Policy-Based Management of Networked Computing Systems

Dakshi Agrawal, Kang-Won Lee, and Jorge Lobo, IBM T. J. Watson Research Center

ABSTRACT

This article provides an overview of the Policy Management for Autonomic Computing (PMAC) platform, and shows how it can be used for the management of networked systems. We present the policy information model adopted by PMAC and the system model for interaction between the policy manager and the managed resource. We also present the main components of PMAC for policy creation, storage, evaluation, and enforcement, and present practical applications of PMAC in networks management.

INTRODUCTION

The need for a more autonomous management of networks and distributed systems has driven research and industry to look for management frameworks that go beyond the direct manipulation of network devices and systems. One approach toward this aim is to build policy-based management systems [1]. In general, policies represent externalized logic that can determine the behavior of managed systems. The promise of policy-based management is that the operation of computing resources can be guided to follow certain rules, and dynamically configured so that they can achieve certain goals and react more nimbly to their environment. Over the years, multiple approaches have evolved to support policy-based management of complex information technology (IT) systems. Some of these approaches are tightly coupled to a specific application domain and operation environment, while others are designed to be generic and more broadly applicable.

Policy Management for Autonomic Computing (PMAC) is a generic policy middleware platform that can be used to manage multiple aspects of a large-scale distributed system such as quality of service (QoS), configuration, and auditing. Another equally important goal of the PMAC platform is to provide software components that can be embedded in software applications to reduce the cost of writing applications capable of taking input from a policy-based management system.

This article provides an overview of the PMAC platform and shows with an example how it can be used in practice to manage networked IT systems and applications. In particular, we present:

- The information and system models of PMAC for policy representation, and the interaction between policy components and managed resources.
- The main components of PMAC for policy creation, policy storage, policy evaluation, and enforcement at managed resources.
- A policy ratification module that certifies a policy by taking into account its relationship with other policies in the systems. For example, a system administrator may want to know if a new policy can conflict with existing policies.

Finally, we present a case study of PMAC application in storage area network management to illustrate how policy-based management can be used in real-life scenarios.

BACKGROUND ON POLICY TECHNOLOGY

POLICY INFORMATION MODEL

To precisely specify the semantics of policy operations, a policy management system must be built on a concrete information model. The policy information model used in PMAC is inspired by the Common Information Model (CIM) policy model [2]. The CIM policy model is defined by the Distributed Management Task Force (DMTF) Policy Working Group to facilitate a unified and consistent representation of polices across a wide spectrum of technical domains, including policies related to configuration and usage of devices and applications.

In PMAC, each policy is a rule containing four components: conditions, actions, priority, and role. The conditions associated with a policy rule specify whether the policy is applicable. We say a policy is applicable when the conditions associated with the rule evaluate to true. If a policy is applicable, the set of actions associated with the policy gets executed. The priority is a nonnegative integer that indicates the relative importance of the associated policy. The priority value determines which policy must be applied when there are multiple applicable policies with potentially conflicting actions (e.g., one policy may allow access to data, while another blocks

At the highest level, PMAC provides four main components: a policy definition tool for policy authoring; policy editor storage for policy deployment and persistence; autonomic manager for policy evaluation; and a managed resource-side component (MR libraries) for policy enforcement.

it). Finally, the role defines the context in which the policy will be relevant. For example, a policy defined for mail servers may have the role "mailserver." For a detailed description of these components, we refer the reader to the PMAC documentation [3].

The System Model of PMAC

The PMAC platform supports the system model adopted by the IBM Autonomic Computing (AC) architecture, which defines a framework for self-managing IT systems [4]. The AC architecture presents two key abstractions:

- An autonomic manager (AM), which monitors computing resources, analyzes the status of the resources, plans action for the resources, and executes the planned actions
- A managed resource (MR), which is a computing system controlled and managed by the AM[1]

The relation between AM and MR is 1-to-*n*: a single AM typically controls one or more MRs, and each MR is controlled by exactly one AM. The communication between an AM and an MR is done through the MR's management interfaces, which exposes two types of hooks, *sensors* and *effectors*. The sensors are used by the AM to read the internal state of the MR, and the effectors are used by the AM to invoke actions on the MR.

