Implementation of a Generic Autonomic Framework

Radu Calinescu
Computing Laboratory, University of Oxford
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
Radu.Calinescu@comlab.ox.ac.uk

***Abstract*—Based on insights from the implementation of commercial products for data-centre resource management, we identified key challenges in the development of cost-effective autonomic solutions, and best practices for overcoming these challenges. In a related paper, we proposed a generic autonomic framework that complies with these best practices, and suggested ways in which existing technologies could be used to realise this framework. In this paper, we describe the actual implementation of our autonomic framework as a service-oriented architecture, and we show how the universal policy engine at its core can be configured to manage the allocation of server capacity to services of different priorities. This case study demonstrates the effectiveness of our generic approach to autonomic solution development in an area of great interest for commercial data centres, research laboratories and application service providers.**

I. Introduction

Today's economy is characterised by revolutionary transformations to the way in which Information and Communication Technologies (ICT) are used to conduct business and research and to provide services in all sectors of the society [1]. The ability to accomplish more, faster and on a broader scale through expert use of ICT is at the core of today's scientific discoveries, newly emerged electronic services and everyday life. Due to unprecedented advances in ICT, business needs are attended to by ever more sophisticated and feature-rich systems and systems of systems [2]. Notwithstanding the benefits of such developments, the spiralling complexity of these systems led to unsustainable increases in the cost and expertise required for their management. In an attempt to address this problem, autonomic computing was proposed as a technology for delegating the management of complex systems to the machines themselves [3]. Over the past few years, significant progress has been made in defining what autonomic systems should look like [4], [5], [6] and in successfully implementing domain-specific autonomic solutions [7], [8], [9], [10].

Several such solutions [8] were implemented using a commercial autonomic system for policy-based resource management that was co-developed by the author [11]. Multiple challenges were encountered in this work, including [12]:

- The lack of standardisation in ICT resource interfaces. Despite an increasing trend to add management interfaces to new ICT components and devices, and to make existing interfaces public, autonomic system development is hindered by the broad diversity of architectures and technologies these interfaces are based upon.
- The tendency to hardcode ICT resource metadata within the control component of the autonomic system. Management frameworks are often intended for handling particular types of resources, and the properties of these resource types are hardcoded in the control element of the system. With careful design, complex systems consisting of supported resources can be successfully managed, however adding in support for additional types of resources cannot be achieved in a cost-effective way.
- The complexity of business policies. Policy design for autonomic solutions is a complex, error-prone and iterative process. A high level of expertise is required to devise and fine-tune the set of policies for a complete autonomic solution.
- The high scalability expectations. As simple, small ICT systems are easy to manage by low-skilled human operators, autonomic solutions are required in areas where the systems to manage are complex and comprise large numbers of resources.

Based on best practices identified while addressing these challenges [12], we proposed a generic autonomic framework for developing cost-effective autonomic solutions, and suggested ways in which existing technologies could be used to realise this framework [13].

In this paper, we describe the actual implementation of our autonomic framework as a service-oriented architecture (SOA), and we show how the universal policy engine at its core can be configured to manage the allocation of server capacity to services. A SOA solution was chosen as the target for the prototype implementation of the framework in order to leverage web service technology benefits such as platform independence, loose coupling and security support—all of which are of uttermost importance in an autonomic solution.

The choice of server capacity allocation for our initial case study was motivated by the importance that this real-world application has had since the release of server-level capacity control APIs such as [14], [15]. Additionally, our prior experience with data-centre resource management [11] helped significantly during the implementation of the solution, and in the interpretation of the case study results.

