intensity of the trail signal.

In nature, food is the attractor for *Physarum polycephalum* and where there is more food there *Physarum polycephalum* moves to. In nature, food flows through *Physarum polycephalum* and influences the movement and shape of *Physarum polycephalum*. In the agent model, the food flow is modeled by a trace that agents leave on their trail, the trace signaling that food is near, or that many agents are heading in this direction, probably because someone found food there. This behavior is similar to ant algorithms, where ants also emit pheromones on their path [16].

The optimization problem to be solved when constructing a fault tolerant graph is to find a solution between the Minimum Spanning Tree (minimum cost, that is sum of length of edges between nodes, but no fault tolerance) and a fully connected graph (maximal fault tolerance but also maximal cost). The slime molds tries to connect all food sources thereby shortening all of its connections as much as it can. Fault tolerance is not a direct goal of the slime mold, fault tolerance is a side effect since the slime mold just connects nodes that are near each other so that automatically several paths are established when a higher number of nodes are crowded in an area. Only isolated nodes will not be connected in a fault tolerant way. Note that the graph constructed by *Physarum polycephalum* will not only consist of direct links between nodes but may also have Steiner tree like connections. As can be observed from the picture in Fig. 1, the agents/chemicals not necessarily 'draw' an easily automatically measurable line between food sources. Thus the resulting graph has to be determined manually by drawing an edge where are 'enough' agents between two food sources. The graph constructed that way can then be analyzed for its quality measures. Because there is not always a state of clear convergence a stopping criteria has to be defined, i.e., when the simulation has to be stopped and the graph has to be defined and analyzed. We did measurements in two ways to account for the dynamics of the simulation: First experiments stopped after a fixed number of iterations (5000), the resulting network was measured then. In the second row of experiments, the simulation was stopped periodically (every 250 iterations), the graph was measured and analyzed, and then the simulation has been continued.

Both experiments were repeated with an additionally activated 3x3 filter, that smoothens the chemical values around each agent by averaging the values, which makes four different experimental setups which will be shown and discussed in Fig. 3 in the next section.

Before presenting the results, the measures describing the quality of the found graph will be introduced. In order to make this comparable to recent results the definition of the quality measures are taken from [1]:

- Graph G
- S: Set of nodes of G
- E: Set of edges e of G
- length(e): weight of edge e representing the distance of two nodes
- MST(G): Minimum Spanning Tree of G
- N: Number of nodes of G
- $SP(v_i, v_j)$: Shortest Path from node i to node j

The evaluation of the found networks is analogous to [1], where the definitions for the following measures of the graph are taken from:

Total length of all connections:

$$\text{TL}(G) = \sum_{e \in E} \text{length}(e)$$

Cost of the network relative to the minimum cost (of the MST):

$$\text{Cost}(G) = \frac{\text{TL}(G)}{\text{TL}(MST(G))}$$

Fault Tolerance FT that takes into account the length of edges, since a long connection is more prone to fault:

$$\text{FT}(G) = \frac{\sum_{e \in E | (G \setminus \{e\}) \text{ isConnected}} \text{length(e)}}{\text{TL}(G)}$$

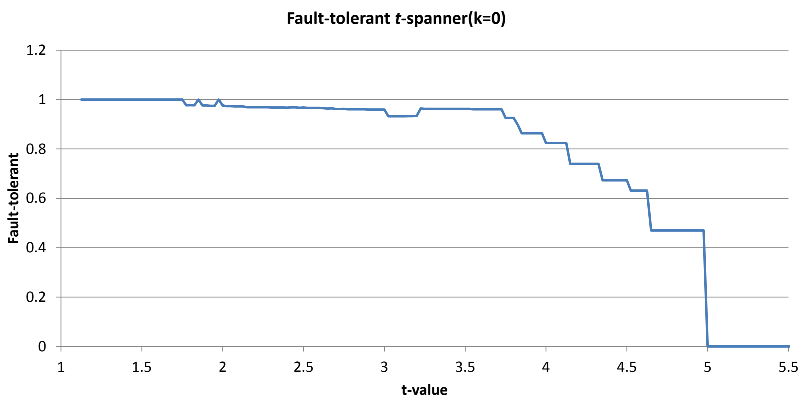


Fig. 2. Fault tolerance of spanner algorithm with varying value of t

The performance of the network as average distance between two arbitrary nodes relative to MST:

$$\text{Performance(G)} = \frac{\text{avgDistance}(G)}{\text{avgDistance}(MST)}$$

IV. Geometrical Spanners

In this section we explain the algorithm from graph theory, that has as input a set of points, and delivers a graph with a certain degree of fault tolerance and a maximal path length between each pair of nodes as result.

For a given set of nodes, a spanner is a graph that connects all nodes. In this context, the notion of fault tolerance is defined independently of the length of edges: a graph is said to be k-fault tolerant when k arbitrary edges can be removed and the graph still remains fully connected.

Obviously, the Minimum Spanning Tree is a spanner with zero fault tolerance.

A. Definition and Construction of a (k, t)-Spanner

Taken from [17], the necessary definitions and the algorithm for construction of a (k, t) spanner, are presented in this section. Note that it has also be proven that vertex and edge fault tolerance are corresponding concepts.

Let S be the set of N points in $\mathbf{R}^2$ as defined above. Let $t > 1$ be a real number, let $k \geq 0$ be an integer, and let $G = (S, E)$ be an undirected Euclidean graph with vertex set S. The spanner with stretch factor t is called t-spanner, if $\delta(p, q) < t|pq|$ for any two points of S. The notation $|pq|$ to denote the Euclidean distance and $\delta(p, q)$ to denote the shortest Euclidean length of a path in a geometric graph G between p and q. The approach is a greedy algorithm. Each edge of the complete graph is considered for construction of the spanner. The edges are processed in increasing order of the edge length, for that purpose they are stored sorted in list L. For each edge the intermediate result graph will be checked, whether already the fault tolerant criterion between the two points connected by that edge is fulfilled. If not, the edge is added to the graph. If yes, but the t criterion is not fulfilled (i.e. the existing paths are too long) then the edge is also added. Formally, that is, if the graph G does not contains $k + 1$ vertex-disjoint t-spanner paths between p and q, the edge (p, q) will be included in the set of edges E. The output of algorithm is a t-spanner for S with k fault tolerance.

Altogether, the algorithm can be formalized as shown in the following algorithm 1 (excerpted from [17]).

Algorithm 1 (k, t)-spanner algorithm

```
Input: A set $S$ of $N$ points in $\mathbf{R}^2$, an integer $k \geq 0$,
        and $a$ real number $t > 1$
Output: A $(k, t)$-spanner for $S$
Initialisation: $G := (S, E)$ with $E := \emptyset$

for each $\{p, q\} \epsilon L$ considered pairs in nondecreasing order
do if $G$ does not contain $k + 1$ vertex-disjoint $t$-spanner
        paths between $p$ and $q$
    then $E := E \cup \{\{p, q\}\}$
        $G := (S, E)$
    endif
endfor
```

Note that there are different definitions of fault tolerance. In the following we will use the definition that includes the length of edges. This has two reasons; first, the networks produced by bio-inspired algorithms nearly never show fault tolerance greater than one. Second, for application in real networks, the definition including the lengths of edges is more realistic, since

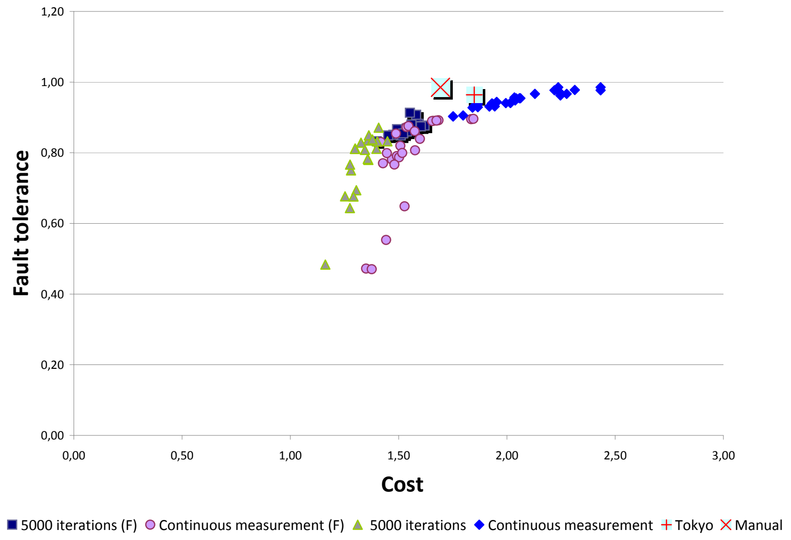


Fig. 3. Agent-based slime mold simulation results for the trade-off fault tolerance vs cost (both normalized to MST)

the fault probability of longer connections usually is higher in reality, be it communication lines or transport networks.

In Fig. 2, the characteristics of the generated networks can be observed for changing values of t (for $k = 0$). For values of t approaching one, the generated networks converge towards the complete graph. Since for large values of t the resulting networks converge towards the Minimum Spanning Tree, the fault tolerance of the resulting spanners initially is one, and falls down on a convex curve towards zero. We conclude that by variation of parameter t a wide range of networks can be generated. In the next section, we will analyse these networks for their properties concerning length and fault tolerance and compare them to those generated by the slime mold simulation.

V. Results

A. Fault Tolerance versus Costs

Fig. 3 presents the results obtained by agent-based simulations of *Physarum polycephalum*, showing the degree of fault tolerance versus the costs of each produced network. The diversity of networks has its origin in the stochastic nature of the agent-based simulation. Additionally, the real Tokyo network is included, as well as a network which has been designed manually, taking the MST and adding several connections manually, where obviously necessary. As first observation, it can clearly be seen that the fault tolerance of the Tokyo network as well as the manually constructed network is better than the networks produced by the slime mold algorithm. The Tokyo network is a bit less fault tolerant and has marginally less costs than the manually constructed network.

This is not surprising since the Tokyo network is grown historically and has to cope with geographical, political and other constraints that are not represented in the graph.

Since fault tolerance is not a primary goal of the slime mold algorithm the results for performance do not reach top values. However, a number of solutions show a good trade off: Especially the simulation results after 5000 iterations without filter reach values for fault tolerance around 0.84 while having costs of only 1.4 (the Tokyo and manually constructed networks have fault tolerance of around 0.98 and costs of around 1.75).

Fig. 3 directly relates to Fig. 3 (A) in [1]. For reference, the results for the natural slime mold and the tubular simulations from [1] are given in Fig. 4 here. It can be seen that the graph for the tubular simulation results starts with zero fault tolerance and cost of one (obviously it found the MST), raising quickly in a straight line to a fault tolerance of approximately 0.9 at normalized cost of 1.5 and from there on converging to one, at raising cost. Contrary to the tubular simulation, the results for the natural *Physarum polycephalum* barely find a non-fault tolerant network, nearly all networks having fault tolerance over 0.6. The convergence to fault tolerant networks at high costs cannot be observed, several networks with costs

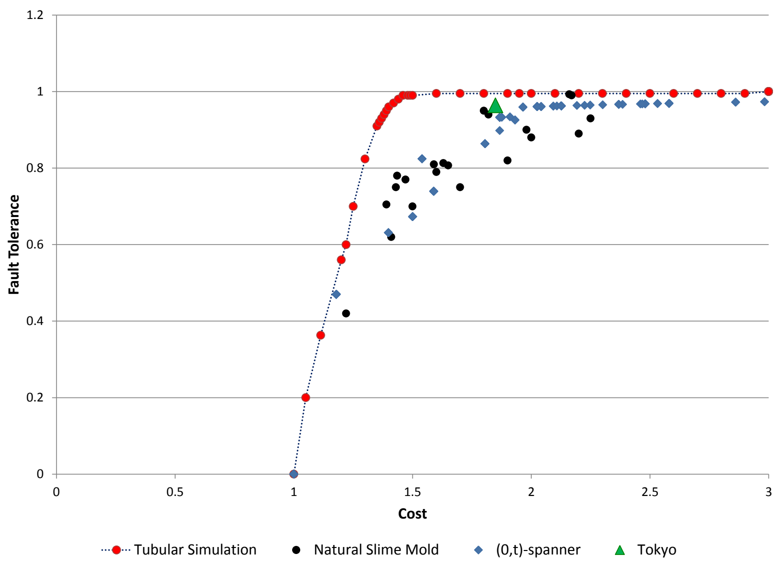


Fig. 4. Fault tolerance versus costs for tubular model, (k,t)-spanner and natural slime mold

of around 1.5 have a fault tolerance of less than 0.8.

The agent-based simulation results reproduce much better the variety of the results obtained by the natural slime mold as can be seen by comparing Fig. 3 and Fig. 4. Taking the results for 5000 iterations without filter and continuous measurements with filter together, the results resemble the natural results much more in their variance, and also not showing less fault tolerance than 0.4. The results for continuous measurement without filter seem to be similar to the tube model, the trend to convergence for fault tolerant networks at higher cost can be clearly observed.

Fig. 4 also contains the results for the (k,t)-spanner. The data points for the spanner algorithm show a good accordance to the naturally produced variety of networks regarding non-extremal values for t. For the extremal values of t the results of course converge against the complete graph and MST, similarly as for the tubular model.

Contrary to the tubular model, the spanner algorithm reproduces the variety of naturally produced networks much better for non-extremal values for t, resulting in networks with costs in the interval of 1.25 and 2, and fault tolerance between 0.5 and 0.95.

Altogether, it can be stated that the (k,t)-spanner algorithm is applicable despite the different definition of fault tolerance and that it is well capable of producing networks whose characteristics show the same variety as networks constructed by real slime molds. This is achieved by changing the value of parameter t.

VI. Discussion and Conclusion

The results in the last section showed that slime mold inspired construction of fault tolerant optimized connection networks is a working approach.

However, it is the question, whether dedicated classical algorithms might be a better choice for this task.

Algorithms for that purpose belong to the class of algorithms for geometrical spanners [13], [17]. It has been shown in this paper that by varying the parameter t in the (k,t)-spanner algorithm, networks with the desired trade-off between fault tolerance and performance can be generated. It has also been shown how to overcome the different definitions of fault tolerance, used in graph theory and in the bio-inspired approaches.

Thus, the (k,t)-spanner algorithm can well be used in order to construct a connection network tailored to the needs, be it fault tolerance, length, etc.

For the Tokyo network, the spanner algorithm worked quite satisfactory and faster than the bio-inspired approaches. However the complexity is $O(|L| \cdot \log|L|)$ stemming from the sorting of $|L|$ possible edges. Note that the problem size should be characterized by the number of points N to be connected, resulting in $\binom{N}{2}$ possible edges. For future work it should be investigated, for which N the (k,t)-spanner algorithm is not applicable anymore and whether the bio-inspired approaches still work for that problem complexity. In order to do this an approach is needed to automatically construct a graph out of the simulation results, since manually marking the result graph in the image of the final agent distribution is not a viable approach for larger problems.

It also showed that the approach of using human intuition, i.e., constructing a Minimum Spanning Tree and adding manually some edges for fault tolerance, is a cheap solution that leads to good results. However, this approach is not viable for larger graphs.

Summarizing, it can be said that naturally inspired algorithms is of course an interesting field that also lead to valuable new algorithmic approaches. However, not everything that works in nature can be transferred to technical solutions easily. As we show in this work, slime mold inspired algorithms are capable to construct fault tolerant connection networks. Drawbacks are found in the analysis of the simulation results and in the runtime for the application of the Tokyo railway system. Furthermore, relatively high effort has to be put in the parameter tuning of the complex agent-based algorithm, until it delivers satisfactory results. Additionally, it turned out that among the different modeling approaches for *Physarum polycephalum*, the agent based model the best one in the context of constructing networks between food sources.

Future work will include evaluation of both bio-inspired and classical approaches for a number of small to large benchmark problems in order to decide for which problem complexity classical or bio-inspired approaches are superior.

REFERENCES

[1] A. Tero, S. Takagi, T. Saigusa, K. Ito, D. Bebber, M. Fricker, K. Yumiki, R. Kobayashi, and T. Nakagaki, “Rules for biologically inspired adaptive network design,” *Science*, vol. 327, no. 5964, p. 439, 2010.

[2] W. Marwan, “Amoeba-Inspired Network Design,” *Science*, vol. 327, no. 5964, p. 419, 2010.

[3] T. Nakagaki, H. Yamada, and A. Toth, “Intelligence: Maze-solving by an amoeboid organism,” *Nature*, vol. 407, 2000.

[4] J. T. Bonner, “Brainless behavior: A myxomycete chooses a balanced diet,” *PNAS*, vol. 107, no. 12, pp. 5267–5268, 2010.

[5] M. Becker, “Design of fault tolerant networks with agent-based simulation of physarum polycephalum,” in *Evolutionary Computation (CEC), 2011 IEEE Congress on*, june 2011, pp. 285 –291.

[6] A. Tero, R. Kobayashi, and T. Nakagaki, “Physarum solver: A biologically inspired method of road-network navigation,” *Physica A: Statistical Mechanics and its Applications*, vol. 363, no. 1, pp. 115 – 119, 2006.

[7] A. Adamatzky and J. Jones, “Road planning with slime mould: If physarum built motorways it would route m6/m74 through newcastle,” *International Journal of Bifurcation and Chaos (IJBC)*, vol. 20, no. 10, pp. 3065–3084, 2010.

[8] S. Tsuda, K. Zauner, and Y. Gunji, “Robot control with biological cells,” *BioSystems*, vol. 87, no. 2-3, pp. 215–223, 2007.

[9] K. Li, K. Thomas, L. Rossi, and C. Shen, “Slime Mold Inspired Protocol for Wireless Sensor Networks,” in *Self-Adaptive and Self-Organizing Systems, 2008. SASO’08. Second IEEE International Conference on*. IEEE, 2008, pp. 319–328.

[10] A. Tero, K. Yumiki, R. Kobayashi, T. Saigusa, and T. Nakagaki, “Flow-network adaptation in physarum amoebae,” *Theory in Biosciences*, vol. 127, pp. 89–94, 2008.

[11] J. Jones, “The emergence and dynamical evolution of complex transport networks from simple low-level behaviours,” *International Journal of Unconventional Computing*, vol. 6, no. 2, pp. 125–144, 2010.

[12] M. Ikebe and Y. Kitauchi, “Evaluation of a multi-path maze-solving cellular automata by using a virtual slime-mold model,” *Unconventional Computing 2007*, p. 238, 2007.

[13] J. Gudmundsson, G. Narasimhan, and M. Smid, *Encyclopedia of Algorithms*. Berlin: Springer-Verlag, 2008, ch. Geometric spanners, pp. 360–364.

[14] A. Adamatzky and J. Jones, “Programmable reconfiguration of Physarum machines,” *Natural Computing*, vol. 9, no. 1, pp. 219–237, 2010.

[15] J. Jones, “Approximating the Behaviours of Physarum polycephalum for the Construction and Minimisation of Synthetic Transport Networks, Unconventional Computation 2009,” *Springer LNCS*, vol. 5715, pp. 191–208, 2009.

[16] M. Dorigo and L. Gambardella, “Ant colony system: a cooperative learning approach to the traveling salesman problem,” *Evolutionary Computation, IEEE Transactions on*, vol. 1, no. 1, pp. 53 –66, Apr. 1997.

[17] G. Narasimhan and M. Smid, *Geometric Spanner Networks*. New York, NY, USA: Cambridge University Press, 2007.

[18] P. Bose, J. Gudmundsson, and M. H. M. Smid, “Constructing plane spanners of bounded degree and low weight,” in *Proceedings of the 10th Annual European Symposium on Algorithms*, ser. ESA ’02. London, UK: Springer-Verlag, 2002, pp. 234–246.

A Technique for Measuring the Level of Autonomicity of Self-managing Systems

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

***Abstract*— Autonomic and self-managing systems are increasingly pervasive across an ever-widening spectrum of application domains. Autonomic technology is advancing at a high rate, yet there are no universal standards for the technology itself and the design methods used. There are also significant limitations to the way in which these systems are validated, with heavy reliance on traditional design-time techniques, despite the highly dynamic behaviour of these systems in dealing with run-time configuration changes and environmental and context changes. These limitations ultimately undermine the trustability of these systems and are barriers to eventual certification. This paper is concerned with setting the groundwork for the introduction of standards for Autonomic Computing (AC), in terms of technologies and the composition of functionality as well as validation methodologies. We propose that the first vital step in this chain is to introduce robust techniques by which the systems can be described in universal language, starting with a description of, and means to measure the extent of autonomicity exhibited by a particular system. We present a novel technique for measuring the Level of Autonomicity (LoA) along several dimensions of autonomic system self-CHOP (self-configuration, self-healing, self-optimisation and self-protection) functionalities.**

Keywords- autonomicity; level of autonomicity; autonomic system; trustworthiness; metrics

I. INTRODUCTION

AC seeks the development of self-managing (or autonomic) systems to address management complexities of systems. The high rate of advancement of autonomic technology and methodologies has seen these systems increasingly deployed across a broad range of application domains yet without universal standards. Also the widening acceptance of Autonomic Systems (AS) is leading to more trust being placed in them with little or no basis for this trust, especially in the face of significant limitations regarding the way in which these systems are validated. The traditional design-time validation techniques fail to address the run-time requirements of AS' environmental and contextual dynamism. These limitations undermine trustability and ultimately impinge on certification. The more this proliferation goes on without these challenges being addressed, the more difficult it gets to introduce standards and eventually achieve certifiable AS. It has therefore become pertinent and timely to address these issues. A vital first step in this course would be standards for the universal description of these systems and a standard technique for measuring LoA achieved by these systems. Standards for AC would be concerned with technologies, composition of functionalities and validation methodologies. By autonomicity we mean the ability of a system to pursue its goal with minimal external interference in the form of configuration or control. Then, the extent of this interference defines autonomicity levels. Now the questions facing the AC community are, for a given system, "How autonomic should a system be?" and "How autonomic is a system and how is this determined?" The two questions address both pre and post system design phases. The first question is of primary importance to the designers of systems where autonomic specification is a critical part of the whole system requirements definition. A good example would be spaceflight vehicles addressed in [1], where a *level of autonomy assessment tool* was developed to help determine the level of autonomy required for spaceflight vehicles. The second question is in two parts. On the one hand is the need to define systems according to a measure of autonomicity and another is the method and nature of the measure. Addressing this issue is the main thrust of this paper. Another significant aspect addressed here is the need for evaluation of systems in terms of individual functionalities. Not only do we measure autonomicity but also look at how systems can be evaluated and compared in terms of their autonomic compositions. We call this '*ranking*' (see Section IV B).

Thaddeus *et al* [2] identified that defining LoA is one of the critical stages along the path towards certifiable AS. Along this path also is the need for an appropriate testing methodology that seeks to validate the AS decision-making process. But to know what testing (validation) is appropriate requires knowledge of the system in terms of its extent of autonomicity. Another issue that underpins the need for measuring LoA is that a means of answering the identified questions is also a solution for AS evaluation and ranking and facilitates a proper understanding of such systems.

Currently, the vast majority of research effort in this direction has progressed in answering the first question ("How autonomic should a system be?") by providing us with scales that describe autonomy in systems. These scales, referenced by many researchers, provide fundamental understanding of system autonomy by categorising autonomy according to level of human-machine involvement in decision-making and execution. Some key works in this area include [1], [3], and [4]. For us, these scales only characterise autonomy levels qualitatively and offer no means of quantitatively measuring extent of autonomicity. We would simply say that they are more sufficient for the purposes of proposing an appropriate level of autonomy during the design of a new system.

ISO/IEC 9126-1 standard [13] decomposes overall software product quality into characteristics, sub characteristics (attributes) and associated measures. Adapting this we define a framework for measuring LoA along several dimensions of AS self-CHOP functionalities. Systems are well-defined by their set of functional capabilities and a measure of these capabilities will form a better representation of the systems. In our proposal we look at the core functionalities of ASs, the self-CHOP functionalities (hereafter referred to as CHOP), and identify specific metrics for each of the functionalities. The cumulative measure of these metrics defines a LoA. Our method is based on the establishment of a generic technique that can be applied to any application domain. This work is novel as it offers a quantitative measure of LoA in terms of system's functionalities-based description. It also opens a new research focus for autonomicity measuring metrics. We believe this is timely because if not addressed we not only run the risk of classifying systems as trusted without basis but also risk losing track and control of these systems as a result of spiraling complexities in terms of technology and methodologies. [15] also raised the concern that if the proliferation of unmanned systems (and by extension ASs) is not checked by putting appropriate measures (or mechanisms) in place that ensure trustworthiness, the systems may ultimately lose acceptance and popularity.

The remainder of this paper is organised as follows: related work is presented in section II. In Section III, we introduce metrics for measuring autonomicity. Our proposed LoA measure and a case study is presented in Section IV. Section V concludes the work.

II. RELATED WORK

The study of AC is now a decade old. However, its rapid advancement has led to a wide range of views on meaning, architecture, and implementations. The criticality of understanding *extent* of autonomicity in defining AC systems has necessitated the need for evaluating these systems. The majority of research in this area has targeted specific application domains with Unmanned Systems Technology (UST) topping the list.

One major proposal for measuring LoA is the scale-based approach. This approach uses a scale of (1– *n*) to define a system's LoA where '1' is the lowest autonomic level usually describing a state of least machine involvement in decision-making and '*n'* the highest autonomic level describing a state of least human involvement. Clough [3] proposes a scale of (1–10) for determining Unmanned Aerial Vehicles' (UAV's) autonomy. Level 1 '*remotely piloted vehicle'* describes the traditional remotely piloted aircraft, while level 10 '*fully autonomous'* describes the ultimate goal of complete autonomy for UAVs. Clough populates the levels between by defining metrics for UAVs. Sheridan [7] also proposes a 10-level scale of autonomic degrees. Unlike Clough's scale, Sheridan's levels 2-4 centre on who makes the decisions (human or machine), while levels 5-9 centre on how to execute decisions. Ryan *et al* [1], in a study to determine the level of autonomy of a particular AS decision-making function, developed an 8-level autonomy assessment tool. The tool ranks each of the OODA (Observe, Orient, Decide and Act) loop functions across Sheridan's proposed scale of autonomy [7]. OODA is decision-making loop architecture for ASs. The scale's bounds (1 and 8) correspond to complete human and complete machine responsibilities respectively. They first identified the tasks encompassed by each of the functions and then tailored each level of the scale to fit appropriate tasks. The challenge here is ensuring relative consistency in magnitude of change between levels across the functions. The levels are broken into three sections. Levels 1-2 (human is primary, computer is secondary), levels 3-5 (computer and human have similar levels of responsibility), and levels 6-8 (computer is independent of human). To determine the LoA needed to design into a spaceflight vehicle Ryan *et al* needed a way to map particular functions onto the scale and determine how *autonomous* each function should be. They designed a questionnaire (sent to system designers, programmers and operators) that considered what they call '*factors for determining LoA*' –these are LoA *trust limit* and *cost/benefit ratio limit*. This implies that a particular LoA for a function is favoured when a balance is struck between trust and cost/benefit ratio limits. Ultimately the pertinent question is "How autonomous should future spaceflight vehicles be?" IBM's 5 levels of automation [4] describes the extent of automation of the IT and business processes. We consider these to be too narrowly defined and [5] observes that the differentiation between levels is too vague to describe the diversity of self-management. In general the autonomy scale approach is qualitative and does not discriminate between behaviour types. We posit that a more appropriate approach should comprise both qualitative and quantitative measures.

Barber and Martin [8] supposes that in a multi-agent system environment, agent autonomy is measured in terms of a system-wide goal. It proposes a collaborative decision-making algorithm for multi-agent systems. In the proposed algorithm, a plan for achieving the system's goal is decided by the agents. Every agent suggests a complete plan with justification for how to achieve the entire system's goal. Each agent evaluates each suggested plan and determines the value of its justification. Each plan receives an integer number of votes from the deciding agents. The plan with the highest votes becomes the plan for the entire system. The ratio of an agent's number of votes (received for suggested plan) to the total number of votes cast is a measure of that agent's autonomy and the extent of its capability to influence the system. This method, however, does not offer a measure for LoA but gives a valuable description of agents' individual influence in a multi-agent system environment.

Fernando *et al* [6] proposes measures for evaluating the autonomy of software agents. It believes that a measure of autonomy (or any other agent feature) can be determined as a function of well-defined characteristics. Firstly it identifies the agent autonomy attributes (self-control, functional independence, and evolution capability) and then defines a set of measures for each of the identified attributes. By normalising the results of the defined measures using a set of functions, the agent's LoA is defined. [6] considers autonomicity measure with reference to system's characteristics and attributes. But in that work *'characteristics'* are a broad range of attributes that describe a system which also include features outside the system's core functionalities and so in a way are vague and limited in offering a proper and conclusive description of that system. We have adapted this approach in our proposal for ASs but with reference to self-CHOP functionalities.

III. AUTONOMICITY MEASURING METRICS

In this section, we introduce metrics for each of the functionalities that define autonomicity of AS. Though metrics are application domain dependent, the metrics presented here are generic and serve as examples. We present at least one metric for each of the functionalities. This is part of a wide (and separate) research focus. This section only focuses on how autonomic metrics can be generated. We also show how metrics can be normalised to yield autonomic values (see Section IV A). We will start with a definition of each CHOP. (For more on these definitions see [10] and [12]).

***Self-Configuring*:** A system is self-configuring when it is able to automate its own installation and setup according to high-level goals. When a new component is introduced into an AS it registers itself so that other components can easily interact with it. The extent of this interoperability (I) is a measure of self-configuration, measured as ratio of the actual number of components ($_{actual}n_i$) successfully interacting with the new component (after configuration) to the number of components expected ($_{expected}n_i$) to interact with the new component.

$$I = \sum_{1}^{i} \frac{_{actual}n_i}{_{expected}n_i} \quad (1)$$

Interoperability ratio I measures to what extent a system is distorted by an upgrade. A system is self-configuring to the extent of its ability to curb this distortion. This example can be related to the problem diagnosis system for AS upgrade discussed in [10]. Here an upgrade introduces 5 software modules. The installation regression testers found faulty output in 3 of the new modules. This implies that only 2 modules out of 5 successfully integrate with the system.

***Self-Optimising*:** A system is self-optimising when it is capable of adapting to meet current requirements and also of taking necessary actions to self-adjust to better its performance. Resource management (e.g., load balancing) is an aspect of self-optimisation. An AS is then required to be able to learn how to adapt its state to meet the new challenges. Also needed is consistent update of the system's knowledge of how to modify its state. State is defined by a set of variables such as current load distribution, CPU utilization, resource usage, etc. The values of these variables are influenced by certain event occurrences like new requirements (e.g., process fluctuations or disruptions). By changing the values of these variables, the event also changes the state of the system. The status of these variables is then updated by a set of executable statements (policies) to meet any new requirement. A typical example would be an autonomic job scheduling system. At first, the job scheduler could assign equal processing time quanta to all systems requiring processing time. The size of the time quantum becomes the current state and as events occur (e.g., fluctuations in processing time requirement, disruptions of any kind, etc.), the scheduler is able to adjust the processing time allocation according to priorities specified as policies. In this way the state of the system is updated. But this may lead to erratic tuning (as a result of over or under compensation) causing instability in the system. We define *Stability* as a measure of self-optimisation. If an event leads to erratic behaviour, incoherent results or system not been able to retrace its working state beyond a certain safe margin (a margin within which instability is tolerated) then the system is not effectively self-optimising.

***Self-Healing*:** A system is self-healing when it is able to detect errors or symptoms of potential errors by monitoring its own platform and automatically initiate remediation [11]. Fault tolerance is one aspect of self-healing. It allows the system to continue its operation possibly at a reduced level instead of stopping completely as a result of a part failure. One critical factor here is latency; the amount of time the system takes to detect a problem and then react to it. We define *reaction time* T as a metric for self-healing capability. This is crucial to the reliability of a system. If a change occurs at time $\boldsymbol{t_a}$ and the system is able to detect and work out a new configuration and ready to adapt at time $\boldsymbol{t_b}$ then (2) defines the reaction time T. (Average is taken instead where variations of T are possible).

$$T = t_b - t_a \quad (2)$$

A case scenario is a stock trading system where time is of paramount importance. The system needs to track changes (e.g., in trade volumes, price, rates etc.) in real time in order to make profitable trading decisions.

Self-Protecting: A system is self-protecting when it is able to detect and protect itself from attacks by automatically configuring and tuning itself to achieve security. It may also be capable of proactively preventing a security breach through its knowledge based on previous occurrences. While self-healing is reactive, self-protecting is proactive. A proactive system, for example, would maintain a kind of log of trends leading to security problems (threats and breaches) and a list of solutions to resolve them (a list of problems and corresponding solutions only applies to self-healing). One major metric here is the ability of the system to prevent security issues based on its experience of past occurrences. For example let's assume $p \in \{p_{ij}\}$ to be true if i^{th} trend leads to j^{th} problem where p_{ij} is a log of all identified trends and corresponding problems. p is a particular instance of trend-problem combination. A self-protecting manager will avoid a situation of same trend leading to the same problem again by blocking the problem, addressing it or preventatively shutting down part of the system. We define *ability to detect repeat events* R as a self-protecting metric. R is a Boolean value (True indicates the manager is able to stop a repeating problem and False otherwise). If we choose two samples of $\{p_{ij}\}$ at different times (t_1 and t_2) then (3) defines R. (Different trends may lead to the same problem but a repeated trend-problem combination indicates a failure of the system to prevent a reoccurrence).

$$R = True \; \forall_{ij} \; if \; \{p_{ij}\}_{t1} \cap \{p_{ij}\}_{t2} = \varnothing \qquad (3)$$

One typical implementation of this is an antivirus system. Some antivirus systems learn about trends or patterns (signatures) and are able to make decisions based on this to proactively protect a system from an attack. The antivirus is able to stop repeatable patterns. Detecting problem reoccurrence is an active research focus in Autonomic Computing [18].

IV. PROPOSED LoA MEASURE

An AS is defined based on its achievement of the CHOP capabilities [11]. In our approach, we define a level of AS in terms of its extent of achieving the identified functionalities. (We understand that these functionalities may overlap i.e., are not necessarily orthogonal, thus some algorithms may influence several functionalities, but to make progress in this area we assume orthogonality for this preliminary work). If a system fails to provide at least a certain level of one of the CHOP, the system is said to be non autonomic. On the other hand if the system provides a full level of all the four capabilities, it is said to have achieved full autonomicity (as defined by our proposed scheme). Each functionality is defined by a set of metrics. An autonomic value contribution is assigned to each functionality which is spread across the set of metrics for that functionality. It then follows that each metric contributes a certain definite level of the assigned value. The cumulative normalisation of the measure of all metrics (for all functionalities) defines a LoA. Let the maximum LoA value for an AS be M. In generic terms, this will mean an AS having a LoA value of N ($0 \leq N \leq M$) and each functionality contributing a value in the range ($0 \leq x \leq M/4$), while each metric of each functionality contributes ($M/4$)/m ($m \neq 0$) autonomic value, where m is the number of identified metrics defining a particular functionality, and the constant 4 represents four CHOP functionalities. (With an ongoing debate on the composition of AS functionalities and the list substantially growing [16, 17], we choose to limit it to the original, and generally accepted four).

Given that any AS is defined by the four autonomic functionalities, the expression (4) is the representation of the possible combinations of the functionalities.

$$\sum_{r=1}^{4} {}^{n}C_{r} \qquad \Rightarrow 16 \text{ Combinations} \qquad (4)$$

This will give 15 possible combinations (excluding zero value which is a special case and not considered further as it means the system provides no autonomic functionality) where (n = 4) is the number of functionalities (the CHOP) and r is a category of the possibilities (a specific implementation combination of the functionalities). The CHOP functionalities may not be of equal importance to an application domain so categories indicate what CHOP is important to an application domain. Category 2 means that only two functionalities are of importance to the system's domain –so for example {C, H, not O, not P} is a specific category representing a system indicated by line 4 in Figure 1.

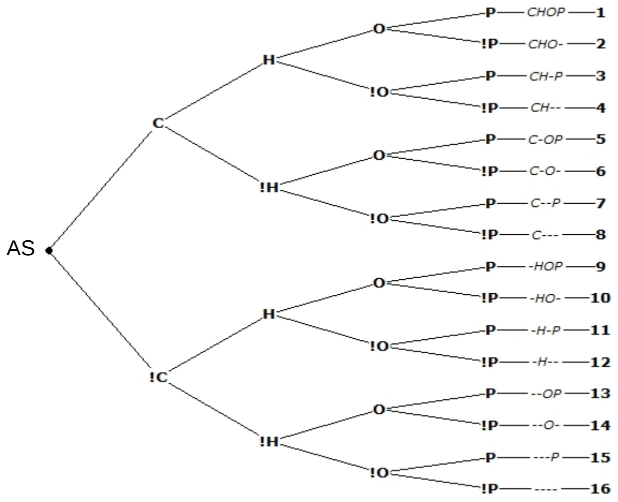


Figure 1: Combination of autonomic functionalities.

Figure 1 implies that, in terms of autonomic functionality composition, a system deemed autonomic (an AS) can be defined (or described) in one of fifteen ways. Each trace of line from start to finish represents an AS except line 16. If we define autonomic metrics for each of the functionalities, then the sum of the autonomicity in each of the constituent functionalities for a particular AS gives the system's LoA (5). For example, the LoA of a system represented by line 9 in Figure 1 will be the summation of the autonomic metrics defining the self-healing, self-optimising and self-protecting functionalities.

$$LoA = \sum_{i=1}^{mc}[c_i] + \sum_{j=1}^{mh}[h_j] + \sum_{k=1}^{mo}[o_k] + \sum_{l=1}^{mp}[p_l] \qquad (5)$$

Subscripted m is the number of identified metrics for the respective functionalities. c_i, h_j, o_k and p_l are the autonomic metric contributions of the functionalities. These can be composed of functions of different measures but as explained in Section III they are normalised to yield autonomic values.

A. *Normalising Autonomic Metrics*

Depending on the application domain, metrics could be scalar (of different measures) or non scalar values (e.g., observing a capability). One challenge here is defining and normalizing appropriate autonomic metrics. The metrics' values (irrespective of units of measure) are normalized into real numbers that are summed to give LoA (N). We identify two simple methods for normalization: 1) By ranking values according to *high*, *medium*, and *low*. The meaning of this ranking is metric-dependent and is based on a defined margin. For example, if a maximum expected value is 6, a value of 0-2 will be ranked *low*, while 3-4 will be ranked *medium* and 5-

6 *high*. A medium value would contribute fifty percent of the metric's autonomic value contribution of ($M/4$)/m, while the two extremes would contribute zero and hundred percents. This can be used for scalar metrics like the *interoperability ratio* and *reaction time* metrics discussed in Section III. 2) By having a Boolean kind of contribution where two values can suggest two extremes –either affirming a capability or not. For example, if a '*True*' outcome affirms a capability then it contributes hundred percent of the autonomic value contribution, while a '*False*' outcome contributes zero. Another example in this category is where an instance of an event either does or doesn't confirm a capability (e.g., the *stability* metric for self-optimising).

B. Evaluating Autonomic Systems

Evaluating Autonomic Systems using (5) gives their separate LoA values. Systems are classified according to categories. This is in terms of what CHOP functionalities are required in their specific application domains. One thing remains to be clarified at this point –'how do we rank each functionality in the autonomic composition of a system?' This can be in terms of importance or extent of functionality provided. We focus on the later –the extent of functionality provided as against what is needed. Take for instance, if two systems are of the same category we may wish to know which of them provides a greater degree of say self-healing or self-protection in any application domain. To address this we adapt a function that measures agent's decision-making power in a multi-agent AS defined in [8]. The rank R of a functionality in the autonomic composition of a system is defined by the ratio of its autonomic contribution x to the total autonomic contribution of all metrics defining the composite functionalities of that system.

$$R = \frac{x}{LoA} \tag{6}$$

where x is the autonomic contribution of the considered functionality which could be the summation of c_i, h_j, o_k or p_l as in (5). With (6) any composite functionality can be ranked in terms of their autonomic contribution. (See case study).

C. Autonomic Systems Evaluation Case Study

Our case study is *Dynamic Qualitative Sensor Selection System* (DQSSS), based on work in [14]. The goal of DQSSS is to *dynamically select a sensor (amongst many) based on continuously variable qualitative characteristics* (e.g., signal quality and noise levels). This is typical of an application that accesses several sensors generating raw data from monitoring a particular context; these could be physical attributes of a system or perhaps information feeds from a service (e.g. financial data). In such applications, it is expected that a DQSSS would generate and differentiate signal characteristics and trends, choose the best signal and without compromising stability, be continuous, unsupervised, dynamic, and detect and react if a sensor goes down. Autonomic metrics are drawn from these characteristics. By definition self-configuration, optimization and healing are of importance to this system (*r=3*). The DQSSS presented in [14] is in three stages which we refer to as systems A, B and C. All three systems are able to differentiate sensors by their signal characteristics such as noise level and spikes. These are then combined in a *utility function* to determine the better quality sensor. Systems B and C are able to generate trends in signal quality using *trend analysis* logic. Only system C ensures stability (avoiding unhealthy oscillation in sensor selection) by implementing *dead zone* logic, while none of the systems has a way of detecting a failed sensor.

TABLE I: REPRESENTATION OF THE DQSSS [14]

Characteristics (metrics)	**Contributing CHOP**	**Sys A**	**Sys B**	**Sys C**
Continuous	C	√	√	√
Unsupervised	C	√	√	√
Trends examination	O	-	√	√
Stability	O	-	-	√
Dynamic (logic switching)	O	-	-	√
Signal characteristics	C	√	√	√
Signal differentiation	C	√	√	√
Failure sensitivity (sensors)	H	-	-	-
Robust (fault tolerance)	H	-	-	√

We adopt the 8-level autonomy assessment scale in [1] as a way of qualitatively interpreting our results. In keeping with this we adopt the arbitrary value 8 as the maximum LoA implying that each CHOP contributes an autonomic value in the range ($0 \le x \le 2$) spread across its metrics. Normalizing the identified metrics in Table I (the numbers of metrics in each category are: C=4, H=2, O=3) in the autonomic value range ($0 \le x \le 2$) gives the result in Table II.

Figure 2 is a radar chart analysis of systems A, B and C in terms of their separate autonomic functionality composition. Recall that only three functionalities (CHO-) are of importance here which means maximum LoA value of 6. Out of this maximum value systems A, B and C achieved the values 2 (i.e., 33%), 2.67 (i.e., 45%) and 5 (i.e., 83%) respectively. This means that in a dynamic sensor selection application domain (as defined), system C can be depended upon to carry out the task with a confidence level of 83% and 0.17 risk factor, B with 45% and 0.55 risk factor, while A with 33% and 0.67 risk factor. Furthermore this can also be interpreted using Ryan *et al*'s level of autonomy assessment scale [1]. The scale as explained in Related Work section is an 8-level autonomy assessment tool used for either identifying (qualitatively) the level of autonomy of an existing system or for proposing an appropriate level of autonomy during the design of a new system. System A falls within level 2 of the scale which points to a situation where '*computer shadows human*' in the self-management process. This indicates that system A only has a narrow envelope of environmental conditions in which it is both autonomic and returns satisfactory behaviour. System B tends toward level 3 on the scale which is '*human shadows computer*' which translates into a wider operational envelope, but once the limits of that envelope are reached human input is needed in the form of retuning, or manual override in the case of oscillation, which for example system C can deal with autonomicaly. System C falls within level 5, which points to '*collaboration with reduced human intervention*'. This indicates that C is sufficiently sophisticated to operate autonomicaly and yield satisfactory results under almost all perceivable operating circumstances.

Employing (6) to rank the functionalities and taking just self-configuration for example, we find that in system A self-configuration contributes 100% of its autonomic achievement, while in systems B and C the contribution is 75% and 40% respectively. This shows that system A is entirely a self-configuring system, while system C is more of a self-optimising system than B.

TABLE II: ANALYSIS RESULT

	Sys A	Sys B	Sys C
C	2	2	2
H	0	0	1
O	0	0.67	2
LoA	**2**	**2.67**	**5**

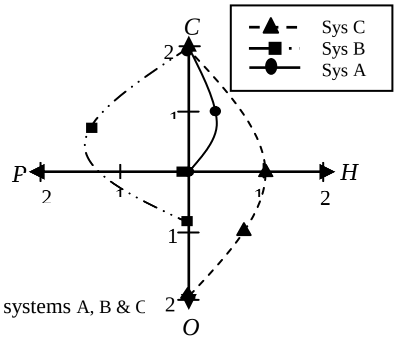


Figure 2: LoA representation of systems A, B & C in the four CHOP domains.

The benefit of analyzing Autonomic Systems in terms of their extent of autonomicity not only offers a path to Autonomic Systems' certification as stated earlier, it also offers a way of comparing these systems, and also facilitates a proper description of these systems to users.

V. CONCLUSION AND FUTURE WORK

A system is better defined by its capabilities and so measuring the LoA of Autonomic Systems without a reference to autonomic functionalities would be inaccurate. We have proposed a CHOP-based LoA measurement. In our proposal a typical AS is defined by the four CHOP functionalities (self-configuring, -healing, -optimising and -protecting) and LoA is measured with respect to these functionalities. Each functionality is defined by a set of metrics. The metrics values are normalised and aggregated to give the autonomic contribution of each functionality which are then combined to yield a LoA value for an AS. We have adopted the maximum autonomic value of 8 to correspond with the autonomy assessment scale defined in [1] to enable a qualitative understanding of the quantitative LoA measure proposed here. We have also shown how systems can further be evaluated by looking at the ratio of autonomic contributions of their separate functionalities. In this, we found that only systems of the same autonomic categorisation can be compared (e.g., a space exploration system cannot be directly compared with a resource allocation system as both are uniquely defined in terms of context and functionalities).

The standardization of a technique for the measurement of LoA will bring many quality-related benefits which include being able to compare alternative configurations of ASs, and even to be able to compare alternate systems themselves and approaches to building ASs, in terms of the LoA they offer. This in turn has the potential to improve the consistency of the entire lifecycle of Autonomic Systems and in particular links across the requirements analysis, design and acceptance testing stages.

As future work, we are looking at exploring areas where the CHOP are not orthogonal and also how to properly define and generate autonomic metrics to strengthen our framework. This is a key component towards our wider research which focuses on the challenge of validating AC systems to achieve trustworthiness in Autonomic Systems.

REFERENCES

[1] Ryan W. Proud, Jeremy J. Hart, and Richard B. Mrozinski. *Methods for Determining the Level of Autonomy to Design into a Human Spaceflight Vehicle: A Function Specific Approach*. http://handle.dtic.mil/100.2/ADA515467 accessed 28/03/11

[2] Thaddeus O. Eze, Richard J. Anthony, Chris Walshaw and Alan Soper. *The Challenge of Validation for Autonomic and Self-Managing Systems.* In proceedings of The 7th International Conference on Autonomic and Autonomous Systems (ICAS), May 22-27, 2011 – Venice/Mestre, Italy

[3] Clough B T. *Metrics, Schmetrics! How The Heck Do You Determine A UAV's Autonomy Anyway?* In Proceedings of PerMis Workshop, pp 1–7. NIST, Gaithersburg, MD, 2002.

[4] IBM Autonomic Computing White Paper, *An architectural blueprint for autonomic computing*. 3rd edition, June 2005

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

[6] Fernando Alonso, José Fuertes, Löic Martínez, and Héctor Soza. *Towards a Set of Measures for Evaluating Software Agent Autonomy*. In proceedings of 8th Mexican Int'l Conference on Artificial Intelligence (MICAI), 2009

[7] Sheridan T. B.. *Telerobotics, Automation, and Human Supervisory Control*. The MIT Press. Cambridge, MA, USA 1992. ISBN:0-262-19316-7

[8] Barber, K. S. and Martin, C. E.. *Agent Autonomy: Specification, Measurement, and Dynamic Adjustment*. In Proceedings of the Autonomy Control Software Workshop at Autonomous Agents 1999 (Agents'99), 8-15. Seattle

[9] Hui-Min Huang, Kerry Pavek, James Albus, and Elena Messina. *Autonomy Levels for Unmanned Systems (ALFUS) Framework: An Update*. In proceedings of SPIE Defense and Security Symposium, Orlando, Florida. 2005

[10] Kephart J., Chess D. *The Vision of Autonomic Computing*. Computer, IEEE, Vol 36, Issue 1, 2003, pp 41-50

[11] Bantz D. F. Bisdikian, C. Challener, D. Karidis, J. P. Mastrianni, S. Mohindra, A. Shea, D. G. and Vanover, M.. *Autonomic Personal Pomputing*. IBM Systems Journal, Vol 42, No 1, 2003

[12] J. A. McCann and M. Huebscher. *Evaluation issues in Autonomic Computing.* In proceedings of Grid and Corporative Computing (GCC) Workshop, LNCS 3252, pp. 597-608, Springer-V erlag, Birlin Heidelber, 2004

[13] ISO/IEC 9126-1:2001(E), Software engineering — Product quality — Part 1: Quality model

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

[15] Gaea Honeycutt, *How Much Do we Trust Autonomous Systems?* Unmanned Systems -2008

[16] H. Tianfield. *Multi-agent Based Autonomic Architecture for Network Management*. In Proc. IEEE International Conference on Industrial Informatics, pp. 462–469, 2003

[17] W. Truszkowski, L. Hallock, C. Rouff, J. Karlin, J. Rash, M. G.Hinchey, and R. Sterritt, *Autonomous and Autonomic Systems*. Springer, 2009

[18] Mark B., Sheng M., Guy L., Laurent M., Mark W., Jon C. and Peter S., *Quickly Finding Known Software Problems via Automated Symptom Matching*, The 2nd International Conference on Autonomic Computing (ICAC), 2005, Seattle, USA

A Framework to Create Multi-domains Autonomic Middleware

Mahdi Ben Alaya and Thierry Monteil and Khalil Drira and Tom Guérout
CNRS; LAAS; 7 avenue du Colonel Roche, F-31077 Toulouse Cedex 4, France
Université de Toulouse; UPS, INSA, INP, ISAE; UT1, UTM, LAAS; F-31077 Toulouse Cedex 4, France
Email: ben.alaya@laas.fr, monteil@laas.fr, khalil@laas.fr, tguerout@laas.fr

***Abstract*—This paper proposes an enumeration and a classification of the services or functionality needed in the autonomic middleware. This allows to propose a second time the foundation for a framework that will be able to generate different middleware implementing autonomic loop and adapted to areas with different constraints and different needs. An illustration in the field of ”Machine to Machine” and more particularly of smart metering is given.**

***Keywords*- autonomic computing; middleware; architecture; components.**

I. Introduction

The increasing complexity for the management of current distributed software and system needs new solutions. The computer system ”selfware” was created in the year 1995 for this purpose. By applying the properties of ”self-*” to the computer systems, Kephart and Chess [2] and Brantz [1] define in 2003 the four paradigms to be implemented at least in such systems to become self-managed: self-configuring, self-optimizing, self-healing and self-protecting.

In the last year, we are witnessing a widespread use of autonomic loop in many areas: high performance computing, service management, M2M (Machine to Machine) system, network, etc. The expression of autonomic behavior in each area often results in the construction of middlware completely different. Yet, the basic principle remains the same even if the elementary actions constituting the various phases of the autonomic loop vary.

We propose in this work in progress paper to enumerate the main ”components”, ”services”, ”features” needed to create a generic framework for building autonomous middleware specific to each area. We then describe a generic architecture between the proposed components. Finally, we give an example of future utilization of this Framework in the case of M2M.

II. Related Work

There are many middlewares to implement autonomic principles. They can be intrusive in the managed system or not.

DeployWare [3], manages the deployment of autonomic distributed applications. This approach defines three roles in the management of the software. The ”expert software” is the specialist in software technology to deploy. ”The system administrator” gives the network configurations (description of the physical infrastructure deployment). ”The end user” is using the application deployed. The peculiarity of DeployWare is that it proposes specific language for deployment (DSL Domain Specific Language) and a virtual machine for this language. DeployWare language is defined by a meta-model and provides a graphical notation in the form of a UML profile. Two concepts are important for us: the importance of defining roles and use of specific language nearest for users.

OceanStore [4] is used for the field of distributed and persistent storage of data. Its main goal is the implementation of the four properties of ”self-management” applied to the high data availability. Its features are for ”self-healing” fault tolerance through data redundancy and automatic repair. Here the focus is on the ability of middleware to provide services. This requires an application of the autonomic loop in the middleware itself.

Oceano [5] is applied to the field of cluster management for computing intensive applications or applications with a processor load varies with time (web servers). The peculiarity of this approach is the way it manages the property of ”self-Optimizing” policies dictated by contracts SLAs (Service Level Agreement) to specify a level of service by type of cluster or client (using one or more clusters). The use of SLA seems an important step by the possibility of applying it to many areas.

Gryphon [6] brings to the monitoring the notion of prioritization of events, as well as processing and agglomeration of events. This is an event management system self-adaptive, in fact, this approach also uses the events described as meta-events that trigger a reconfiguration of its internal functioning. Intelligent processing of events is a prerequisite for scalability. This will also be addressed in the framework that we propose.

Astrolabe [7] is an approaches providing an API to develop applications with properties of ”self-management”. It is used to collect the states of a very large scale (several thousand to several million nodes) according to zones. The area is also cutting into a solution that we wish to implement through the use of the concept of group and adapted communication patterns.

TUNe [8] is based on a component model. Its particularity is to add autonomous behavior to different types of existing legacy software. It provides a uniform vision of controlled softwares using the method of encapsulation with components. The administration then uses the standardized interface provided by the component model and a set of generic sensors or probes reusable skeletons. we will implement a model of components and services based on SCA (Service Component Architecture)[10].

None of those middlewares can address different domains. Each one has some specific characteristics. The goal of our framework is to build different autonomic middlewares with specific properties covering the needs of the domain of utilization.

III. Framework principle

A. Functionalities

In this part, we presented the functionalities that should be provided by our framework. We are inspired by the list of the M2M (Machine to Machine) functionalities detailed by ETSI (European Telecommunications Standards Institute) [9] to specify our classification. We decided to structure our framework features into six classes which are: communication, security, data toolkit, autonomic, management and entity classes. **Complex/structured Communication class** involves machine-to-machine, machine-to-man, and man-to-machine communications based on multiple communication means, e.g. SMS, GPRS and IP Access:

- Event processing: integrate different kind of event processing style: simple, stream and complex flow.
- Service oriented interactions: support service invocation between requester and provider.
- Transmission scheduling: Manage the scheduling of network access and of messaging.
- Delivery modes: support any-cast, uni-cast, multi-cast and broadcast communication.
- Flow management: handle asymmetric flows and support flow priority.
- Multi path: support physical paths diversity.
- Addressing: abstraction of the underlying network structure including any network addressing mechanism.

Security class involves structures and processes needed to protect the system and the connected users and devices against danger, damage, loss, and crime:

- Authentication: support two-way authentication and strength level selection.
- Encryption: support appropriate confidentiality of the data exchange.
- Anonymous: Possibility to hide the identity and the location of the requestor.
- Data integrity: support verification of the integrity of the data exchanged.
- Privacy: System shall be capable of protecting privacy.
- Security credential and software upgrade: secure updates of application security software and context (keys and algorithm).

Data toolkit class contains modules used for collection, representation and reporting of data:

- Data Base: gives a tool to store all necessary data
- Data collection: it includes pre collection activities (target data, definitions, method, etc.), collection and present findings.
- Reporting: Supports many type of reporting: periodic, on-demand, scheduled and event-based reporting.
- Graph modeling: provide mechanism to represent data in advanced structures like tree or graph to have a mapping describing in details the physical system.

Autonomic class contains modules making a system able to manage itself [2] (self-configuring, self-healing, self-optimization and self-protecting) and dynamically adapt to change in accordance with business policies:

- Monitoring: Provides the mechanisms that collect, aggregate, filter and report details collected from managed entities.
- Analyzing: provides the mechanisms that correlate and model complex situations. Help to learn about the environment and predict future situations.
- Planning: provides mechanisms that construct the actions needed to achieve objectives using policy information to guide its work.
- Executing: provides the mechanisms that control the execution of a plan with considerations for dynamic updates.
- Policy: supports different type of behavior for the planning component.

Management class contains modules that allow remotely configuring and controlling connected devices.

- Configuration: supports maintaining consistency of a system performance and its functional and physical attributes with its requirements, design, and operational information.
- Deployment: manages components life-cycle and activities of release, install, uninstall, activate, deactivate, update, adapt, built-in and version tracking.
- Remote administration: Supports advanced control request and receive acknowledgments to administrate the middleware
- HMI (Human-Machine Interface) system: helps to manage the system graphically .

Entity class it handles resource that exist in the run-time environment of an IT system and that can be managed:

- Group: support a mechanism to create and remove groups, to introduce an entity into a group, modify the invariants of the members, remove an entity, list members, search entities in a group, identify entity groups where the entity is a member, etc.
- Session: start and stop session supporting cooperation between two or more communicating entities [11].

- Profile: support computer representation of a user and device model [11].
- Role: possibility to assign role to a connected entity to manage their behaviors, rights and obligations.
- Discovery: a connected entity to a network should be able to advertise itself and to discover other entities.
- Description: each entity should be able to describe itself and to detail its hosted services in a standard format.
- Registration: allowing an entity to subscribe to asynchronous event messages produced by a given service.
- Meta-data exchange: provide dynamic access to a device's hosted services and to their meta-data.

B. Architecture

The different classes and services defined above are included in an architecture based on the SCA standard. This will have great flexibility in the use and construction of a middleware. Indeed, the interactions between components can be defined by different means: rmi, web-service, java, etc. Similarly, it is very easy to replace the instantiation of a component by another, or to take only part of the available components.

It is planned to establish the composition of middleware from eclipse by drawing in architecture and components available and so generate the autonomic middleware corresponding to his need. In Figure 1, a UML components diagram shows a part of the relation between components of the framework.

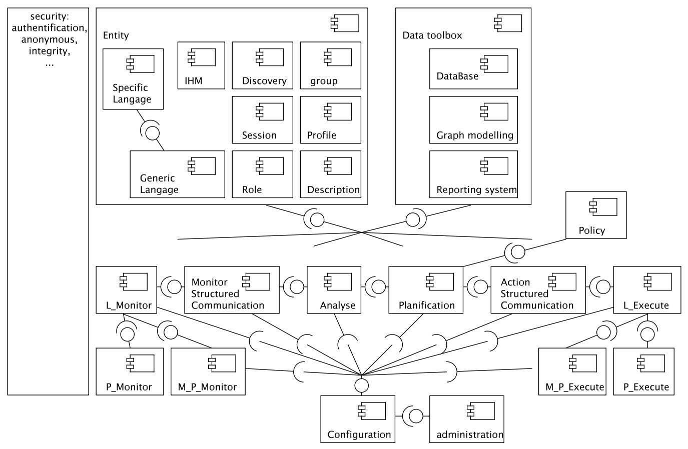


Figure 1. Framework architecture

We find the different elements of the autonomic loop with the monitoring of the system to manage (*P_monitor*) but also the monitoring of the autonomic middleware (*M_P_monitor*). The observed data are transformed into a generic vision through the *L_monitor* to be transmitted via a component of effective communication (*Structured Communication Monitor*, one or more component of the class communication) to be used by the component analysis (*Analysis*). The latter built a diagnosis that will generate a reaction (*Planning*) based on various policies (*Policy*). The set of elementary actions are effectively transmitted via the communication component (*Action Structured Communication*) to component (*L_Execute*) responsible for transforming the logical actions in specific actions to be executed by the actuators of the managed system (*P_Execute*) or middleware (*M_P_Execute*).

Depending on the area treated, there is a set of component toolbox class entity that can be used to provide specific services needed. There should be also the possibility in the toolbox to define specific languages readily available in various trades. There is also the use by the major components part of a toolbox to manage data from different patterns. Safety aspects are transversal to all this by using the notion of politics in SCA. All these components are configurable via the component *configuration* that orchestrates the system. Components of administration (*Administration*) and visualization (*HMI*) also allow to control the use of middleware in its execution.

IV. Example in M2M domain

The smart metering is a domain of M2M where autonomic loop could be used. Information such as energy consumption, temperature, light etc are collected with sensors. They are networked into a communication network that allows the sensed information to be fed to a central system where data can be analyzed then a list of actions can be planned. Actuators and appliances can next be automatically configured such as remotely reducing the level of the lamps or turning off the heating. ETSI specified six functionalities [12] related to smart metering expressed in broad terms, so that they can be related to electricity, gas, heating/cooling and water. Identifying functionalities at high level will permit flexibility, innovation and competition:

- Remote reading of metro-logical registers and provision to designated market organization.
- Provide two-way communication between the metering system and designated market organization.
- Support advanced tariffing and payment systems.
- Remote activation and deactivation of supply.
- Communicating with (and where appropriate directly controlling) individual devices within the building
- Providing information via gateway to an in-home display or auxiliary equipment.

The Figure 2 describes our smart metering architecture which involves the smart building ADREAM [13] that will serve as a real experimental platform to test our solution capabilities.

MAPE-K Loop modules will be used to self-manage smart metering operations: (*P_Monitor*) collects data from smart meters (electric, gaz, water and photovoltaic meters) and also from sensors (temperature, light, presence, etc). After analysing and planning, (*P_Execute*) executes required

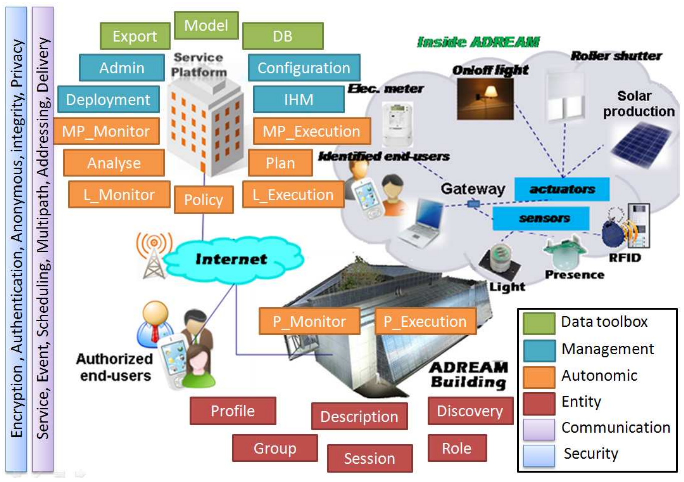


Figure 2. Smart metering architecture

actions to control different kind of actuators (roller shutter, on/off light, heating level, etc.).

(*M_P_Monitor*) supervises the middleware components and devices. It collects information about the middleware distributed machines context (Server memory, CPU, Rooting, etc.). If a problem is detected (Server down, big number of users, etc.) so, after analysing and planning, (*M_P_Execution*) executes actions such as deploying additional servers in new distributed machines or re-configuring gateways parameters to optimise the communication flows to add more scalability to the middleware.

For example, consider a scenario where a Customer decides to add a new sensor to regulate his consumption as a function of luminosity, so he looks his consumption and changes the way he wants that energy is consumed in his house. After being authenticated, the system characterizes his rights (*role*). It connects the new sensor that will be inserted dynamically into the system after authentication (*discovery*, *Registry*, *description* and inclusion in the *database*). It then displays (*HMI*) the consumer consumption and decides to change the behavior of the system of energy regulation (*profile*) because the system has automatically update the new possibilities offered in terms of regulation of energy thanks to this new sensor. The new autonomous policy (*policy*) is connected to the planning module (*Planning*).

V. Conclusion

In this paper, we present the basis for a framework that aims to create autonomic middleware specific to different application areas. The goal is to build a single tool that will be enriched and developed with many new components. The use of SCA should facilitate this. This framework will be used in various research projects and industry. A first prototype showing the feasibility of concepts is underway with the first application as the area of M2M. Future work will develop the framework and use it as part of large scale distributed computing. We can even think about interpretability scenarios between multiple autonomic middleware allowing to link several M2M domains.

References

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

[2] J. O. Kephart, and D. M. Chess, *The vision of autonomic computing*, Computer, pp. 41-50, 2003

[3] Areski Flissi, J Dubus, Nicolas Dolet, and Philippe Merle, *Deploying on the Grid with DeployWare*, 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID), Lyon France, pp. 177-184, 2008

[4] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and others, *Oceanstore: An architecture for global-scale persistent storage*, ACM SIGARCH Computer Architecture News, V. 28 N. 5, pp. 190-201, 2000

[5] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing and B. Rochwerger, *Oceano-SLA based management of a computing utility*, Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management, Seattle USA, 2001

[6] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward, *Gryphon: An information flow based approach to message brokering*, IBM TJ Watson Research Center Reports, 1998

[7] R. Van Renesse, K. P Birman, and W. Vogels, *Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining*, ACM Transactions on Computer Systems (TOCS), V. 21 N. 2, pp. 164-206, 2003

[8] Remi Sharrock, Thierry Monteil, Patricia Stolf, Daniel Hagimont, and Laurent Broto, *Non-intrusive autonomic approach with self-management policies applied to legacy infrastructures for performance improvements*, International Journal of Adaptive, Resilient and Autonomic Systems (IJARAS), V. 2 N. 2, pp. 1-20, 2010

[9] ETSI TS 102 689 Machine-to-Machine communications (M2M); M2M service requirements.

[10] Simon Laws, Mark Combellack, Raymond Feng, Haleh Mahbod, and Simon Nash, *Tuscany SCA in Action*, February, 2011 — 472 pages ISBN 9781933988894, manning

[11] M.Ben Alaya, V.Baudin, and K.Drira, *Dynamic deployment of collaborative components in service-oriented architectures* 11th International Conference of New Technologies in Distributed Systems (IEEE NOTERE2011), Paris, France, 2011.

[12] ETSI TR 102 691 Machine-to-Machine communications (M2M); Smart Metering Use Cases.

[13] http://www.laas.fr/ADREAM/

State-Space Feedback Control for Elastic Distributed Storage in a Cloud Environment

M. Amir Moulavi, Ahmad Al-Shishtawy, and Vladimir Vlassov
KTH Royal Institute of Technology
Stockholm, Sweden
Email: {moulavi,ahmadas,vladv}@kth.se

***Abstract*—Elasticity in Cloud computing is an ability of a system to scale up and down (request and release resources) in response to changes in its environment and workload. Elasticity can be achieved manually or automatically. Efforts are being made to automate elasticity in order to improve system performance under dynamic workloads. In this paper, we report our experience in designing an elasticity controller for a key-value storage service deployed in a Cloud environment. To design our controller, we have adopted a control theoretic approach. Automation of elasticity is achieved by providing a feedback controller that automatically increases and decreases the number of nodes in order to meet service level objectives under high load and to reduce costs under low load. Every step in the building of a controller for elastic storage, including system identification and controller design, is discussed. We have evaluated our approach by using simulation. We have developed a simulation framework EStoreSim in order to simulate an elastic key-value store in a Cloud environment and be able to experiment with different controllers. We have examined the implemented controller against specific service level objectives and evaluated the controller behavior in different scenarios. Our simulation experiments have shown the feasibility of our approach to automate elasticity of storage services using state-space feedback control.**

***Keywords*-elasticity; key-value store; Cloud; state-space feedback control**

I. Introduction

Web-based services frequently experience high workloads during their lifetime. A service can become popular in just an hour, and the occurrence of such high workloads has been observed more and more recently. Cloud computing has brought a great solution to the problem by requesting and releasing VM (Virtual Machine) instances that provide the service on-the-fly. This helps to distribute the loads among more instances. However, the high level load typically does not last for long and keeping resources in the Cloud costs money. This solution has led to Elastic Computing where a system running in the Cloud can scale up and down based on a dynamic property that is changing from time to time.

In 2001, P. Horn from IBM [1] marked the new era of computing as Autonomic Computing. He pointed out that the software complexity would be the next challenge of Information Technology. Growing complexity of IT infrastructures can undermine the benefits IT aims to provide. One traditional approach to manage the complexity is to rely on human intervention. However, considering the expansion rate of software, there would not be enough skilled IT staff to tackle the complexity of its management. Moreover, most of the real-time applications require immediate administrative decision-making and actions. Another drawback of the growing complexity is that it forces us to focus on management issues rather than improving the system itself.

Elastic Computing requires automatic management that can be provided using results achieved in the field of Autonomic Computing. Systems that exploit Autonomic Computing methods to enable automated management are called self-managing systems. In particular, such systems can adjust themselves according to the changes of the environment and workload. One common and proven way to apply automation to computing systems is to use elements of control theory. In this way a complex system, such as a Cloud service, can be automated and can operate without the need of human supervision.