In PMAC, the AM is a policy-based manager, which monitors, analyzes, and plans according to the policies that have been defined for the resources managed by the AM. In this respect, the role of the AM is similar to that of the policy decision point (PDP) as defined in RFC 3198 (for an overview and references to different policy models see [5, 6]). In fact, an AM includes the functionality of a PDP and supports additional features, such as state monitoring, event correlation, and notification, that many traditional PDPs do not provide. Likewise, there is a similarity between the managed resources and the traditional policy enforcement point (PEP) component.

We note that even though PMAC borrows the AC vocabulary, it substantially extends the bindings model of AC, which primarily relies on Web services. More precisely, the AM in PMAC exposes a set of Java application programming interfaces (APIs) that are useful when the AM and MR are running in the same Java virtual machine (i.e., the policy module is embedded as a library in applications). Alternatively, the AM can be run inside an application server and be offered as a stateless session Enterprise Java Bean (EJB) or as a Web service to remotely located managed resources. Thus, a managed resource can request policy guidance to the AM through RMI (in the EJB case) or SOAP (in the Web service case) protocols.

[1] *The AC architecture is related to International Telecommunication Union — Telecommunication Standardization Sector (ITU-T) Recommendation X.700, "Management Framework for Open Systems Interconnection," which defines the relationship between a manager and managed elements.*

PMAC Policy Middleware

An Overview of PMAC

The primary goal of PMAC is to reduce the cost of enabling software applications and IT systems to obtain guidance from policy-based management systems. More specifically, the design goals of PMAC are as follows:

- PMAC should be generic, independent of platform, software applications, and IT domains.
- PMAC should support open formats and be compliant with existing and emerging standards for service object-oriented architectures.
- PMAC should leverage existing technologies for parsing, validation, distribution, and execution of policies.
- PMAC policy should be flexible and extensible so that application-specific functions can be added.

As a result, PMAC is a generic policy middleware based on open format (XML) and standard technologies (e.g., Web Services Resource Property and J2EE), and supports a flexible and extensible policy language.

At the highest level, PMAC provides four main components: policy definition tool (PDT) for policy authoring, policy editor storage (PES) for policy deployment and persistence, autonomic manager (AM) for policy evaluation, and managed resource-side component (MR libraries) for policy enforcement. Figure 1 presents a simple illustration of the overall architecture of the PMAC platform for two PDTs, two AMs, and five MRs in three computer systems.

A policy definition tool (PDT) is a user interface by which policy authors create and modify policies. PMAC supports the concurrent use of policy definition tools to create and modify policies by multiple authors. Therefore, the consistency of distributed policies must be checked at the policy editor storage (PES). In PMAC policies are written and stored in the Autonomic Computing Policy Language (ACPL). ACPL is an XML-based policy language whose syntax closely mirrors the policy information model of PMAC. Even though PMAC includes a PDT, it is expected that most applications would provide their own definition tool integrated in the overall application user interface. In that case, the application developer can use the policy object builder component to process policies.

The PES component stores policies and policy-related metadata such as policy templates (for repeated use of similar policies). This component acts as a central repository where multiple policy definition tools store all of their policies. In turn, policy updates are pushed from a PES to autonomic managers according to their scope. Depending on the configuration and application requirement, the PES component can store policies either on a file system or in a relational database. Currently PMAC supports major database systems including DB2, Microsoft SQL, Oracle, and Cloudscape.

The autonomic manager component is the main component of PMAC. It obtains its policies from the policy editor storage and registers managed resources that are interested in receiving a policy guidance from it. In order to provide policy guidance to an MR, the AM reads the state of the MR using the sensor interface of the MR. The AM then evaluates relevant policies by using the state of the MR, and plans actions. The policy evaluation may occur either due to