II. Related work

The autonomic infrastructure proposed in [16] is retrofitting autonomic functionality onto legacy systems by using *sensors* to collect resource data, *gauges* to interpret these data and *controllers* to decide the "adaptations" to be enforced on

the managed systems through *effectors*. This infrastructure was successfully used to monitor, analyse and control legacy systems in applications such as spam detection, instant messaging quality-of-service management and load balancing for geographical information systems [17]. Our generic autonomic framework addresses several key areas that are not supported by the approach in [16], [17]. By using a system model for the configuration of its policy engine, our architecture can be used for the autonomic management of heterogeneous types of resources. Moreover, our managed system can include resources beyond the software components handled by the infrastructure in [16].

In [18], the authors define an autonomic architecture meta-model that extends IBM's autonomic computing blueprint [19], and use a model-driven process to partly automate the generation of instances of this meta-model. Each instance is a special-purpose *organic computing system* that can handle the use cases defined by the model used for its generation. Our general-purpose autonomic architecture eliminates the need for the 19-step generation process described in [18] by using a universal policy engine that can be dynamically repurposed to handle any use cases encoded within its system model and policy set.

A number of other projects have investigated isolated aspects related to the development of autonomic systems out of non-autonomic components. Some of these projects addressed the standardisation of the policy information model, with the Policy Core Information Model [20] representing the most prominent outcome of this work. Recent efforts such as Oasis' Web Services Distributed Management (WSDM) project were directed at the standardisation of the interfaces through which the manageability of a resource is made available to other applications [21]. An integrated development environment for the implementation of WSDM-compliant interfaces is currently available from IBM [22].

In a different area, expression languages were proposed for the specification of policy conditions and actions, and used to implement a range of policies [23], [24], [25], [26]. In addition to the development of standards and technologies, complete autonomic computing solutions have been produced recently [14], [11], [15], typically for the management of specific systems, and with limited ability to function in different scenarios from those they were originally intended for.

III. Overview of the Generic Autonomic Framework

Fig. 1 depicts the general-purpose autonomic architecture used by our framework. This architecture, originally introduced in [13], is designed around a reconfigurable, model-driven *policy engine* that organises heterogeneous collections of legacy and autonomic ICT resources into self-managing systems. In order to support the great diversity of existing and future ICT resources encountered in real-world applications, the policy engine is capable of handling resources whose types are unknown at implementation and deployment time. This unique (re)purposing capability is achieved through runtime

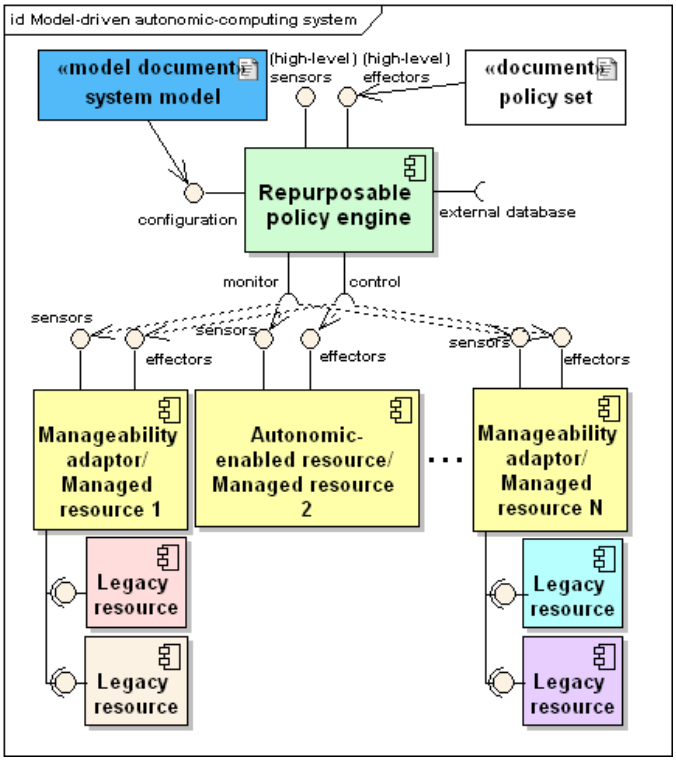

Fig. 1. General-purpose autonomic architecture.

configuration: a comprehensive specification (or *model*) of the system to be managed is supplied to the policy engine for this purpose. The engine will then implement the high-level goals given by a set of user-specified policies that are expressed in a declarative language, and which make reference to the resources defined in the system model. The policy engine performs this task by monitoring the state of the managed resources and controlling their configuration parameters accordingly, while resource-specific *manageability adaptors* ensure that this is achieved without any modification to existing ICT resources.