In this paper, we report our experience in designing an elasticity controller for a key-value storage service deployed in a Cloud environment. To design our controller, we have adopted a control theoretic approach. Automation of elasticity is achieved by providing a feedback controller that continuously monitors the system and automatically changes (increases or decreases) the number of nodes in order to meet Service Level Objectives (SLOs) under high load and to reduce costs under low load. We believe that this approach to automate elasticity has a considerable potential for practical use in many Cloud-based services and Web 2.0 applications including services for social networks, data stores, online storage, live streaming services.

Our second contribution presented in this paper is an open-source simulation framework called EStoreSim (Elastic key-value Store Simulator) that allows developers to simulate an elastic key-value store in a Cloud environment and be able to experiment with different controllers.

The rest of the paper is organized as follows. In Section II, we define the problem of automated elasticity and describe the architecture of an elastic Cloud-based key-value store with feedback control. Section III presents different approaches to system identification. In Section IV, we show how we construct a state-space model of our elastic key-value store. We continue in Section V by presenting the controller designing for our storage. Section VI summarises steps of controller design including system identification. In Section VII, we describe the implementation of our simulation framework

EStoreSim. Experimental results are presented in Section VIII followed by a discussion of related work in Section IX. Finally, our conclusion and our future work are given in Section X.

II. Problem Definition and System Description

Our research reported here aims at automation of elasticity of a key-value store deployed in a Cloud environment. We want to automate the management of elastic storage instances depending on workload. a Cloud environment allows the system that is running in the Cloud to scale up and down in few minutes in response to load changes. In-time and proper decisions regarding the size of the system in response to the changes in the workload is very critical when it comes to enterprise and scalable applications.

In order to achieve elasticity of a key-value store in the Cloud, we adopt a control theoretic approach to designing a feedback controller that automatically increases and decreases the number of storage instances in response to changes in workload in order to meet SLOs under high load and to reduce costs under low load. The overall architecture of the key-value store with the feedback controller is depicted in Fig. 1.

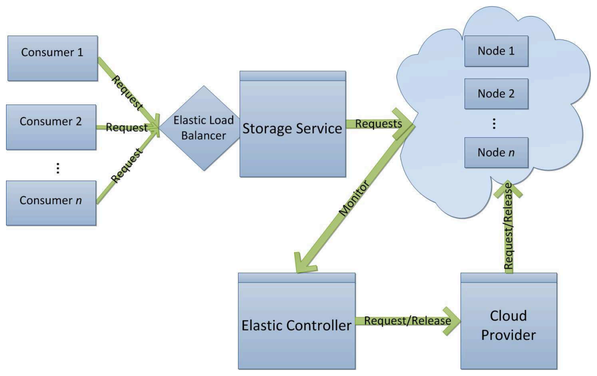


Fig. 1. Architecture of the Elastic Storage with feedback control of elasticity

End-users request files that are located in the storage Cloud nodes (instances). All the requests arrive at the Elastic Load Balancer (ELB) that sits in front of all storage instances. The Elastic Load Balancer decides to which instance the request should be dispatched. In order to do this, the Elastic Load Balancer tracks the CPU load and the number of requests sent previously to each instance and based on that it determines the next node that can serve the incoming request. In addition to the performance metrics that it tracks, ELB has the file tables with information about file replica locations since more than one instance can have a replica of the same file in order to satisfy the replication degree.

The Cloud Provider (Fig. 1) is an entity that is responsible for launching a new storage instance or terminating the existing one on requests of the Elasticity Controller.

Our system contains the Elasticity Controller, which is responsible for controlling the number of storage instances in the Cloud in order to achieve the desired SLO (e.g., download time). The Controller monitors the performance of the storage instances (and indirectly the quality of service) and issues requests to scale the number of instances up and down in response to changes in the measured quality of service (compared to the desired SLO). These changes are caused by changes in the workload, which is not controllable and is considered to be a disturbance in terms of control theory.

In the following two sections, we provide the relevant background and present steps of the *design of the controller* including *system identification* [2].

III. Approaches to System Identification

In this section, we present methods of system identification, which is the most important step in the design of a controller. It deals with how to construct a model to identify a system. System identification allows us to build a mathematical model of a dynamic system based on measured data. The constructed model contains a number of transfer functions, which define how the output depends on past/present inputs and outputs. Based on the transfer functions and desired properties and objectives, a control law is chosen. System identification can be performed using one of the following approaches.

First principle approach is one of the de facto approaches to identification of computer systems [3]. It can be considered as a consequence of the queue relationship. The first principle approach is developed based on knowledge of how a system operates. For example, this approach has been used in some studies and systems like [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14]. However, there are some shortcomings with this approach that have been stated in [2]. It is very difficult to construct a first principle model for a complex system. Since this approach considers detailed information about the target systems, it requires an ongoing maintenance by experts. Furthermore, this approach does not address model validation.

Empirical approach starts by identifying the input and output parameters like the first principle approach. But rather than using a transfer function, an *autoregressive moving average* (ARMA) model is built and common statistical techniques are employed to estimate the ARMA parameters [2]. This approach is also known as *Black Box* [3]; and it requires minimal knowledge of the system. Most of the systems in our studies have employed a black-box approach rather than a first-principle approach for system identification, e.g., [2], [15], [16], [17], [18], [19]. This is mainly because the relationship between inputs and outputs of the system is complex enough so that the first-principle system identification cannot be done easily. One of the empirical approaches is to build a State-Space Model, which requires more knowledge of the internals of the system. We use the state-space model approach for system identification as described in the next section.

IV. State-Space Model of the Elastic Key-Value Store

A state-space model provides a scalable approach to model systems with a large number of inputs and outputs [3]. The state-space model allows dealing with higher order target systems without a first-order approximation. Since the studied system executes in a Cloud environment, which is complex and dynamic in a sense of dynamic set of VMs and applications, we have chosen state-space modeling as the system identification approach. Another benefit of using the state-space model is that it can be extended easily. Suppose that after the model is built, we find more parameters to control the system. This can be accommodated by the state-space model without affecting the characteristic equations as shown later in Section VI where we summarize a generic approach for system identification and controller design

The main idea of the state-space approach is to characterize how the system operates in terms of one or more variables. These variables may not be directly measurable. However, they can be sufficient in expressing the dynamics of the system. These variables are called *state variables*.

A. State Variables and the State-Space Model

In order to define the state variables for our system, first we need to define the inputs and measured outputs since the state variables are related to them. In particular, state variables can be used to obtain the measured output. It is possible for a state variable to be a measured output like it is in our case.

In our case, the system input is the number of nodes (instances) denoted by $\texttt{NN}(k)$ at time k. The measured system outputs (and hence state variables) are the following:

- *average CPU load* $\texttt{CPU}(k)$: the average CPU load of all instances currently running in the Cloud during the time interval $[k-1,k]$;
- *interval total cost* $\texttt{TC}(k)$: the total cost of all instances during the time interval $[k-1,k]$;
- *average response time* $\texttt{RT}(k)$: the average time required to start a download during the time interval $[k-1,k]$.

The value of each state variable at time k is denoted by $x_1(k)$, $x_2(k)$ and $x_3(k)$. The offset value for input is $\bar{u}_1(k) = \texttt{NN}(k) - \widehat{\texttt{NN}}$, where $\widehat{\texttt{NN}}$ is the operating point for the input. The offset values for outputs are

$$\bar{y}_1(k) = \texttt{CPU}(k) - \widehat{\texttt{CPU}} \quad (1)$$
$$\bar{y}_2(k) = \texttt{TC}(k) - \widehat{\texttt{TC}} \quad (2)$$
$$\bar{y}_3(k) = \texttt{RT}(k) - \widehat{\texttt{RT}} \quad (3)$$

where $\widehat{\texttt{CPU}}$, $\widehat{\texttt{TC}}$ and $\widehat{\texttt{RT}}$ are operating points for corresponding outputs.

The state-space model uses state variables in two ways [3]. First, it uses state variables to describe the dynamics of the system and how $\mathbf{x}(k+1)$ can be obtained from $\mathbf{x}(k)$. Second, it obtains the measured output $\mathbf{y}(k)$ from state $\mathbf{x}(k)$.

State-space dynamics for a system with n states is described as follows

$$\mathbf{x}(k+1) = \mathbf{A}\mathbf{x}(k) + \mathbf{B}\mathbf{u}(k) \quad (4)$$
$$\mathbf{y}(k) = \mathbf{C}\mathbf{x}(k) \quad (5)$$

where $\mathbf{x}(k)$ is a $n \times 1$ vector of state variables, $\mathbf{A}$ is a $n \times n$ matrix, $\mathbf{B}$ is a $n \times m_I$ matrix, $\mathbf{u}(k)$ is a $m_I \times 1$ vector of inputs, $\mathbf{y}$ is a $m_O \times 1$ vector of outputs and $\mathbf{C}$ is a $m_O \times n$ matrix.

According to equations 4 and 5, we can describe dynamics of our system as follows:

- Average CPU Load ($\texttt{CPU}$) is dependant on the number of nodes in the system and previous CPU load, thus it becomes

$$\begin{aligned} x_1(k+1) = \texttt{CPU}(k+1) = \\ a_{11}\texttt{CPU}(k) + \\ b_{11}\texttt{NN}(k) + \\ 0 \times \texttt{TC}(k) + 0 \times \texttt{RT}(k) \end{aligned} \quad (6)$$

- Total Cost ($\texttt{TC}$) is dependant on the number of nodes in the system (the more nodes we have, the more money we should pay) and the previous $\texttt{TC}$, hence it becomes

$$\begin{aligned} x_2(k+1) = \texttt{TC}(k+1) = \\ a_{21}\texttt{TC}(k) + \\ b_{21}\texttt{NN}(k) + \\ 0 \times \texttt{RT}(k) + 0 \times \texttt{CPU}(k) \end{aligned} \quad (7)$$

- Average Response Time ($\texttt{RT}$) is dependant on the number of nodes in the system and the CPU load, so it is

$$\begin{aligned} x_3(k+1) = \texttt{RT}(k+1) = \\ a_{31}\texttt{CPU}(k) + a_{33}\texttt{RT}(k) + \\ b_{31}\texttt{NN}(k) + \\ 0 \times \texttt{TC}(k) \end{aligned} \quad (8)$$

In each equation (6, 7, 8) terms with zero factor include those state variables that do not affect the corresponding state variable definition. Thus their coefficient is zero. This is to ensure that there is no relation between those state variables or the relation is negligible and can be ignored. Their presence in the equations is for the sake of clarity and completeness. In order to prove that there is no relation or that it is negligible one should do a sensitivity analysis to investigate this, but it is out of the scope of this paper.

The output for the system at each time point k is equivalent to the corresponding state variable:

$$\mathbf{y}(k) = I_3\mathbf{x}(k) \quad (9)$$

The outputs are the same as the internal state of the systems at each time. That is why the matrix $\mathbf{C}$ is an identity matrix, i.e., a diagonal matrix of 1's. The matrices of coefficients are:

$$\mathbf{A} = \begin{bmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ a_{31} & 0 & a_{33} \end{bmatrix} \quad (10)$$

$$\mathbf{B} = \begin{bmatrix} b_{11} \\ b_{21} \\ b_{31} \end{bmatrix} \quad (11)$$

$$\mathbf{C} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (12)$$

B. Parameter Estimation

In Section IV-A, we have derived the State-Space model (Equations 6-12) that describes the dynamics of an elastic key-value store. There are two matrices **A** and **B** that contain the unknown coefficients for the equations 6-8. In order to use the model to design the controller we need to estimate the coefficient matrices **A** and **B**.

Parameter estimation is done using experimental data. In this research, we use data obtained from the simulation framework EStoreSim that we have built, rather than from a real system, because the major focus is on controller design and the simulation framework allows us to experiment with different controllers. We have implemented a simulation framework EStoreSim (described in Section VII) of a Cloud system. Using the framework we can obtain experimental data for system identification.

To get the data, we have designed and run an experiment, in which we feed the system with an input signal and observe the output and internal state variable periodically. We change the input (which is the number of nodes in the system) by increasing it from a small number of nodes a to a large number of nodes b and then back from b to a in a fixed period of time, and measure outputs (CPU load, cost, and response time). In this way, we ensure the complete coverage of the output signals in their operating regions by the input signal (the number of nodes). Load should be generated according to an arbitrary periodic function to issue a number of downloads per seconds. The period of the function should be chosen such that at least one period is observed during the time of changing the input between $[a, b]$.

For example, using the modeler component of our framework EStoreSim (Section VII), we scale up the number of nodes from 2 to 10 and then scale down from 10 to 2. Every 225 seconds a new node is either added or removed (depending on whether we scale up or down); sampling of training data (measuring outputs) is performed every 10 seconds.

When identifying the system, the workload is modeled as a stream of requests issued by the request generator component where the time interval between two consecutive requests forms a triangle signal in the range $[1, 10]$ seconds as follows: the first request is issued after 10 seconds, the second after 9 seconds, etc. The requests are received by the load balancer component in the Cloud provider component. After each scaling up/down the system will experience 2 triangle loads of requests between 1 to 10 seconds. The time needed to experience 2 triangles is $4\sum_{i=1}^{10} i$, which is 220 seconds. That is why we have selected 225 seconds as the action time.

Once training data are collected, they can be used to compute the matrices **A** and **B** using the *multiple linear regression* method. We use the `regress(y,X)` function of Matlab to calculate matrices:

$$\mathbf{A} = \begin{bmatrix} 0.9 & 0 & 0 \\ 0 & 0.724 & 0 \\ 5.927 & 0 & 0.295 \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} 2.3003 \\ 0.0147 \\ 77.8759 \end{bmatrix}$$

V. Controller Design

In this section, we describe how the feedback controller for the elastic storage deployed in a Cloud environment is designed. The controller design starts by choosing an appropriate controller architecture according to system properties. There are three common architectures for state-space feedback control, namely, *Static State Feedback*, *Precompensated Static Control* and *Dynamic State Feedback*. A good comparison between these architectures can be found in [3]. A close investigation in this comparison reveals that dynamic state feedback control is more suitable for a Cloud system since it has disturbance rejection that the other two architectures lack. Disturbance (in terms of control theory) is observed in a Cloud in the form of changes in the set of virtual machines and workload of Cloud applications. Thus we choose dynamic state feedback control as our controller architecture for autonomic management of elasticity.

A. Dynamic State Feedback

Dynamic state feedback can be viewed as a State-Space analogous to PI (Proportional Integral) control that has good disturbance rejection properties. It both tracks the reference input and rejects disturbances. We need to augment the state vector with the control error $e(k) = r - y(k)$ where r is the reference input. We use integrated control error, which describes the accumulated control error. The integrated control error is denoted by $x_I(k)$ and computed as

$$x_I(k+1) = x_I(k) + e(k)$$

The augmented state vector is $\begin{bmatrix} \mathrm{x}(k) \\ x_I(k) \end{bmatrix}$. The control law is

$$u(k) = -\begin{bmatrix} \mathrm{K}_p & K_I \end{bmatrix} \begin{bmatrix} \mathrm{x}(k) \\ x_I(k) \end{bmatrix} \qquad (13)$$

where K_p is the feedback gain for $\mathrm{x}(k)$ and K_I is the gain associated with $x_I(k)$.

B. LQR Controller Design

An approach to controller design is to focus on the trade-off between control effort and control errors. The *control error* is determined by the squared values of state variables, which are normally the difference from their operating points. The *control effort* is quantified by the square of $u(k)$, which is the offset of the control input from the operating point. By minimizing control errors we improve accuracy and reduce both settling times and overshoot and by minimizing control effort, system sensitivity to noise is reduced.

Least Quadratic Regulation (LQR) design problem is parametrized in terms of relative cost of control effort (defined by matrix **R**) and control errors (defined by matrix **Q**). The quadratic cost function to minimize is the following [3]:

$$J = \frac{1}{2}\sum_{k=0}^{\infty}\left[\mathbf{x}^\top(k)\mathbf{Q}\mathbf{x}(k) + \mathbf{u}^\top(k)\mathbf{R}\mathbf{u}(k)\right] \qquad (14)$$

where **Q** must be positive semidefinite (eigenvalues of **Q** must be nonnegative) and **R** must be positive definite (eigenvalues of **R** must be positive) in order for J to be nonnegative.

After selecting the weighting matrices **Q** and **R**, the controller gains **K** can be computed using the Matlab `dlqr` function that takes as parameters the matrices **A**, **B**, **Q**, and **R**. The performance of the system with the designed controller can be evaluated by simulation. If the performance is not appropriate, the designer can select new **Q** and **R** and recompute the vector gain **K**.

In our example, the matrices **Q** and **R** are defined as follows:

$$\mathbf{Q} = \begin{bmatrix} 100 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R} = \begin{bmatrix} 1 \end{bmatrix}$$

We have given 100 to the element that corresponds to CPU Load to emphasize that this state variable is more important compared to the others. One can give a high weight to total cost TC to trade off cost for performance. Using the Matlab `dlqr` function we compute the controller gains `K = dlqr(A, B, Q, R)`. For example, using the results of system identification in the example in Section IV-B, the controller gains (corresponding to the measured outputs of the elastic storage, CPU, TC, and RT) are:

$$\mathbf{K} = \begin{bmatrix} 0.134 & 1.470162e-06 & 0.00318 \end{bmatrix}$$

C. Fuzzy Controller

The main purpose in using an additional fuzzy controller is to optimize the control input produced by the Dynamic State Feedback Controller that we have designed in Section V-B. A fuzzy controller uses heuristic rules that define when and what actions the controller should take. The output of the Dynamic State Feedback Controller (control input) is redirected together with measured outputs to the fuzzy controller, which decides if the control input should affect the system or not. The overall architecture for controllers is demonstrated in Fig. 2.

There is one important case that the dynamic state feedback controller cannot act accordingly. Let us assume that there are some instances with high CPU load. Since the average is high, the controller will issue a control request to add a number of new instances. The new instances will be launched and will start to serve requests. But at the beginning of their life cycle they have low CPU load, thus the average CPU load that is reported back to the controller can be low. The controller then assumes that the CPU load has dropped, and it requests to remove some nodes.

A closer look at the CPU loads reveals that we can not judge the system state by only the average CPU load. Hence the fuzzy controller also takes into account the standard deviation

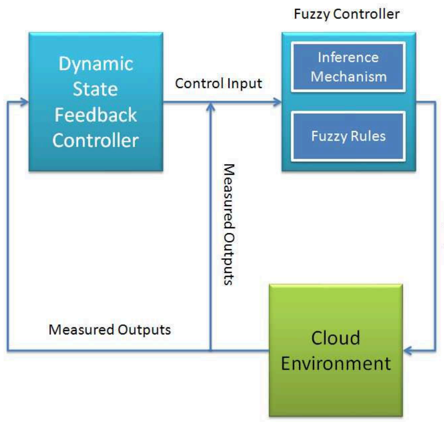


Fig. 2. Controllers Architecture

of CPU load. In this way, if the feedback controller gives an order to reduce the number of nodes when there is high standard deviation for CPU loads, the fuzzy controller will not allow this control input to affect the system, thus reducing the risk of unexpected results and confusions for the controller that may cause oscillations. This will lead to a more stable environment without so many unnecessary fluctuations.

D. Stability Analysis of Controller

A system is called stable if all bounded inputs produce bounded outputs. The BIBO theorem [3] states that for a system to be stable, its poles must lie within the unit circle (have magnitude less than 1). In order to calculate the poles for the controller we need to get the eigenvalues of matrix `A` that are 0.2951, 0.9 and 0.7247. As it is obvious from the values, all of the poles reside within the unit circle thus the controller is stable.

VI. Summary of Steps of Controller Design

This section summarizes the steps needed to design a controller for an elastic storage in a Cloud environment. The steps described below are general enough to be used to design a controller for an elastic service in a Cloud. The design process consists of two stages: system identification and controller design. The design steps are as follows: the system identification stage includes steps 1-9; and the remaining steps (10-12) belong to the stage of the controller design.

1) Study system behavior in order to identify the inputs and outputs of the system.
2) Place inputs and outputs in $\mathbf{u}(k)$ and $\mathbf{y}(k)$ vectors respectively.
3) Select n system outputs that you want to control and place them in state variable vector **x**. The outputs should be related to SLOs and performance metrics.
4) Select m system inputs that you will use to control. These system inputs will be the outputs of your controller. The system outputs should depend on the system inputs These inputs should have the highest impact in your system. In some systems there might be only one

input that has high impact whereas in other systems there might be several inputs that together have high impact. To assess the impact you might need to do sensitivity analysis.

5) Define state variables that describe the dynamics of the system. State variables can be equivalent to system outputs selected in step 3. Each state variable can depend on one or more other state variables and system inputs. Find the relation between the next value for a state variable to other state variables and system inputs and construct the characteristic equations as follows (see also Equation 4).

$$\begin{aligned} x_1(k+1) &= a_{11}x_1(k) + \ldots + a_{1n}x_n(k) \\ &\quad + b_{11}u_1(k) + \ldots + b_{1m}u_m(k) \\ x_2(k+1) &= a_{21}x_1(k) + \ldots + a_{2n}x_n(k) \\ &\quad + b_{21}u_1(k) + \ldots + b_{2m}u_m(k) \\ &\vdots \\ x_n(k+1) &= a_{n1}x_1(k) + \ldots + a_{nn}x_n(k) \\ &\quad + b_{n1}u_1(k) + \ldots + b_{nm}u_m(k) \end{aligned}$$

6) Place coefficients from the previous equations into two matrices **A** and **B**. Some of the coefficients can be zero:

$$\mathbf{A}_{n\times n} = \begin{bmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \ldots & a_{nn} \end{bmatrix}$$

$$\mathbf{B}_{n\times m} = \begin{bmatrix} b_{11} & \ldots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{n1} & \ldots & b_{nm} \end{bmatrix}$$

7) In order to simplify the design of controller, one can assume that outputs of the systems are equal to state variables, thus matrix **C** is an identity matrix:

$$\mathbf{C}_{n\times n} = \begin{bmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{bmatrix}$$

8) Design an experiment, in which the system is fed with its inputs. Inputs in the experiment should be changed in such a way that they cover their ranges at least one time. A range for an input is the interval that the values of the input will most likely belong to when the system operates. The selection of ranges can be based on industry's best practices. All inputs and outputs should be measured periodically with a fixed time interval T. Store collected data for each equation in a separate file called x_i.
9) In Matlab, for each file x_i, load the file and extract each column of data in a separate matrix. Use the function `regress` to calculate the coefficients. Repeat this for every file. At the end you will have all the coefficients that are required for matrices **A** and **B**.
10) Choose a controller architecture for feedback control: such as dynamic state feedback control, which is, in our opinion, more appropriate for a Cloud based elastic service (as discussed in Section V).
11) Construct matrices **Q** and **R** as described in Section V-B. Remember to put high weights in matrix **Q** for those state variables that are of more importance.
12) Use the Matlab function `dlqr` with matrices **A**, **B**, **Q** and **R** as parameters to calculate the vector **K** of controller gains. Perform stability analysis of the controller checking whether its poles reside within the unit circle (Section V-D).

VII. EStoreSim: Elastic Key-Value Store Simulator

We have implemented a simulation framework, which we call EStoreSim, that allows developers to simulate an elastic key-value store in a Cloud environment and to experiment with different controllers. We have selected Kompics as the implementation tool. Kompics [20] is a message-passing component model for building distributed systems using event-driven programming. Kompics components are reactive state machines that execute concurrently and communicate by passing data-carrying typed events through typed bidirectional ports connected by channels. For further information please refer to the Kompics programming manual and the tutorial on its web site [20].

Implementation is done in Java and Scala languages [21] and the source is publicly available at [22]. The overall architecture of EStoreSim is shown in Fig. 3. The simulator includes the following components.

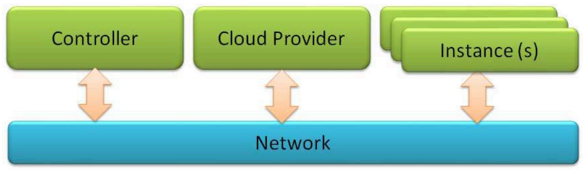


Fig. 3. Overall Architecture of the EStoreSim Simulation Framework

Cloud Instance Component represents an entire storage instance within a Cloud. The component architecture for instance is shown in Fig. 4.

Cloud Provider Component represents an important unit in the implementation. It is the heart of a simulated Cloud computing infrastructure and provides vital services to manage and administer the nodes (VM instances) within the Cloud. The Cloud provider component architecture is shown in Fig. 5.

Elasticity Controller represents the controller that can connect to the Cloud provider and retrieve information about the current nodes in the system. The main responsibility of the controller component is to manage the number of nodes currently running in the Cloud. In other words, it attempts to optimize the cost and satisfy some SLO parameters. The overall component architecture is shown in Fig. 6.

For further information on EStoreSim please refer to [22].

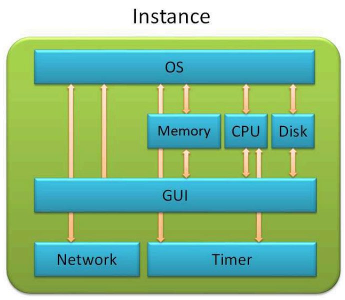


Fig. 4. Cloud Instance Component

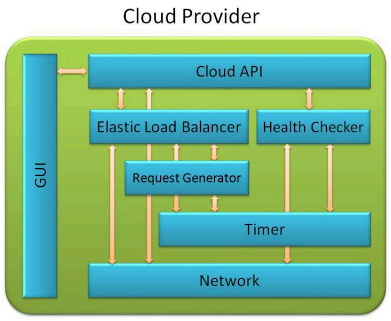


Fig. 5. Cloud Provider Component

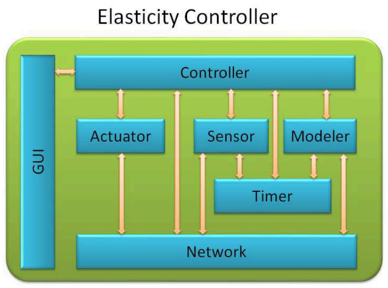


Fig. 6. Elasticity Controller Component

VIII. Experiments

We have conducted a number of simulation experiments using EStoreSim in order to evaluate how the use of an elasticity controller in a Cloud-based key-value store improves the operation of the store by reducing the cost of Cloud resources and the number of SLO violations. The baseline in our experiments is a non-elastic key-value store, i.e., a key-value store without the elasticity controller.

For evaluation experiments, we have implemented a dynamic state feedback controller with the parameters (controller gains) calculated according to the controller design steps (Section V-B). The controller is given reference values of the system outputs that correspond to SLO requirements. Values of system outputs (average CPU load $\mathtt{CPU}$, Total Cost $\mathtt{TC}$, and average Response Time $\mathtt{RT}$) are fed back into the controller periodically. When the controller gets the values, it calculates and places the next value of the number of nodes $\mathtt{NN}$ on its output. The controller output is a real number that should be rounded to a natural integer. We round it down in order to save the total cost the Cloud generates. One can assume two boundaries, which are defined as follows:

- L (Lower boundary): the minimum number of instances that should exist in the Cloud at all times;
- U (Upper boundary): the maximum number of instances that is allowed to exist in the Cloud at all times.

Hence if the value of controller output is smaller than L or greater than U, then the value should be discarded. If the calculated output of the controller is Θ, the number of nodes is defined as follows:

$$\mathtt{NN} = \begin{cases} L & \text{if} \quad \Theta \leqslant L \\ \Theta & \text{if} \quad L < \Theta < U \\ U & \text{if} \quad U \leqslant \Theta \end{cases} \tag{15}$$

If the number of current nodes in the system is $\mathtt{NN}'$ and the control input (output of the controller) is $\mathtt{NN}$, then the next control action is determined as follows:

$$\text{Next action} = \begin{cases} \text{scale up with } \mathtt{NN} - \mathtt{NN}' \text{ nodes} & \text{if} \quad \mathtt{NN}' < \mathtt{NN} \\ \text{scale down with } \mathtt{NN}' - \mathtt{NN} \text{ nodes} & \text{if} \quad \mathtt{NN} < \mathtt{NN}' \\ \text{no action} & \text{otherwise} \end{cases} \tag{16}$$

We have conducted two series of experiments to prove our approach to elasticity control. By these experiments we check whether the elasticity feedback controller operates as expected. In the first series (which we call SLO Experiment), the load is increased to a higher level. This increase is expected to cause SLO violation that is detected by the feedback controller, which adds nodes in order to meet SLO under high load. In the second series (which we call Cost Experiment), the load decreases to a lower level. This causes the controller to release nodes in order to save cost under low load. The instance configuration for these experiments are as follows:

- CPU frequency: 2 GHz;
- Memory: 8 GB;
- Bandwidth: 2 MB/s;
- Number of simultaneous downloads: 70.

There are 10 data blocks in the experiments with sizes between 1 to 5 MB. Note that the same configuration is used in the system identification experiments.

A. SLO Experiment: Increasing Load

In this series we conducted two experiments: one with controller and another without controller. In the results and figures presented below, they are denoted by `w/controller` and `w/o controller`, respectively. Each experiment starts with three warmed up instances. By a warmed up instance we mean that in this instance each data block is requested at least once thus it resides in the memory of this instance.

Workload that is used for this experiment is of two levels: normal and high. Under the normal load the time interval between consecutive requests is selected from a uniform random distribution in the range [10, 15] seconds that corresponds to an average request rate of 4.8 requests per minute. Under the high load the time interval between consecutive requests is selected from a uniform random distribution in the range [1, 5] seconds that corresponds to an average request rate of 20 requests per minute. The experiment starts with normal load and after 500 seconds the workload increases to the high level. This is shown in Fig. 7.

Sensing of instance output is done every 25 seconds. In the case of controller, actuation is performed every 100 seconds. Thus there are 4 sets of measured data at each actuation time that the controller should consider. In order to calculate values of the system output, the controller computes averages of data

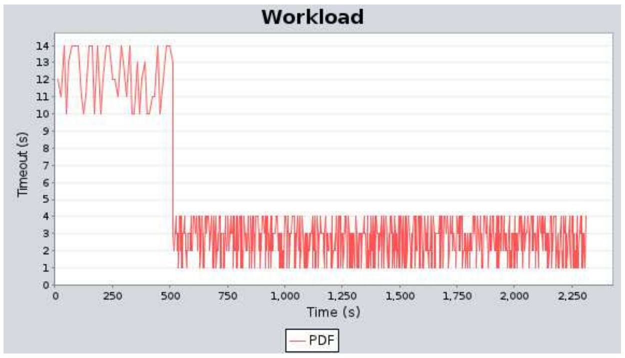


Fig. 7. SLO Experiment Workload

TABLE I
SLO VIOLATIONS

SLO Parameter Violation (%)	w/ Controller	w/o Controller
CPU Load	17.94	72.28
Response Time	2.12	7.073
Bandwidth	35.89	74.69

sets. The duration of each experiment is 2000 seconds with warm up of 100 seconds. SLO requirements are as follows:

- Average CPU Load: $\leqslant 55\%$
- Response Time: $\leqslant 1,5$ seconds
- Average Bandwidth per download: > 200000 B/s

For each experiment the percentages of SLO violations are calculated for each aforementioned SLO requirement based on Equation 17. The result is shown in Table I.

$$SLO\ Violations = 100\% \times \frac{Number\ of\ SLO\ Violations}{Total\ Number\ of\ SLO\ Checks} \quad (17)$$

Checking of SLO is done at each estimate (sensing) of the *Average CPU Load* and *Average Bandwidth per download* and each estimate of *Response Time*.

This experiment gives us interesting results that are discussed in this section. `NL` and `HL` in figures 8-12 indicate periods of *Normal Load* and *High Load* respectively.

Fig. 8 depicts the Average CPU Load for the aforementioned experiments. The Average CPU Load is the average of all nodes' CPU Loads at each time the sensing is performed. As one can see in Fig. 8, CPU loads for the experiment with the controller is generally lower than the same experiment without the controller. This is due to the controller that launches new instances under high workloads causing a huge drop in average CPU Load.

Fig. 9 depicts the Average Response Time for the experiments. By response time we mean the time that it takes for an instance to respond to a request that download is started and not the actual download time. As it is seen from the diagram, the average response time for the experiment with the controller is generally lower than the experiment without controller. This is because in case of having a fixed number of instances (3 in this experiment), there would be congestion by

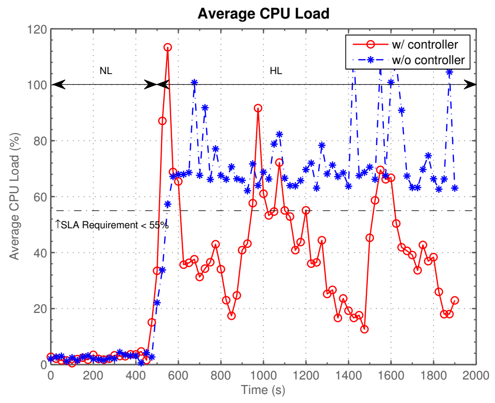


Fig. 8. SLO Experiment - Average CPU Load

	w/ controller	w/o controller
Total Cost ($)	14.4528	8.6779

TABLE II
TOTAL COST FOR EACH SLO EXPERIMENT

the number of requests an instance can process. This increases the responsivity of an instance. However, in the case that the controller launches new instances, no instance will actually go under high number of requests.

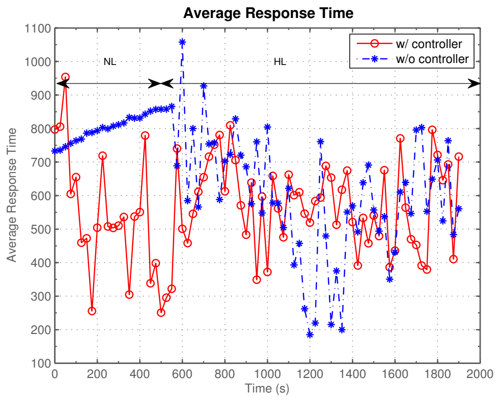


Fig. 9. SLO Experiment - Average Response Time

Fig 10 shows the total cost for the experiments. Interval total cost means that total cost is calculated for each interval in which the senses are done. As can be observed from the diagram, the interval total cost for the experiment with the controller is much higher than the experiment without the controller. This is because launching new instances will cost more money than having a fixed number of instances available in the Cloud. This experiment has high load of requests for the system in which the controller is more likely to scale up and resides in that mood than to scale down. It should be noted that costs are computed according to Amazon EC2 price list.

Calculated total cost for each experiment is given in Table II.

Fig. 11 depicts the Average bandwidth per download. If an instance has a bandwidth of 4 Mb/s and has two current downloads running, the bandwidth per download is 2 Mb/s. As

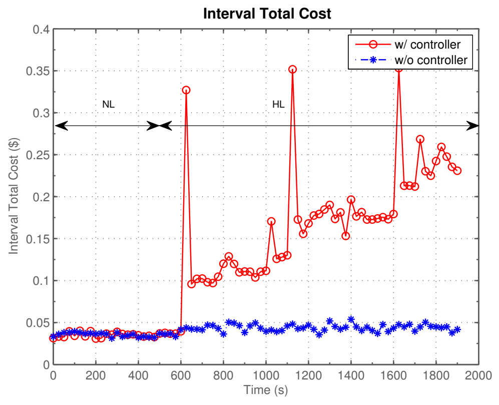


Fig. 10. SLO Experiment - Interval Total Cost

can be seen from the diagram, the experiment with controller shows significantly higher bandwidth per download. This is mainly because the instances receive less number of requests and bandwidth is divided among less requests also. This will end up having higher bandwidth available on each instance.

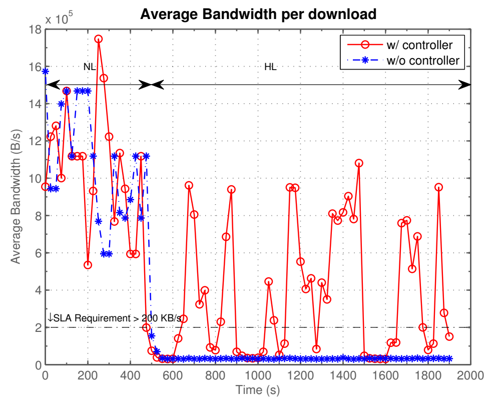


Fig. 11. SLO Experiment - Average Bandwidth per download (B/s)

Fig 12 shows the number of nodes for each experiment. As we discussed earlier the number of nodes is constant for experiment without controller. However, for the experiment with the controller the number of nodes is changed over time hence the SLO requirements can be met.

B. Cost Experiment: Decreasing Load

The purpose of this series of experiments is to show that the controller can save the total cost by releasing instances when the load is low. Each experiment in this series starts with 7 instances. The duration of the experiment is 2000 seconds.

In this series we use different workloads of two levels: high and low. In the high load the time interval between consecutive requests is selected from a uniform random distribution in the range [1, 3] seconds that corresponds to a request rate of 30 requests per minute. In the low load the time interval between consecutive requests is selected from a uniform random distribution in the range [15, 20] seconds that corresponds to a request rate of about 3.4 requests per minute. Unlike the SLO experiment, the cost experiment starts with a high load, which changes to a low load after 500 seconds as shown in Fig. 13.

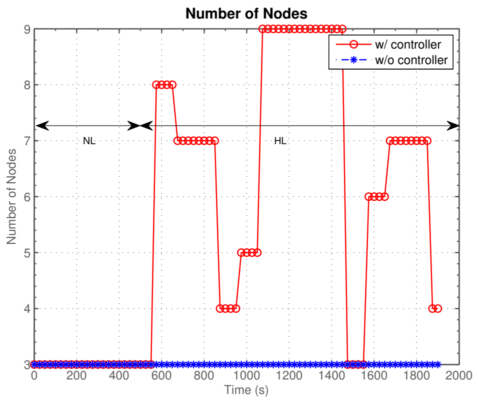


Fig. 12. SLO Experiment - Number of Nodes

	w/ controller	w/o controller
Total Cost ($)	10.509	16.5001

TABLE III
TOTAL COST FOR COST EXPERIMENT

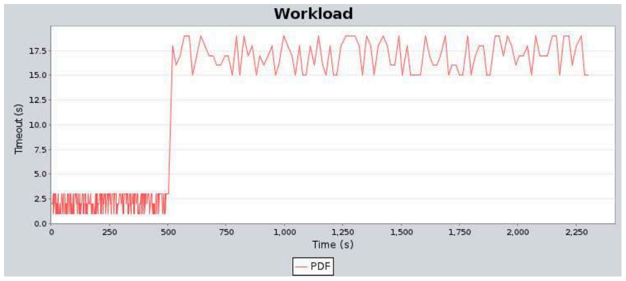


Fig. 13. Cost Experiment workload

The result of the cost experiment shown in Table III is interesting. It is observed that the total cost in the experiment with the controller is actually lower than the total cost in the experiment without the controller unlike in the SLO experiment. This is because the controller removes instances under low load and that results in cost savings. The reason that this experiment has lower cost than the previous one is that L (lower bound on number of nodes) is not equal to the initial number of nodes and it is smaller. Hence controller can scale down the number of nodes to L.

IX. RELATED WORK

There are many projects that use elements of control theory for providing automated control of computing systems including Cloud-based services [2], [7], [8], [9], [12], [15], [16], [17], [18], [19], [23]. Here we consider two related pieces of work [17], [23], which are the closest to our research aiming at automation of elasticity of storage services.

The SCADS Director proposed in [23] is a control framework that reconfigures a storage system at run time in response to workload fluctuations. Reconfiguration includes adding/removing servers, redistributing and replicating data between servers. The SCADS Director employs the Model-Predictive Control technique to predict system performance for the given workload using a performance model of the system and make control decisions based on prediction. Performance modeling is performed by statistical machine learning.

Lim et al. [17] have proposed a feedback controller for elastic storage in Cloud environment. The controller consists of three components: Horizontal Scale Controller responsible for scaling the storage; Data Rebalancer Controller that controls data transfer for rebalancing after scaling up/down; and the State Machine that coordinates the actions of the controllers in order to avoid wrong control decisions caused by interference of rebalancing with applications and sensor measurements.

To our knowledge both aforementioned projects do not explicitly use cost as a controller input (state variable, system output) in the controller design. In contrast, we use state-space feedback control and explicitly include the total cost of Cloud instances as a state (system output) variable in the state-space model (when identifying the system) and as a controller input in the controller design (when determining controller gains). This allows us to use a desired value of cost in addition to the SLO requirements to automatically control the scale of the storage by trading off performance for cost.

X. Conclusion and Future Work

Elasticity in Cloud computing is an ability of a system to scale up and down (request and release resources) in response to changes in its environment and workload. Elasticity provides an opportunity to scale up under high workload and to scale down under low workload to reduce the total cost for the system while meeting SLOs. We have presented our experience in designing an elasticity controller for a key-value store in a Cloud environment and described the steps in designing it including system identification and controller design. The controller allows the system to automatically scale the amount of resources while meeting performance SLO, in order to reduce SLO violations and the total cost for the provided service. We also introduced our open source simulation framework (EStoreSim) for Cloud systems that allows to experiment with different controllers and workloads. We have conducted two series of experiments using EStoreSim. Experiments have shown the feasibility of our approach to automate elasticity control of a key-value store in a Cloud using state-space feedback control. We believe that this approach can be used to automate elasticity of other Cloud-based services.

In our future work, we will study other controller architectures such as model predictive control, and conduct experiments using real-world traces. We will also research on using feedback control for other elastic Cloud based services.

Acknowledgments

This research is supported by the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research, the Complex Service Systems focus project, a part of the ICT-TNG Strategic Research Area initiative at the KTH Royal Institute of Technology, and by the Testbed for E2E Clouds RCLD-project funded by EIT ICT Labs.

References

[1] P. Horn, “Autonomic computing: IBM’s perspective on the state of information technology,” IBM, Tech. Rep., October 2001.

[2] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus, “Using control theory to achieve service level objectives in performance management,” *Real-Time Syst.*, vol. 23, pp. 127–141, July 2002.

[3] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems*. Wiley-IEEE Press, September 2004.

[4] D. Chiu and R. Jain, “Analysis of the increase and decrease algorithms for congestion avoidance in computer networks,” *Computer Networks and ISDN*, vol. 17, no. 1, pp. 1–14, 1989.

[5] S. Keshav, “A control-theoretic approach to flow control,” *SIGCOMM Comput. Commun. Rev.*, vol. 21, pp. 3–15, August 1991.

[6] B. Li and K. Nahrstedt, “A control-based middleware framework for quality-of-service adaptations,” *Selected Areas in Communications, IEEE Journal on*, vol. 17, no. 9, pp. 1632 –1650, sep 1999.

[7] A. Kamra, V. Misra, and E. M. Nahum, “Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites,” *In International Workshop on Quality of Service (IWQoS*, pp. 47–56, 2004.

[8] T. F. Abdelzaher, K. G. Shin, and N. Bhatti, “Performance guarantees for web server end-systems: a control-theoretical approach,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 1, pp. 80–96, August 2002.

[9] A. Robertson, B. Wittenmark, and M. Kihl, “Analysis and design of admission control in web-server systems,” in *American Control Conference. Proceedings of the 2003*, vol. 1, june 2003, pp. 254–259.

[10] T. Abdelzaher and N. Bhatti, “Web content adaptation to improve server overload behavior,” in *WWW8 / Computer Networks*, 1999, pp. 1563–1577.

[11] B. Li and K. Nahrstedt, “A control theoretical model for quality of service adaptations,” in *In Proceedings of Sixth International Workshop on Quality of Service*, 1998, pp. 145–153.

[12] H. D. Lee, Y. J. Nam, and C. Park, “Regulating i/o performance of shared storage with a control theoretical approach,” *NASA/IEEE conference on Mass Storage Systems and Technologies (MSST)*, April 2004.

[13] S. Mascolo, “Classical control theory for congestion avoidance in high-speed internet,” in *Decision and Control, 1999. Proceedings of the 38th IEEE Conference on*, vol. 3, 1999, pp. 2709 –2714 vol.3.

[14] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole, “A feedback-driven proportion allocator for real-rate scheduling,” in *Proceedings of the third symposium on Operating systems design and implementation*, ser. OSDI ’99. Berkeley, CA, USA: USENIX Association, 1999, pp. 145–158.

[15] M. Karlsson, C. Karamanolis, and X. Zhu, “Triage: Performance isolation and differentiation for storage systems,” in *In International Workshop on Quality of Service (IWQoS)*, 2004, pp. 67–74.

[16] N. Gandhi, D. Tilbury, Y. Diao, J. Hellerstein, and S. Parekh, “Mimo control of an apache web server: modeling and controller design,” in *American Control Conference, 2002. Proceedings of the 2002*, vol. 6, 2002, pp. 4922 – 4927 vol.6.

[17] H. C. Lim, S. Babu, and J. S. Chase, “Automated control for elastic storage,” *International Conf. on Autonomic Computing*, pp. 1–10, 2010.

[18] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant, “Automated control of multiple virtualized resources,” in *4th ACM European conf. on Computer systems*, 2009, pp. 13–26.

[19] C. Lu, T. Abdelzaber, J. Stankovic, and S. Son, “A feedback control approach for guaranteeing relative delays in web servers,” in *Real-Time Technology and Applications Symposium, 2001. Proceedings. Seventh IEEE*, 2001, pp. 51–62.

[20] “Kompics,” http://kompics.sics.se/, accessed Oct 2011.

[21] “Scala language,” http://www.scala-lang.org/, accessed Oct 2011.

[22] “EStoreSim: Elastic storage simulation framework,” https://github.com/amir343/ElasticStorage, accessed Oct 2011.

[23] B. Trushkowsky, P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, “The scads director: scaling a distributed storage system under stringent performance requirements,” in *Proceedings of the 9th USENIX conference on File and stroage technologies*, ser. FAST’11. Berkeley, CA, USA: USENIX Association, 2011.

Towards Autonomic Marketing

Carl Adams
University of Portsmouth, Portsmouth, Hampshire UK
carl.adams@port.ac.uk

Richard John Anthony
Dept. Smart Systems Technologies
The University of Greenwich
London, UK
R.J.Anthony@gre.ac.uk

Wendy Powley
School of Computing
Queen's University
Kingston, ON Canada
wendy@cs.queensu.ca

David Bell, Chris White
School of Electronics, Electrical Engineering and Computer Science,
Queen's University of Belfast,
Belfast, UK
da.bell@qub.ac.uk
Cwhite06@qub.ac.uk

Chun Wu
Mount Marty College
Division of Natural Sciences
Mount Marty College
Yankton, USA
cwu@mtmc.edu

***Abstract*— This paper explores one of the current innovation waves within computing technology, that of the application of Autonomic Computing (AC) to the marketing domain – termed "Autonomic Marketing", the result being an adaptive, highly effective marketing strategy set to significantly change target marketing and a company's relationship with customers. Marketing has often been at the forefront of business adoption and utilization of the latest computing technologies and functionality. Indeed, the marketing function is interlinked with technology and has been proactively using the capabilities of new technologies from the earliest databases and mail merge functionality to sophisticated Customer Relationship Management systems and intelligent behavioural marketing systems. The Autonomic Computing paradigm provides a framework in which marketing systems could become self-configuring and context-aware, using a variety of learning and decision-making techniques, providing the potential of even more refined targeting of marketing information to customers. In this paper, we introduce the concept of Autonomic Marketing and outline some of the research issues involved in the implementation of such a system that will, indeed revolutionize the marketing world.**

Keywords-Autonomic Computing; Autonomic Marketing; Marketing Intelligence Systems.

I. INTRODUCTION

There has always been a close relationship between information and communication technology (ICT) and marketing where marketers are often at the forefront of exploiting new technological capabilities. From the early days of using customer databases to the later multiple channel Customer Relationship Management (CRM) systems, marketers have used technology to better target marketing information to customers. Each technological wave has resulted in a step change in marketing activity and capabilities [1][2]. Technology brings marketers closer to their customers, and new technologies bring new channels and approaches to marketing [3]. Database systems enabled the recording and collation of many data items relating to customers. Every interaction between a company and a customer could be recorded and analyzed to generate customer profiles which could then be used to target specific marketing information. Similarly, Internet technologies have enabled close monitoring and interaction of a user on the Internet where preferences, behaviour, use patterns and much more can be tagged and logged [1][4]. The move towards Web 2.0 technologies bring more advanced monitoring capabilities, drawing upon a wealth of personal information [5].

Mitch [6] argued that, historically, there have been three stages of marketing initiated by technological changes, these being:

- *Research based* that used qualitative and statistical data to guess what people wanted.
- *Transaction based* that used the stored transaction data collected by organizations to help analyze and profile customers. This saw the growth of database marketing and CRM type systems.
- *Volunteered Personal Information*, a technology based on customers being active participants in providing key data, enabled by the later Web 2.0 technologies. Often the data is personal and interlinked with a host of customer context information, both personal and business information. Companies have a far richer set of data to collect, collate and analyze.

Each of the different stages has fundamentally changed marketing activity.

We propose a new marketing approach, "Autonomic Marketing" (AM), which employs the fundamentals of Autonomic Computing (AC) to monitor the current environmental state (including social trends, world events as well as characteristics of the target consumer base) and use these inputs to formulate an appropriate marketing strategy to yield the best results given the current conditions. The approach is adaptable, using feedback loops to monitor, analyze, plan and subsequently execute the new marketing plans on the fly.

The remainder of the paper is organized as follows. Section II presents the overview of Autonomic Computing. In Section III, we introduce the Autonomic Marketing concept architecture. Section IV discusses the research issues arising from the AM concept. We conclude the paper in Section V.

II. Autonomic Computing Principles

Two seminal papers, Ganek and Corbi [7] and Kephart and Chess [8], consolidated a theme within computer science - that of the practical emergence and development of sophisticated autonomic systems which have the ability to operate autonomously in remote dynamic environments with limited intervention from human operators. Ganek and Corbi discussed the ‘dawning of the Autonomic Computing era’ describing the main attributes of Autonomic Computing systems as being self managing systems with self-configuring, self-healing, self-optimizing and self-protecting capabilities. Kephart and Chess explored the grand challenges to create self-managing computing systems that can manage themselves according to an administrator's goals.

The concepts of autonomic computer systems derive from the human autonomic (vegetative) nervous systems [9], the regulatory mechanisms of visceral functions such as digestion, respiration, and the circulation of the blood, etc. These biological systems operate autonomically, that is, without conscious control by the individual.

In the past decade, many autonomic systems have been developed for self-management of complex systems. For instance, AC has been applied to database management systems [10], web services environments [11], elastic services in the cloud [12], workload management [13], and complex networks [14]. Autonomic systems have the ability to cope with dynamic environments [15]. Autonomic systems have begun to reach a level of maturity with a variety of models, applications and techniques [16][17]. A further aspect of Autonomic Computing is the ability of units and agents to interact with each other [18]. There are many potential models for interaction, such as sharing resources, sharing environment information, collaborating on a joint activity or a multitude of transaction based information sharing models.

Autonomic Computing brings together the advances in artificial intelligence, intelligent agents and autonomic, or self*, capabilities that enable real-time, contextual adaptability and learning that can be applied to marketing activities.

We use the term Autonomic Marketing (AM) to describe a step-change in the sophistication of automated marketing systems, in which the marketing activity itself is dynamically configured and contextualized to suit the current market conditions. AM can be succinctly defined as ‘it knows you want it’ technology, effectively the Holy Grail of marketing that provides previously unprecedented levels of personalization and targeting of product and service information to customers - just when they need it, or when they are likely to be most receptive to it. The technology capabilities are already here and they are beginning to be applied in this area, however, activity is piecemeal. In addition, there is little consideration of the full capabilities, and the likely consequences, of Autonomic Marketing.

III. Autonomic Marketing

The Autonomic Marketing Interest Group [19] defined some initial concepts and promise of Autonomic Marketing, that of "it knows you want it" technology or, applying it to a real-time setting, "it knows you want it, when and where you want it". Effectively, Autonomic Marketing is heading towards the marketing Holy Grail where marketers can utilize artificial intelligence capabilities to trawl through the mass of data and data channels to closely match customers’ needs, likes and preferences.

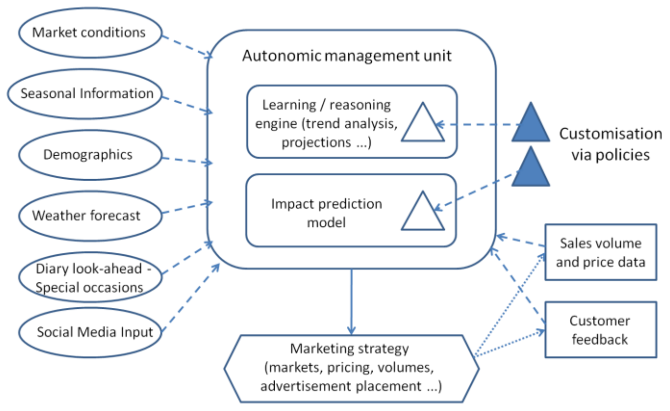

Market conditions
Seasonal Information
Demographics
Weather forecast
Diary look-ahead - Special occasions
Social Media Input
Autonomic management unit
Learning / reasoning engine (trend analysis, projections ...)
Impact prediction model
Customisation via policies
Sales volume and price data
Customer feedback
Marketing strategy (markets, pricing, volumes, advertisement placement ...)

Figure 1. The Autonomic Marketing Architecture

Figure 1 outlines the AM architecture. The autonomic management control unit monitors current state, taking input from various sources such as the market conditions, customer demographics, significant world events, trends emerging from social media analysis, weather, and seasonal information. The information is time correlated, analyzed and fed into prediction models to formulate the best possible marketing strategy. Success metrics include sales volume and customer reports, which are used as feedback to the autonomic manager. The behaviour of the autonomic manager is controlled by customizable policies.

Autonomic Marketing may take two distinct approaches: company-centric or customer-centric. Company-centric is a more targeted “information push” approach, whereas consumer-centric is more of an information pull technology. In a company-centric approach, marketing strategies are defined using data that is readily accessible and targets the population as a whole. A consumer-centric approach is one in which an individual provides context data and the system sifts through potential offerings to select items most relevant to the consumer based on the context provided.

As an example of company-centric AM, consider an organization that sells sportswear. The company wishes to run an adaptable marketing campaign on television with the ads to be aired on different networks during different (unpredictable) events. The ads feature different products, and the content is adaptable to be appealing to different audiences with different demographics. Before each ad is run, the current viewer demographics are analyzed (based on past and/or present statistics gathered by the television network), information about the current television show being aired is considered (is it a sporting event, a comedy or a reality television show?), if applicable, the current state of the show is taken into account (who is winning the game?) and social networking trends (such as fan favourites) are determined. Different ads are run depending upon the current conditions. For instance, if a football match is currently airing, and Manchester United is the fan favourite and are currently winning, Man U merchandise may be advertised and the ad may feature Man U fans celebrating a win, wearing their gear. Conversely, if FC Barcelona is currently winning, more generic, non-team related merchandise may be a better choice, or perhaps the ad should not be run at that time. The audience demographics may further refine the merchandise that is shown and the type of ads that are aired at any particular time.

Alternatively, in a consumer-centric approach, the system would gather customer profile data (demographics and preferences, if available), prompt the customer for some additional information and then sift through offerings to identify products that may be of interest. Predictive models would provide guidance as to what products would be displayed. For example, the system may know that the customer is 20 years of age, is located in Valencia, Spain and knows, from previous searches, that the customer is a FC Barcelona fan. The system may prompt for more information, such as the fact that the customer is looking for a jacket. It would then search for items from a variety of companies and show the customer the items he/she is most likely to purchase. Predictive models would base decisions based on historical data (companies that the consumer has purchased from previously, or companies frequented by customers with similar demographics etc).

The way in which the various stages of the autonomic control loop {*monitor, analyze, plan, execute*} map onto such use cases can be explained in terms of Figure 1. '*Monitor*' current market conditions, and factors directly influencing this including diverse information such as weather, seasonal, economic, as well as customer feedback (direct via reviews etc, and also indirect in terms of sales figures) and current pricing model; provide the current sensed 'state'. '*Analyze'* takes this information, combines it with historical data, and searches for patterns / trends and thus identifies opportunities to improve the system's performance (ultimately to increase sales value or volume). '*Plan*' decides what changes could be made to the current marketing strategy and attempts to predict their impact. This could use a utility function or fuzzy reasoning, for example, to determine which of several possible adaptations yields the most beneficial results under the various conditions. '*Execute*' subsequently applies the changes by adjusting tuning parameters on the marketing strategy.

IV. RESEARCH ISSUES IN AUTONOMIC MARKETING

Although the principles of Autonomic Computing are well developed, the application of these principles to new domains remains challenging and a number of issues will need to be researched and resolved in order to make AM a reality. Infrastructure must be put in place to collect, store, analyze and mine the data sources, adaptable models must be built to predict the impacts of taking particular actions, and feedback mechanisms must be put in place.

AM requires data from many data sources – contextual information from the potential consumer base (demographics, geographical information, preferences etc), trends emerging from social networking sites or significant world events from news sources, weather/seasonal information, market conditions, historical information, etc. How and, from where, will this information be collected? What exactly is contextual data? What tools are in place for collecting this data? Who owns this data, and how do we deal with the privacy issues? Will the data be stored? If so, how, where, and for how long? Can the data be collected and analyzed efficiently to allow for real-time AM? Is all the data time-stamped to allow for time-correlation? Most organizations already collect part of this information, but it is held in disparate systems (e.g. for supply-chain management, logistics, stock control, and traditional marketing), and not used effectively in the way we propose. AM is to work smarter, not harder.

The system must be able to predict the outcome of applying AM strategies. How do we build and test the predictive models? These models must be adaptable and updated regularly, based on obtained feedback. How will we detect that the models are outdated and how often should they be updated? As with most current AC systems, a human supervisor is still required for such high-level decisions. The key goal of AM is to manage the complexity of making targeted management actions with low latency, freeing up the marketing executives to focus on only the highest-level concerns. Ultimately, when dealing with situations that have not been met before, or when new entities are encountered in the application environment, plasticity and flexibility must be greatly enhanced beyond the present state of the art. In this way restrictions arising due to having only the pre-programmed functionality familiar in current, predominantly 'playback', systems, can be overcome when dealing with novel situations [20].

The customer- & company-centric approaches are quite different, with one being an information push based on current knowledge and the other relying on gathering information from the customer in order to find relevant information. In terms of technology, what are the requirements of each, and how do they differ?

Along with the technological aspects of AM, we need to consider the implications of failure. AC systems take action based on current conditions. They are unpredictable as it is impossible to test all possible cases, especially with the large number of parameters involved in the AM system. If a marketing strategy employed by the AM fails, what are the implications? How can trust be established?

Some of the questions posed above have been studied extensively in the literature and solutions have been proposed or are already in widespread use. For example, the storage, correlation and analysis of massive amounts of data using data warehouses and/or open source tools such as HBase [21], Hadoop [22] and Mahout [23]. Other areas, however, require additional research. Our further work is to prototype a system and to pursue some of the open areas.

V. CONCLUSIONS AND FUTURE WORK

Autonomic Marketing, making use of the powerful capabilities that Autonomic Computing provides, offers many significant changes in marketing activity. As with any new wave of technology, the impact could be far reaching for customers and companies. Marketing budgets may be significantly reduced as large numbers of marketers are replaced by intelligent Autonomic Marketing agents that operate 24/7 providing increasingly accurate targeting and ad customization.

There are clearly challenges ahead, but Autonomic Computing is a key area in computer science with over 10 years of development of principles and techniques. We foresee that a natural extension of this science to marketing, a complex "system" that requires self-configuration and self-optimization, is the next technological wave that will sweep the industry.

REFERENCES

[1] C.F. Hofacker, Internet Marketing (3rd Ed). Chichester: Wiley, 2001

[2] W. Rowan, Digital Marketing: Using New Technologies to Get Closer to Your Customers. London : Kogan pp. 2002.

[3] A.J. Kimmel, Marketing Communication: New Approaches, Technologies, and Styles. Oxford University Press, 2005.

[4] P. Martin, M. Matheson, J. Lo, J. Ng, D. Tan, and B. Thomson, B., Supporting Smart Interactions with Predictive Analytics. SITCON: The CAS/NSERC Strategic Workshop in Smart Internet Technologies. Ontario, Canada. 2010.

[5] S. Agresta, and B.B. Bough, Perspectives on Social Media Marketing: The Agency Perspective/The Brand Perspective. Boston: Course Technology, 2011.

[6] A. Mitch, "The Rise of Volunteered Personal Information", Journal of Direct, Data and Digital Marketing Practice, 12, pp. 154-164, 2010.

[7] A. G. Ganek, and T.A. Corbi, "The Dawning of the Autonomic Computing Era". IBM Systems Journal, March, 2003.

[8] J.O. Kephart, and D.M. Chess, "The Vision of Autonomic Computing". Computer IEEE 36(1), 2003, pp. 42-50.

[9] E.H. Ackerknecht, "The History of the Discovery of the Vegatative (Autonomic) Nervous System". Med Hist. January; Vol. 18, no.1, 1974, pp. 1–8.

[10] S. Elnaffar, W. Powley, D. Benoit and P. Martin, "Today's DBMSs: How *Autonomic* Are They?", 1st International Workshop on Autonomic Computing Systems (DEXA 03). May 2003.

[11] W. Tian, F. Zulkernine, J. Zebedee, W. Powley and P. Martin. "An Architecture for an Autonomic Web Services Environment", Proceedings of the Joint Workshop on Web Services and Model-Driven Enterprise Information Systems WSMDEIS (ICEIS 2005), May 2005, pp. 54-66.

[12] P. Martin, A. Brown, W. Powley, J.L. Vazquez-Poletti, "Autonomic Management of Elastic Services in the Cloud", Workshop on Management of Cloud Systems (MoCS 2011), June 28, 2011, pp. 135-140.

[13] B. Niu, P. Martin, W. Powley, "Towards Autonomic Workload Management in DBMSs", *Journal of Database Management*, 20(3), July - Sept 2009, pp. 1-17.

[14] L. W. Russell, S.P. Morgan, and E.G. Chron, "Clockwork: A New Movement in Autonomic Systems". IBM Systems Journal, vol 42, no1, 2003, pp. 77-84.

[15] C. Adams, "Autonomic Systems, Coping Strategies and Dream Functions". ICAS 2007, The Third International Conference on Autonomic and Autonomous Systems, June 19-25, 2007.

[16] K. Amina, A. Haye, M. Jahan, and S. Shamail, "Survey of Frameworks, Architectures and Techniques in Autonomic Computing", Proceedings of the Fifth International Conference on Autonomic and Autonomous Systems, 2009, pp. 220-225.

[17] H. Psaier, and S. Dustdar, "A Survey on Self-Healing Systems: Approaches and Systems. Computing 91(1), 2011, pp. 43-73.

[18] C. Adams, "Collaboration Within the IoT: A Self-Conscious Approach for Autonomic Units". Internet of Things Workshop, December 3rd, 2011.

[19] AMIG (2011) Principles of Autonomic Marketing. Autonomic Marketing Interest Group (Adams C., Anthony R.J.., Bell D., Powley W., White C. and Wu C.), International Conference of Autonomic and Autonomous Systems (ICAS 2011), Venice May 2011.

[20] C. White, D. Bell, "Towards the Measurement of Plasticity and Innateness in Artificial Agents". AISB 2011, The 2nd Towards a Comprehensive Intelligence Test (TCIT), Reconsidering the Turing Test for the 21st Century Symposium, York, UK, April 4-7, 2011.

[21] Apache HBase. http://hbase.apache.org

[22] Apache Hadoop. http://hadoop.apache.org

[23] Apache Mahout. http://mahout.apache.org/

A Dynamic Load Balancing Model Based on Negative Feedback and Exponential Smoothing Estimation

Di Yuan, Shuai Wang, Xinya Sun
Tsinghua University
Beijing, 100084, China
{yuan-d09, wangshuai04}@mails.tsinghua.edu.cn, xinyasun@tsinghua.edu.cn

***Abstract*—Server clusters can be used to manage the massive number of requests that a hot website will receive, so as to meet the rapid development of Internet application. The Linux Virtual Server provides a good solution for cluster revision, and there is software that can be used for management and monitoring. However, the scheduling algorithms of Linux Virtual Server are not sufficient to deal with the heavy load balancing required today. A dynamic load balancing scheduling algorithm has been proposed to solve the problems of static algorithms, but we find that there are some drawbacks in actual use. In this paper, we suggest an improved dynamic load balancing model that overcomes the limits or drawbacks of the simple dynamic algorithm. In the suggested model, negative feedback and exponential smoothing estimation methods have been used to improve the load balancing effect. Besides, service response time has been used to adjust the weight variation to achieve better effect. The suggested model is implemented in our dynamic load balancing algorithm. Experiments show that, our algorithm can achieve better performance than the existing static and dynamic algorithms.**

Keywords-load balancing; dynamic algorithm; negative feedback; exponential smoothing estimation; throughput

I. INTRODUCTION

With the rapid development of the Internet, hot web sites must cope with greater demands than before. Increasing number of users or clients makes a single server not sufficient to handle this aggressively increasing load. As a result, we ought to use a server cluster to solve this problem. A server cluster can help to keep computing service in good quality by adjusting the server nodes dynamically when the number of requests suddenly changes. However, we should find a method to assign the new connections to the computing elements properly.

Server cluster can be built with either expensive hardware as F5 load balancer, or Linux Virtual Server (LVS) [1]. LVS is a good solution to companies for cost factors. Generally, the LVS is a software tool assigns connections to multiple servers, which can be used to build highly scalable and highly available services. An LVS cluster is composed by the load balancer and real server nodes. The load balancer receives requests and schedules them to real servers following certain rules [2].

The LVS clusters are always built by Direct Routing method, because load balancer is independent from OS and the load balancer's burden is less than server nodes [3]. LVS has ten scheduling algorithms [4]. The WLC algorithm, which schedules the new connections according to servers' weights and number of active connections, is most commonly used for its good balancing performance [3]. However, it is usually difficult to locate proper weight to a server, and the weight can only be adjusted manually while LVS is running. Moreover, if the requests vary in their processing time or package size, the workload of servers will be skewed.

A basic dynamic load balancing algorithm based on negative feedback has been proposed [5]. Daemon tools like *Keepalived* or *HeartBeat* can be used to manage server nodes. The load balancer collects load information of a server node, which can be used to update its weight through the Simple Network Management Protocol (SNMP). Aggregated load of a server node can be calculated by load information, and new weight of the server node can be solved by

$$W_i = W_{i-1} + W_{\text{step}} \sqrt[3]{A - \text{Aggregate\_Load}_i} \tag{1}$$

In the above equation, 'W_{step}' denotes the step of weight adjustment, while 'A' denotes the expected value of the aggregate load. Through analysis and experiments, we have found this dynamic algorithm have drawbacks in actual use. Firstly, the current weight 'W_i' only relies on 'W_{i-1}' and the current aggregate load. Once the collection or calculation of load is interfered, the adjustment of weight may not reflect actual variation of load. Secondly, the response time of the server, an important factor of server's load, is aggregated to the 'Aggregate_Load', which may undermine its importance. Lastly, no matter how much the variation of aggregate load is, the upper bound of the weight adjustment is 'W_{step}'. As a result, the update of the weight has limitations, which may affect the load balancing effect.

In this paper, we suggest a new load balancing model to improve the load balancing performance. We set up a cluster system with the characteristics of high availability and high reliability based on LVS and open source software *Keepalived*, in order to implement our model and algorithm. The load balancer checks server nodes by using *Keepalived*, and collects the real-time load information through a user-defined monitoring module. Then new weights of the server nodes are calculated through weight evaluation module with the load information and updated into Linux kernel. The load balancer assigns new connections by using weighted scheduling algorithm of LVS according to new weights [6]. In weight evaluation module, we collect load information of the server nodes and evaluate the aggregated load. Besides, the module detects response time from each server node to correct 'W_{step}'. Furthermore, the module calculates weight estimation through exponential smoothing method, the purpose of which is to make the adjustment of weight consistent with the actual variation of load. We suggest an improved dynamic load balancing algorithm based on improvements above and do experiments through open source software *Apache JMeter* [7]. The new algorithm shows better result of balancing effect than the existed WLC algorithm and simple dynamic algorithm above.

The remainder of this paper is organized as follows: Section II is a focus of this paper, our dynamic load balancing model is described. In Section III, framework and flow of corresponding algorithm is presented. In Section IV, experiments of three algorithms are done to compare the balancing performance. In Section V, some conclusions are drawn through the experiments.

