Policy Based Self-Management in Distributed Environments

Lin Bao, Ahmad Al-Shishtawy, and Vladimir Vlassov
Royal Institute of Technology
Stockholm, Sweden
{linb, ahmadas, vladv}@kth.se

***Abstract*—Currently, increasing costs and escalating complexities are primary issues in the distributed system management. The policy based management is introduced to simplify the management and reduce the overhead, by setting up policies to govern system behaviors. Policies are sets of rules that govern the system behaviors and reflect the business goals or system management objectives.**

This paper presents a generic policy-based management framework which has been integrated into an existing distributed component management system, called Niche, that enables and supports self-management. In this framework, programmers can set up more than one Policy-Manager-Group to avoid centralized policy decision making which could become a performance bottleneck. Furthermore, the size of a Policy-Manager-Group, i.e. the number of Policy-Managers in the group, depends on their load, i.e. the number of requests per time unit. In order to achieve good load balancing, a policy request is delivered to one of the policy managers in the group randomly chosen on the fly. A prototype of the framework is presented and two generic policy languages (policy engines and corresponding APIs), namely SPL and XACML, are evaluated using a self-managing file storage application as a case study.

I. Introduction

To minimize complexities and overheads of distributed system management, IBM proposed the Autonomic Computing Initiative [1], [2], aiming at developing computing systems which can self-manage themselves. In this work, we address a generic policy-based management framework. Policies are sets of rules which govern the system behaviors and reflect the business goals and objectives. Rules define management actions to be performed under certain conditions and constraints. The key idea of policy-based management is to allow IT administrators to define a set of policy rules to govern behaviors of their IT systems, rather than relying on manually managing or ad-hoc mechanics (e.g. writing customized scripts) [3]. In this way, the complexity of system management can be reduced, and also, the reliability of the system's behavior is improved.

The implementation and maintenance of policies are rather difficult, especially if policies are "hard-coded" (embedded) in the management code of a distributed system, and the policy logic is scattered in the system implementation. The drawbacks of using "hard-coded" and scattered policy logic are the following: (1) It is hard to trace policies; (2) The application developer has to be involved in implementation of policies; (3) When changing policies, the application has to be rebuilt and redeployed that increases the maintenance overhead. In order to facilitate implementation and maintenance of policies, a language support, including a policy language and a policy evaluation engine, is needed.

This research was partially supported by the FP6 project Grid4All (contract IST-2006-034567) funded by the European Commission.

This paper presents a generic policy-based management framework which has been integrated into Niche [4], [5], a distributed component management system for development and execution of self-managing distributed applications. The main issues in development of policy-based self-management for a distributed system are programmability, performance and scalability of management. Note that robustness of management can be achieved by replicating management components. Our framework introduces the following key concepts to address above issues: (1) Abstraction of policy that simplifies the modeling and maintenance of policies; (2) Policy Manager Group that allows improving scalability and performance of policy-based management by using multiple managers and achieving good load balance among them; (3) Distributed Policy-Manager-Group Model that allows to avoid centralized policy decision making, which can become a performance bottleneck. We have built a prototype of the policy-based management framework and applied it to a distributed storage service called YASS, Yet Another Storage Service [4], [6] developed using Niche. We have evaluated the performance of policy-based management performed using policy engines, and compared it with the performance of hard-coded management.

The rest of the paper is organized as follows. Section II briefly introduces the Niche platform. In Section III, we describe our policy based management architecture and control loop patterns, and discuss the policy decision making model. We present our policy-based framework prototype and performance evaluation results in Section IV followed by a brief review of some related work in Section V. Finally, Section VI presents some conclusions and directions for our future work.