The generality of the architecture allows the implementation of the engine using different technologies, e.g., as a software agent running on a data-centre server or a physical device incorporated into a factory automation equipment or a mobile phone. A full description of this novel autonomic framework is provided in [13].

IV. Prototype Policy Engine

The prototype policy engine and the manageability adaptors enabling its interoperation with legacy resources were implemented as web services in order to leverage the platform independence, loose coupling and security features of this technology. The runtime reconfiguration of the policy engine necessitated the extensive use of techniques available only in an object-oriented environment:

- dynamic data type generation was required to support new types of resources when the policy engine was repurposed through configuration with a new system model;

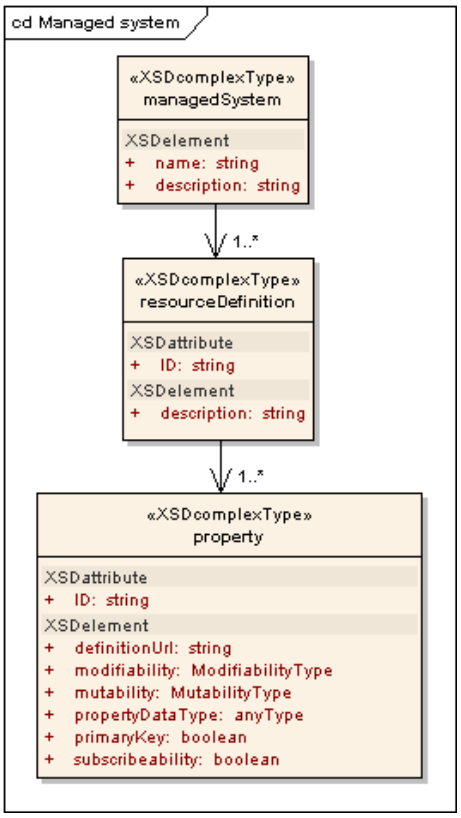

Fig. 2. UML meta-model of a managed system

- runtime generation of web service proxies was required to enable the policy engine to interoperate with new, resource-specific manageability adaptors;
- reflection was heavily used to access the values of the resource properties, both to read their values once the policy engine obtained them from the manageability adaptors and to set new values for the modifiable properties;
- generics were used to encode most of the functionality of manageability adaptors in a base abstract class, and to obtain resource-specific manageability adaptors by parameterising this abstract class with the resource data types.

Based on these requirements, J2EE and .NET were selected as candidate development environments for the prototype engine, with .NET being eventually preferred due to its better handling of dynamic proxy generation and slightly easier-to-use implementation of reflection.

A model was developed for the system model used to supply a specification of the ICT resources to be managed to the policy engine. As shown in Fig. 2, this *meta-model of a managed system* specifies a managed system as a named set of resource definitions. Each resource definition (i.e., *resourceDefinition* in the UML diagram) comprises a unique identifier *ID*, a description and a set of resource properties with their characteristics. A resource property has a data type (i.e., *propertyDataType*), and is associated a unique *ID* and the metadata repository URL where its definition is available. Several other property characteristics are exposed:

- *modifiability*—taken from the WS-ResourceMetadataDescriptor (WS-RMD) 1.0 specification [27], specifies if the property is "read-only" or "read-write";
- *mutability*—the WS-RMD MutabilityType [27] specifies if the property is "constant", "mutable" or "appendable";
- *subscribeability*—specifies whether a client such as the policy engine can subscribe to receive notifications when the value of this property changes;
- *primaryKey*—indicates whether the property is part of the property set used to identify a resource instance among all resource instances of the same type.