II. DYNAMIC LOAD BALANCING

A. *Negative Feedback Model of Dynamic Load Balancing*

The WLC algorithm schedules the new connections according to weights and number of active connections. As the former factor is static during the scheduling process, WLC is essentially a static scheduling algorithm [8]. By contrast, our dynamic load balancing algorithm schedules the connections according to both active connections number and load information. The load balancer sends request to server nodes to get load information, and then the weight evaluation module calculates new weight according to former weights and aggregated load. The load balancer schedules the new connections from client to server nodes according to new weights. It is obvious that the dynamic load balancing is a negative feedback process.

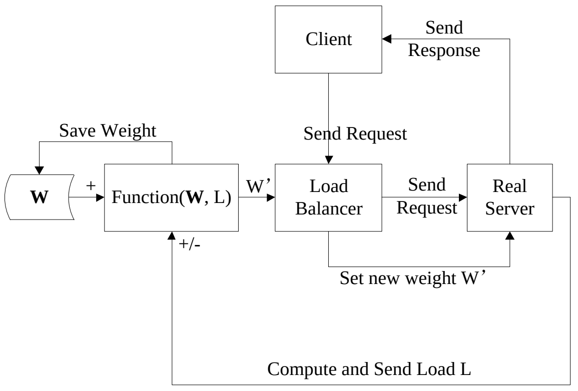


Figure 1. Negative feedback model of dynamic load balancing

In order to compute the weight of the server node, load information collection service is running in each server node. The load balancer collects load information 'L' periodically. Weight evaluation module 'Function(**W**, L)' calculates the new weight by former weight vector '**W**' and aggregated load 'L' and then update the IPVS scheduling table. This dynamic algorithm can overcome the drawbacks of WLC, and the effect of load balancing will be enhanced [9].

B. *Weight Evaluation Module*

As we discussed above, the weight evaluation module of load balancer is an important part of this dynamic algorithm. The load information can be used to calculate new weight of the server.

Assume vector $\mathbf{L}=[L_1, L_2, L_3, L_4]$, ($L_i<1$) denotes the load parameters and vector $\mathbf{Q}=[Q_1, Q_2, Q_3, Q_4]$ denotes proportionality factor of the parameters, where L_1 to L_4 represent CPU usage rate, memory usage rate, file system usage rate, and one server's new connections proportion of the total number. Thus, $0<\mathbf{QL}^{\mathrm{T}}<1$, the aggregated load parameter, denotes the current load of a server node [10]. Further, we assume that T_{delay} denotes the real response time of the computing service and T_{ideal} denotes the ideal value of T_{delay}, then the ratio of them may represent the current network state between load balancer and the server node.

The basic dynamic load balancing model assumes that, current weight of a server node is only related to the former one [5]. Considering the fact that the aggregated load may be interfered, this mechanism may lead to deviation of the weight calculation. Our improved dynamic algorithm solves this problem through three former weights. Assume vector $\mathbf{W} = [W_{i-2}, W_{i-1}, W_i]$ denotes weights before time i+1 and vector $\mathbf{P} = [P_1, P_2, P_3]$ denotes proportionality factor of the weights. As we discussed above, estimated weight at time i+1 should be used to adjust the 'W_{step}', in order to make the step more proper. Assume 'A' denotes the expected value of the aggregate load, the weight update formula is

$$W_{i+1} = \mathbf{PW}^{\mathrm{T}} + W_{\text{step}} \left(\frac{\hat{W}_{i+1}}{\mathbf{PW}^{\mathrm{T}}}\right)\left(\frac{T_{\text{ideal}}}{T_{\text{delay}}}\right)^{\operatorname{sgn}\left(A-\mathbf{QL}^{\mathrm{T}}\right)} \sqrt[3]{A-\mathbf{QL}^{\mathrm{T}}} \qquad (2)$$

For each server node, *Keepalived* may set its weight from 1 to 253 [6]. We can set the weight range $[w_0, 10w_0]$, ($0<w_0<25$) for simplicity. The ideal service response delay T_{ideal} can be estimated through experiments. In order to properly reflect the impact of service response time, we set $T_{\text{ideal}}<T_{\text{delay}}<1.5T_{\text{ideal}}$. If the aggregated load $\mathbf{QL}^{\mathrm{T}}$ is greater than A, weight adjustment and T_{delay} is proportional, and vice versa. W_{step} and A are two important parameters. Generally, we set $W_{\text{step}}= w_0/2$ and $0.45<A<0.95$. As there must be some differences between different cluster systems, the exact value of them should be determined through experiments [9]. We set W_0 the reference value of W_1 to W_3, and then the complete weight evaluation module is

$$W_{i+1} = \begin{cases} W_0 + W_{\text{step}} \dfrac{T_{\text{delay}}}{T_{\text{ideal}}} \sqrt[3]{A-\mathbf{QL}^{\mathrm{T}}} & i<3 \\ \mathbf{PW}^{\mathrm{T}} + W_{\text{step}} \left(\dfrac{\hat{W}_{i+1}}{\mathbf{PW}^{\mathrm{T}}}\right)\left(\dfrac{T_{\text{ideal}}}{T_{\text{delay}}}\right)^{\operatorname{sgn}\left(A-\mathbf{QL}^{\mathrm{T}}\right)} \sqrt[3]{A-\mathbf{QL}^{\mathrm{T}}} & i \ge 3 \end{cases} \qquad (3)$$

C. *Exponential Smoothing Estimation*

As discussed above, new weight should be estimated to adjust 'W_{step}' to make the adjustment of weight consistent with the actual variation of load. We estimate the new weight through linear quadratic exponential smoothing method, the essence of which is to get the estimation result through the weighted average of historical data. According to exponential smoothing theory, time series trend with stability or regularity, so they can be reasonably extended to estimate the future trend [11]. Exponential smoothing method, mainly used for variable parameter linear trend time series, may estimate the current value according to the historical ones, which could be helpful to weight estimation.

Assume the true value of the weight at time t is W_t. Besides, assume that the first and second exponential smoothing result is $S_t^{(1)}$ and $S_t^{(2)}$, the smoothing factor is a and the estimation cycle from time t is 1, then the estimation formula is shown below [12].

$$\begin{aligned} S_t^{(1)} &= aW_t + (1-a)S_{t-1}^{(1)} \\ S_t^{(2)} &= aS_t^{(1)} + (1-a)S_{t-1}^{(2)} \\ \hat{W}_{t+1} &= \frac{2-a}{1-a}S_t^{(1)} - \frac{1}{1-a}S_t^{(2)} \end{aligned} \tag{4}$$

III. IMPLEMENTATION OF IMPROVED DYNAMIC LOAD BALANCING ALGORITHM

A. *Framework of Improved Dynamic Algorithm*

The improved dynamic load balancing algorithm we suggest is based on the WLC scheduling algorithm. Besides, *Keepalived* has been used to implement health checking and weight update of server nodes. To be specific, the MISC_CHECK module of *Keepalived* allows a user-defined script or executable program to run as the health checker [6]. The exit code of the script or program can be used to update the LVS scheduling table. If the exit code is 0, weight of a server node remains unchanged. Computing service of a server node is unavailable when the exit code is 1. In addition to the two cases, the weight of a server node will be set to 'exit code-2' when the range of exit code is 2-255 [6]. According to this idea, we can achieve our dynamic load balancing algorithm through MISC_CHECK module of *Keepalived*. Each modules of this dynamic algorithm is shown in Figure 2.

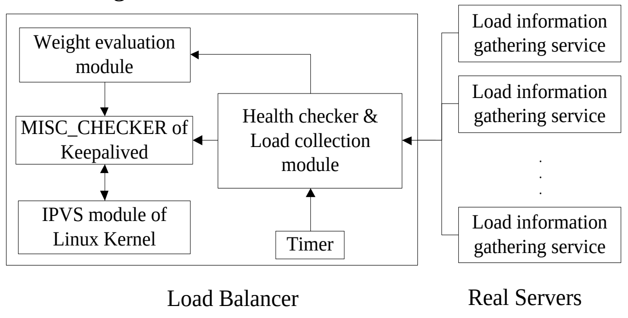

Weight evaluation module
MISC_CHECKER of Keepalived
IPVS module of Linux Kernel
Health checker & Load collection module
Timer
Load information gathering service
Load information gathering service
Load information gathering service
Load Balancer
Real Servers

Figure 2. Modules of improved dynamic algorithm

The user defined module get the health status and the load information periodically. Then the weight evaluation module calculates new weight of the server node. The LVS scheduling table is refreshing through MISC_CHECKER module. If the health checking fails, the module will remove the server node from the server pool automatically. Else, the new weight of a server node will be update to the one calculated by weight evaluation module.

B. *Flow of Improved Dynamic Algorithm*

The load balancer collects load information periodically, so load gathering service should be running real-time. We collect load information above through some system files of Linux. The load balancer and the real server exchange load information by using the client and server communication mechanism, which are both user-defined. We set the program *mon_srv* running in the server node to gather and send load information to the load balancer. The program *mon_cli*, running on the load balancer, sends request to get load information periodically.

Computing service is also running real-time on the server nodes. The load balancer checks the health of a server node's computing service firstly by using *mon_cli*, and then gets the server response time if the server is health. Then the balancer gets load information from the load information gathering service. The load balancer checks the computing service by using TCP connection. If the service is healthy, the load balancer checks the scheduling table to check whether the node exists or not, and then get the response time. Else, the load balancer removes the node from the scheduling table and set the weight of the node to 0. After that, the load balancer collects load information from the server node by using UDP connection. Then the weight evaluation module calculates new weight of the server node by using our weight evaluation model. Finally, new weight of the server is updated by *Keepalived*. The flow chart of the algorithm is shown in Figure 3.

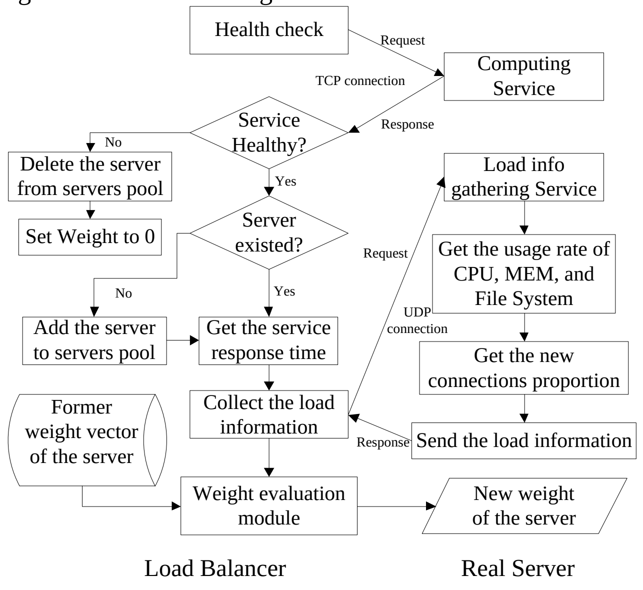

Health check
Request
TCP connection
Computing Service
Response
Service Healthy?
No
Delete the server from servers pool
Set Weight to 0
Yes
Server existed?
No
Yes
Add the server to servers pool
Get the service response time
Collect the load information
Former weight vector of the server
Weight evaluation module
New weight of the server
Load info gathering Service
Request
Get the usage rate of CPU, MEM, and File System
UDP connection
Get the new connections proportion
Response
Send the load information
Load Balancer
Real Server

Figure 3. Flow chart of improved dynamic algorithm

IV. EXPERIMENTS

In order to test the performance of improved dynamic load balancing algorithm, we have set up an experiment environment, on which WLC algorithm, simple dynamic load balancing algorithm and our improved dynamic algorithm has been implemented.

A. Hardware Environment

In our experiments, 6 blade servers and 2 industrial computers has been used, among which 1 blade server serves as the client, 2 blade servers serve as load balancers, and the other 3 blade servers serve as server nodes together with the 2 industrial computer. The 6 blade servers use one switch, while the 8 devices use one. The hardware and OS parameters of the 8 devices are shown in Table I.

TABLE I. CONFIGURATION PARAMETERS OF HARDWARE

Parameters / Hosts	CPU (Core/GHz)	MEM (GB)	Storage (GB/R)	OS
Client	16/2.40	8	320/7200	Win 2008
Load Balancer(M)	16/2.40	8	320/7200	SUSE 11
Load Balancer(B)	8/2.40	8	320/7200	SUSE 11
Real Server 1	16/2.40	8	320/7200	SUSE 11
Real Server 2	8/2.40	8	320/7200	SUSE 11
Real Server 3	8/2.40	8	320/7200	SUSE 11
Real Server 4	4/2.26	2	160/5400	SUSE 11
Real Server 5	2/2.50	4	120/5400	SUSE 11

Hardware devices' configuration parameters are shown in Table 1. There are two load balancers to implement failover through VRRP, both of which have the same LVS configuration, virtual IP address, and *Keepalived* configuration [6]. Topological relations between the devices above are shown in Figure 4.

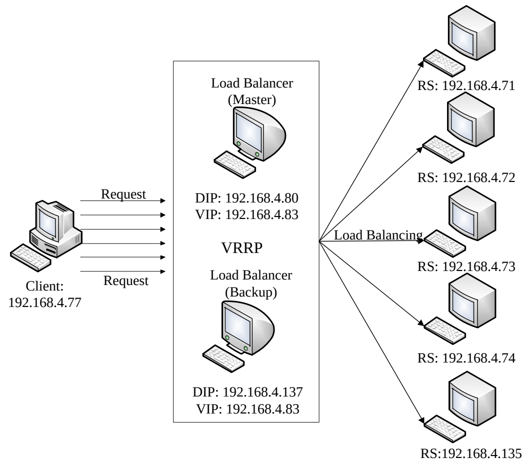


Figure 4. Topological relations between the devices

B. Software Environment

The operating systems of the devices have been shown in Table 1. Each server node supplies the same computing services and we choose five of them to do our experiments. The request processing time and the result data packet size of each service is shown in Table II.

TABLE II. EFFICIENCY AND PACKAGE SIZE OF COMPUTING SERVICES

Service Name	Request Processing Time (ms)	Size of Data Packet (KB)
te_tl_ksp_radix_multi	20	30
te_ts_ksp_radix_multi	10	20
te_ts_sp_bidij_buckets	40	35
te_ae_ksp_astar_heap_multi	60	40
price_svc	1	2

The '*price_svc*' service is the most efficient among these services, while the '*te_ae_ksp_astar_heap_multi*' service is the least. It's important to point out that, the two indexes are the average result of 430 test samples, and the latter one has more impact on the network load.

The computing services are compiled by GCC, while the client program is Java application, which can be generated to JAR file for testing. The open source software Apache JMeter has been used to simulate multiple clients, which could send requests to the server nodes. JMeter is an Apache top level project that can be used as a load testing tool for analyzing and measuring the performance of a variety of services. The concurrent test is implemented by using multi-threaded method [13]. Our experiments are performance test with Java application and the concurrent test and analyzing function of JMeter can meet our requirements.

A client and a server node communicate with each other through a TCP connection. The client should do login and authentication after a connection is established. Then the client send request data package with specific service name and parameters through the socket instance. After the login and authentication process, multiple requests can be send to the server node until the connection is closed. For each request, the service is chosen randomly in our experiments.

C. Content of Experiments

The test objects of our experiments is the original WLC scheduling algorithm, simple dynamic scheduling algorithm based on WLC, and improved dynamic scheduling algorithm based on WLC. The three algorithms can be recorded as WLC, DWLC and IDWLC for convenience. We use *JMeter* as the test and analysis tool and the test index is the throughput of the system shown in Figure 4.

The cycle index of the threads group in *JMeter* should be set to a constant. For each cycle, there are three parameters to adjust, which are number of concurrent connections, the ramp-up period of the concurrent threads, and number of requests per connection. We denote the three parameters as P, Q, and R. We study the system throughput variation tendency when parameters P, Q, and R changes, and then draw some conclusions of three algorithms through analysis.

D. Results and Analysis

In order to test the performance of three algorithms, we set the cycle index to 10 in each experiment, so as to make the test closer to the real situation. For the WLC algorithm, we set the weights of real server nodes in Figure 4 to 50, 50, 50, 40, and 40. For the DWLC algorithm, we set the original weights of real server nodes the same to 50 and the expected value of the aggregate load to 0.70.

Then we determine the parameters of the IDWLC algorithm. Firstly, the ideal service response time and the true time should be determined through lots of experiments. As the average server response time is greater than 0.2 milliseconds in our 100000 experiments, we set the T_{ideal} to 0.2 milliseconds. Through the data analysis, we find that if the computing service is healthy, the range of response time will be 0.20-0.35 milliseconds. In order to set the parameter more properly, we set the upper bound of T_{delay} to 0.35 milliseconds. Then, we set the value of other important parameters. As the weight value range of LVS server node is [0, 253], we set w_0=20 for convenience. Considering the importance of former weights and load parameters, the vector **P** is set to [0.2, 0.3, 0.5], **Q** is set to [0.4, 0.2, 0.1, 0.3]. Through some experiments, we set expected value of the aggregate load A to 0.7, and initial weight W_0 to 50. According to the exponential smoothing prediction theory, the greater the fluctuation range of predicted target is, the more the predicted value depends on the true value of the previous moment [11]. As a result, the value of smooth factor 'a' ought to be greater. Considering the weight sequence variation, we set a=0.6. The weight prediction and adjustment formula of our experiments is

$$\begin{aligned} S_i^{(1)} &= 0.6W_i + 0.4S_{i-1}^{(1)} \\ S_i^{(2)} &= 0.6S_i^{(1)} + 0.4S_{i-1}^{(2)} \\ \hat{W}_{i+1} &= 3.5S_i^{(1)} - 2.5S_i^{(2)} \end{aligned} \tag{5}$$

$$W_{i+1} = \begin{cases} 50 + 10 \times \dfrac{0.2}{T_{\text{delay}}} \sqrt[3]{0.70 - \mathbf{QL}^{\mathrm{T}}} & i < 3 \\ \mathbf{PW}^{\mathrm{T}} + 10 \times \dfrac{\hat{W}_{i+1}}{\mathbf{PW}^{\mathrm{T}}} \left(\dfrac{0.2}{T_{\text{delay}}} \right)^{\operatorname{sgn}(0.70-\mathbf{QL}^{\mathrm{T}})} \sqrt[3]{0.70 - \mathbf{QL}^{\mathrm{T}}} & i \geq 3 \end{cases} \tag{6}$$

1) System throughput T and concurrent connections number P: We set the ramp-up period of the connections to 10 seconds, requests number per connection to 50. When concurrent connections number changes from 50 to 5000, throughput curves are shown in Figure 5.

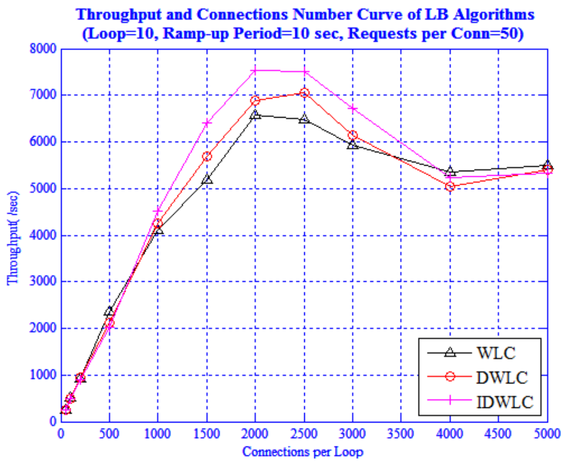


Figure 5. Throughput and connections number curves of three algorithms

As shown in Figure 5, when concurrent connections number is smaller than 1000, difference among throughputs of the system under three algorithms is insignificant. The reason is that, when the connections number is small, the processing ability of real server nodes is enough to handle the requests from the client, so the dynamic algorithms' balancing effect is no better than the WLC's. When concurrent connections number is greater than 1000, load of the server nodes increases, and the dynamic algorithms come into play and assigns the new connections more balanced. It is shown in Figure 5 that, when the connections number is greater than 1000 and smaller than 3500, the two dynamic algorithms achieve greater throughput than the WLC algorithm. Especially, our dynamic load balancing algorithm achieved greater throughput than the other two. When the number of connections is greater than 3500, the load of the server nodes is greater than the processing ability of them, and the effect of dynamic algorithm becomes insignificant compared with the WLC algorithm. As a result, system throughput of the three algorithms tends to be close to each other.

2) System throughput T and threads ramp-up period Q: We set the number of concurrent connections to 2500 and the number of requests per connection to 50. When the ramp-up period changes from 1 second to 40 seconds, throughput curves are shown in Figure 6.

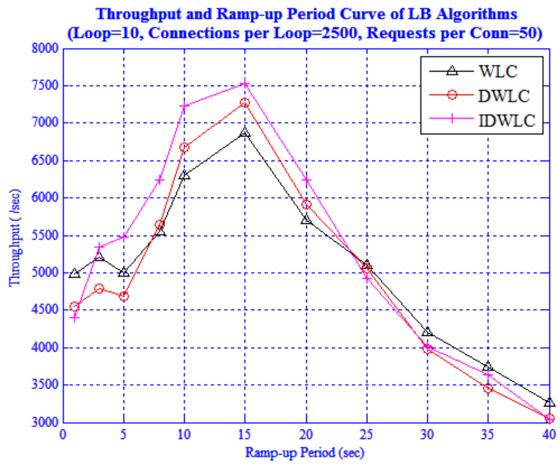


Figure 6. Throughput and ramp-up period curves of three algorithms

This experiment set the number of connections and number of requests constant, so the total load per cycle is constant too. As shown in Figure 6, when the ramp-up period is small, system throughput of three algorithms is low. The reason is that, the server nodes are under excessive load during the ramp-up period, which leads to queuing or waiting phenomenon and decreases the system throughput. When the ramp-up period increases, the queuing or waiting phenomenon has been alleviated. As a result, the two dynamic algorithms achieve greater throughput than the WLC algorithm and our improved dynamic algorithm shows better result than the simple one. When the ramp-up period is greater than 20 seconds, the assignment of new connections is sparse, and the load of the server nodes gets smaller, which lead to almost the same throughput for the three algorithms.

3) System throughput T and requests number R per connection: We set the number of concurrent connection to 2500 and the ramp-up period to 10 seconds. When the requests number per connection changes from 10 to 160, throughput curves are shown in Figure 7.

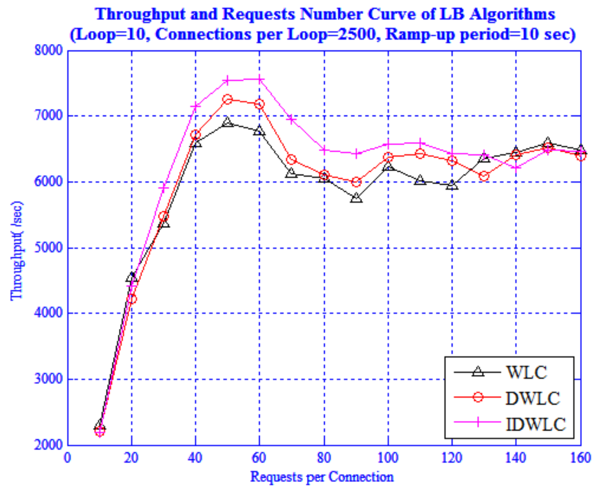


Figure 7. Throughput and requests number curves of three algorithms

This experiment set the connections number and ramp-up period to constant. When the requests per connection is smaller than 20, the load of server nodes is low. As a result, the effect of two dynamic algorithms is not better than the WLC algorithm. With the increase of the requests number, dynamic algorithms could assign new connections more proper, system throughput gets greater than the WLC algorithm. We can find in Figure 7 that, our improved algorithm achieves greater throughput than the other algorithms when the requests number changes from 30 to 120. As the requests number gets greater than 120, the total load is too heavy to the server nodes, which could lead to similar system throughput for the three algorithms.

V. CONCLUSION

As described above, a static load balancing algorithm is not sufficient to assign client connections when processing time requests vary, and thus the scheduling programs of the Linux Virtual Server are not useful [5]. A dynamic load balancing algorithm has proposed before to solve this problem. However, the algorithm has some problems, which may reduce its usefulness, and thus we propose a more efficient load balancing algorithm that achieves better results.

The improved model we suggest could solve the shortcoming of the simple dynamic algorithm and improve the stability of the dynamic scheduling process. For one thing, computing service response time has been used to adjust the weight variation, aims to highlight the important role of the network delay for load balancing. For another, the exponential smoothing estimation method has been used to make the adjustment of weight consistent with the actual variation of load. The experimental results show that, our improved dynamic load balancing algorithm could achieve greater system performance than the other two, if the total load is proper to the real server nodes.

ACKNOWLEDGMENT

This work was supported in part by the National Key Technology R&D Program under Grant 2009BAG12A08 and the Research Foundation of Easyway Company.

REFERENCES

[1] Linux Virtual Server Project, “Linux Virtual Server, ” 2007, http://www.linuxvirtualserver.org

[2] Zhang Wensong, “Linux Virtual Server for Scalable Network Services,” Ottawa Linux Symposium 2000, 2000(7)

[3] Suntae Hwang, Naksoo Jung, “Dynamic Scheduling of Web Server Cluster,” Proc. Ninth Int’l Conf. Parallel and Distributed Computer Systems (ICPADS ‘02), 2002

[4] Zhang Wensong, “Job Scheduling Algorithms in Linux Virtual Server,” 2005, http://www.linuxvirtualserver.org

[5] Zhang Wensong, “Dynamic Feedback Load Balancing Algorithm”, 2005, http://zh.linuxvirtualserver.org.

[6] Alexandre Cassen, “Keepalived for LVS datasheet”, 2002, http://www.keepalived.org/pdf/UserGuide.pdf

[7] Ma Wei, “A New Approach to Load Balancing Algorithm in LVS Cluster,” Master Degree thesis. Wuhan, Hubei, China: Hua Zhong Normal University. 2006.5.

[8] Shen Wei, “Research and Realization of an Improved Load Balancing Algorithm based on LVS Cluster,” Master Degree thesis. Bejing, China: China University of Geosciences. 2010.6.

[9] Qin Liu, Lan Julong. Design and Implementation of Dynamic Load Balancing in LVS. Computer Technology and Its Applications, 2007, 09:116-119.

[10] Yang Jianhua, Jin Di. A Method of Measuring the Performance of A Cluster-based Linux Web Server. Computer Development & Applications, 2006, 04: 58-60.

[11] Diao Mingbi, Theoretical Statistics. Beingjing: Publishing House of Electronics Industry. 1998.

[12] Luo Bin, Ye Shiwei, Server Performance Prediction using Recurrent Neural Network. Computer Engineering and Design, 2005,08: 2158-2160

[13] The Apache Jakarta Project, Apache JMeter, 1999-2011, http://jakarta.apache.org/jmeter

Interactive Rendering of Huge 3D Meshes in Cloud Computing

Daeyoung Kim and Haeyoung Lee
Computer Engineering Dept.
Hongik University
Seoul, Korea
dykim99@gmail.com, leeh@hongik.ac.kr

***Abstract* — This paper presents a new hierarchical representation of huge 3D meshes for fast and seamless rendering in cloud computing. Shape-outlines, simplified meshes, and uniform mesh partitions construct a hierarchy. Our hierarchy enables on-demand rendering of huge 3D meshes in cloud computing.**

Keywords-cloud computing, 3D meshes, interactive rendering, hiearchical representation.

I. INTRODUCTION

Rapid advances in 3D scanning technologies now enable us to create huge and exquisite 3D meshes for medical imaging and cultural heritage preservation. Nevertheless, it is hard and even impossible to render huge meshes on consumer computers and mobile devices due to limited resources. Various techniques such as mesh compression [1], [2], simplification [3], [4] and chartification [2] allow the transfer and display of huge meshes on mobile devices; however, interactive rendering in real time is hard to achieve using these methods. Though image-based rendering [5] has recently been introduced, pre-rendered images and grid-based sparse meshes cannot provide detailed views of original meshes. Moreover, these methods do not allow for the control of file size. Advances in CPU-related technologies have dramatically decreased CPU processing times so that I/O time contributes to almost the entire processing time. Uniformly sized files optimize I/O processing time and are especially necessary for mobile computing.

In this paper, we present an interactive rendering method of a hierarchical data structure for huge meshes for cloud computing platforms. Moreover, with our method the file size for each of a series of files for hierarchical data structures can be uniformly controlled for optimized and predictable I/O processing time.

The remainder of the paper is organized as follows: the basics of hierarchical rendering are described in detail in Section II; uniform mesh partitioning and simplifications are explained in Section II, parts A and B; our interactive view modes are listed in Section II, parts C through E; our conclusions and future work are presented in Section III.

II. HIERARCHICAL 3D MESH RENDERING

New hierarchical representations of huge 3D meshes allow for fast and seamless rendering of 3D meshes in cloud computing. The hierarchical display structure for a large 3D mesh is composed of several view options: a thumbnail-preview mode (TP), a coarse-whole-view mode (CWV), a zoomed-sector-view mode (ZSV), and finally a deep-zoom-of-the-mesh mode (DZM). Shape-outlines (TP mode), simplified meshes (CWV, ZSV modes), and the original mesh partitions (DZM mode) hierarchically represent large 3D meshes. For example, Table I depicts a huge mesh David with 28,184,526 vertices totaling 1.1 GB which cannot be loaded and displayed on a mobile device. Using our interactive rendering method, David can be displayed in real time on a mobile device with a hierarchical data structure as illustrated in Fig. 1. First, David is selected from a 3D shape-outline in TP mode in (a). A selected CWV is then generated with a simplified mesh of only 10,820 vertices in (b). After selecting the head in (b), a more detailed mesh of 10,649 vertices is rendered in ZSV mode as depicted in (c). For a DZM-mode view of the eye, partitions of the original mesh having 5,625 vertices are loaded and rendered in (d).

An overview of our interactive 3D mesh rendering for cloud computing is depicted in Fig. 2. A huge mesh is

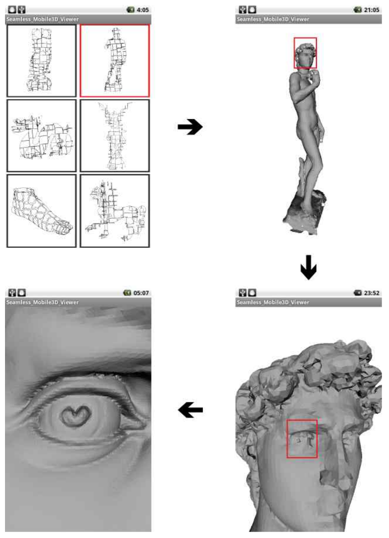


Figure 1. A hierarchical rendering of a huge 3D mesh on a mobile device[1]. David has 28,184,526 vertices at 1.1GB. (a) 3D shape-outlines in TP mode; (b) simplified David of 10,820 vertices; (c) more detailed head of 10,649 vertices; (d) original resolution eye of 5,625 vertices fully rendered.

This work was supported by Seoul R&DB program (ST100035).
[1] A mobile device ODROID-7 with Samsung S5PC 110 Cortex-A8 1Ghz CPU and 512MB RAM.

TABLE I. LOADING AND RENDERING TIME ON A MOBILE DEVICE[2]

	Original Model		Our Simplified Model	
File Name	*Vertices*	*Time(s)*	*Vertices*	*Time(s)*
feline	49,864	3.48	10,379	0.59
foot	160,226	9.00	8,324	0.56
dragon	399,332	25.69	9,797	0.61
ihigenie	351,750	23.90	9,999	0.62
bddha	541,366	33.99	10,275	0.63
xyzrgb_dragon	3,609,455	N/A	9,589	0.60
lucy	14,027,872	N/A	10,180	0.64
david	28,184,526	N/A	10,820	0.66
Average		**19.21**		**0.61**

Uniformly partitioned and simplified meshes provide a hierarchical representation for huge 3D meshes so that interactive renderings on mobile devices can be performed with optimized and predictable processing times.

uniformly partitioned based on a user-specified equal number of vertices for uniform I/O processing time. Then a 3D shape-outline of each uniform partition is extracted and simplified through its own boundaries. Mesh simplifications in multi-resolution are then executed by calculating the representative vertex for a group of vertices in each partition. The number of partitions can be controlled by the user allowing the file size of a simplified mesh to be easily manipulated. This enables the client to transfer and render hierarchical structures of huge 3D meshes according to the user's interaction with the server in cloud computing.

Partitioned mesh files, simplified meshes, and a shape-outline are generated and stored on a server as a hierarchy automatically whenever a huge mesh is uploaded. Then a client can access the hierarchy starting from a shape-outline as shown in Fig.1. Our work will add interactive mesh simplifications to provide appropriate simplified meshes according to a user's choice of views for server-side processing in the future.

A. *Uniform Mesh Partitioning*

The main goal of partitioning a large mesh is to minimize processing time while maintaining load balance. The CPU in most systems today have improved radically resulting in input-output (I/O) processing becoming the main factor in the overall processing time. Uniform partitioning is the division of a large mesh into partitions with an equal number of user-specified vertices. Uniform partitioning is essential for a 3D mesh in cloud computing, so as to enable the assignment of standardized times to the processing of each partition as well as to optimize I/O processing time. Typically, mesh partitioning has been implemented by clustering vertices or faces. Clustering has been accomplished through either space subdivisions [1][7] or incremental additions [2][6]. The octree method provides fast hierarchical clustering [1][7]. However, the numbers of vertices or faces in partitions are varied because the division is performed not by the numbers of vertices or faces but by the sizes of the cells. K-means clustering [6] can generate partially uniform mesh partitions. However, it does not

[2] A smart phone LG-SU660 with 1GHz Dual Core CPU and 512MB RAM.

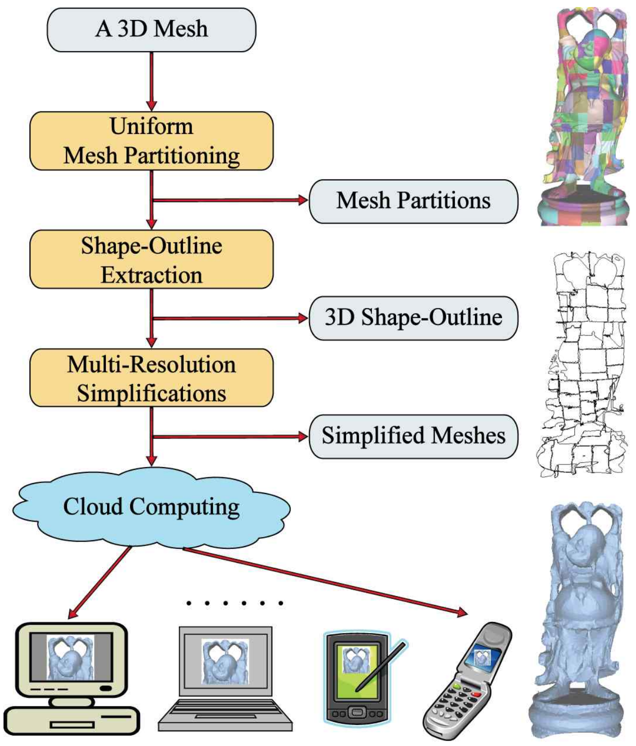


Figure 2. Overview of our interactive 3D mesh rendering in cloud computing.

provide uniform mesh partitioning and hierarchical clusters. Also, initial positions of random seeds must be carefully selected and an elaborate cost function must be designed to attain quality results. Optimization takes a great deal of time requiring many repetitions for large 3D meshes [2].

Our algorithm constructs a kd-tree for a mesh. Each cell in the kd-tree represents a vertex cluster which forms a single partition of the mesh. For a given mesh, our kd-tree divides space based on the object median where the objects are vertices of the mesh. Our kd-tree splits a cell into two sub-cells each containing half the vertices of the cell. Instead of cycling the axis from x to y to z-axis for a perpendicular splitting plane, our kd-tree determines the axis adaptively according to the longest axis of a bounding box. Compactness is a quantity for measuring the degree to which a shape is compact. Given a partition with area w and perimeter p, we define the compactness c of the partition as a ratio of its squared perimeter p2 to its areas w [9].

$$c = \frac{p^2}{4\pi w} \qquad (1)$$

A square figure has better compactness than a long thin rectangular figure. To avoid long thin shaped partitions, our algorithm considers compactness when determining an axis for perpendicular splitting planes to subdivide cells in the kd-tree. Fig. 3 depicts the steps from level 1 to level 4 of the kd-tree in a simplified 2D format. The dotted-line is the bounding box of vertices in a cell. The solid lines are

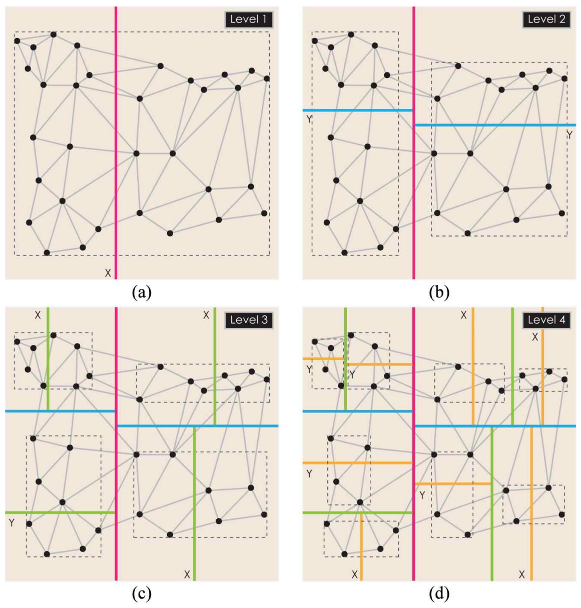


Figure 3. An example of our kd-tree construction in 2D. Our kd-tree is based on the uniform number of vertices in each cell. An axis to be split is adaptively determined to lower the compactness of the cell.

determined by, and are perpendicular to, the longer of the two axes, x and y, of the bounding box and colored red for two cells in level 1, blue for four cells in level 2, green for eight cells in level 3, and yellow for sixteen cells in level 4. The axes for cells at the same level may be chosen differently depending on the shape of the bounding boxes as depicted in Fig. 3. Median values are computed to split vertices in half. Finally, sixteen uniform partitions of the mesh are created from sixteen clusters of vertices in sixteen leaf-cells in the kd-tree. To construct a kd-tree of a mesh, a median value of the vertices in a cell needs to be determined so as to split a cell into two subcells with equal numbers of vertices. For an out-of-core mesh which has more data than the size of the main memory, external sorting needs to be applied; however, external sorting takes a lot of time. Therefore, we plan to introduce an improved out-of-core sorting method to find median values.

In Fig. 4, two previous partitioning methods are compared with our method for a model foot of 40,058 vertices in (a). Partitioning results are listed in (b) by k-means clustering, (c) by octree-based clustering, and (d) by our kd-tree based clustering. The numbers of vertices in each partition are charted in (e). K-means clustering generates 128 partitions in 6.22 seconds with a compactness measure of 3.155. Octree clustering runs fast in 3.76 seconds with a compactness of 1.806 for 126 partitions. Our kd-tree clustering generates 128 uniform partitions in 3.85 seconds with a compactness of 1.929. Only our kd-tree based clustering creates uniform partitioning with quality shapes in a relatively fast processing time.

B. *Mesh Simplification Using Our Mesh Partitioning*

A uniform number of vertices in a partition plays a key role in the quality of the mesh simplification. A single representative vertex for a partition was calculated for the vertices of the partition. Triangulations were performed with simplified vertices according to the original connectivity. The size and the shape of simplified triangle faces are more regular with our kd-tree method since each simplified vertex represents a uniform number of vertices, whereas each simplified vertex using the octree method represents various numbers of vertices as depicted in Fig. 5. In (a), the mesh is simplified to 1,104 vertices using our kd-tree method whereas in (b) the mesh is simplified to 1,206 vertices using the octree method. The mean distortion to the original mesh is 0.4149 by our kd-tree and 0.4563 with the octree [8]. Our simplification preserves the original shape better with better triangulation.

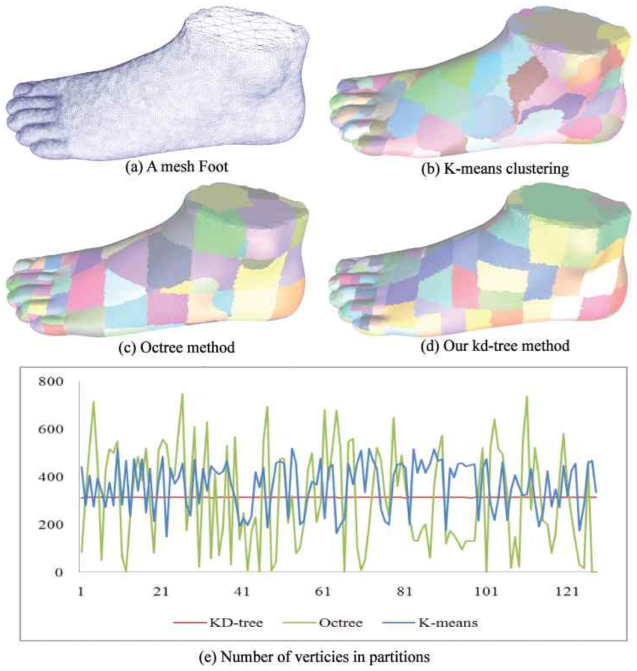


Figure 4. Examples of mesh partitioning. A mesh Foot of 40,058 vertices and 80,112 faces is rendered in wire frame in (a). Partitioning results are listed in (b) by k-means clustering, (c) by octree-based clustering, and (d) by our kd-tree based clustering. The numbers of vertices in each partition are charted in (e).

C. *3D Shape-Outlines: Interactive 3D Previews*

For 2D images, TP mode provides small thumbnail images so that a user can easily and quickly identify and select a specific image. Until present, an interactive TP mode for 3D meshes has not been available. As such, we offer our TP mode which uses shape-outlines for interactive 3D mesh previews to dramatically reduce file size. As shown in Fig. 1(a), a series of shape-outlines can easily be displayed on mobile devices with no need for file names. A shape-outline also depicts how the mesh is partitioned. As illustrated in Fig. 6, each TP shape-outline can be interacted with to translate or rotate the thumbnail with no need to fully display the mesh.

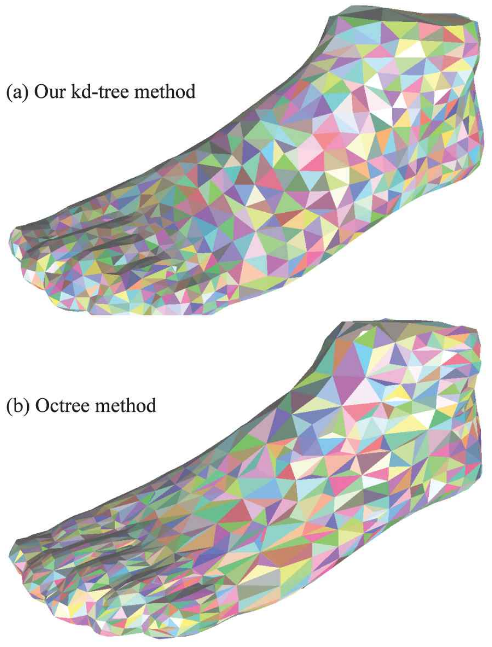


Figure 5. Examples of simplified meshes with a mesh Foot. In (a), the mesh is simplified to 1,104 vertices using our kd-tree method whereas in (b), the mesh is simplified to 1,206 vertices using the octree method.

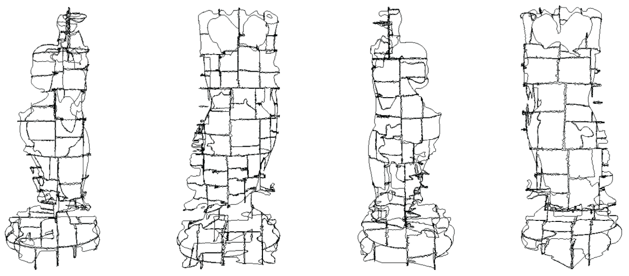


Figure 6. A shape-outline of a model Buddha in various views. A user can interactively control TP mode views of the shape-outline.

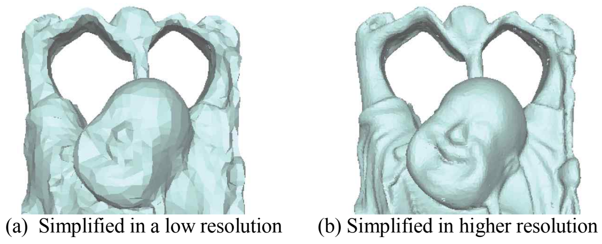

(a) Simplified in a low resolution
(b) Simplified in higher resolution

Figure 7. Simplified meshes in multi-resolution for a mesh Buddha. Rather than zooming in to a lower resolution of the simplified mesh in CWV mode in (a), rendering can automatically switch to ZSV mode to get a higher resolution zoom.

D. *Simplified Meshes in Multi-Resolution*

Simplified meshes in multi-resolution can provide CWV and ZSV modes of 3D meshes. As shown in Fig. 7, the more detailed Buddha of 9,539 vertices in (b) provides higher resolution than simply an enlarged but degraded view of the simplified mesh of 2,569 vertices in (a).

E. *Mesh Partitions for the Closest View*

Finally, for DZM-mode views of meshes, uniformly partitioned files of the selected area of the original mesh are transferred and rendered as shown in Fig. 1(d). The number of vertices in each partition can be specified by a user to provide optimized and predictable processing time for each partition.

III. CONCLUSION AND FURTURE WORK

This paper introduced a hierarchical representation of 3D meshes for interactive rendering in cloud computing. As listed in Table I, the rendering of 3D meshes on a mobile device took 19.21 seconds on average while huge meshes could not be loaded due to device memory limitations. With our hierarchical method, interactive rendering can be provided in real time in about 0.6 seconds on average even for huge meshes. In our future work, we will investigate how to automatically control simplification levels on the server or the client. Our research has led us to conclude that texture mapping to simplified meshes should be further studied. Moreover, how to approximate texture coordinates for simplified meshes also needs further investigation.

REFERENCES

[1] D. Kim, S. Lee, H. Lee, and S. Cho, “A distance-based compression of 3D meshes for mobile devices,” IEEE Trans. Consumer Electron., vol. 54, no. 3, pp. 1398-1405, 2008.

[2] S. Choe, J. Kim, H. Lee, and S. Lee, “Random accessible mesh compression using mesh chartification,” IEEE Trans. Visualization and Computer Graphics, vol. 15, no. 1, pp. 160-173, 2009.

[3] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno, “External memory management and simplification of huge meshes,” IEEE Trans. Visualization and Computer Graphics, vol. 9, no. 4, pp. 525-537, 2003.

[4] S. Schaefer and J. Warren, “Adaptive vertex clustering using octrees,” SIAM Geometric Design and Computing, 2003.

[5] Y. Okamoto, T. Oishi, and K. Ikeuchi, ”Image-Based Network Rendering of Large Meshes for Cloud Computing,” International Journal of Computer Vision, vol. 94, no. 1, pp. 23-35, August 2011.

[6] S. Lloyd, “Least square quantization in PCM,” Information Theory, IEEE Transactions, vol. 28, no. 2, pp. 129-137, 1982.

[7] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno, “External memory management and simplification of huge meshes,” IEEE Trans. Visualization and Computer Graphics, vol. 9, no. 4, pp. 525-537, 2003.

[8] P. Cignoni, C. Rocchini and R. Scopigno, “Metro: measuring error on simplified surfaces,” Computer Graphics Forum, Blackwell Publishers, vol. 17, no. 2, pp. 167-174, June 1998.

[9] M. Garland, A. Willmott, and P. Heckbert, “Hierarchical face clustering on polygonal surfaces,” Proc. ACM Symposium on Interactive 3D Graphics, pp. 49-58, 2001.

Fault Tolerant Approaches in Cloud Computing Infrastructures

Alain Tchana, Laurent Broto, Daniel Hagimont
Institut de Recherche en Informatique de Toulouse (IRIT)
Toulouse, France
Email: alain.tchana@enseeiht.fr, laurent.broto@enseeiht.fr, daniel.hagimont@enseeiht.fr

***Abstract*—Based on the pay-as-you-go strategy, cloud computing platforms are spreading very rapidly. One of the main characteristics of cloud computing is the splitting into many layers. From a technical point of view, most cloud computing platforms exploit virtualization, which implies that they are split into 3 layers: hosts, virtual machines and applications. From an administration point of view, they are split into 2 layers: the cloud provider who manages the hosting center and the customer who manages his application in the cloud. This structuring of cloud makes it difficult to implement effective management policies. This paper focuses on fault tolerance in cloud computing platforms and more precisely on autonomic repair in case of faults. It discusses the implications of this splitting in the implementation of fault tolerance. In most of current approaches, fault tolerance is exclusively handled by the provider or the customer, which leads to partial or inefficient solutions. Solutions, which involve a collaboration between the provider and the customer are much promising. We illustrate this discussion with experiments where exclusive and collaborative fault tolerance solutions are implemented in an autonomic cloud infrastructure that we prototyped.**

***Keywords*-Cloud Computing, Fault tolerance, Virtualisation.**

I. Introduction

Due to the difficulty to maintain an internal infrastructure technology and the associated rising costs, companies are increasingly externalizing their IT services, which are therefore managed by specialized companies (called providers). This trend led to the emergence of the so-called cloud computing approach. One of the most important objective of cloud computing is to allow customers to pay only for the amount of resources they effectively consume. This option, summarized by the term pay-as-you-go, is permitted in cloud platforms through the partitioning of their resources.

Virtualization techniques are commonly used in cloud platforms to implement partitioning of resources. Instead of having direct access to cloud resources, customers have access to virtual machines, which represent a fraction of a physical machine. Then, we identify three layers in such a cloud infrastructure: the physical resource layer (containing the overall cloud resources), the virtualization layer (containing virtual machines) and the applications layer (containing applications of external companies, which are hosted in the cloud).

From an administration point of view, we consider two main roles, which correspond to the administration of the hosting infrastructure (the provider) and the administration of the application deployed in the cloud (the customer).

These multiple layers and roles make difficult the management of cloud platforms and particularly the management of failures in these infrastructures. Indeed, handling failures become more complex because those who intervene in the cloud (customers and provider) have different views (and access rights) of the different layers of the cloud. Costumers are limited to only detecting faults of virtual machines and their applications, while the provider can only manage real resources (physical machines) and virtual machines faults. Therefore, possible Fault Tolerance (FT) solutions vary according to the involved participants and according to the implementation level.

Although current cloud platforms take in account many challenges, their implementation usually propose no fault tolerance solution ([1], [2]) or basic FT solutions ([3]). For those who implement FT services ([4], [5], [6], [7]), we retain that their solutions only entrust the responsibility of fault management either to the customer or to the provider. No collaboration between the two types of participants is considered.

The purpose of this paper is to investigate FT policies in cloud platforms. We identify two types of policies: one, which is exclusively handled by one participant (customer or provider) and another, which is a collaborative management between the provider and customers. This second type constitutes an interesting tradeoff between exclusive management by the provider and exclusive management by the customer. This discussion is illustrated by experiments and evaluations with an operational prototype of autonomic cloud platform.

The rest of the paper is organized as follows. Section II introduces the cloud computing technology (concepts and architecture) and its challenges (including fault management). Section III covers fault detection and management techniques in the cloud. Section IV discusses related work. Section V presents experiments and evaluations, which illustrate our reflection. Section VI concludes and outlines areas for future works.

II. Cloud Computing overview

Due to the lack of consensus on the definition of cloud computing, let us refer to the CISCO [8] one: ”*IT resources and services that are abstracted from the underlying infras-*

tructure and provided on-demand and at scale in a multitenant environment". Therefore, cloud computing consists in: (1) providing on demand services to external customers with the illusion of infinite resource, (2) and then using the same resource pool for all customers. This strategy offers several advantages including:

- Reduced costs for the customer. He no longer needs to manage his own infrastructure and is billed according to the use of cloud services.
- Flexibility for the customer. He can increase the capacity of his infrastructure without major investments, resources of the cloud are dynamically allocated on demand.
- Less waste. Internal IT systems managed by customers are often under-utilized while the cloud infrastructure is shared between many customers, which increases the average resource utilization rate. A important example of this waste is the energy consumption of such infrastructures.

Several models of cloud are presented in the literature, including: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). In this article, we consider a cloud as an IaaS: a virtualized infrastructure managed by a provider, in which external customers deploy and execute their applications. Then, the parties involved in a cloud platform are grouped into three categories: cloud providers, cloud customers and end users. A cloud provider is responsible for the administration of the cloud resources (hardware and virtual machines(VM)) and services. He is responsible for managing the accommodation of the capacity of the cloud. Cloud customers use the resources provided by the cloud to deploy and execute their applications. They do not have a global view and direct access to the cloud environment. They use cloud resources through VMs, which host their applications and, which represent a confined portion of physical resource. End users are using customer applications deployed in the cloud. Figure 1 summarizes and shows the vertical cloud architecture and the scope of each cloud participant: the cloud provider has access to physical resources and VMs; cloud customers have access to VMs and their applications; and end users have access to customer applications.

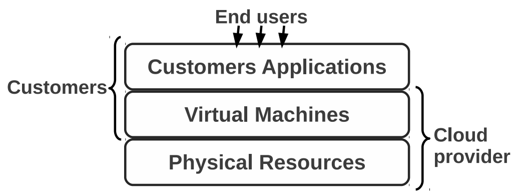


Figure 1. Cloud Computing architecture

III. Fault Tolerance Techniques in Cloud Platforms

As shown in Figure 1, three layers are identifiable in a cloud platform: resources, VMs and applications. Each of them is concerned with failures. Therefore, we identify three types of failure in a cloud platform: hardware failure, VM failure and application failure. A FT strategy includes two distinct phases: detection phase and repair phase. One of the difficulties to implement FT in a cloud architecture can be summarize by this question: which cloud participant (provider or customer) is the best able to implement the two phases of fault management, depending on its access rights in the cloud architecture? In other words, is it reasonable to leave exclusively the responsibility of FT to one cloud participant knowing that: hardware failures can only be detected and repaired by the cloud provider; VM failures can be detected by the two participants but only repaired by the cloud provider; and application failures can only be detected by the customer but can be repaired by the two participants.

We present in this section two visions of FT management in a cloud platform. The first one consists in giving both the detection and repair responsibilities to one cloud participant (exclusively) while the second is to harness the skills of the two types of participant. According to these two visions, we present some FT techniques for the three types of failure we have identified: hardware, VM and application.

A. Exclusive FT Management

We discus about exclusive FT solutions in this section.

1) Application FT: As mentioned above, application failures are detectable only at the customer level. The failure detection policy depends on the application. However, the mechanism used to implement a detection policy is generally the same. For each application, the customer deploys in the cloud special software components called sensors, which monitor the liveness of the application. According to this monitoring, a sensor may trigger the execution of a procedure for repairing the application when it is considered as malfunctioning.

Two methods are possible for repairing a faulting application. The first one concerns stateless application (such as loadbalancers, e.g., HAProxy or MySQL-Proxy used in our experiments). Repair consists in restarting the faulting server on the same VM. The second method concerns statefull servers (e.g., the MySQL database). In this case, the customer must implement a mechanism for saving the server state so that it can be restored before the server is restarted. In the case of a database server for example, this save/restore mechanism can be implemented by trapping and storing all modification requests, allowing replay of these requests.

The advantage of this solution is that the customer has full control over the solution. Then, the application can be hosted in any cloud platform without assuming that the platform implements any FT solution or that the platform is dedicated to a particular kind of application. However, this solution requires that customers have expertise on their applications so that they can implement application state save/restore procedures notably, and they must implement their own monitoring system.

2) VM Fault Tolerance: VM failures can be detected and repaired by the two cloud participants. We consider here repair policies that are implemented exclusively by one of them.

Customers implement VM FT by deploying sensors in the cloud, which monitor VM state during their lifetime. In this case, it is not recommended to give to a single sensor the responsibility of probing one VM because a failure of the VM hosting this sensor would compromise the repair mechanism. The repair of the failed VM is organized as follows: (1) the customer level requests the cloud to free the failed VM; (2) it allocates a new VM; (3) it deploys and starts the servers that were running on the failed VM; and (4) it restores the state of these servers in case of statefull servers. One disadvantage of this solution is that each customer must implement his own VM monitoring system, which leads to complexity and network resource wasting, while VM monitoring can be factorized and implemented by the cloud provider.

At the cloud provider level, an exclusive VM FT technique can be implemented. With a direct access to VM hypervisors, the cloud provider is more likely to implement VM FT. Firstly, such an implementation decreases the number of VM sensors (and their associated communication) as they are integrated in hypervisors. Actually, a single sensor per physical machine can monitor all the VMs hosted on this machine. Secondly, through the hypervisor, the provider can collect more detailed information about VM status. This information allows him to implement more accurate FT solutions according to VM status. In contrast, when a VM failure is detected by the customer, it is more precisely the VM inaccessibility, which is detected since the customer is not supposed to be granted access to hypervisor information. At the cloud level, the provider is able to identify several types of VM failure. For example, considering the Xen hypervisor, a VM can be broken when its state is blocked, pending, or error. The provider can then implement VM repair so that an appropriate repair method is applied for each state of failure. Also, the analysis of Xen logs allows releasing a VM (instead of reallocating a new VM as the customer level would do) whose state is blocked waiting for a device. A more agnostic solution for VM FT is to regularly store VM states using the checkpointing ability offered by virtualization systems. Thus, the cloud will just restart a failed VM from its last saved state.

The advantage of provider level solutions resides in the fact that it factorizes VM FT tasks that would be implemented by the customers. However, being not aware of the particularities of VM hosted applications, the provider repair solution may not be efficient in certain cases. For example, repairing a VM, which hosts MPI processes (used by other VMs) cannot be done by restoring the VM to its last state. MPI applications do not support rollbacks, even negligible. In this case, a collaboration between the provider and the customer can be the solution.

3) Physical Machine Fault Tolerance: Hardware failures are more difficult to take into account in the cloud because they trigger failures at the other levels (VM and application). We consider repair implemented exclusively at the customer or provider level.

At the customer level, it is impossible to detect a physical machine failure. At this level, a physical machine failure is perceived as multiple VM failures (the VMs running on the failing machine). The customer level only sees VMs and even if VM monitoring sensors are deployed in the cloud, they may not be aware of a hardware failure if they are all deployed on the same physical machine (sensors deployed on VMs hosted by the faulting machine). The customer would need to implement constraints regarding the physical location of sensors, which is a form of collaboration between the customer and the cloud provider (we will come back to this collaboration in the next section).

At cloud provider level, hardware FT is implemented with a monitoring system composed of sensors deployed on different physical machines. For repair, the provider will start on a new machine (or several machines according to its capacities) the same number of VMs, which were hosted on the failed machine. In addition, all VM states must be saved by checkpointing so that VM restoration is possible. This solution is used in [5] and [9] for example.

B. Collaborative Fault Tolerance Management

Exclusive FT techniques presented in the previous sections highlight difficulties regarding techniques applied for certain types of failure. As will be discussed in this section, these difficulties can be taken into account if a collaborative management between the cloud provider and customers is considered.

1) Application Fault Tolerance: In Section III-A1 we have described an application fault detection without specifying the impact on other levels of failure (VM and hardware). If the sensor is deployed on a separate VM than the application, then a detected failure can have three origins: application (already treated in Section III-A1), VM or hardware. In the case of a VM failure, the collaboration between the application sensor and the VM sensor (possibly at the customer level) will give more clarification to the application sensor. If the VM sensor does not detect any error, then it can conclude that it is an application failure. Otherwise, it can deduce that it is a VM or a hardware failure. In this latter case, the customer alone has no way to know much. A better collaboration with the cloud level would provide confirmation of the failure type.

Concerning the repair of a failed application, a collaborative repair may be considered. Indeed, the cloud can offer regular backups of VM states so that a customer can subscribe to and invoke the cloud level for the restoration of a VM from a saved image.

2) VM Fault Tolerance: If the VM fault detection is done by the customer, we will have the same problem as described in the previous section. At the customer level, a VM failure detected by a sensor can be due to a hardware failure (the machine, which hosts the VM), which is out of scope for the customer. A VM fault detection at the cloud level allows getting a more accurate decision.

If the VM fault detection is implemented at the cloud level, a collaboration with the customer level can provide a better solution than the checkpoint-based one described in section III-A2. Concretely, once the fault is detected by the cloud, it starts a new VM with the same features (networking, memory, CPU, image) as the failed VM and then calls the customer to redeploy, restart and synchronize the new VM. This solution would probably perform better than the checkpointing one regarding the cost of save/restore operations.

3) Physical Machine Fault Tolerance: From the point of view of the customer, the failure of a physical machine is identical to the VM crashes. When detected by the cloud provider, such a failure can be resolved collaboratively during the repair of each VM hosted on the failed machine. Each VM is restarted with the same characteristics on a new physical machine and the completion of the repair is asked to the customer (as seen in the previous section).

C. Synthesis

We have discussed possibilities for implementing FT (repair) in a cloud environment. In a cloud environment composed of a hosting center managed by a provider and applications deployed in this hosting center and managed by customers, we distinguish:

- FT managed exclusively at the customer level. At this level, it is possible to detect application and VM faults, but detecting hardware faults is difficult and it is not possible to have details about detected VM fault (this information is only available at the hypervisor level). Repair can be implemented at this level by restarting VMs (VM and hardware faults) and redeploying and restoring applications (all faults). The main drawback is that each customer must implement the whole FT policy.
- FT managed exclusively at the provider level. At this level, application faults cannot be detected, but detailed information can be given for VM and hardware faults. Repair can be based on VM checkpointing, which is independent from applications, but can be quite costly.
- FT managed collaboratively at both levels. Applications faults must be detected and repaired at the customer level. VM and hardware faults can be detected at the provider level (with details). The repair of the VMs (upon VM and hardware faults) can then be accurately performed at the provider level, and finally the recovery of the application that were running on these VMs can be requested and performed at the customer level. The recovery (redeployment and restoration) of applications on VMs can also have a significant cost.

Naturally, collaborative techniques appear to be the best suited. However, most of the proposed solutions are exclusively implemented in one level, as it is strategically difficult to split an FT policy between two participants.

IV. Related work

As we mentioned in the introduction, few works addressed the issues of FT in cloud environments. Some platforms such as Eucalyptus [1] or CLEVER [6] provide no solution to take into account hardware, VM or customer application failures. CLEVER addresses FT management, but only for its own components.

OpenNebula [3] offers exclusive VM FT implemented at the cloud level. It allows the cloud provider to associate hooks (scripts or programs) with each type of VM failure (according to hypervisor information). Hardware failures are not addressed by OpenNebula for two reasons: it provides no hardware sensors and all VM sensors are located on the same machine than the VM. So a machine failure cannot be detected by OpenNebula.

As OpenNebula, the Microsoft Windows Azure platform [5] offers an exclusive FT management at the cloud level. Windows Azure replicates each VM so that a VM failure is covered by its replicas. The Azure solution is limited to web applications developed in the Windows Azure platform. Moreover, no solution is proposed to repair the failed VM. In addition, for VMs which are not instantiated by the Azure development platform, the entire responsibility of FT management is left to the customer. This is also the case in the Amazon EC2 [7] cloud platform.

Kaushal [4] proposes an FT solution in the cloud at the customer level by replicating servers queries, based on the HA-Proxy load distributor. Other researches such as [10] and [11] propose a collaborative solutions for specifics applications (MPI for example). However, they do not consider the splitting of the cloud between a (VM) provider and its customers, so their works are only applicable to an SaaS cloud.

Uesheng Tan [12] describes a replication solution for VM FT, exclusively managed by the cloud provider. It proposes to improve efficiency by using passive VM replicas (with very few resources), which become active when a failure is detected. A mechanism is introduced to transfer/initialize the state of VM. This solution is similar to the exclusive FT solution (at the provider level) that we presented previously and that we evaluate in the next section.

Finally, Wenbing Zhao [13] proposes a FT middleware, which can be used by a cloud customer to implement software FT. Their main purpose is to implement a synchronized server replication strategy so that a failed server can be repaired with a consistent state.

V. Evaluations

In these experiments, we evaluate the two visions that we presented above: exclusive and collaborative FT strategies. Due to the limited space in this paper, we only present our evaluation with VM FT. In these evaluations, we detect a VM failure when the VM does not respond to a *ping* request or when the hypervisor indicates an error state.

The evaluation environment is composed as follows. For the cloud platform, we use a prototype called CloudEngine, based on an adaptable autonomic management system called TUNeEngine, which were both implemented in our research group. Briefly, CloudEngine is similar to the OpenNebula system, but it is more adaptable and flexible because it is based on an adaptable system (TUNeEngine). For example, it allows easy addition of new functionalities, management policies and offers collaborative API. This prototype allows, at the cloud level, the deployment of VMs and the definition of VM level reconfiguration policies (repair in our case), and at the customer level, the deployment of application servers and their reconfiguration/repair.

Our prototype is used to allocate VM in the cloud, deploy and start a J2EE application (RUBIS [14]) as our customer application. Our J2EE experimental application is composed of an Apache web server, two Tomcat servlet containers, a MySQL-Proxy database loadbalancer and two MySQL database servers. The MySQL stage is our replicated layer in which failures will be triggered. For the evaluation of our two FT techniques, we apply a pyramidal RUBIS workload (upload phase, constant load phase and download phase) in which we simulate a VM failure during the constant load phase. The objective of the experiment can be summarized by this question: what does the FT technique cost in term of RUBIS performance (throughput). To answer this question, we ran the same workload without failure in order to have a reference execution with which the others can be compared. We observe the request throughput during the benchmark and the number of untreated requests during repair.

A. Exclusive Fault Tolerance

The exclusive FT technique we evaluate in this experiment is implemented in the cloud level. The implemented FT policy starts a checkpointing program on each IaaS node, which role is to save the status of each VM every 7 seconds. The choice of the backup frequency should not be too small nor too large for two reasons: the risk of penalizing the performance of the VM (as discussed) and the risk of having a large gap between VM status after and before the failure. Upon failure, the last checkpoint is used to restore a recent image of the failed VM.

Figure 2 shows the comparison between the experimental landmark (red curve) and the experiment with the checkpointing (green curve) in which no failure was simulated. These first curves allow us to observe the overhead of VM checkpointing. We estimate this overhead is about 46% due to the Xen VM checkpointing implementation. Actually, the checkpointing implemented by Xen causes the unavailability of the VM during the checkpoint, which explains this overhead. Notice however that this unavailability does not cause request loss since the TCP/IP protocol (on which our network is based) retransmits a request when communication fails. Thus, during checkpointing, the downtime of the VM does not break TCP/IP communications and it only postpones request handling. It explains the large fluctuations in the green curve.

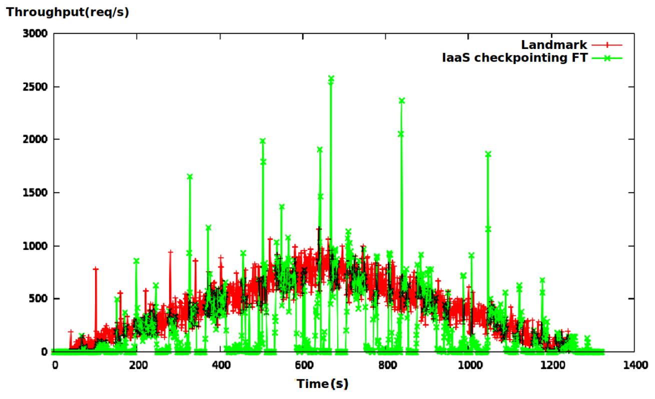


Figure 2. Checkpointing cost during VM repair in the IaaS

Figure 3 shows the result of applying our FT method (at the cloud level) with fault simulation on a MySQL server VM. T_p represents the failure date while T_r marks the end of the repair. The repair time in this experiment is about 22 seconds. This time includes the time taken to detect the failure and also the duration of VM restarting (from its last saved state).

Notice here that requests are lost when we recover from the last saved checkpoint. This can be tolerated for a web application but it would not for more sensitive applications such as MPI applications.