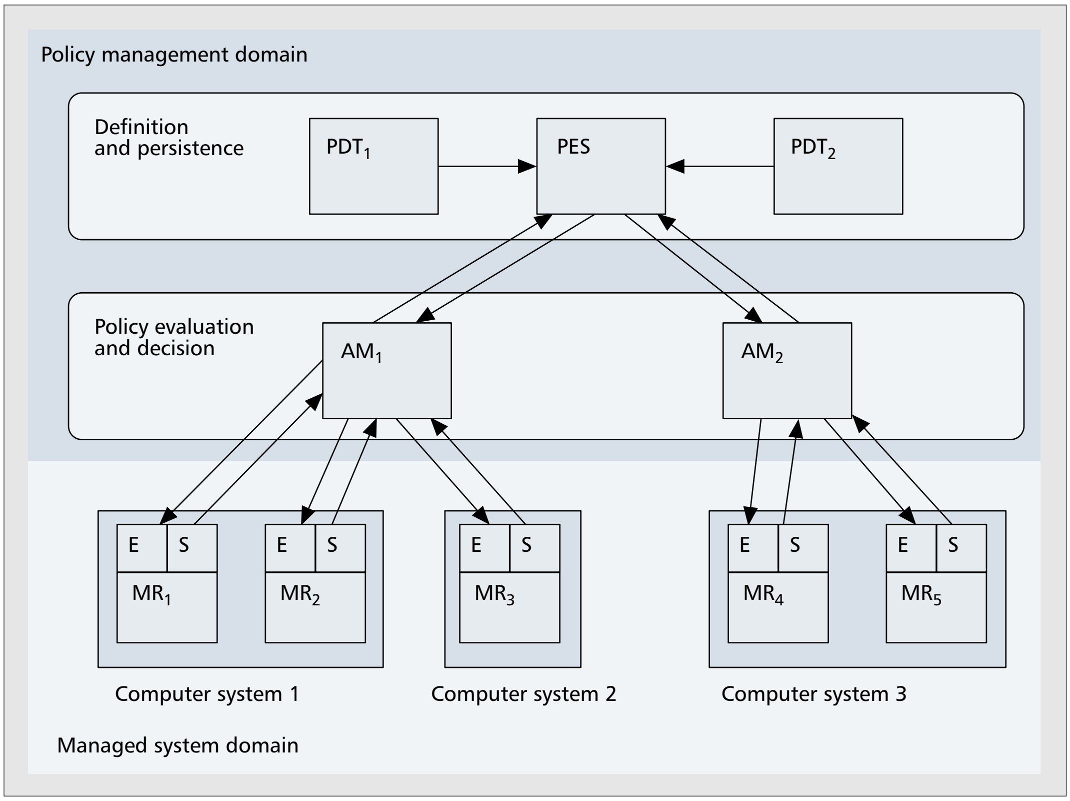

Figure 1. *PMAC architecture overview.*

At the core of ACPL is a rich expression language — the Autonomic Computing Expression Language (ACEL) — that facilitates writing policy rules. ACEL has been carefully designed so that it can express most common policy conditions while closely following standard XML conventions.

an explicit request from the managed resource for policy guidance or on a schedule determined by the AM configuration. Note that in the former case, the MR can send its state along with the request to avoid the AM coming back to read it. Based on the type of the result of policy evaluation, the AM may invoke an action on the MR via the effector interface of the MR, or simply return the result to the MR. When the MR receives directives from the AM, the MR can change its behavior in order to comply with the policy guidance.

PMAC comes with the policy object builder component as a user library. This component is capable of parsing policies written in ACPL and creating a Java Policy Object from it. The policy object builder also provides a capability to validate policies written in ACPL against the ACPL grammar using schema validation. PMAC also includes a design patterns library that implements a set of operations commonly required by certain types of applications. For example, the storage area network (SAN) management application presented later is based on the *auditor* pattern.

Autonomic Computing Expression Language

At the core of ACPL is a rich expression language, the Autonomic Computing Expression Language (ACEL), that facilitates writing policy rules [7]. ACEL has been carefully designed so that it can express most common policy conditions while closely following standard XML conventions. The result is a strongly typed language that can be parsed and type checked almost entirely by XML parsers, thereby making it attractive to applications that can consume XML format (e.g. Web services policies). Moreover, extending the language with new operations can easily be done by modifying the schema and plugging in the extension operators.