II. Niche: A Distributed Component Management System

Niche [4], [5] is a distributed component management system for development and execution of self-managing distributed systems, services and applications. Niche includes a component-based programming model, a corresponding API, and a run-time execution environment for the development,

deployment and execution of self-managing distributed applications. Compared to other existing distributed programming environments, Niche has some features and innovations that facilitate development of distributed systems with robust self-management. In particular, Niche uses a structured overlay network and DHTs that allows increasing the level of distribution transparency in order to enable and to achieve self-management (e.g. component mobility, dynamic reconfiguration) for large-scale distributed systems; Niche leverages self-organizing properties of the structured overlay network, and provides support for transparent replication of management components in order to improve robustness of management.

Niche separates the programming of functional and management (self-*) parts of a distributed system or application. The functional code is developed using the Fractal component model [7] extended with the concept of component groups and bindings to groups. A Fractal component may contain a client interface (used by the component) and/or a server interface (provided by the component). Components interact through bindings. A binging connects a client interface of one component to a server interface of another component (or component group). The component group concept brings on two communication patterns "one-to-all" and "one-to-any". A component, which is bound to a component group with a one-to-any binding, communicates with any (but only one) component randomly and transparently chosen from the group on the fly. A component, which is bound to a group with a one-to-all binding, communicates with all components in that group at once, i.e. when the component invokes a method on the group interface bound with one-to-all binding, all components, members of the group, receive the invocation. The abstraction of groups and group communication facilitates programming of both functional and self management parts, and allows improving scalability and robustness of management.

The self-* code is organized as a network of distributed management elements (MEs) (Fig. 1) communicating with each other through events. MEs are subdivided into Watchers (W), Aggregators (Aggr), Managers (Mgr) and Executors, depending on their roles in the self-* code. Watchers monitor the state of the managed application and its environment, and communicate monitored information to Aggregators, which aggregate the information, detect and report symptoms to Managers. Managers analyze the symptoms, make decisions and request Executors to perform management actions.

III. Niche Policy Based Management

A. Architecture

Fig. 2 shows the conceptual view of policy based management architecture. The main elements are described below.

A Watcher (W) is used to monitor a managed resource[1] or a group of managed resources through sensors that are placed on managed resources. Watchers will collect monitored information and report to an Aggregator.

[1]Further in the paper, we call, for short, *resource* any entity or part of an application and its execution environment, which can be monitored and possibly managed, e.g. component, component group, binging, component container, etc.

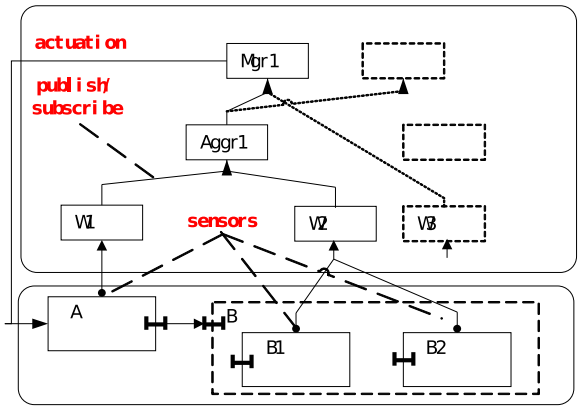

Fig. 1. Niche Management Elements

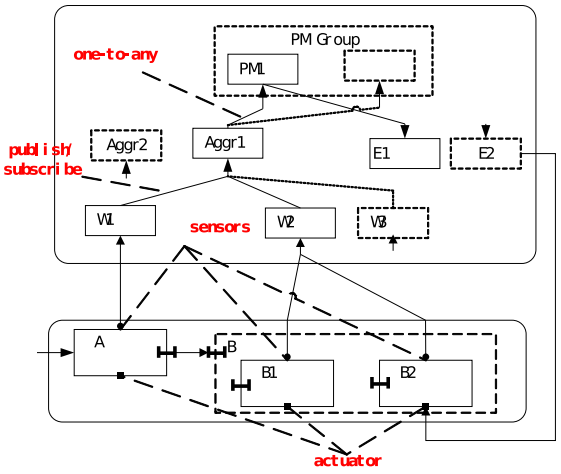

Fig. 2. Policy Based Management Architecture

Aggregators (Aggr) aggregate, filter and analyze the information collected from Watchers or directly from sensors. When a policy decision is possibly needed, the aggregator will formulate a policy request event and send it to the Policy-Manager-Group through one-to-any binding.