B. Collaborative Fault Tolerance

The second FT technique we evaluated is a collaborative one in which VM fault detection and repair operations are performed (collaboratively) by CloudEngine (the IaaS) and the customer (via our TUNeEngine autonomic administration system at the application level). CloudEngine detects VM failure, restarts the VM from its original image and finally invokes the customer level TUNeEngine system to

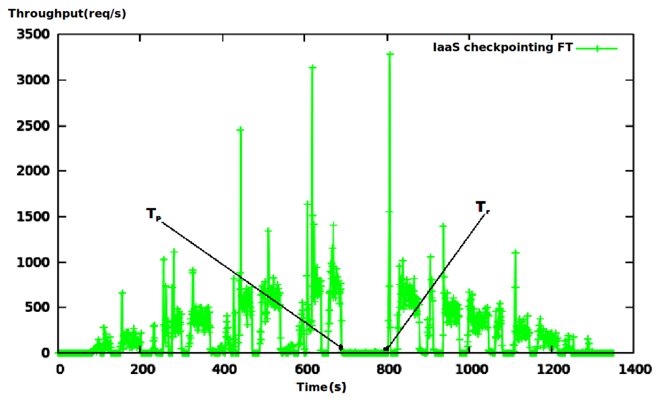


Figure 3. Exclusive VM Fault Tolerance Technique

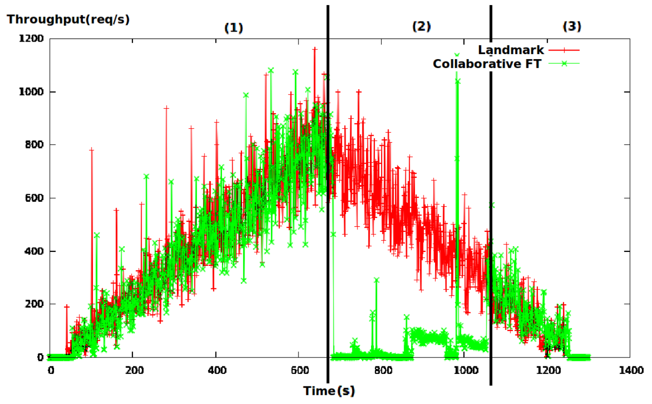


Figure 4. Collaborative VM Fault Tolerance Technique

complete the repair. Then, TUNeEngine deploys on the restarted VM the application that was running on it before the failure (MySQL in this experiment), configures and starts the MySQL application. The size of the deployed application and the reconfiguration and start operations influence the overall duration of the repair.

Figure 4 shows two curves: the red curve represents the reference execution (without failure nor FT management) while the green curve represents the result of our collaborative FT technique. We observe that this repair method, unlike the previous one, has no impact on RUBIS performance when no failure is involved. This is shown in Figure 4 areas (1) and (3). Zone (2) represents the duration of the repair. It includes: the failure detection (at the cloud level), the VM deployment and restart, the MySQL server binaries copy and restart. For these reasons, we measured in this experiment a repair time of 5 minutes 30 seconds. Notice that the use of a mirror server, which is usually the case for such applications, would keep the service available during the repair of the failed server.

Even if the repair time is much higher with this solution, it seems best suited as it does not incur any overhead on execution.

VI. Conclusion and Future Works

In this paper, we studied two visions for FT management in cloud computing platforms: the first one consists in leaving exclusively the responsibility of FT management to one cloud participant (cloud customer or provider); the second one consists in sharing the responsibility between the two cloud participants. We reviewed all possible fault situations in the cloud: application level, virtualization level and hardware level. We proposed for each of them some solutions involving exclusive or collaborative FT visions. Given the limited space in this article, we only evaluated two FT techniques (one exclusive and one collaborative). However, this evaluation illustrates that sharing FT management between the two cloud participants opens interesting perspectives.

In the near future, we intend to complete our experiments with all the techniques described in this paper. Furthermore, VM FT techniques based on checkpointing can be improved by new VM checkpointing solutions, which consist in storing only the difference between successive VM states (rather than the entire VM state).

References

[1] Daniel Nurmi, Richard Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov "The eucalyptus open-source cloud-computing system," in *9th International Symposium on Cluster Computing and the Grid (CCGRID), vol. 0, pp. 124-131. Washington, DC, USA*, 2009.

[2] University. of Chicago, "Nimbus is cloud computing for science," http://www.nimbusproject.org/, [retrieved: january, 2012].

[3] OpenNebula, "Opennebula.org: The open source toolkit for cloud computing," http://opennebula.org, [retrieved: january, 2012].

[4] Vishonika Kaushal and Vishonika Kaushal, "Autonomic fault tolerance using haproxy in cloud environment," *International Journal of Advanced Engeneering Sciences and Technologies*, vol. 7, 2010.

[5] Microsoft, "Windows azure: Microsoft's cloud services platform," http://www.microsoft.com/windowsazure/, [retrieved: january, 2012].

[6] Francesco Tusa, Maurizio Paone, Massimo Villari, and Antonio Puliafito, "Clever: A cloud-enabled virtual environment," in *IEEE Symposium on Computers and Communications (ISCC), pp. 477-482. Riccione, Italy*, 2010.

[7] Amazon, Inc, *Amazon Elastic Compute Cloud (Amazon EC2)*. Available: http://aws.amazon.com/ec2/#pricing, [retrieved: january, 2012].

[8] Kapil Bakshi, "Cisco cloud computing - data center strategy, architecture, and solutions point of view white paper for u.s. public sector 1st edition," 2009, http://www.cisco.com/web/strategy/docs/gov/ CiscoCloudComputing_WP.pdf, [retrieved: january, 2012].

[9] Walters John Paul and Chaudhary Vipin, "A fault-tolerant strategy for virtualized hpc clusters," *The Journal of Supercomputing*, vol. 50, 2009.

[10] Qin Zheng, "Improving mapreduce fault tolerance in the cloud," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), pp. 1-6*, 2010.

[11] Magdalena Slawinska, Jaroslaw Slawinski, and Vaidy Sunderam, "Unibus: Aspects of heterogeneity and fault tolerance in cloud computing," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), pp. 1-10*, 2010.

[12] Uesheng Tan, Dengliang Luo, and Jingyu Wang, "Cc-vit: Virtualization intrusion tolerance based on cloud computing," in *2nd International Conference on Information Engineering and Computer Science (ICIECS), pp. 1-6. Wuhan, China*, 2010.

[13] Wenbing Zhao, Michael Melliar-Smith, and Louise E. Moser, "Fault tolerance middleware for cloud computing," in *3rd International Conference on Cloud Computing (CLOUD 2010), pp. 67-74. Miami, FL, USA*, 2010.

[14] Cristiana Amza, Emmanuel Cecchet, Anupam Chanda, Alan L. Cox, S Elnikety, Romer Gil, Julie Marguerite, Karthick Rajamani, and Willy Zwaenepoel, "Specification and implementation of dynamic web site benchmarks," in *5th Annual Workshop on Workload Characterization (WWC-5), pp. 3-13. Austin, Texas, USA*, 2002.

BtrScript: A Safe Management System for Virtualized Data Center

Rémy Pottier, Jean-Marc Menaud
École des Mines de Nantes, Ascola (EMN/INRIA, LINA)
Nantes, France
Email: remy.pottier@mines-nantes.fr, jean-marc.menaud@mines-nantes.fr

***Abstract*—Virtual machine management in data centers is more and more complex due to the increasing total number of virtual machines. Virtual machine resources and scheduled policies (e.g., consolidation) define the virtual machine placement. This placement is difficult to compute for large infrastructures. Administrators maintain a correct placement by performing actions (e.g., migrate virtual machines, power off servers, etc.) and some time using autonomic schedulers. We propose btrScript: a safe autonomic system for virtual machine management that includes actions and placement rules. Actions are imperative operations to reconfigure the data center and declarative rules specify the virtual machine placement. Administrators schedule both actions and rules, to manage their data center(s). They can also interact with the btrScript system in order to monitor the data center and compute the correct virtual machine placement.**

***Keywords*-cloud computing; virtualization; management; domain specific language.**

I. Introduction

With the emergence of cloud computing, the hosting capacity of the data centers has been continuously growing to support the non-stop increasing clients demand. Currently, *Amazon Web Service* adds each day enough capacity to support the whole *Amazon.com* infrastructure as it was during its five first years [1].

Managing a data center implies to regularly manipulate both the virtual machines (VMs) and the servers. Common operations include snapshotting, starting, stopping, or resetting of VMs [2], but also starting, halting, rebooting or performing maintenance operations on servers. Each hosted VM has specific expectations regarding its quality of service and each action must be executed in accordance to its requirements. Typically, it is expected to have a sufficient amount of resources to run the VM at peak level, but also a placement that may be compatible with fault tolerance or networking requirements. Finally, its availability may be required at given periods (*e.g.* business hours for remote desktops).

Infrastructure As A Service (IaaS) solutions such as OpenStack [3] or VMWare VCenter [4] extremely simplify creations and deployments of virtual environments. However, the management of the VMs concepts did not follow these changes. Virtualized infrastructure management is still relying on manual changes on the environment as well as a reaction to problems after they occur. Such an approach is no longer compatible with an infrastructure composed of thousands of VMs. A system administrator cannot manipulate a large set of VMs insuring that his actions are compatible with the expected quality of service at a given time, but also will be compatible in the future. This situation has led to some new approaches for the infrastructure management employing automation to replace the traditional manual approach. For example, VMWare DRS [2] can react to a load spike and dynamically adapt the VMs placement following a set of rules given by the administrator.

Close to the VMWare DRS functionality, we have worked with an autonomic system called Entropy [5], [6]. To achieve autonomic computing, an architectural view called the MAPE-K (Monitor, Analyse, Plan, Execute, Knowledge) loop, has been suggested in [7]. Mainly, an autonomic system is a software component, configured by human administrators using high-level goals. It uses monitored elements (M), internal knowledge (K) of the system to analyse (A), plan (P) and execute (E) tasks based on these high-level goals. The low-level actions to achieve these goals are automatically calculated and executed.

Entropy implements a classical MAPE-K loop [7]. It specially focuses on the planning part (P). Based on constraint programming, the main Entropy's high-level goal is to ensure that placement rules are constantly satisfied, both on system rules (CPU, RAM) and for the administrative rules. From a given current configuration (initial VM placement and rules), Entropy proposes at each loop a new configuration and its associated ordered operations called the reconfiguration plan. Essentially, based on VM live migration, the reconfiguration plan allows to switch from the current to the new configuration. The main reason to use an autonomic system like VMWare DRS or Entropy, is that administrators define high-level goals by specifying criteria that characterise desirable states, but leave to the system the task of finding how to achieve that state.

The main drawback of these systems is that they react after a problem occurs. Thus, daily maintenance operations realised by administrators, like VM migration or creation, are verified by the system after completion, at the next MAPE-K execution loop. Therefore, placement rules can be unsatisfied for a time, which may cause degradation of the quality of service.

This paper, « BtrScript, a safe management system for virtualized data center », follows our research into Domain Specific Languages (DSL) on VM management [8]. BtrScript checks, according to active rules, the validity of all actions and rules performed by administrators. If an action or a rule is invalid, btrScript detects conflicts and displays all active rules involved for each conflict. To resolve a conflict, btrScript proposes a combination of two modules. The first one is a rule management system to modify, suspend or activate rules. The other one, by interacting with Entropy, proposes (if possible) to the administrator a reconfiguration plan that satisfies all rules. In addition, btrScript has an action scheduler that allows specifying time-based operations and rules. Each interactive or scheduling operation is ensured to be compatible with the current but also the future active rules. Finally, we extend the language proposed in [8] to allow administrators an easy placement rules management.

We evaluate btrScript by comparing our rule management with the scripting tool, vSphere PowerCLI, which can be used to manage rules in a VMWare infrastructure.

This paper is structured as follows. The next section presents the new modules of the btrScript architecture. Section III presents the VMScript syntax then introduces the language extension. Section IV describes mechanisms to ensure the consistency in the infrastructure. Section V explains how to compute reconfigurations that solve infrastructure issues. Section VI details the comparison of the rule management with btrScript and with the vSphere PowerCLI. Section VII summarizes the former research in the field. Finally, Section VIII concludes and presents future work.

II. System overview

In virtualized infrastructures, the number of VMs and servers, as well as the resource utilization of the VMs, evolves. Administrators have to regularly re-organize the infrastructure to optimize the resource usage. Moreover, specific requirements as fault tolerant mechanisms have to be defined by placement rules.

BtrScript uses mandatory rules existing in Entropy and in the vmWare DRS, the most widely adopted VM scheduler. Actually, rules define by administrators can restrict the VM placement (for example, enforce a VM to stay in the same server). As a rule does not describe actions, when administrators enable rules, no infrastructure reconfiguration is performed even if the rule is broken. The broken rule detection requires the following data:

- static resources: the cpu and memory capacities of VMs and servers ;
- dynamic resources: the cpu and memory utilization of VMs and servers ;
- the VM placement.

BtrScript is based on VMScript and so it is able to introspect and reconfigure a virtualized infrastructure (see figure 1). The placement rule management implies additional modules to help administrators to maintain the rule consistency and, consequently, the consistency of the infrastructure.

Administrators interact with btrScript modules through the *management console*. This console is an interpreter of the domain specific language that manages placement rules. This language is described in Section III-C1.

Scheduling VMs in a large infrastructure is a complex problem [9]. Furthermore, some issues in the infrastructure do not require to be fixed because they are temporary (for example, a cpu consumption peak). The *guardian* periodically analyzes the infrastructure to report violated rules and overloaded servers. In our opinion, a server is overloaded when the sum of the hosted VM cpu consumption equals the cpu capacity of the server. The VM memory is not considered because we assume memory overcommitment is not used and, consequently, the memory utilization of VMs can not be greater than the host memory. This module warns administrators about issues but it does not try to solve them. Administrators choose the appropriate moment to solve issues manually (by executing actions) or by means of the *placement module*.

The *placement module* is an algorithm that computes actions to execute in order to solve infrastructure issues. It has to integrate all the btrScript placement rules to solve them. Entropy is connected to btrScript to use it as a *placement module*.

III. btrScript Language

The VMScript language allows administrators to introspect and reconfigure the virtualized infrastructure. BtrScript is a VMScript extension that reuses these low-level operations and adds the placement rule management. Before the extension language presentation (Section III-C1), Section III-A briefly reminds the VMScript syntax to select, introspect and reconfigure elements in the infrastructure.

A. VMScript background

VMScript operations are based on set manipulation in order to deal with a large amount of VMs and servers. This part describes the selection of elements in the infrastructure to, afterwards, perform introspection and reconfigurations.

1) Selection: The architecture of virtualized infrastructures is a compound of two views, a physical view and a virtual one. For example, the physical view can be clusters that includes servers, and, the virtual view can be virtual jobs (vjobs) that includes VMs. These views define language types and their hierarchical organization. The hosting relation between servers and VMs maps both of the views.

Each element of the infrastructure is typed and has a unique name and some properties to enhance the infrastructure with additional information such as resource consumptions, operating systems (OS), IP addresses, states, etc.

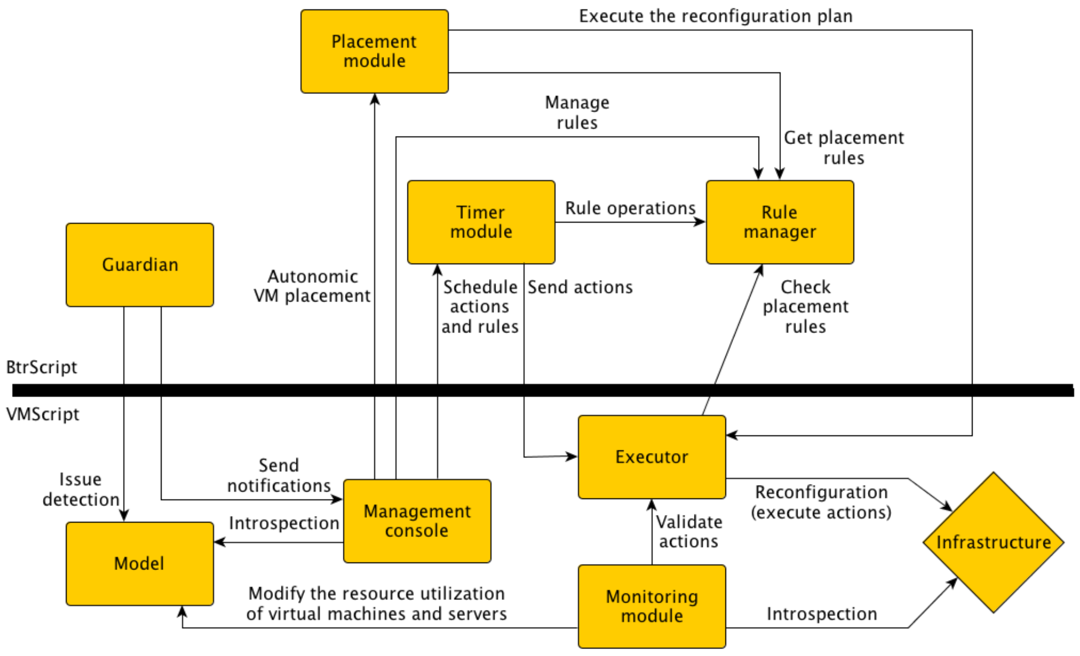


Figure 1. Global architecture

VMScript uses names to get elements. The "[]" operator allows the selection of a set of elements using a sequence of consecutive numbers, or an enumeration. The following expression selects the elements named *pm1*, *pm2*, *pm3* and *pm-master*:

```
pm[1−3,−master]
```

The "[]" operator can also make an union, a difference or an intersection of sets. The selection of the five servers named *pm1*, *pm2*, *pm3*, *hostA* and *hostB* is written by:

```
[ pm[1−3] , host[A,B] ]
```

The binary "/" operator allows to select elements from their type. The first parameter is a set defined with the previous syntax and the second parameter is a type. If the first element is omitted, all elements of the infrastructure with the specific type are selected. For example, the selection of servers of the infrastructure and the selection of VMs of the *cl1* cluster is, respectively, written by:

```
/server
cl1/vm
```

Element properties added to the model can be used to refine a selection thanks to the "{}" operator. For example, get all servers with a *Linux* OS:

```
/server{os == Linux}
```

B. Introspection and reconfigurations

Once elements to manipulate are selected, the operator ":" allows to introspect and reconfigure the infrastructure.

The introspection is available by getting the value of a property. The resource utilization is one of the default property of servers and VMs, so that it is easy to display the cpu consumption of VMs running on the server *s1*:

```
s1/vm:cpu_cons
```

VMScript actions perform infrastructure reconfigurations such as starting or migrating VMs. From a selection, actions can be applied on VMs and servers. For example, start servers and then migrate 3 VMs on one of them is written by:

```
s[2,6]:start
vm[1−3]:migrate s2
```

Supported actions for servers and VMs in the VMScript language are *start*, *stop*, *suspend*, *resume*, *reboot*. The supported action for servers are *createvm* and supported actions for VMs are *snapshot*, *migrate*.

C. The language extension

1) The rule management: Usually, administrators use imperative actions like creating, migrating, stopping VMs in their data center. However, some specific requirements can not be defined in a imperative way because they describe an infrastructure configuration that lasts for a certain period (i.e., an invariant). Imperative actions describe reconfiguration operations to perform whereas administrators want to describe a configuration without defining how to obtain it.

In the VMScript extension *btrScript*, declarative rules aim to specify VM placement and states with rules.

Placement rules implement in the btrScript language are a subset of the constraints used by Entropy [6]. The btrScript language supports the following rules:

- the *group* rule keeps the VMs on one physical server, for example, to optimize network connections. The following rule makes sure the VMs named *vm1*, *vm3* and *myvm* stay on the same server:

```
strongConnection: group [vm[1,3], myvm]
```

- the *spread* rule avoids the VMs to run on a same server (e.g., in a fault tolerance context). The following rule makes sure the VMs named *vm1*, *vm3* and *myvm* run on three servers:

```
dbReplica: spread [vm[1,3], myvm]
```

- the *on* rule forces a VM to run on specific servers. Administrators can restrict where VMs run to enforce the use of a specific hardware. The following rule makes sure the VM named *vm1* runs on the server *hostA* or on the server *hostB*:

```
vm1Host: vm1 on host[A,B]
```

- the *!on* rule is the negation of the *on* rule. It avoids a VM to run on specific servers. The use of *on* or *!on* operators rely on sets to describe. The smaller the set is, the easier is its description. The following rule makes sure the VM named *vm1* runs on a server different from *hostA* or *hostB*:

```
vm1Blacklist: vm1 !on host[A,B]
```

- the *run* rule forces VMs to run. It avoids to unfortunately stop or suspend VMs. The following rule makes sure VMs named *vm1* and *vm2* are in a *running* state.

```
alive: run vm[1,2]
```

These rules define more accurately the VM placement. As the data center architecture evolves (e.g., VMs are created, cpu consumptions fluctuate, etc.), rules can be added, enabled, disabled or deleted at run-time.

The previous rule declarations show rules that are defined and enabled at the same time. However, administrators can enable and disable rules as they wish, respectively, from the language operators *enable* and *disable*. In the below example, the *rules* property lists rules. Afterwards, rules are managed from their identifier.

```
blacklist: vm[1-100] !on host[A,B]
vm1:rules
<blacklist> vm[1-100] !on host[A,B] ON
blacklist:disable
vm1:rules
<blacklist> vm[1-100] !on host[A,B] OFF
blacklist:del
vm1:rules
```

2) Timed actions and rules: Reconfigurations in the infrastructure can occur at well-known dates such as the end of a development project or outside business hours. This language extension proposes to schedule actions, rule activations and rule inactivations. The date definition is close to the crontab syntax, with one additional parameter specifying the year of the execution.

Thanks to the date syntax, repetitive tasks such as VM reboot can be scheduled. The following operations stop VMs of the *myproject* vjob during the night every day:

```
[00:21:*:*:*:*] myproject/vm:stop
[00:8:*:*:*:*] myproject/vm:start
```

For a rule, an activation date and/or an inactivation date allow to automate the rule management. The first following declaration inserts the rule and wait the June 3rd 2011 at ten o'clock to enable it permanently. The second one inserts the rule, wait the June 3rd 2011 to enable it and disable it the June 6th 2011 at six pm.

```
[00:10:3:6:*:11] vm1_2: group [@vm1,@vm2]
[00:10:3:6:*:11-00:18:6:6:*:11] vm1_2: group [@vm1,@vm2]
```

IV. Ensuring the infrastructure consistency

Administrators manage virtualized infrastructures with placement rules and actions. Nevertheless, rule insertions must be checked to avoid inconsistencies that imply no correct VM placement exists.

Once rules are inserted, administrators need to know which rules are unsatisfied. The guardian module allows to warn administrators about infrastructure issues as broken rules.

Finally, infrastructure reconfigurations can not violate placement rules. And so, a rule verification occurs by simulating action effects on the model before each action execution.

A. Rule insertion verification

Administrators describe rules from the btrScript language. Before the insertion of a rule in the system, the placement module checks if this rule does not conflict with existing rules. A conflict leads to a resource organization with no viable placement possible. For example, one *on* rule enforcing 3 VMs, each one having 1 Gb of memory, to run on a server with 2 Gb of memory can never be satisfied. The conflict detection only considers static resources because dynamic resources quickly evolves and can not be predicted. Conflicts may also appear with the combination of several rules. The following example illustrates such a situation in an infrastructure composed of 3 servers (*pm1*, *pm2* ,and *pm3*):

```
together: group vm[1,2]
vm1Ban: vm1 !on pm[1-2]
vm2Ban: vm2 !on pm3
```

vm1 and *vm2* must be hosted on the same server. *vm1* must neither be hosted on *pm1* nor *pm2*, and *vm2* must not be hosted to *pm3*. This last rule produces a conflict as the VMs can not be hosted on any server because the group [*vm1*, *vm2*] is excluded from all servers [*pm1*, *pm2*, *pm3*].

The verification of *group* and *spread* rules ensures there is no intersection between *group* and *spread* VMs. For example, 2 placement rules define VM groups. The first one is composed by *vm1*, *vm2* and the second one by *vm1*, *vm3*. These 2 rules involves *vm1*, *vm2* and *vm3* must belong to the same server. So a *spread* rule with *vm2* and *vm3* can not be inserted.

The verification of *on* and *noton* rules ensures each VM or group can be hosted by, at least, one server. In the previous example, all *on* or *noton* rules about *vm1* modify the *vm2* and *vm3* placement. Consequently, the verification ensures at least one server can host the group *vm1*, *vm2* and *vm3*.

Moreover, server sets defined in both a *on* rule and a *noton* rule for a same VM must have an empty intersection. For example, the following rules are not correct:

```
vm1Host: vm1 on pm[1-3]
vm1Ban: vm1 !on pm2
```

Indeed, the server *pm2* is included in both rules, consequently, that does not represent what the administrator wants. This ambiguous declaration is not allowed.

When the rule module detects a rule insertion will lead to a conflict, the rule is not inserted and administrators receive a warning with rules that conflicts. Administrators can disable these rules to insert the new one.

When a scheduled rule is inserted, the rule module computes its activation period. It selects all the active rules that will be enabled during this period. If the rule to insert leads to a conflict, it is not inserted.

The rule module ensures the consistency of the set of rules but it does not check if a rule is true or false. So broken (i.e., false) rules can be inserted and administrators have to fix them in the future.

B. The infrastructure monitoring

In large infrastructures, administrators need information about the infrastructure architecture to maintain it. Monitoring systems like Ganglia [10] allow to collect lots of data from such an architecture in an efficient way. However, the large amount of VMs and servers and the number of metrics to observe is too huge to be analyzed by an administrator. In btrScript, a guardian module is proposed to analyze monitoring data and warn the administrator when it detects an overloaded server or a broken rule.

The guardian module periodically analyzes monitoring data. As rules can be added even if they are false, the guardian module periodically checks all activated rules and sends warnings to administrators when a broken rule is detected. Afterwards administrators can solve manually unsatisfied rules, remove them (e.g., for out-of-date rules) or use the placement module to solve the placement issues automatically.

C. Actions and placement rules

Administrators perform and schedule actions to reconfigure the infrastructure. As rules restrict the VM placement, when an imperative action is invoked or planned, rules are checked. This verification occurs by:

- selecting satisfied rules including servers and VMs involved in the action at the action execution date. If rules are not satisfied before the action execution, they are not included in the action verification ;
- simulating the action on the model ;
- checking selected rules.

If the action is not compatible with one of the selected rules, it is not executed.

V. The placement module

When overloaded servers (i.e., a server with a cpu consumption equals to 100%) or violated rules occur in the data center, administrators have to fix them. However, finding a VM placement considering rules and dynamic resources is a tedious task for hundreds or thousands of VMs and servers [9]. So btrScript is linked to a placement module in order to compute a list of actions required to obtain a right placement with respect to a scheduling policy, the placement rules and the physical and VM resources. The policy defines how to do the mapping between VMs and servers. Common policies for VM scheduling are load balancing and consolidation. The load balancing policy [11] distributes the cpu load uniformly across the servers whereas the consolidation policy [12] reduces the number of servers which host VMs. Placement rules customize the policy with specific needs described by administrators. The placement module is used for administrators to solve issues in the data center. In our case, the placement module is Entropy [5], a VM scheduler based on constraint programming. As btrScript does not implement its own placement algorithms, the choice of the placement module define the policy range. Entropy provides a *checker* policy, which satisfies cpu and memory requirements of VMs, and a *consolidation* policy.

Administrators invoke the placement module so as to solve issues (broken rules and overloaded servers) in the infrastructure. During this invocation, the placement module builds a problem that includes:

- A set of servers to analyze;
- A set of VMs corresponding to VMs that run on the server set;
- A set of rules to apply on the two previous sets.

From the btrScript model, the placement module obviously gets the actual VM placement and the resource usage required to solve the problem. Afterwards, this module computes a plan, that is to say a list of actions, and executes it. In the following example, the placement module solves issues in all the data center.

```
/server:solve checker
```

Due to the *run* rule, it is easy to start or resume VMs without specifying any server. For example, the VM *vm1* is in the *off* state. The administrator can start it with the placement module:

```
vm1r: run vm1
/server:solve checker
```

If the administrator wants to run the VM on servers *pm1* or *hostA*, he adds a *on* rule:

```
vm1r: run vm1
vm1On: vm1 on [ pm1, hostA ]
/server:solve checker
```

The application of one policy on all servers of the infrastructure is not always relevant. Administrators may want to consolidate VMs on a cluster and use load balancing on another one. Moreover, a huge problem is longer to solve than a small one. A plan for a small problem, that is to say 50 servers and 200 VMs, are solved in few seconds. For a larger problem (1000 servers and 5000 VMs), the solving time is few minutes [13].

Nevertheless, building a problem from a subset of server infrastructure is complex because the set of servers implies the set of rules added to the problem. Therefore the problem defined is smaller but some rules are cut for being integrated and so these rules are partially solved in the infrastructure context. As an example, a data center is composed by 3 servers (*pm1*, *pm2* and *pm3*) and 2 VMs (*vm1* and *vm2*). *vm1* runs on *pm1* and *vm2* runs on *pm2*. The administrator only adds one rule that enforces the VMs *vm1* and *vm2* to belong to the server *pm3*. As the rule is unsatisfied, the administrator decides to fix it by using the placement module. He invokes the placement module on *pm2* and *pm3* servers after adding the rule:

```
vm1On: vm[1,2] on pm3
pm[2,3]:solve checker
```

Consequently, the placement module considers servers *pm2* and *pm3*. The associated set of VMs only contains the VM *vm2* because *vm1* does not run on the server set. The rule is therefore cut and modified to the following rule:

```
vm2On: vm2 on pm3
```

So, the placement module executes a migration of *vm2* to *pm3*. The rule is still violated because *pm3* does not host *vm1*.

Rule modifications are mandatory to avoid side effects and transform a small problem to a big one. From the previous example, the hypothesis of the insertion of the whole rule in order to fix the *vm1* placement involves to add *vm1* and its host *pm1*. Now the placement module needs all VMs running on *pm1* and their rules to compute the plan. The rules of these VMs can add other servers and VMs and so on. At the end, the problem to solve can include all servers and VMs of the infrastructure.

VI. Evaluation

In this section, we present a comparison between the VMware vSphere PowerCLI and the btrScript language. Only few solutions can perform operations and add placement rules to virtualized infrastructures. The VMware vSphere solution includes the most popular distributed resource scheduler (DRS) that enables dynamic scheduling with two kinds of VM-to-VM rules: affinity and anti-affinity rules. Affinity rules keep VMs together on the same host and anti-affinity rules separate VMs on different hosts. The vSphere PowerCLI is a command-line and scripting tool based on PowerShell that provides useful functionality for vSphere management. Throughout the rest of the section we discuss about managing a rule (i) to the DRS from the PowerCLI and (ii) to btrScript. This rule ensures that the VMs named *proxy1*, *proxy2* and *proxy3* do not run on the same host. Two other VMs with similar names (*proxy4* and *proxy5*) exist in the infrastructure that is annoying to use regular expressions used by both PowerCLI and btrScript tools.

The syntax for the rule with the PowerCLI tool is:

```
New-DrsRule -Name Proxy -Cluster cl1 -KeepTogether:$false
    -VM Proxy1,Proxy2,Proxy3
```

The first parameter is the name of the rule to identify it. As rules are associated to a cluster in the DRS, we assume all VMs run to the cluster *cl1*. The *KeepTogether* parameter defines if the VMs must run on the same host ($true) or if VMs must run on different hosts ($false). At the end, VMs are selected from their name.

The syntax for the rule with the btrScript language is:

```
Proxy: Proxy[1-3] spread
```

In the btrScript tool, rules are not associated to a cluster and so the cluster name does not appear in the rule declaration. The operator *spread* is used instead of the 'KeepTogether' parameter. In the btrScript language, each rule has one operator to keep clear confusion. So, the *group* operator makes affinity rules and the *spread* operator makes anti-affinity rules. The syntax of the btrScript language allows regular expressions that is why the VM selection is shorter with btrScript.

The *New-DrsRule* command allows to enable or disable a rule at the declaration time. That is not allowed with btrScript. However, the DRS is an autonomic system where the administrator cannot solve rules when needed. In btrScript, the administrator explicitly calls the placement module enabling him to disable the rule before the rule resolution. Moreover, the rule modification is easier with btrScript than with PowerCLI. The syntax to add the VM *Proxy4* to the previous rule in PowerCLI is:

```
Set−DrsRule −Rule Proxy −VM Proxy1 , Proxy2 , Proxy3 , Proxy4
```

The administrator has to redefine the whole set of VMs. With btrScript, the administrator modify the VM set by including or excluding VMs:

```
Proxy:vms + Proxy4
```

If a rule is added or modified and it conflicts with another rules the vSphere policy is to disable the new one. As this management policy can hide issues, in btrScript, the insertion of a rule that conflicts with other ones is canceled and a notification showing conflicted rules is sent to the administrator. If he wants to insert the new rule, he has to disable conflicted rules.

Affinity rules between a group of VMs and a group of hosts also exist in vSphere. This VM-to-Host rules correspond to the *on* and *noton* rules in btrScript. Nevertheless, their manipulation from the PowerCLI [14] is more complicated than the rules above described. Moreover, there is no conflict detection for this kind of rules. So, the administrator can insert one rule and its opposite without receiving any notification.

When an action is invoked by the vmWare administrator, placement rules are not verified. So, actions can violate rules. In btrScript, administrators can not execute an action that violates one or more active rules. An error about the broken rules is reported.

To conclude, brtScript and PowerCLI propose a similar approach of placement rules. However, the rule management is easier from btrScript thanks to advanced selection mechanisms and more verifications, especially those on actions.

VII. Related Work

A. Business Rule Manager System (BRMS)

BRMS, like Drools [15], allows to set business rules to manage a system. However, these rules are simple with the syntax: “when something is true, do these operations". In btrScript, placement rules do not define actions to execute if the rule is not satisfied because rule satisfactions depend on the resource organization and other rules.

B. Virtual machine manager

VMWare [2] [4] can manage servers and VMs. Placement rules (called affinity rules in the VMWare documentation) are used to restrict placement between VMs and servers. These affinity rules include required and preferential rules. Required rules are similar to the btrScript rules and preferential rules can be violated to allow the proper functioning of the VMWare placement module. Preferential rules are excluded from btrScript but the definition of affinity rules from the VSphere GUI is not appropriate for managing large infrastructures. Select thousand VMs and servers in a GUI is not relevant. Administrators have to script themselves functions to add, remove and list affinity rules through the one of the vSphere API. Moreover, the consistency checking must be added for scheduled rules. In btrScript, these functions are integrated into the language operators. Further activation period for rules are not designed and VMWare actions does not take care of placement rules whereas btrScript does.

OpenNebula [16] is an open source toolkit for cloud computing designed to manage a large amount of VMs. A placement module called mm_sched (i.e., match making scheduler) allows to choose three different policies (compared with two policies implemented in btrScript) for the VM placement. However, these policies can not be tuned with specific rules.

C. Domain Specific Languages (DSL)

Puppet [17] [18] is a declarative configuration language for auditing and configuring large infrastructures (with virtualization or not) from one centralized node. A visual dashboard and reporting tools monitor servers to report every change. Puppet deploys large infrastructures but, at run-time, there is a lack of operations to handle servers or VMs.

VGrADS [19] and its virtual grid execution system allows to describe jobs with vgDL [20] and run them with time constraints. These tools address issues about deploying and scheduling jobs but, like Puppet, it is not designed to handle and perform reconfigurations on VMs and servers.

The former tools are designed to deploy and use resources of a virtual infrastructure while btrScript handles resources after deployment like Usher [21]. Usher is a shell for VM management. It is designed to local management and so it does not provide information about the whole grid.

Plasma [6] is the Entropy DSL to add constraints. It allows to define constraints by selecting VMs and servers from their name. However, physical and logical hierarchies do not exist and there is no selection on element properties. Moreover, no operation exists in the language to perform actions or query resource utilization.

VIII. Conclusion

Our paper presented btrScript, a safe management system for virtualized data center. It focuses on secure scheduled actions and placement rules. Scheduled actions allow to plan in advance tasks and schedule repetitive tasks for the automation of administrative tasks. Placement rules, which can be scheduled too, allow administrators to define the VM placement more accurately. These rules also restrict scheduled and immediate actions. A guardian module monitors the data center and reports issues such as overloaded servers and violated rules. We introduced a placement module named *Entropy* that enables to compute a plan with respect to rules and dynamic virtual and physical resources. This module can solve issues on all the data center or on a specific designated part. The evaluation is a comparison between the command line interface, PowerCLI, that can insert placement rules in the vmWare vSphere client. The btrScript system provides safer management by checking rules when an action is invoked and detecting more contradiction than the vSphere client.

Future work focus on running btrScript in a larger virtual infrastructure. BtrScript only runs with small architectures (20 servers) based on the kvm hypervisor. The use of the experimental platform grid5000 [22] is planned to set up a large infrastructure.

IX. Acknowledgements

This work is partially funded by the SelfXL ANR/ARPEGE project (http://selfxl.gforge.inria.fr/dokuwiki/doku.php). We would like to thank Rémi Sharrock for their contributions.

References

[1] J. Hamilton, "Sigmod keynote," 2011, "Each day Amazon Web Services adds enough new capacity to support all of Amazon.com's global infrastructure through the company's first 5 years, when it was a $2.76B annual revenue enterprise".

[2] VMWare, "VMWare Infrastructure: Resource Management with VMWare DRS," VMWare, Tech. Rep., 2006.

[3] OpenStack, "Openstack web site," March, 2012, http://openstack.org/.

[4] VMware, "What's new in vmware vsphere 4.1 - availability and resource management," VMWare, Tech. Rep., 2010.

[5] F. Hermenier, A. Lèbre, and J.-M. Menaud, "Cluster-wide context switch of virtualized jobs," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 658–666.

[6] F. Hermenier, J. Lawall, J.-M. Menaud, and G. Muller, "Dynamic Consolidation of Highly Available Web Applications," INRIA, Research Report RR-7545, 02 2011.

[7] A. Computing, "An architectural blueprint for autonomic computing." *Quality*, vol. 36, no. June, p. 34, 2006.

[8] R. Pottier, M. Léger, and J.-M. Menaud, "A reconfiguration language for virtualized grid infrastructures," in *10th IFIP international conference on Distributed Applications and Interoperable Systems (DAIS)*, vol. 6115, June 2010, pp. 42–55.

[9] R. Sirdey, J. Carlier, H. Kerivin, and D. Nace, "On a resource-constrained scheduling problem with application to distributed systems reconfiguration," *European Journal of Operational Research*, vol. 183, no. 2, pp. 546 – 563, 2007.

[10] M. L. Massie, B. N. Chun, and D. E. Culler, "The Ganglia Distributed Monitoring System: Design, Implementation, and Experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, July 2004.

[11] A. Goel and P. Indyk, "Stochastic load balancing and related problems," in *Foundations of Computer Science, 1999. 40th Annual Symposium on*, 1999, pp. 579 –586.

[12] A. Beloglazov and R. Buyya, "Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers," in *Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science*, ser. MGC '10. New York, NY, USA: ACM, 2010, pp. 4:1–4:6.

[13] F. Hermenier, S. Demassey, and X. Lorca, "Bin repacking scheduling in virtualized datacenters," in *Proceedings of the 17th international conference on Principles and practice of constraint programming*, ser. CP'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 27–41.

[14] A. van Lieshout, "Managing vmware drs rules using powercli," March, 2012, http://www.van-lieshout.com/2011/06/drs-rules/.

[15] C. L. Forgy, "Expert systems," in *Expert systems*, P. G. Raeth, Ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990, ch. Rete: a fast algorithm for the many pattern/many object pattern match problem, pp. 324–341.

[16] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster, "An Open Source Solution for Virtual Infrastructure Management in Private and Hybrid Clouds," *IEEE Internet Computing*, vol. 13, no. 5, pp. 14–22, 2009.

[17] L. Kanies, "Puppet: Next-generation configuration management," *j-LOGIN*, vol. 31, no. 1, Feb. 2006.

[18] D. Edwards*, A. Schafer†, T. Tyree†, A. Shortland*, A. Honor‡, and L. Thompson*, "Web ops 2.0: Achieving fully automated provisioning," DTO Solutions* and Puppet Labs† and ControlTier Project‡, Tech. Rep., 2009.

[19] L. Ramakrishnan, C. Koelbel, Y. S. Kee, R. Wolski, D. Nurmi, D. Gannon, G. Obertelli, A. YarKhan, A. Mandal, T. M. Huang, K. Thyagaraja, and D. Zagorodnov, "Vgrads: enabling e-science workflows on grids and clouds with fault tolerance," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–12.

[20] Y.-s. Kee and C. Kesselman, "Grid resource abstraction, virtualization, and provisioning for time-targeted applications," in *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 324–331.

[21] M. McNett, D. Gupta, A. Vahdat, and G. M. Voelker, "Usher: an extensible framework for managing custers of virtual machines," in *LISA'07: Proceedings of the 21st conference on Large Installation System Administration Conference*. Berkeley, CA, USA: USENIX Association, 2007, pp. 1–15.

[22] Grid5000, "Grid5000 web site," March, 2012, https://www.grid5000.fr.

RFDMon: A Real-time and Fault-tolerant Distributed System Monitoring Approach

Rajat Mehrotra
Electrical and Computer Engineering
Mississippi State University
Mississippi State, MS, USA
Email: rm651@msstate.edu

Sherif Abdelwahed
Electrical and Computer Engineering
Mississippi State University
Mississippi State, MS, USA
Email: sheirf@ece.msstate.edu

Abhishek Dubey
Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN, USA
Email: dabhishe@isis.vanderbilt.edu

Krisa W. Rowland
US Army Engineer Research and Development Center
Vicksburg, MS, USA
Email: Krisa.W.Rowland@usace.army.mil

***Abstract*—One of the main requirements for building an autonomic system is to have a robust monitoring framework. In this paper, a systematic distributed event based (DEB) system monitoring framework "RFDMon" is presented for measuring system variables (CPU utilization, memory utilization, disk utilization, network utilization, etc.), system health (temperature and voltage of Motherboard and CPU) application performance variables (application response time, queue size, and throughput), and scientific application data structures (PBS information and MPI variables) accurately with minimum latency at a specified rate and with controllable resource utilization. This framework is designed to be tolerant to faults in monitoring framework, self-configuring (can start and stop monitoring the nodes and configure monitors for threshold values/changes for publishing the measurements), aware of execution of the framework on multiple nodes through HEARTBEAT messages, extensive (monitors multiple parameters through periodic and aperiodic sensors), resource constrainable (computational resources can be limited for monitors), and expandable for adding extra monitors on the fly. Since RFDMon uses a Data Distribution Services (DDS) middleware, it can be used for deploying in systems with heterogeneous nodes. Additionally, it provides a functionality to limit the maximum cap on resources consumed by monitoring processes such that it reduces the effect on the availability of resources for the applications.**

***Keywords*- Distributed Monitoring; ARINC-653; and Data Distribution Services.**

I. Introduction

Autonomic distributed computing infrastructure imposes requirements of consistency, synchronization, and security over multiple nodes. Additionally, in enterprise domain, there is a tremendous pressure to achieve quality of service (QoS) objectives in all possible scenarios of the system operation. To this end, an aggregate picture of the distributed infrastructure should always be available to analyze and to provide feedback for computing control commands if needed. The desired aggregate picture can be achieved through an infrastructure monitoring system that is extensive enough to accommodate system parameters, application performance, and application data structures.

The desired monitoring technique should also be customizable for various types of applications and their wide range of parameters. Autonomic management of distributed systems requires an effective monitoring technique, which can work in a distributed manner similar to the underlying system and reports each event to the system administrator with maximum accuracy and minimum latency. Furthermore, the monitoring system should not be a performance burden or interfere with the applications executing in the system. Also, the monitoring system should be robust and should be able to identify the internal faults to isolate the faulty component and correct it immediately. However, typical distributed monitoring techniques suffer from synchronization issues among the nodes, communication delay, non deterministic nature of events, large amount and asynchronous nature of measurements, and limited bandwidth. All of these issues may result into inconsistent global view of the infrastructure.

This paper describes a distributed monitoring approach **"RFDMon"** that utilizes the concepts of OMG Data Distribution Services [1] for an efficient exchange of measurements among the computing nodes. This monitoring framework is built upon the ACM: ARINC-653 Component Framework [2] and an open source DDS implementation, Open splice [3]. The primary principles in design of the proposed approach are static memory allocation for determinism, spatial isolation between monitoring framework and real applications for fault containment, specification, and adherence to real time properties such as periodicity and deadlines in monitoring framework.

Experimental results show that "RFDMon" has small overhead on the computational resources of the system. It also identifies the faults in infrastructure and in itself with minimum delay, and reconfigures itself to resume the monitoring of infrastructure without further delay. "RFDMon" can be combined easily with a fault diagnosis module due to its standard interfaces.

Currently, "RFDMon" is installed at Fermi Lab, Batavia, IL for monitoring scientific clusters, which consist of 100 to 800 computing nodes [4]. This approach is utilized in data processing within the Large Quantum Chromo Dynamics (LQCD) project at Fermi Lab [4]. This data processing is carried as analysis campaigns (scientific workflows) that consists of an input dataset and a set of interdependent processing steps (named as jobs). These jobs are executed over large commodity computer clusters. These large clusters can result in systematic failure if operated over a long continuous period of time executing these analysis campaigns. Typical execution time of a campaign may span several months and it executes hundreds of data and computational intensive parallel MPI Jobs that require several computational nodes during its lifetime. A campaign can fail even by failure of a single job. "RFDMon" is used by administrators to diagnose job problems and failures in this complex environment and quickly respond to the intermittent faults.

This paper is organized as follows. Preliminary concepts of the proposed approach are presented in Section II. Related distributed monitoring products are described in Section III and detailed description of the proposed approach is given in Section IV. Details of each sensor is presented in Section V and a set of system monitoring experiments is described in Section VI. The major benefits of the approach is highlighted in Section VII and applications of the approach is described in Section VIII. Conclusions are presented in Section IX.

II. Preliminaries

The proposed monitoring approach **"RFDMon"** consists of two major modules: **Distributed Sensors Framework** and **Infrastructure Monitoring Database**. Distributed Sensors Framework utilizes Data distribution services (DDS) middle-ware standard for communication among the nodes of distributed infrastructure. It uses Opensplice Community Edition [3]. It executes DDS sensor framework on top of ARINC Component Framework [5]. Infrastructure Monitoring Database uses Ruby on Rails [6] to implement a web service that is used to update the database with monitoring data and to display the data on administrator web browser. In this section, the primary concepts of DDS, ARINC-653, and Ruby on Rails are briefly presented.

Data Distribution Services: Data Distribution Service (DDS) specifications are defined by Object Management Group [7] for communication in distributed real-time systems through publish-subscribe mechanism. This mechanism overcomes the typical shortcomings of client-server model, where client and servers are tightly coupled. Each message is associated with a topic. In DDS, publishers and subscribers are not coupled to each other. Publishers or subscribers need only the *name* and *definition* of the data in order to communicate. Publishers do not need any information about the *location* or *identity* of the subscribers, and vice versa.

ARINC-653: ARINC-653 software specification has been utilized in safety-critical real time operating systems (RTOS) that are used in avionic systems and recommended for space missions [8]. The specification specifies the OS interfaces and its associated services to ensure spatial and temporal separation among various applications for fault-containment in integrated modular avionics [9]. Spatial partitioning ensures exclusive use of a memory region by an application. It guarantees that a faulty process in a partition cannot corrupt or destroy the data structures of other processes that are executing in other partitions. This space partitioning is useful to separate the low-criticality vehicle management components from safety-critical flight control components in avionics systems [2]. Temporal partitioning ensures sharing of computational resources through fixed periodic schedule among multiple applications. ARINC-653 Emulation Library [2] (available to download from [10]) provides a UNIX based implementation of ARINC-653 interface specifications. This scheduling scheme guarantees that a partition will relinquish the CPU after its execution duration has expired.

Ruby on Rails: Rails [6] is a web application development framework that uses Ruby programming language [11]. Rails uses Model View Controller (MVC) architecture for application development [12]. In "RFDMon", a web service is developed to display the monitoring data collected from the distributed infrastructure. These monitoring data includes the information related to clusters, nodes in a cluster, node states, measurements from various sensors on each node, MPI and PBS related data for scientific applications, web application performance, and process accounting. Currently, "RFDMon" is using MYSQL open source database. Other databases (e.g., Sqlite, PostgreSQL, etc) can also be used as per the support in Ruby on Rails. Schema information of the database is shown in [13].

III. Other Distributed Monitoring Systems

Various distributed monitoring systems have been developed by industry and research groups in past many years. Ganglia [14], Nagios [15], Zenoss [16], and Nimsoft [17] are a few most popular enterprise products developed for monitoring distributed systems.

Ganglia [14] is developed upon the concept of hierarchical federation of clusters. In this architecture, multiple nodes are grouped as a cluster which is attached to a module, and then multiple clusters are again grouped under a monitoring module. Nodes and applications utilize a multi-cast based listen/announce protocol for sending their measurements to all of the other nodes. The primary advantage of Ganglia is auto-discovery of the nodes, easy portability and manageability, and aggregation of cluster measurements at each node. **Nagios** [15] is developed upon plug-in based agent/server architecture, where agents can report the abnormal events from the computing nodes to the server node (administrators) through email, SMS, or instant messages. Nagios consists of three components - Scheduler: This is the administrator component that checks the plug-ins and take corrective actions if needed. Plug-in: These small modules are placed on the computing node, configured to monitor a resource, and then send the reading to the "Nagios" server module over SNMP interface. GUI: This is a web based interface that presents the measurements from the system through various colourful buttons, sounds, and graphs.

Zenoss [16] is a model-based monitoring solution that has comprehensive and flexible approach of monitoring with an extremely detailed GUI interface. It is an agentless monitoring approach where central monitoring server collects measurements from each node over SNMP interface through ssh commands. In Zenoss, the computing nodes can be discovered automatically and specified with their types (Xen, VMWare, etc.) that ensures appropriate and complete monitoring using pre-defined templates, thresholds, and event rules. **Nimsoft Monitoring Solution** [17](NMS) offers a light-weight, reliable, extensive, and GUI based monitoring of the entire infrastructure. NMS uses a message BUS for exchange of messages among the applications residing in the infrastructure. These applications (or components) are configured with the help of a software component (HUB) and are attached to the message BUS. Monitoring action is performed by small probes and the measurements are published to the message BUS by software components (ROBOTS) deployed over each managed device. NMS also provides an Alarm Server for alarm monitoring and a GUI portal to visualize the comprehensive view of the system.

These distributed monitoring approaches are significantly scalable in number of nodes, responsive to changes at the nodes, and comprehensive in number of parameters. However, these approaches do not support capping of the resource consumed by the monitoring framework, fault containment in monitoring unit, and expandability of the monitoring approach for new parameters in the already executing framework. Additionally, these approaches are stand-alone and are not easily extendible to associate with other modules that can perform fault diagnosis for the infrastructure at different granularity (application level, system level, and monitoring level). Furthermore, they work in a server/client or host/agent manner (except NMS) that requires direct coupling of two entities, where one entity has to be aware of the location and identity of the other entity.

Therefore, “RFDMon” utilizes the data distribution service (DDS) methodology to report the events or monitoring measurements from each node to a central node in a decoupled manner. In “RFDMon”, all monitoring sensors execute on the ARINC-653 Emulator [2]. This enables the monitoring agents to be organized into one or more partitions and each partition has a fixed periodic schedule to use the processing resources (temporal partitioning). The processes executing under each partition can be configured for real-time properties (priority, periodicity, duration, soft/hard deadline, etc.). Additionally, ARINC-653 uses spatial partitioning [9] that ensures exclusive use of a memory region by an ARINC partition and introduces fault containment property in the monitoring framework. The details of the approach are described in later sections of the paper.

IV. Architecture of the Framework

The proposed monitoring framework “RFDMon” is based upon data centric publish subscribe communication mechanism. Modules (or processes) in the framework are separated from each other through concept of spatial and temporal locality as described in section II. Architecture of “RFDMon” is shown in Figure 1. The proposed framework has following key concepts and components.

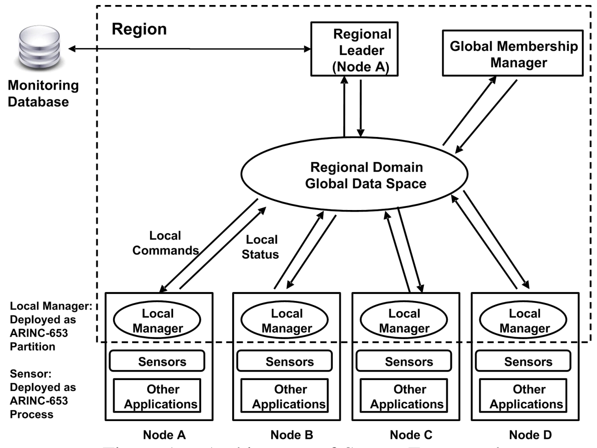

Region
Monitoring Database
Regional Leader (Node A)
Global Membership Manager
Regional Domain Global Data Space
Local Commands
Local Status
Local Manager: Deployed as ARINC-653 Partition
Sensor: Deployed as ARINC-653 Process
Local Manager
Sensors
Other Applications
Node A
Node B
Node C
Node D

Figure 1. Architecture of Sensor Framework

A. *Sensors*

Sensors are the primary component of the framework. These are lightweight processes that monitor a device on the computing nodes and read it periodically or aperiodically to get the measurements. These sensors publish the measurements under a topic (described in next subsection) to DDS domain. There are various types of sensors in the framework which are described in section V.

B. *Region*

The proposed monitoring framework organizes the nodes in regions (or clusters). Nodes can be homogeneous or heterogeneous. Nodes are combined only logically. These nodes can be located in a single server rack or on single physical machine (in case of virtualization). However, physical closeness is recommended to combine the nodes in a single region to minimize the unnecessary communication overhead in the network.

C. *Local Manager*

Local Manager is a module that is executed as an agent on each computing node of the monitoring framework. These agents are executed on each node with knowledge about its pre-defined region name. However, it is not provided with any information related to other nodes or configuration of the region. The primary responsibility of the local manager is to set up sensor framework on the node.

D. *Regional Leader*

Among multiple local manager nodes that belongs to same region, there is a local manager node which is selected as “Regional Leader” for updating the monitoring database for sensor measurements from each local manager. Regional leader will also be responsible for updating the changes in state (UP, DOWN, FRAMEWORK_DOWN) of various local manager nodes. Each local manager is supplied with the pre-defined URLs to Ruby on Rails web service for database updates and update is done over http interface using **“libcurl”** library [18]. However, these URLs are used to update the database only when a local manager is selected as regional leader. Once a regional leader terminates, a new leader will be selected for the region. Selection of the leader is done by Global Membership Manager module as described in Section IV-G.

E. *Topics*

Topics are the primary unit of information exchange in DDS domain. Details about the type of topic (structure definition) and key values (keylist) to identify the different instances of the topic are described in interface definition language (IDL) file. CORBA IDL files are used to promote the interoperability among the monitoring frameworks developed in different programming languages (e.g., C, C++, Java, etc.) using the same interface. Keys can represent arbitrary number of fields in the topic. These topics are categorized in following categories based upon their content.

MONITORING_INFO: System resource and hardware health monitoring sensors publish measurements under this topic.

HEARTBEAT: Heartbeat Sensor uses this topic to publish its heartbeat in the DDS domain to notify the framework that the node is attached to the framework. All nodes which are listening to HEARTBEAT topic can keep track the health condition of other nodes in the framework through this topic.

NODE_HEALTH_INFO: When a Regional Leader node (defined in Section IV-D) detects change in state (UP, DOWN, or FRAMEWORK_DOWN) of any other node by observing the change in node's heartbeat, it publishes a message with NODE_HEALTH_INFO topic to notify all other nodes regarding change in status of the node.

LOCAL_COMMAND: This topic is used by the Regional Leader to send the control commands to other local nodes for START, STOP, or POLL the sensors.

GLOBAL_MEMBERSHIP_INFO: This topic is used for communication between local nodes and Global Membership Manager (defined in Section IV-G) for selection of Regional Leader and for providing information related to existence of the leader.

PROCESS_ACCOUNTING_INFO: Process accounting sensor reads records from the process accounting system and publishes the records under this topic.

MPI_PROCESS_INFO: This topic is used to publish the execution state (STARTED, ENDED, KILLED) and MPI or PBS information variables of MPI processes executing on the computing node.

WEB_APPLICATION_INFO: This topic is used to publish the performance measurements of a web application that contains information logged from the web service related to average response time, heap memory usage, number of JAVA threads, and pending requests inside the system.

F. *Topic Managers*

Topic Managers are classes that create subscriber or publisher for a pre-defined topic. This publisher publishes the data received from various sensors under the same topic name. Subscriber receives data from the DDS domain under the same topic name and delivers it to underlying application for further processing.

G. *Global Membership Manager*

Global Membership Manager (GMM) module is responsible to maintain the membership of each node for a particular region and for selection of a Regional Leader. Once a local node comes alive, it first contacts the GMM module with node's region name using topic GLOBAL_MEMBERSHIP_INFO to get the information regarding Regional Leader. GMM module replies with the name of Regional Leader (if leader exists) or assign the new node as Regional Leader. GMM module updates the leader information in file ("REGIONAL_LEADER_MAP.txt") on disk in colon separated format (RegionName:LeaderName). When a local node sends message to GMM module that its leader is dead, GMM module selects a new leader for that region and replies to the Local Node with leader name.

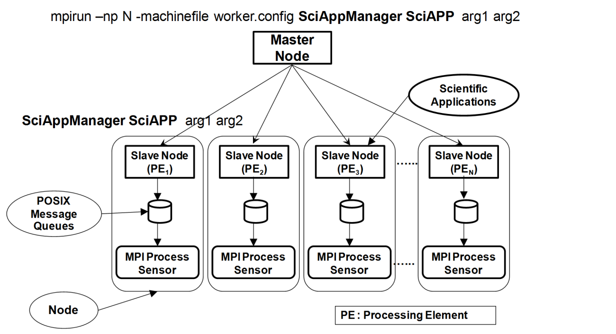


Figure 2. Architecture of Scientific Application Health Monitoring Sensor

This leader re-election functionality enables the fault tolerant nature in the framework with respect to regional leader that ensures periodic update of the infrastructure monitoring database with measurements even in case of leader failure. The leader selection for the region is performed by a single GMM module that ensures that there will be only one leader of a region. Because the leader selection or re-selection is performed by communication between only two nodes, this process is unaffected by the size of the region. Communication delay of message exchange in DDS domain is the only factor that can delay the leader selection process. Additionally, other more sophisticated algorithms can be easily plugged into the framework by modifying the GMM module for leader selection.

GMM module is executed through a wrapper executable "**GMM_Monitor**" as a child process. "GMM_Monitor" keeps track of execution state of the GMM module and starts a fresh instance of GMM module if previous instance terminates due to some error. New instance of the GMM module receives updated data from "REGIONAL_LEADER_MAP.txt" file. It provides the fault tolerant abilities in the framework with respect to GMM module.

V. Sensor Implementation

The proposed monitoring framework implements various software sensors to monitor system resources, network resources, node states, MPI and PBS related information, and performance of web applications (see Table I).

These sensors are executed as a ARINC-653 process on top of the ARINC-653 emulator [2]. All sensors on a node are deployed in a single ARINC-653 partition on top of the ARINC-653 emulator. The ARINC-653 emulator monitors the deadline and schedules the sensors such that their periodicity is maintained. Furthermore, the emulator performs static cyclic scheduling of the ARINC-653 partition of the

Sensor Name	Period	Description
CPU Utilization	30 seconds	Aggregate utilization of all CPU cores on the machines.
Swap Utilization	30 seconds	Swap space usage on the machines.
Ram Utilization	30 seconds	Memory usage on the machines.
Hard Disk Utilization	30 seconds	Disk usage on the machine.
CPU Fan Speed	30 seconds	Speed of CPU fan that helps keep the processor cool.
Motherboard Fan Speed	10 seconds	Speed of motherboard fan that helps keep the motherboard cool.
CPU Temperature	10 seconds	Temperature of the processor on the machines.
Motherboard Temperature	10 seconds	Temperature of the motherboard on the machines.
CPU Voltage	10 seconds	Voltage of the processor on the machines.
Motherboard Voltage	10 seconds	Voltage of the motherboard on the machines.
Network Utilization	10 seconds	Bandwidth utilization of each network card.
Network Connection	30 seconds	Number of TCP connections on the machines.
Heartbeat	30 seconds	Periodic liveness messages.
Process Accounting	30 seconds	Periodic sensor that publishes the commands executed on the system.
MPI Process Info	-1	Aperiodic sensor that reports the change in state of the MPI Processes.
Web Application Info	-1	Aperiodic sensor that reports the performance data of Web Application.

Table I
LIST OF MONITORING SENSORS

local manager. The schedule is specified in terms of a hyper period, the phase and the duration of execution in that hyper period [13]. Effectively, it limits the maximum CPU utilization of the local managers.

Sensors are constructed with following attributes:

Name: Name of the sensor (e.g., UtilizationAggregatecpuScalar).

Source Device: Name of the device to monitor for the measurements (e.g., "/proc/stat").

Period: Periodicity of the sensor (e.g., 10 seconds for periodic sensors and -1 for aperiodic sensors).

Deadline: A sensor has to finish its work within a specified deadline. A HARD deadline violation is an error that requires intervention from the underlying middle-ware while a SOFT deadline violation results in a warning.

Priority: Sensor priority indicates the priority of scheduling the sensor over other processes in to the system. In general, normal (base) priority is assigned to the sensor.

Dead Band: Sensor reports the value only if the difference between current value and previous recorded value becomes greater than the specified sensor dead band. It reduces the number of sensor measurements in the DDS domain if sensor measurement is changing slightly.

Sensors support three types of commands for publishing the measurement: START, STOP, and POLL. START command starts the already initialized sensor to start publishing the sensor measurements. STOP command stops the sensor thread to stop publishing the measurement. POLL command tries to get the latest measurement from the sensor. Sensors publish the data as per the predefined topic to the DDS domain (e.g., MONITORING_INFO). Sensors are categorized based upon their functionality as follows.

System Resource Utilization Monitoring Sensors: These sensors monitor utilization of the system resources: CPU, RAM, Disk, Swap, and Network. These sensors (periodic in nature), follow SOFT deadlines, contain normal priority, and provide monitoring of system devices (e.g., /proc/stat) to collect the measurements. These sensors publish the measurements under MONITORING_INFO topic.

Hardware Health Monitoring Sensors: These sensors monitor health of the system hardware components: CPU Fan Speed, CPU Temperature, Motherboard Temperature, and Motherboard voltage. These sensors are periodic.Theses follow soft deadlines, contain normal priority, and read the measurements over Intelligent Platform Management Interface (IPMI) interface [19]. These sensors publish the measurements under MONITORING_INFO topic.

Node Health Monitoring Sensors: Each local manager executes a Heartbeat sensor that periodically sends its own node name to DDS domain under topic "HEARTBEAT" to inform other nodes regarding its existence in the framework.

Scientific Application Health Monitoring Sensor: This Sensor logs the information in case of state change (Started, Killed, Ended) of the processes related to scientific applications and reports the data to the centralized database. In the proposed framework, a wrapper application (**SciAppManager**) is developed that can execute the real scientific applications (e.g., **SciAPP** in Figure 2) internally as a child process. MPI "run command" is issued to execute **SciAppManager** application from master nodes in the cluster (see Figure 2). SciAppManager writes the state information of scientific application in a POSIX message queue that exists on each node. Scientific application sensor will listen on that message queue and publishes message to the DDS domain under MPI_PROCESS_INFO topic.

Web Application Performance Monitoring Sensor: This sensor keeps track of performance behaviour of the web application executing over the node through the web server performance logs written to a POSIX message queue (different from SciAppManager). This sensor will listen on that message queue and publishes the message to the DDS domain under WEB_APPLICATION_INFO topic.

VI. EXPERIMENTS

A set of experiments have been performed to exhibit the system resource overhead, fault adaptive nature, and responsiveness towards fault in the developed monitoring framework. During these experiments, the monitoring framework is deployed in a Linux environment (2.6.18-274.7.1.el5xen) that consists of five nodes (ddshost1, ddsnode1, ddsnode2, ddsnode3, and ddsnode4). Ruby on Rails based web service and MYSQL database are hosted on ddshost1 node. These experiments have been performed to measure the impact of executing monitoring framework over the computational

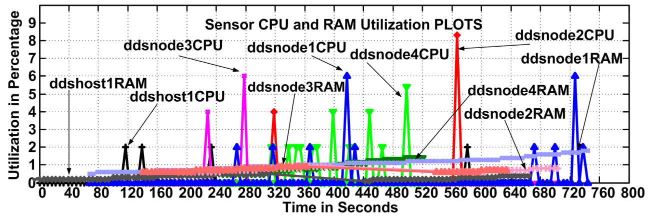


Figure 3. CPU and RAM Utilization by the Sensor Framework at Nodes

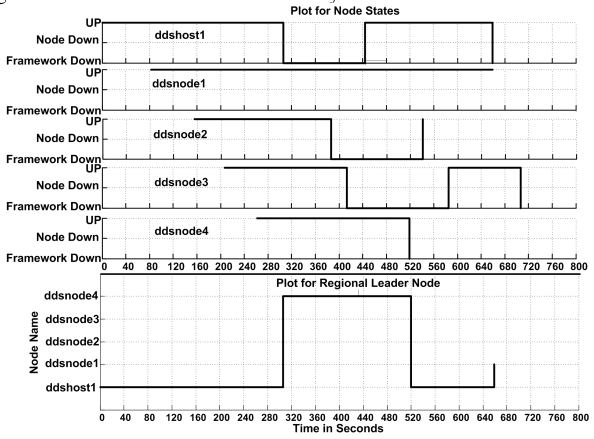


Figure 4. State Transition of Nodes and Leaders of the Sensor Framework.

resources (e.g., CPU and RAM) of system and to display the fault tolerant and self-configure properties of the framework in case of failures in the framework itself.

In one of these experiments, all of the nodes (ddshost1, and ddsnode1..4) are started one by one with a random time interval. Once all the nodes have started executing the framework, local manager on a few nodes are killed through **KILL** system call. During this experiment, the CPU and RAM consumption by local manager at each node is monitored through "TOP" system command. Results from the experiment are presented in Figures 3, 4, and 5.

Figure 3 describes the CPU and RAM utilization by monitoring framework (local manager) at each node during the experiment. It is evident from Figure 3 that CPU utilization is mostly in the range of 0 to 1 percent with occasional spikes. However, even in case of spikes, CPU utilization is under ten percent. Similarly, RAM utilization by the monitoring framework is less than even two percent. These results clearly indicates that overall resource overhead of the developed monitoring approach "RFDMon" is extremely low. As mentioned earlier, it is possible to cap this resource usage by specifying the hyper period and duration of execution of the local manager within the hyper period. However, due to space constraints this experiment is not shown in the paper. More details are available in [13].

Transition of various nodes between states UP and FRAMEWORK_DOWN is shown in Figure 4. According to the figure, ddshost1 is started first, then followed by ddsnode1, ddsnode2, ddsnode3, and ddsnode4. ddshost1 is selected as the regional leader in the beginning. At time sample 310 (approximately), local manager of host ddshost1 was killed, therefore its state has been updated to FRAMEWORK_DOWN. Similarly, state of ddsnode2 and ddsnode3 is also updated to FRAMEWORK_DOWN once

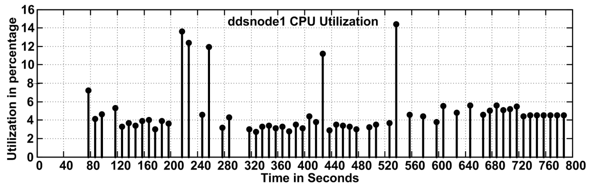


Figure 5. CPU Utilization at node ddsnode1 during the Experiment

their local manager is killed on time sample 390 and 410 respectively. local manager at ddshost1 is again started at time sample 440; therefore its state is updated to UP at the same time. Figure 4 also represents the nodes which were regional leaders during the experiment. According to the figure, initially ddshost1 was the leader of the region, while as soon as local manager at ddshost1 is killed at time sample 310 (see Figure 4), ddsnode4 is elected as the new leader of the region (as per the procedure specified in Section IV-G). Similarly, when local manager of the ddsnode4 is killed at time sample 520 (see Figure 4), ddshost1 is again elected as the leader of the region. From Figure 4, it is clearly evident that as soon as there is a fault in the framework related to the regional leader, a new leader is elected instantly without any further delay. This specific feature of the monitoring framework exhibit that it is robust with respect to failures of the regional leader and it can adapt to the faults in the framework instantly without delay.