ACEL defines nine primitive types: Boolean, Short, Integer, Long, Float, Double, String, Calendar, and URI, which are directly lifted from the XML standards; and two composite types: CompositeData and Collection. CompositeData is equivalent to the XML complex type. A Collection object represents an unordered collection of ACEL expressions. ACEL also allows the definition of macro expressions.

ACEL provides various types of operators: type cast functions (e.g., `ToInt`, `ToFloat`, `ToBoolean`), standard arithmetic functions (e.g., `Plus`, `Max`, `Log`), Boolean functions (e.g., `And`, `Not`, `Equal`), string functions (e.g., `ToUpper`, `LeftSubString`), calendar operations (e.g., `GetDayOfWeek`), and operations to traverse an XML document using Xpath expressions (e.g., `XPathIntExpression`). Variables of different types can be part of an expression, but the values associated with these variables must be assigned before the evaluation of the expression.

A condition in a PMAC policy can be any ACEL expression of type Boolean with variables corresponding to sensor names. The value of the variable is the same for all occurrences of the variable in the AM policies. Variables in ACEL are represented by an XML element type called `PropertySensor` with an attribute `PropertyName` identifying the name of the variable. For

All policies written in SPL are internally translated into ACPL, and both versions, the SPL and translated ACPL, are stored in the PES. Parsing and evaluation is then done in the same manner as for policies originally written in ACPL.

example, an expression that multiplies a `Float` constant 3.14159 times the `PropertySensor` diameter would be written as:

```
<Product>
   <FloatConstant>
     <Value>3.14159</Value>
   </FloatConstant>
   <PropertySensor propertyName=
   "diameter" />
</Product>
```

We note that the XML representation of ACPL is mainly for internal processing, policy persistence, and deployment. In such cases, it is expected that policies will be created and updated using a PDT. However, for cases when a PDT is not used, PMAC also supports a simple policy language called SPL. SPL is more human friendly, and policies in SPL can easily be written using a text editor. For example, consider the following Boolean expression:

```
<And>
 <Not>
  <Equal>
    <PropertySensor propertyName=
    "NumberOfPorts" />
    <IntConstant>
      <Value>16</Value>
    </IntConstant>
  </Equal>
 </Not>
 <Equal>
    <PropertySensor propertyName=
    "VendorId" />
  <IntConstant>
      <Value>5</Value>
  </IntConstant>
 </exp:Equal>
 <Equal>
    <PropertySensor propertyName=
    "Type" />
    <StringConstant>
      <Value>Core Switch</Value>
  </StringConstant>
 </Equal>
</And>
```

Using SPL, we can write the same expression as follows:

```
(Sensor(NumberOfPorts) != 16) and
(Sensor(VendorId) = 5) and
(Sensor(Type) = "Core Switch")
```

All policies written in SPL are internally translated into ACPL, and both versions, the SPL and translated ACPL, are stored in the PES. Parsing and evaluation is then done in the same manner as for policies originally written in ACPL. The simplicity and convenience of SPL, however, comes at a cost. Theoretically, it is possible to define and implement SPL so that it provides functionality equivalent to that of ACPL. However, such an implementation will be a large undertaking since it cannot rely on standardized tools and libraries similar to those available for XML. Therefore, the PMAC implementation of SPL provides a subset of ACPL functionality.

POLICY RATIFICATION

Policies can interact with each other, often with undesirable effects; therefore, a policy administrator needs to be aware of such relations among policies. Understanding and controlling the overall effect of policies is particularly important in a distributed system, where a policy author may only have a partial view of the entire system, and multiple authors may write policies for the same set of resources without coordination.

Policy ratification is the process of certifying a policy by taking into account its relationships with other policies in the system before the policy is activated or *ratified*. In general, there are different ways to specify policies. In some cases policies are specified in a key-value pair (e.g., the configuration policy of the Microsoft Exchange server). In other cases the rules are of the form event-condition-action (when *event e* occurs if *condition c* is true then perform *action a*). In certain policy languages, policies are specified in a subject-action-target model (*subject s* must [or can or must not or cannot] perform *action s* to *target t*). The PMAC policy is an example of an event-condition-action policy. In this section we present a set of general operations that are used for policy ratification: dominance check, conflict check, and coverage check. We note that although these operations have been designed for PMAC, the fundamental ideas can be applied to other types of policy.

