Policy Management for Networked Systems and Applications

Dakshi Agrawal, Seraphin Calo, James Giles, Kang-Won Lee, Dinesh Verma
IBM T. J. Watson Research Center
19 Skyline Drive, Hawthorne, NY 10532, USA
{*agrawal,scalo,gilesjam,kangwon,dverma*}*@us.ibm.com*

Abstract

In this paper, we present a novel *policy middleware* architecture for managing IT systems and applications that span multiple networks and administrative domains. The proposed policy middleware provides a standard infrastructure for the creation, storage, distribution, and execution of policies, and helps in reducing the cost of making IT systems policy-aware. In particular, we focus on three aspects of the proposed policy middleware that help in making the middleware fully general: (1) a platform-neutral and extensible specification of policies; (2) the local ratification of policies, which lets system administrators accept, reject, or flag an incoming policy; and (3) the transformation of policies, which allows system administrators to transform incoming policies to match their local environment. We present our experience in building an application on the proposed middleware to audit the configuration of a storage area network. We also present performance results from a prototype and show that our policy middleware design can scale to handle a large number of policies.

1. Introduction

The prevalence of networked systems and applications in business environments has created an immense challenge for IT administrators. As networks become ever more heterogeneous and connected to each other, scalable and distributed management of IT infrastructure becomes imperative. To address this issue, policy-based management has been proposed in recent years. Instead of running customized scripts and manually configuring and auditing networked devices and applications, policy-based management allows IT administrators to specify high level directives or *policies* for various management tasks such as network planning, problem detection, security, and quality of service (QoS) provisions.

Policies can be specified in many different ways and multiple approaches have been suggested for different application domains [1]. KAoS [2] is a collection of componentized policy and domain management services originally designed for governing software agent behavior, and then adapted to grid computing. Rei is a policy framework that integrates support for policy specification, analysis and reasoning in pervasive computing applications [3]. Ponder is a declarative, object-oriented language that supports the specification of several types of management policies for distributed object systems [4]. Many proposals have also focused on specific system management tasks. In [5], Al-Shaer et al. have presented a set of algorithms for detecting anomalies in firewall policies. In [6], Flegkas, et al. have presented a policy-based resource management tool for capacity

planning using MPLS in the IP networks to enable differentiated services. In [7], Muruganantha et al. have proposed a policy-based system to manage the quality of service of a video streaming application. However, as the heterogeneity of devices increases and the connectivity of networks grows, an integrated approach to policy management is needed.

In this paper, we present a *policy middleware* architecture. The goal of the policy middleware is to provide a standard infrastructure for the creation, storage, distribution, and execution of end-to-end policies in many different applications and network environments. To meet this goal, the following are some of the key requirements* placed on the middleware architecture. First, the policy specification should be platform and architecture independent. Second, it should provide the local administrator convenient means to accept, reject, or flag an incoming policy. Third, it should provide means to transform policies between different system abstractions and views. The focus of this paper is on presenting solutions to these three requirements.

In particular, the paper presents a policy language design that has been architected such that a minimum amount of custom code is required to build an editor and a compiler for policies on a particular platform. This is achieved by specifying the language (as well as its possible extensions) in XML schemas [8] and by reflecting the information model of policies in the XML schemas. Note that it is not sufficient to just have an 'XML based' policy language to achieve the goal. Without a careful design, the standard XML tooling would provide capabilities that are in a mismatch with the natural operations to be performed on policies, leaving a large amount of custom coding to be done, for example, to support type checking. Our design is able to capitalize on XML tooling available on a variety of platforms by aligning XML schemas with the information model of the policies.

The second novel aspect of the proposed middleware is a built-in mechanism for the ratification of incoming policies by local administrators. In the proposed design, a local administrator can use declarative criteria to decide which incoming policies should be accepted, rejected, or flagged for further inspection. The challenge here is to make policy ratification simple, intuitive, and yet sufficiently expressive. We propose a ratification mechanism that is built as an application of the policy middleware. This recursive approach results in a small code footprint, and helps focus future research and development efforts on increasing the expressiveness of ratification criteria instead of on issues related to the management of such criteria.

The third aspect of the proposed middleware is policy transformation which is performed by specifying a mapping between the abstract view and the physical view of the managed system. For example, a higher-level service policy may mandate a certain quality of service parameter for "gold" customers. A translation of this policy to a lower-level policy may map "gold" customers to certain user-groups existing in a local shared file system. The transformation modules provided with the middleware take such mappings as their knowledge-base and transform incoming policies specified at a particular view of the system to policies specified at another view that is more relevant to the local environment.

