Supervisory Control As Pattern for Integrated IT Management

Sven Graupner, Daniel Gmach

HP Laboratories
HPL-2011-244R1

Keyword(s):

integrated management; management automation; control system; supervisory control; data center energy efficiency

Abstract:

Saving energy in large data centers has become a focus of research and development. For its realization, IT and facility control systems must be connected and coordinate their activities. Building such an integrated management system that operates automatically and reliably is a challenging task. The paper describes such a system that has been built in the HP Labs Palo Alto research data center. We discuss a particular aspect of such a system: how the integration logic of the various management and control systems can be designed more systematically, referring to a theoretic framework of supervisory control that had been developed for industrial automation in the late 1990's. We adopt the framework for constructing a set of coordinated controllers that are performing a set of desired management tasks.

External Posting Date: August 21, 2012 [Fulltext] Approved for External Publication
Internal Posting Date: August 21, 2012 [Fulltext]

Supervisory Control As Pattern for Integrated IT Management

Sven Graupner and Daniel Gmach
Hewlett-Packard Laboratories
1501 Page Mill Rd, Palo Alto, CA 94304, USA
{sven.graupner, daniel.gmach}@hp.com

***Abstract*—Saving energy in large data centers has become a focus of research and development. For its realization, IT and facility control systems must be connected and coordinate their activities. Building such an integrated management system that operates automatically and reliably is a challenging task. The paper describes such a system that has been built in the HP Labs Palo Alto research data center.**

We discuss a particular aspect of such a system: how the integration logic of the various management and control systems can be designed more systematically, referring to a theoretic framework of supervisory control that had been developed for industrial automation in the late 1990's. We adopt the framework for constructing a set of coordinated controllers that are performing a set of desired management tasks.

Keywords: integrated management, management automation, control system, supervisory control, data center energy efficiency.

I. INTRODUCTION

Several factors continue to contribute to the increasing complexity of IT management in data centers. First, there is the need to connect management systems that have not been connected in the past and that have not been designed for such a connection, e.g. for coordinated power and workload management. Second, virtualization of compute, storage and networking resources offers unprecedented flexibility, but at a price of increased management complexity. And third, people are still the limiting resource to actively observe, decide and operate managed environments leading to the need for more automation.

We consider a case of integrated management [1] by an automated management system that has been built in HP Labs over the past three years as an experimental system. It manages (shifts) workloads between virtualized server farms. When overall workloads decline, such as during night, workloads are concentrated on fewer servers in certain regions in a data center. Cooling in other regions thus can be reduced saving not only power on vacated servers, but also on facilities equipment by lowering air flows and raising temperature levels in the underutilized regions in a data center [2].

In such a case, coordinated control must occur between workload and facilities management systems. While the scenario appears simple, constructing such an integrated function is difficult, even more so when the function should be performed automatically, robustly and continuously with as little human intervention as possible. In [3], we discussed interoperability and interface design for this integrated IT management system based on the WBEM architecture [4].

In this paper, we address design and operational issues for reliably operating such an integrated management system focusing on the integration logic that ties the various sub-systems together. This logic realizes coordinated data transfer and the coordinated execution of operations in sub-systems that are performing tasks such as the migration of virtual machines (VM), power cycling servers or controlling air vents.

The reason why constructing such integration and automation logic is difficult is caused by several factors:

- coordination of data flows between systems (with different data formats and delivery mechanisms),
- evaluation and decision making based on that data (goals, policies),
- coordinated control flows (actions) triggered between sub-systems,
- observation of execution of control flows (verification of achieving expected system states),
- detection of unexpected system states, and the
- handling of such conditions (correction or elevation).

Implementing these requirements can make integration logic complex with software that is hard to understand and hard to maintain. Observations from prior management system integrations [5] have shown that integrated IT management standards mainly address issues of data interoperability (data formats, protocols), such as specifications ASN.1, MIB, CIM, MOF, or WBEM, or the numerous web services management standards [6, 7]. Coordination and control, however, have not been considered accordingly for creating the operational logic that ties the different parts together operationally. This logic today is mainly implemented in programming or scripting languages, sometimes in process execution languages, which mostly only support basic control flows. Higher-ordered control such as coordination, decision making, observation and evaluation of conditions often turns into nested 'if-then-else' constructions. Often, they only consider the 'happy-paths' of execution, which model the expected behavior of a system. In case of error or unexpected conditions, integration logic tends to throw exceptions or abruptly aborts leaving the underlying managed environment in an undefined state. As a result, management system integrations are brittle, unreliable and leaving chances for undefined states in the managed systems.