The prototype policy engine (class **policyEngine** in Fig. 3) was implemented itself as an instance of a *resourceDefinition* from the system meta-model in Fig. 2, such that an instance of the engine can be configured to manage other policy engine instances. The four properties of the policy engine are:

1) the policy evaluation period (i.e., 'period');
2) the set of policies to implement ('policySet');
3) the locations of the resources to manage ('resourceUrls'), which for the current version of the prototype are set explicitly (the use of a discovery technique [28] is intended for future versions);
4) the model of the managed system ('system'); note that the type of the 'system' property is precisely our managed-system meta-model (Fig. 2).

The prototype supports operators for the manipulation of primitive data types and a few of the more sophisticated operators recommended in [12] (e.g., set comprehension and scheduling). Support for additional operators is added on a regular basis as new case studies are being explored.

The policy engine reconfiguration is achieved through an instance of the **PolicyEngine** class in Fig. 3. **PolicyEngine** is a subclass of ***ManagedResource***<>, a generic abstract class that is the base class for the manageability adaptors in our architecture and which comprises three web methods:

- SupportedResource returns the ID of the supported resource type.
- GetResources returns the list of all available resource instances. The method takes as argument a list of resource property IDs, and only the values of these properties are assessed and returned to the caller, thus preventing unnecessary resource property evaluation.
- SetResources takes as argument a list of resources of the supported type, and assigns any new values specified by the caller for the resource properties declared modifiable in the system model. The resources whose properties need to be modified are uniquely identified by the value of the resource properties marked as "primary key" components in the system model.

These three web methods rely on resource-specific methods that are declared abstract in ***ManagedResource***<>, and which any of its subclasses (including **PolicyEngine**) implements:

- *GetRawResources* builds a list of all available resource instances. The values of the resource properties need not

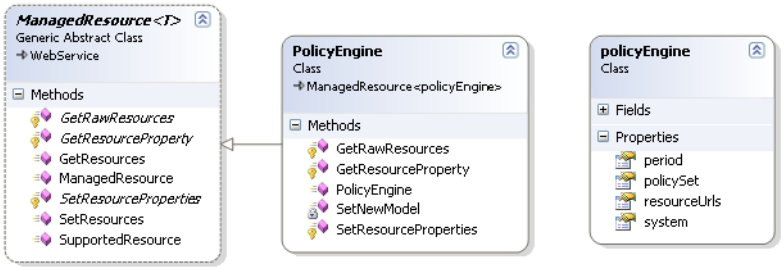

Fig. 3. The prototype policy engine—class diagram

be provided by this method.

- *GetResourceProperty* takes as arguments a resource instance and the ID of a resource property, and ensures that the property value is set in the resource object. The method is used by GetResources to fill in the required property values after obtaining a "raw" resource list from *GetRawResources*.
- *SetResourceProperties* takes a resource object and ensures that the modifiable properties of the corresponding real-world resource are assigned any new values specified in the resource object.

A web administration client was implemented to set the properties of the prototype policy engine (Fig. 4). This client uses the three web methods of **PolicyEngine** to read and to modify the policy engine parameters. In particular, the SetResources web method is used to (re)configure the engine:

- Changes to the system model result in a repurposing of the policy engine for the management of new types of ICT resources, which involves the dynamic generation of new data types, and of proxies for the manageability adaptors specific to the new resource types.
- Changes to the resource URLs trigger the engine to contact the manageability adaptors at the specified locations in order to establish the type of resources they expose.
- Policy changes lead to re-analysis of the new policies and to their parsing into an internal format that makes the periodical evaluation of policies computationally efficient.
- The web client can also be used to alter the policy evaluation period.