*There are several other key requirements pertaining to the standard middleware issues such as security, scalability, and reliability. These requirements can be met by using standard solutions, e.g., the use of a database for storing policies and the use of a pub-sub system for distributing policies. Discussion of such requirements and their solutions is outside the scope of this paper.

As a real-world usage of the policy middleware, we describe an application built on it that audits storage area network to ensure its configurations matches best practices. Using this example, we illustrate storage network audit policies specified using the proposed policy language and describe policy ratification and transformation applied in this context. We have also performed performance measurement with up to 10,000 *active* policies on different platforms and different invocation scenarios. In this paper, we show the case of evaluating 1,000 active policies on Linux and Windows machines. In both cases, we observe that policy decision requests get a response in about 20 ms and the transaction rate can be as high as 50 requests/sec when the policy invocation is on local machine. On the other hand, when the requests are made to a remote policy engine using Web Services the response time is around 35 ms and the sustained transaction rate drops to 30 requests/sec.

The remainder of the paper is as follows: Section 2 presents background on policy and system modeling. Section 3 presents an overview of the proposed policy middleware. Section 4 presents the key challenges in policy specification, ratification, and transformation. Section 5 presents an example usage of the policy middleware for storage area networks validation. Section 6 discusses the performance of the policy middleware, and finally, Section 7 concludes the paper with a summary and future research direction.

2. Background

In IT systems, policies represent externalized logic that determines the behavior of managed systems and applications. To precisely specify the semantics of policy operations, the policy middleware needs to be built upon some prescribed information models of policies and managed resources. In the following subsections, we describe the information models used by the proposed policy middleware.

2.1 Policy Information Model

According to the policy core information model (PCIM), jointly developed by IETF (Internet Engineering Task Force) Policy Framework Work Group and DMTF (Distributed Management Task Force), a policy for managing a device is specified as a set of policy rules (see RFC 3060 for more details). A policy rule contains four main components: *Condition, Action, Priority*, and *Role*. The Role indicates the context in which a policy rule is relevant. The Priority is a non-negative integer which indicates the relative importance of the policy rule. In a given round of evaluation, a policy rule is *applicable* if its Condition evaluates to true. The Action part of a policy rule specifies the action to be taken if the rule is applicable.*

The policy information model implemented by our policy middleware slightly extends the PCIM model. For example, our model allows free form boolean expression in the Condition component (as opposed to the DNF (disjunctive normal form) or CNF (conjunctive normal form) assumed by the PCIM). It also specifies different subclasses of the Action component: a configuration action (potentially) changes the configuration of a managed device, a goal action sets a goal for a managed device, etc. The objective of these extensions is to make the policy expression more flexible and semantically richer.

*Henceforth, in this paper, we will use the terms policy rule and policy interchangeability.

2.2 System Models

The proposed policy middleware defines linkages to service-oriented as well as object-oriented architecture to effectively specify, distribute, and execute end-to-end policies in a distributed environment. In our prototype implementations, we have focused on providing full binding to the Autonomic Computing Reference architecture* In this paper, as a reference, we use terms defined in the Autonomic Computing architecture [9]

The Autonomic Computing architecture defines two fundamental abstractions: an *autonomic manager* (AM) and a *managed resource* (MR). An AM controls one or more MRs and each MR is controlled by exactly one AM. An AM communicates with a MR through MR's *manageability interface*. Operations in the manageability interface are divided into *sensors* and *effectors*. An AM can read the internal state of an MR via the sensors of the MR, and it can invoke actions to an MR via the effectors of the MR. An AM and an MR managed by it implement a *control-loop*: the AM monitors the MR state using the MR sensors, it analyzes the MR state, and plans actions based on the assigned goals and objectives, and finally executes these actions using the MR effectors which in turn affect the MR state.

A policy-based AM uses policies to analyze the MR state and plan actions. It includes the traditional Policy Decision Point (PDP) functions of making policy decisions; however it goes much beyond that. For example, it supports state monitoring, event correlation, notification etc. Similarly, an MR includes the traditional Policy Enforcement Point (PEP) functions. It is important to appreciate the distinction between an AM and a traditional PDP. In particular, we note that in a distributed system, there would be several AMs each managing a set of MRs and communicating with each other. An AM is the focal point of analysis and planning (instead of just being a piece of it) for a particular context and system administrators interact with AMs to manage their resources.