Sensor framework at ddsnode1 was allowed to execute during the complete experiment (see Figure 4) and no fault was introduced in this node. The primary purpose of executing this node continuously was to observe the impact of introducing faults in the framework over monitoring capabilities of the framework. In the most ideal scenario, entire monitoring data of ddsnode1 should be reported to the centralized database without any interruption even in case of faults (leader re-election and nodes going out of the framework). Figure 5 shows the CPU utilization of ddsnode1 from the centralized database as reported by regional leader through CPU monitoring sensor from ddsnode1. According to the Figure 5, monitoring data from ddsnode1 was collected successfully during the entire experiment. Even in case of Regional Leader re-election at time sample 310 and 520 (see Figure 4), only one or two (max) data samples are missing from the database (see Figure 5). Henceforth, it is evident that there is a minimal impact of faults in the framework over the monitoring functionality of the framework.

VII. Benefits of the approach

"RFDMon" can monitor system resources, hardware health, node availability, MPI job state, and application performance data in a comprehensive manner. "RFDMon" is easily scalable with the number of nodes because it is based upon data centric publish-subscribe mechanism. Publish-subscribe mechanism is extremely scalable with respect to number of nodes. Also, in proposed framework, new sensors can be easily added to increase the number of monitoring parameters. It is fault tolerant with respect to faults in the framework due to partial outage (if regional leader or global membership manager terminates). It can self-configure (Start, Stop, and Poll) the sensors and can be applied in the heterogeneous environment. The major benefit of using this framework is that the total resource consumption by the sensors can be limited by applying ARINC-653 scheduling policies and due to spatial isolation features of ARINC-653 emulation, monitoring framework will not corrupt the memory area or data structures of applications in execution on the node. Additionally, framework has a small computational overhead.

VIII. Application of the Framework

The initial version of the proposed approach was combined with a hierarchical workflow management system in [20] to monitor the scientific workflows for failure recovery. Another direct implementation of "RFDMon" is presented in [21] where virtual machine monitoring tools and a model based predictive controller were combined with the proposed monitoring framework to manage the multi-dimensional QoS data for a multi-tier web service. An extended version of this paper is available as technical report [13].

IX. Conclusion and Future Work

In this paper, detailed design of "RFDMon" is presented. "RFDMon" is a real-time and fault-tolerant distributed system monitoring approach based upon data centric publish-subscribe paradigm. Basic concepts of OpenSplice DDS, ARINC-653, and Ruby on Rails are also described in the paper. Additionally, it is shown that "RFDMon" can efficiently and accurately monitor the system resource consumption, system health, application performance variables, and scientific application data structures with minimum latency. Furthermore, fault tolerance and self configurable properties of "RFDMon" is also demonstrated through experiments.

An administrator can easily find the location and possible causes of the faults in system by visualizing the measurements. To make this fault identification and diagnosis procedure autonomic, we are developing a fault diagnosis module that can detect or predict the faults in the infrastructure by observing and co-relating the various sensor measurements. Additionally, we are developing a self-configuring hierarchical control framework (extension of our work in [22]) to manage multi-dimensional QoS parameters in multi-tier web service environment.

X. Acknowledgement

R. Mehrotra and S. Abdelwahed are supported for this work from NSF I/UCRC CGI Program grant number IIP-1034897 and The Engineer Research and Development Center (ERDC) at Vicksburg, MS. A. Dubey is supported in part by Fermi National Accelerator Laboratory, operated by Fermi Research Alliance, LLC under contract No. DE-AC02-07CH11359 with the United States Department of Energy (DoE), and by DoE SciDAC program under the contract No. DOE DE-FC02-06 ER41442. We are grateful to the help and guidance provided by T. Bapty, S. Neema, J. Kowalkowski, J. Simone, D. Holmgren, A. Singh, N. Seenu and R. Herber.

References

[1] Catalog of omg data distribution service (dds) specifications. http://www.omg.org/technology/documents/dds_spec_catalog.htm [Nov 2010].

[2] Abhishek Dubey, Gabor Karsai, and Nagabhushan Mahadevan. A component model for hard real-time systems: Ccm with arinc-653. *Software: Practice and Experience*, 41(12):1517–1550, 2011.

[3] Opensplice dds community edition. http://www.prismtech.com/opensplice/opensplice-dds-community.

[4] Fermilab lattice gauge theory computational facility. http://www.usqcd.org/fnal/ [Nov 2010].

[5] Arinc specification 653-2 : Avionics application software standard interface part 1required services. Technical report, Annapolis, MD, December 2005.

[6] Ruby on rails. http://rubyonrails.org/ [Sep 2011].

[7] Object management group. http://www.omg.org/ [Nov 2010].

[8] Nuno Diniz and Jose Rufino. Arinc 653 in space. In *Proceedings of the DASIA 2005 "Data Systems in Aerospace" Conference*, May/June 2005.

[9] A. Goldberg and G. Horvath. Software fault protection with arinc 653. In *Aerospace Conference, 2007 IEEE*, pages 1 –11, March 2007.

[10] Model-based software health management. https://wiki.isis.vanderbilt.edu/mbshm/index.php/Main_Page [Nov 2011].

[11] Ruby. http://www.ruby-lang.org/en/ [Sep 2011].

[12] J. Deacon. Model-view-controller (mvc) architecture. *JOHN DEACON Computer Systems Development, Consulting & Training*, 2005.

[13] Abhishek Dubey Rajat Mehrotra and Sherif Abdelwahed. Rfdmon: A real-time and fault-tolerant distributed system monitoring approach. Technical Report ISIS-11-107, Institute for Software Integrated Systems, Vanderbilt University, Oct 2011.

[14] Ganglia. http://ganglia.sourceforge.net/ [Sep 2011].

[15] Nagios. http://www.nagios.org/ [Sep 2011].

[16] Zenoss. http://www.zenoss.com/ [Sep 2011].

[17] Nimsoft unified manager. http://www.nimsoft.com/solutions/nimsoft-unified-manager [Nov 2011].

[18] Curl. http://curl.haxx.se/ [Nov 2011].

[19] Intelligent platform management interface (ipmi). http://www.intel.com/design/servers/ipmi/ [Sep 2011].

[20] Pan Pan, Abhishek Dubey, and Luciano Piccoli. Dynamic workflow management and monitoring using dds. In *7th International Workshop on Engineering of Autonomic & Autonomous Systems (EASe)*, 2010.

[21] Rajat Mehrotra, Abhishek Dubey, Sherif Abdelwahed, and Weston Monceaux. Large scale monitoring and online analysis in a distributed virtualized environment. *Engineering of Autonomic and Autonomous Systems, IEEE International Workshop on*, 0:1–9, 2011.

[22] Rajat Mehrotra, Abhishek Dubey, Sherif Abdelwahed, and Asser Tantawi. *A Power-aware Modeling and Autonomic Management Framework for Distributed Computing Systems*. CRC Press, 2011.

Online Spectrum-based Fault Localization for Health Monitoring and Fault Recovery of Self-Adaptive Systems

Éric Piel*, Alberto Gonzalez-Sanchez*, Hans-Gerhard Gross* Arjan J.C. van Gemund*, and Rui Abreu†
*Department of Software Technology
Delft University of Technology
Mekelweg 4, 2628CD Delft, The Netherlands
{e.a.b.piel, a.gonzalezsanchez, h.g.gross, a.j.c.vangemund}@tudelft.nl
†Department of Informatics Engieering
Faculty of Engineering of University of Porto
Rua Dr. Roberto Frias s/n, 4200-465 Porto, Portugal
rui@computer.org

***Abstract*—Software systems used in the industry are often large and complex. Even with an extensive validation phase, it is impossible to ensure that a software system is fault-free and will remain so all along its evolution. When a failure happens in operation, the time to solve the fault should be minimized. The major challenge in this realm is the localization of a fault in one of the constituent components of the overall system. We strive at simplifying the localization of the fault that led to a failure by adapting existing techniques to the online context in such a way that allows the system to be aware of its own internal faults and react to it. This article first proposes to apply the Spectrum-based Fault Localization (SFL) method for online fault localization and health monitoring. Several implementation approaches are presented with a performance that depends on the architecture and the framework used. Evaluation is done through simulation of online failure scenarios, and through implementation in a demonstration surveillance system. The results of the studies performed confirm that applying SFL online, using monitoring, can successfully provide health information and locate problematic components, so that a software failure can be addressed adequately and timely.**

***Keywords*-Fault localization; diagnosis; self-awareness; autonomous system; monitoring; component-based system.**

I. Introduction

It is generally accepted that all but the most trivial software systems will inevitably contain residual defects. Large and complex software systems, such as systems of systems, will face these problems. Nowadays, the high reliability, availability, and flexibility imposed on many systems require support for online reconfiguration and join/leave of external components (a coupled and cohesive part of a system). This further increases the chances of unexpected behavior during execution, as they are hard to take into account in the validation phase. As such problems cannot be avoided, the system should be prepared to handle them as quickly as possible. Typically, after a failure (a deviation from the expected behavior) has been detected the following steps are taken: diagnosis, bug fix design, re-validation, and update. To reduce the time of this process, we focus here on automating the diagnosis step, which very few previous works in adaptive systems have tried to automate. This step focuses on finding the location of the fault, i.e., the cause of one or more failures in the system.

So far automated diagnosis techniques, also called fault localization, have been applied solely offline, during the testing phase. In this article, we detail approaches to apply fault localization in an online context, i.e., when the system is in operation. One of the obstacles is that typical *active testing* used offline cannot be applied online, because of interference with the normal operations. So continuous validation must come from observations provided by monitors, also referred to as *passive testing*. While there exist other approaches to fault localization [1], [2], [3], SFL is one of the most light-weight fault localization techniques available to be used for the provision of health information and for identifying problematic components in software systems.

In this paper, we make the following three contributions. (1) We demonstrate how SFL can be applied to online fault localization by introducing three main adaptations to the original technique. (2) We describe two different approaches for the implementation of online fault localization according to the characteristics of the software system. (3) We assess the performance of our proposed techniques in simulations as well as in a real industrial case study.

The original SFL technique is described in Section II. Section III presents the modeling of the problem. Our proposal of online fault localization is presented in Section IV. Section V summarizes the main approaches we have used to implement fault localization on actual software systems. Section VI evaluates the technique on a case study. Finally, Section VII discusses related work, and Section VIII concludes the article.

II. Spectrum-Based Fault Localization

The objective of fault localization is to pinpoint the precise locations of faults in a system. Before delving into

$\mathcal{C}$	Program: Character Counter	t_1	t_2	t_3	t_4	t_5	t_6	SC
	`function count(char *s) {`							
	`int let, dig, other, i;`	0	0	0	0	0	0	0
c_0	`let = dig = other = i = 0;`	1	1	1	1	1	1	0.87
c_1	`while (c = s[i++]) {`	1	1	1	1	1	1	0.87
c_2	`if('A'<=c && 'Z'>=c)`	1	1	1	1	0	1	0.93
c_3	**`let += 2;`**	1	1	1	1	0	0	1.0
c_4	`else if('a'<=c && 'z'>=c)`	1	1	1	1	0	1	0.93
c_5	`let += 1;`	1	1	0	0	0	0	0.71
c_6	`else if('0'<=c && '9'>=c)`	1	1	1	1	0	1	0.93
c_7	`dig += 1;`	0	1	0	1	0	0	0.71
c_8	`else if(isprint(c))`	1	0	1	0	0	1	0.47
c_9	`other += 1;}`	1	0	1	0	0	1	0.47
c_{10}	`printf("%d %d %d\n", let, dig, other);}`	1	1	1	1	1	1	0.87
	Test case outcomes	1	1	1	1	0	0	

Table I
EXAMPLE PROGRAM, SPECTRUM, AND OUTPUT IN SFL.

the usage of the SFL approach for online fault localization, and the provision of health information, let us introduce SFL in its offline version.

The following data are usually used as inputs in SFL approaches:

- A finite set $\mathcal{C} = \{c_1, c_2, \ldots, c_j, \ldots, c_M\}$ of M *components* (e.g., source code statements, functions, classes) which are potentially faulty. We will denote the number of faulty components in the system as M_f.
- A finite set $\mathcal{T} = \{t_1, t_2, \ldots, t_i, \ldots, t_N\}$ of N given tests with binary outcomes $O = (o_1, o_2, \ldots, o_i, \ldots, o_N)$, where $o_i = 1$ if test t_i failed, and $o_i = 0$ otherwise.
- An $N \times M$ *coverage matrix*, $A = [a_{ij}]$, where $a_{ij} = 1$ if test t_i involves (covers) component c_j, and 0 otherwise. Each row a_i of the matrix is called a *spectrum*.

Table I shows an example of SFL applied on a small program with a component granularity at the statement level. This program aims at counting different types of characters. The component c_3 contains a fault, mishandling uppercase characters. 6 tests are executed against this implementation. The columns t_1 to t_6 present the coverage spectrum and the test outcomes when executing each of the tests. The last column shows the similarity coefficients, a value computed by the SFL, which we will describe later.

The output of fault localization is a *diagnosis*, which is a ranking of the components ordered according to their assumed likelihood to contain a fault.

In program debugging, the granularity of a *component* is often very small, typically at the statement level, since SFL benefits from variations in program control flow (i.e., different branches of a `if` are taken). However, in an online context, a larger grain size for components is more appropriate. This still permits to monitor a system and to take the appropriate actions in case of degradation, while it reduces the performance overhead, and represents a more realistic component granularity for large systems. In the later study, we selected a granularity at the level of the source code functions.

A. Statistical Spectrum-Based Fault Localization

Statistical SFL is a well-known approach originating in software engineering [4], [5], [6]. Fault likelihood l_j (and thus assumed health) is quantified in terms of *similarity coefficients*. Intuitively, the goal is to identify the component whose line of test coverage is most similar to the test outcomes. Similarity coefficients measure the statistical similarity between component c_j's test coverage $(a_{1j}, \ldots, a_{Nj})$ and the observed test outcomes, $(o_1, \ldots, o_N)$. It is computed by four values $n_{pq}(j)$ counting the number of times a_{ij} and o_i form the combinations $(0,0), (0,1), (1,0), (1,1)$, respectively, i.e.,

$$n_{pq}(j) = |\{i : a_{ij} = p \wedge o_i = q\}| \;\; p, q \in \{0, 1\} \qquad (1)$$

For instance, $n_{10}(j)$ and $n_{11}(j)$ are the number of tests in which component c_j is executed, and which passed or failed, respectively. For each component, the four counters sum up to the number of tests N. There are several different known similarity coefficients which are efficient. For example, Tarantula [5], and Ochiai [4] are both very common similarity coefficients. We use the latter one, given by

$$\text{Ochiai:} \quad SC = \frac{n_{11}(j)}{\sqrt{(n_{11}(j)+n_{01}(j)) \cdot (n_{11}(j)+n_{10}(j))}} \qquad (2)$$

Ordering the components by their similarity coefficients results in the ranking of the diagnosis algorithm.

In Table I, the similarity coefficient for each component is indicated. As c_3 was the part most used when a test failed and less used when a test passed, its similarity coefficient is the highest. The SFL will therefore rank c_3 as the most likely location of the fault, which is correct.

A by-product of statistical SFL is the *component health*. The health of a given component can be simply approximated by $h = 1 - SC$, where SC is the similarity coefficient. This permits the system, or system of systems, to also be self-adapting to the failures. Components which have access to redundant information can adapt the weight of each input depending on the health of the components that provide it. For example, in the maritime safety and security context, when a radar starts behaving incorrectly, the situation awareness component can reduce automatically the importance of the data from this radar in its computations.

Despite their lower diagnostic accuracy [7], similarity coefficients have a ultra-low computational complexity (compared with probabilistic diagnosis approaches, such as Bayesian reasoning [5]), which is ideal for online diagnosis. Another advantage is the fact that statistical SFL is incremental. Only the counters n_{pq} must be kept per component, so there is no need to compile a (possibly huge) test coverage matrix. Finally, unlike other approaches, statistical SFL is robust with respect to uncertainties in the test outcomes.

While all techniques tolerate false negatives (i.e., a test involving a faulty component and not returning a failure), statistical approaches are more robust with respect to false positives (i.e., a test reports a failure although the system actually behaved correctly), which is essential in online monitoring as the oracles are often less sophisticated than in offline testing.

B. Diagnosis Effort

In order to compare different diagnosis approaches, there is a need to measure how well a diagnosis performed. This measure, the diagnostic performance, should represent how well the diagnosis algorithm can pinpoint the true root cause of an observed problem. In software fault localization, this performance is often expressed in terms of a metric C_d that measures the theoretical *effort* still needed for a diagnostician to find all faulty components after reading the generated diagnosis [7]. C_d is expressed as the position of the last faulty component in the ranking given by the fault localization. C_d measures *wasted effort*, independent of the number of faulty components M_f in the system, to enable an unbiased evaluation of the effect of M_f on C_d. Thus, regardless of M_f, $C_d = 0$ represents an ideal diagnosis technique (all M_f faulty components are ranked at the top, and no effort is wasted for a human to check healthy components), while $C_d = M - M_f$ represents the worst diagnosis technique (checking all $M - M_f$ healthy components before the M_f faulty ones), with M the total number of components. For example, consider a diagnosis algorithm that returned the ranking $\langle c_{12}, c_5, c_6, \ldots \rangle$, while c_6 contains the actual fault. This diagnosis leads the developer to inspecting c_{12} and c_5 first. As both components are healthy, C_d is increased by 2, and the next component to be inspected is c_6. As it is faulty, no more effort is wasted and $C_d = 2$. To ease comparison between systems, a *relative wasted effort* is often used: $\frac{C_d}{M-M_f}$. A perfect diagnosis gives therefore a relative effort of 0, while the worse possible one gives an effort of 1, and an algorithm picking randomly a component gives on average a relative effort of 0.5.

III. Simulation of a Faulty System

For initial illustration and evaluation of online SFL we use synthetic system simulations next to an actual case study. The main advantage of the simulations is that they can be executed quickly (e.g., for our case study system we can simulate one hour of operation in just a few seconds). They allow to vary many properties of a base system, in order to generalize the findings according to many different (synthetic) system configurations, and they also avoid implementation details which could cause noise in the observations (e.g., test outcomes with false positives).

A. System Model

The simulations use system models with different topologies all based on the surveillance system used as case study, which is presented in Section VI. The simulation of a system generates outputs similar to the ones given by the actual SFL algorithm, i.e., a ranking of the components according to their assumed health over the whole period of execution of the simulation. The simulator and example models are available for download [8].

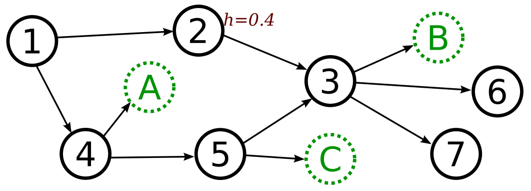


Figure 1. Example topological layer with 7 functional components and 3 monitors.

Fig. 1 shows an example of a system model, with 7 functional components and 3 monitors (A, B, and C). As we will see in Section IV, monitors are placed in order to replace test cases in an online context. Component 2 is set to be faulty, with a fault happening 60% of the time it is used. The model represents a typical data-flow system where component 1 receives the inputs and passes them on to the other components. More information about the simulation setup and a description of the type of model that is used in the simulator can be found in [9].

B. Simulated System Generation

One of the most difficult parts of simulation is to obtain models of systems which are representative of the reality. If a model is generated fully randomly with respect to every possible parameter, there is little chance that it corresponds to a potential real system. That is because only some topologies, order of execution, etc. are reasonable for a software system. Therefore, as basis for creating many simulations, we used the topology of a known surveillance system. It comprises 63 components for the functionality. For each component, a configuration was generated with that component being the faulty one. For each fault location, 10 different system configurations were generated by randomly placing 15 monitors, and producing a set of 20 execution paths (with random frequencies between 0.2 Hz and 50 Hz). Therefore, each technique can be evaluated on 630 system configurations. Results are presented in the next section.

IV. Online Fault Localization

Applying SFL online brings up three issues: (1) test cases would disrupt the normal operation of the system (to be discussed in Section IV-A, (2) the range of a coverage spectrum (to be discussed in Section IV-B), (3) the adequacy of the diagnosis with the current system behavior (to be discussed in Section IV-C). In an offline context, tests are run separately, so the start and end of a test and the coverage spectrum are clear, as well as associated inputs and outputs. However, in the case of continuous diagnosis these boundaries disappear, or, at least, become blurred. In this section, we present solutions for adapting SFL to an online context.

A. *Obtaining Test Outcomes Online*

In order to bring fault localization online, the usage of test cases must be reevaluated. Test cases are *active*, as they provides their own inputs to the system. If done during operation, such input can interfere with the usual behavior of the system, and can cause a large performance overhead. Therefore, in the online context, *monitoring* is more fitting. Monitoring is well-understood, easy to apply, and event-based, due to its passive nature, e.g., triggered by the arrival of new data, or a timer interrupt. A monitor is a specific component in the system that observes and assesses the correctness of the functionality without interfering through test inputs.

A monitor observes data or behavior at specific locations and decides based on built-in oracle logic whether an observation is expected (*pass*) or unexpected (*fail*), for example through checking the range of a variable, consistency between different data, or through comparison with a state model. The monitor outcomes replace the test outcome. Because SFL requires to know when the system is deemed behaving both correctly and incorrectly, it is of prime importance when writing a monitor that whenever a *fail* could be sent, it sends a *pass* if no failure is detected.

B. *Spectrum Sampling*

In many cases, interactions in a live system are not clearly separable by time or space boundaries (such as a complete test transaction in testing). Input stimuli are continuously arriving and the system responds accordingly changing its internal state and/or producing some output. For example, in our case study (cf Section VI), input messages arrive at any time, and sometimes simultaneously in separate threads. Previous inputs influence the behavior of a component either explicitly such as in a database, or implicitly by affecting its internal state. When applying SFL offline, the coverage spectrum is recorded since the system was started for a test case. In an online context, after a short period of operation, the coverage matrix will contain only 1's: "everything covered". Although this approach would guarantee a theoretical strong causal relationship between fault execution and failure observation (i.e., if a failure is observed, the spectrum will contain the fault information), a solid 1's spectrum does not provide any diagnostic information for the SFL, because it infers the diagnosis from differences of the various spectra in the coverage matrix A and the outcome O. The curve named *time inf* of Fig. 2 shows the result of never resetting the spectrum. The average diagnostic cost is approximately 0.5 all the time. Guessing the fault locations randomly would yield a similar performance.

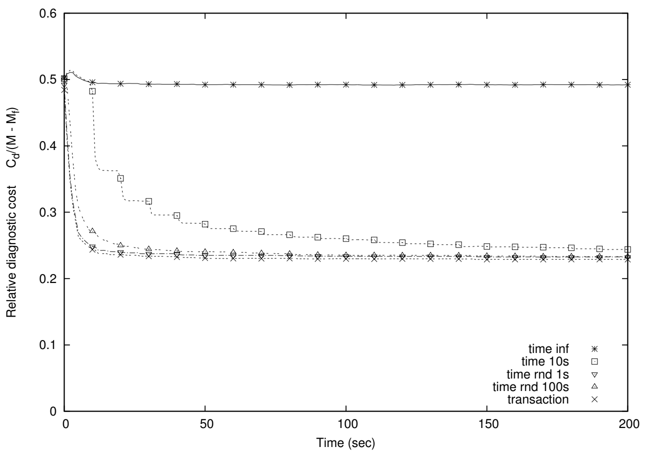


Figure 2. Average diagnostic cost along the time of observation for various observation policies, with simulated systems having one fault.

The coverage of components represented as binary values in the spectrum must be reset regularly, in order to provide a meaningful diagnosis. We propose two different solutions, which are adapted to different development contexts, that is, a transactional approach, and a time frame approach.

1) Transactional: A monitor validates the correctness of a specific component transaction in the system, corresponding to particular interactive functionality. The provision of an outcome through the monitor correlates to the end of this transaction. The *transactional* policy assigns a separate spectrum to every monitor. Every monitor is also associated to a scope, which represents which components might be involved in the monitored interaction[1]. Each time a component is involved, the current spectrum of every monitor whose scope contains that component is updated. When a monitor generates an outcome, its associated spectrum is used as a row for the matrix A and is then completely reset to zero.

The list of the components in the scope associated to each monitor is provided before the start of the system (and is updated after each modification). It is either manually created by the user (the developer of the monitor, most likely), or it could be determined by code or configuration analysis. Fig. 2 shows with the curve *transaction* that this solution is the most effective one, with a low average diagnostic cost throughout the execution of the systems. The curve tends towards an asymptote close from 0.2. This asymptote corresponds to the average diagnostic cost that can be achieved by the SFL algorithm with all possible spectra for the specific set of systems in the simulation.

However, if a fault modifies how components interact (i.e., the control flow is modified), the difference between the expected behavior and the implementation could lead to an inaccurate scope. In such a case, this policy would

[1]Each execution of the interaction can be considered a *transaction*, hence the name of the policy.

cause a faulty component to be omitted from every spectrum associated with a *fail* outcome. The quality of the diagnosis would be adversely affected. In addition, pre-analysis of the system for every monitor can be time consuming, and needs to be done every time the system is modified. It might be difficult to perform if external components (from different companies) are used. In order to avoid this analysis we investigate a technique requiring less information about the system, i.e., the *time frame* technique.

2) Time Frame: The *time frame* policy uses expiration of time as transaction boundary to establish causality between components covered and monitor outcome. Over a given time period, the component activity is recorded into a global "current spectrum". When the time expires, the bits of the involved components are reset and the recording of a new current spectrum is started. Every monitor outcome during this period, is associated with the current spectrum.

Time frame-based sampling avoids spectra with too many 1's if the time window is properly adjusted to the working speed of the system. To avoid using a period which could hide a specific fault our approach uses a random frame length. After expiration of a time frame, the length of the next frame is determined randomly within reasonable bounds. An exponential distribution is used, in order to have a broad set of period sizes. An average period must be selected according to the system under observation, but it can be relatively roughly estimated to the average processing time of a typical transaction. In Fig. 2 it can be seen how a fixed time period leads a limited accuracy of the fault localization, with the curve *time 10s*. The curves *time rnd 1s* and *time rnd 100s*, corresponding respectively to a randomized time frame with an average of 1 s and 100 s, both provide on average a low diagnostic cost.

We recommend that the observation policy should be selected according to the system context: if it is possible to gather precise information on which interaction is observed by a monitor, then the transactional policy should be applied. Otherwise the randomized time frame policy should be implemented, with just enough validation to ensure the average period is adapted to the system.

C. Spectrum Matrix Size

When using SFL offline, the size of the spectrum matrix and the test outcome vector are finite and, in practice, relatively small, which is not the case online. For example, in our case study, approximately 100,000 monitor outcomes are generated for a single hour of observation. This could eventually lead to excessive storage requirements and processing overheads. This potential size problem is addressed through application of statistical SFL, on which our approach relies. It is incremental, so that accumulating counters can be used.

However, another issue is the timeliness of a spectrum, for example "is a week-old observation relevant for the current state of the system?" A fault may appear long time after the system was started (e.g., memory leakage, unexpected combination of inputs that affect the internal state of the system, an unnoticed third-party component update). Old spectra might mislead the fault localization. The detection of a new failure should always lead to the same diagnosis, independent of how long the system has been running.

Note however that the problem is not symmetric, when conversely, a fault is fixed, or the failures are not observed anymore. If the fault is fixed, it is easy to reset the matrix at the same time to avoid this "aging effect". If the failures stop appearing without the fault having been fixed, it is better to still report the component as faulty for some sufficiently long time to acknowledge the problem and deal with it.

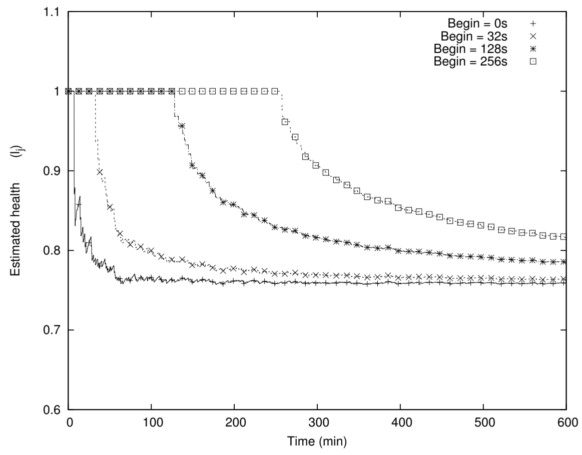


Figure 3. Estimated health with an infinite window.

Fig. 3 shows the health estimated by the SFL algorithm for a faulty component yielding a failure at different times, when all spectra are kept. The later the failure surfaces, the slower is the convergence of health. From the point of view of the system maintainer, when a given failure happens, the algorithm output should be identical independently from the time system has been running previously.

To overcome this problem, we defined the *sliding window* policy. Spectra that are older than a given age are discarded. In practice, as the SFL counters are accumulated, we approximate the window by decomposing it into a fixed number of small periods. An array of counters allows to keep track of the SFL counters for each period. When the current period is over, the oldest set of counters is discarded and replaced by a new set for the next coming period. The global counters are replaced by an addition of the counters for each available period. In our implementation we used 32 sub-periods, which appeared to be of sufficient precision.

The ideal window size (leading to stable health values) depends on the frequency of the monitors generating observations and the frequency of failures being detected. In our experiments, we observed that short sliding windows yield a relatively high diagnostic cost and unstable output

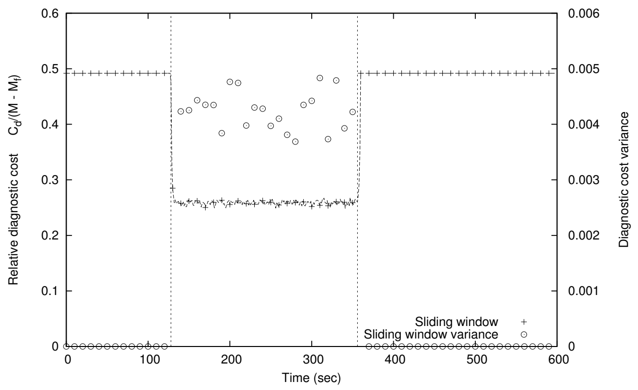


Figure 4. Average diagnostic cost on simulated systems with a sliding window policy of length of 4 s (component fails in the period 128 s – 356 s, dotted lines).

over time, because they are too small to contain enough test outcomes for adequate diagnosis. When the size of the window is extended, it reaches a point where the diagnostic performance does not improve anymore. Increasing further the length solely leads to a bigger latency to react to the failure disappearance. The size of the window after which the diagnosis presents no more noise depends on the frequency at which the failures are detected. We observed that the minimum efficient window size depends on the amount of *fail* outcomes that are captured. The amount of *pass* outcomes is usually far superior, so it is not a bottleneck. We observed that if a window is long enough to contain at least approximately 10 *fail* outcomes, it is sufficient to keep a good quality diagnosis.

Therefore, we recommend selecting a size of the window which is sufficiently long to receive many monitor outcomes. The main restriction on the maximum length is to ensure a fairly fast reaction in terms of health. The window size can be set as the minimum duration for which a single failure occurrence should be seen when looking at the diagnosis.

In order to observe the effect of applying the sliding window policy, we simulate a system where a new failure is seen, lasts for 228 s, and disappears. Fig. 4 shows the average diagnostic cost when a window size of 4 s is applied. Approximately 4 seconds after the first failure appears the diagnostic cost reaches its minimum. Similarly, the diagnostic reacts within seconds to the disappearance of the failures. As the failure frequency is high enough that a window contains several *fail* observation outcomes, the diagnostic variance is relatively low. Increasing the window size would stabilize even further the diagnostic over the period that the failure happens.

D. Self-Adaptation to Faults

By localizing properly and precisely the faults, a system has two main ways to react in order to improve its behavior. Firstly, it can attempt to fix the failure origin by applying an automated fix such as described in [10]. Such automated fixes rely usually on a set of generic fixes. Each of the generic fix can be apply sequentially after each other, on the each of the most suspicious fault location provided by the online SFL. The search for a fix ends when the online SFL detects that the health of the system goes back to an acceptable level (or when all the fixes have been tried unsuccessfully).

A second way to adapt to a fault, orthogonal to the first one, is to take into account the estimated health of the components into the functional behavior. As seen previously, SFL computes for each component a *similarity coefficient*, which can be converted into an estimated health value approximating how likely the component provides a correct output. The *confidence* of a data is the product of the health of each component which was involved in generating it. In dependable systems, it is usual to obtain data from multiple independent sources and/or process the data via redundant paths. Instead of relying equally on all the redundant data, components which receive data from multiple sources can weight the data according to their confidence value. Therefor, the system adapts automatically to faults by avoiding to rely on the incorrect data.

V. Implementation of Online Fault Localization

There are many ways to implement the proposed techniques. We outline here two different implementation approaches that we have carried out successfully. The first approach is centralized, while the second one is metadata-based.

A. Centralized Approach

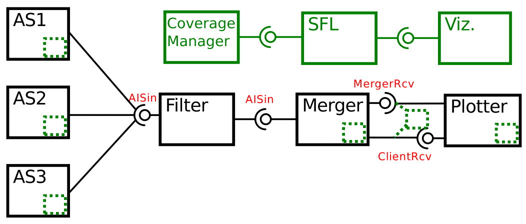


Figure 5. Architecture of the case study system, which is based on the centralized approach.

A first implementation approach, which we have used in our Atlas framework [11], relies on a centralized spectrum recording. Its architecture can be broken down into five parts. An example system using such an approach is displayed in Fig. 5. For each architectural part, we will refer to this example. The *coverage manager* component takes care of keeping the coverage spectrum of the system. In the example, this component is represented by the box of the same name. The spectrum is reset periodically according to the randomized time frame policy as described previously. By request from the coverage instrumentation part (discussed later), it sets a position in the spectrum to indicate a specific component has been covered. When a monitor sends a new observation, the coverage manager receives this observation, attaches the current spectrum, and forwards it to the SFL component.

The *SFL* component (which is represented by the *SFL* box in the example) receives every monitor observation and adds it to the matrix according to the sliding window policy. In practice, a whole matrix is not needed, only a set of accumulators, which permits a fast processing. Running at a slower frequency, the similarity coefficient and ranking of the faulty components is computed. This might require a noticeable amount of processing power, but it can be done independently from the rest of the system, even offloaded to separate hardware.

Every functional component of the system is instrumented to report whenever one of its methods is called. In the example, every component part of the core functionality is instrumented. We use Aspect Orientation [12] and Java self-reflection to apply the same code to all the components. This allows to dynamically instrument any component, even when provided by a third party or added a posteriori. However, it brings a high overhead to each method call. A static approach, such as found in many code profilers, would likely be more efficient.

Finally, the behavior of the system is validated by a set of monitors, positioned at various places between or around the normal components. Monitors are represented as dash boxes in the example. Every monitor observation, both *fails* and *passes*, is transmitted to the *coverage manager*. A monitor can be replacing what would traditionally be a warning or error check, or can be more complex piece of code which validates the outputs of a component compared to the previously received input (based for instance on a state machine). Watchdogs, which detect the loss of service provided by a component can also be implemented as monitors but care should be taken to report in case of failure not the actual spectrum, but the spectrum that would be expected (so that SFL can point towards the non-responding part of the system).

Last but not least, the *visualization* component receives the measurements from the *SFL* component and displays to the user a graph of the health of the components (approximated by their similarity coefficient) over time.

B. Metadata-Based Approach

The centralized approach is easy to implement and efficient on systems where all components can access the *coverage manager* with a low latency and where communications have a low overhead. In systems where components are running on physically separate nodes such as systems-of-systems, or systems which are message-based, it might be more efficient to use a different approach, based on metadata. All data transmitted between components is associated to metadata that contains a coverage spectrum indicating all the components used to generate this data. Every time an output is generated, its metadata must be set, based on the metadata of the inputs. Note that computing the spectrum might be difficult in some cases where many inputs are used. There is still a central component for the coverage, but it is only accessed to request a position in the spectrum when a component initializes. Monitors work similarly to the previous implementation approach except that the spectrum associated to an observation comes from the metadata of the output which is validated. This observation can then be sent directly to the SFL component.

To handle dynamic system architectures, where components can be added and removed online, the coverage spectrum needs to have positions updated when there is a change. We treat this requirement by having the *coverage manager* assign positions to new components. When a component is removed the positions which were assigned to it can be reused, once a certain delay corresponding to the time window length has passed.

VI. Case Study

All techniques for realizing online fault localization with SFL have been introduced. Synthetic system simulations were used to compare different techniques to each other on a large set of systems. In the following, we evaluate our contributions on a real system. The main goal is to validate the techniques on practical ground, and verify that the simulated systems behave similarly to the actual ones.

The surveillance system that we use as case receives information broadcasts from ships, called *AIS messages* [13], and it processes them in order to form a situational picture of a maritime area. The system is made of Atlas components in Java. In total it is comprised of 63 methods (the granularity of the SFL) for the core functionality with an average of 10 lines of Java code each.

The monitoring infrastructure comprises four monitors, each of them guarding different functional and non-functional aspects of the system. Coverage of components is recorded through an ad-hoc Java aspect, as described in Section V.

A. Injected Faults

We simulate two types of faults, loss of data between components (for example due to reset of the component, or unstable connection), and software faults caused by the functionality. Data loss faults are simulated through intermittent connection drops between two components. Software faults are introduced through mutations in the original code (a set of 100 mutants which was created with μJava [14] and manually verified to affect the behavior of the system). For each of the mutation faults, the system was executed for one hour with the recorded input, producing approximately 100,000 monitor outcomes in total. A posteriori, it is then possible to determine the diagnostic cost at each moment in time. 12 mutations lead to early system crash (within a minute) and are sorted out (in practice, such a bug would be directly noticed and investigated off-line). 55 mutations have faults not detected by the monitors, leaving 33 configurations with detectable faults.

B. Results

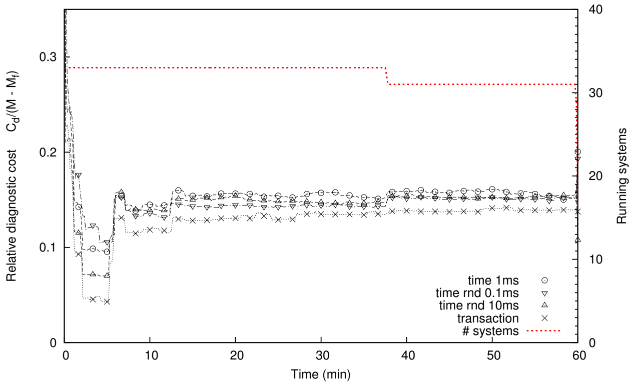


Figure 6. Average diagnostic cost (33 configurations) over the time for three different observation policies.

The average C_d for *transactional* and *randomized time frame* observation strategies is presented in Fig. 6. The *# systems* indicate the number of systems still running at a given time. It decreases whenever a system crashes or stop responding. The SFL algorithm uses a sliding window of 5 minutes, in order to ensure a good quality of the diagnosis while keeping a relatively fast reaction to any fault correction.

The diagnostic cost C_d, which starts at 0.5, decreases until it reaches some relatively constant value after around a minute. This is similar to the results seen in the simulations (Fig. 2). After 5 minutes of execution (i.e., the length of the sliding window), all C_d graphs increase. This is because some faults lead to failures only at initialization, i.e., they are located in components only used at that time. When these first spectra are removed from the matrix (through the sliding window) the SFL loses information about their location, and assumes a better health, leading typically to a $C_d = 0.5$. Hence, the average C_d increases.

As in the simulation, the *transactional* observation performs best, with an average $C_d = 0.14$. The *time frame* observation yields its best results with 1 ms ($C_d = 0.16$). A shorter or longer period impairs the results, leading to C_d around 0.3 (not shown in the figure to improve readability). This suggests that observation periods of 1 ms are optimal for this system. The *randomized time frame* observation performed equally well as the best fixed time period, for all periods tried between 0.1 ms and 100 ms.

In our case, transactional observation provides the best results. Nevertheless, this requires that for each monitor the information about which components are observed is known and correct. Otherwise, a randomized time frame allows diagnosis with comparable quality, with only a rough estimation of the processing time needed.

This case study demonstrates the feasibility of online fault localization using the SFL technique in a system inspired by industry. With a diagnostic cost ranging on average below 0.2 just after a minute, it also shows that fault localization is able to point into the right direction for identifying problematic components in software systems. Of course, this works only if residual defects can be detected by the monitors. The fact that the results are relatively similar to the results obtained by simulation suggests that the model employed for the simulation is representative of this real case.

The case study shows also that a relatively small number of monitors (compared to the number of components) is sufficient to locate faults. Although no complete study has yet be done on the needed number of monitors for a given system, our first observations are that 1) this can vary considerably depending on the topology of the system and the false negative rate of the faults, and 2) for a system of N components, a large number of fault locations can be correctly found when the number of monitors is above $log(N)$.

VII. Related Work

The role of fault diagnosis for realising more adaptive, intelligent, and self-aware systems has been recognized for at least a decade (e.g., [15], [16]). Some researchers have looked at online defect detection [17], [18], but did not address the specific issues of finding the root causes of defects, i.e., the diagnosis.

Seltzer and Small [19] and Chen [20] have proposed system infrastructures for enabling self-monitoring and -adaptation. However, their approaches focus on system performance, ignoring all the other software quality issues, that our approach is able to treat. The biggest drawbacks of these approaches is that they rely on ad-hoc localization algorithms, which are based on long observations performed in test systems rather than in the operational systems, and that they often require manual adjustments. The usage of automatic diagnosis in our approach avoids these drawbacks. Our approach can be applied in a generic way, and relies only on the latest observations.

In [2], an invariant-based approach is presented and applied online. However, they use specialized active unit-testing instead of monitoring, and the system state is recorded every time a test is executed, which leads to a very high overhead (execution time multiplied by ∼100). An additional issue are interferences that active testing can cause in a running system.

In [21], an approach for self-repair, coined Rainbow, which allocates the diagnosis process to the individual repair handlers is presented. Rainbow defines a set of repair strategies that are triggered when certain architectural invariants are violated in a running system. Thus, each strategy is responsible to realize (i.e., diagnose) whether or not its set of actions will fix the observed problem.

In [22], an approach for architecture-based run-time fault diagnosis is presented. Conversely to our approach, the one presented in [22] applies a lightweight model-based approach to fault diagnosis based on the architecture description of the system at the granularity level defined by the architecture (typically, coarse granularity). Similar to the Rainbow approach, pass/fail information is obtained by checking whether architectural invariants are violated in a running system or not.

VIII. Conclusions and Future Work

While fault localization is a fundamental step towards adaptive and self-managing systems, in order to identify the part of the system which should be corrected, little work so far has focused on adopting existing diagnosis approaches into this domain. In this article, we present an approach for realizing online spectrum-based fault localization to be used in self-adaptive systems. We introduce techniques to obtain a significant spectrum for the SFL algorithm in order to yield good diagnoses. The usage of a sliding window, provides a diagnostic outcome which is always relevant to the current state of the system. Furthermore, we presented two different implementation approaches which fit either to centralized architectures or distributed architectures.

Our contributions are validated first by simulation of a large set of randomly generated systems, and through a case study with a system inspired by industry. The diagnostic results on a set of real, mutated systems corroborate the results of the simulation and confirm that, with our contributions, SFL and monitoring can be applied successfully to online fault localization.

Additional challenges could be investigated in future work in order to improve the quality of online fault localization in real systems. One of the main topics we will investigate is the usage of runtime testing to complement the monitors. When a fault is detected but its location cannot be precisely pinpointed, a small set of runtime tests could be executed on the system in order to obtain more information.

Acknowledgement

This work is part of the ESI Poseidon project, partially supported by the Dutch Ministry of Economic Affairs under the BSIK03021 program.

References

[1] H. Cleve and A. Zeller, “Locating causes of program failures,” in *ICSE ’05: Proceedings of the 27th international conference on Software engineering*. St. Louis, MO, USA: ACM Press, May 2005, pp. 342–351.

[2] D. Slane, “Fault localization in in vivo software testing,” Master’s thesis, Bard College, Massachusetts, USA, 2009.

[3] B. C. Williams, M. D. Ingham, S. H. Chung, and P. H. Elliott, “Model-based programming of intelligent embedded systems and robotic space explorers,” in *IEEE Special Issue on Modeling and Design of Embedded Software*, 2003, pp. 212–237.

[4] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, “On the accuracy of spectrum-based fault localization,” in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*. Windsor, UK: IEEE Computer Society, Aug. 2007, pp. 89–98.

[5] J. A. Jones, M. J. Harrold, and J. Stasko, “Visualization of test information to assist fault localization,” in *ICSE ’02: Proceedings of the 24th International Conference on Software Engineering*. Orlando, FL, USA: ACM, May 2002, pp. 467–477.

[6] P. Zoeteweij, R. Abreu, R. Golsteijn, and A. van Gemund, *Spectrum-based fault localization in practice*. Eindhoven, The Netherlands: Embedded Systems Institute, 2009, ch. 10, pp. 113–124.

[7] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, “Spectrum-based multiple fault localization,” in *24th IEEE/ACM International Conference on Automated Software Engineering*. Auckland, New Zeeland: IEEE Computer Society, Nov. 2009, pp. 88–99.

[8] “Sofl: simulator of fault localization website,” http://swerl.tudelft.nl/bin/view/Main/SOFL, 2011, last accessed Jan. 2012.

[9] É. Piel, A. Gonzalez-Sanchez, H.-G. Gross, and A. J. V. Gemund, “Spectrum-based health monitoring for self-adaptive systems,” in *5th IEEE Int’l Conference on Self-Adaptive and Self-Organizing Systems (SASO’11)*. Ann Arbor, USA: IEEE Computer Society, Oct. 2011.

[10] V. Dallmeier, A. Zeller, and B. Meyer, “Generating fixes from object behavior anomalies,” in *24th IEEE/ACM International Conference on Automated Software Engineering*, Auckland, New Zeeland, Nov. 2009, pp. 550–554.

[11] “Atlas component framework website,” http://swerl.tudelft.nl/bin/view/Main/Atlas, 2010, last accessed Jan. 2012.

[12] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-oriented programming,” in *Proceedings European Conference on Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds. Berlin, Heidelberg, and New York: Springer-Verlag, 1997, vol. 1241, pp. 220–242.

[13] *Technical characteristics for a universal shipborne Automatic Identification System using time division multiple access in the VHF maritime mobile band*, International Telecommunications Union, 2001, Recommendation ITU-R M.1371-1.

[14] Y.-S. Ma, J. Offutt, and Y. R. Kwon, “Mujava: an automated class mutation system: Research articles,” *Software Testing Verification and Reliability*, vol. 15, pp. 97–133, Jun. 2005.

[15] "Berkeley/stanford recovery-oriented computing website," http://roc.cs.berkeley.edu/, 2005, last accessed Jan. 2012.

[16] "XEROX Model-Based Computing project website," http://www2.parc.com/spl/projects/mbc/, 1997, last accessed Jan. 2012.

[17] G. K. Baah, E. Gray, and M. J. Harrold, "On-line anomaly detection of deployed software: a statistical machine learning approach," in *Proc. of the 3rd International Workshop on Software Quality Assurance*. Portland, Oregon, USA: ACM, Nov. 2006, pp. 70–77.

[18] C. Rabejac, J.-P. Blanquart, and J.-P. Queille, "Executable assertions and timed traces for on-line software error detection," in *Annual Symposium on Fault Tolerant Computing*, Sendai, Japan, Jun. 1996, pp. 138–147.

[19] M. Seltzer and C. Small, "Self-monitoring and self-adapting operating systems," in *Proc. of the Workshop on Hot Topics in Operating Systems*, Cape Cod, Massachusetts, USA, May 1997, pp. 124–129.

[20] Z. Chen, "Service fault localization using probing technology," in *Proceedings of the Conference on Networking, Sensing and Control*, Ft. Lauderdale, Florida, USA, Apr. 2006, pp. 937–942.

[21] D. Garlan, S.-W. Cheng, A.-C. Huang, B. R. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *IEEE Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[22] P. Casanova, B. R. Schmerl, D. Garlan, and R. Abreu, "Architecture-based run-time fault diagnosis," in *Proc. of the 5th European Conference on Software Architectures (ECSA'11)*, Essen, Germany, Sep. 2011, pp. 261–277.

Augmenting Reinforcement Learning Feedback with Prediction for Autonomic Management

Khandakar Rashed Ahmed
Department of Computer Science
The University of Western Ontario
London Ontario, Canada, N6A5B7
kahmed25@uwo.ca

Raphael Bahati
Department of Computer Science
The University of Western Ontario
London Ontario, Canada, N6A5B7
rbahati@uwo.ca

Michael Bauer
Department of Computer Science
The University of Western Ontario
London Ontario, Canada, N6A5B7
bauer@csd.uwo.ca

***Abstract*—Autonomic management depends on a feedback loop between the managed system and the autonomic manager. Adding a learning component to the autonomic manager introduces a second feedback loop – between the manager and the learning agent. In this paper, we describe a policy-based autonomic manager that makes use of a reinforcement learning agent. The reinforcement learning model is based on a state-transition model formed from an active set of policies and the actions of the manager. Based upon this model, this paper describes approaches for prediction of potential policy violations and examines the accuracy of the prediction approaches. Experimental results show that a prediction approach based on the likelihood of a violation performs better than a non-prediction approach and has a positive impact on avoiding policy violations.**

Keywords-autonomic management, prediction, policies, reinforcement learning.

I. INTRODUCTION

Autonomic systems are commonly conceived around the notion of a feedback loop, usually involving monitoring, analysis, planning and execution [1]. In some cases, this process may involve a learning component [2-6] which can enable an autonomic manager to adapt aspects of its behavior over time, e.g., to "learn" which actions are better than others in certain situations. Some of our previous work investigated the role of reinforcement learning [7, 8] as a key element in a policy-based system for autonomic management.

Policies are often used to specify the required or desired behavior of a system and its applications. In autonomic systems, policies have been used as the basis for the management system to adjust application or system tuning parameters in order to meet operational requirements [2, 7], i.e., the policies are used to drive the feedback needed to change the system's behavior. When these policies are violated, the autonomic management system tries to identify the actions needed to take based on the policies or, in some cases, based on the past behavior of the system. That is, the management system may incorporate some sort of learning in order to enhance the decisions. The general model of the approach is illustrated in Figure 1. The autonomic management system makes adjustments to the system being managed. Actions taken by the management system and values of metrics are used by a learning component to determine the best actions in the future.

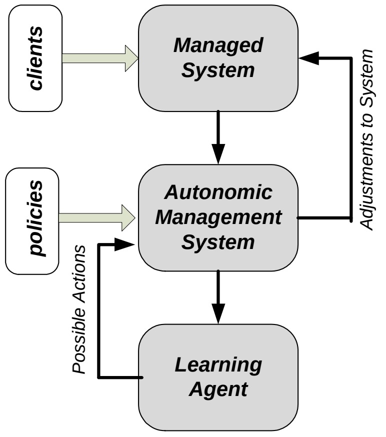


Figure 1. Feedback Loops for Autonomic Manager with Learning Agent

In this paper, we consider how prediction might be considered in the context of such feedback control. Our approach to reinforcement learning entails the construction of a "state model" based on an active set of policies. In this case, the "state model" does not directly correspond to the states of the managed system, but rather captures the states representing the "health" of the system as dictated by the active set of policies. In the simplest form, such a state may indicate that the system is "OK" or "not OK", i.e., has or has not violated a policy. Using this "state model" we look at approaches to prediction – one based on predicting a future state and one predicting whether the system will be "unhealthy", i.e., in any "unhealthy state". We describe the approach and report on results of experiments for a system incorporating prediction.

II. RELATED WORK

A variety of different approaches to prediction in network and system management have been explored. Most of the techniques have dealt with prediction of faults or prediction of resource usage. A fuzzy logic controller prediction method was in [9] to predict computational demand in a utility computing environment.

The probabilistic framework of a Bayesian network has been used to do prediction in several research studies

[10-12]. The work in [10], for example, tries to predict network anomalies that typically precede a fault. Specifically, the authors propose an approach to predict node failure. The intelligent agent learns the normal behavior of each measurement variable and combines the information into a Bayesian network. Work in [12] also used a Bayesian based algorithm to predict disk failures, while in [13], a specific analytical method is developed for fast detection of faults in I/O systems.

Some approaches have looked at data mining and learning approaches learn and classify failure patterns as rules from historical data rather than generating probabilistic models ahead of time [14-16]. Sahoo, et al. applied association rules to predict failure events in a 350-node IBM cluster [17]. Meta-learning [14, 16] methods have been investigated to explore the merits of combining various data mining techniques. The use of reinforcement learning in autonomic management [3-5, 7] has also attracted significant interest.

Our work is similar to work that makes use of probabilistic approaches. The key difference in our work from other work in the autonomic area is that we incorporate prediction based on the learned model. Our work also differs in that our reinforcement learning model uses "policy" states rather than "system" states.

III. MODEL OF REINFORCEMENT LEARNING

A policy-based management system has been developed by Bahati [7] where reinforcement learning is used to determine the best use of a set of active (enabled) policies to meet different performance goals. The learning approach is based on the analysis of past experience of the system and the learning model is used to train the system to dynamically adapt the choice of actions for adjusting application and system tuning parameters in response to policy violations.

Reinforcement learning is a learning paradigm [19-20] where an agent learns how to best map situations to actions through trial and error interaction with its environment. It uses a "reward and punishment" approach, where, for each action, a numeric reward is generated by the agent which indicates the desirability of the agent being in a particular state. The only way to maximize this reward is to discover which action generates the most reward in a given state by trying them. The learning agent must also consider a trade off between whether it should use its current knowledge to select the best action to take (exploit) or to try new, untried, actions (explore) in order to improve its performance.

We assume that policies are used to specify desired behavior and are of the form of event-triggered, condition-action rules [7]. An event triggers the evaluation of a rule of the form "if [conditions] then [actions]". An event is generated when some condition about the state of a system becomes true. The appropriate action is chosen from the policy specification for that event. A policy consists of one or more conditions and an ordered list of actions which can be used by the management system to make adjustments to system tuning parameters. Table 1 illustrates examples of polices; these form the basis for the discussion in this paper.

In Table 1, ***p1*** illustrates a policy where different actions can be taken when the Apache response time (*ART*) is greater than 2000 ms and the trend of the response time (*ARTT*) is increasing. Action ***a1,*** for example, indicates that the limit on the maximum number of active Apache clients should be increased by 25.

TABLE 1. EXAMPLES OF POLICIES

p1*: If Apache's response time (ART) > 2000ms and the trend of the Apache response time (ARTT)) > 0, then*
a1*: Increase MaxClients by 25, or*
a2*: Decrease MaxKeepAliveRequests by 30, or*
a3*: Decrease MaxBandwidth by 128*

p2*: If Apache's response time < 250 ms, then*
a4*: Decrease MaxClients by 25, or*
a5*: Increase MaxKeepAliveRequests by 30, or*
a6*: Increase MaxBandwidth by 64*

The following introduces a number of key terms and concepts related to how we model learning; a more detailed and formal description can be found in [7]. A *policy p* is a pair $<C, A>$, where C is a finite set of *conditions*, $C = c_1, \ldots, c_m$, and $A = a_1, \ldots a_k$ is an ordered set of *actions*. Each *condition*, $c_i \varepsilon C$, is defined by a tuple $c_i = <metricName, operator, \Gamma>$, where *metricName* is the name of a metric measured/monitored by the management system, *operator* is a relational operator, and Γ, is a constant threshold value. The set of active policies at any time within the management system is then $P = \{p_1, \ldots, p_n\}$.

To model the dynamics of the environment from an active set of policies, we define a set of states whose structure is derived from the metrics associated with the active policies. The set of metrics that must be monitored to support an active set of policies $P = \{p_1, \ldots, p_n\}$ is then $M = \{m_1, \ldots, m_t\}$, such that:

$$\forall\, p_j = <C_j, A_j> \varepsilon\, P,\;\; M = \cup_{c_i\, \varepsilon\, C_j} \{c_i.\, metricName\}.$$

The set M is the set of all metrics occurring in any of the active policies. For each metric in this set, there are a finite number of threshold values to which the metric is compared; these can be ordered to form "regions". For each metric $m_i \varepsilon M$, let the set $\sigma_{m_i} = \{\Gamma_1, \Gamma_2, \ldots, \Gamma_k\}$ be the set of thresholds from the conditions associated with metric m_i, such that, $\Gamma_i < \Gamma_j$ if $i < j$. Then, σ_{m_i} induces a set of *metric regions* associated with metric m_i:

$$R_{m_i} = \{R^1{}_{m_i}, R^2{}_{m_i}, \ldots, R^{k+1}{}_{m_i}\}, \text{ where } R^1{}_{m_i} = (-\infty, \Gamma_1), R^2{}_{m_i} = (\Gamma_1, \Gamma_2), \text{ etc., and } R^{k+1}{}_{m_i} = (\Gamma_k, \infty).$$

In essence, for any metric, a measured value of the metric can be mapped uniquely onto one of the regions (i.e., intervals) as defined by the thresholds of the policy conditions. For example, if $\sigma_{m_i} = \{\Gamma_1, \Gamma_2\}$, then there would be three regions:

$$R^1_{m_i} = (-\infty, \Gamma_1), \quad R^2_{m_i} = (\Gamma_1, \Gamma_2), \text{ and } R^3_{m_i} = (\Gamma_2, \infty).$$

We also define a weighting function f over metrics and their regions where $f(R^j_{m_i}) \rightarrow R$, which assigns a numeric value to the j^{th} region, $R^j_{m_i}$, such that, $f(R^j_{m_i}) > f(R^k_{m_i})$, if $k < j$. An example of such a mapping, which we make use of in our current implementation, is defined by:

$$f(R^j_{m_i}) = 100 - (100/(n - 1)) \times (j - 1),$$

where n is the total number of regions in R_{m_i}.

This function assigns a numeric value between 100 and 0 for each metric's region in R_{m_i}, starting from 100 for the most *desirable region* and decrementing at equal intervals towards the opposite end of the spectrum, whose region is assigned a value of 0. Here we assume that the smaller the values of $f(R^j_{m_i})$ are *more desirable*, though in general this is not a necessary requirement. The idea is that regions of greater "desirability", i.e., preferred quality of service, are assigned higher values. Table 2 illustrates the metrics and their regions from the example policies of Table 1.

TABLE 2. METRIC AND REGION FROM SAMPLE POLICIES

Metric	***Condition***	***Region***	$f(R^j_{m_i})$
m_1: ART	ART < 250.0 ART > 2000.0	$R^1_{m_1} = (-\infty, 250.0)$ $R^2_{m_1} = (250.0, 2000.0)$ $R^3_{m_1} = (2000.0, 250.0)$	100 50 0
m_2: ARTT	ARTT > 0.0	$R^1_{m_2} = (-\infty, 0.0)$ $R^1_{m_2} = (0.0, \infty)$	100 0

The key role of these regions is that they partition the space of values that a metric can take on with respect to the thresholds in conditions involving that metric. We use these to define a state within our model. A set of active policies, P, with metrics M, derives a set of states $S = \{s_i\}$, where $s_i = \langle P(s_i), A(s_i), M(s_i), \mu \rangle$, where:

- $P(s_i)$ is the set of policies that were violated when the system was in state s_i.
- $A(s_i)$ is the set of actions associated with the policies in $P(s_i)$, plus the γ-action, representing the "null" or "no-op" action.
- $M(s_i)$ is the set $\{(value_1, R^{r_1}{}_{m_1}, f(R^{r_1}{}_{m_1})), \ldots, (value_n, R^{r_n}{}_{m_n}, f(R^{r_n}{}_{m_n}))\}$, where $value_j$ is the observed measurement of metric m_j or its average value when state s_i is visited multiple times and $R^{r_j}{}_{m_j} = (\Gamma_1, \Gamma_2)$, where $\Gamma_1 < value_j < \Gamma_2$, i.e., the region of m_i in which the measured value $value_j$ falls. Essentially, each state has a unique region from <u>each</u> metric of M along with a measured value of that metric, i.e., for a set of policies with n metrics, each state would have n metrics $\{m_1, m_2, \ldots, m_n\}$ and for each of those metrics there would be a single metric region.
- μ defines the "health" of the state, that is, is either "violation" or "acceptable" depending, respectively, on whether or not there are any policies violated when visiting this particular state.

Transitions are determined by the actions taken by the management system and labeled by a value determined by the learning algorithm. A state transition, $t_i(s_p, a_p, s_c)$, is a directed edge corresponding to a transition originating in state s_p and ending at state s_c as a result of taking action a_p while in state s_p and is labeled by $\langle \lambda, Q_{t_i}(s_p, a_p) \rangle$, where λ is the frequency (i.e., the number of times) the transition has occurred and $Q_{t_i}(s_p, a_p)$ is the action-value estimate from the reinforcement learning algorithm associated with taking action a_p in state s_p. In our current implementation, this value is computed using a one-step Q-Learning [20] algorithm which has been described elsewhere [7].

For a set of active policies, P, the state-transition model can be defined by the graph $G^P = \langle S, T \rangle$, where S is a set of states and T is a set of state transitions. The construction of states and transitions is naturally done at run-time (i.e., on-line) and not *a priori* given an active set of policies (though, this could be done). In practice, many of the states may never occur, thus keeping the size of the model manageable.

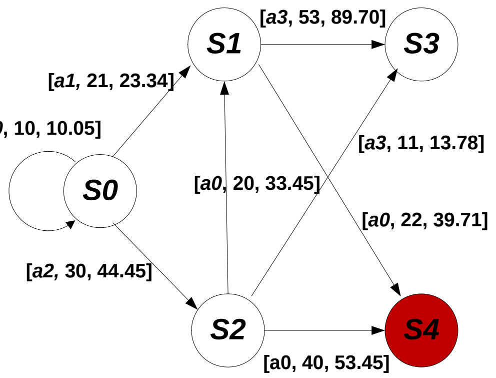


Figure 2. State Graph

Figure 2 shows several states, and for each transition the action taken, the number of times that action was taken, and a reward value as determined from the learning algorithm. Action ***a0*** represents the "null" action, that is, no action was taken, but the system moved from one state to another (e.g., ***S0*** to itself). State ***S4*** (colored) is a "violation" state. After each management cycle, the system updates the state graph information either by adding a new state or by updating the previous state information which includes an update of the transition frequency and reward value of actions.

IV. PREDICTION

In this section, we outline our approaches for prediction based on the state-transition model introduced in the previous section. We consider two different strategies for prediction - prediction of the next "state" within the state-transition graph and prediction of whether there might be some policy violation, i.e., move to an "unhealthy" state. The first strategy is a straightforward approach given that our model is comprised of states and transitions with frequencies of occurrences included. The latter, though, similar, originates from our specific interest in policy-based autonomic management and that the primary concern is to avoid policy violations. As a result, this strategy tries to predict the likelihood of any policy violation or not.

For our current work, we decided to predict two management cycles ahead – a single cycle ahead was "too close" while two cycles ahead seemed to be a good starting point, though more might be of more interest. This means that if we are currently at management cycle *t*, then we will try to predict whether there will be any policy violation at management cycle *t+2* by predicting which state is most likely or by predicting the likelihood of some policy being violated. As indicated, states in the reinforcement learning model contain frequency information as well as reward values generated from the learning algorithm. We further consider prediction using the frequency values (*probability* approach) and one based on just using reward values (*reward* approach).

A. Probability Approach

The probability is calculated from the action frequency values (from the labels of each edge in the state-transition graph). The action frequency value indicates the number of times that an action has been taken from a particular state. From the frequency values, we can compute the probability of transitioning from a state to an adjacent state and then states two transitions away. Considering Figure 2 and assuming that the system is in state ***S0,*** the probability of states two transitions away is presented in Table 3. Multiple values in a single cell of Table 3 indicate multiple paths, e.g., from ***S0*** to ***S1*** there are two paths and so there are two separate probability values.

B. Reward Approach

The reward approach only considers the action reward value (generated from the reinforcement learning algorithm) for prediction analysis. The action reward values are summed for all states on paths two transitions away from the current state. These are shown in Table 3, again, assuming that state ***S0*** is the current state.

C. Predicting State

When we want to predict the state, we compute the probabilities of reaching each state two transitions away. The state with the highest probability is the chosen state and depending on whether that state is a "violation" state or not determines whether the prediction indicates a violation or not. Similarly, in using reward, the sum of the reward values is used and the state reached with transitions that have the highest reward total is the state selected. In the previous example, state ***S4*** is selected based on using probabilities and state ***S3*** is selected based on reward values.

TABLE 3. PROBABILITIES AND REWARDS FOR STATES

State (2 transitions from S0)	***Probability***	***Reward***
S0	0.03	20.10
S1	0.06 0.14	33.39 77.90
S2	0.08	54.50
S3	0.24 0.07	113.04 58.23
S4	0.10 0.28	63.05 97.90

D. Predicting Likelihood of a Violation

In contrast, predicting the "likelihood" of a violation involves computing a score for all "violated" states reachable in two steps from the given state. In this case, we compute a score for "not violated" and one for "violation" states. We do this by summing the probabilities or summing the rewards for states that are "not violated" and those that are "violated".

Using the probabilities and rewards from Table 3, the likelihood scores are shown in Table 4. Here, we see that using approaches based on the probabilities and on the reward suggest that there is no expected violation. This is consistent with the state graph of Figure 2.

TABLE 4. PREDICTION OF FUTURE CONDITIONS

Future Condition	***Likelihood (Probability)***	***Likelihood (Reward)***
No Violation	0.72	420.21
Violation	0.28	97.90

V. EXPERIMENTAL RESULTS

But, how accurate are these predictions? In the following, we outline experiments to evaluate the prediction approaches.

A. Experimental Environment

The experimental environment consists of networked workstations. A Linux workstation with a 2.0 GHz processor and 2.0 Gigabytes of memory is used to host an Apache Web Server, the Knowledge Base and the MySQL database server. Three network workstations are used to run the traffic load tool for generating server requests. The three workstations represent load for gold, silver and bronze users and their service classes. Linux Traffic Controller (TC) Tool is used to control the bandwidth associated with the gold, silver, and bronze service classes. Thus, given a ratio of bandwidth for each of the service classes, the bandwidth is shared accordingly; for our experiments this ratio was 85:10:5. A tuning parameter MaxBandwidth determines bandwidth which needs to be assigned to each service class. Apache Jmeter is used as a traffic load generator. The Jmeter application runs in each of the workstations where each has a dynamic load testing plan. All workstations generate traffic load using the same plan. The load plan contains dynamic requests which create situations where the system resource usage is increased at significant rate.

B. *Prediction Accuracy*

Experiments were run with the above experimental environment for 1 and 4 hours. The accuracy of prediction results is present in Table 5.

TABLE 5. PREDICTION ACCURACY

Approach	***State (1 hour)***	***State (4 hours)***	***Violation (1 hour)***	***Violation (4 hours)***
Probability	20.00%	3.90%	29.62%	29.66%
Reward	26.19%	7.69%	45.76%	37.03%

Predicting a single state is clearly less successful that predicting the likelihood of a violation which could include multiple states. In predicting a single state, the accuracy dramatically decreases during the four hour run. This is because the size of the state graph has grown and so predicting a single state is much harder. There is a much smaller reduction in accuracy for the four hour experimental run when predicting the likelihood of a violation. It is also interesting to note that the use of the reward values for prediction proved to be more accurate in both cases than the uses of probabilities.

B. *Experiments with Prediction*

Given the evaluation of the accuracy of the prediction approaches, we decided to evaluate the likelihood approach to prediction in the context of our prototype web environment and autonomic manager. Our objective for looking at prediction was to be able to avoid policy violations, that is, if our predictive mechanism did predict that a violation was likely, then the autonomic manager could take action prior to the violation.

Our approach is outlined as follows. If the prediction mechanism (probability based or reward based) predicts that a violation was likely to occur, then our prediction component would look for possible safe states and the transitions that would take the system to a safe state two steps away (our consideration of what happens at management cycle *t+2*). The state selected is the safe state with the highest value as per the prediction computation. The algorithm determines the two actions on the transitions to that safe state from the current state. These are then passed to the autonomic manager for execution.

If no safe state is available, then there are two possibilities – do nothing, i.e., let the autonomic manager rely strictly on its reinforcement learning algorithm to select an action, or have another mechanism for choosing an action. We have explored the latter [21], but details of how this works is beyond the scope of this paper.