Dominance Check — A policy is dominated by a group of policies *S* when the addition of the new policy does not affect the behavior of the system governed by *S*. For example, a policy "passwords must be longer than 4 characters" is dominated by another policy "passwords must be longer than 6 characters" because the former policy is subsumed by the later. In another example, a policy "Joe has access to file server from 1 p.m. to 5 p.m." is dominated by another policy "Joe has access to file server from 8 a.m. to 7 p.m." From these examples, we observe that dominance checking demands capability to determine whether a Boolean expression *implies* another Boolean expression: in the first example, we need to determine that whenever (password length > 6) is true, (password length > 4) is also true, while for the second example, we need to determine the whenever (13:00 < *t* < 17:00) is true, (08:00 < *t* < 19:00) is also true.

Conflict Check — We say that two policies are in *conflict* if there are situations in which they may issue directives that cannot be achieved simultaneously. For configuration policies, two policies will conflict when they specify different configuration values: "mailbox-quota=2 GB" and "mailbox-quota=1 GB." In the event-condition-action model, a conflict between two policies may arise when the conditions of the two policies can simultaneously be true, but specify incompatible actions. For example, the policy "if a telnet connection comes after 5 p.m. then serve the connection with QoS level LOW" will potentially conflict with the policy "if a telnet connection comes from the headquarters then serve the connection with QoS level HIGH."

Therefore, the key ratification operation here is to determine whether two Boolean expressions can be made simultaneously true (i.e., they are *satisfiable*).

Coverage Check — In many application domains, the administrator may want to know if policies have been explicitly defined for a certain range of input parameters. For example, when a firewall policy has been specified in the event-condition-action model, the administrator may want to make sure that at least one policy has a true condition for the entire IP address space. In another example, when policies controlling printer queue have been specified, an administrator may want to know if the policies cover all priority classes for all days of the week and all hours of the day. The key operation in this case is to find out if a set of Boolean expressions implies another Boolean expression, where the second expression represents the value space we want to cover.

From the above observations, we can conclude that the primitive operations to support policy ratification are solving the implication and satisfiability problems of Boolean expressions. Finding general solutions for these problems is known to be computationally hard. Thus, we have taken a practical approach to identify the types of Boolean expressions that occur frequently in policy rules, and provide efficient solutions for such cases. In particular, we support:
- Boolean expressions describing equality and inequality constraints of a single variable per equality or inequality
- Boolean expressions with constraints over time intervals
- Regular expression constraints over strings
- A set of linear constraints over the real numbers

The interested reader can find details of our ratification algorithms in [8].

Conflict Resolution: The Latter Step of Ratification — When the conflict check process suggests a new policy can potentially conflict with existent policies in the system, we must resolve the conflict. A common practice to resolve conflicts is to provide the author with a mechanism to specify different priorities to conflicting policies: a policy with a higher priority has precedence over those with lower priority. In PMAC, priorities are positive integers where a greater number represents a higher priority. After a policy author is presented with a set of policies that can conflict with the new policy being ratified, the author needs to resolve the conflict by either disabling some policies or assigning a priority to the new policy. The assignment of priority values, however, may be tricky when many policies are involved in conflict. In particular, inappropriate priority assignment may require adjustment of the priority of many already installed policies. To illustrate the problem, in Fig. 2 we show policies represented by circles with numbers indicating their priorities and the arcs between them indicates conflicts. For illustration, we denote the priority relation by directed arcs where the arc is directed from a higher priority to a lower one. The node with the question mark represents a new policy about to be installed.