3. Policy Middleware Architecture Overview

Figure 1 shows an architectural view of the policy middleware consisting of an editing tool, federator, autonomic manager, and managed resource. The policy federator provides a central hub for a publish/subscribe service for the policies, and it also provides repository functionality for the specification of any extensions to the policy language and system models (for example, WSDL specification of services provided by an IT subsystem).

In a simple deployment, a policy author writes policies using GUI-based policy editing tools. The policy editing tool obtains relevant information (which sensors and effectors are exposed by the MRs, their types, etc.) from the policy federator for the class of MRs for which the author wants to write policies. After a new policy is committed to the policy federator, there are several possible ways in which the policy can be deployed in the system. An AM may choose to contact the policy federator at scheduled times and receive all policy updates, or it may choose to receive notifications from the policy federator

*The Autonomic Computing Initiative by IBM [9] is a leading service-oriented architecture built around web-services. We currently also have discipline specific bindings to the CIM management framework (for Storage Network Industry Association CIM specification) and we are working towards generalized binding to the CIM framework.

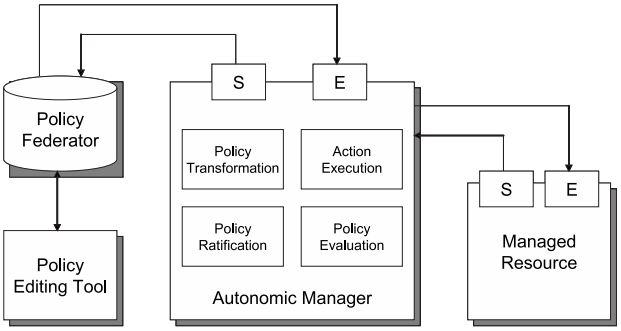


Figure 1: Policy Middleware Basic Architecture

whenever a policy update is available. The autonomic managers may even be configured to accept policies pushed to them by the policy federator. The *Role* field of policies is used in subscriptions or requests to determine which policies an autonomic manager receives. Once an AM receives a policy, it performs policy ratification. As a result it may reject or ignore a policy if the policy is deemed ineffective or conflicting. All accepted policies then undergo a transformation process where a higher-level policy may be converted into lower-level policies.

At the time of policy evaluation, one or more policies may need to be evaluated. The AM gathers sensor values required for the evaluation of these policies. Some of the sensor values may be provided by the MRs in their request for policy guidance, and the AM may obtain the rest by querying MRs. After all the required sensor values have been obtained, a policy evaluation engine is used to evaluate which policy conditions are true. If more than one policy have true conditions, then we consider the policies to be in conflict because the action of one policy may contradict the action of another policy. In the proposed architecture, these conflicts are resolved by an external conflict resolution module using domain specific semantics. Finally, the actions indicated by resolved policies are executed by an action executor within the AM. At present, we have implemented two linkage models, local Java calls (for the local case when both AM and MR are running in the same Java virtual machine) and web-services (for the remote case).

4. Architectural Features and Implementation Details

4.1 Policy Language

The proposed policy middleware defines a policy language which provides syntactic constructs to represent the policy information model. The policy language is based on XML, and it uses XML schema [8] to define and validate the policy syntax. The proposed policy language syntax can be divided into two parts. The first part represents high-level constructs that describe constituents of a policy such as Condition and Action. This part mirrors the policy information model (in as much detail as is possible within the constraints of XML schema) so that standard tools for parsing XML can provide fragments that are readily used by the class stubs (for different programming languages) to cre-

ate instances of policy objects (obtained from the policy information model rendered in UML). This minimizes the custom codes required to edit and compile policies in different programming languages.

The second part consists of an optional expression language. Expressions are used in defining condition statements, arguments for actions, and the priority of a policy. The proposed expression language is strongly typed, and it provides the following datatypes: `Integer`, `Float`, `Long`, `Short`, `Double`, `Boolean`, `String`, `Calendar`, and `Composite`. First seven types respectively correspond to XML schema datatypes of `int`, `float`, `long`, `short`, `double`, `boolean`, and `string`. The `Calendar` datatype implements the Java time specification, which is equivalent to the `dateTime` XML schema datatype except that it allows for a more generic timezone specification. The datatype `Composite` is equivalent to XML schema complex types. In addition to these types, the language also supports a datatype `Collection` which represents a collection of element expressions. The expressions provided by the proposed language represent *variables* such as the sensors of a MR as well as *functions* and *constants*.