V. Case Study

In a realistic case study, we configured the policy engine to allocate the CPU capacity of a server to a set of services of different priority and subjected to variable workloads. The only resource defined in the server model was 'service' with the properties:

- *name*—a string used to distinguish among services;
- *priority*—an integer value;
- *cpuAllocation*—the percentage of the server CPU allocated to the service;
- *cpuUtilisation*—the amount of CPU utilised by the service, expressed as a percentage of its CPU allocation.

The policy depicted in Fig. 4 allocates a percentage of the CPU capacity of the server to each 'service' resource, as selected

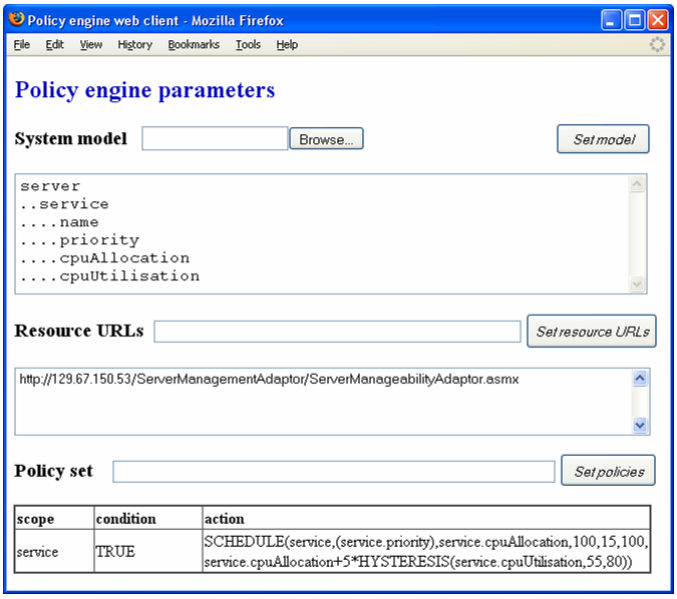

Fig. 4. Snapshot of the web client used to configure the policy engine

by the policy scope. The 'TRUE' policy condition requires that the policy action is applied at all times (i.e., in line with the policy evaluation period of the engine). The policy action is specified by means of an expression that uses the SCHEDULE(R, $ordering$, $property$, $capacity$, min, max, $optimal$) operator that

- sorts the resources in R in non-increasing order of the comparable expressions in $ordering$;
- in the sorted order, sets the specified resource $property$ to a value never smaller than min or larger than max, and as close to $optimal$ as possible;
- ensures that the overall sum of all $property$ values does not exceed the available $capacity$.

Accordingly, the policy action

$$\text{SCHEDULE}(service, \langle service.priority \rangle, service.cpuAllocation, 100, 15, 85, service.cpuAllocation + 5 * \text{HYSTERESIS}(service.cpuUtilisation, 55, 80))$$

in Fig. 4 will set the $cpuAllocation$ property of all services to a value between 15% and 100% subject to the overall CPU allocation staying within the 100% available capacity. Optimally, the $cpuAllocation$ should be left unchanged if the $55 \leq$

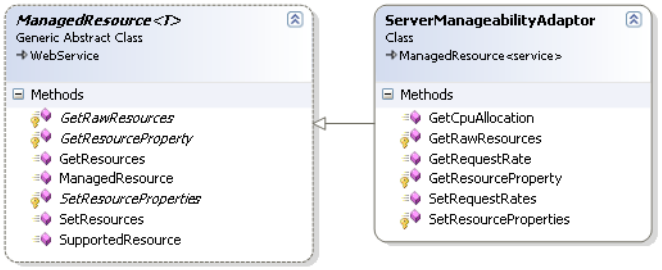

Fig. 5. The server manageability adaptor

$cpuUtilisation \leq 85$, decrease by 5(%) if $cpuUtilisation < 55$ and increase by 5(%) if $cpuUtilisation > 85$.[1]

Like the policy engine itself, the manageability adaptor used to interface the engine with the server simulator was implemented as a sub-class of ***ManagedResource***<>—Fig. 5.