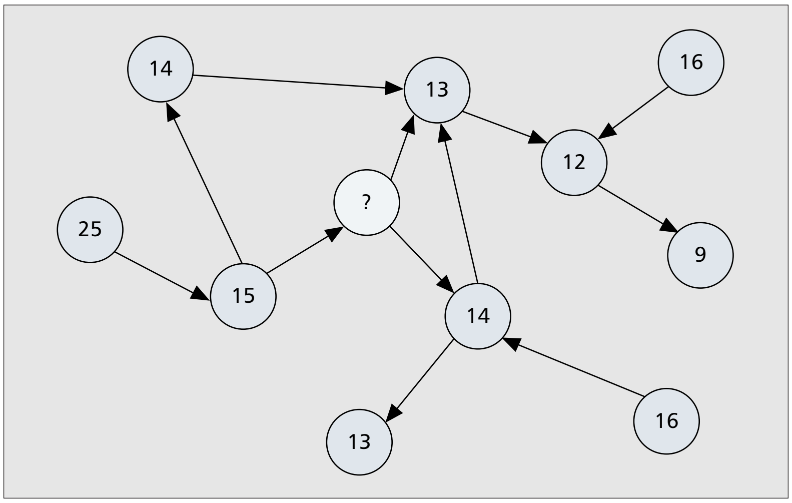

Figure 2. *Priority assignment problem.*

In this example, if we assign 14 as the priority to the new node, four policies need to change their priorities to maintain relative priority. On the other hand, if we assign 15 to the new policy, only one policy needs to change its priority. When we have a large number of policies, determining the right priority value can be nontrivial. The conflict resolution module of PMAC helps the user by automatically assigning the priority values to new policies and adjusting the values of the related policies when given only the *relative* priority of a new policy. For this, PMAC has adapted algorithms to maintain ordered lists under insertion and deletion operations where we can guarantee that, on average, the amortized reassignment of priorities is done in constant time.

There are other methods to detect and resolve conflicts at runtime using monitors and meta-policies [5]. For example, consider a case when a set of security policies and a set of service differentiation policies have been defined for a storage system. In this case, the user may simply indicate that the security policies have priorities over the service differentiation policies using meta-policies. In other cases, more recent policies may take precedence over older policies. These approaches provide more flexible means to handle policy conflicts rather than by just simply assigning priorities. Although these methods can be computationally more expensive, they are useful in certain application domains.

Case Study: Network Configuration Checking

In this section we present the configuration checking problem of storage area networks as one of the applications of policy-based system management in real life. To give a brief introduction, storage area networks (SANs) are dedicated switched networks between servers and storage so that the storage system can be shared among multiple computers. Currently, SANs are

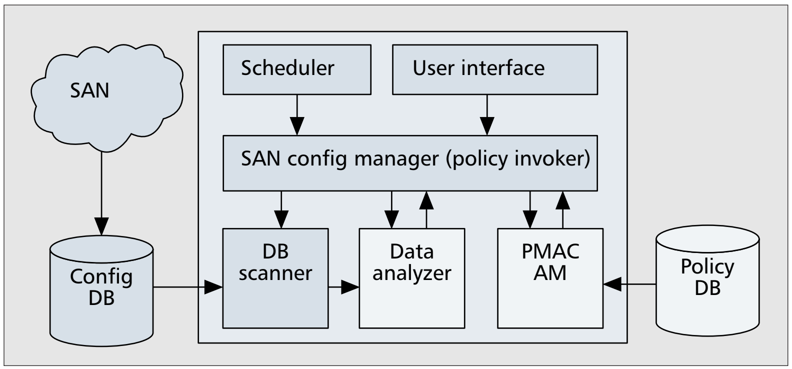

Figure 3. *Extended SAN configuration manager.*

predominantly based on the Fibre Channel protocol, which supports 1–10 Gb/s raw bandwidth. One of the main challenges in SAN management is the complexity encountered during the system setup and reconfiguration at a later time. Typically, a SAN consists of a large number of components from multiple manufacturers, and many of them have interoperability constraints with each other. For example, a storage device from a certain vendor can only work with certain types of Fibre Channel switches with certain firmware levels. In addition, over time, SAN administrators have developed best practices to avoid any problems that may arise from misconfigurations. We list a few sample best practices from field practitioners as follows:

- All zones should be configured so that the same host bus adapter (HBA)[2] cannot talk to both tape and disk devices.
- Both Windows and Linux servers should not be members in the same zone.
- Every active and connected port should be a member of at least one active zone.