Corresponding to the data types, the language supports the following substitution groups* of functions: `NumericExpression`, `BooleanExpression`, `StringExpression`, `CalendarExpression`, `CollectionExpression`, and `CompositeExpression`. For each of the listed groups, members are variables and constants of the type corresponding to the group, and functions that return results of the type corresponding to the group. In the following we list a subset of the standard functions supported by the policy middleware:

1. *Numeric functions*: Type cast functions (*ToInt, ToFloat, ToLong,* etc.), Arithmetic operators (*Plus, Product, Remainder, Max, Log, Pow, Ceiling,* etc.), Calendar field extraction functions (*GetDayOfMonth, GetHour, GetYear,* etc.)
2. *Boolean functions*: Type cast function (*ToBoolean*), Logical functions (*And, Or, Not, Xor*), Relational functions (*Greater, Equal, GreaterEqual,* etc.), String matching functions (*Begins, Contains, Ends*), Calendar comparison functions (*IsAfter, IsBefore, IsWithin*), Set functions (*Belongs*)
3. *String functions*: Type cast functions (*ToString*), Substring extraction functions (*LeftSubstring, RightSubstring, ReplaceSubstring,* etc.), Case conversion functions (*ToUpper, ToLower*)
4. *Collection functions*: Set functions (*Union, Intersection,* etc.)

Due to the lack of space we omit the descriptions of the functions, since the meanings of most of them are quite obvious from their names. Although our decision has been to express the policies in XML for its flexibility, open format, and the availability of standard tools (e.g. parser and schema validator), we understand creating and editing policies in XML is not very appealing for human administrators. Thus we provide a GUI-based policy editor and a wizard-like authoring tool to ease the process of policy writing and editing.

---

*An XML substitution group allows XML elements to be substituted for a given element wherever the given element is valid in an XML fragment. If a function `MyExtendedFunction` has `NumericExpression` as its substitution group, then `MyExtendedFunction` can appear anywhere that a `NumericExpression` can appear in an XML document.

4.2 Language and Functional Extensibility

Our policy language provides generic data types and functions to express complex arguments needed for specifying policies. However, certain deployers may want to have new data types or functions that are specific to their applications (see the case study in Section 5). Consequently, the policy language and corresponding middleware implementation should be extensible. To the best of our knowledge, this problem has not been addressed by the earlier attempts since policy languages are often tightly coupled with a discipline.

The proposed policy language allows deployers to add new types and functions by producing a new XML schema under a new namespace, which extends the original policy middleware schema by importing it in the new schema.* When writing a new policy that uses new data types and extended functions, the deployer makes a reference to the new extended schema.

In the policy middleware, the transparency of the new extensions is achieved by representing expressions as trees with variables and constants as leaf nodes, and operators as intermediate nodes. The tree structure is reflected in the XML schemas for both the built-in and extended expressions. The middleware uses a global expression factory, called the `GlobalExpressionFactory`, and an `Expression` interface which represents an expression tree. The `GlobalExpressionFactory` implements the following interface:

```
public interface ExpressionFactory {
   public Expression CreateExpression(Element element);
}
```

At a high level, the `GlobalExpressionFactory` takes an XML element that represents an expression in its XML form, and returns an object implementing the `Expression` interface. However, for extensibility, the `GlobalExpressionFactory` does not itself possess the knowledge needed to make this conversion. Instead, the `GlobalExpressionFactory` calls a namespace-specific expression factory that knows how to convert XML elements in their namespaces into the instances of `Expression`. Namespace-specific expression factories resolve an XML element into an `Expression` instance by recursively calling the `GlobalExpressionFactory` for each of the element's children nodes. Thus, the deployer does not need access to the source code of the middleware and vice-versa to enable expression extensions and the deployer only needs to implement the extension logic. By implementing the extensibility via XML schema import, we can leverage off-the-self libraries for building graphical editors, parsers, and validation engines even for the extended functions.

4.3 Policy Ratification

In a distributed system, it is likely that a policy author would have only a partial view of the entire system, and therefore she may write policies that are either irrelevant in some local environment, are dominated by other policies,* or are conflicting with already ex-

*We note that extensibility is not a unique feature to languages; most programming languages support language extension in one way or the other (e.g., standard libraries). However, the challenge lies in the fact that such an extension is not explicitly supported by XML. Thus we had to provide a mechanism to extend the basic policy schema by defining a clean extension point in our grammar.

*We say a policy is *dominated* by a group of policies if whenever the dominated policy is applicable, the group contains at least one applicable policy with a higher priority.

isting local policies. For example, a policy author may define a policy for Cisco switches. However, when the policy is deployed, the local domain may not have Cisco switches, or the existing local policies for network switches may conflict with the incoming policies on Cisco switches. The goal of policy ratification is to provide local administrators control over which policies to deploy when certain policies are not necessary or when certain policies conflict with each other.