Policy-Managers (PM) take the responsibility of loading policies from the policy repository, making decisions on policy request events, and delegating the obligations to Executors (E) in charge. Obligations are communicated from Policy-Managers to Executors in the form of policy obligation events.

Niche achieves reliability of management by replicating management elements. For example, if a Policy-Manager fails when evaluating a request against policies, one of its replicas takes its responsibility and continues with the evaluation.

Executors execute the actions, dictated in policy-obligation-events, on managed resources through actuators deployed on managed resources.

Special Policy-Watchers monitor the policy repositories and policy configuration files. On any change in the policy reposi-

tories or policy configuration files (e.g. a policy configuration file has been updated), a Policy-Watcher issues a Policy-Change-Event and sends it to the Policy-Manager-Group through the one-to-all binding. Upon receiving the Policy-Change-Event, all Policy-Managers reload policies. This allows administrators to change policies on the fly.

Policy-Manager-Group is a group of Policy-Managers, which are loaded with the same set of policies. Niche is a distributed component platform. In the distributed system, a single Policy-Manager, governing system behaviors, will be a performance bottleneck, since every request will be forwarded to it. It is allowed in Niche to have more than one Policy-Manager-Group in order to avoid the potential bottleneck with centralized decision making. Furthermore, the size of Policy-Manager-Group, that is, the number of Policy-Managers it consists of, depends on its load, i.e. the intensity of requests (the number of requests per time unit). When a particular Policy-Manager-Group is highly loaded, the number of Policy-Managers is increased in order to reduce burdens of each member. Niche allows changing the group members transparently without affecting components bound to the group.

A Local-Conflicts-Detector checks that the new or modified policy does not conflict with any existing local policy for a given Policy-Manager-Group. There might be several Local-Conflicts-Detectors, one per Policy-Manager-Group. A Global-Conflicts-Detector checks whether the new policy conflicts with other policies in a global system-wise view.

B. Policy-Based Management Control Loop

Self-management behaviors can be achieved through control loops. A control loop keeps watching states of managed resources and acts accordingly. In policy-based management architecture described above, a control loop is composed of Watchers, Aggregators, a Policy-Manager-Group and Executors (Fig. 2). Note that the Policy-Manager-Group plays a role of Manager (see Fig. 1).

Watchers deploy sensors on managed resources to monitor their states, and report changes to Aggregators that communicate policy request events to the Policy-Manager-Group using one-to-any bindings. Upon receiving a policy request event, the randomly chosen Policy-Manager retrieves applicable policies, along with any information required for policy evaluation, and evaluates policies with information available.

Based on rules and actions prescribed in the policy, the Policy-Manager will choose the relevant change plan and delegate to executor in charge. The executor executes the plan on the managed resource through actuators.

C. Policy-Manager-Group Model

Our framework allows programmers to define one or more Policy-Manager-Groups to govern system behaviors. There are two ways of making decisions in policy management groups: centralized and distributed.

In the centralized model of Policy-Manager-Group, there is only one Policy-Manager-Group formed by all Policy-Managers with common policies. The centralized model is easy to implement, and it needs only one Local-Conflict-Detector and one Policy-Watcher. However, a centralized decision making can become a performance bottleneck in policy based management for a distributed system. Furthermore, management should be distributed, based on spatial and functional partitioning, in order to improve scalability, robustness and performance of management. The distribution of management should match and correspond to architecture of the system being managed, taking into account its structure, location of its components, physical network connectivity, management structure of an organization where the system is used.