The question should be asked why automation and integration of systems has reached a much more advanced state in other domains, such as in industrial automation, in automated machinery or assembly lines that are common in modern manufacturing.

We have been investigating controller-based approaches for the design of integrated IT management systems for some time. Control theory and controller design have widely been researched in mechanical and electrical engineering from the 1950s through the 1970s, paving the road for the advances in industrial automation. The controller pattern, however, has been applied in software engineering to a much lesser extend, and only recently in the IT management domain, mainly only considering continuous control variables such as for managing resource utilization, admission control and workloads [8–11].

Integration logic, in contrast, is of a discrete nature. Other principles apply. For our energy/workload management system, we apply a particular area of controller research from the 1990's: *Supervisory Control* [12]. The reason for choosing supervisory control is because it uses discrete state sets. The approach is of a general nature and can be applied to a range of IT management automation and integration scenarios.

The paper is structured as follows: after a brief description of the state of integration and automation in IT management in Section II, we introduce supervisory control in Section III. Section IV then introduces our case of an integrated and automated energy/workload management system. We present a model-driven approach to interfaces and designs of the various controllers for our system. Section V concludes the paper.

II. Related Approaches for Integration and Automation in IT Management

A number of approaches have been developed for IT management integration and automation over the years. Most managed hardware systems today offer standardized management interfaces from which data can be obtained and that can be programmed. While standardization has penetrated managed hardware, software systems (managed and management) largely use proprietary interfaces, which is part of the data integration problem that was addressed in [3].

Integration of control flows between management systems was mainly mapped into management protocols, for which also a number of standards have emerged. However, while these technologies allowed interoperability with managed and among management systems, they did not address automation. Automation involves a cycle of information gathering from the managed environment, evaluation and decision making and execution of decisions performed by logic rather than a human. A number of approaches exist for IT automation:

Scripting and programming merge logic for information gathering, evaluation, decision making and execution into the framework of a scripting or a programming language (e.g., Shell scripts, Perl or Java). Logic quickly becomes complex.

Process languages and workflows are similar to scripting, but are often facilitated through visual design tools suggesting simplicity and maintainability. However, practical cases show a lack of scale due to the visual designs, the lack of test and debugging support as well as a lack of control in unexpected conditions, e.g., when commands do not fully execute or components do not respond.

Policies aim to separate decision making from processes of data gathering and execution. Extensive research has been conducted on IT management policies [13–17]. Policy frameworks allow the separation of concern and the isolation of decision making removing some complexity.

Controllers also separate decision-making from data gathering and execution. Inputs and outputs are clearly separated from the control function. Controller operate on input *e* to a control function *cf()*, which translates *e* into control signals as output *r* to the controlled environment (which is often called 'plant' in control literature).

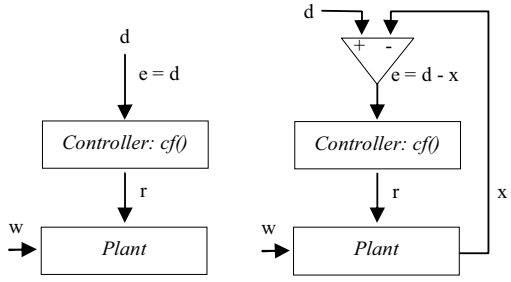

Figure 1: **Simple Feed Forward and Feedback Control.**

Simple feed forward control, as shown on the left in Figure 1, uses only a reference *d* as input. As changes are made to *d*, *cf()* generates new values *r* to actuate these changes in the plant. Feed forward control cannot react to disturbances *w* that may occur in the plant. To stabilize the controller against those disturbances, a feedback loop is usually considered creating a channel *x* back to the reference *d*, which both combined become the input *e* for the controller. A feedback controller is shown on the right hand side in Figure 1.