In the proposed architecture, our approach is to provide a generic ratification module that can be tailored by a local administrator to their specific requirements by using declarative criteria specified as *ratification policies*. For example, a ratification policy could state that "if the input policy is applicable only in the context of security against Windows worms and viruses, then do not ratify the input policy" since the local environment is based exclusively on the Linux operating system. In our design, defining and managing ratification policies is exactly the same as the case with "regular policies." The only difference is that ratification policies are written against a model of incoming policies, whereas regular policies are written against a model of managed resources.

Using policies for the ratification of new policies has many advantages. First, since ratification policies themselves conform to the policy information model and are written in the proposed policy language, policy editing and verification tools can be reused for authoring ratification policies. Second, the local administrator can use a policy federator to store and distribute ratification policies. Finally, when a new policy is added to the system, or an existing one is modified, the ratification process is equivalent to evaluating and executing policies. Essentially, the generic ratification module becomes an application built on the policy middleware components. This frees us to focus on providing functions that are useful to express ratification criteria.

Functions to Support Ratification: For a given scope and context, sensors define a *sensor value-space*. A corresponding policy carves out an *applicability subspace* of the sensor value-space where its condition is true. For example, a sensor named `DayOfWeek` may have a value-space corresponding to {`Monday, Tuesday, ..., Sunday`} anda policy applicable for weekends may have an applicability subspace {`Saturday, Sunday`}. Thus, determining if a policy is dominated by a group of policies is equivalent to determining if the applicability subspace of the given policy is a subset of the applicability subspaces of policies with a higher priority. Similarly, checking whether two policies can simultaneously be applicable is equivalent to checking if the applicability subspaces of these policies intersect with each other. Finally, checking if a policy is not applicable in a local environment is equivalent to checking that the scope of the local environment does not intersect with the scope of the policy.

There are many challenges in determining whether applicability subspaces of two policies intersect with each other. First, it may not be possible to compute applicability subspaces explicitly–instead it may only be implicitly defined in terms of non-linear functions. Second, the policy language provided by the middleware contains several primitive data types and a complex data type, which have different value spaces.* The state of the art

*Of these data types, `Double` and `Float` have real numbers as their value-space; `Short`, `Integer`, and `Long` have integers as their value-space; `String` has a totally-ordered discrete domain; `Calendar` has a totally-

is that there is no generic algorithm that can determine whether two applicability spaces defined over such diverse domain intersect or not.

The policy middleware takes the approach of providing some commonly used functionality out of the box. In particular, it provides functions to find intersection of four different types of applicability subspaces:

- Applicability subspaces that are defined by *linear constraints in an n-dimensional real space*. Current algorithms in the middleware use a modified simplex algorithm to determine whether two linearly constrained subspaces intersect each other or not.
- Applicability subspaces that are defined by a *single variable per inequality linear constraints* in the form of $a \cdot x + b \ \triangleright \ c$, where x is a variable, a, b, c are constants, and $\triangleright \in \{=, <, \leq, >, \geq\}$. The variable and constants in each inequality are of type `Integer, String, Calendar`. The algorithm used in this case simply restricts the domain of each sensor after examining an inequality until either all inequalities have been examined, or until the domain of a sensor becomes empty.
- Applicability subspaces that are defined by equalities and disequalities on `Composite` or `Boolean` data types in the form of $x \triangleright c$, where x is a variable, c is a constant, and $\triangleright \in \{=, \neq\}$. In this case, the algorithm works by creating equivalence classes until either all equalities and disequalities have been examined or a contradiction has been found.

We note that the above cases have been derived from our preliminary efforts to tackle the most practical scenarios. Finding systematic approaches to solve more general form of the subspace constraint problem is a topic of our ongoing research.

4.4 Policy Transformation

As previously mentioned in Section 3, policies can be defined by multiple policy authors with different levels of abstract views on the system to be managed. For example, at the highest level, the policies may be specified in terms of "platinum", "gold", "silver" classes of services, whereas at a lower level these policies may be translated into setting up firewall rules, enforcing file access control, or sending data via a more secure channel. Multiple levels of abstraction provide benefits by allowing the policy authors to write policies using the most relevant and meaningful parameters at their levels. Also it makes the job of policy management easy by allowing the policies at one level to be changed without affecting the policies at another level.