In the distributed model of Policy-Manager-Group, each policy manager knows only partial policies of the whole system. Policy managers with common policies form a policy-manager group associated with a Policy-Watcher. There are several advantages of the distributed model. First, it is a more natural way to realize policy based management. For the whole system, global policies are applied to govern system behaviors. For different groups of components, local polices are governing their behaviors based on the hardware platforms and operating systems they are working on. Second, this model is more efficient and scalable. Policy-managers reading and evaluating fewer policies will shorten the evaluation time. However, policy managers from different groups need to coordinate their actions in order to finish policy evaluation when the policy request is unknown to a policy manager, which, in this case, needs to ask another policy manager from a different group. Any form of coordination is a lost to performance. Last, the distributed model of policy-based management is more secure. Not all policies should be exposed to every policy manager. Since some policies contain information on the system parameters, they should be protected against malicious users. Furthermore, both Global-Conflict-Detector and Local-Conflict-Detector are needed to detect whether or not a newly added, changed or deleted policy is in conflict with other policies for the whole system or a given policy-manager-group.

IV. Niche Policy-based Management Framework Prototype

We have built a prototype of our policy-based management framework for the Niche distributed component management system by using policy engines and corresponding APIs for two policy languages XACML (eXtensible Access Control Markup Language) [8], [9] and SPL (Simplified Policy Language) [10], [11].

We have had several reasons for choosing these two languages for our framework. Each of the languages is supported with a Java-implemented policy engine; this makes it easier to integrate the policy engines into our Java-based Niche platform. Both languages allow defining policy rules (rules with obligations in XACML, or decision statements in SPL) that dictate the management actions to be enforced on managed resources by executors. SPL is intended for management of distributed system. Although XACML was designed for access control rather than for management, its support for obligations can be easily adopted for management of distributed system.

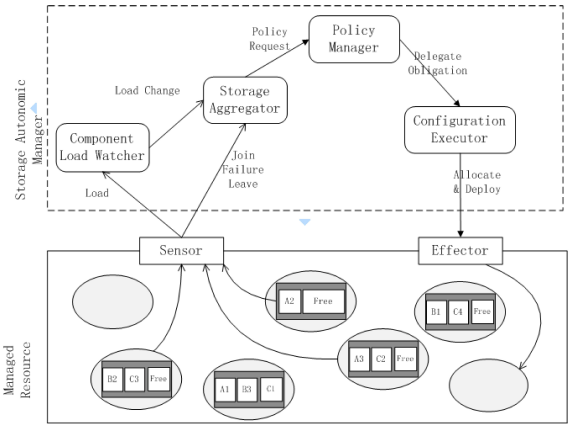

Fig. 3. YASS self-configuration control loop

		Policy Load	First evaluation	Second evaluation
XACML	MAX	379	36	7
	MIN	168	11	1
	AVG	246.8	18.9	3
SPL	MAX	705	7	7
	MIN	368	3	2
	AVG	487.4	5.7	5.7
Java	AVG	—	≈0	≈0

TABLE I
POLICY EVALUATION RESULT (IN MILLISECONDS)

		Policy Re-Load	1st evaluation	2nd evaluation
XACML	MAX	27	4	5
	MIN	23	1	1
	AVG	24.5	3	3
SPL	MAX	62	8	6
	MIN	53	2	2
	AVG	56.3	5.8	5.3

TABLE II
POLICY RELOAD RESULT (IN MILLISECONDS)

In order to test and evaluate our framework, we have applied it to YASS, Yet Another Storage Service [4], [6], which is a self-managing storage service with two control loops, one for self-healing (to maintain a specified file replication degree in order to achieve high file availability in presence of node churn) and one for self-configuration (to adjust amount of storage resources according to load changes). For example, the YASS self-configuration control loop consists of Component-Load-Watcher, Storage-Aggregator, Policy-Manager and Configuration-Executor as depicted in Fig. 3. The Watcher monitors the free storage space in the storage group and reports this information to Storage-Aggregator. The Aggregator computes the total capacity and total free space in the group and informs Policy-Manager when the capacity and/or free space drop below predefined thresholds. The Policy-Manager evaluates the event according to the configuration policy and delegates the management obligations to Executor, which tries to allocate more resources and deploy additional storage components on them in order to increase capacity and/or free space.