We compared the use of prediction to that of no prediction. The "no prediction" method relied on the autonomic manager and the reinforcement learning component, which performed very well in adapting the system in previous experiments [7]. Experiments were done for each of the 1hr and 4hr testing periods with traffic load varying during the test periods. Since our aim is to reduce policy violations, we counted the number of policy violations that occurred during the testing period; each experiment was run three times and the average used.

The results are presented in Table 6. The existing management system encountered 77 and 280 policy violations in the 1hr and 4hr time periods, respectively. When we add prediction, the number of policy violations is reduced to 60 and 226 in the 1hr and 4hr time periods.

TABLE 6: POLICY VIOLATIONS: WITH AND WITHOUT PREDICTION

Approach	***Policy Violations (1 hour)***	***Policy Violations (4 hours)***
Reinforcement Learning (Existing)	77	280
Prediction: Likelihood of Violation (Probability)	62	220
Prediction: Likelihood of Violation (Reward)	61	226

Using prediction resulted in approximately a 20% reduction in the number of policy violations encountered in both the one hour and four hour test periods.

VI. CONCLUSIONS AND FUTURE WORK

Given the results, it is clear that our prediction technique should only predict whether a policy violation is likely to occur or not, rather than trying to predict a state. The results of prediction with the reinforcement learning resulted in useful feedback to the autonomic manager with experimental results showing roughly a 20% improvement in the number of violations encountered. This result is a little surprising in that the accuracy of the likelihood prediction approach was only around 38% for the reward approach and 30% for the probability approach (4 hour test period). Would this continue for a longer test period? If the prediction accuracy was increased, would the improvement in the number of violations continue? These are future areas of study.

There are, of course, a number of other areas for exploration, the obvious being to consider this approach in a different scenario and with more policies. More immediate work could include looking at some combination of probability and reward or some combination of predicting a state and predicting the likelihood of a violation to see if there might be a useful alternative evaluation mechanism that could result in increased prediction accuracy. Other work could look at prediction more than two cycles ahead to see how accuracy changes. Finally, it would be useful to develop a more formal basis for understanding how prediction and reinforcement learning are dependent on each other and their use in autonomic management.

REFERENCES

[1] R. Murch, Autonomic Computing. IBM Press., 2004.

[2] J. O. Kephart and W. E. Walsh, "An Artificial Intelligence Perspective on Autonomic Computing Policies", IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'04), 2004, pp. 3–12.

[3] G. Tesauro, "Online Resource Allocation Using De-compositional Reinforcement Learning", Association for the Advancement of Artificial Intelligence (AAAI'05), 2005.

[4] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, "A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation", International Conference on Autonomic Computing (ICAC'06), Dublin, Ireland, June 2006, pp. 65–73.

[5] D. Vengerov and N. Iakovlev, "A Reinforcement Learning Framework for Dynamic Resource Allocation: First Results", International Conference on Autonomic Computing (ICAC'05), Seattle, WA, USA, January 2005, pp. 339–340.

[6] P. Vienne and J. Sourrouille, "A Middleware for Autonomic QoS Management based on Learning", International Conference on Sofware Engineering and Middleware, Lisbon, Portugal, September 2005, pp. 1–8.

[7] R. M. Bahati and M. A. Bauer, "Modelling Reinforcement Learning in Policy-driven Autonomic Management", International Journal On Advances in Intelligent Systems, 2008, vol. 1, no. 1, pp. 54-79.

[8] R. M. Bahati, M. A. Bauer, and E. M. Vieira, "Adaptation Stratergies in Policy-Driven Autonomic Management", International Conference on Autonomic and Autonomous Systems (ICAS'07), Athens, Greece, July 2007, pp. 16-21.

[9] A. Andrzejak S. Graupner and S. Plantikow. "Predicting Resource Demand in Dynamic Utility Computing Environments", International Conference on Autonomic and Autonomous Systems (ICAS), 2006, pp. 6-6.

[10] C. Hood and C. Ji. "Intelligent Agents for Proactive Network Fault Detection", IEEE Internet Computing, 1998, Vol.2, 65-72.

[11] J. Ding and X. Li and N, Jiang and Kramer, B.J. and Davoli, "Prediction Strategies for Proactive Management in Dynamic Distributed Systems", International Conference on Digital Telecommunications. 2006, pp. 74-79.

[12] G. Hamerly and C. Elkan, "Bayesian Approaches to Failure Prediction for Disk Drives", Proceedings of International Conference on Machine Learning (ICML), 2001, pp. 202-209.

[13] K. Shen, M. Zhong, C. Li., "I/O System Performance Debugging Using Model-driven Anomaly Characterization", 4th USENIX Conference on File and Storage Technologies, 2005, pp. 309-322.

[14] P. Gujrati and Y. Li and Z. Lan and R. Thakur and J. White, "A Meta-learning Failure Predictor for Bluegene/L Systems", Proceedings of International Conference on Parallel Processing (ICPP), 2007, pp. 40-40.

[15] Y. Liang and Y. Zhang and A. Sivasubramanium and R. Sahoo, "BlueGene/L Failure Analysis and Prediction Models", Proceedings of Dependable Systems and Networks (DSN), 2006, pp. 425-434.

[16] J. Gu and Z. Zheng and Z. Lan and J. White and E. Hocks and B. Park, "Dynamic Meta-Learning for Failure Prediction in Large-Scale Systems: A Case Study", Proceedings of International Conference Parallel Processing (ICPP), 2008, pp. 157-164.

[17] R.K. Sahoo and A.J. Oliner et al., "Critical event prediction for proactive management in large-scale computer clusters", Proceedings of Knowledge Discovery and Data Mining (KDD), 2003, pp. 426-435.

[18] G. A. Hoffmann, F. Salfner and M. Malek. Advanced Failure Prediction in Complex Software Systems, Research Report, No. 172, Department of Computer Science, Humboldt University Berlin, 2004.

[19] R. S. Sutton and A. G. Barto, Reinforcement Learning: an Introduction. MIT Press, 1998.

[20] L. P. Kaelbing, M. L. Littman, and A. W. Moore, "Re-inforcement Learning: A Survey", Journal of Artificial Intelligence Research, April 1996, pp. 237–285.

[21] R. A. Khandekar. Policy-Based Proactive System Management: Predicting Faults in Advance. MSc. Thesis, Department of Computer Science, The University of Western Ontario, 2010.

Autonomic Computing in the First Decade: Trends and Direction

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

***Abstract* — The Autonomic Computing (AC) concept has received strong interest amongst the academic and industrial research communities since its introduction exactly a decade ago. It is important, after the first decade, to evaluate the actual work done in achieving the original vision of this concept. In this short paper we present a brief report of our work in this direction. We have analyzed all the proceedings (2004 – 2011) of two leading AC conferences (ICAC and ICAS) to show the trends in and direction of AC research and to identify current and future research challenges.**

Keywords- autonomic computing; trends and direction

I. INTRODUCTION

The International Conference on Autonomic Computing (ICAC) and the International Conference on Autonomic and Autonomous Systems (ICAS) are two leading AC conferences and have together published about 647 high quality research papers in eight years of the first ten years of AC research. We believe that the two conferences give a true representation of the distribution of interest, work done, and trends in AC research. Papers used in this work are sourced from [1]. AC research is widely viewed to have started with the publication of [46] in 2001 introducing the concept of AC and [47] elucidating further the AC vision. However, only high level analysis, requirements and challenges of AC were presented.

Jeffrey Kephart in a keynote during ICAC 2011 presented an excellent analysis of the extent to which the original AC vision has been realized, and some discussion and speculation about the remaining research challenges [2]. While Kephart concentrated more on the various technological threads, their origins and how they have progressed, our focus is mainly on the level of maturity in terms of the types of, and scale of, problems targeted at the various stages. This enables us to reflect on the overall progress in the field, and to be able to identify current and future challenges. Our work is not just a review but also a validation of our earlier proposed roadmap (pathway) to achieving the goal of autonomic computing [21].

We reviewed a total of 647 research publications including keynotes (336 of which are from ICAC and 311 from ICAS) using webometrics and direct analysis techniques. These are analyzed in terms of main application domain, emphasis, and technical approach as well as author distribution. Our result is an empirical evaluation of the overall impact, trends and state-of-the-art of AC research activity.

An analysis-by-problem approach reveals a particular pattern (problem definition to issues of scale) in tackling the AC vision. On the horizon there is the challenge of coexistence and interoperability between Autonomic Managers and yet beyond the current state-of-the-art, and even further beyond state-of-practice are issues of validation, trustworthiness and certification, requiring solutions specifically tailored for run-time self-adaptive systems.

Overall, very impressive progress has been made in the first decade, and this has been driven by the interest of the main sponsors – industry leaders such as IBM, Sun, Motorola, Google, Microsoft and Hewlet Packard, amongst others.

The remainder of this paper is organised as follows: Section II gives a high level and general analysis of all conference proceedings. Section III discuses trends and direction, showing the pattern of how the research challenge is being tackled by the AC research community while Section IV concludes the work.

II. HIGH LEVEL AND GENERAL ANALYSIS

Tables I and II are high level analysis of conference proceedings mainly taken from IEEE Computer Society Digital Library [1]. A select few areas have been chosen and some of these are discussed in this first report. In terms of authoring, the academic community has the most publications. While ICAS is academic dominant, ICAC has been predominantly industry driven until recently. This explains why on the average even though ICAS has more publications ICAC has a far greater number of datacenter-oriented papers and has been somewhat dominated by this application domain. In terms of emphasis, contrary to popular assumption that self-optimization takes the top shot, our investigation actually shows that the predominance of work in the field continues to focus on self-healing followed by self-configuration, self-optimization and then self-protection. Both conferences maintain the same trend. Out of all the self-CHOP (self-configuration, self-healing, self-optimization and self-protection) based publications in Tables I and II put together, 35% focus on self-healing while 27% on self-configuration, 22% are on self-optimization and 16% on self-protection (Figure 1). In terms of technical approaches, good progress has been made in using specific techniques including machine learning [3, 4, 5], fuzzy logic [6, 7], utility functions [8, 9] and policies [10, 11, 12] to define and achieve self-managing capabilities. Alternative autonomic architectures (e.g., Intelligent Machine Design [13]) have also been proposed.

Figure 1: Self-CHOP analysis in terms of emphasis of work in the community.

In terms of application domain, the datacentre clearly tops the ranking in terms of interest to the community. This is partly because the AC vision is industry-borne and has continued to be driven by the industry. This is evidenced by the number of papers (including on datacentre) that are authoured, co-authoured or sponsored by the industry partners. Datacentres are very complex; in fact have many dimensions of complexity; which arise from their scale, necessary speed of operation, and large number of tuning parameters. In addition they have high power costs, including a significant cost component for the cooling systems. Autonomic Computing arose because of the need for automatic management of such complexity and successful autonomic techniques in this domain translate into significant financial reward for the owners and users of such systems. This high complexity is also attractive to academic researchers as it provides a rich domain in which to evaluate a wide range of techniques, tools and frameworks for AC.

With the vested interest, it is clear why the industry takes the lead in datacentre related research when the industry led ICAC is compared with the academic led ICAS (Figure 2). While the influence is understandably obvious for ICAC, the academic community, in ICAS, has diversified the research to cover other areas more evenly.

TABLE I. ICAC PROCEEDINGS DISTRIBUTION

Distribution	icac 04	icac 05	icac 06	icac 07	icac 08	icac 09	icac 10	icac 11	Total
Authouring									
Academic	39	30	20	15	15	18	18	32	187
Industry	17	18	09	06	05	10	04	01	70
Joint	08	16	14	11	06	06	05	13	79
Total	64	64	43	32	26	34	27	46	336
Main Application Domain									
Datacentre	03	11	11	11	09	10	09	12	76
Distributed Systems	17	06	05	04	00	01	02	04	39
Networks	08	02	00	01	00	00	01	03	15
Robotics	01	00	00	00	00	00	00	02	03
Storage & Dbase Mgt	05	05	04	02	00	00	01	04	21
Others									
Design/ Architecture	07	12	01	02	04	03	03	03	35
Learning/ knowledge	08	04	03	01	06	03	01	03	29
Performance Mgt	09	05	05	03	01	06	03	08	40
Policy	02	06	03	02	02	00	01	00	16
Self-CHOP	11	09	04	05	07	06	04	02	48
Survey	00	00	00	00	00	00	00	01	01
VTC	04	03	03	04	02	03	00	00	19
Actual VTC proposal	01	01	01	03	01	01	00	00	08

TABLE II. ICAS PROCEEDINGS DISTRIBUTION

Distribution	icas 05	icas 06	icas 07	icas 08	icas 09	icas 10	icas 11	Total
Authouring								
Academic	20	39	53	34	48	27	23	244
Industry	01	10	13	00	04	01	01	30
Joint	02	09	03	09	05	02	07	37
Total	23	58	69	43	57	30	31	311
Main Application Domain								
Datacentre	01	06	04	03	03	04	02	23
Distributed Systems	05	12	07	01	05	01	02	33
Networks	04	07	06	02	05	03	01	28
Robotics	01	03	01	04	04	01	03	17
Storage & Dbase Mgt	00	04	03	01	03	00	01	12
Others								
Design & Architecture	03	15	07	02	09	03	07	46
Learning & knowledge	00	01	04	06	04	00	01	16
Performance Mgt	01	05	07	03	06	02	00	24
Policy	00	02	02	03	03	02	00	12
Self-CHOP	00	01	01	01	03	03	01	10
Survey	00	01	02	01	03	00	01	08
VTC	01	03	01	00	00	01	03	09
Actual VTC proposal	00	00	01	00	00	00	00	01

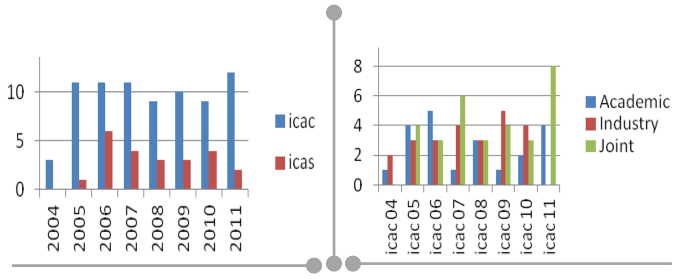


Figure 2: Distribution of datacentre related publications.

But, there is also a noticeable industry influence on ICAS; In the first year (2005) of ICAS there was only one datacentre related paper but the second year saw a jump and at the same time the industry participation on ICAS also saw a jump almost with the same margin. This could be arguably one other reason why the academic community's interest has significantly drifted towards datacentre (Figure 3).

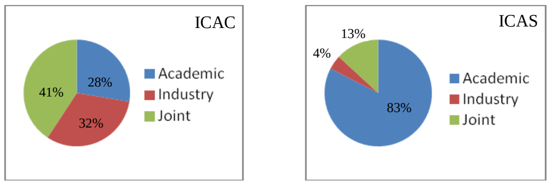


Figure 3: Authour distribution of datacentre related publications.

In general, good progress has been recorded in achieving the AC vision with growing inter-disciplinary collaborations as well as industry and academic partnerships. The industry has a visible influence over the research direction notwithstanding the lead by the academia (in terms of number of publications –Tables I and II). This is a key factor in why datacentre is the most addressed application domain. Figure 3 shows how the academic community is responding to this influence. Industry inspired and driven ICAC is one of the first conferences to address the AC vision while ICAS is a leading academia response to the challenge. Kephart [2] also concludes that in terms of application domain, the datacentre has emerged as the primary area of interest to the AC research community. With this fact we draw, in Section III, the AC research trends, direction and remaining challenges using datacentre as case study.

III. TRENDS AND DIRECTION

We believe that trends in datacentre research will reflect similar patterns in other application domains. So the analysis in this section will mainly focus on datacentre. We use analysis-by-problem approach (Figure 4) to show the pattern (in terms of maturity stages) of how the research challenge is being tackled by the AC research community.

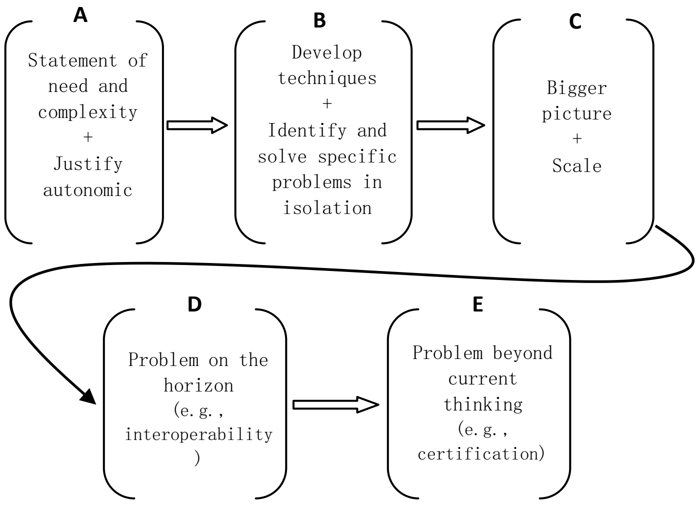

A
Statement of need and complexity + Justify autonomic
B
Develop techniques + Identify and solve specific problems in isolation
C
Bigger picture + Scale
D
Problem on the horizon (e.g., interoperability)
E
Problem beyond current thinking (e.g., certification)

Figure 4: Observed trend and direction of AC research.

Figure 4 shows the stages (A - C) the community has adopted in addressing autonomic computing and our view of the future challenges (D and E) towards achieving the goal of autonomic computing. We keep this to a high level, but appreciate that finer-grained sub stages exist. We classify the stages against a maturity timeline, as shown in Table III.

TABLE III. STAGE CLASSIFICATION FOR ALL PROCEEDINGS

	early stage (A)	middle stage (B)	current stage (C)
ICAC	2004 - 2005	2006 - 2008	2009 - 2011
ICAS	2005 - 2006	2007 - 2008	2009 - 2011

Our investigation reveals that in the early stage research focused mainly on stating the problem and challenge of ever growing system complexity [14, 15], the need for solution and justifying autonomicity as that solution [16, 17]. Majority of work in this area are hinged on dynamic resource allocation [18, 19, 20] and are industry (e.g., IBM, HP, Sun, etc) dominant (Table III). Towards the middle stage the community intensified effort in developing and applying techniques which have now been established and are increasingly used in today's research e.g., policy-driven autonomics [11, 41], utility functions [42, 43], fuzzy logic [6, 44] etc. Also progress was made in identifying and solving specific problems in isolation. A significant number of papers offered specific solutions to specific problems, e.g., [23, 24, 25, 26, and 27]. Some examples of the variety of these include; [27] proposes a control scheme for dynamic resource provisioning in a virtualized datacentre environment to address issues of power management without trading performance. Experiments report that the controller, while still maintaining QoS goals, is able to conserve power by 26%. [25 and 26] investigate thermal load management to address heating in datacenters. While Justin *et al* [25] concentrated on predicting the effects of workload distribution and cooling configurations on temperature (deducing heat profile), Saeed *et al* [26] based their work on workload scaling. Radu Calinescu in [24] implemented an earlier proposed generic autonomic framework (based on service-oriented architecture) and demonstrated the effectiveness of his framework in resource allocation while [23] presents an automatic diagnosis framework to dynamically identify bottlenecks in large systems. Virtualisation and power management [27, 28] are also of interest in this area. Work in this stage largely comprise of implementations, demonstrations and presentation of experimented results of proposed ideas. Towards the end of our list of reviewed papers we discover that the community is now addressing the bigger picture with concern now more to do with scale [29, 30, 31], and generalisation of techniques so as to make re-usable solutions. At this stage issues of heterogeneity of services and platforms [32, 33] began to arise. The community is now addressing large scale datacentres with diverse heterogeneous platforms. The increase in scale and size of datacentres coupled with heterogeneity of services and platforms means that more Autonomic Managers could be integrated to achieve a particular goal. This has led to the need for interoperability between Autonomic Managers.

Interoperability has been somewhat neglected as a challenge to date. Earlier work was fundamentally concerned with getting autonomic computing to work and establishing fundamental concepts and demonstrating viability. Many mechanisms and techniques have been explored. Now that the concept of autonomic computing is well understood and widely accepted the focus can shift to the next level; - i.e. how to manage multi-manager scenarios, to govern interactions between managers and to arbitrate when conflicts arise. These are the kind of problems on the horizon. For example, when more than one autonomic manager is needed to coordinate a system, there may be situations where one manager counters the decision of another. There have been a few mentions and general discussion around this problem [34, 35, 37] lately. The community has not yet made good progress on this though there are efforts on the way. For example Richard *et al* [36] evaluates the nature and scope of the interoperability challenges for AC systems, identifies a set of requirements for a universal solution and proposes a service-based approach to interoperability to handle both direct and indirect conflicts in a multi-manager scenario. In this approach, an Interoperability Service interacts with autonomic managers through a dedicated interface and is able to detect possible conflicts of management interest. In this way the Interoperability Service manages all interoperability activities by granting or withholding management rights to different autonomic managers as appropriate.

On the other hand, beyond current mainstream thinking are problems of validation, trustworthiness and certification. A lot of questions have not been considered or fully answered. For example, 'what are the processes to ensure that component upgrades that are tested and confirmed in isolation will not cause harm in a multi-system environment?', 'how can certified autonomic systems be achieved?' and 'how can users be confident that a system does what it says?' [38]. In Tables I and II a number of Validation, Trustworthiness and Certification (VTC) related papers have been published but only a few are actual VTC methodologies and only one of these [39] considers datacentre. The number for VTC includes mainly those papers that incorporated validation, testing and reliability into their architectures, frameworks or implementations and not necessarily as a core or critical feature. For example, in seven years of ICAS only one paper [37] proposes a method. The work in [37] presents a framework (based on model checking) for verifying and detecting constraint violation when two or more workflows are executed on the same system as a way of ensuring system trustworthiness. The few in ICAC include [38], [39] and [40]. Hoi *et al* [38] 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 AC system users. Shinji *et al* [39] 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. In all the reviewed papers, this is the only VTC method implemented with datacentre case study. Heo and Abdelzaher [40] presented '*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.

Our research group has been working on this problem for some time and in ICAS 2011 we presented several works [13, 21, and 22] identifying the problems of robust design, validation and related issues on trustworthiness leading to certification. In [21], we outline the challenges in current autonomic system validation methods and propose a strategy leading to the achievement of autonomic systems certification. This strategy is a roadmap defining the stages or processes in the journey towards full autonomic computing. We posit that there are significant limitations to the way in which AC systems are validated, with heavy reliance on traditional design-time techniques, despite the highly dynamic behaviour of these systems in dealing with run-time configuration changes and environmental and context changes. These limitations ultimately undermine the trustability of these systems and are barriers to eventual certification. Haffiz, Richard and Mariusz [13] proposed a framework that will allow for proper certification of AC systems. Central to this framework is an alternative autonomic architecture based of Intelligent Machine Design which draws from the human autonomic nervous system. James, Richard and Miltos [22] demonstrated 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. We strongly believe that certification is critical to achieving the full goal of AC. We have a longer term vision to develop trustworthy and certifiable autonomic systems and hope to progress towards this through defining validation techniques. We propose that one vital step in this chain is to introduce robust techniques by which the systems can be described in universal language, starting with a description of, and means to measure the type and extent of autonomicity (autonomic functionalities) they provide [45]. Another of our current focus area is interoperability [36] where we are evaluating the nature and scope of the interoperability challenges for AC systems, identifying a set of requirements for a universal solution and proposing a service-based approach to interoperability to handle both direct and indirect conflicts in a multi-manager scenario.

IV. CONCLUSION

We have presented a review and analysis of the actual work done in achieving the original vision of autonomic computing (AC) after the first decade. We reviewed all ICAC and ICAS proceedings (2004 – 2011) and have shown what the trends and directions there are in the AC research. Our investigation transcends technologies and how they have progressed to include areas, origins and scale of maturity. Our results also show the current and future (or remaining) challenges facing the AC research community. Beyond being a review, this work also illustrates a pathway to achieving the goal of AC and validates our earlier proposed roadmap [21].

The community has made good progress in terms of autonomic technologies and in terms of collaboration or partnership between the industry and academia. Though the research is driven by the industry (the major sponsors) the academia has also woken to the challenge. In terms of application domain, the datacentre appears to dominate the interest of the community. This is chiefly because AC is industry borne and also the datacentre provides the academia a rich and complex environment for diverse implementations and testing. As systems grow in complexity and scale, the community must now deal with addressing issues of interoperability in multi-manager scenarios. This is one of the critical issues on the horizon. Beyond current thinking, the community will need to provide answers to issues of validation, trustworthiness, standardisation and certification of autonomic computing systems.

REFERENCES

[1] IEEE Computer Society Digital Library via http://www.computer.org/portal/web/csdl/proceedings/i#1 – last viewed 12th January 2012

[2] Jeffrey Kephart, *Autonomic Computing: The First Decade*, Keynote at the 8th International Conference on Autonomic Computing (ICAC), 2011, Germany

[3] Han Li and Srikumar Venugopal, *Using Reinforcement Learning for Controlling an Elastic Web Application Hosting Platform*, The 8th International Conference on Autonomic Computing (ICAC), 2011, Karlsruhe, Germany

[4] Jonathan Wildstrom, Peter Stone and Emmett Witchel, *CARVE: A Cognitive Agent for Resource Value Estimation*, The 5th International Conference on Autonomic Computing (ICAC), 2008, Illinois, USA

[5] Artur Andrzejak, Sven Graupner and Stefan Plantikow, *Predicting Resource Demand in Dynamic Utility Computing Environments*, The 2nd International Conference on Autonomic and Autonomous Systems (ICAS), 2006, USA

[6] Ting-Jung Yu, Robert Lai, Menq-Wen Lin and Bo-Rue Kao, *A Fuzzy Constraint-Directed Autonomous Learning to Support Agent Negotiation*, The 3rd International Conference on Autonomic and Autonomous Systems (ICAS), 2007, Athens, Greece

[7] Biplav Srivastava, Joseph Bigus and Donald Schlosnagle, *Bringing Planning to Autonomic Applications with ABLE*, The 1st International Conference on Autonomic Computing (ICAC), 2004, New York, USA

[8] Gerald Tesauro,, Rajarshi Das, William Walsh and Jeffrey Kephart, *Utility-Function-Driven Resource Allocation in Autonomic Systems*, The 2nd International Conference on Autonomic Computing (ICAC), 2005, Seattle, USA

[9] William Walsh, Gerald T Tesauro, Jeffrey Kephart and Rajarshi Das, *Utility Functions in Autonomic Systems*, The 1st International Conference on Autonomic Computing (ICAC), 2004, New York, USA

[10] Andres Quiroz, Manish Parashar, Nathan Gnanasambandam and Naveen Sharma, *Autonomic Policy Adaptation using Decentralized Online Clustering*, The 7th International Conference on Autonomic Computing (ICAC), 2010, Washington, USA

[11] Richard Anthony, *Policy-centric Integration and Dynamic Composition of Autonomic Computing Techniques*, The 4th International Conference on Autonomic Computing (ICAC), 2007, Florida, USA

[12] Liliana Rosa, Ant´onia Lopes and Lu´ıs Rodrigues, *Policy-Driven Adaptation of Protocol Stacks,* The 2nd International Conference on Autonomic and Autonomous Systems (ICAS), 2006, California, USA

[13] Haffiz Shuaib, Richard Anthony, and Mariusz Pelc, *A Framework for Certifying Autonomic Computing Systems*, The 7th International Conference on Autonomic and Autonomous Systems (ICAS), 2011, Venice, Italy

[14] Sharath Musunoori, Geir Horn, Frank Eliassen, and Alia Mourad, *On the Challenge of Allocating Service Based Applications in a Grid Environment*, The 2nd International Conference on Autonomic and Autonomous Systems (ICAS), 2006, California, USA

[15] Anand Ranganathan and Roy Campbell, *Self-Optimization of Task Execution in Pervasive Computing Environments*, The 2nd International Conference on Autonomic Computing (ICAC), 2005, Seattle, USA

[16] Sai Mahabhashyam and Natarajan Gautam, *Dynamic Resource Allocation of Shared Data Centers Supporting Multiclass Requests*, The 1st International Conference on Autonomic Computing (ICAC), 2004, New York, USA

[17] Daniel Menasc´e and Mohamed Bennani, *Autonomic Virtualized Environments*, The 2nd International Conference on Autonomic and Autonomous Systems (ICAS), 2006, California, USA

[18] David V. Nikolai I., *A Reinforcement Learning Framework for Dynamic resource Allocation: First Results*, The 2nd International Conference on Autonomic Computing (ICAC), 2005, Seattle, USA

[19] Mohamed Bennani and Daniel Menasc´e, *Resource Allocation for Autonomic Data Centers using Analytic Performance Models,* The 2nd International Conference on Autonomic Computing (ICAC), 2005, Seattle, USA

[20] Choong Lee and Hyun Kim, A Part Release considering Tool Scheduling and Dynamic Tool Allocation in Flexible Manufacturing Systems, The 2nd International Conference on Autonomic and Autonomous Systems (ICAS), 2006, California, USA

[21] Thaddeus Eze, Richard Anthony, Chris Walshaw, and Alan Soper, *The Challenge of Validation for Autonomic and Self-Managing Systems,* The 7th International Conference on Autonomic and Autonomous Systems (ICAS), 2011, Venice, Italy

[22] James Hawthorne, Richard Anthony, and Miltos Petridis, *Improving the Development Process for Teleo-Reactive Programming Through Advanced Composition*, The 7th International Conference on Autonomic and Autonomous Systems (ICAS), 2011, Venice, Italy

[23] Darcy G. Benoit, *Performance Diagnosis for Changing Workloads*, The 3rd International Conference on Autonomic and Autonomous Systems (ICAS), 2007, Athens, Greece

[24] Radu Calinescu, *Implementation of a Generic Autonomic Framework*, The 4th International Conference on Autonomic and Autonomous Systems (ICAS), 2008, Gosier, Guadeloupe

[25] Justin Moore, Jeffrey Chase and Parthasarathy Ranganathan, *Weatherman: Automated, Online, and Predictive Thermal Mapping and Management for Data Centers*, The 3rd International Conference on Autonomic Computing (ICAC), 2006, Dublin, Ireland

[26] Saeed Ghanbari, Gokul Soundararajan, Jin Cheng and Cristiana Amza, *Adaptive Learning of Metric Correlations for Temperature-Aware Database Provisioning*, The 4th International Conference on Autonomic Computing (ICAC), 2007, Florida, USA

[27] Dara Kusic, Jeffrey Kephart, James Hanson, Nagarajan Kandasamy and Guofei Jiang, *Power and Performance Management of Virtualized Computing Environments via Lookahead Control*, The 5th International Conference on Autonomic Computing (ICAC), 2008, Illinois, USA

[28] Jordi Torres, David Carrera, Vicenç Beltran, Nicolás Poggi, Kevin Hogan, Josep Berral, Ricard Gavaldà, Eduard Ayguadé, Toni Moreno and Jordi Guitart, *Tailoring resources: the energy efficient consolidation strategy goes beyond virtualization,* The 5th International Conference on Autonomic Computing (ICAC), 2008, Illinois, USA

[29] Hui Zhang, Guofei Jiang, Kenji Yoshihira, Haifeng Chen, and Akhilesh Saxena, *Resilient Workload Manager: Taming Bursty Workload of Scaling Internet Applications*, The 6th

International Conference on Autonomic Computing (ICAC), 2009, Barcelona, Spain

[30] Mahendra Kutare, Greg Eisenhauer, Chengwei Wang, Karsten Schwan, Vanish Talwar and Matthew Wolf, *Monalytics: Online Monitoring and Analytics for Managing Large Scale Data Centers*, The 7th International Conference on Autonomic Computing (ICAC), 2010, Washington, DC, USA

[31] Chengwei Wang, Karsten Schwan, Vanish Talwar, Greg Eisenhauer, Liting Hu and Matthew Wolf, *A Flexible Architecture Integrating Monitoring and Analytics for Managing Large-Scale Data Centers*, The 8th International Conference on Autonomic Computing (ICAC), 2011, Karlsruhe, Germany

[32] Ramon Nou and Jordi Torres, Heterogeneous *QoS Resource Manager with Prediction*, The 5th International Conference on Autonomic and Autonomous Systems (ICAS), 2009, Karlsruhe, Germany

[33] Vinod Ramachandran, Manish Gupta, Manish Sethi, and Soudip Chowdhury, *Determining Configuration Parameter Dependencies via Analysis of Configuration Data from Multi-tiered Enterprise Applications*, The 6th International Conference on Autonomic Computing (ICAC), 2009, Barcelona, Spain

[34] Dominic Jones, John Keeney, David Lewis, and Declan O’Sullivan, *Knowledge Delivery Mechanism for Autonomic Overlay Network Management*, The 6th International Conference on Autonomic Computing (ICAC), 2009, Barcelona, Spain

[35] Jeffrey Kephart, Hoi Chan, Rajarshi Das and David Levine, *Coordinating multiple autonomic managers to achieve specified power-performance tradeoffs*, The 4th International Conference on Autonomic Computing (ICAC), 2007, Florida, USA

[36] Richard Anthony, Mariusz Pelc and Haffiz Shauib, The Interoperability Challenge for Autonomic Computing, The 3rd International Conference on Emerging Network Intelligence (EMERGING), 2011, Lisbon, Portugal

[37] Shinji Kikuchi, Satoshi Tsuchiya, Motomitsu Adachi, and Tsuneo Katsuyama, *Constraint Verification for Concurrent System Management Workflows Sharing Resources*, The 3rd International Conference on Autonomic and Autonomous Systems (ICAS), 2007, Athens, Greece

[38] Hoi Chan, Alla Segal, Bill Arnold and Ian Whalley, *How Can We Trust an Autonomic System to Make the Best Decision?* The 2nd International Conference on Autonomic Computing (ICAC), 2005, Seattle, USA

[39] Shinji Kikuchi, Satoshi Tsuchiya, Motomitsu Adachi and Tsuneo Katsuyama, *Policy Verification and Validation Framework Based on Model Checking Approach*, The 4th International Conference on Autonomic Computing (ICAC), 2007, Florida, USA

[40] Jin Heo and Tarek Abdelzaher, *AdaptGuard: Guarding Adaptive Systems from Instability*, The 6th International Conference on Autonomic Computing (ICAC), 2009, Barcelona, Spain

[41] Raphael Bahati, Michael Bauer and Elvis Vieira, *Adaptation Strategies in Policy-driven Autonomic Management*, The 3rd International Conference on Autonomic and Autonomous Systems (ICAS), 2007, Athens, Greece

[42] Julien Perez, Cecile Germain-Renaud, Balazs Kegl and Charles Loomis, *Utility-based Reinforcement Learning for Reactive Grids*, The 5th International Conference on Autonomic Computing (ICAC), 2008, Illinois, USA

[43] Rajarshi Das, Jeffrey Kephart, Ian Whalley and Paul Vytas, *Towards Commercialization of Utility-based Resource Allocation*, The 3rd International Conference on Autonomic Computing (ICAC), 2006, Dublin, Ireland

[44] Jing Xu, Ming Zhao, José Fortes, Robert Carpenter and Mazin Yousif, *On the Use of Fuzzy Modeling in Virtualized Data Center Management*, The 4th International Conference on Autonomic Computing (ICAC), 2007, Florida, USA

[45] Thaddeus Eze, Richard Anthony, Chris Walshaw, and Alan Soper, *A Technique for Measuring the Level of Autonomicity of Self-managing Systems*, In review, The 8th International Conference on Autonomic and Autonomous Systems (ICAS), 2012, Maarten, Netherlands Antilles

[46] Horn Paul, *Autonomic computing: IBM perspective on the state of information technology*, IBM T.J. Watson Labs, NY, 15th October 2001. Presented at AGENDA 2001, Scottsdale.

[47] Jeffrey Kephart and D. M. Chess, *The vision of autonomic computing*, In IEEE Computer, volume 36, pp 41–50, January 2003

A Deliberative Reasoner for Model-Based Software Health Management

Abhishek Dubey, Nagabhushan Mahadevan, Gabor Karsai
Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN 37212, USA
{*dabhishe, nag, gabor*}@*isis.vanderbilt.edu*

***Abstract*—While traditional design-time and off-line approaches to testing and verification contribute significantly to improving and ensuring high dependability of software, they may not cover all possible fault scenarios that a system could encounter at runtime. Thus, runtime 'health management' of complex embedded software systems is needed to improve their dependability. Our approach to Software Health Management uses concepts from the field of 'Systems Health Management': detection, diagnosis and mitigation. In earlier work we had shown how to use a reactive mitigation strategy specified using a timed state machine model for system health manager. This paper describes the algorithm and key concepts for an alternative approach to system mitigation using a deliberative strategy, which relies on a function-allocation model to identify alternative component-assembly configurations that can restore the functions needed for the goals of the system. An example is used to show how such an approach can be used for performing automatic system reconfigurations, when faulty components are diagnosed.**

***Keywords-Component-based systems*; *fault diagnosis*; *autonomic computing*; *fault removal*.**

I. Introduction

Self-adaptive software systems, while in operation, must be able to adapt to latent faults in their implementation, in the computing and non-computing hardware; even if they appear simultaneously. Software Health Management (SHM) extends classical software fault tolerance techniques [1], [2], [3] by applying anomaly detection, fault source identification (diagnosis), fault effect mitigation (in operation), maintenance (offline), and fault prognostics (online or offline), as used in System Health Management of complex engineering systems [4], [5]. It is performed at run-time, and it includes fault detection, fault source isolation, and mitigation activities to remove fault effects. A good motivation for software health management is provided in [6].

We have developed an approach and model-based tools for implementing software health management functions for component-based systems. The foundation of the architecture is a real-time component framework (built upon an ARINC-653 platform) that defines a specific model of computation for software components [7]. This framework brings the concept of temporal isolation, spatial isolation, strict deadlines from ARINC-653 and combines them with the well-defined component interaction patterns described in CORBA Component Model [8]. Health management in the framework is performed at two levels. The Component-Level Health Manager (CLHM) provides localized and limited service for managing the health of individual components.

Higher-level management is provided by a System Health Manager (SLHM) that manages the health of the overall system. SLHM includes a diagnosis engine that uses a Timed Failure Propagation (TFPG) [9] model of the software that is automatically synthesized from the model of the software component assembly. The diagnostic engine reason about cascading fault effect in the system and isolates the fault source components. This is possible because the data and behavioral dependencies (and hence the fault propagation) across the assembly of software components can be deduced from the well-defined and restricted set of interaction patterns supported by the framework [10]. In the past, we showed how system-wide mitigation can be performed based on reactive timed state machines specified by the designer at the system integration time [11]. However, one of the problems with this approach to system-mitigation is the complexity of the specification required to cover all possible combination of failure scenarios.

This paper describes our work on system-level mitigation using a deliberative search-based technique that relies on the models of system goals/functionalities and the component groups allocated to provide these functionalities. Our approach is based on:

- Maintaining a model of the desired system functions and their sub-functions that all are required to provide that function.
- Maintaining an allocation tree for each function where the function is the root, the configuration groups (AND, OR, M of N) are the intermediate nodes and the software components are the leaf nodes. This tree captures the multi component configurations that are required to provide the service listed as the root function node.
- Identifying the current operational system goals.
- Identifying the affected operational goals based on the list of faulty components.
- Searching for alternative configuration that can satisfy the functions, while shutting down faulty components.

The outline of this paper is as follows: first we cover the related research. Then we present a short overview of our architecture and earlier results. Next, we discuss the deliberative reasoner, followed by a case study and conclusions.

II. Related Research

Our approach focuses on latent faults in software systems, it follows a component-based architecture with a model-based development process, and implements all steps in the Collect/Analyze/Decide/Act loop [12].

Rohr et al. advocate the use of architectural models for self-management [13]. They suggest the use of a run-time model to reflect the system state and provide reconfiguration functionality. From a development model they generate a causal graph over various possible states of its architectural entities. Garlan et al. [14] and Dashofy et al. [15] have proposed an approach which bases system adaptation on architectural models representing the system as a composition of several components, their interconnections, and properties of interest. They make reconfiguration decisions using rule-based strategies.

While these works have tended to the structural part of the self-managing computing components, some have emphasized the need for behavioral modeling of the components. For example, Zhang et al. described an approach to specify the behavior of adaptable programs in [16]. Their approach is based on separating the adaptation behavior specification from the non-adaptive behavior specification in autonomic computing software. Williams' research [17] concentrates on model-based autonomy. The paper suggests that emphasis should be on developing techniques to enable the software to recognize that it has failed and to recover from the failure. Their technique lies in the use of a Reactive Model-based Programming Language (RMPL)[18] for specifying both correct and faulty behavior of the software components. They also use high-level control programs [19] for guiding the system to the desirable behaviors.

Work described here is related to the larger field of software fault tolerance: principles, methods, techniques, and tools that ensure that a system can survive software defects that manifest themselves at run-time [20], [21].

III. The ARINC Component Framework

System-level health management and fault tolerance approaches are based on the notion of interacting components. Our work is based upon the ARINC-653 component model (ACM) [7]. ACM combines the CORBA Component Model [8] with ARINC-653 [22]. ACM components interact with each other via well-defined patterns, facilitated by ports. In ACM, a component can have four kinds of ports for interactions: **publishers, consumers, facets** (a.k.a. provided interfaces - where an interface is a collection of related methods) and **receptacles** (a.k.a. required interfaces). The component can interact with other components through **synchronous** call/return interfaces (associated with facets or receptacles), and/or via **asynchronous** publish/subscribe event connections (between publisher and consumer). A component can also have internal methods that are periodically triggered. Systems are designed as composition of components using a modeling environment, which includes a domain specific modeling language and associated tools.

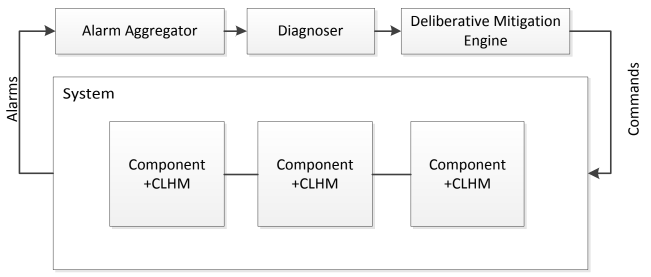


Figure 1. SLHM architecture.

Component Execution and Failure Scenarios: A software component can be in one of the following three states: **active**, **inactive** and **semi-active**. When a component is in the inactive state, none of the ports of the component are operational. The active state of a component is the exact opposite of the inactive, and all the component ports are operational. In a semi-active state, only the consumer and receptacle ports of a component are operational, the publisher and provided ports are disabled. While the component is executing , i.e., it is in the active or semi-active state, the code in the component ports can introduce faults in the system, which can lead to anomalies in either the same component or in a connected component. We consider two root failure sources for each component port (a) a concurrency fault that is indicated by a timeout event in the act of obtaining the lock associated with the component, (b) or a latent bug in the code written by the developer to implement the operation associated with the port. Every component has a lock to ensure that at any given time at most one thread is active in the component.

Example: Figure 2 shows the assembly for a notional GPS system with a redundant set of Sensor and GPS components (Sensor2, GPS2). Deployment information is not shown in this figure. Sensors publish an event every 4 sec for their associated GPS. The GPS consumes the event published by its sensor at a periodic rate of 4 sec. Afterwards, it publishes an event, which is sporadically consumed by the Navigation Display. Thereafter, the display component updates its location by using getGPSData facet of the GPS Component. In the initial setup of the assembly, the Sensor, GPS, and NavDisplay components are used and hence set to be in *active mode*. The redundant sensor and GPS (Sensor2 , GPS2) are not used. The GPS2 is set to a *semi-active mode*, leaving the Sensor2 component in active mode. This would allow the GPS2 to keep track of the current state (by being in semi-active mode where the GPS2's consumers are active) but not affect NavDisplay.

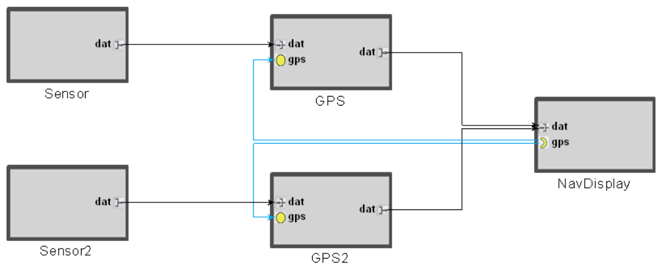


Figure 2. GPS Software Assembly.

A. Health Management in ACM

Health Management in ACM happens at two levels. The first level of protection is provided by a component level health management (CLHM) strategy, which is implemented in all components. It provides a localized timed state machine with state transitions triggered either by a local anomaly or by timeouts, and actions that perform the local mitigation. The System Level Health Manager (SLHM) is at the second, top level in our health management strategy. The deployment of the SLHM requires the addition of three special SLHM components to an ACM assembly: the *Alarm Aggregator*, The *Diagnosis Engine*, and the *Deliberative Mitigation Engine*, as shown in Figure 1.

The *Alarm Aggregator* is responsible for collecting and aggregating inputs from the component level health managers (local alarms and the corresponding mitigation actions). This information is collected using a moving window two hyperperiods long. The events are sorted based on their time of occurrence and then sent to the *Diagnosis Engine*.

The *Diagnosis Engine* hosts an instance of a Timed Failure Propagation Graph reasoner engine. This engine is initialized by a Timed Failure Propagation Graph (TFPG) [9] model that captures the failure-modes, discrepancies (possibly indicated by the alarms), and the failure propagations from failure modes to discrepancies and from discrepancies to other discrepancies, across the entire system [10], [11]. The reasoner uses this model to isolate the most plausible fault source: a software component that could explain the observations, i.e., the alarms triggered and the CLHM commands issued. The result, i.e., the list of faulty components is reported to the next component that provides the system level mitigation: the *Deliberative Engine*, discussed in the next section.

IV. Deliberative Engine

The mitigation engine in a system has to map the diagnosis results to a set of actions that remove the faults in the system and restore the functionality. There are four basic commands that can be sent to each component (a) RESET : Instructs a component to Reset itself, (b) STOP : Instructs a component to switch to inactive mode, (c) START: Instructs a component to switch to active mode, and (d) REWIRE

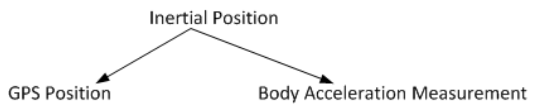


Figure 3. Example of Functional Decomposition for an Inertial Measurement Unit

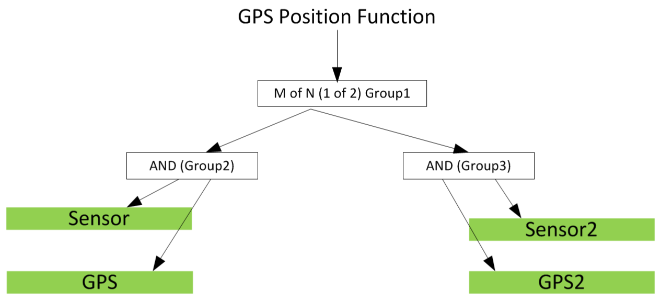


Figure 4. Example showing allocation of the GPS position function shown in Figure 3 to the components shown in assembly of Figure 1.

(ri,pc): Instructs a Component to rewire its receptacle Interface (ri) to connect to the appropriate facet interface in another component (pc). In case of REWIRE, the appropriate facet to be used is identified by the component which stores a map of component to facet for every receptacle in the component.

Our initial approach towards fault mitigation in SLHM included a *Reactive Mitigation Engine* wherein the mitigation strategy was specified as a hand-crafted, timed state machine model at design time. The updated fault status of the components in the assembly was used to trigger the SLHM state machine. For details on this mitigation specification, please see [10], [11].

The new SLHM mitigation approach uses a *Deliberative Mitigation Engine* which, like the reactive mitigation engine receives the diagnosis results: the set of faulty components, and responds with an appropriate system-level command to mitigate the fault and its effects. The deliberative mitigation approach relies on modeling the system goals as functions, the functional dependency on other sub-functions and the function-allocation model, i.e., the component group configurations that can provide the function (or sub-function). At run-time, the deliberative engine searches through the space of the function-allocation model to identify an alternate configuration of healthy components that can restore the functions (functionalities) affected by the faulty components.

In the timed-state machine approach, the modeler needs to specify the specific mitigation strategy for each fault (component) and/ or fault - combination (set of faulty components). We realized that the state-machine based approach (of modeling fault mitigation actions) is very tedious, error-prone and gets extremely complicated as the number of components and their fault combinations grow. In the current SLHM Mitigation strategy using Deliberative Reasoner, the reasoner builds a graph of the function allocation model

Table I
$IsUsable$ SEMANTICS

Type	Defintion
Component	$isUsable(c) \Leftrightarrow \neg isFaulty(c)$
AND-Group	$isUsable(g) \Leftrightarrow (\forall x \in child(g))(isUsable(x))$
XOR-Group	$isUsable(g) \Leftrightarrow (\exists x \in child(g))(isUsable(x))$
MofN-Group	$isUsable(g) \Leftrightarrow (\exists X \subseteq child(g))(\lvert X\rvert \geq M)$ $(\forall x \in X)(isUsable(x))$
Function	$isUsable(f) \Leftrightarrow (\forall x \in child(g))(isUsable(x))$

Table II
$isActive$ SEMANTICS.

Type	Defintion
Component	$isActive(c)$ is marked by the deployment scheme and any previous action of the reasoner
AND-Group	$isActive(g) \Leftrightarrow (\forall x \in child(g))(isActive(x))$
XOR-Group	$isActive(g) \Leftrightarrow (\exists x \in child(g))(isActive(x))(\forall y \in child(g))(y \neq x)(\neg isActive(y))$
MofN-Group	$isActive(g) \Leftrightarrow (\exists X \subseteq child(g))(\lvert X\rvert \geq M)(\forall x \in X)(isUsable(x))(\forall y \in child(g)/X)(\neg isUsable(y))$
Function	$isActive(f) \Leftrightarrow (\forall x \in child(g))(isActive(x))$

and the assembly model and searches this graph for an appropriate mitigation action to restore the functionality. The deliberative reasoning approach using the function allocation model allows a better modeling scalability for faults and fault combinations.

Modeling the Functional Decomposition of System: During the design time, the system integrator enumerates the system functions as a collection of simple AND trees. That is, if $\mathbb{F}$ is the set of all immediate children of a function node, f_p, in the functional decomposition tree, then $isActive(f_p) = (\forall f \in \mathbb{F})(isActive(f))$.

Example Model: Figure 3 shows the functional decomposition of portions of an Inertial Measurement Unit. The Inertial Position function requires the GPSPosition function and the BodyAccelerationMeasurement function. In the running system one or more such function trees can be active. Additionally, a lower level function may be required in multiple trees.

Modeling the Functional Allocation: A function at any level of the functional decomposition directed acyclic graph can depend on other child functions and can depend upon the availability of a set of components at that level. The set of components related to a function can be hierarchically organized into groups. There are three kinds of groups: (a)**AND** of some components (all), (b) **XOR** of some components (exactly one of N), and (c) **MofN** of some components (at least M out of N components.). Note that both XOR and MofN groups are used to model redundancy.

Once specified, the functional allocation tree has the **function at the root, groups as intermediate nodes and components as the leaf nodes**. Components have two attributes in this tree: $isFaulty$ and $isActive$. While $isFaulty$ is determined based on the diagnoser output, $isActive$ is determined by the initial configuration. The deliberative reasoning process could result in marking a component (healthy or faulty) to be inactive , i.e., setting $isActive = false$. This results in sending a STOP command to the component. When a component is not faulty it is considered to be usable , i.e., $isUsable(c) \implies \neg isFaulty(c)$.

Procedure 1 Driver - RunDR

```
1: CLEAR LIST GRC, GRN.
2: for component c ∈ DR do
3:     MarkAsFaulty(c)
4: end for
5: RunReconfig();
6: return GRC; {set of possible reconfig commands}
```

Usable attribute for the groups can be set based on the immediate child groups and child components. An AND group is usable if and only if all its children are usable. A XOR group is usable if any one of the children is usable. A MofN group is usable if at least M children are usable. These rules are summarized in Table I. Note in the table g means group, c means component. Operator $parent(x)$ returns the set of all immediate parents of x in the function allocation Directed Acyclic Graph (DAG). Operator $child(x)$ returns the set of immediate children of x, and $|.|$ is the cardinality operator.

Similarly $isActive(c)$ can be evaluated from leaf to the root of the function allocation tree. A root function in this tree is usable if all its immediate groups are usable. It is active if all its immediate groups are active. Table II summarizes all the rules. Note that due to the maximal nature of the $isActive(c)$ definition for MofN group, any reconfiguration action that requires turning a MofN group active requires to turn all its usable children active.

Example Function Allocation Model: Figure 4 shows the allocation diagram for one of the functions in figure components shown in Figure 3 using the components in the assembly depicted in Figure 2. The model indicates that the GPS Function can be provided by an M of N (1 of 2) Group. It requires the services of at least one of the two AND Groups (Group2 or Group3). The AND groups in turn require the services of all of their child nodes (here components).

A. Search and Reconfiguration Algorithms

During run-time, the deliberative engine is seeded with the functional allocation model translated into a XML form and the initial configuration of the system components. Internally, it maintains three lists: (a) Global List GFC: set of components that have been diagnosed as faulty, (b) Global List GRC: set of possible reconfiguration commands, and (c) Global List GRN: set of possible reconfiguration nodes.

Furthermore, the deliberative engine assigns an index to each node in the functional allocation graphs. All leaves are assigned index 0. Any other node has a level Number

Procedure 2 Mark as faulty

```
Input: Faulty Component c
Given: RN is an empty set.
 1: if c ∈ GFC then
 2:     return
 3: end if
 4: c.isFaulty=true
 5: c.isUsable=false
 6: GFC.add(c)
 7: RN = visitParent(c)
 8: if isempty(RN) then
 9:     output-command = RESET(c)
10: else
11:     output-command=STOP(c)
12:     GRN.add(RN)
13: end if
14: GRC.add(output-command)
```

that is Max(Level-Index of its children) +1. This Level-Index is used to sort the elements in GRN (the Possible-Reconfig Nodes) based on the graph topology. During the reconfiguration search, the elements in GRN are explored for reconfiguration in the increasing order of Level-Index , i.e., the reconfigurable nodes closest to the source is explored first.

Each time the deliberative reasoner is invoked, it receives an input list DR of components diagnosed as faulty. It invokes the steps detailed in Procedure 1. The deliberative reasoner uses the Procedure 2 to mark the faulty component in the functional allocation graphs. This algorithm does nothing if the component is already faulty. Otherwise, it marks it as faulty and invokes the visitParent Procedure 3. If the visitParent Procedure returns a possible reconfiguration node, then it records a command to STOP the faulty component. Otherwise, it records a command to RESET the faulty component. The reconfiguration node is added to the GRN, the command is added to the GRC.

The visitParent Procedure 3 is used to visit a parent node of the current node in the function allocation graph. It evaluates the IsUsable property for each parent node. If a parent node is still usable, then it returns the node as reconfiguration node. Otherwise, it recursively invokes VisitParent on the parent node. Note that each group has only one parent group or one or more function parent node.

Once the reconfiguration node, i.e., the node in the allocation tree where the change has to take place is identified, the Run Reconfig Procedure 4 is invoked to compute the reconfiguration that would restore the functionality. This algorithm loops through each node that is stored in the GRNlist. It invokes the Reconfig Procedure 5 on each node, which returns true if an alternative exists, else it returns false. It also invokes the ReconfigStop Procedure 6 on those child nodes that need to be stopped as they are no longer usable. Components that are marked as active but do not belong to any active parent group are commanded to be stopped. As a last step, it checks if any of the receptacles need to be

Procedure 3 visitParent

```
Input: Node N
Output: Set of Reconfig Node RN
 1: $\mathbb{P} = parent(N)$
 2: for $p \in \mathbb{P}$ do
 3:     if isUsable(p) then
 4:         RN.add(p) {add a usable node to possible reconfig nodes}

 5:         return RN
 6:     else
 7:         if $p \in GRN$ then
 8:             GRN.remove(p)
 9:         end if
10:         return visitParent(p)
11:     end if
12: end for
13: return 0
```

Procedure 4 RunReconfig

```
 1: for n ∈ GRN do
 2:     Result= Reconfig(n)
 3:     if Result then
 4:         $CN = child(N)$ {set of children}
 5:         for ch∈ CN do
 6:             if $\neg isUsable(ch) \wedge isActive(ch)$ then
 7:                 ReconfigStop(ch)
 8:             end if
 9:         end for
10:     end if
11: end for
12: Check for Rewiring
```

rewired to a facet on a newly activated provider component. This step of rewiring is required if any component servicing a facet has been stopped in the current reconfiguration.

B. Example Reconfiguration

Consider the assembly captured in Figure 2. Initially Sensor, GPS, NavDisplay components are active. Sensor2 is also active. But, GPS2 is semi-active. Thus, GPS2 consumes data from the Sensor2 but does not publish data to NavDisplay. At this time, the Global List of Fault Candidates GFC is initialized as an empty list. The deliberative engine records the initial states of the component and identifies if the currently active functionality shown in fig 4 is satisfied or not.

The Deliberative Engine is invoked if there is any fault diagnosis reported by the Diagnosis Engine component. Consider that GPS component is reported faulty. This will lead to the invocation of the MarkAsFaulty Procedure 2, causing GPS to be set as faulty and unusable. When the VisitParent Procedure 3 is invoked, the parent group of GPS (And Group2) will be marked as unusable because it requires all children to be usable. A recursive call to the same Procedure will identify that the MofN Group 1 is still usable because at least 1 of the 2 AND groups, Group 3 is still usable. At the end of these two Procedures, Group1 will be added to the GRN and a command to stop the GPS

Procedure 5 Reconfig

```
Input: Node N
1: if $isUsable(N)$ then
2:   if N.type() ==COMPONENT $\neg isActive(N)$ then
3:     N.isActive= true
4:     Output-Command = START(N)
5:     GRC.add(Output-Command)
6:   end if
7:   if N.type() ==MofNGROUP then
8:     CN=child(N)
9:     for x $\in$ CN do
10:      Reconfig(x)
11:    end for
12:    N.isActive= true
13:  end if
14:  if N.type() ==ANDGROUP then
15:    if $isUsable(N)$ then
16:      CN=child(N)
17:      for x $\in$ CN do
18:        Reconfig(x)
19:      end for
20:      N.isActive= true
21:    end if
22:    if N.type() ==XORGROUP then
23:      CN=child(N)
24:      for x $\in$ CN do
25:        if isUsable(x) then
26:          Reconfig(x)
27:          for y $\in$ CN and $y \neq x$ do
28:            if isActive (y) then
29:              ReconfigStop(y)
30:            end if
31:          end for
32:          N.isActive= true
33:          return {It will return as soon as the first is usable child is found}
34:        end if
35:      end for
36:    end if
37:  end if
38:  return
39: end if
```

Procedure 6 ReconfigStop

```
Input: Node N
1: if N.type() $\neq$ COMPONENT then
2:   N.isActive=false
3:   CN=child(N)
4:   for x $\in$ CN do
5:     ReconfigStop(x)
6:   end for
7: end if
8: if N.type() == COMPONENT then
9:   PN=parent(N)
10:  deactivate= true
11:  for x $\in$ PN do
12:    if isActive(x) then
13:      deactivate = false
14:      BREAK
15:    end if
16:  end for
17:  if deactivate then
18:    output-command=STOP(N) {Deactivate a component, when none of its parents are active}
19:    GRC.add(output-command)
20:  end if
21: end if
```

component will be added to the GRC.

Once the reconfigurable nodes are identified, RunReconfig Procedure 4 will be invoked to identify the exact reconfiguration commands to restore the functionality. The Reconfig Procedure 5 will be performed on Group 1 which is of type MofN. This will result in iterative invocation of the Reconfig Procedure 5 on Group2 and Group3. While nothing will happen in the context of Group2 as it is no longer usable, Reconfig step will be invoked on Group3's children: Sensor2 and GPS2. Commands to START GPS2 component will be added to the Global Reconfig Command (GRC) list. The RunReconfig Procedure 4 will invoke ReconfigStop Procedure 6 on the other AND group (Group2) that will result in the GRC list being updated with a stop command for the Sensor component. An additional check will be performed to see if any receptacle ports need to be rewired. This results in the rewire of the receptacle in NavDisplay to use the facet in GPS2. Note the details of this check have not been included in the paper due to space constraints.

C. Limitation

The algorithm described above suffers from a limitation exposed by the XOR group. The XOR group dictates that one and only one component associated with that group is active at any time. This condition associated with the XOR group could be violated in the algorithm described above. If one or more components appeared under an XOR group as well as in other branches of the function allocation model, the algorithm does not ensure that the XOR conditions are honored. Currently, we impose a restriction that a component featured under an XOR group may not feature elsewhere in the function allocation model.

V. Conclusion

This paper discussed our approach towards restoring the health of a software system based on identifying the alternative component configurations using the function-allocation model for the system. We described the design language for modeling the function allocation, the algorithm employed to update the usable branches of the function allocation model based on the fault report from the diagnosis engine and identify suitable component reconfigurations that can restore the function. An example was described to illustrate the function allocation design and the algorithm. In order to relax the restrictions associated with function allocation model, we are exploring other approaches such as using a SAT solver to identify alternate configurations.

Acknowledgement

This paper is based upon work supported by NASA under award NNX08AY49A. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Aeronautics and Space Administration. Authors would like to thank Dr Paul Miner, Eric Cooper, and Suzette Person of NASA LaRC for their help and guidance on the project.

References

[1] Michael R. Lyu. *Software Fault Tolerance*, volume New York, NY, USA. John Wiley & Sons, Inc, 1995.

[2] Wilfredo Torres-Pomales. Software fault tolerance: A tutorial. Technical report, NASA, 2000. Available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.8307.

[3] R.W. Butler. A primer on architectural level fault tolerance. Technical report, NASA Scientific and Technical Information (STI) Program Office, Report No. NASA/TM-2008-215108, 2008. Available at http://shemesh.larc.nasa.gov/fm/papers/Butler-TM-2008-215108-Primer-FT.pdf.

[4] S. Ofsthun. Integrated vehicle health management for aerospace platforms. *Instrumentation Measurement Magazine, IEEE*, 5(3):21 – 24, September 2002.

[5] S.B. Johnson, T. Gormley, S. Kessler, C. Mott, A. Patterson-Hine, K. Reichard, and P. Scandura Jr. *System Health Management: With Aerospace Applications*. John Wiley & Sons, Inc, 2011.

[6] Ashok Srivastava and Johann Schumann. The Case for Software Health Management. In *Fourth IEEE International Conference on Space Mission Challenges for Information Technology, 2011. SMC-IT 2011.*, pages 3–9, August 2011.

[7] Abhishek Dubey, Gabor Karsai, and Nagabhushan Mahadevan. A component model for hard real-time systems: CCM with ARINC-653. *Software: Practice and Experience*, 41(12):1517–1550, 2011.

[8] Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan. Overview of the CORBA component model. *Component-based software engineering: putting the pieces together*, pages 557–571, 2001.

[9] S. Abdelwahed, G. Karsai, N. Mahadevan, and S. C. Ofsthun. Practical considerations in systems diagnosis using timed failure propagation graph models. *Instrumentation and Measurement, IEEE Transactions on*, 58(2):240–247, February 2009.

[10] Abhishek Dubey, Gabor Karsai, and Nagabhushan Mahadevan. Model-based Software Health Management for Real-Time Systems. In *Aerospace Conference, 2011 IEEE*, pages 1–18. IEEE, 2011.

[11] Nagabhushan Mahadevan, Abhishek Dubey, and Gabor Karsai. Application of software health management techniques. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 1–10, New York, NY, USA, 2011. ACM.

[12] Betty H Cheng. Software engineering for self-adaptive systems. chapter Software Engineering for Self-Adaptive Systems: A Research Roadmap, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2009.

[13] Matthias Rohr, Marko Boskovic, Simon Giesecke, and Wilhelm Hasselbring. *Models in Software Engineering, Workshops, and Symposia at MoDELS 2006*, volume 4364, chapter Model-driven Development of Self-managing Software Systems. 2006.

[14] David Garlan, Shang-Wen Cheng, and Bradley Schmerl. Architecting dependable systems. chapter Increasing system dependability through architecture-based self-repair, pages 61–89. Springer-Verlag, Berlin, Heidelberg, 2003.

[15] Eric M. Dashofy, Andre van der Hoek, and Richard N. Taylor. Towards architecture-based self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 21–26, New York, NY, USA, 2002. ACM Press.

[16] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 371–380, New York, NY, USA, 2006. ACM.

[17] Paul Robertson and Brian Williams. Automatic recovery from software failure. *Commun. ACM*, 49(3):41–47, 2006.

[18] B.C. Williams, B.C. Williams, M.D. Ingham, S.H. Chung, and P.H. Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. *Proceedings of the IEEE*, 91(1):212–237, 2003.

[19] Brian C. Williams, Michel Ingham, Seung Chung, Paul Elliott, Michael Hofbaur, and Gregory T. Sullivan. Model-based programming of fault-aware systems. *AI Magazine*, 24(4):61–75, 2004.

[20] Michael R. Lyu. Software reliability engineering: A roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 153–170, Washington, DC, USA, 2007. IEEE Computer Society.

[21] Laura L. Pullum. *Software fault tolerance techniques and implementation*. Artech House, Inc., Norwood, MA, USA, 2001.

[22] ARINC specification 653-2: Avionics application software standard interface part 1 - required services.

Efficient Alignment of Aerial Images Based on Virtual Forces

Claudius Stern, Christoph Rasche, Lisa Kleinjohann and Bernd Kleinjohann
Faculty of Computer Science, Electrical Engineering and Mathematics,
Department of Computer Science, C-LAB,
University of Paderborn, Germany
e-mail: {claudis, crasche, lisa, bernd}@c-lab.de

***Abstract*—Getting a contemporary aerial overview of a disaster area is a foremost task in search and rescue operations. We previously have introduced a novel method for registering a large amount of aerial images when camera parameters are almost unknown and no reference images are available. This paper provides two new methods, which improve our method for image registration based on virtual forces. The goal is to improve the performance of creating a contemporary overview map of a disaster area assembling several images taken by unmanned aerial vehicles (UAVs) equipped with cameras. In this paper, two new methods are introduced: a method for rotation estimation and a method for scale estimation. Both methods use fast heuristic approximation approaches and statistical methods to provide high robustness. We discuss the methods in detail and compare them to our previous approach regarding robustness and calculation speed. We can show that the new methods significantly increase the performance of the image registration process.**

***Keywords*-Virtual forces, image registration, UAV, map building**

I. Introduction

For large-scale disasters—natural or human made—getting a contemporary overview of the disaster area is an important task in order to find victims and to plan the rescue actions. Often an area is affected, which is too large to efficiently get an overview at ground level. Classically, a manned aircraft or helicopter is used to gather aerial pictures of the whole scene. Nevertheless only in few cases the pictures are used to build a map. In the cases where a helicopter equipped with video cameras has been assigned an observation task, the video is seen only by the observer located in the helicopter. Typically, the video is not analyzed automatically and sometimes not even recorded. Slow changes of the environment like, for instance, the imminence of a flood may be recognized too late. Also typically, there is only one helicopter over the area due to the costs of such a mission. Hence, there is no possibility to get up-to-date images of more than one place at the same time. Furthermore, manned helicopters, which are not directly involved in rescue missions are often not allowed to fly over persons at low altitudes.

In recent years, Unmanned Aerial Vehicles (UAVs) have increasingly become a more viable choice for such situations. The research project SOGRO (German: "Sofortrettung bei Großunfall mit Massenanfall von Verletzten", English: "Immediate rescue in a large-scale accident with mass casualties") also makes use of UAVs for exploring a terrain where a large-scale accident or a natural disaster has caused many casualties.

In this paper, we present the improvement of an image registration approach developed in the course of the SOGRO project. In [1], we tackle the challenge to register sequentially arriving images to a consistent map of a disaster area. To achieve this, aerial images are delivered by a swarm of UAVs, which spread out over the area [2], [3]. This enables us to deliver up-to-date images of different parts of the disaster area at the same time.

Each UAV produces many images, which are partially overlapping. The images contain distinctive feature points, which can be extracted automatically. To improve the old-fashioned observer approach, we store the images in a database, analyze them, e.g., by extracting distinctive features, and using them to create a contemporary overview map of the disaster area. The extracted features are also stored in the database. They are used to compare different images to each other such that corresponding parts of images can be determined.