For correct operation, these conditions must always be satisfied. Thus, it is important to verify that the SAN configuration is valid after adding or removing devices, upgrading firmware, and/or making changes to network configuration. It is possible to address these problems by using storage management software, which may query the underlying devices to discover their current status and detect potential configuration errors. The state-of-the-art SAN management software typically hard-codes the logic to detect configuration problems. We can enhance the SAN management system and make it more flexible and extensible by externalizing SAN configuration rules as policy [9].

Figure 3 presents an overview of a SAN configuration management system extended with PMAC. The original SAN manager system consists of the SAN configuration manager module, SAN configuration database, database scanner, user interface, and scheduler. In the target SAN environment, monitoring agents are deployed over the storage network to keep track of the status of SAN devices and configuration changes. When a configuration change happens in the SAN, it is detected and stored in the SAN configuration database. Based on a predefined schedule or a trigger from the user, the SAN configuration manager invokes a database scanner, which queries the database, identifies the configuration changes, and reads them into the SAN management system. In the original system, the configuration manager module verifies the validity of the new configuration using the internal hard-coded interoperability constraints. In the policy-enabled version, the raw configuration data is transformed into a format that can be understood by the data analyzer module (AM). The configuration manager then makes a request to the AM for policy evaluation. In effect, the configuration manager module works as a managed resource in the AC architecture. Upon this request, the AM checks whether the configuration change violates the interoperability constraints by looking up the local policy database. If a policy violation is detected, the violation is notified to the SAN administrator via various channels (e.g., log file, SAN manager console, and email to the administrator). In addition, it can trigger an action to invoke a workflow to automatically reconfigure the SAN to correct the error.

As explained earlier, PMAC can be either incorporated as a standalone service component (EJB or Web service in an application server) or embedded as a library. In this example we have shown the latter case, where the PMAC component and data analyzer have been integrated into the SAN manager system.

[2] *A host bus adapter is a Fibre Channel network interface card on the server side machine.*

SUMMARY

Policy-based network management promises to reduce the burden on the human administrator by providing systematic means to create, modify, distribute, and enforce policies for managed computing resources. PMAC is a policy middleware platform that has been developed based on the CIM policy model. PMAC features an open format extensible policy language, a standard-based flexible binding and invocation model for the managed system, both database- and file-based policy persistence mechanisms, and user support capabilities such as policy ratification. This article provides an overview of the PMAC architecture highlighting each component, and presents an example of PMAC application in storage network management to show how policy management is used in a real-life scenario.

ACKNOWLEDGMENT

The development of PMAC has been a huge collaborative effort among teams from IBM T. J. Watson Research Center, IBM India Research Laboratory, IBM Tokyo Software Laboratory, Tivoli Software, and IBM Autonomic Computing Initiative. We are grateful to everyone involved in the process that led to the release of PMAC on IBM alphaWorks [3]. We are also grateful to our colleagues at IBM Almaden Research Laboratory for their help and collaboration throughout the design and implementation of the SAN configuration checking system.

REFERENCES

[1] M. Sloman, "Policy Driven Management for Distributed Systems," *J. Net. and Sys. Mgmt.*, vol. 2, no, 4, 1994, pp. 333–60.

[2] Distributed Management Task Force, "CIM Policy Model, v. 2.8," http://www.dmtf.org/standards/cim/cim_schema_v281/CIM_Policy28-Final.pdf, Jan. 25, 2004.
[3] IBM, "Policy Management for Autonomic Computing," http://www.alphaworks.ibm.com/tech/pmac, Mar. 4, 2005.
[4] IBM, "Autonomic Computing: Creating Self-Managing Computing Systems," http://www.ibm.com/autonomic/ 2004.
[5] I. Aib *et al.*, Analysis of Policy Management Models and Specification Languages," *Network Control and Engineering for QoS, Security and Mobility II*, D. Gaïti *et al.*, Eds., Norwell, MA: Kluwer, 2003.
[6] G. N. Stone, B. Lundy, and G. G. Xie, "Network Policy Languages: A Survey and a New Approach," *IEEE Network*, vol. 15, no. 1, 2001, pp. 10–20.
[7] D. Agrawal *et al.*, "Autonomic Computing Expression Language," IBM DeveloperWorks Tutorial, Mar. 2005.
[9] D. Agrawal *et al.*, "Policy-Based Validation of SAN Configuration," *Proc. IEEE Int'l. Wksp. Policies for Distrib. Sys. and Net.*, June 2004.
[8] D. Agrawal *et al.*, "Policy Ratification," *Proc IEEE Int'l. Wksp. Policies for Distrib, Sys. and Net.*, 223-232, June 2005.