Feedback controllers have been explored for IT management since the early 2000's (also as part of autonomic computing [18, 19]) with the focus on continuous control variables. Controllers were demonstrated for managing workloads, resource utilization and admission control [8–11]. Fewer publications exist on discrete state control for automating IT management tasks. One example is the management controller using executable Petri Nets [20], another is [21].

III. Supervisory Control

Supervisory control [12] was introduced as a technique to systematically aggregate controller hierarchies. Supervisory controllers observe a control system and intervene when the situation demands. An interface layer is introduced that connects the supervisor with the underlying supervised system. [12] describes an underlying controllers to be of any nature (continuous, discrete, linear or differential). The supervisory controller is then defined as an event-driven, asynchronous discrete event system (DES) [22] with a discrete event model (DEM) [23].

A DES is a dynamic system with a discrete state space model and with piecewise constant state trajectories; the time instant at which state transitions occur, and with transitions. The state transitions of a DES are initiated by events and can be labeled with elements from an alphabet. Finite automata [24] or Petri Nets [25] are suitable for modeling DES.

The supervisory controller in [12] is a DES that is modeled as a deterministic finite automaton [24]. This automaton is specified by $S = (\tilde{S}, \tilde{X}, \tilde{R}, \delta, \phi)$, where $\tilde{S}$ is the set of states, $\tilde{X}$ is the set of plant symbols (controller input), $\tilde{R}$ is the set of controller symbols (controller output), $\delta : \tilde{S} \times \tilde{X} \to \tilde{S}$ is the state transition function, and $\phi : \tilde{S} \to \tilde{R}$ is the output transition function. The symbols in set $\tilde{R}$ are called controller symbols. They are generated by the controller. Likewise, the symbols in set $\tilde{X}$ are called plant symbols and are generated based on events occurring in the plant. The action of the controller can be described by the following equations:

(1) $\tilde{s}[n] = \delta(\tilde{s}[n-1], \tilde{x}[n])$ – state at time *n* is a function of the previous state at time *n-1* combined with an event *x*, and

(2) $\tilde{r}[n] = \phi(\tilde{s}[n])$ – controller output *r* is a function of the state at time *n*, where $\tilde{s}[n] \in \tilde{S}, \tilde{x}[n] \in \tilde{X}$, and $\tilde{r}[n] \in \tilde{R}$. The index *n* is a time index that specifies the order of the symbols in their temporary sequence. A tilde sign indicates a set or a sequence. For example, $\tilde{X}$ is the set of plant symbols, also called the alphabet, and $\tilde{x}[n]$ is the *n*-th symbol of a sequence of plant symbols.

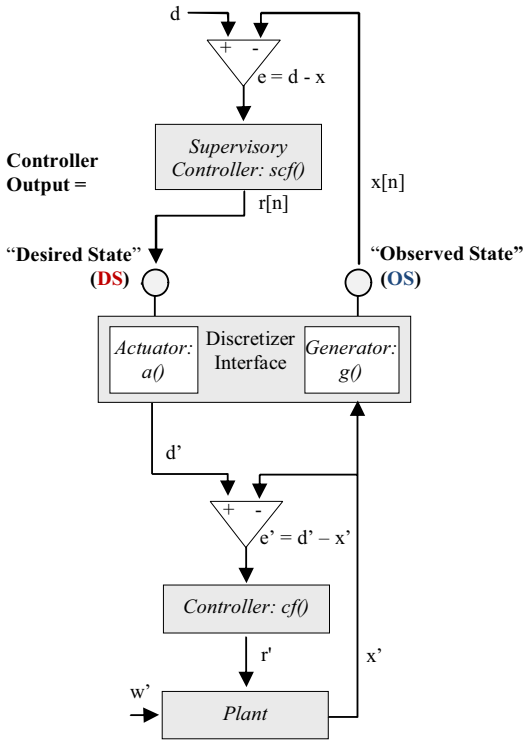

Figure 2: **Supervisory controller connected to supervised control system through discrete state sets.**

Figure 2 shows what [12] refers to as a hybrid control system. What is particular is the interface between the controller and the underlying supervised control system. It discretizes the events *x'* into sequences of discrete states $\tilde{x}[n]$ by a generator function *g()* producing input for the supervisory controller (quoted symbols such as *x'* refer to the supervised controller). Reversely, sequences of output states $\tilde{r}[n]$ generated by the controller are translated into control actions *r* for the supervised control system by the actuator function *a()*. Its output then becomes input for the controlled system, e.g., as new reference value *d'* as shown in Figure 2.