Corresponding feature points (key points) are connected with virtual forces. Our image registration approach introduced in [1] translates, rotates and scales the images such that the forces' lengths becomes minimal. We will describe that approach shortly in Section II.

Two new methods are presented in this paper, which increase the performance of our previous approach, supporting the ability using low-cost UAVs to automatically create a contemporary overview map of an area. We introduce a new fast heuristic rotation estimation method and a new fast heuristic scale estimation method. Both methods are compared to the former ones in terms of calculation speed and resulting scale factors.

The paper is organized as follows. In the next section, we will point out the problems of classical image registration and introduce the main idea of using virtual forces for image registration. In Section III, we will give an overview of key point extraction methods and compare our work with regard to other approaches. Section IV describes our new approach for rotation estimation in detail and Section V explains the new approach for scale estimation. In Section VI, we show some results of the evaluation regarding the use of the new methods compared to the old ones. In Section VII, we conclude the paper and give an outlook on future research.

II. Virtual Forces for Image Registration

In [1], we have described the first steps towards using virtual forces for image registration. For better understanding, we motivate the use of virtual forces here again.

In order to build a contemporary map of a disaster area only using civil UAVs equipped with relatively inexpensive camera equipment, we encountered a challenge. Using classical approaches did not lead to satisfying results. We used SIFT [4] as key point extractor, later also SURF [5]. After a descriptor matching, RANSAC [6] was applied to filter the matches. This approach is well-known and broadly used, e.g., for satellite images. Assuming only rare information about flight attitude (describing, e.g., the position, the nick-angle, the roll-angle, and the yaw-angle) and the inavailability of reference imagery, only few images could be stitched due to accumulating perspective errors. This behavior already has been described by Brown and Lowe when successively concatenating homographies [7].

The challenge in our scenario is to be able to create a contemporary map of a disaster area without having flight attitude information or any reference imagery. In field tests, we also evaluated the reliability of off-the-shelf GPS receivers and decided not to rely on them. The GPS signal delivered by our receiver had a relatively high tolerance and even disappeared intermittently. So, we decided not to rely on any measured reference for the first steps. Our approach does not take any measured reference into account at this time. In future research, different measurements shall be included to increase the stability even more or to increase the achievable resolution.

Classical approaches typically use a static mapping function, local or global. In the global case, all images are needed in advance to calculate a consistent mapping function. In the local case, errors would propagate if the first image's parameters are erroneous. Using virtual forces for image registration addresses with both problems, mapping images in a flexible way and balancing the errors.

Images are regarded to be masses connected by forces that could be imagined to work like rubber bands or springs, which tend to have a length of zero. A virtual force has a start point, an end point, a length and a direction. The force's strength is equivalent to its length. Like in the classical image registration process, key points (distinctive image features) are extracted from the images. These key points are compared and matched against each other. Then each matching pair is fitted with a new virtual force, pulling the newest image to the previous ones, which it partially overlaps (otherwise no matchings would have been found). Here connections between the last image and its direct predecessor are established as well as connections to all other images, which it overlaps by a certain amount. Afterwards, an iterative process moves, rotates and scales the masses until an equilibrium is established, namely, the sum of all remaining forces is minimal. Assuming geometrically consistent matches, corresponding key points then will be next to each other, ideally with zero distance.

III. Related Work

To find matching interest points (features) is essential for image registration. Several interest point detectors were introduced in the last decades. Such detectors are usually chosen by means of requirements of an application. There are simple feature detectors for edges and/or corners, e.g., like the Harris corner detector [8], which are easy to compute. As these detectors only indicate the presence of a feature, they are not directly usable for image registration. When using a group of features to match against another group of features, taking the geometrical structure of the groups into account, these kind of features are nevertheless usable for image registration. Especially when regarding video sequences, these features or, e.g., the features Shi and Tomasi introduced in [9] are trackable in subsequent video frames assuming some kinematic restrictions.

More general, for image registration purposes features are needed, which can be compared in terms of similarity. Two well-known feature detectors of this kind are the Scale-invariant feature transform (SIFT) [4] and Speeded Up Robust Features (SURF) [10], [5].

As described by Zitová and Flusser in [11], most image registration techniques use a kind of keypoints, which are compared and matched against each other. The matched points are then considered to represent the same location in the scene, which was captured. According to Zitová and Flusser, image registration is the process of overlaying two or more images of the same scene taken at different times, from different viewpoints, and/or by different sensors. A comprehensive survey of the image registration process and an overview of solution methods used by current approaches for different processing steps is given in [11]. They also provide a classification of different approaches and according to them the application of this work belongs to their classes *multiview analysis* and *multitemporal analysis*. The images used in our scenario can be delivered by several UAVs at different positions over the area, and also the same part of the area can be recorded repetitively as the intention is to provide a contemporary overview map.

IV. Fast Heuristic Rotation Estimation

In this section, we introduce the novel rotation estimation method and compare it to the former approach.

We assume two masses that are connected by some virtual forces. The former method presented in [1] was mainly inspired by physical application of forces to a mass. Besides an acceleration, a torque-like quantity was calculated and used for rotation. In the very first implementation, the center of gravity (CoG) of the mass itself was used as center of rotation. This had been enhanced so that the CoG of the forces' attraction points was used. Hence, the former method needed many iterations to stabilize.

The new method also uses the CoG of the key points in two images as center of rotation. Two center points are calculated: one for the start points in the first image and one for the end points in the second image. A CoG point is calculated for both masses, using the start points CoG_{start} and the end points CoG_{end} of the forces respectively.

$$CoG_{start} = \frac{\sum_{F_u} \mathrm{start}(f_i)}{\|F_u\|} \qquad CoG_{end} = \frac{\sum_{F_u} \mathrm{end}(f_i)}{\|F_u\|}$$

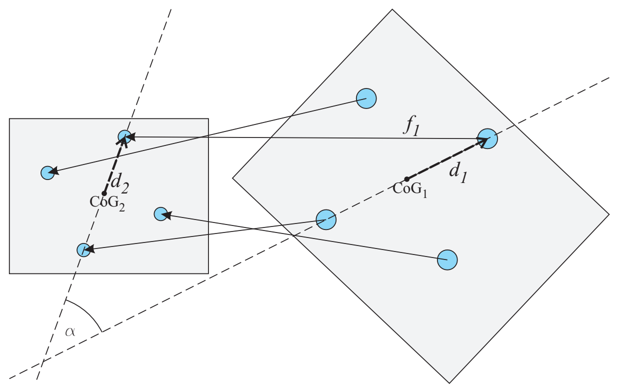


Fig. 1. Simple example for rotation calculation. The rotation angle α is derived by intersecting the lines going through the CoG points and a matching pair of key points. In this example, only one matching pair is used for demonstration.

Forces can be marked as ineffective, so $F_u \subset F$ denotes the forces, which are actually used from the set of totally available forces F, $f_i \in F$ denotes one force with a start point $\mathrm{start}(f_i)$ and an end point $\mathrm{end}(f_i)$. Both, start point and end point are given in local coordinates with respect to the according mass object.

For each force two lines are built: a first line through the CoG point of the start points and the start point of the force, and a second line through the CoG point of the end points and the end point of the force. These two lines are intersected and the angle between them is calculated. This angle represents the rotation angle, which is necessary to rotate the end-point-mass such that it is equally oriented like the start-point-mass. Fig. 1 shows a simple example for the rotation calculation described above. For demonstration only one force (f_1); is considered.

$$\begin{aligned}\overrightarrow{d_1} &= \overrightarrow{\mathrm{start}(f_1)} - \overrightarrow{CoG_1}\\ \overrightarrow{d_2} &= \overrightarrow{\mathrm{end}(f_1)} - \overrightarrow{CoG_2}\\ \alpha &= \arccos \frac{\overrightarrow{d_1} \times \overrightarrow{d_2}}{\|\overrightarrow{d_1}\|\|\overrightarrow{d_2}\|}\end{aligned}$$

With only one force considered, only two distance vectors are considered ($\overrightarrow{d_1}$, $\overrightarrow{d_2}$). Here, $\overrightarrow{d_2}$ denotes the distance vector from the CoG point of the start points to the actual start point of f_1, and $\overrightarrow{d_1}$ denotes the distance vector from the CoG point of the end points to the actual end point of f_1. The angle α can then be derived by calculating the arc cosine of the cross product of the two vectors divided by their length.

In general, the mean angle $\bar{\alpha}$ between the matching pairs' directional vectors is used:

$$\begin{aligned}\overrightarrow{d_i^{start}} &= \overrightarrow{\mathrm{start}(f_i)} - \overrightarrow{CoG_{start}}\\ \overrightarrow{d_i^{end}} &= \overrightarrow{\mathrm{end}(f_i)} - \overrightarrow{CoG_{end}}\\ \alpha_i &= \arccos \frac{\overrightarrow{d_i^{start}} \times \overrightarrow{d_i^{end}}}{\|\overrightarrow{d_i^{start}}\|\|\overrightarrow{d_i^{end}}\|}\\ \bar{\alpha} &= \frac{\sum_{F_u} \alpha_i}{\|F_u\|}\end{aligned}$$

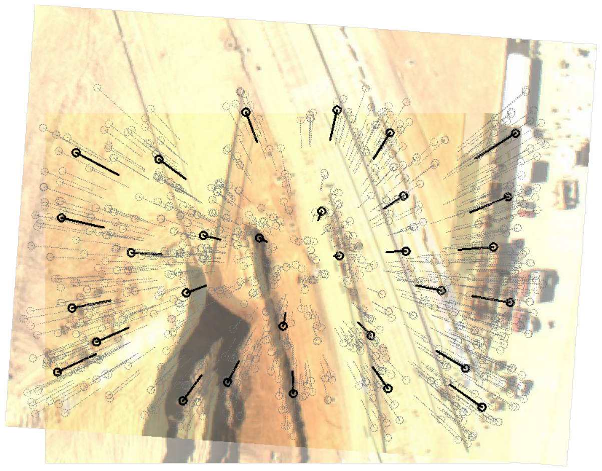


Fig. 2. Old scale method. Two previously aligned images with different scales are shown. The virtual forces, which connect the images, form a star shape. These forces act like gravitational forces and constrict the bigger image.

V. Fast Heuristic Scale Estimation

In this section, we introduce a novel scale estimation method. The new method needs less computation time for scale changes and the algorithm itself is much more numerically stable than the former one. In non-controlled environments the UAV faces ascending or descending air currents and has to level them out. During that leveling, scale changes appear, as well as, when the territory rises or falls; whereas the altitude of the UAV remains stable. Hence, scale changes will occur and have to be tackled. Following is a brief description of the former method.

The former method needed previously adjusted images. Then the forces formed a star shape and the forces were applied like gravitational forces. Fig. 2 shows two images connected by forces, which were previously adjusted. The forces also have been filtered and only the solid black ones are effective. Nevertheless, that method required scaling iterations interleaved with adjustments to stabilize: The forces' vectors are treated as lines and are randomly selected for intersection. The mean of the intersection points is used as the center point for scaling. Due to the randomness of the intersection, the center point moves and after a few steps a readjustment of the mass is necessary. In particular, the calculation of the intersection points is numerically unstable due to the shrinking vector lengths when the matching points come close to each other. Nevertheless, that approach is robust due to the use of statistical methods to overcome this numerical instability. However, this procedure needed many calculation steps in iterations when it came to scale changes, e.g., due to an altitude change of the UAV. To cope with this behavior, a new method has been created.

The new method also uses a statistical approach to derive the center of the scale operation. In contrast to the former method, the CoG of the key points is used instead of the intersection

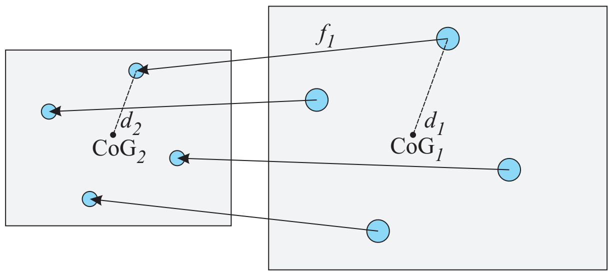


Fig. 3. Simple example for scale calculation. The scale factor for one iteration to scale the right mass to the left one is calculated as $1.001^{\|d_2\|-\|d_1\|}$. In this example, only one distance is used for demonstration.

```
scaleWithCoG()
  $CoG_{start}$ = calculateCenter(startPoints($F_u$))
  $CoG_{end}$ = calculateCenter(endPoints($F_u$))
  foreach $f_i \in F_u$ do:
    start_diff = $\|\text{start}(f_i) - CoG_{start}\|$
    end_diff = $\|\text{end}(f_i) - CoG_{end}\|$
    mean_diff = mean_diff + (end_diff - start_diff)
  mean_diff = mean_diff / $\|F_u\|$
  mean_scale = pow(1.001, mean_diff)
```

Listing 1. Basic algorithm to scale the last mass according to the virtual forces. The mean scale is an exponential function of the mean difference of the distances from the start points to the start point center and the distances from the end points to the end point center.

points. This operation is independent of the actual position of the two masses. Like in the previous section, a CoG point is calculated for both masses. CoG_{start} is calculated using the start points and CoG_{end} is calculated using the end points of the forces.

Fig. 3 shows a simple example for scale calculation. Two masses are shown, both with key points and connected by forces. On both masses the CoG points are marked. For demonstration, only one distance pair is depicted. Typically all distance pairs are taken into account and the arithmetic mean is used as scale factor. Listing 1 shows the basic algorithm to calculate the scale of two masses. Each force connected to both images has a start point and an end point. The start points are located on one image, the end points on the other. For both point sets a center of gravity is calculated. Then the mean distance of each start point to the start point center is set in relation to the mean distance of each end point to the end point center. The resulting ratio is used as scale factor.

$$\begin{aligned}
\textit{start diff}_i &= \|\text{start}(f_i) - CoG_{start}\| \\
\textit{end diff}_i &= \|\text{end}(f_i) - CoG_{end}\| \\
\textit{mean diff} &= \frac{\sum_{F_u} \textit{end diff}_i - \textit{start diff}_i}{\|F_u\|} \\
\textit{mean scale} &= 1.001^{\textit{mean diff}}
\end{aligned}$$

A more sophisticated variant has been implemented to support multiple masses as potential force sources. There, at first, the forces are grouped according to their source mass. For each group of forces the basic algorithm is used to calculate a scale factor. Then a mean scale factor is calculated from the individual scale factors of each group.

As mentioned before, the scale center is of great importance, too. If two masses were aligned before, the scale operation should not move the mass. In the former approach, the scale center was a bit volatile, so the pure scale operations had to be interleaved by adjusting operations. The new approach improves this behavior. The scale center is calculated as the center of gravity of all start points and end points respectively. Assuming aligned masses and geometrically consistent matches (i.e., the matching points on the images represent the same area in reality), the two centers for start points and end points would lie above each other. Scaling around this point

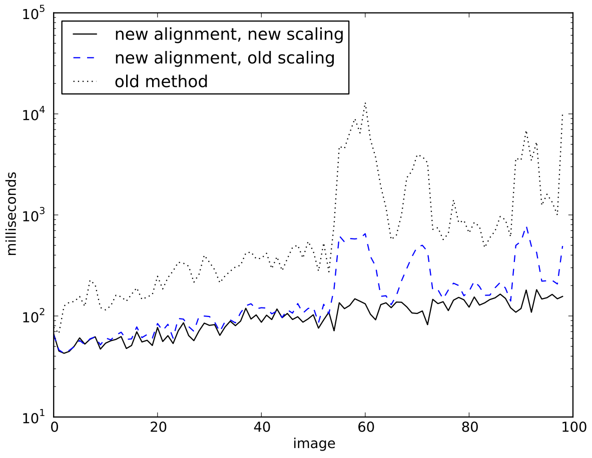


Fig. 4. Comparison of the computation times between old and new method. The total computation times including alignment and scaling are shown. The values are the mean of ten independent runs on identical image sets. Note the logarithmic scale for computation time.

assures that the scale operation does not move the center of gravity in terms of global coordinates.

Summarized, the new method has a faster convergence rate, is independent from position and rotation of the mass to be scaled and is numerically stable.

VI. Results

Using the new method for rotation estimation, we have achieved a significant speed-up of the registering performance. In Fig. 4, the computation times of the old method [1] and the new method are compared to each other. The total computation times including alignment and scaling are shown. The values are given in milliseconds and are calculated as the geometric mean of ten independent runs on identical image sets of 100 images. As it can be seen, the new method outperforms the old one and provides a much more stable runtime behavior. Also the impact of the new scaling method can be seen here. The middle curve depicts the runtime of the new rotation estimation method combined with the old scaling method.

For this evaluation, images taken the bottom-view camera of a “Parrot AR.Drone” [12] are used. The camera lacks a stabilization so it is a good test to the robustness of our

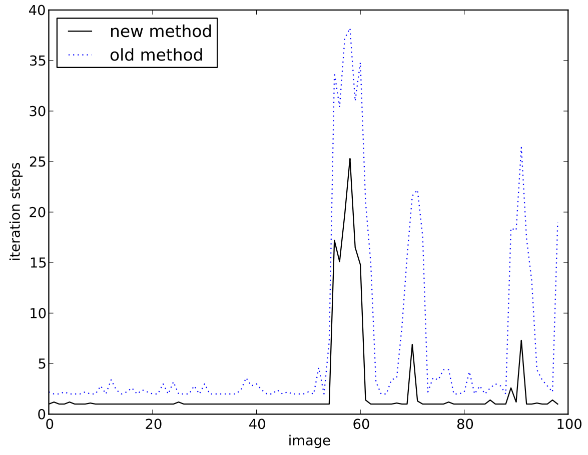


Fig. 5. Comparison of the iteration steps needed to stabilize between old and new method. The values are the mean of ten independent runs on identical image sets. A scale change starts at image 55. Here, the iteration steps of the old method increase significantly. The new method better adapts to scale changes due to its faster convergence.

approach against perspective distortions. Nevertheless, some problems are to be expected at increased camera tilt levels, when the perspective distortion becomes too large. Here a false scale change is to be expected.

Fig. 5 shows the performance of the new scale estimation method compared to the former one in terms of iteration steps. In the first 54 images, the performance of both approaches is nearly equal. A scale change starts at image 55 and immediately the number of iteration steps of the old method increases significantly over the number of iteration steps of the new method. Most of the time, the new method only needs one iteration step and while the scale changes, it significantly outperforms the old method.

In Fig. 6, a related comparison of the computation times and the scale factor is shown. The underlying set of images was taken with the aforementioned ”Parrot AR.Drone”'s bottom-view camera. The image set was taken at approximately the same height above ground for all images. Here the effect of large perspective distortion occurs, which causes a false reaction of the heuristic scale estimation. Nevertheless, it can be seen that the new method of scale estimation is much less sensitive to such distortions. The resulting scale factor is as expected: flat lines in phases of plain flight and a limited scale change during phases with increased camera angles. These occur as the “Parrot AR.Drone” is a Quadrotor, and it has to change its flight attitude in order to fly in a given direction.

At last we evaluated different implementations of the new scaling approach. We changed the scale estimation function to a function, which should be able to calculate the scale difference between two images in one single step. For this purpose we used the mean quotient of the distances of the start points to their center and the distances of the end points to their center. In the aforementioned example showed in Fig. 3,

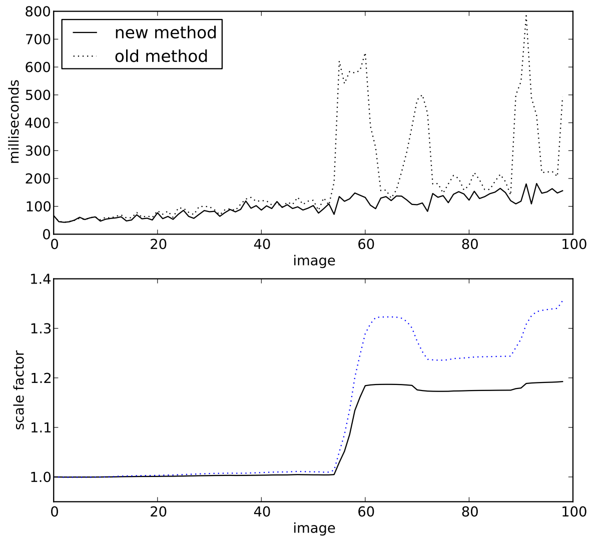


Fig. 6. Comparison of the computation times between old and new method related to the scale factor. The values are the mean of ten independent runs on identical image sets. A scale change starts at image 55. The scale changes are false positives due to perspective distortion. The new method's reaction is much less than that of the old method.

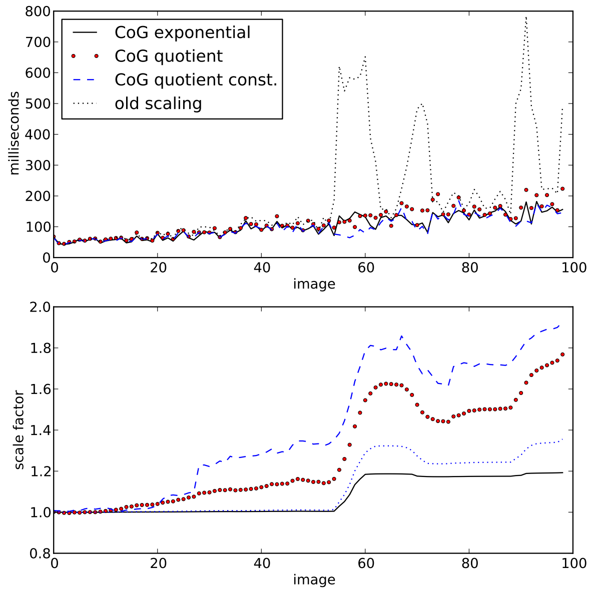


Fig. 7. Comparison of the computation times and the resulting scale factor of the old method with different versions of the new scale estimation method.

the scale factor would be calculated as d_1/d_2 to scale the right image to the left one. Fig. 7 shows the calculation times and the resulting scale factors of four different scale estimation implementations. All implementations, which use a variant of the new scale estimation, are using the CoG points of the start points and the end points respectively. The chart lines

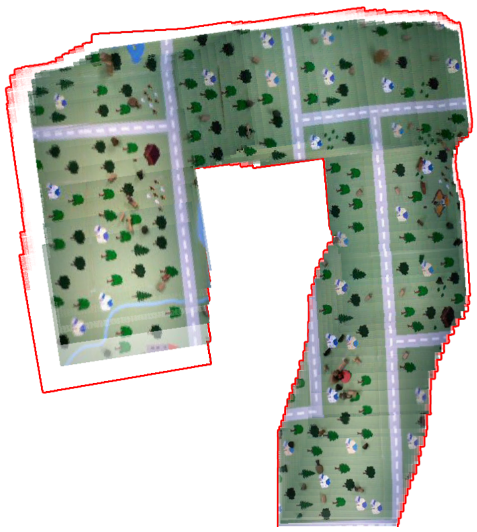


Fig. 8. Comparison between new and old scaling method. Shown are the registered images using the new scaling method. The underlying contour marks the extent of the registered images, if the old scaling method would have been used.

are named correspondingly. The scaling method using the direct quotient method (CoG quotient const.) needs constant computation time but is very sensitive to any distortions and reacts with false scale changes. Executing that method iteratively (CoG quotient) increases its scaling performance but yet the quotient method is worse than the old scaling method. Using the exponential function for scale estimation in combination with the CoG points (CoG exponential) reaches our goal and outperforms the old scale method in terms of calculation time as well as in the resulting scale factor.

Fig. 8 shows the registered images using the new methods. Underneath the images a contour is drawn. It marks the extent of the images, if the old scaling method would have been used. The contour, starting very close to the images, soon deviates in scale. That deviation starts to grow, when the “Parrot AR.Drone” moves sidewards due to the assymetric horizontal and vertical flare angles.

VII. Conclusion & Outlook

We presented new methods for rotation estimation as well as for scale estimation. The fast heuristic rotation estimation significantly increases the overall performance of the map building process and provides a smoother runtime behavior than its predecessor.

Different implementations of the new approach for the fast heuristic scale estimation were compared to the old scale estimation method and against each other in terms of numerical stability, calculation time and resulting scale factor calculation. The best one—based on an exponential function of the mean difference of two vector lengths—provides a scale estimation method independent from position and rotation of the object to be scaled. It outperforms the old method in all tests: the calculation is numerically stable, the calculation is faster and is less dependent on scale changes, and the calculated scale factor is less sensitive to perspective distortions.

The next step will be the integration of perspective projection into the mapping as well as the usage of sporadic reference information. On the one hand, this will allow the building of geo-referenced maps and, on the other hand, we can make use of the sporadic reference information, e.g., to increase the resolution of the map. Another future work will be the evaluation of different kinds of features in order to increase the mapping speed even more.

Acknowledgments

This contribution was developed in the course of the SOGRO project funded by the BMBF (German ministry of education and research) under grant number 13N10164.

References

[1] C. Stern, C. Rasche, L. Kleinjohann, and B. Kleinjohann, “Towards using virtual forces for image registration,” in *The 5th International Conference on Automation, Robotics and Applications (ICARA 2011)*, Wellington, New Zealand, Dec. 2011.

[2] C. Rasche, C. Stern, W. Richert, L. Kleinjohann, and B. Kleinjohann, “Combining autonomous exploration, goal-oriented coordination and task allocation in multi-uav scenarios,” in *Autonomic and Autonomous Systems (ICAS), 2010 Sixth International Conference on*, Mar. 2010, pp. 52 –57, accessed 25-January-2012. [Online]. Available: http://dx.doi.org/10.1109/ICAS.2010.16

[3] C. Rasche, C. Stern, L. Kleinjohann, and B. Kleinjohann, “Coordinated exploration and goal-oriented path planning using multiple uavs,” in *International Journal on Advances in Software*, vol. 3, no. 3&4. IARA, 2010, pp. 351–370.

[4] D. Lowe, “Object recognition from local scale-invariant features,” in *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, vol. 2, 1999, pp. 1150 –1157 vol.2, accessed 25-January-2012. [Online]. Available: http://dx.doi.org/10.1109/ICCV.1999.790410

[5] H. Bay, A. Ess, T. Tuytelaars, and L. V. Gool, “Speeded-up robust features (surf),” *Computer Vision and Image Understanding*, vol. 110, no. 3, pp. 346 – 359, 2008, similarity Matching in Computer Vision and Multimedia. [Online]. Available: http://www.sciencedirect.com/science/article/B6WCX-4RC2S4T-2/2/c2c03b6165996e30312e5b7c7b681155

[6] M. A. Fischler and R. C. Bolles, “Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography,” *Commun. ACM*, vol. 24, pp. 381–395, June 1981, DOI: 10.1145/358669.358692.

[7] M. Brown and D. Lowe, “Recognising panoramas,” in *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, Oct. 2003, pp. 1218 –1225 vol.2, accessed 25-January-2012. [Online]. Available: http://dx.doi.org/10.1109/ICCV.2003.1238630

[8] C. Harris and M. Stephens, “A combined corner and edge detector,” in *Fourth Alvey Vision Conference*, 1988, pp. pp. 147–151, accessed 25-January-2012. [Online]. Available: http://www.assembla.com/spaces/robotics/documents/abzMnAOEer3zB7ab7jnrAJ/download/harris88.pdf

[9] J. Shi and C. Tomasi, “Good features to track,” in *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR ’94., 1994 IEEE Computer Society Conference on*, jun 1994, pp. 593 –600, accessed 25-January-2012. [Online]. Available: http://dx.doi.org/10.1109/CVPR.1994.323794

[10] H. Bay, T. Tuytelaars, and L. Van Gool, “Surf: Speeded up robust features,” in *ECCV 2006*, ser. Lecture Notes in Computer Science, A. Leonardis, H. Bischof, and A. Pinz, Eds. Springer Berlin / Heidelberg, 2006, vol. 3951, pp. 404–417, accessed 25-January-2012. [Online]. Available: http://dx.doi.org/10.1007/11744023_32

[11] B. Zitová and J. Flusser, “Image registration methods: a survey,” *Image and Vision Computing*, vol. 21, no. 11, pp. 977 – 1000, 2003, accessed 25-January-2012. [Online]. Available: http://dx.doi.org/10.1016/S0262-8856(03)00137-9

[12] “Parrot AR.Drone,” 2012, accessed 25-January-2012. [Online]. Available: http://ardrone.parrot.com

Coordinating Energy-aware Administration Loops Using Discrete Control

Soguy Mak-Karé Gueye
LIG / UJF
Grenoble, France
soguy-mak-kare.gueye@inria.fr

Noël De Palma
LIG / UJF
Grenoble, France
noel.de_palma@inria.fr

Eric Rutten
LIG / INRIA
Grenoble, France
eric.rutten@inria.fr

***Abstract*—The increasing complexity of computer systems has led to the automation of administration functions, in the form of autonomic managers. One important aspect requiring such management is the issue of energy consumption of computing systems, in the perspective of green computing. As these managers address each a specific aspect, there is a need for using several managers to cover all the domains of administration. However, coordinating them is necessary for proper and effective global administration. Such coordination is a problem of synchronization and logical control of administration operations that can be applied by autonomous managers on the managed system at a given time in response to events observed on the state of this system. We therefore propose to investigate the use of reactive models with events and states, and discrete control techniques to solve this problem. In this paper, we illustrate this approach by integrating a controller obtained by synchronous programming, based on Discrete Controller Synthesis, in an autonomic system administration infrastructure. The role of this controller is to orchestrate the execution of reconfiguration operations of all administration policies to satisfy properties of logical consistency. We apply this approach to coordinate energy-aware managers for self-optimization and self-regulation of processor frequency.**

***Keywords*-autonomic computing, coordination of multiple autonomic managers, modeling, synchronous programming, discrete controller synthesis.**

I. Introduction

A. Green computing and the need for administration loops

The increasing complexity of computer systems, integrating several distributed components operating in a heterogeneous and dynamic environment, had led to a problem of hand administration to be time-consuming, expensive, and error-prone. In response to this problem, many research works contribute to the automation of administration functions, in the form of autonomic managers.

One important aspect requiring such management is the issue of energy consumption of computing systems, in the perspective of green computing. Its dynamic management is based on the fact that the deployment and configuration of systems can modified in response to changes in workload, infrastructure and resource availability, or power supply. A variety of mechanisms can be designed for power-aware administration, using the autonomic loop framework. For example, they can contribute at the level of processor frequency, or at the level of server provisioning.

When multiple loops run concurrently, their interactions have to be managed themselves, in order to avoid side-effects annihilating the management actions. Our work focuses on this problem, and proposes a solution for the coordination and synchronization of administration managers, seen themselves as manageable elements.

B. Autonomic administration loops

Autonomic computing [9] aims at providing self-management capabilities to systems. As shown in Figure 1, the managed system or resource is monitored through sensors, and an analysis of this information is used, in combination with knowledge about the system, to plan and execute reconfigurations, through the administration actions offered by the system API.

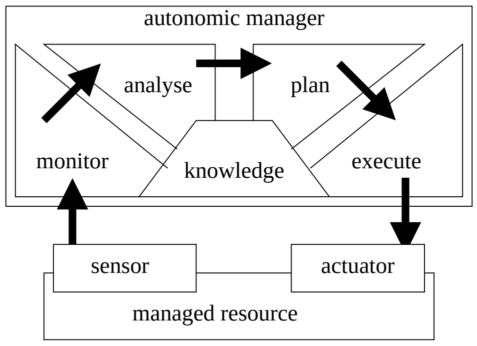


Figure 1. Architecture of an autonomic system

Typical self-management issues handled in this framework are self-configuration, self-optimization, self-healing (fault tolerance and repair), and self-protection. They are managed in closed loop, for which one design methodology is to apply techniques from control theory, continuous or discrete.

C. The problem of coordinating administration loops

Classically, an autonomic manager focuses on one specific concern of system administration. Often, several autonomic managers must be used concurrently to cover all the administration domain. However, using multiple autonomic managers is not enough for ensuring a correct and efficient global system administration. The administration policy followed by each autonomic manager does not take into account the objectives of others aspects: this can of course lead to inconsistencies. In order to benefit from the re-use of several existing autonomic managers, one has to care for coordinating their executions, according to global criteria and properties of their assembly. Most of the proposed solutions for coordinating autonomic managers are based on software infrastructures, which are in charge of ensuring a global view of the managed system for all managers and synchronizing managers' operations.

However, coordination is a problem of synchronization and logical control of administration operations that can be applied by autonomic managers on the managed system at a given time in response to events observed on the state of this system. Therefore, its solution requires the use of models with events and states, where properties on the order of events or the mutual exclusion of parallel states can be addressed. Such models are at the basis of reactive or synchronous programming languages, and their compilation and analysis tools, as well as discrete control techniques.

D. Our proposed approach

Our approach is to consider the coordination as a synchronization management problem, and to design an additional layer, as shown in Figure 2, above the individual administration loops, which constitutes a coordination controller. This relies upon access to information about local controllers, such as their current state or execution mode, their controllable features (e.g., suspendability), and relevant events. We will build this hierarchical controller using models of reactive systems, which are automata-based, and Discrete Controller Synthesis to generate automatically the correct coordination constraint, so that logical coherence invariants are enforced.

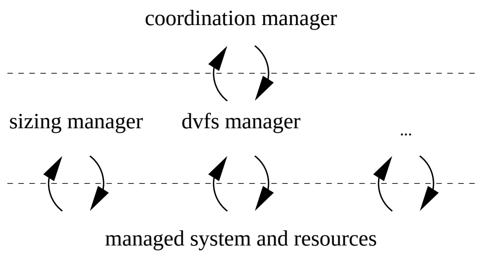


Figure 2. Coordination architecture of multiple loops

In this paper, we apply this approach to the case of the coordination of energy-aware controllers, which manage respectively Sizing (server provisioning) and Dvfs (processor frequency).

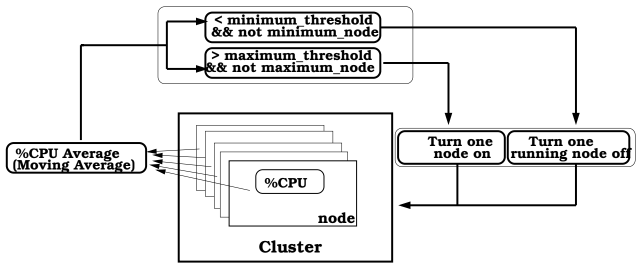

< minimum_threshold && not minimum_node
> maximum_threshold && not maximum_node
%CPU Average (Moving Average)
%CPU
node
Cluster
Turn one node on
Turn one running node off

Figure 3. Optimization controller

The rest of the paper is organized as follow. In section II, we present two energy-aware autonomic managers. In section III, we present the tools used in our approach for designing an efficient and correct coordination controller for autonomic managers. In section IV, we present the design of the coordination controller for the managers presented in Section II with our approach. In section V, we present a simulation of the generated coordination controller. Section VI presents the integration of the generated controller into a real system. In section VII, we discuss background and related work. Finally, in section VIII, we conclude the paper and outline directions for future work.

II. Uncoordinated control loops

We present two controllers dealing with energy optimization and performance of a system. They are developed independently. They try to optimize the energy consumption of a system while preserving a good performance. They are based on performance thresholds that describe an optimal performance region where the system must be depending on its workload.

A. Optimization controller: Sizing

This controller is for replicated servers based on a load balancer scheme. Its role is to dynamically adapt the degree of replication according to the system load. It dynamically turns cluster nodes on when the load of the system cannot be handled by resources it uses before the overload. When the system is underloaded, it turns cluster nodes off to save power under lighter load.

Figure 3 shows the execution scheme of the optimization controller. The controller analyzes the nodes CPU usage to detect if the system load is in the optimal performance region. It computes a moving average of collected load monitored by sensors. When the controller receives a notification from sensors, if the average exceeds the maximum threshold and the maximum number of replication (max node) is not reached, it increases the degree of replication by selecting one of the unused nodes. If the average is under the minimum threshold and the minimum number of replication is not reached, it decreases replication by turning a node off.

B. CPU-frequency controller: Dvfs

This controller targets single node management. Its role is to dynamically adapt the CPU-frequency of a node according to the load this node receives. It dynamically increases or decreases the CPU-frequency depending on the load.

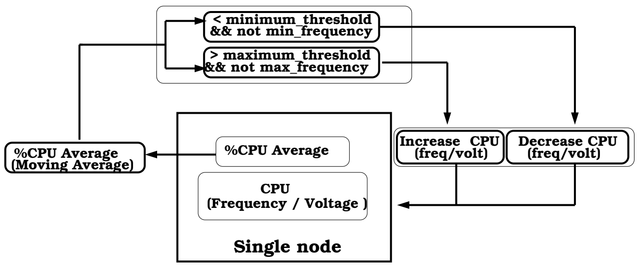

< minimum_threshold && not min_frequency
> maximum_threshold && not max_frequency
%CPU Average (Moving Average)
%CPU Average
CPU (Frequency / Voltage)
Single node
Increase CPU (freq/volt)
Decrease CPU (freq/volt)

Figure 4. CPU-frequency controller

Figure 4 shows the execution scheme of this controller. The controller analyzes the node CPU usage monitored by sensor. If the observed load exceeds the maximum threshold and the maximum CPU frequency is not reached, it increases the CPU frequency. If the load is under the minimum threshold and the minimum CPU frequency is not reached, it decreases the CPU frequency. This controller is local to the node it manages and is implemented either in hardware or software. The one we use is a user-space software and follows the on-demand policy.

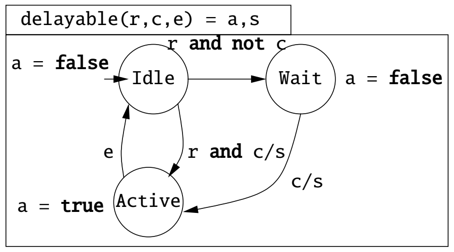
C. Uncoordinated execution

Here, we analyze the control of replicated servers composed of both controllers Sizing and Dvfs described above. Sizing deals with the whole system while Dvfs deals with each node separately.

When adding self-management capabilities to a system, one can use these two controllers to manage the energy consumption. The objective of using these two controllers together could be to optimize the energy consumption locally on each used node by acting on the CPU frequency and globally by managing the degree of replication. The objective is to optimize the energy consumption, without any coordination, however in some case this objective is not met. For example, when the system is overloaded, it is detected by Sizing and an upsizing operation is performed. But the system is overloaded means that some or all nodes that compose this system are overloaded, which implies CPU-frequency increase operation on nodes that are overloaded. If increasing the CPU frequency of these overloaded resources could be enough to restore the system performance to the optimal performance region, the upsizing operation become irrelevant, useless and leads to waste of energy since a new node is added while the previous nodes were able to handle the load received by the system after increasing the frequency of their CPU. There is a need to delay as long as possible upsizing operations when CPU-frequeny increase can be done.

III. Synchronous programming and discrete controller synthesis

For our contribution, we use the language BZR [3]. This language allows to describe reactive systems by means of generalized Moore machines, i.e., mixed synchronous dataflow equations and automata [11], with parallel and hierarchical composition. The basic behavior is that at each reaction step, values in the input flows are used in order to compute the values in the output flows for that step. Inside the nodes, this is expressed as a set of declarations, which takes the form of equations defining, for each output and local, the values that the flow takes, in terms of an expression on other flows, possibly using local flows and values computed in preceding steps (also known as state values).



```
node delayable(r,c,e:bool) returns (a,s:bool)
  let
    automaton
      state Idle
        do a = false ; s = r and c
        until r and c then Active
            | r and not c then Wait
      state Wait
        do a = false ; s = c
        until c then Active
      state Active
        do a = true ; s=false
        until e then Idle
     end
   tel
```

Figure 5. Delayable task in graphical and textual syntax.

Figure 5 shows a small program in this language. It programs the control of a task, which can either be idle or active. When it is idle, i.e., in the initial Idle state, then the occurrence of the input `r` *requests* the launch of the task. Another input `c` (which will be controlled further by the synthesized controller) can either allow the activation, or temporarily block the request and make the automaton go to a waiting state. When active, the task can end and go back to the idle state, upon the notification input `e`. This `delayable` node has two outputs, `a` representing activity of the task, and `s` being emitted on the instant when it becomes active : this latter is connected to the OS with the task starting operation.

The main feature of the BZR language is that its compilation involves *discrete controller synthesis* (DCS). DCS allows to compute automatically a controller, i.e., a function which will act on the initial program so as to enforce a given temporal property. Concretely, the BZR language allows the declaration of *controllable variables*, the value of which are not defined by the programmer. These free variables can be used in the program to describe choices between several transitions. These variables are then defined, in the final executable program, by the controller computed by DCS. DCS produces, when it exists, the maximally permissive constraint on the values of controllable variables, such that the resulting inhibited behavior satisfies the objective.

$\texttt{twotasks}(r_1, e_1, r_2, e_2) = a_1, s_1, a_2, s_2$

enforce not $(a_1$ **and** $a_2)$

with c_1, c_2

$$(a_1, s_1) = \texttt{delayable}(r_1, c_1, e_1)$$
$$(a_2, s_2) = \texttt{delayable}(r_2, c_2, e_2)$$

Figure 6. Mutual exclusion enforced by DCS in BZR.

Figure 6 shows an example of use of these controllable variables. This example consists in two instances of the `delayable` node, as defined in Figure 5. These instances run in parallel, defined by synchronous composition: one global step corresponds to one local step for every equation, i.e., here, for every instance of the automaton in the `delayable` node. Then, the `twotasks` node so defined is given a *contract* composed of two parts: the **`with`** part allowing the declaration of controllable variables (c_1 and c_2), and the **`enforce`** part allowing the programmer to assert the property to be enforced by DCS, using the controllable variables. Here, we want to ensure that the two tasks running in parallel will not be both active at the same time. Thus, c_1 and c_2 will be used by the computed controller to block some requests, leading automata of tasks to the wating state whenever the other task is active.

IV. MODEL-BASED COORDINATION

We propose a coordination solution, based on such reactive models, to avoid inconsistencies induced by these controllers running in parallel. This solution consists of designing a coordination controller on top of these controllers. This coordination controller is responsible of controlling the execution of Sizing and Dvfs in order to prevent any execution which may lead to inconsistencies.

The design of such a coordination controller is based on the synchronous approach. We use the synchronous programming to model the behavior of each controller. The models represent all the states in which they can be during their execution, with some control on transitions. The composition of these models describes the parallel execution

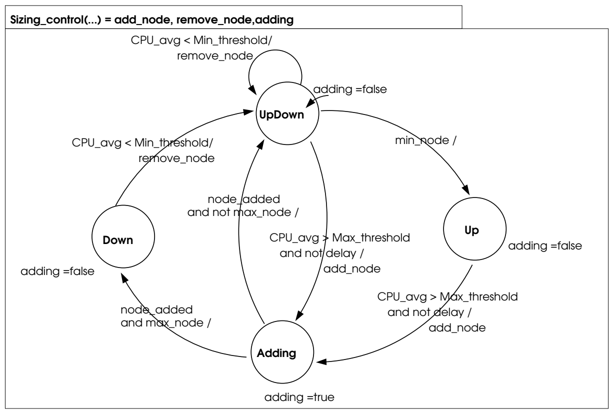


Figure 7. Optimization controller

of the controllers, which means both desired and undesired behaviors. We use discrete control synthesis techniques to automatically compute and generate the coordination controller based on the composition of the models and a coordination policy. The coordination policy is expressed as properties that should be enforced by the desired behaviors.

A. *Optimization controller model*

The model of this controller is composed of two automata.

Figure 7 represents the automaton for Sizing, where we add a control of the upsizing operations. The control is represented by the Boolean **delay**, upsizing operations are possible only when this variable is false. The outputs of this automaton are three Booleans, **add_node** and **remove_node** being signals allowing the controller to request the resuming or suspension of a node, **adding** being true whenever an adding operation is performed.

The initial state is **UpDown**, both upsizing and downsizing operations are possible. When the CPU average reaches the maximum threshold and the upsizing operations are allowed, the controller requests a new node, and goes to the **Adding** state, where it awaits for the new node to be actually available. In this **Adding** state, nodes can neither be added nor removed. When node_added occurs, the controller can either go back to **UpDown**, or if there is no more nodes able to be resumed, go to the **Down** state where only downsizing operations can be performed. This **Down** state is left once one node is suspended. The **Up** state is used when no node can be removed, i.e., when the minimum number of replication is reached. In the **Up** state, only the upsizing operations can be applied.

Figure 8 represents how the Sizing manager can be controlled, by inhibiting add_node operations in some global states. The output of this automaton is the Boolean **delay**, which enables upsizing operations when it is false. This output feeds the input **delay** in Figure 7. Initially, the automaton is in the state **Idle** where upsizing operations are delayed. When **c** is true, it goes to the state **Active** where upsizing operations are allowed. It stays there until **c** is false.

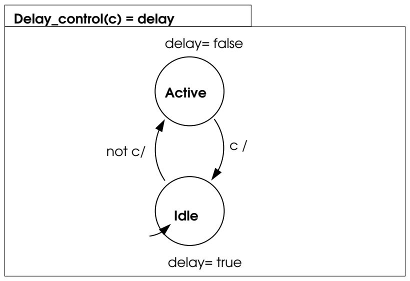


Figure 8. Upsizing operations control

B. CPU-frequency controller model

In our coordination problem, it is not necessary to control the execution of local dvfs controllers. We only need their current state in order to allow/deny upsizing operations. Therefore, a global observer is used for collecting information about current state of the set of Dvfs, each Dvfs provides two outputs, *min* being true when it can not decrease the CPU-frequency and *max* being true when it can not increase the CPU-frequency. This observer is a sensor that monitors the global state of set of local Dvfs. It has two outputs, one corresponding to the conjunction of all *min* outputs and the others to the conjunction all *max* outputs. Figure 9 represents the automaton for the observer. The outputs of this automaton are two booleans, **max_freq** being true when all local Dvfs reach the maximum frequency and **min_freq** being true when all local Dvfs reach the minimum frequency.

The inital state is **Normal** where **max_freq** and **min_freq** are false. In this state, at least one of the set of Dvfs can apply both CPU-frequency increase and CPU-frequency decrease operations. When all nodes are in their maximum CPU-frequency, the observer goes to the state **Max**. If all nodes are in their minimum CPU-frequency, the controller goes to the state **Min**. In the state **Max**, all Dvfs can only apply CPU-frequency decrease operations. In the state **Min**, they can only apply CPU-frequency increase operations.

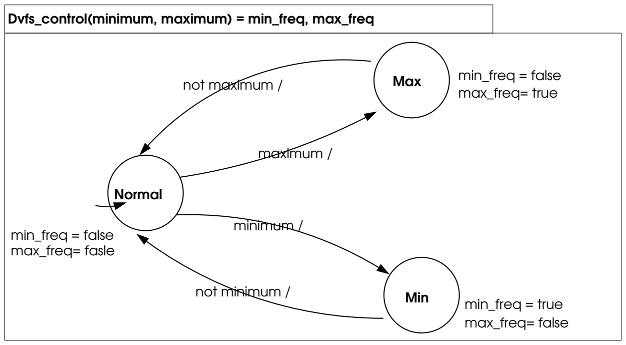


Figure 9. CPU-frequency controller

When the observer is in the state **Max** or the state **Min**, it stays there until at least one of the nodes is neither in its maximum CPU-frequency nor its minimum CPU-frequency.

C. Coordination policy

Finally, we present the coordination controller design.

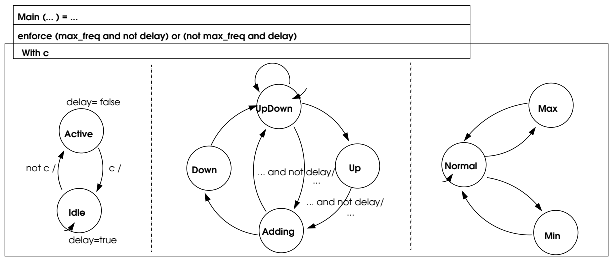


Figure 10. BZR program for coordination policy

Figure 10 is the coordination program built with the BZR language. The three automata presented before are composed in parallel, and a contract is added to enforce the coordination policy. Here, we want that the upsizing operations to be delayed when CPU-frequency increase operations can be performed. This coordination policy is stated by (max_freq **and not** delay) **or** (**not** max_freq **and** delay). The variable c is declared as a controllable.

V. Coordination controller simulation

The generated controller behavior can be simulated before its integration in the system with the SIM2CHRO chronogram tool (Verimag). It allows to test if the generated program reaction, represented by its outputs, respects the coordination policy expressed as logical invariant whatever the inputs are. Figure 11 represents a snapshot example of the complete simulation.

It shows a scenario illustrating the generated coordination controller in action. The input and output variables are Booleans. The input **minimum** notifies that all used nodes

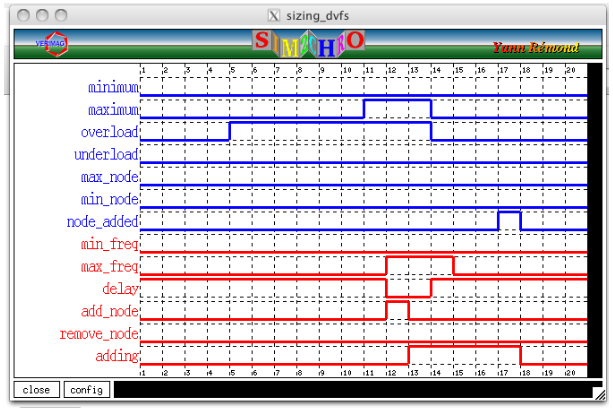


Figure 11. coordination controller simulation

are in their minimum CPU frequency and **maximum** notifies that all used nodes are in their maximum CPU frequency. The input **overload** represents the condition **CPU_avg > Max_threshold** and **underload** the condition **CPU_avg < Min_threshold**. **node_added** notifies that the previous adding node request have been treated succesfully. At the beginning, all used nodes are neither in their maximum CPU frequency nor in their minimum CPU frequency and the upsizing operations are not allowed (output **delay**).

At step 5, the input **overload** becomes true. This event should trigger an upsizing operation but, since all nodes are not in their maximum CPU frequency (output **max_freq**), this operation is not performed. An upsizing operation is performed only after the step 11 where all nodes are in their maximum CPU frequency.

VI. Implementation

A. Integrating the synchronous program into the system

The automata are composed in one BZR program. It is compiled and the generated code has two main functions: *reset* and *step*. The *reset* function allows to initialize the program and *step* to compute a reaction to events that correspond to the inputs of this function. The generated program is a reactive one. It has to be encapsulated into a loop that is responsible of executing the function *step* periodically or when an event occurs to get a reaction.

The coordination controller corresponds to the loop that encapsulates the BZR program. We have designed a program that is responsible of getting events from sensors for average load and Dvfs state, calling the function *step* with these events and transmitting the outputs of this function *step* to managers.

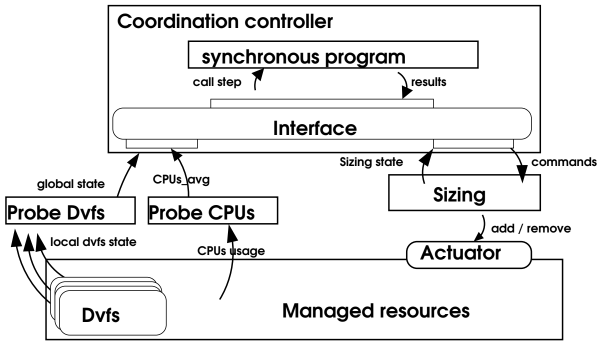


Figure 12. Integration

Figure 12 represents the architecture of a system in which the coordination controller is integrated. Since the role of this coordination controller is to control which manager should react or not to events, all sensed information is transmitted to the coordination controller instead of the managers. The outputs of the coordination, in reaction to events, are forwarded to the controlled managers i.e., in this case the manager Sizing. The interface allows interaction between the synchronous program, the sensors and the manager.

B. Connecting the automata

The automata are connected to the system by its input events, and by outputs which are commands to be applied in the system.

The automata represent the current state of a part of the system, which the coordination controller needs in order to make a decision when events occur. The inputs of automata have to be fed with events occurred in the system for making them evolve and their outputs have to be applied to the system for acting on its state.

The automaton *Dvfs_control* informs about the global state of the set of local Dvfs and its outputs serve only for the decision the controller has to make. The inputs of this automaton correspond to the outputs of the probe Dvfs. The automaton *Sizing_control* manages Sizing execution. It describes the current state of Sizing and decision it takes in response to events. The input **CPU_avg** corresponds to the average of system load. The inputs **max_node** and **min_node** correspond to the capability for Sizing to apply operations, max_node informing about the capability to perform an upsizing operation and min_node a downsizing operation. The output **add_node** respectively **remove_node** are triggering the operations Sizing performs when **CPU_avg** is over **Max_threshold** respectively **CPU_avg** is under **Min_threshold**. **add_node** being true respectively **remove_node** being true means Sizing has to add a new node respectively remove a node. The automaton *Delay_control* has one input which is managed by the generated controller. Its output is used as input for the automaton *Sizing_control*, in order to control some transitions Sizing_control can take.

C. Implementation architecture

This approach has been implemented for the management of a clustered web server. The managed system consists of one server Apache and replicated servers Tomcat.

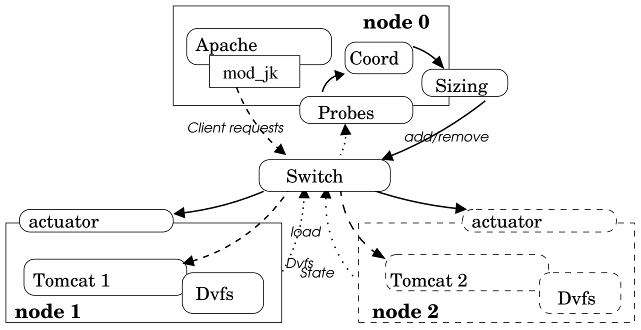


Figure 13. Experimental platform: architecture

The experimental platform, as shown in Figure 13, consists of a network of three nodes. Node 0 hosts the Apache server, each of node 1 and node 2, one Tomcat server. Initially, only one Tomcat server is active. Both Tomcat servers are active when the requests received cannot be handled by one. Unlike to the execution without coordination, where undesired behaviors have been observed, we observe that the coordination execution follows the defined policy. Upsizing operations are performed only when all active nodes hosting a server Tomcat are in their maximum CPU frequency.

In order for this controller to work well, it is important that it runs sufficiently frequently compared to the load dynamics : for every load peak to be detected and managed, the frequency of sampling and the communication must be fast enough.

VII. Related work

Concerning energy control, many works addressed energy management on datacenters. Some of these researches are based on (i) hardware with voltage and frequency control (e.g., DVFS [6]), (ii) resource allocation: Reducing power consumption by reducing the clock frequency of the processor has been widely studied [7] [18], Flautner et al. [5] explored a software managed dynamic voltage scaling policy that sets CPU speed on a task basis rather than by time intervals. [4] proposes a power budget guided job scheduling policy that maximizes overall job performance for a given power budget. [1] [13] [14] focused on dynamic resource provisioning in response to dynamic workload changes. These techniques monitor workloads or other SLA (Service Level Agreement) metrics experienced by a server and adjust the instantaneous resources available to the server. Depending on the granularity of the server (single or replicated), the dynamically provisioned resources can be a whole machine in the case of replicated servers. Energy efficiency is achieved using a workload-aware, just-right dynamic provisioning mechanism and the ability to power down subsystems of a host system that are not required.

While these works are relevant, they did not adress the problem of coordinating multiple energy managers. Our work is complementary since it can be used to build a system that includes more that one of the previous approaches. Few works have also investigated manager coordination for energy efficiency. Kumar [10] proposes vManage, a coordination approach that loosely couples platform and virtualization management to improve energy savings and QoS while reducing VM migrations. Kephart [2] addresses the coordination of multiple autonomic managers for power/performance tradeoffs based on a utility function in a non-virtualized environment. Nathuji [12] proposes VirtualPower to control the coordination among virtual machines to reduce the power consumption. These works involve coordination between control loops, but these loops are applied to the managed applications. However, these work propose adhoc specific solutions that have to be implemented by hand. If new managers have to be added in the system the whole coordination manager need to be redesigned.

In contrast with [15], which relies on formal specification to derive a formal model that is guaranteed to be equivalent to the requirements, our work can be related to the applications of control theory to autonomic or adaptive computing systems [8]. In particular, Discrete Event Systems in the form of Petri nets models and control have been used for deadlock avoidance problems [17]. Compared to these works, we rely on synchronous programming and discrete controller synthesis. Once an autonomic manager is modeled as automata, inserting this autonomic manager with other pre-existing just require to update the coordination invariants. The new coordination manager is automatically generated from the managers models and the coordination invariants. In contrast with [16], which addresses the management of datacenters based on thermal awarness with external sensing infrastructure for energy and cooling efficiency, the work, presented in this paper, focuses on coordinating multiple workload-aware managers to ensure an energy efficiency.

VIII. Conclusion and future work

One major challenge in system administration is the coordination of multiple autonomic managers for correct and coherent administration. In this paper we presented an approach for coordinating multiple self-management modules in a consistent manner to manage a system. This approach, based on synchronous programming and Discrete Controller Synthesis, has the advantage of generating the required controller to enable the correct by construction coordination of multiple autonomic managers. The advantages of this approach are following:

- High-level of programming
- Correctness of the controller
- Automated generation/synthesis of the controller
- That is maximally permissive

We have tested this approach for coordinating two energy-based self-management modules: Sizing, which manages the degree of replication for a system based on a load balancer scheme, and Dvfs, which manages the level of CPU frequency for a single node. In this case, the coordination policy was to allow Sizing to add new node only when all Dvfs modules cannot apply increase operations at all in response to the increasing load the system receives.

For future work, we plan to evaluate this approach for large scale coordination with more complex coordination policies and several managers, combining both self-optimization and self-regulation frequency managers with self-repair manager that heal fail-stop clustered multi-tiers system.

ACKNOWLEDGEMENT

This research is supported by ANR INFRA under a grant for the project ctrl-Green.

REFERENCES

[1] Sara Bouchenak, Noel De Palma, Daniel Hagimont, and Christophe Taton. Autonomic management of clustered applications. In *Cluster Computing*, 2006.

[2] Rajarshi Das, Jeffrey O. Kephart, Charles Lefurgy, Gerald Tesauro, David W. Levine, and Hoi Chan. Autonomic multi-agent management of power and performance in data centers. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: industrial track*, AAMAS '08, pages 107–114, Richland, SC, 2008.

[3] Gwenaël Delaval, Hervé Marchand, and Eric Rutten. Contracts for modular discrete controller synthesis. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, LCTES '10, pages 57–66, 2010.

[4] Maja Etinski, Julita Corbalan, Jesus Labarta, and Mateo Valero. Optimizing job performance under a given power constraint in hpc centers. In *Proceedings of the International Conference on Green Computing*, GREENCOMP '10, pages 257–267, Washington, DC, USA, 2010. IEEE Computer Society.

[5] Krisztián Flautner, Steve Reinhardt, and Trevor Mudge. Automatic performance setting for dynamic voltage scaling. *Wirel. Netw.*, 8:507–520, September 2002.

[6] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 78–91, New York, NY, USA, 1997. ACM.

[7] Kinshuk Govil, Edwin Chan, and Hal Wasserman. Comparing algorithm for dynamic speed-setting of a low-power cpu. In *MOBICOM'95*, pages 13–25, 1995.

[8] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. Wiley-IEEE, 2004.

[9] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36:41–50, January 2003.

[10] Sanjay Kumar, Vanish Talwar, Vibhore Kumar, Parthasarathy Ranganathan, and Karsten Schwan. vmanage: loosely coupled platform and virtualization management in data centers. In *Proceedings of the 6th international conference on Autonomic computing*, ICAC '09, pages 127–136, New York, NY, USA, 2009. ACM.

[11] Jean louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with state machines. In *In ACM International Conference on Embedded Software (EMSOFT' 05*, pages 173–182. ACM Press, 2005.

[12] Ripal Nathuji and Karsten Schwan. Virtualpower: coordinated power management in virtualized enterprise systems. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 265–278, New York, NY, USA, 2007. ACM.

[13] Eduardo Pinheiro, Ricardo Bianchini, Enrique V. Carrera, and Taliver Heath. Load balancing and unbalancing for power and performance in cluster-based systems, 2001.

[14] Ivan Rodero, Juan Jaramillo, Andres Quiroz, Manish Parashar, Francesc Guim, and Stephen Poole. Energy-efficient application-aware online provisioning for virtualized clouds and data centers. pages 31–45, 2010.

[15] Roy Sterritt, Michael Hinchey, James Rash, Walt Truszkowski, Christopher Rouff, and Denis Gracanin. Towards Formal Specification and Generation of Autonomic Policies. In *Embedded and Ubiquitous Computing*, pages 1245–1254, 2005.

[16] Hariharasudhan Viswanathan, Eun Lee, and Dario Pompili. Self-organizing sensing infrastructure for autonomic management of green datacenters. *Ieee Network*, 25(4):34–40, 2011.

[17] Yin Wang, Terence Kelly, and Stéphane Lafortune. Discrete control for safe execution of it automation workflows. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 305–314, New York, NY, USA, 2007. ACM.

[18] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

Action Learning with Reactive Answer Set Programming: Preliminary Report

Michal Čertický
Department of Applied Informatics
Comenius University Bratislava, Slovakia
certicky@fmph.uniba.sk

***Abstract*—Action learning is a process of automatic induction of knowledge about domain dynamics. The idea to use Answer Set Programming (ASP) for the purposes of action learning has already been published in [2]. However, in reaction to latest introduction of *Reactive ASP* and implementation of effective tools [6], we propose a slightly different approach, and show how using the *Reactive ASP* together with more *compact knowledge encoding* can provide significant advantages. The technique proposed in this paper allows for real-time induction of action models in larger domains, and can be easily modified to deal with sensoric noise and non-determinism. On the other hand, it lacks the ability to learn the conditional effects.**

***Keywords*-ASP; learning; actions; induction;.**

I. Introduction

A. Action Learning and Reasoning about Actions

Knowledge about domain dynamics, describing how certain actions affect the world, is essential for planning and intelligent goal-oriented behaviour of both living and artificial agents. In artificial systems, this knowledge is referred to as **action model**. It is an expression of all the **actions** that can be executed in a given domain, with all their **preconditions** and **effects** in some kind of representation language.

Action learning, as a type of reasoning about actions, is a process of automatic generation and/or modification of action models based on sensoric observations. Autonomous and automated systems may benefit from action learning, since it allows them to adapt to unpredicted changes in environment's behaviour (caused for example by addition of new unknown agents, by hardware malfunction, etc.).

Action models in general are used specifically for the purposes of planning. As a motivating example, imagine an autonomous *Mars Rover* robot (similar to one described in [4]). Such robot acts in an unknown environment, but plans its actions based on a static hard-wired knowledge about about their effects. Now imagine, that one of its wheels gets damaged, which would change the effects of the *"moveForward"* action. If such robot was capable of action learning, it could update his action model accordingly, and continue acting successfully towards its goals.

B. Background and Methods

Recent action learning methods take various approaches and employ the wide variety of AI tools. We should mention the action learning technique based on heuristic greedy search, introduced in [16], perceptron algorithm-based method which can be found in [11], two solutions based on the reduction of action learning into different classes of satisfiability problems, available in [1] and [15], learning with inductive logic programs described in [12], or learning module written in Answer Set Programming (ASP), described by M. Balduccini in [2].

The method we propose in this paper is closest to Balduccini's learning module, in that it also uses ASP for induction and representation of action models. There are however several differences, that make it usable under different conditions. We will address these differences in detail in Section IV.

C. Reactive ASP

Answer Set Programming (ASP [7], [3]) has lately become a popular declarative problem solving paradigm, with growing number of applications [6], among others also in the field of reasoning about actions. Semantics of ASP enables us to elegantly deal with incompleteness of knowledge, and makes the representation of action's properties easy, due to nonmonotonic character of negation as failure operator [10]. With ASP, our knowledge is represented by so-called extended logic programs - sets of rules of the following form:

$$c \leftarrow a_1 \dots a_n, not\ b_1 \dots not\ b_m.$$

where a_i, b_j and c are literals, i.e., either first-order logic atoms, or atoms preceded by explicit negation sign "$\neg$". The "not" symbol denotes negation by failure. Part of the rule before "$\leftarrow$" sign is called *head* and part after it is *body*. Rule with an empty body (n = m = 0) is called a *fact* and rule without a head is an integrity *constraint*. Every logic program has a corresponding (possibly empty) finite set of **answer sets**.

Since we are dealing with dynamic systems, we can take advantage of so-called incremental logic programs. An incremental logic program is a triple $(B, P[t], Q[t])$ of logic programs, with a single parameter t ranging over natural numbers [5]. While B only contains **static** knowledge, parametrized $P[t]$ constitutes a **cumulative** part of our knowledge ($Q[t]$ is so-called **volatile** part, but we don't use it in our solution). In our method, t will always identify the most recent time step, and $P[t]$ will describe how the newest observation affects our previous beliefs.

In [6], Gebser et al. go further, and augment the concept of incremental logic programming with asynchronous information, refining the statically available knowledge. They also introduce their **reactive ASP solver** *oClingo* which we use in our solution.

II. Learning with Reactive ASP

We will now describe our learner, which is basically a short incremental logic program $(B, P[t], \emptyset)$. It has a large (but gradually decreasing) number of **answer sets**, each **corresponding to** a single **possible action model**.

At time step 1, the online ASP solver *oClingo* [6] computes the first answer set and stores it in memory (along with all the partial computations it has done so far). At every successive time step, we confront this answer set with new observations.

Note 1: The *oClingo* is a server application, which listens on a specified port and waits for new knowledge. This knowledge is sent to *oClingo* by a *controller* application, and in our case always represents the latest action and fluent observations.

A. Encoding the Action Model

First of all, we needed to choose the **encoding** of action models into a logic programs that is sufficiently expressive, but at the same time remains **as compact as possible**.

The main idea behind our encoding lies in the fact that for a given fluent F, every possible action A can either:

1) **cause** a fluent F to hold (we encode this by a fact `causes(A,F).`),
2) **cause** a **complementary** fluent $\neg F$ to hold (`causes(A,¬F).`),
3) or **keep the value** of that fluent literal (`keeps(A,F).` resp. `keeps(A,¬F).`).

In addition to that, we want our action models to contain the information about action's executability. In that respect, every action A can either:

1) **have** a fluent F as its **precondition** (encoded by a fact `pre(A,F).`),
2) or **not have** it as its **precondition** (`¬pre(A,F).`).

B. Generating Answer Sets

Answer sets corresponding to all the possible action models are generated by **static** part of our program (logic program B). It consists of the set of rules depicted in figure 1.

In the first part, we have three **choice rules**, that **generate answer sets** where A either causes F, causes $\neg F$, or keeps F. Next two **constraints filter out** the answer sets corresponding to **impossible models** - where A causes F and $\neg F$ at the same time, or where A both keeps and changes the value of F. Last two rules merely express the equivalence between two possible notations of A keeping F.

```
% Effect generator and axioms:
causes(A,F) ← not causes(A,¬F), not keeps(A,F).
causes(A,¬F) ← not causes(A,F), not keeps(A,F).
keeps(A,F) ← not causes(A,F), not causes(A,¬F).
← causes(A,F), causes(A,¬F).
← causes(A,F), keeps(A,F).
keeps(A,F) ← keeps(A,¬F).
keeps(A,¬F) ← keeps(A,F).

% Precondition generator and axioms:
pre(A,F) ← not ¬pre(A,F).
¬pre(A,F) ← not pre(A,F).
← pre(A,F), ¬pre(A,F).
← pre(A,F), pre(A,¬F).
```

Figure 1. Static (time-independent) part of our learner logic program.

Second part is very similar. Here we have two choice rules generating answer sets where A either has, or doesn't have a precondition F. Constraints here eliminate those answer sets, where A has and doesn't have precondition F at the same time, or where it has both F and $\neg F$ as its preconditions.

Note 2: Keep in mind, that the logic programs in this paper are simplified to improve the readability and save some space. You can download the exact ready-to-use ASP solver compatible encodings from [9].

C. Answer Set Elimination

Now, we need to process new knowledge received at successive time steps[1]. We use a time-aware, **cumulative** program $P[t]$ for that.

```
#external obs/2.
#external exe/2.
← obs(F,t), exe(A,t), causes(A,¬F).
← obs(¬F,t), obs(F,t-1), exe(A,t), keeps(A,F).
← exe(A,t), obs(¬F,t-1), pre(A,F).
```

Figure 2. Cumulative (time-aware) part of our learner logic program.