In the initial implementation of YASS, all management was coded in Java; whereas in the policy-based implementation, a part of management was expressed in a policy language (XACML or SPL).

We have used YASS as a use case in order to evaluate expressiveness of different policy languages, XACML and SPL, and the performance of policy-based management compared with hard-coded Java implementation of management. It is worth mentioning that a hard-coded manager, unless specially designed, does not allow changing policies on the fly.

In the current version, for quick prototyping, we set up only one Policy-Manager, which can be a performance bottleneck when the application scales. We have evaluated the performance of our prototype (running YASS) by measuring the average policy evaluation times of XACML and SPL policy managers. We have compared performance of both policy managers with the performance of the hard-coded manager explained above. The evaluation results TABLE I show that the hard-coded management implementation performs better (as expected) than the policy-based management implementation. Therefore, it could be recommended to use policy-based management framework to implement less performance-demanding managers with policies or objectives that need to be changed on the fly. The time needed to reload the policy file by both XACML and SPL policy managers is shown in TABLE II. From these results we have observed that the XACML management implementation is slightly faster than the SPL management implementation; however, on the other hand, in our opinion based on our experience, SPL policies was easier to write and implement than XACML policies.

A. Scalability Evaluation using Synthetic Policies

The current version of YASS is a simple storage service and its self-management requires a small number of management policies (policy rules) governing the whole application. It is rather difficult to find a large number of real-life policies. To further compare the performance and scalability of management using XACML and SPL policy engines, we have generated dummy synthetic policies in order to increase the size of the policy set, i.e. the number of policies to be evaluated on a management request. In order to force policy engines to evaluate all synthetic policies (rules), we have applied the Permit-Overrides rule combining algorithm for XACML policies, where a permitting rule was the last in evaluation, and the Execute_All_Applicable strategy for SPL policies.

Fig. 4 shows the XACML preprocessing time versus the number of policies in a one-layered policy. We observe that there is an almost linear correlation between the preprocessing time of XACML and the number of rules. This result demonstrates that the XACML-based implementation is scalable in the preprocessing phase.

Fig. 5 shows the processing time of SPL versus the number of policies. We observe that there is almost exponential correlation between the processing time of SPL and the

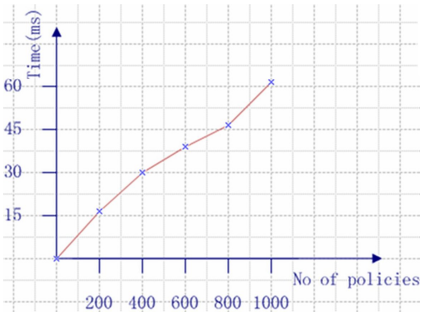

Fig. 4. XACML policy evaluation results

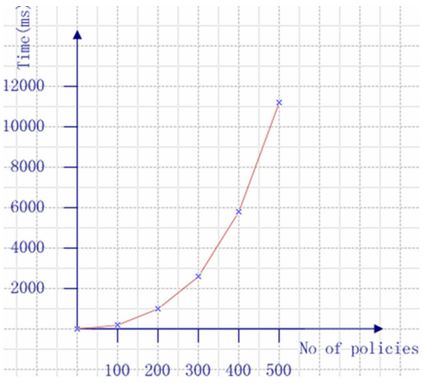

Fig. 5. SPL policy evaluation results

number of policies. This result demonstrates that the SPL-based implementation is not scalable in the processing time.

V. Related Work

Policy Management for Autonomic Computing (PMAC) [12], [13] provides the policy language and mechanisms needed to create and enforce these policies for managed resources. PMAC is based on a centralized decision maker Autonomic Manager and all policies are stored in a centralized policy repository. Ponder2 [14] is a self-contained, stand-alone policy system for autonomous pervasive environments. It eliminates some disadvantages of its predecessor Ponder. First, it supports distributed provision and decision making. Second, it does not depend on a centralized facility, such as LDAP or CIM repositories. Third, it is able to scale to small devices as needed in pervasive systems.