The discretizer interface allows the supervisory controller to normalize and filter events and control actions—shielding both from underlying control (or management) systems. Similarly, inputs can be collected from multiple underlying management systems and can be combined into *x*[*n*]. Outputs can be distributed to more than one supervised system.

The filter and translation function of the Discretizer Interface acts like an *abstraction layer* that is used, for instance, in operating systems to shield the operating system logic (e.g., scheduling) from the variety of underlying drivers, interfaces and hardware components. [12] provides a mathematical formulation for the filtering and transformation functions *g()* and *a()* based on hypersurfaces that are defined in overlapping state spaces. Events *x'* in the underlying system trigger an event *x*[*n*] when they cut through the hypersurface defined in $\tilde{X}$.

In the following section we show how these hypersurfaces can be interpreted as model-based interfaces for controllers. These models contain state variables that do not only provide views on the control domain; they also play an active role in a sense that changes to state variables (by a *set()* operation in $\tilde{r}[n]$) can trigger corrective action in underlying controllers.

For our convenience, we refer to the discrete state sets $\tilde{x}[n]$ and $\tilde{r}[n]$ as *Observed State:* $OS[n] = \tilde{x}[n]$ and *Desired State:* $DS[n] = \tilde{r}[n]$, respectively [20]. Both, DS and OS are defined as sets of discrete state variables that are modeled and maintained about a managed component, a subsystem or an entire management environment.

IV. Design Case: Integrated Energy/Workload Management for a Data Center

This section presents the energy/workload management system we refer to in this paper. More details about this system are published in [2] and [3]. We select a control function that spans across a number of systems: the dynamic distribution and re-distribution of workloads among servers in a data center using VM migration. Under normal operation, the system follows the goal to concentrate workloads (in VM's) on as few server nodes as possible, while at the same time maintaining all application-level SLA. It prefers server nodes that are physically concentrated in regions in a data center that can be cooled more efficiently. As result of this concentration, server and cooling power can be reduced in the vacated regions.

A. Design Principles

The overall system consists of a number of components, which we design and implement as interacting, autonomous controllers using the pattern of supervisory control. Our aim is that controllers can operate (and fail) independently without causing global effect on the overall system. Containing failures and avoiding undesired chain reactions is hard to achieve in centralized integration logic. To prevent it, we establish the following design principles:

- Autonomy of components (remain operational when other parts fail) by decoupling component execution.
- Distributed system design, avoiding central points of failure, e.g., by using the central CMDB.
- Data-driven approach as opposed to control driven approach. Operations are triggered by changing state variables, which are remembered.
- Consider not just information of the current state, but also of the expected (desired) state of the managed domain.

B. *Controller Surfaces*

We consider DS and OS as the two input interfaces for a controller. These interfaces contain sets of state variables that are necessary for the controller to operate. Four operations are defined for DS and OS: *get(s)* $\rightarrow v$; *set(s,v)*; *subscribe(s,cb)*; *unsubscribe(s,cb)*; with *s* referring to a state variable in DS or OS, *v* referring to its value and *cb* to a callback state variable in an endpoint that wishes to be notified when the value of *s* changes. We refer to this interface definition as model-based, and, since actions are facilitated by *set()* operations in DS, model-driven. It is an alternative interface design to rich functional management API.