First two statements merely instruct *oClingo* that it should accept two kinds of binary atoms from the *controller* application[2]: fluent observations `obs` and executed actions `exe`.

Remaining **three constraints take care of actual learning**, by **disallowing answer sets** that are **in conflict with** the latest **observation**. First constraint says, that if F was observed after executing A, then A cannot cause $\neg F$. Second constraint tells us, that if F changed value after executing A, then A does not keep the value of F. And the last one means, that if the action A was executed when $\neg F$ held, F cannot be a precondition of A.

[1]Note, that we allow at most one action to be executed in every time step.

[2]Telling *oClingo* what to accept is not necessary, if we use new `--import=all` parameter. This option was implemented only after we designed our logic programs.

```
#step 9.
exe(move(b1,b2,table),9).
obs(on(b1,table),9).
obs(¬on(b1,b2),9).
obs(on(b2,table),9).
obs(¬on(b2,b1),9).
#endstep.
```

Figure 3. Observation example from Blocks World domain[14] sent to *oClingo* at time step number 9. It describes the configuration of two blocks $b1$ and $b2$ on the table after we moved $b1$ from $b2$ to the $table$.

At every time step, these constraints are added to our knowledge with parameter t substituted by a current time step number. Also, we need to add the latest observation. These observations are sets of `obs` and `exe` atoms. See the example of observation that is sent to *oClingo* in figure 3.

We say, that a constraint “fires” in an answer set, if its body holds there. In that case, this answer set becomes illegal, and is thrown away. Now recall, that in time step 1, *oClingo* generated the first possible answer set and stored it in memory. An observation like the one above can cause some of our constraints to fire in it and eliminate it. For example, if our answer set contained the `causes(`$move(b1,b2,table)$`,`$on(b1,b2)$`)` atom, the first constraint would fire.

If that happens, *oClingo* simply **computes** the **new answer set**, that is not in conflict with any of our previously added constraints. This is how we update our knowledge, so that we always have an action model consistent with previous observations at our disposal. Note, that each observation potentially reduces the number of possible answer sets of our logic program $(B, P[t], \emptyset)$, thus making our knowledge more precise. After a sufficient number of observations, $(B, P[t], \emptyset)$ will have only one possible answer set remaining, which will represent the correct action model.

III. Dealing with Noise and Non-determinism

The problem may arise, in the presence of sensoric noise or non-deterministic effects, since the noisy observations could eliminate the correct action model. This could eventually leave us with an empty set of possible models. However, we propose a workaround, that can overcome this issue.

The problem with non-determinism is, that we cannot afford to eliminate the action model after the first negative example. We need to have some kind of error tolerance. For that reason, we should modify the cumulative part of our program $P[t]$, so that our constraints fire only after a certain number of negative examples:

Here we can see, that our observations don’t directly appear in the bodies of constraints. Instead, they are capable of increasing the negative example count (which is kept in the variable `C` of `negExCauses`, `negExKeeps` and `negExPre` predicates). Constraints then fire, when the number of negative examples is higher than 5.

```
negExCauses(A,F,C+1) ←
        negExCauses(A,F,C), obs(¬F,t), exe(A,t).

negExKeeps(A,F,C+1) ←
        negExKeeps(A,F,C), obs(¬F,t),
        obs(F,t−1), exe(A,t).

negExPre(A,F,C+1) ←
        negExPre(A,F,C), exe(A,t), obs(¬F,t−1).

← causes(A,F), negExCauses(A,F,C), C > 5.
← keeps(A,F), negExKeeps(A,F,C), C > 5.
← pre(A,F), negExPre(A,F,C), C > 5.
```

Figure 4. Modified $P[t]$ should be able to deal with sensoric noise, by introducing error tolerance.

Note 3: The *error tolerance threshold* is a numeric constant 5 here to keep things simple, but we can easily imagine better, dynamically computed threshold values (for example based on the percentage of the negative examples in the training set, etc.).

IV. Comparison to Alternative Method

Let us now take a closer look at the similarities and differences between our method and Balduccini’s *learning module* approach described in [2].

	Inc. kn.	Prec.	Cond. eff.	Noise	Real-time
Balduccini	yes	yes	yes	?	no
Our method	yes	yes	no	yes	yes

A. Incomplete Knowledge

From the viewpoint of domain compatibility, both methods share the ability to deal with incompleteness of knowledge. The absence of complete observations may slow down the learning process[3], but cannot lead us to induce incorrect action models.

B. Action’s Preconditions

Our method learns not only the effects, but also preconditions of actions. Similarly, Balduccini’s learning module supports the induction of preconditions, in a form of so-called *impossibility conditions*.

C. Conditional Effects

Balduccini’s learning module allows for direct induction of *conditional effects*, which is a greatest advantage over our method. Having the conditional effects allows for more elegant representation of resulting action models. (Note however, that they are not necessary, and can be expressed by a larger number of actions with right preconditions.) On

[3]By *slowing down* we understand, that we might require more time steps to induce precise action models. Incompleteness of observations will *not* hinder the computation times at individual time steps.

the other hand, we must keep in mind that learning them is in general harder and more time-consuming problem.

D. Encoding of Action Models

An action model is in the case of Balduccini's learning module encoded by a set of atoms of the following types: $d\_law(L)$, $s\_law(L), head(L, F), action(L, A)$, and $prec(L, F)$, with L substituted by a name (unique constant identifier) of a C-language [8] law, A by an action instance, and F by a fluent literal. Notice, that this way, we directly encode the syntactic form of individual C-language laws into logic programs. See figure 5 for a simplistic example of this encoding.

Following C-language law:

caused on(b1,table) after move(b1,b2,table), on(b1,b2), free(b1), free(table).

Is in Balduccini's learning module translated into:

```
d_law(dynamicLaw25).
head(dynamicLaw25, on(b1,table)).
action(dynamicLaw25, move(b1,b2,table)).
prec(dynamicLaw25, on(b1,b2)).
prec(dynamicLaw25, free(b1)).
prec(dynamicLaw25, free(table)).
```

Figure 5. Example of C-language $\rightarrow$ LP translation used in Balduccini's learning module.

Every time the observation is added, the whole history is confronted with this set. If the observation is not explained [2] by it, we add more atoms to it (either creating new laws, or adding effect conditions to existing ones).

In our case, the action model encoding is much more compact[4]. We don't translate an action model from any given planning language, which allows us to omit the d_law and s_law predicates and L parameter. Instead we have chosen an abstract, semantics-based, direct encoding of a domain dynamics, where every effect or precondition is represented by a single atom. Notice also, that the size of our action model doesn't increase over time.

E. Extending the Techniques

The bottom line here is, that our representation structures are simpler, for the price of lower expressiveness. The semantic-based encoding makes it fairly easy to extend learning by an ability to deal with noise. It is probable, that Balduccini's learning module could also be similarly extended, but it would be far less straightforward process (it consists of 14 rules, describing a syntactic structure of action model, rather than focusing directly on semantics).

[4]Note, that the main reason we can afford more compact encoding is the fact, that we don't use conditional effects.

F. Performance and Online Computation

Our method can be considered **semi-online**, in a sense that we only consider the most recent observation as relevant input for our computation. This is possible because of the use of **Reactive ASP**: At each time step, the solver keeps everything that it has computed at previous steps in memory, and only adds new observation. If the current model is disproved, some revisions might be needed, but significant part of the computation has already been performed and results are stored in memory. This, together with relatively compact encodings allows us to learn action models in **real-time**.

G. Experiments and Conclusion

In [2], we can find an experimental comparison of Balduccini's learning module and Otero and Varela's *Iaction* learning system [12]. Their experiment was conducted with 5 narratives of 4 blocks and 6 actions. *Iaction* system found a solution in 36 seconds on Pentium 4, 2.4GHz, while the results of Balduccini's module was fairly comparable with 14 seconds on somewhat faster computer (Pentium 4, 3.2Ghz).

To demonstrate the speedup resulting from using a compact encoding and Reactive ASP, we have decided to try significantly larger problem instance. Our domain was also a Blocks World with 4 blocks + table (`b1,b2,b3,b4,table`), but we had **32** possible **actions** (valid instances of `pickup(Block,From)` and `puton(Block,To)`) and our training set consisted of **150 observations** of randomly chosen legal actions.

Processing the training set with full observations took 17.9 seconds, while similar set with partial observations took 14.98 seconds. The experiment was performed with *oClingo* solver, version 3.0.92b, under 64bit Linux system with Intel(R) Core(TM) i5 3.33GHz CPU. The input files that we used can be downloaded from [9], together with detailed instructions.

We conclude, that the decrease in the size of encoding, together with preservation of results from previous time steps (using reactive ASP approach), enables us to learn action models considerably faster. This seems to result from the fact, that the complexity of answer set computation algorithms is exponential in the number of input atoms [13] (thus reducing the input for solver even a little can speed up the computation significantly), and that an important part of computation can be recycled from previous time steps.

In the near future, we are planning to provide an in-depth comparison of our method to a wide variety of action learning approaches, followed by a specification of the most apropriate practical applications in the area of autonomous and automated systems.

Acknowledgment

We acknowledge the support from VEGA project of Slovak Ministry of Education and Slovak Academy of Sciences no. 1/0688/10.

References

[1] E. Amir and A. Chang. *Learning partially observable deterministic action models*. Journal of Artificial Intelligence Research, Volume 33 Issue 1, pp. 349-402. 2008.

[2] M. Balduccini. *Learning Action Descriptions with A-Prolog: Action Language C*. In Proceedings of Logical Formalizations of Commonsense Reasoning, 2007 AAAI Spring Symposium. 2007.

[3] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press. 2003.

[4] T. Estlin et al. *Increased Mars rover autonomy using AI planning, scheduling and execution*. Proceedings 2007 IEEE International Conference on Robotics and Automation, pp. 4911-4918. 2007.

[5] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. *Engineering an Incremental ASP Solver*. In Proceedings of the 24th International Conference on Logic Programming (ICLP'08), pp. 190-205. 2008.

[6] M. Gebser, T. Grote, R. Kaminski, and T. Schaub. *Reactive Answer Set Programming*. In Proceedings of 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'2011), pp. 54-66. 2011.

[7] M. Gelfond and V. Lifschitz. *Classical Negation in Logic Programs and Disjunctive Databases*. New Generation Computing, pp. 365-387. 1991.

[8] E. Giunchiglia and V. Lifschitz. *An Action Language based on Causal Explanation: Preliminary Report*. In proceedings of 15th National Conference of Artificial Intelligence (AAAI'98), pp. 623-630. 1998.

[9] www.dai.fmph.uniba.sk/upload/9/9d/Oclingo-learning.zip.
Last accessed: 13 February 2012.

[10] V. Lifschitz. *Answer Set Programming and Plan Generation*. Artificial Intelligence, vol. 138, pp. 39-54. 2002.

[11] K. Mourao, R. P. A. Petrick, and M. Steedman. *Learning action effects in partially observable domains*. In Proceeding of the 2010 conference on ECAI 2010: 19th European Conference on Artificial Intelligence, pp. 15-22. 2010.

[12] R. Otero and M. Varela. *Iaction, a System for Learning Action Descriptions for Planning*. In Proceedings of 16th International Conference on Inductive Logic Programming, ILP 06. 2006.

[13] P. Simons, I. Niemela, and T. Soininen. *Extending and Implementing the Stable Model Semantics*. Artificial Intelligence, 138(1-2), pp. 181-234. 2002.

[14] J. Slaney and S. Thiebaux. *Blocks World revisited*. Artificial Intelligence 125, pp. 119-153. 2001.

[15] Q. Yang, K. Wu, and Y. Jiang. *Learning action models from plan examples using weighted MAX-SAT*. Artificial Intelligence, Volume 171, pp. 107-143. 2007.

[16] L. S. Zettlemoyer, H. M. Pasula, and L. P. Kaelbling. *Learning probabilistic relational planning rules*. MIT TR. 2003.

Development and Evaluation of a Self-Adaptive Organic Middleware for Highly Dependable System-on-Chips

Benjamin Betting, Mathias Pacher, and Uwe Brinkschulte
Institute for Computer Science
Johann Wolfgang Goethe-University
Frankfurt am Main, Germany
Email: {betting, pacher, brinks}@es.cs.uni-frankfurt.de

***Abstract*—This article presents our concept of an artificial hormone system for realizing a completely decentralized self-organizing task allocation using self-X properties. Besides the basics of the prior hormone concept and possible realizations in soft- as well as hardware, we present latest results of our research: evaluation of a completely AHS-controlled SoC implementing the different approaches, verifying the work and stability criteria, analysis of upper timing boundaries and showing the improvement of the reliability of the system.**

***Keywords*-Reliability; Artificial Hormone System; Heterogeneous System-on-Chips.**

I. Introduction

Because the performance of nowadays computational systems is still increasing rapidly within each generation, the complexity to handle and manage such systems has grown in a similar way, too. Today's systems offer a high bandwidth of functionality, considering an integration of large numbers of distributed heterogeneous processing resources, showing a highly dynamic behavior in time. Although the architectural design of distributed systems differs strongly, a common layer is still provided by Middleware, managing the coordination of tasks on the corresponding resources and also hiding the distribution from the application. To be precise, Middleware is responsible for seamless task interaction on distributed hardware. All tasks are controlled by the Middleware layer and are able to operate beyond processing element boundaries as if they would reside on a single hardware platform. Besides complexity, other system criteria like reliability have become important, too. In consequence of the increasing integration density of today's SoCs, systems got likely open to system failure even during the early stages of operation. Crashing of resources can be caused, e.g., by radiation, aging or temperature hot spots. Hence, in order to handle the complex task management as well as the reliability problems of today's and even more future distributed systems, self- organizing and adapting techniques are necessary. As the term 'self' denotes, these techniques must be achieved autonomously by the system itself without any further external intervention (introduced in [7][8]). In fact, a system should be able to find a suitable initial configuration of task assignment by itself, to adapt or optimize itself to changing environmental and internal conditions, to heal itself in case of system failures or to protect itself against attacks. In this paper, we present the solution of an organic Middleware - implemented by an Artificial Hormone System (AHS) - providing self-configuration, self-optimization and self-healing for an autonomous decentralized task assignment. Furthermore, the proactive task behavior to prevent the system from arising failure conditions are implemented. In Section II, we introduce the basic principles of the AHS. Sections III and IV show theoretical constraints and implementations of the approach, which are evaluated in Section V. Topics of related work are presented in Section VI. Finally, we conclude the paper with Section VII.

II. The Artificial Hormone System

According to organic endocrine systems, the AHS considers elemental exchange of different hormone types for special communication and controlling interaction. In fact, it is the main function of the AHS to assign tasks to resources without any further external intervention. The proper assignment is handled in a self-organizing way, implemented via simple resource competition upon tasks using three major types of hormones:

- ***Eager value*** This hormone type represents the suitability of a resource to execute a task. The higher the hormonal value, the better the ability of the resource to execute the task.
- ***Suppressor*** This hormone type lowers the suitability of a task execution on a resource. Suppressors are subtracted from eager values.
- ***Accelerator*** This hormone type favors the execution of a task on a resource. Accelerators are added to eager values.

These basic hormone types are divided in further subtypes. Detailed information about these subtypes is presented when needed because they are used for fine tuning of the AHS and do not affect its basic understanding.

Keeping consistent formalism we introduce special notation, which distinguishes between received hormones, hormones to be sent and also between tasks and processing resources (like, e.g., processing cores). Therefore, tasks and

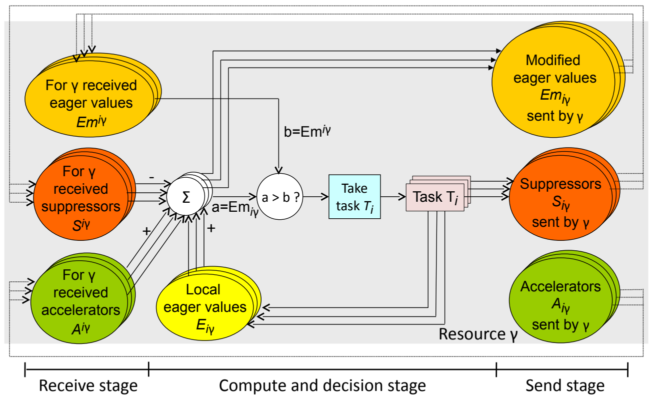

For γ received eager values $Em^{i\gamma}$
Modified eager values $Em_{i\gamma}$ sent by γ
b=$Em^{i\gamma}$
For γ received suppressors $S^{i\gamma}$
Σ
a=$Em_{i\gamma}$
a > b ?
Take task T_i
Task T_i
Suppressors $S_{i\gamma}$ sent by γ
For γ received accelerators $A^{i\gamma}$
Local eager values $E_{i\gamma}$
Accelerators $A_{i\gamma}$ sent by γ
Resource γ
Receive stage
Compute and decision stage
Send stage

Figure 1. Hormone based control loop

resources are referenced by different indices using Latin letters such as i for tasks and Greek letters such as γ for resources. The notation of raised $H^{i\gamma}$ indents a hormone, which will be received by resource γ, dedicated and effecting task T_i. In turn subscript representation of $H_{i\gamma}$ declares resource γ and task T_i as emitter of the hormone, which is send to other resources. Each resource periodically executes the hormone based control loop presented in Figure 1. Each iteration consists of three stages.

- *Receive stage*: Resource γ receives the modified eager values $Em^{i\gamma}$, suppressors $S^{i\gamma}$ and accelerators $A^{i\gamma}$ for each task T_i from each resource inside the network. The communication between the different resources is depicted by the dashed lines.
- *Compute and decision stage*: Resource γ computes the modified eager values $Em_{i\gamma}$ for all of its tasks by the following rule. The local static eager value $E_{i\gamma}$ indicates how suited γ is to execute task T_i. From this value, all suppressors $S^{i\gamma}$ received by task T_i are subtracted, and all accelerators received by task T_i are added:

$$Em_{i\gamma} = E_{i\gamma} - \sum S^{i\gamma} + \sum A^{i\gamma} \qquad (1)$$

  The modified eager value $Em_{i\gamma}$ of each task T_i is finally broadcasted to the same task T_i on the other resources in the send stage. In each iteration a single task T_i is selected and the resource decides on allocation. For this purpose, it compares its own modified eager value $Em_{i\gamma}$ with the received modified eager values $Em^{i\gamma}$ (from all other resources) for this task. If $Em_{i\gamma} > Em^{i\gamma}$ is true for all received modified eager values $Em^{i\gamma}$, it decides to take the task. In case of equality, a second criterion, e.g., the unique identifier of the resources, is used to get an unambiguous decision. Otherwise another resource has the highest modified eager value for T_i and γ decides to not take it.
  In the next iteration step the resource selects another task and decides whether it will be taken. A resource selects the tasks in a cyclic way, i.e., each task will be selected in each m^{th} iteration, if m tasks have to be assigned. By selecting only one task at each iteration, the suppressors and accelerators can take effect. Otherwise, the decision of taking a task would happen instantaneously and the hormones would have no effect.
- *Send stage*: As already mentioned above, resource γ broadcasts the modified eager values $Em_{i\gamma}$ to each task T_i on the other resources. The strength of these values depends on the results of the computation in the last phase.
  If a task T_i is taken on resource γ, it also broadcasts suppressors $S_{i\gamma}$ dedicated to the same task on all other resources. On one hand sending the suppressors indicates the resource has taken the task, and on the other hand, it limits the number of further allocations of the same task somewhere else.
  Furthermore, the resource multicasts accelerators $A_{i\gamma}$ to its neighbored resources to attract tasks cooperating with task T_i to neighbored resources, thus forming clusters of tasks.

Our approach is completely decentralized, each resource is responsible for its own tasks and the communication to other resources is realized by a unified hormone concept. The AHS offers the following so called self-X properties:

- The approach is self-organizing, because no external influence controls the task allocation.
- It is self-configuring, an initial task allocation is found by exchanging hormones. The self-configuration is finished as soon as all modified eager values become zero, meaning no more tasks have to be taken.
- The self-optimization is done by re-offering tasks for allocation. The point in time for such an offer is determined by the task or by the resource.
- The approach is self-healing: In case of a task or resource failure, the emission of related hormones stops. This results in an automatic reassignment of the task to the same resource (if it is still active) or to another resource.

In addition, the self-configuration, self-optimization and self-healing is real-time capable. Tight upper time bounds are given for self-configuration, these are presented in detail in [2][4][12].

III. Implementation Constraints

Additional to the theoretical concept of Section II, the real implementation of AHS has to consider further aspects of the environmental system surrounding. Based on the primary decentralized control loop mechanism of Section II, every resource or processing element holds its own local instance of the AHS during runtime. In fact today's SoCs offer the significant opportunity of an dual realization of the AHS in

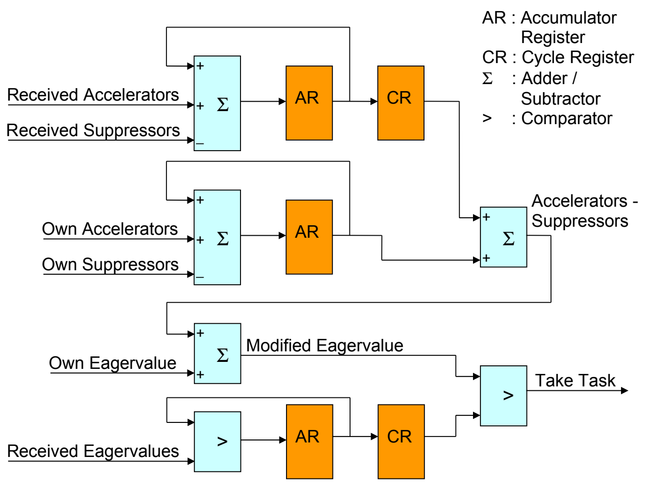


Figure 2. Simply buffered solution of the hormone cycle implementation using self-synchronization of cycles

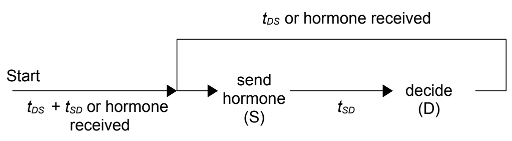

t_{DS} or hormone received
Start
t_{DS} + t_{SD} or hormone received
send hormone (S)
t_{SD}
decide (D)

Figure 3. Self-synchronized approach of the hormone cycle using waiting period after startup

both domains of soft- as well in hardware. Obviously this is reducible to the likely natural representation of hormone signals in terms of simple coded messages easily delivered between tasks and resources.

First a total hormone buffered solution, using a local memory backing up all received hormones for each task within each cycle, is traced. This approach keeps the advantage of no further required modifications of the origin hormone cycle and its steps, retaining a still asynchronous behavior during run time. Due to the hormone memory, jitter effects can be compensated. Duplicate hormones can be identified and eliminated while missing hormones can be bridged by previous values. But the approach also considers a high effort of required management, controlling those hormone memories, for, e.g., keeping them consistent. In cause of this, the real implementation uses a simplified second approach discarding the complex hormone memory and exploits the accumulation character of the hormones as described in Section II (adding accelerators and subtracting suppressors from eager values). As shown in Figure 2, hormone values of the current cycle are just accumulated and the overall sum of the previous cycle is stored in a register. Instead of storing many hormone values for each task within each cycle, just the last recent accumulations for accelerators and suppressors as well as the highest modified eager value are kept. Considering this, the approach takes a high benefit and requires a less level of buffering complexity than the first approach. But due to the asynchronous processing resources the risk of evaluating inconsistent hormones remains.

Therefore, this approach considers a modification of the origin cycle scheme tracing an active self-synchronization of the hormone cycles. This feature is simply achieved by using the received hormones for synchronization. To be precise, each cycle holds a waiting period right after startup, before emitting hormones to other resources. This waiting period ends after the total hormone cycle time or the receipt of a hormone from another resource, whatever happens first. So, one resource will be the first and the others will follow when receiving the hormones from the first resource. The same happens after each loop period. This keeps all resources synchronized within the maximum communication time t_k between the resources. Figure 3 shows a detailed timing schedule of the hormone cycle, where t_{DS} denotes the maximum time span between the decision and sending stage and vice versa t_{SD} between the stage of sending and decision. Hence the maximum total cycle time is set up by accumulation of t_{SD} + t_{DS}. To specify both spans as constraints, the maximum time displacement between the earliest cycle evaluating resource P_γ and the latest one P_δ, which is the already mentioned maximum communication time t_k between P_γ and P_δ, must be considered. In order to satisfy the receipt of a hormone sent by the later P_δ to the earlier P_γ, t_{SD} must be at least $2 * t_k$ (t_k needed for the communication + t_k as maximum time displacement between P_δ and P_γ). The definition of t_{DS} is rather simple. To guarantee a synchronized restart of both the early P_γ and the late P_δ, t_{DS} has to be at least t_k. Due to tolerances caused by timers on local resources, an additional jitter compensation factor $\triangle t_{SD}$ has to be considered. So t_{DS} has to be at least $t_K + t_{SD}$, which guarantees a synchronous start of the next hormone cycle on every resource. Finally the total time of the self-synchronized hormone cycle is set up by the accumulation of both constraints to at least $3 * t_K + \triangle t_{SD}$.

In summary, the major advantage of the self-synchronized approach is the feature of hormone consistency. Within each cycle, resources are capable to take correct decisions right after the receipt of all necessary hormones sent by other resources. Furthermore this modification does not influence the already taken assumptions about other self-X properties like healing or configuration, unless the basic processing procedure of the hormone cycle will not be modified. As a disadvantage, it causes a hormone cycle time increase by $t_K + \triangle t_{SD}$ compared to the asynchronous hormone buffered solution, which requires only $2 * t_k$ as cycle time (see [3]).

IV. Implementation

Because the AHS is intended to control a decentralized assignment of tasks in software- as well as in hardware related domain, specific implementations regarding different system environments are required. In fact, two implementations of the AHS have been developed, providing a specific realization of the prior hormone loop for the processing of software and hardware tasks.

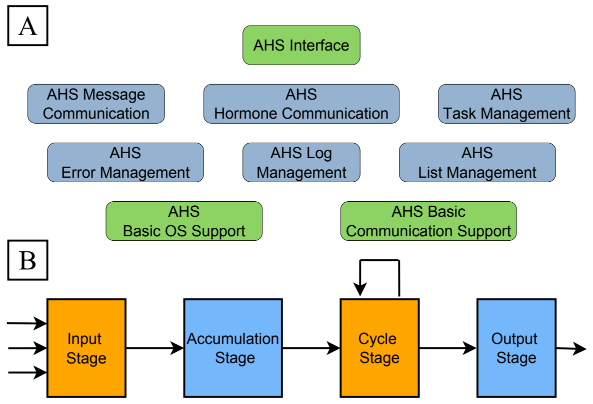


Figure 4. Structure of software (A) and hardware (B) implemented AHS, coded in ANSI-C and VHDL

A. AHS on software-level

In order of compatibility reasons and platform independence, the AHS features a pure coded ANSI-C implementation. Considering this, the implementation possesses a high level of reusability and achieves the goal of platform independent Middleware. This fact also leads to a usability in the domain of low-performance microcontroller computing. The implementation of AHS is set up by different modules and interfaces related to the AHS kernel as well as the attached Operating System (OS) and distributed application. The implementation structure is shown in Figure 4A. Starting on the lower operating system level, elementary modules for task handling and I/O communication are supported. In case of platform porting, these modules must be implemented. Regarding this abstraction, the approach takes high benefit and requires less effort in modification and maintenance. Upon OS modules, the main kernel of the AHS takes place, isolated of the attached platform depended components.

B. AHS on hardware-level

Porting the basic hormone concept to real hardware implementation takes the approach one step further. Against pure software implementation, the entire hormone loop mechanism is spread up into 4 different pipeline stages (Figure 4B). Each stage represents a single isolated hardware component, which implements a specific step of the proper hormone loop shown in Section II. Further, this pipeline has currently been fully implemented on register transfer level by hardware description using Very High Speed Integrated Circuit Hardware Description Language (VHDL). Within each single hormone cycle all stages of the pipeline are passed, issuing the decision whether task T_i is taken by the corresponding resource or not. Because the current hardware implementation realizes the self-synchronized variant of AHS only, an cycle stage is attached for buffering already accumulated hormone values, avoiding the use of heavy weighted local memories. This stage also confirms the criteria of taking correct decisions upon consistent hormone data by delaying progress of the current evaluation cycle until all necessary hormone data of all other resources is received. To avoid everlasting stall of the pipeline due to missing data, this stage is passed lately after the timing constraint of $T_{SD} \geq 2 * t_k$ (shown in Section III) is expired.

V. Evaluation

As next step, an evaluation to show the increase of dependability using the hormone system for task assignment in a distributed system is conducted. To achieve full insights in hormone processing, a hormone cycle accurate hormone simulator for the AHS has been developed [13]. Besides the capability of simulating a dynamic processing grid, containing multiple mixed signal resources, the provision of self-X properties is ensured. We used the grid of 16 heterogeneous resources with 16 different tasks to be executed. This simulation focuses on self-healing of dynamic failures during runtime. As a reference, we first run a simulation with deactivated self-healing. This means, after the self-configuration process the hormone cycle was deactivated. Failures caused by single event upsets, aging and temperature effects have been created by a stochastic process according to corresponding failure models described. In the first simulation, transient and permanent failures leading to task or core crashes are considered. Figure 5 shows the result. Initially, all 16 tasks are allocated by self-configuration. This process is finished at hormone cycle 6, so at that time the system is operational. Already at hormone cycle 7 the first failure, a single event upset, crashed one task. More task crashes due to single event upsets followed at hormone cycles 25, 37, 47 and 51 further reducing the number of active tasks. No aging or temperature based failures occurred up to that point in time. So, starting from hormone cycle 7 the system is no longer operational as can be seen by the linearly increasing system downtime (violet line) resulting in a downtime ratio of $50/51 = 0.98 = 98\%$ at hormone cycle 51. Figure 6 shows the same scenario with self-healing activated by the hormone cycle. To be comparable, the stochastic process creating the failures was initialized with the same random seed to produce identical events. Again, the system comes operational by allocating all 16 tasks at hormone cycle 6 while at hormone cycle 7 the first single event upset occurred crashing a task. This caused the corresponding task suppressor to vanish. Due to

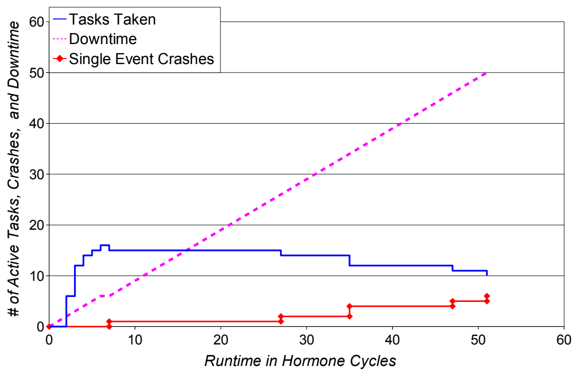


Figure 5. Behavior of 16 AHS resources with deactivated self-healing

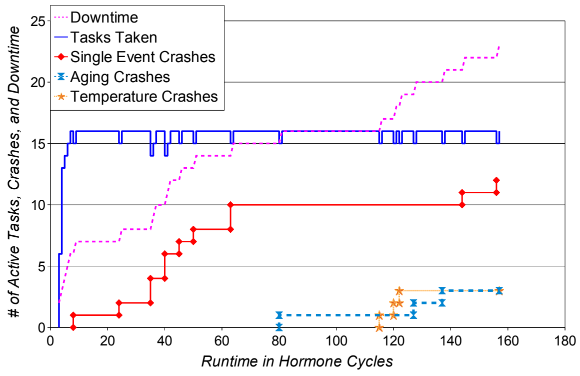


Figure 6. Behavior of 16 AHS resources with reactive self-healing

the resulting hormone imbalance, this task is reallocated at cycle 8 bringing the system back online. The same happens for the following failures. Every time a task is crashed by a failure, the hormone system compensates this event by task reallocation or reassignment*. Beginning at hormone cycle 81, aging and temperature based crashes occur as well and are compensated. Even so the system downtime still increases due to these crashes, it increase much slower and the system always comes back online, as long as enough cores are available to take all tasks (either by other cores or regeneration of the crashed cores). The downtime ratio is $23/158 = 0.14 = 14\%$ at hormone cycle 158.

The behavior shown is a pure reactive self-healing process. To allow proactive failure handling, additional suppressors can be applied. By monitoring, e.g., failures and temperature, suppressors can be emitted for resources with high temperature or failure count. This favors reliable and cool resources in comparison to unreliable and hot ones. The

*In case of a permanent failure, the task is reallocated to another core. In case of a transient failure, the task might be reallocated to another core or reasigned to the same core. This depends on the hormone balance induced by the current task distribution.

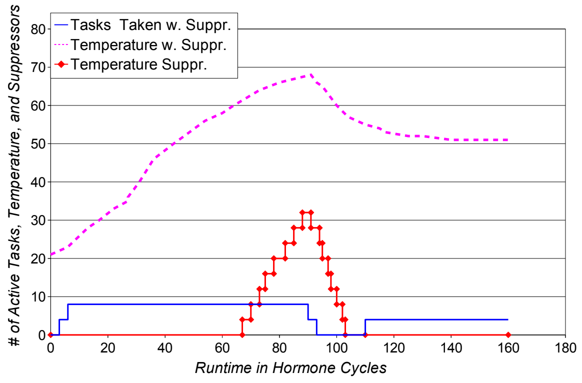


Figure 7. Temperature proactive task behavior of a local AHS resource

major effect of this proactive reallocation behavior is shown in Figure 7, where temperature suppressors are proportional emitted to the raising temperature load. This successively reduces the suitability (eagervalue) of the core until tasks get migrated to other cores. As a result of the sinking workload, the temperature and the temperature suppressor are declining. The temperature and load are balancing at a reasonable level. The proactive task assignment increases the reliability by preventing cores from total failing, using a rebalance of the workload via task distribution on different cores. In conclusion, the evaluation shows the major advantage of the hormone cycle for task assignment. Comparing with the results of the first simulation the system achieves an excellent enhancement in downtime optimization thus improving dependability in a significant way.

VI. Related Work

Currently, there exist only a few approaches of task assignment in Middleware on future CMP based mixed-signal SoCs. The approach of [11] traces the assumption of a reliable multi-layered MPSoC architecture against thermal issues due to increasing task processing. This concept internalizes a proactive task migration on cooler resources by an distributed hierarchical agent-based system. Like our approach the system is widely capable handling single point of failure within the exception of high-level agent errors, which is resisted by hardware redundancy. Furthermore, the system is restricted to thermal management domain on none intermixed circuit technology only.

Another approach using also the assumption of an agent distributed system is shown in [1]. The author presents an algorithmic schedule for task distribution on a processing grid. Against our solution, this approach uses centralized elements, so called Group Manager's (GM), responsible for the internal controlling in a clustered bunch of tasks. Unless a single GM-instance fails, there is no possibility for restoring the corresponding group information, which implies a single point of failure occurrence.

In [9], two algorithms for task scheduling on heterogeneous systems are presented. Within both schedulings task priorities are computed statically or dynamically. The first algorithm, Fast Critical Path (FCP) uses dynamic increase of priorities to ensure time constraints are kept. The second approach, Fast Load Balancing (FLB) uses a workload balanced task assignment to ensure every processor core will be used. In contrast to our approach, both algorithms do not regard crashing of cores and tasks.

Heiss and Schmitz [6] presented another approach for a decentralized load balanced task assignment. The authors consider a physical model where tasks are represented as particles, which are influenced by forces like, e.g., load balancing force (issued by the load potential of cores) or communication force (intensities between tasks). Depending on the resultant force action tasks are assigned to corresponding cores.

Other approaches for workload balanced assignment are presented in [5][10]. In summary, none of the concepts above covers a decentralized assignment of tasks including the spectrum of self-X properties and real-time conditions like our approach.

VII. Conclusion and Future Work

In summary, this paper presented an approach of a self-adaptive organic Middleware solution for highly dependable SoCs. The organic Middleware is represented by the AHS - an artificial hormone system providing a decentralized self-organized assignment of tasks on processing resources.

In prospection and future work, the whole project investigates further analysis of the reliability especially in a field of real SoC computing, facing timing behavior on real prototypes leaving the accomplished sector of simulation behind. Therefore, we intend the development of a highly dependable mixed signal SoC. The prior AHS is used in combination with a generalized core and task concept to assign software and hardware tasks to suitable mixed signal resources, so called processing elements (PEs). Every PE represents a specific SoC function, which can be any type of processor core like timer, memory, analog or digital PE. Since different hardware tasks and analog PEs are involved, the hormone controlled concept is extended for time continuous processing of analog hormone signals. The interconnection and communication throughout both systems is realized by a common inter-core network with redundant interfaces. Overall the interaction of the resultant reliable mixed signal SoC has to be achieved facing a real demonstrator application settled in the predestined area of automotive driven assistance control. For this, the SoC is admitted controlling a complex model helicopter.

References

[1] L. F. Bittencourt, E. R. M. Madeira, F. R. L. Cicerre, and L. E. Buzato. A path clustering heuristic for scheduling task graphs onto a grid. In *3rd International Workshop on Middleware for Grid Computing (MGC05)*, Grenoble, France, 2005.

[2] Uwe Brinkschulte, Mathias Pacher, and Alexander von Renteln. Towards an artificial hormone system for self-organizing real-time task allocation. *5th IFIP Workshop on Software Technologies for Future Embedded & Ubiquitous Systems (SEUS)*, pages 339–347, 2007.

[3] Uwe Brinkschulte, Mathias Pacher, and Alexander von Renteln. An Artificial Hormone System for Self-Organizing Real-Time Task Allocation in Organic Middleware. In *Organic Computing*. Springer, 2008.

[4] Uwe Brinkschulte, Mathias Pacher, and Alexander von Renteln. An artificial hormone system for self-organizing real-time task allocation. *Springer*, 2008.

[5] Jorge Finke, Kevin M. Passino, and Andrew Sparks. Stable task load balancing strategies for cooperative control of networked autonomous air vehicles. In *IEEE Transactions on Control Systems Technology*, volume 14, pages 789– 803, 2006.

[6] Hans-Ulrich Heiss and Michael Schmitz. Decentralized dynamic load balancing: The particles approach. In *Proc. 8th Int. Symp. on Computer and Information Sciences*, Istanbul, Turkey, November 1993.

[7] Paul (IBM Research) Horn. Autonomic computing manifesto: IBM's perspective on the state of information technology, October 2001.

[8] Christian Mueller-Schloer, Christoph von der Malsburg, and Rolf P. Wuertz. Organic computing. *Informatik Spektrum*, 27(4):332–336, 2004.

[9] Andrei Radulescu and Arjan J. C. van Gemund. Fast and effective task scheduling in heterogeneous systems. In *IEEE Computer - 9th Heterogeneous Computing Workshop*, Cancun, Mexico, 2000.

[10] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A simple load balancing scheme for task allocation in parallel machines. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, 1991.

[11] Thomas Ebi, Holm Rauchfuss, Jörg Henkel and Andreas Herkersdorf. Agent-based Thermal Management using Real-Time I/O Communication Relocation for 3D Many-Cores. *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation (PATMOS), Madrid, Spain*, September 26-29 2006.

[12] Alexander von Renteln and Uwe Brinkschulte. Reliablity of an Artificial Hormone System with Self-X Properties. In *Parallel and Distributed Computing and Systems*, Cambridge, Massachusetts, USA, November 19 - 21 2007.

[13] Alexander von Renteln, Michael Weiss, and Uwe Brinkschulte. Examining Task Distribution by an Artificial Hormone System Based Middleware. In *11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*, Orlando, Florida, USA, May, 5-7 2008.

FUZZBUSTER: A System for Self-Adaptive Immunity from Cyber Threats

David J. Musliner, Jeffrey M. Rye, Dan Thomsen, David D. McDonald, Mark H. Burstein
Smart Information Flow Technologies (SIFT)
Minneapolis, MN, USA
Email: {musliner, rye, dthomsen, dmcdonald, burstein}@sift.net

Paul Robertson
Dynamic Object Language Labs
Boston, MA, USA
Email: paulr@dollabs.com

***Abstract*—Today's computer systems are under relentless attack from cyber attackers armed with sophisticated vulnerability search and exploit development toolkits. To protect against such threats, we are developing FUZZBUSTER, an automated system that provides adaptive immunity against a wide variety of cyber threats. FUZZBUSTER reacts to observed attacks and proactively searches for never-before-seen vulnerabilities. FUZZBUSTER uses a suite of fuzz testing and vulnerability assessment tools to find or verify the existence of vulnerabilities. Then FUZZBUSTER conducts additional tests to characterize the extent of the vulnerability, identifying ways it can be triggered. After characterizing a vulnerability, FUZZBUSTER synthesizes and applies an adaptation to prevent future exploits.**

***Keywords*-self-adaptive immunity, cyber-security, fuzz-testing.**

I. INTRODUCTION

Modern computer systems face constant attack from sophisticated adversaries, and the number of cyber-intrusions increases every year [1], [2]. Cyber-attackers use numerous vulnerability scanning tools that automatically probe target software systems for a wide array of vulnerabilities. For example, attackers use fuzz-testing tools (such as Peach and SPIKE) that try to crash target applications, and SQL injection tools (such as sqlmap and havij) that attempt to manipulate the contents of databases. Upon discovering a potential vulnerability, attackers use powerful exploit development toolkits (such as Metasploit and Inguma) to quickly craft exploits that take advantage of identified vulnerabilities.

Under DARPA's Clean-slate design of Resilient, Adaptive, Survivable Hosts (CRASH) program, we are developing FUZZBUSTER to provide adaptive immunity from these and other cyber-threats. FUZZBUSTER provides long-term immunity against both observed and novel (zero-day) cyber-attacks.

As shown in Figure 1, FUZZBUSTER operates *proactively* to find vulnerabilities before they can be exploited, and *reactively* to address exploits observed "in the wild." FUZZBUSTER directs the execution of custom and off-the-shelf *fuzz-testing* tools to find and characterize vulnerabilities. Fuzz-testing tools find software vulnerabilities by exploring millions of semi-random inputs to a program. Given time and expert guidance, fuzz-testing has proven effective at finding a wide variety of software flaws, including defects that account for the most severe security problems [3].

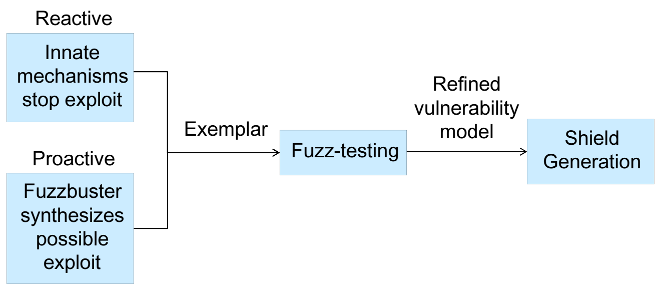

Reactive
Innate mechanisms stop exploit
Proactive
Fuzzbuster synthesizes possible exploit
Exemplar
Fuzz-testing
Refined vulnerability model
Shield Generation

Figure 1. When reacting to a fault, FUZZBUSTER creates an exemplar test case that reflects the environment and inputs at the time of the observed fault. During proactive exploration, FUZZBUSTER synthesizes exemplar test cases that could lead to a fault.

FUZZBUSTER uses fuzz-testing tools to find and characterize vulnerabilities, determining what inputs to a program can cause a fault. FUZZBUSTER then synthesizes defenses to shield or repair the flaw, protecting against entire classes of exploits that may be encountered in the future.

In this paper, we describe our rapidly-evolving implementation of the FUZZBUSTER architecture, and present some preliminary results.

II. ARCHITECTURE

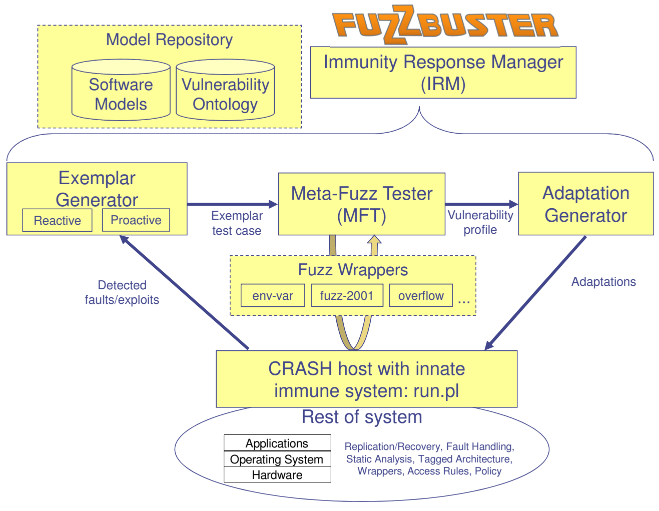


Figure 2. FUZZBUSTER's IRM guides its efforts to automatically find, refine, and adaptively shield vulnerabilities.

Figure 2 illustrates FUZZBUSTER's major components and how they interact to provide adaptive immunity. FUZZBUSTER uses both proactive and reactive exploration to identify (and then shield) vulnerabilities in a CRASH host. For each vulnerability, FUZZBUSTER creates a *vulnerability profile* representing the nature of the vulnerability, including what ranges of inputs lead to the vulnerability. These vulnerability profiles represent as much of the vulnerability as FUZZBUSTER can identify. After constructing a vulnerability profile, FUZZBUSTER creates and applies an adaptation that prevents future exploits of the vulnerability.

Processing begins with the Exemplar Generator creating an exemplar test case. The Exemplar Generator may create an exemplar in response to a fault notification from the CRASH innate immune system or in response to an instruction from the Immunity Response Manager (IRM) to initiate proactive exploration. At some point, the IRM determines that looking for vulnerabilities relating to a particular exemplar test case is the next highest priority activity, and the IRM assigns this activity to the Meta-Fuzz Tester (MFT). Based on the nature and attributes of the exemplar test case, the MFT chooses a fuzz-testing tool to search for or assess vulnerabilities associated with the exemplar test case. Each fuzz-testing tool refines a vulnerability profile based on the results of its exploration. The MFT may use multiple fuzz-testing tools to construct as complete a vulnerability profile as possible given the available time or resources. For each vulnerability profile, the Adaptation Generator creates one or more candidate adaptations to protect the system against being exploited via the vulnerability. When appropriate, the IRM directs the Adaptation Generator to verify and then subsequently apply these patches to the CRASH host. Since fuzz-testing and patch verification both run tests that may require significant time or resources to complete, the IRM is intended to balance the priorities of these operations with the available resources on the system, to minimize FUZZBUSTER's impact on system performance.

When an actual exploit or flaw is encountered and trapped by the CRASH innate immune system, FUZZBUSTER responds reactively. In our design, the IRM puts a high priority on responding to a live exploit, and may immediately choose to use the Adaptation Generator to synthesize a customized adaptation to shield the application while also engaging the MFT to refine the vulnerability profile. FUZZBUSTER may be conservative when reacting to an exploit, initially disabling useful features of the subject software while disabling the vulnerability. As the tests yield additional information, FUZZBUSTER revises the adaptation to relax (or tighten) the behavior restrictions it enforces. In this way, FUZZBUSTER acts as a self-protecting, self-regenerative system, initially clamping down on security and limiting functions when attacked, and then gradually relaxing limits and restoring functions as it gets a better picture of the vulnerabilities that are being exploited.

A. Infrastructure

1) GBBopen: Our FUZZBUSTER implementation is built within the GBBopen blackboard system [4], [5], which supports object-oriented data storage and event-triggered procedural code. The functional components shown in Figure 2 are implemented as blackboard Knowledge Sources (KSs) that respond to events on blackboard objects representing the ongoing tasks and results.

For instance, exemplar objects are used to describe cases raised by the immune system and cases where proactive exploration is suggested. Vulnerability profile objects capture a progressively-refined model of the set of situations that leading to security warnings for a particular software system.

During FUZZBUSTER's exploration, it often needs to initiate resource-intensive testing tasks or perform operations that change the behavior of the CRASH host. To prevent these activities from executing concurrently, and thus interfering with each other, FUZZBUSTER defines a set of task objects that are managed by the IRM. The processing of these tasks is started and stopped by the IRM as necessary to ensure effective operation of the system and to prevent conflicts between concurrent activities. The KSs that perform the processing have specific code to manage task status and their own work. In the near future, we will replace this *ad hoc* mechanism with a more automated and rigorous meta-control scheme. The more complete meta-control scheme will control the starting and stopping of tasks to ensure consistent and correct behavior, while easing the effort required to specify the desired properties and interactions between them.

2) Interface to CRASH Host (`run.pl`): FUZZBUSTER is designed to run in the context of a CRASH host whose innate immune system provides alerts to violations that may indicate vulnerabilities. Since a physical instantiation of the CRASH host is not yet available, FUZZBUSTER defines a proxy that serves as a stand-in. The proxy is currently implemented as a Perl script that provides key CRASH functionality on existing systems. In particular, the proxy mimics the CRASH innate immune system by identifying and reporting certain classes of faults. The proxy also provides an adaptation mechanism that allows FUZZBUSTER to modify the environment and inputs of executing programs.

B. Exemplar Generator

The Exemplar Generator captures relevant inputs and environmental aspects of an observed or suspected vulnerability as an exemplar test case. Ideally, an exemplar test case contains all of the information required to generate a repeatable test case for the MFT. However, since it is not always possible to record every relevant piece of information, and knowing the relevant bits is impossible in the context of proactive exploration, the MFT treats an exemplar test case as a starting point for exploration. An exemplar test case may also indicate that it pertains to a frequently-observed attack or applies to a mission-critical software system, which the IRM will use to set a suitably high priority on the discovery of the vulnerability and the implementation of a defense.

When the CRASH proxy `run.pl` detects a fault, it sends the Exemplar Generator a description of the environment and inputs that triggered the fault. The description includes each environment variable and its value, the path of the command, the command line arguments, and the content passed thru open streams including standard-input.

For proactive exploration, the Exemplar Generator also synthesizes exemplar test cases from application models. FUZZBUSTER stores models of applications that include the absolute path to the command, a specification of the allowed (or expected) command line arguments, and a flag indicating whether the application accepts input via standard-input. The Exemplar Generator translates these application models into exemplar test cases by choosing specific command line arguments or inputs.

C. Meta-Fuzz Tester

The Immunity Response Manager invokes the MFT to conduct an analysis of subsystem or protocol vulnerabilities, focused by the exemplar test case and limited by some computing resource constraints (initially, just execution time). The MFT attempts to identify the specific cause of an observed defect, or probe for a latent vulnerability in the case of proactive analysis. Starting from the exemplar test case, the MFT constructs a belief state describing the vulnerabilities that could be present. Then, as long as the MFT has remaining resources, it chooses a fuzz-testing tool and uses it to try to gain more information about the possible vulnerability. This analysis culminates in a vulnerability profile describing the observed aspects of the vulnerability and providing a basis for the Adaptation Generator to generate an adaptation that protects the system. The choice of fuzz-testing tool should be guided by a “performance profile” model of each tool’s capabilities, in terms of what types of vulnerabilities they can detect and how long they may take. Our early experiments, discussed below, will help develop those performance profiles. In the meantime, our preliminary implementation uses a simpler method to allocate effort to different fuzz-testing tools.

To facilitate the integration of diverse fuzz-testing tools, FUZZBUSTER defines a *fuzz-tool wrapper* interface to each tool, providing a common API for controlling tool execution. Each fuzz-tool wrapper defines an action that may be taken by the MFT. Moreover, each wrapper interprets the results of execution, updating the vulnerability profile with additional information. Fuzz-tool wrappers provide a simple mechanism for FUZZBUSTER to incorporate dumb or smart tools, with or without knowledge of the internals of the test object.

D. Adaptation Generator

FUZZBUSTER’s Adaptation Generator improves system security by creating and applying custom adaptations that prevent exploitation of the flaws characterized by vulnerability profiles. The Adaptation Generator uses a variety of adaptive techniques, making the choice between them based on an adaptation’s needs and the facilities available for the relevant input channels. Our design anticipates that adaptations could be defined at any level in the system, from an atomic instruction, to a function call, to a high-level function of an application. Our initial implementation operates only at the application-input level, using the facilities provided by the `run.pl` CRASH proxy.

To safely adapt a live system, the Adaptation Generator follows two core principles. First, adaptations only restrict or reduce capability or privilege. Second, adaptations do not disable key functionality. To enforce the second principle, FUZZBUSTER will capture a set of test cases during vulnerability analysis. Some of these tests will trigger the vulnerability and others will exercise the vulnerable application without triggering the vulnerability. These tests will be used during adaptation creation to verify that an adaptation successfully prevents the vulnerability without otherwise changing the results or behavior of the vulnerable application. Once an adaptation is applied to the CRASH system, the tests will be added to a regression suite that FUZZBUSTER will use to ensure that future patches do not conflict with or invalidate existing adaptations.

The Adaptation Generator performs the following actions on adaptations:

- **Create —** Search for adaptations that make the best trade-off between performance, functional impact, and security.
- **Verify —** Execute recorded test cases to ensure that an adaptation prevents exploitation of a vulnerability without otherwise affecting execution.
- **Apply —** Apply adaptations to the system to prevent exploitation of vulnerabilities.
- **Revoke —** Remove previously applied adaptations from the system because they are no longer desirable, due to external software updates, more comprehensive adaptations, or to improve performance.

When creating an adaptation, the Adaptation Generator maps the constraints in the vulnerability profile to a set of actions that the adaptation can take to prevent the fault. FUZZBUSTER’s initial set of actions includes “remove,” “modify,” “truncate,” and “filter”. An adaptation using the remove action completely removes the fault-inducing input, for instance unsetting an environmental variable. An adaptation using the modify action performs an arbitrary modification of the input, for example replacing the value with another one. The truncate and filter actions apply common modifications to inputs. Truncate reduces the size of the input channel to a specific threshold, for example shortening the length of an argument to prevent a buffer overflow. The filter action replaces specific substrings in the input channel. The Adaptation Generator examines the vulnerability profile to derive parameters for these actions, such as the target length for truncation or the content to remove.

When instructed by the IRM, the Adaptation Generator verifies an adaptation by temporarily applying the adaptation to the system and running the accumulated test-cases. Once an adaptation passes verification, the IRM may instruct the Adaptation Generator to apply it to the system, thus preventing a vulnerability from being exploited. An adaptation fails verification if it changes the behavior for non-fault-inducing inputs or if it fails to prevent a fault.

E. Immunity Response Manager

The IRM oversees and manages FUZZBUSTER's adaptive immunity processes, ensuring that FUZZBUSTER's proactive and reactive protection functions are effective, while avoiding undue burden on the resources of the protected system.

The IRM's chief roles include initiating proactive vulnerability exploration, assigning test priorities, and tasking the MFT and Adaptation Generator. Across these activities, the IRM controls the system by creating, assigning, and pausing tasks. Each task specifies a unit of work to be performed by a component in the system. FUZZBUSTER defines tasks for exploring an exemplar test case, verifying a patch, applying a patch to the system, and revoking a patch. By controlling which tasks are active, the IRM controls the balance between proactive and reactive testing, decides when to allocate resources to verifying that patches are acceptable, and controls when FUZZBUSTER modifies the system.

Our initial implementation of the IRM uses a hand-coded, static prioritization scheme that ranks tasks based on their order of arrival. This initial implementation ensures that FUZZBUSTER eventually explores all exemplar test cases and attempts to apply adaptations for all identified vulnerability profiles. In the future, the IRM will evolve into an MDP-based meta-controller similar to the approach described in [6].

III. EXPERIMENTAL RESULTS

With the first version of each FUZZBUSTER module now functional, we have conducted numerous small tests and one significant series of long experiments. In those experiments, we used FUZZBUSTER to proactively search for vulnerabilities in a set of 53 command-line utilities. We ran the exploration on a Debian VM and a laptop running OS X; both systems were fully patched at the time of the experiment. FUZZBUSTER used a wrapper around Barton Miller's fuzz-testing tool to generate random byte sequences to feed to the programs being tested. For each trial, we configured FUZZBUSTER to use a specific set of options to the fuzz-testing tool, varying:

- whether the inputs could contain non-printable characters or not,
- whether the inputs could contain null characters or not,
- the initial seed to use for randomization, and
- the length of the input.

Each trial ran for a fixed period of time (usually 20 seconds), or until FUZZBUSTER found a fault. We relied on our CRASH stand-in (`run.pl`) to identify faults. For the purposes of this experiment, we identified faults as program crashes (abnormal exits, such as from segmentation faults).

FUZZBUSTER ran 3,380 trials in just over 18 hours, encountering 49 faults. Fifteen of those faults were "duplicates" caused by the same input as another trial but with additional, unnecessary, content at the end. For example, we found a fault in `tcsh` with a 1,000 byte input created with both printable and non-printable characters, no nulls, and a seed of 1,002; that same fault was subsequently encountered using a 10,000 byte input created with the same parameters. Another eleven faults differed only in that one fault was caused by feeding an input string to standard-input and the other was caused by feeding the same string via a file argument. The remaining 23 faults correspond to unique crashes in five programs: `a2p`, `dc`, `indent`, `tcsh`, and `troff`. FUZZBUSTER considers these to be unique vulnerabilities, as the inputs have unique combinations of printable/non-printable characters, presence of null characters, and seeds. However, we recognize that it is probable that these inputs are triggering fewer than 23 software problems, perhaps as few as five (one per program). Even if several of these faults stem from a single vulnerability, the full FUZZBUSTER will identify and shield the common vulnerability, thus protecting the system against the original and related faults.

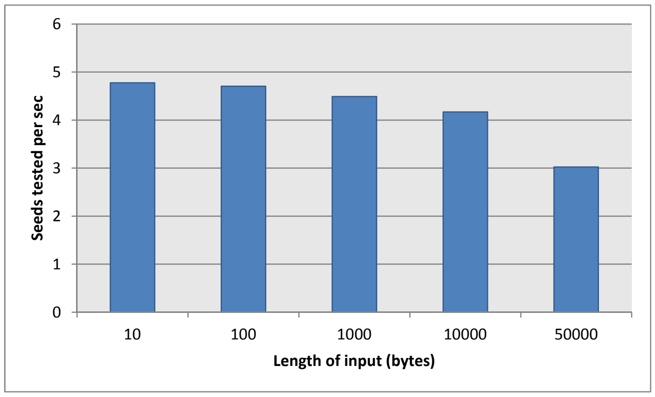


Figure 3. FUZZBUSTER executes 4.7 test cases per second when the inputs are short (10 to 100 bytes). The testing rate decreases with larger test inputs, falling to 3.0 tests per second when the inputs are 50,000 bytes long.

We configured FUZZBUSTER to repeat the proactive exploration numerous times, varying each of the conditions.

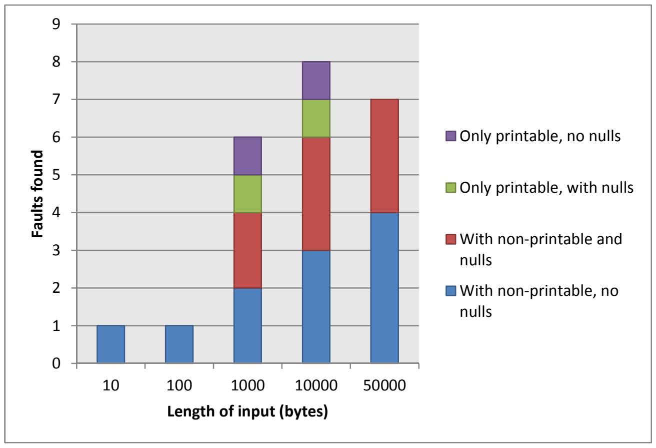


Figure 4. FUZZBUSTER requires larger input files to uncover all the faults encountered during the experiment. Inputs with non-printable characters led to discovery of 19 faults.

We tested with inputs of length 10, 100, 1,000, 10,000, and 50,000 bytes. As shown in Figure 3, running with the larger inputs modestly reduces the rate at which FUZZBUSTER runs individual test cases. Figure 4 shows that FUZZBUSTER usually needs inputs of at least 1,000 bytes to trigger the faults. Since we removed faults that were duplicates except for size, Figure 4 illustrates unique faults discovered at each size. Thus, FUZZBUSTER requires inputs with a length of 50,000 bytes to discover all of the faults in the test. From this graph we can also see that testing with non-printable characters is more effective than testing without, accounting for 19 out of the 23 faults (82.6%). Inputs containing null characters account of 10 faults and inputs without nulls account for 13.

While this suggests that FUZZBUSTER should focus on inputs containing non-printable characters, two applications (`a2p` and `indent`) *only* faulted when the inputs consisted entirely of printable characters. Moreover, our experiment encountered only a single fault in each of these applications. Thus, we can see the benefit of the MFT producing multiple actions for each fuzz-tool wrapper.

As shown in Figure 5, FUZZBUSTER discovers faults much more frequently using larger inputs. This graph illustrates how much more effective it is to test with larger inputs, despite the decrease in the number of test cases per second. We would like the MFT to try to optimize its use of limited fuzz-testing resources, so we're also interested in estimating how long FUZZBUSTER should run a test configuration before giving up and trying another fuzz-tool wrapper or abandoning the task. We can begin to answer this by examining how many inputs FUZZBUSTER tried before finding one that caused a fault. Figure 6 shows how many non-faulting seeds were tried in each trial that found a fault. The graph shows that it took, at most, 53 test-cases for inputs of 10,000 bytes and 50 test-cases for inputs of 1,000 bytes before finding the fault. Given the test rates from Figure 3, this shows that each of the successful tests took less than fifteen seconds, for larger inputs. While the precise test duration will vary according to the type of software being tested, these values provide a good starting point for our upcoming MDP-based Immunity Response Manager meta-controller.

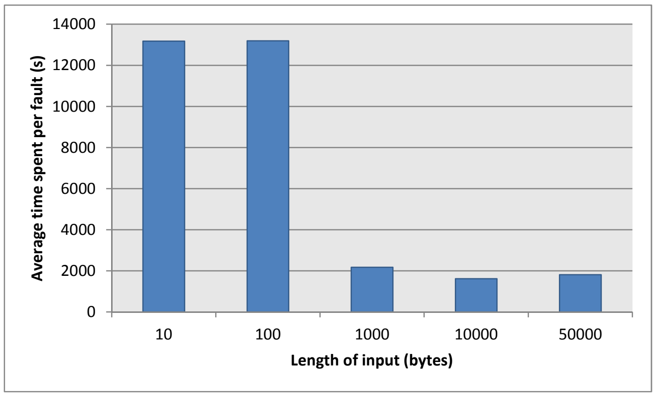


Figure 5. Even though short inputs result in faster test execution, due to the relative rarity of faults identified by short inputs, the average time spent searching per fault is much higher.

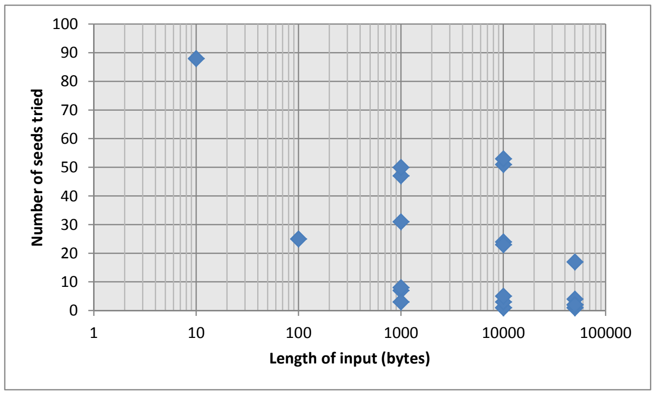


Figure 6. Number of seeds tested in a trial before finding one that induces a fault.

IV. RELATED WORK

As previously noted, the FUZZBUSTER approach has roots in fuzz-testing, a term first coined in 1988 in the context of software security analysis [7]. It refers to invalid, random, or unexpected data that is deliberately provided as program input in order to identify defects. Fuzz-testers and the closely related "fault injectors" are good at finding buffer overflow, XSS, denial of service (DoS), SQL Injection, and format string bugs. They are generally not highly effective in finding vulnerabilities that do not cause program crashes, *e.g.*, encryption flaws and information disclosure vulnerabilities [8]. Moreover, existing fuzz-testing tools tend to rely significantly on expert user oversight, testing refinement, and decision-making in responding to identified vulnerabilities.

FUZZBUSTER is designed both to augment the power of fuzz-testing and to address some of its key limitations. FUZZBUSTER fully automates the process of identifying seeds for fuzz-testing, guides the use of fuzz-testing to develop general vulnerability profiles, and automates the synthesis of defenses for identified vulnerabilities.

To date, several research groups have created specialized self-adaptive systems for protecting software applications. For example, both AWDRAT [9] and PMOP [10] used dynamically-programmed wrappers to compare program activities against hand-generated models, detecting attacks and blocking them or adaptively selecting application methods to avoid damage or compromises.

The CORTEX system [11] used a different approach, placing a dynamically-programmed proxy in front of a replicated database server and using active experimentation based on learned (not hand-coded) models to diagnose new system vulnerabilities and protect against novel attacks.

While these systems demonstrated the feasibility of the self-adaptive, self-regenerative software concept, they are closely tailored to specific applications and specific representations of program behavior. FUZZBUSTER provides a general approach to adaptive immunity that is not limited to a single class of application. FUZZBUSTER does not require detailed system models, but will work from high-level descriptions of component interactions, such as APIs or contracts. Furthermore, FUZZBUSTER's proactive use of intelligent, automatic fuzz-testing identifies possible vulnerabilities before they can be exploited.

V. CONCLUSION AND FUTURE WORK

FUZZBUSTER is intended to augment and eventually outmode various post-exploit security tools such as virus scanners. Rather than scanning a computer all night to see if it has been compromised by an exploit, FUZZBUSTER will scan for vulnerable software and repair or shield it. Our preliminary experiments have shown that there are still many such vulnerabilities to be found, even on heavily used software in mature systems. As we extend FUZZBUSTER to address more complex applications with more forms of input, we expect that FUZZBUSTER will find vulnerabilities even more frequently. We hope that FUZZBUSTER will play a crucial role in proactively finding and eliminating vulnerabilities, making fuzz-testing no longer an effective strategy for cyber-attackers.

ACKNOWLEDGMENTS

This work was supported by DARPA and Air Force Research Laboratory under contract FA8650-10-C-7087. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

REFERENCES

[1] T. Kellerman, "Cyber-threat proliferation: Today's truly pervasive global epidemic," *Security Privacy, IEEE*, vol. 8, no. 3, pp. 70 –73, May-June 2010.

[2] G. C. Wilshusen, "Cyber threats and vulnerabilities place federal systems at risk: Testimony before the subcommittee on government management, organization and procurement," United States Government Accountability Office, Tech. Rep., May 2009.

[3] "Automated penetration testing with white-box fuzzing," 2008. [Online]. Available: http://msdn.microsoft.com/en-us/library/cc162782.aspx#Fuzzing_topic1

[4] D. D. Corkill, "Countdown to success: dynamic objects, gbb, and radarsat-1," *Commun. ACM*, vol. 40, pp. 48–58, May 1997.

[5] ——, "Blackboard systems," *AI expert*, vol. 6, no. 9, pp. 40–47, 1991.

[6] D. J. Musliner, R. P. Goldman, and K. D. Krebsbach, "Deliberation scheduling strategies for adaptive mission planning in real-time environments," in *Proc. Third International Workshop on Self Adaptive Software*, 2003.

[7] B. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, December 1990.

[8] C. Anley, J. Heasman, F. Linder, and G. Richarte, *The Shellcoder's Handbook: Discovering and Exploiting Security Holes, 2nd Ed.* John Wiley & Sons, 2007, ch. The art of fuzzing.

[9] H. Shrobe, R. Laddaga, B. Balzer, N. Goldman, D. Wile, M. Tallis, T. Hollebeek, and A. Egyed, "AWDRAT: a cognitive middleware system for information survivability," *AI Magazine*, vol. 28, no. 3, p. 73, 2007.

[10] H. Shrobe, R. Laddaga, B. Balzer *et al.*, "Self-Adaptive systems for information survivability: PMOP and AWDRAT," in *Proc. First Int'l Conf. on Self-Adaptive and Self-Organizing Systems*, 2007, pp. 332–335.

[11] "Cortex: Mission-aware cognitive self-regeneration technology," Final Report, US Air Force Research Laboratories Contract Number FA8750-04-C-0253, March 2006.