Policy transformation supports *policy refinement* by allowing policy authors and administrators to translate one policy to one or more other policies. The policy middleware provides a generic transformation module that can be tailored by the local administrator by using *transformation policies*. Similar to the case of policy ratification, the local administrator can write a transformation policy in the following form: "if certain conditions on a policy are satisfied (e.g. if the Role is `WindowsSecurity`) then update a certain portion of the policy with a new entity (e.g. replace the Condition and the Action by a new Boolean expression)." Using policies for transforming policies has the same benefits as using policies for ratification—a large amount of infrastructure to support transformation can be provided by the already existing middleware.

In our experience with building applications with the policy middleware, we often

---

ordered continuous-domain; and `Composite` and `Boolean` have a domain with only an equality relation (i.e., no ordering is defined).

find that for a given context, the *form* of policies is fixed and well-known. The form of a policy specifies constraints on the 4-tuple of the policy. For example, a particular form of policies may dictate that the Boolean expression in the condition can only be of logical operations and equalities, and the action can only be sending certain configuration parameters to the managed resource. An author of transformation policies would exploit this knowledge to map higher-level policies into lower-level policies. To ease the specification of transformation policies, we add the following functional extensions: `ContainsSubexpression(Expression e1, Expression e2)`, which checks to see if `e2` is contained in `e1` or not, and `ReplaceSubexpression (Expression e, Expression e1, Expression e2)`, which replaces all the instances of `e1` in `e` with `e2`.

5. Case Study: Storage Network Management

Internally, we have demonstrated the utility of policy middleware in various contexts including network planning, configuration validation, and resource virtualization. In this section, we present a brief overview of how the policy middleware could be used for validating the configuration of a storage area network (SAN).

5.1 SAN management problem

Storage area networks allow storage to be shared among multiple servers by providing fast interconnects to storage devices. One of the main challenges of SAN management is the complexity of system configuration and the large number of configuration problems that are encountered when (a) adding or removing devices, (b) provisioning new storage, and (c) upgrading firmware or device drivers. These SAN configuration problems can be addressed to a certain degree by using SAN management software, which queries devices to discover their current status and detects potential configuration problems. The state-of-the-art in SAN management software typically implements the configuration policies inside the internal logic or retrieves hard-coded policies from a repository. We can enhance the SAN management system using our policy middleware so that the SAN policies are externalized and managed in a flexible manner.

5.2 Planning Steps

Step 1: Collect sample policies to be implemented in plain English. For the SAN validation application, we first collected SAN configuration rules and device interoperability constraints widely used by the SAN administrators. Below we present some sample policies. A comprehensive list of 64 different types of SAN configuration policies has been reported in [10].

Step 2: Translate the plain English policies into the formal policy language. At first look, it may seem that the SAN configuration policies do not present a structure for parsing and evaluation. However, more insight can be gained by converting them into the formal representation using the policy language. For example, a English language policy “All

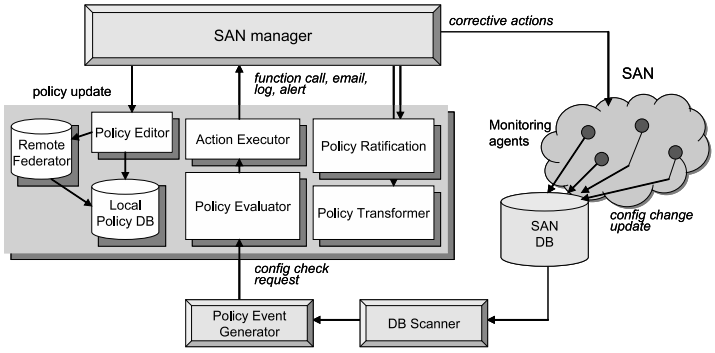


Figure 2: SAN management system extended with Policy Middleware

host bus adapters (HBAs)* of vendor Emulex must have firmware level of either 3.81a or 3.81b." can be restated using the language constructs as follows:

- *Role:* Host Bus Adapter
- *Condition:* $\exists h \in H \; s.t. \; (h.vendor = \text{Emulex}) \wedge h.version \notin \{3.81a, 3.81b\}$, where H represents the set of HBAs.
- *Action:* Send alert to the administrator.
- *Priority:* High

Step 3: If needed, extend the managed resource or the policy middleware to support the new data types and operations identified in Step 2. It turns out that in the SAN management application, the generic operators provided by the policy language cannot express all the conditions of the sample configuration policies. In particular, we identified that four additional operations on a collection of elements are needed.