Web services technologies use hierarchical documents as containers for data. The following fragment illustrates a document for DS for a simple CRM application:

```
<app:crm43 name="customer relationship management system">
  <sla:crm43>
    <sla:EURespTime op="le" value="1000" unit="msec" />
  </sla:crm43>
  <comp:ws16 name="web server">
    <ut:ws16 cpu="0.6" mem="0.8" io="0.6" />
    <status:ws16>up</status:ws16>
  </comp:ws16>
  <comp:crmlogic64 name="crm application server"> ... </comp:crmlogic64>
  <comp:db88 name="crm DB"> ... </comp:db88>
</app:crm43>
```

The fragment first shows a simple SLA defined as end user response time for the CRM system of less or equal 1000msec. Then the DS model shows three application components: a CRM web, application and backend database server. For the web server, utilization targets (UT) are shown with values for CPU, memory and IO. Furthermore, it shows that the desired state of the web server is up. These state variables describe the desired operating conditions for the VM in which application components are deployed (single VM deployment assumed).

Each DS model is complemented by an OS model that contains state values that are actually observed (discovered, detected or monitored) in the managed environment. The document schema typically is the same, but now containing values reported from the managed environment, for example:

```
<app:crm43 name="customer relationship management system">
  <sla:crm43>
    <sla:EURespTime value="386" unit="msec" />
  </sla:crm43>
  <comp:ws16 name="web server">
    <ut:ws16 cpu="0.23" mem="0.21" io="0.12" />
    <status:ws16>up</status:ws16>
  </comp:ws16>
  <comp:crmlogic64 name="crm application server"> ... </comp:crmlogic64>
  <comp:db88 name="crm DB"> ... </comp:db88>
</app:crm43>
```

The OS document shows a monitored application response time of 386ms, the up-status of the web server and measured resource utilizations reported from the VM of 0.23cpu, 0.21mem and 0.12 for IO. The controller now can compare the observed values against the desired values in the DS and determine corrective actions, if needed.

C. *Controller Designs*

The overall system consists of thee types of controllers, each controlling a domain: an application, a server node, and a server pool. The overall system is connected to the Palo Alto Data Center Smart Cooling (DSC) infrastructure [26], which allows programmatic control over cooling and air flows.

Application Controller. Each application is composed of one or more application components, with each component hosted in a VM. An application controller exists per application and monitors the application performance, which is reported into its OS. The expected performance of an application is expressed in its SLA, typically in terms of end-to-end response time or throughput targets. The SLA serves as the DS reference point for the application controller, against which the monitored OS can be compared. Examples of such DS and OS models were shown before as XML documents.

Figure 3 shows the application controller. It's control function *ACf()* compares the OS against the DS and computes as output so-called resource utilization targets (UT[]) for all VM's in which application components reside.

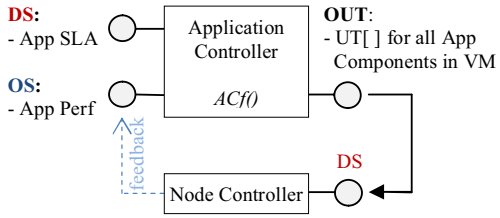

Figure 3: **Application Controller.**

Utilization targets are derived from performance models, which must be known about an application. A number of techniques exist to derive such performance models [27]. In our case, studies have been conducted using layered queuing models to correlate application performance to resource use on individual application components. Results about the accuracy of these models are reported in [28].

Utilization targets are passed to DS of underlying node controllers hosting VM's with application components. New UT values in their DS lead to a difference to the current values and can trigger adjustment actions in the node controllers. No messages are sent back to the application controller to confirm or report the completion of the UT adjustment. The application controller just continues monitoring application performance and should observe the effects of the new UT values. This indirect link is shown as dashed feedback arrow in Figure 3.

If, after some time, no such effect can be observed, e.g., because the UT adjustment for the VM in the underlying node controller was not possible at the time, the application controller may simply reissue the UT (in case the previous *set()* operation might have gotten lost). Since *set()*-operations in DS

are idempotent, repetition has no harmful effect. If still no reaction can be observed, the application controller may follow other options, and ultimately raise an alert, if no option is left.

In Figure 3, the application controller acts as a supervisor for the underlying node controller. The node controller receives control instructions from the application controller in its DS by setting new UT for VM. The application controller monitors the effect of these changes locally in its OS.

Node Controller. Figure 4 shows the node controller, which exists for each server node managing all VM's on this node. The node controller supports operations of: creation, launch, adjustment of UT, migration, shutdown and destruction of VM. Following our design principles, these operations are not implemented as methods. They are modeled as state variables in the DS in form of a VM lifecycle model and are "invoked" by *set()* operations on lifecycle states in the DS.