The policy engine was configured to manage remotely a server simulator running a high-priority 'premier' service and a lower-priority 'standard' service. The two services handled simulated user requests with normally-distributed CPU utilisation and exponentially-distributed inter-arrival time. Fig. 6 shows the change in the system parameters when the request inter-arrival time of the two services was varied to simulate different workloads, and the policy engine was configured to implement the policy described earlier in this section:

a Both services are lightly loaded (5000μs request inter-arrival time) and have the minimum amount of CPU allocated (i.e., 15% each).
b The load increases for the standard service, and its allocated CPU is increased by the policy engine accordingly.
c For a brief period of time, the standard service uses its allocated CPU completely; no requests timeout though as its CPU allocation is increased swiftly.
d The premium service workload starts to increase, and the policy engine grows its CPU allocation. Accordingly, the standard service starts to get less CPU.
e As the workload for the premium service peaks and the policy engine schedules additional CPU capacity for this service, the standard service is allocated insufficient CPU and some of its client requests time out.[2]
f The inter-arrival time for the premium service increases, and some of the CPU capacity allocated to it during the previous time interval is re-deployed by the policy engine to the standard service. No more requests time out.
g Under constant workload, the CPU allocation is mostly stable.
h To explore the role of the hysteresis, we replaced the hysteresis term in the policy action with HYSTERESIS$(service.cpuUtilisation, 80, 80)$, basically eliminating the hysteresis. This led to significant oscillations in the CPU capacity allocated to the services. The reinstatement of the original policy after this time interval brings the system back into a stable state.

The policy evaluation period was set to 3 seconds for this experiment, so that the system could self-adapt to the rapid variation in the workload of the two services. This allowed us to measure the CPU overhead of the policy engine, which was under 1% with the engine service running on a 1.8 GHz Windows XP machine. In a real scenario, such variations in the request inter-arrival time are likely to happen over longer intervals of time, and the system would successfully self-configure with far less frequent policy evaluations. Note also that since the policy engine service is implemented as a managed resource, its policy evaluation period can be adjusted by another policy engine instance, so that it stays in step with the rate of change in the request inter-arrival time—a scenario that we are in the process of experimenting with.

VI. Conclusion

This paper described the SOA-based implementation of the general-purpose autonomic architecture originally introduced in [13]. Based on previous commercial work [11], [12] and on recent advances in autonomic computing [16], [23], [26], our implementation can be used to build policy-based autonomic systems out of non-autonomic ICT resources. Experimental work was carried out to validate the effectiveness of the implementation in an application to allocate server capacity to services of different priorities and varying workloads. The experimental results showed that our general-purpose framework could perform the planned management task successfully, and similarly to a dedicated commercial system for data-centre resource management [8], [11]. However, unlike the commercial resource-management system, our novel approach has the unique ability to handle resources whose types are unknown at implementation and deployment time, therefore enabling the cost-effective development of autonomic solutions across a broad variety of application domains.

Work is in progress to extend the prototype policy engine with operators supporting the implementation of "utility function" policies [29], and with functionality that takes advantage of the notification mechanism permitted by the system model used for its configuration. Additional case studies will be carried out to validate the applicability of the framework to a broader spectrum of use cases related to data-centre resource management in the first instance, and to other application domains in the longer term.

[1]The HYSTERESIS$(val, lower, upper)$ operator used to achieve this behaviour returns -1, 0 or 1 if $val < lower$, $lower \leq val \leq upper$ or $upper < val$, respectively.

[2]Requests time out after spending T=5s in a service request queue.