1. **Cartesian Property:** Return all elements in a collection C that satisfy a Boolean expression.
   – Example: All HBAs of vendor Emulex must have firmware level of either 3.81a or 3.81b.
2. **Exclusion:** Return all elements in a collection C that satisfy a Boolean expression if another element in the collection satisfies another Boolean expression.
   – Example: Tape drives should not exist in a zone if it contains disk drives.
3. **Many-to-One:** The value of an attribute p_i should be the same for all elements in C.
   – Example: All HBAs in a host logical partition must be the same model from the same vendor.
4. **One-to-One:** The value of an attribute p_i should be different for all elements in C.
   – Example: No two devices in the system can have the same WWN (World-Wide Name).

In [10], we present a comprehensive list of SAN configuration policies and show how each policy type can be modeled using the above four collection operations. Once this planning stage is complete and the policy middleware is primed with necessary extension modules to evaluate extended functions, the middleware is ready to provide policy guidance to the SAN management system.

5.3 Policy-Based SAN Management System

We now illustrate the operation of the policy-based SAN management system. Figure 2 presents an overview of a SAN management system extended using the proposed policy middleware. In the figure, the policy middleware components are represented as white

*Host bus adapters play a similar role to network adapters and allow hosts to connect to the storage network.

boxes. The original SAN management infrastructure consists of the SAN manager, monitoring agents, SAN DB, DB scanner, and event generator. In the target SAN environment, monitoring agents are deployed over the storage network to keep track of the status of hosts, switches, storage, and the interconnection between them. When any configuration change happens in the SAN, it is reported and stored at the SAN DB using a format consistent with the Storage Management Initiative Specification (SMI-S) representation [11]. Periodically or triggered by DB update events, the DB scanner module queries the database, identifies the configuration changes, and filters them to construct a report to the event generator module. In the extended SAN manager, the event generator effectively functions as managed resource and sends configuration check requests to the policy evaluator via the policy middleware APIs.

The Policy Evaluator takes a list of roles associated with an event from the event generator and selects the corresponding policies from the local policy DB. The local policy DB can be updated by the policy editor tool invoked by a human operator, or from a remote policy federator using a method similar to the automatic update of anti-virus profiles. For policy evaluation, the evaluation engine identifies the SAN attributes referenced by the policies and requests those attributes from the SMI-S Database Interface. The gathered attributes are used to compute the condition of each policy. If the condition of a policy is *true* then the action portion of the policy is passed to the action execution module. The action execution module in turn generates actions (e.g., send an alert to console, invoke a workflow to automatically reconfigure the SAN) via an interface between the policy middleware and the SAN manager application.

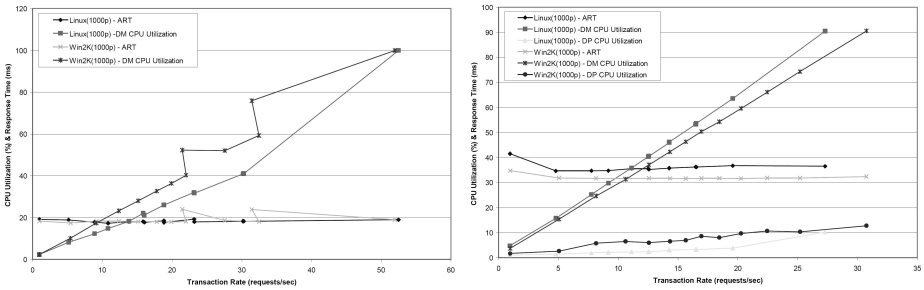
When the policies are updated or pushed from the policy federator, the policy middleware performs policy ratification on the incoming policies. In the SAN management system, policy ratification can be used to select only the relevant policies for the system. For example, the deployed SAN managed by policy middleware may not have any Brocade switches. Therefore policies related to Brocade SAN fabric switches are not relevant to this particular SAN environment. The local administrator can specify this point in a ratification policy defined in Section 4.3.

Finally, the policy transformation module may be triggered to convert high level policies into policies with resource-level details or to refine higher level policies. For example, an incoming policy may be concerned with certain interoperability problems between Brocade and Cisco switches. However, the local administrator may have applied a patch that takes care of some of the known problems. In such a case, the local administrator may specify a transformation policy that refines the condition of the incoming policies by adding a clause about the problems that have been locally taken care of.

6. Performance Evaluation

In this section, we briefly present performance result of policy evaluation of the proposed policy middleware. We note that this implementation has not been tuned for performance. However, our initial result shows that the proposed architecture can effectively handle large numbers of active policies (up to 10,000 policies).