One of the tasks of the node controller is to maintain the UT for VM and enact those targets by dynamically adjusting resource allocations of the VM. Using utilization targets in the DS instead of allocation targets decouples the controller from underlying physical resource allocation details. For its OS, the node controller collects the average resource consumption of each VM from monitors and determines the required resource allocation to achieve the specified UT. For a given consumption, a change in allocation leads to the change in utilization. If the sum of resource allocation requests for all VM on a node is less than the node's capacity, then all requests are granted, which is path a) in Figure 4. It uses the locally installed VM management system (VM Controller in the figure) adjusting the VM's resource allocation settings. This control cycle is in the range of seconds.

If the sum exceeds the node's capacity, e.g., by utilization value >0.8, the supervising pool controller is notified by setting the node's utilization value in the pool controller's OS, which is shown as path b) in the figure. The node controller notifies over- and underutilization to the pool controller's OS model.

Feedback for the node controller is achieved for path a) by adjusting VM resource allocation settings in the underlying VM controller, and for path b) by subsequent VM migration operations issued by the pool controller adding or removing VM from the node such that UT for all VM on a node are met.

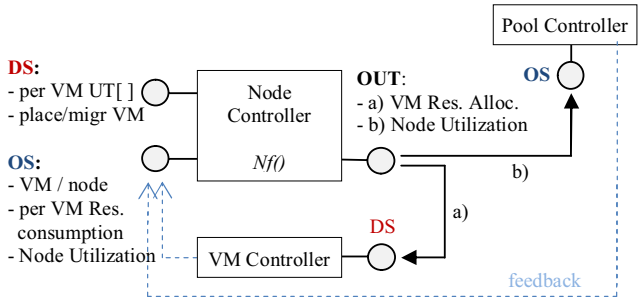

Figure 4: **Node Controller.**

Two independent update cycles support resilience and responsiveness. One is periodically initiated by the pool controller (in range of minutes) reading the utilizations from each node's OS and collecting them in its own OS. The other is initiated by node controllers, which may observe their over- or underutilization and reporting this condition by setting utilization value in the pool controller's OS.

Pool Controller. The pool controller supervises underlying node controllers and also has control over the power status of the server nodes in a pool. A pool is a group of physical server nodes that are located in close physical proximity. There can be multiple physical server pools in a data center in different regions. Figure 5 shows the pool controller. The Node PWR Controller allows a node to be turned on or off.

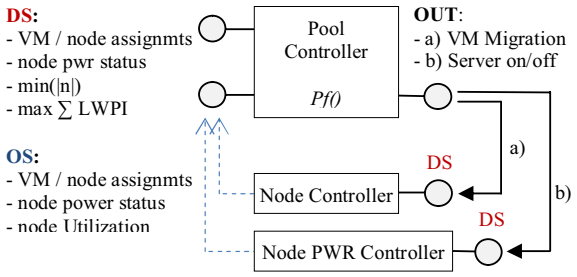

Figure 5: **Pool Controller.**

The Pool Controller's OS contains the current node utilization values of all nodes in the pool and the power status of server nodes. In addition, the pool controller maintains information about thermal conditions in the region of the server pool, expressed in a dynamic factor called the *Local Workload Placement Index* (LWPI), which states how efficiently a location in the data center can be cooled [29]. Thermal conditions and LWPI are values set by the Data Center Smart Cooling (DSC) [26] infrastructure. The pool controller function *Pf()* is a complex optimization. It enables the pool controller to make decisions about VM placements and migrations following its overall goal of minimizing server nodes that are necessary to support the current aggregate workload in the pool and concentrate workloads on servers that allow additional energy savings on facility savings by maximizing the LWPI (higher LWPI values indicate higher thermal efficiency). Output of *Pf()* are decisions about VM migrations among server nodes. Their effects are reported into its OS section describing the current locations of VMs on server nodes.

D. Realization and Results

A controller toolkit [30] was developed for implementing controllers using the Globus Toolkit [31] with extensions to support the WS-Management standard [7]. It allowed access to DS and OS models that were represented as XML documents behind WS-endpoints. Access to state variables in documents was facilitated through XPath queries.