References

[1] T. Lenard and D. Britton, *The Digital Economy Factbook*. The Progress and Freedom Foundation, 2006.
[2] Y. Bar-Yam *et al.*, "The characteristics and emerging behaviors of system-of-systems," New England Complex Systems Institute, 2004.
[3] IBM Corporation, "Autonomic computing: IBM's perspective on the state of information technology," October 2001.
[4] J. Kephart and D. Chess, "The vision of autonomic computing," *IEEE Computer Journal*, vol. 36, no. 1, pp. 41–50, January 2003.
[5] R. Murch, *Autonomic Computing*. IBM Press, 2004.

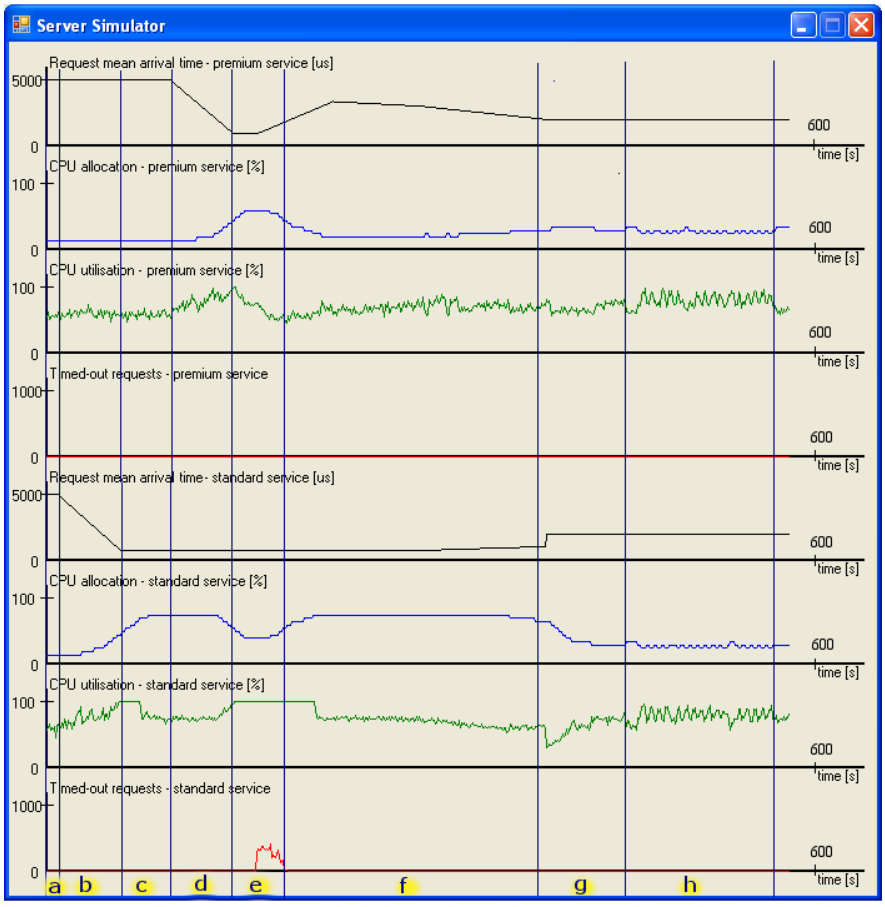

Fig. 6. Snapshot of a typical server simulation experiment

[6] M. Parashar and S. Hariri, "Autonomic computing: An overview," in *Unconventional Programming Paradigms*, ser. LNCS, vol. 3566, 2005, pp. 257–269.

[7] C. Lefurgy *et al.*, "Server-level power control," in *Proc. 4th IEEE Intl. Conf. Autonomic Computing*, Jacksonville, Florida, June 2007.

[8] B. McColl, "Intelligent, policy-driven orchestration of sensors and effectors across the data center in real-time," Sychron Inc," White paper, April 2004, http://hosteddocs.ittoolbox.com/BM042304.pdf.

[9] W.-S. Li *et al.*, "Load balancing for multi-tiered database systems through autonomic placement of materialized views," in *Proc. 22nd IEEE Intl. Conf. Data Engineering*, Atlanta, Georgia, April 2006.