ADDITIONAL READING

[1] J. Chomicki, J. Lobo, and S. Naqvi, "Conflict Resolution using Logic Programming," *IEEE Trans. Data on Knowledge*, vol. 15, no. 1, 2003, pp. 244–49.

BIOGRAPHIES

DAKSHI AGRAWAL (agrawal@us.ibm.com) received a B.Tech. in 1993 from the Indian Institute of Technology-Kanpur (IITK), an M.S. in 1995 from Washington University, St. Louis, Missouri, and a Ph.D. in 1999 from the University of Illinois-Urbana-Champaign (UIUC), all in electrical engineering. He worked as a visiting assistant professor at UIUC during 1999–2000. After that, he joined T. J. Watson Research Center, IBM Corporation, Hawthorne, New York, as a research staff member. He was awarded a certificate of merit for securing second rank in the Indian National Mathematical Olympiads in 1988. He received the Robert T. Chen Memorial Award for 1999 for excellence in doctoral research in the Department of Electrical and Computer Engineering at UIUC. He also received the Ross J. Martin Memorial Award in 2000 for outstanding research achievement by a graduate student in the College of Engineering at UIUC. Most recently, he was awarded the IBM Research Division Award for contributing to the architecture of IBM Autonomic Computing Policy Infrastructure and Products.

KANG-WON LEE (kangwon@us.ibm.com) received a B.S. in 1992 and an M.S. in 1994 from Seoul National University in computer engineering, and a Ph.D. in 2000 from UIUC in computer science. He served the Republic of Korea Air Force during 1994–1995, and worked as a research assistant for TIMELY research group at the Center for Reliable High Performance Computing, Urbana, Illinois, during 1996–2000. In 2000 he joined IBM T. J. Watson Research Center as a research staff member. He was awarded a Magna Cum Laude from Seoul National University in 1992, Korean Government Overseas Scholarship from the Ministry of Education in 1996, KFSA Scholarship from the Korea Foundation for Advanced Studies in 1997, the C. W. Gear Outstanding Graduate Student Award from the Computer Science Department at UIUC in 1999, and the Best Student Paper Award from Packet Video Workshop in 2000. Recently, he received the IBM Research Division Award for contributing to the architecture of IBM Autonomic Computing Policy Infrastructure and Products. Currently he is secretary of IEEE Technical Community on Computer Communications.

JORGE LOBO (jlobo@us.ibm.com) joined IBM T. J. Watson Research Center in 2004. Previous to IBM he was principal architect at Teltier Technologies, a startup company in the wireless telecommunication space acquired by Dynamicsoft and now part of Cisco System. Before Teltier he was a tenured associate professor of computer science at the University of Illinois at Chicago and a member of the Network Computing Research Department at Bell Laboratories. At Teltier he developed a policy server for the availability management of presence servers. The servers were successfully tested inside two GSM networks in Europe. He also designed and co-developed PDL, one of the first generic policy languages for network management. A policy server based on PDL was deployed for the management and monitoring of Lucent first-generation softswitch networks. He has more than 50 publications in international journals and conferences in the areas of networks, databases, and AI. He is co-author of an MIT book on logic programming and is co-founder and member of the steering committee for the IEEE International Workshop on Policies for Distributed Systems and Networks. He has a Ph.D. in computer science from the University of Maryland at College Park, and M.S. and B.E. degrees from Simon Bolivar University, Venezuela.

Policy-based network management promises to reduce the burden on the human administrator by providing systematic means to create, modify, distribute, and enforce policies for managed computing resources.