Table 1 summarizes results of four RUBiS application scenarios tested over a 10h experiment in our data center [2]. The column "App. Perf." shows the percentage of requests below the SLA response time target. "Server power" shows the power consumption of the 20 servers in the testbed and the column "Blower power" shows the total power consumption of six Computer Room Air Conditioner (CRAC) units in the DC.

Four scenarios were evaluated. A fixed approach (A) used fixed utilization targets for workloads (as baseline). The integrated performance management approach (B) dynamically adjusted utilization targets such that response time targets could

be met. (C) adjusted server power based on utilization accepting some performance loss, and (D) integrated with the CRAC. The average power consumption from (C) and (D) was reduced by 35% compared to (A) and (B). (D) furthermore reduced the CRACs blower power by additional 15%.

Approach	App. Perf.	Server power	Blower power
A: Fixed Approach	61 %	3.81 KW	24.7 KW
B: Integrated Performance mgmt.	79 %	3.87 KW	24.7 KW
C: Integrated IT mgmt.	70.1%	2.61 KW	24.7 KW
D: Integrated data center mgmt.	70.1%	2.47 KW	20.9 KW

Table 1: Experimental Results.

Blower power, however, represents a fraction of the power consumption of the cooling infrastructure. The larger part is thermodynamic work performed by the central chiller plant providing chilled water to the CRACs. From earlier experience, we estimated savings of ~38 KW over the 10 hours for the chiller plant representing another 16% of savings for cooling.

V. Summary and Conclusions

The paper presented a controller-based design for the integration logic for management and control systems that need to cooperate in a data center. The paper referred to a research system that has been built in the HP Labs Palo Alto data center.

Challenges continue to exist for controller-based management automation, such as:

- Atomicity and integrity of complex, multi-model state updates in DS or OS across multiple controllers are a problem.
- Inconsistency of the OS model with the managed domain is another problem, e.g., caused by the delay between a failure and its detection and update in OS models.
- Indirect coupling can cause oscillation and undesired controller behavior, especially when those are cascading across controllers.

Future work includes further study of behavior in failure conditions. Another direction of research is to exploit the body of theoretic work for validating properties in supervisory control, e.g. state reachability. And third, simulation systems can help reduce the effort for designing, implementing, and testing controllers, individually and in integration scenarios.

Research in the field of IT management integration and automation is a rich area of research that continues to be relevant for the IT industry.

References

[1] Hegering, H. G., Abeck, S., Neumair, B.: Integrated Network of Networked Systems, Morgan Kaufmann Series in Networking, 1999.

[2] Chen, Y., Gmach, D., Hyser, C., Wang, Z., Bash, C., Hoover, C., Singhal, S.: Integrated Management of Application Performance, Power and Cooling in Data Centers, NOMS 2010, Osaka, Japan, 2010.

[3] Graupner, S., Gmach, D.: Integration of a Power/Workload Control System into IT Management Architecture. Proceedings of CNSM 2010, pp. 174-181, Niagara Falls, Canada, Oct 25-29, 2010.

[4] DMTF, Web-Based Enterprise Management (WBEM), http://www.dmtf.org/standards/wbem.

[5] Graupner, S., Nitzsche, T.: Model-driven Software Configuration With the Radia SDC, Proc. of the 12th HP OpenView University Association Conference (HP-OVUA), pp.397-400, Porto, Portugal, July 10-13, 2005.

[6] OASIS: Web Services Distributed Management (WSDM), http://www.oasis-open.org/specs, Working Draft, September 29 2004.

[7] Web Services for Management (WS-Management), v1.0, April 2006, http://www.dmtf.org/standards/wsman.

[8] Liu, X., Zhu, X., Singhal, S., Arlitt, M.: Adaptive Entitlement Control of Resource Containers on Shared Servers, IM'2005, Nice, May 2005.

[9] Xu, W., Zhu, X., Singhal, S., Wang, Z.: Predictive Control for Dynamic Resource Allocation in Enterprise Data Centers, NOMS'2006, Vancouver, Canada, April 3-7, 2006.