[10] C. Ghanbari *et al.*, "Adaptive learning of metric correlations for temperature-aware database provisioning," in *Proc. 4th IEEE Intl. Conf. Autonomic Computing*, Jacksonville, Florida, June 2007.

[11] R. Calinescu and J. M. D. Hill, "System providing methodology for policy-based resource allocation," July 2004, united States Patent Application no. 10/710322.

[12] R. Calinescu, "Challenges and best practices in policy-based autonomic architectures," in *Proc. 3rd IEEE Intl. Symp. Dependable, Autonomic and Secure Computing*, Columbia, Maryland, USA, 2007, pp. 65–74.

[13] ——, "Model-driven autonomic architecture," in *Proc. 4th IEEE Intl. Conf. Autonomic Computing*, Jacksonville, Florida, June 2007.

[14] Microsoft Corporation, "Windows System Resource Manager (WSRM) White Paper," August 2003.

[15] Sun Microsystems, Inc, "Sun^{TM} Grid Compute Utility—Reference guide," June 2006, http://www.sun.com/service/sungrid/SunGridUG.pdf.

[16] J. Parekh *et al.*, "Retrofitting autonomic capabilities onto legacy systems," *Cluster Computing*, vol. 9, no. 2, pp. 141–159, April 2006.

[17] G. Kaiser *et al.*, "Kinesthetics extreme: An external infrastructure for monitoring distributed legacy systems," in *Proc. 5th Intl. Active Middleware Workshop*, June 2003, http://www.psl.cs.columbia.edu/ftp/psl/CUCS-019-03.pdf.

[18] H. Kasinger and B. Bauer, "Towards a model-driven software engineering methodology for organic computing systems," in *Proc. 4th IASTED Intl. Conf. Computational Intelligence*, Calgary, Alberta, Canada, July 2005, pp. 141–146.

[19] IBM Corporation, "An architectural blueprint for autonomic computing," 2004, http://www-03.ibm.com/autonomic/pdfs/ACBP2_2004-10-04.pdf.

[20] B. Moore, "Policy Core Information Model (pcim) extensions," January 2003, iETF RFC 3460, http://www.ietf.org/rfc/rfc3460.txt.

[21] B. Murray *et al.*, "Web Services Distributed Management: MUWS primer," February 2006, OASIS WSDM Committee Draft.

[22] IBM Corporation, "Autonomic integrated development environment," April 2006, http://www.alphaworks.ibm.com/ tech/aide.

[23] D. Agrawal *et al.*, "Autonomic Computing Expression Language (ACEL) 1.2: User's Guide," 2005, http://www-128.ibm.com/developerworks/edu/ac-dw-ac-acel-i.html.

[24] R. Anthony, "A policy-definition language and prototype implementation library for policy-based autonomic systems," in *Proc. 3rd IEEE Intl. Conf. Autonomic Computing*, Dublin, Ireland, June 2006, pp. 265–276.

[25] N. Damianou *et al.*, "The Ponder policy specification language," in *Policies for Distributed Systems and Networks*, ser. LNCS, vol. 1995, Bristol, UK, 2001, pp. 18–38.

[26] IBM Corporation, "Policy Management for Autonomic Computing," 2005, http://dl.alphaworks.ibm.com/technologies/pmac/PMAC12_sdd.pdf.

[27] OASIS, "Web Services Resource Metadata 1.0," November 2006.

[28] R. Harbird *et al.*, "Adaptive resource discovery for ubiquitous computing," in *Proc. 2nd Workshop Middleware for Pervasive and Ad-hoc Computing*, Toronto, Canada, October 2004, pp. 155–160.

[29] S. R. White *et al.*, "An architectural approach to autonomic computing," in *Proc. 1st IEEE Intl. Conf. Autonomic Computing*. 2004, pp. 2–9.