Figure 3(a) presents the CPU utilization and response time of the policy-based au-



(a) Local (b) Web Service

Figure 3: CPU utilization and response time with 1,000 policies

tonomic manager (AM) when 1,000 policies are evaluated at a local AM. For this experiment, we used a 2.4 GHz Pentium 4 machines with 1GB of memory. The x-axis represents the transaction rate (requests/second) and the y-axis represents the CPU utilization (%) and response time (ms). For both the Windows and Linux implementation, we observe that the performance characteristics are similar. In particular, we observe that the response time is relatively stable at around 20 ms, and the transaction rate of up to 50 requests/sec can be supported. Figure 3(b) presents the CPU utilization of AM and managed resource (MR) and the response time of policy evaluation when 1,000 policies are evaluated at a remote AM that is accessed via Web Service. We used the same system configuration as the local case, and the figure shows the response time (ms), and the CPU utilization of both the AM (represented as DM) and MR (represented as DP) in the figure. From the figure we observe similar performance characteristics for both Windows and Linux. However the sustainable transaction rate is smaller than the local AM case (up to 30 requests/sec) and the response time is also a bit higher at around 35 ms.

In addition, we have performed extensive testing with various configurations and settings. In general, we observe that CPU utilization has correlation with both the number of policies and the transaction rates; but memory utilization is mostly correlated with the number of policies.

7. Conclusion

Policy middleware provides a standard infrastructure for policy management, and makes it easier to policy-enable networked systems and applications. It is our conclusion that system building aspects, in particular, platform independence and interface to the system administrator, are the key challenges in policy management. In this paper, we outlined an architecture for policy middleware, and described three of its key features: language for policy specification, mechanisms for policy ratification, and mechanisms for policy transformation.

The key for policy middleware to be successful lies in gathering deployer requirements and implementating features most requested by the deployers. We are currently engaged with a wide range of policy deployers to find the most important middleware features, and, in the future, we plan to develop additional technologies to implement such features. In

particular, we are studying the feasibility of implementing policy ratification techniques that can handle applicability subspaces specified by *two variables per inequality* constraints using Constraint Logic Programming (CLP). We are also studying the possibility of policy authoring tools that can be extended based on an extension schema. To help policy authors understand how a new policy would interact with other policies in the system, we are developing policy-impact analysis techniques and policy adviser tools.

Acknowledgment

The authors are grateful to Arlindo Chiavegatto, Tom Ellingwood, Allen Gilbert, and David Kaminsky for numerous discussions on policy management. The authors enjoyed working closely with middleware deployers Khalid Filali-Adib, Surendra Maheswaran, and Kaladhar Voruganti. Mandis Beigi, David Olshefski, and Xiping Wang implemented several modules of a research prototype of the policy middleware. Katie Nguyen provided performance results of policy evaluation.

References

[1] Wright, S., Chadha, R., Lapiotis, G., (eds.): Special issue on policy based networking. IEEE Network **16** (2002)

[2] Uszok, A., Bradshaw, J., Jeffers, R., Suri, N., Hayes, P., Breedy, M., Bunch, L., Johnson, M., Kulkarni, S., Lott, J.: KAoS policy and domain services: toward a description-logic approach to policy representation,deconfliction, and enforcement. In: IEEE Policy 2003. (2003) 93–96

[3] Kagal, L., Finin, T., Johshi, A.: A policy language for pervasive computing environment. In: IEEE Policy 2003. (2003) 63–76

[4] Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder policy specification language. In: IEEE Policy 2001. (2001)

[5] Al-Shaer, E., Hamed, H.: Firewall policy advisor for anomaly discovery and rule editing. In: IEEE IM 2003. (2003)

[6] Flegkas, P., Trimintzios, P., Pavlou, G., Liotta, A.: Design and implementation of a policy-based resource management architecture. In: IEEE IM 2003. (2003)

[7] Nathan Muruganantha, H.L.: Policy specification and architecture for quality of service management. In: IEEE IM 2003. (2003)

[8] Brown, A., Fuchs, M., Robie, J., Wadler, P.: XML Schema: Formal description. http://www.w3.org/TR/xmlschema-2/ (2001)

[9] I.B.M.: Autonomic computing: Creating self-managing computing systems. http://www.ibm.com/autonomic/ (2004)

[10] Agrawal, D., Giles, J., Lee, K.W., Voruganti, K., Filali-Adib, K.: Policy-based validation of SAN configuration. In: IEEE Policy 2004. (2004)

[11] Storage Networking Industry Asociation: SNIA storage management initiative specification. http://www.snia.org/ smi/ tech_activities/ smi_spec_pr/ spec/ SMIS_v101.pdf (2003)