[10] Hellerstein, J. L., Diao, Y., Parkh, S., Tilbury, D.: Feedback Control of Computing Systems, John Wiley & Sons, New York, 2004.

[11] Gmach, D., Rolia, J., Cherkasova, L.: Satisfying Service Level Objectices in a Self-Managing Resource Pool. IEEE SASO 2009, pages 243-253, San Francisco, Sep 14-18, 2009.

[12] Koutsoukos, X.D., Antsaklis, P.J., Lemmon, M.D.: Supervisory Control of Hybrid Systems, Proceedings of the IEEE, Vol. 88, No. 7, July 2000.

[13] Sloman, M., Damianou, N., Dulay, N., Lupu, E.: The Ponder Policy Specification Language, pages 18─38, POLICY 2001.

[14] Web Services Policy 1.5 Framework, http://www.w3.org/ TR/ws-policy.

[15] Burgess, M.: An Approach to Understanding Policy Based on Autonomy and Voluntary Cooperation, DSOM'05, Barcelona, Oct 24-26, 2005.

[16] Kagal, L.: Rei: A Policy Language for the Me-Centric Project, HP Labs Technical Report http://www.hpl.hp.com/techreports/2002/HPL-2002-270.pdf , Sep 26, 2002.

[17] Pell, A.R., Eshghi, K., Moreau, J.J., Towers, S.J.: Managing in a Distributed World, Proceedings of the 4th IFIP/IEEE International Symposium on Integrated Network Management, May 1995.

[18] Kephart, J., Chess, D.M.: The Vision of Autonomic Computing, IEEE Computer 36(1), 41-50, http://researchweb.watson.ibm.com/ autonomic.

[19] Lupu, E., Dulay, N., Sloman, M., Sventek, J., Heeps, S., Strowes, S., Twidle, K., Keoh, S.L., Filho, A.: AMUSE: Autonomic Management of Ubiquitous e-Health Systems. Concurrency and Computation: Practice and Experience 20(3): 277-295, 2008.

[20] Graupner, S., Cook, N., Coleman, D.: Automation Controller for Operational IT Management, the 10th IFIP/IEEE Symposium on Integrated Management (IM), Munich, Germany, May, 2007.

[21] Bandara, A.K., Lupu, E.C., Russo, A., Dulay, N., Sloman, M., Flegkas, N., Charalambides, M., Pavlou, G.: Policy Refinement for IP Differentiated Services Quality of Service Management. IEEE Transactions on Network and Service Management 3(2): 2-13, 2006.

[22] Ramadge, P.J., Wonham, W.M.: The Control of Discrete Event Systems, Proceedings IEEE, Vol. PROC-77, No. 1, pp. 81–98, January, 1989.

[23] Inan, K., Varaiya, P.: Finitely Recursive Process Models for Discrete Event Systems, IEEE Trans. on Automatic Control, Vol. AC-33, No. 7, pp. 626–639, July, 1988.

[24] Hopcroft, J.E. and Ullman, J.: Introduction to Automata Theory, Languages and Computation. Reading, MA: Addison-Wesley, 1979.

[25] Giua, A.: Petri Nets as Discrete Event Models for Supervisory Control, PhD Dissertation, Rensselaer Polytechnic Institute, Troy, NY, 1992.

[26] Bash, C., Patel, C., Sharma, R.: Dynamic Thermal Management of Air-cooled Data Centers, in Proc. of the Intersociety Conf. on Thermal and Thermomechanical Phenomena in Electronic Systems (ITHERM), 2006.

[27] Jain, R.K.: The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, 685 pages, Wiley, 1991.

[28] Rolia, J.A., Krishnamurthy, D., Min, X., Graupner, S.: APE: An Automated Performance Engineering Process for Software as a Service, HP Labs Technical Report, HPL-2008-65, 2008.

[29] Bash, C., Forman, G.: Data Center Workload Placement for Energy Efficiency, in Proceedings of the ASME InterPACK Conference, Vancouver, Canada, 2007.

[30] Graupner, S., Cook, N., Coleman, D., Nitzsche, T.: Management Middleware for Enterprise Grids, 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006), Singapore, May, 2006.

[31] Globus Toolkit GT4, http://www.globus.org.