VI. Conclusions and Future Work

This paper proposed a policy based framework which facilitates distributed policy decision making and introduces the concept of Policy-Manager-Group that represents a group of policy-based managers formed to balance load among Policy-Managers.

Policy-based management has several advantages over hard-coded management. First, it is easier to administrate and maintain (e.g. change) management policies than to trace the hard-coded management logic scattered across codebase. Second, the separation of policies and application logic (as well as low-level hard-coded management) makes the implementation easier, since the policy author can focus on modeling policies without considering the specific application implementation, while application developers do not have to think about where and how to implement management logic, but rather have to provide hooks to make their system manageable, i.e. to enable self-management. Third, it is easier to share and reuse the same policy across multiple different applications and to change the policy consistently. Finally, policy-based management allows policy authors and administrators to edit and to change policies on the fly (at runtime).

From our evaluation results, we can observe that the hard-coded management performs better than the policy-based management, which uses a policy engine. Therefore, it could be recommended to use policy-based management in less performance-demanding managers with policies or management objectives that need to be changed on the fly (at runtime).

Our future work includes implementation of Policy-Manager-Group in the prototype. We also need to define a coordination mechanism for Policy-Manager-Groups, and to find an approach to implement the local conflict detector and the global conflict detector. Finally, we need to specify how to divide the realm of each Policy-Manager-Group governs.

References

[1] P. Horn, "Autonomic computing: IBM's perspective on the state of information technology," Oct. 15 2001.

[2] IBM, "An architectural blueprint for autonomic computing, 4th edition," http://www-03.ibm.com/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf, June 2006.

[3] D. Agrawal, J. Giles, K. Lee, and J. Lobo, "Policy ratification," in *Policies for Distributed Systems and Networks, 2005. Sixth IEEE Int. Workshop*, T. Priol and M. Vanneschi, Eds., June 2005, pp. 223– 232.

[4] A. Al-Shishtawy, J. Höglund, K. Popov, N. Parlavantzas, V. Vlassov, and P. Brand, "Enabling self-management of component based distributed applications," in *From Grids to Service and Pervasive Computing*, T. Priol and M. Vanneschi, Eds. Springer, July 2008, pp. 163–174.

[5] Niche homepage. [Online]. Available: http://niche.sics.se/

[6] A. Al-Shishtawy, J. Höglund, K. Popov, N. Parlavantzas, V. Vlassov, and P. Brand, "Distributed control loop patterns for managing distributed applications," in *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2008)*, Venice, Italy, Oct. 2008, pp. 260–265.

[7] E. Bruneton, T. Coupaye, and J.-B. Stefani, "The fractal component model," France Telecom R&D and INRIA, Tech. Rep., Feb. 5 2004.

[8] Oasis extensible access control markup language (xacml) tc. [Online]. Available: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml#expository

[9] Sun's xacml programmers guide. [Online]. Available: http://sunxacml.sourceforge.net/guide.html

[10] Spl language reference. [Online]. Available: http://incubator.apache.org/imperius/docs/spl_reference.html

[11] D. Agrawal, S. Calo, K.-W. Lee, J. Lobo, and T. W. Res., "Issues in designing a policy language for distributed management of it infrastructures," in *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium*, June 2007, pp. 30–39.

[12] IBM, "Use policy management for autonomic computing," https://www6.software.ibm.com/developerworks/education/ac-guide/ac-guide-pdf.pdf, April 2005.

[13] D. Kaminsky, "An introduction to policy for autonomic computing," http://www.ibm.com/developerworks/autonomic/library/ac-policy.html, March 2005.

[14] K. Twidle, N. Dulay, E. Lupu, and M. Sloman, "Ponder2: A policy system for autonomous pervasive environments," in *Autonomic and Autonomous Systems, 2009. ICAS '09. Fifth International Conference*, April 2009, pp. 330–335.