Towards policy-based management of sensor networks

Nelson Matthyss and Wouter Joosens
IBBT-DistriNet, Dept. of Computer Science, KULeuven
Celestijnenlaan 200A, B-3001 Leuven, Belgium
{nelson.matthys, wouter.joosens}@cs.kuleuven.be

ABSTRACT
Contemporary distributed software systems for realistic business applications have become extremely heterogeneous, dynamic, and large scale. They offer services to many different types of users and include various hardware such as back-end servers, regular PCs, and various mobile and embedded devices, as well as diverse network infrastructures, such as sensor networks. However, managing these systems is typically a complicated task due to several factors. This paper proposes a policy-based approach that offers a higher level of abstraction to manage these distributed applications in an easier way. We first give an overview of existing research on policies in traditional middleware and describe some of the shortcomings when applying them for managing sensor networks. Secondly, we propose an architecture for a policy-driven middleware that addresses these limitations and provides a solution for them.

Categories and Subject Descriptors
C.2.4 [Computer-Communication Networks]: Distributed Systems, Distributed Applications

General Terms
Design, Management

Keywords
Policy-based management, middleware, architecture

1. INTRODUCTION
Sensor networks emerge as a key technology in the evolution towards extremely large-scale distributed software systems. These systems are typically composed of geographically dispersed systems using a variety of devices and platforms to deliver services that cross device, platform and system administration boundaries. A broad range of industrial or business applications, for instance, tracking and tracing, industrial monitoring, building automation, or e-healthcare will definitely benefit from them.

Figure 1 illustrates a representative example of such a business application where sensor networks are used for facility management. A facility operator’s production planning process uses the services offered by several heterogeneous sensor networks, deployed inside factories, to deliver status information about resources and location data from pallets and carts. This data is used to schedule the factory’s resources and coordinate their usage. Besides the production planning process, the facility operator also uses the sensor network to control the air conditioning inside its factories. For securing the site, a contracted security company requires alert-data from door/window monitoring and fire detection sensors deployed inside the facility. Instead of managing this complex infrastructure itself, the facility operator outsources its management to a specialized network management company, whose responsibility is to keep the entire infrastructure operational, maintain its installed software services, and integrate the business requirements of the individual users. These requirements may range from relatively simple functional requirements (e.g., the ability to retrieve the temperature inside a warehouse) to complex, possibly conflicting, non-functional requirements from end-users (e.g., securing sensitive sensor data, or desiring the most accurate sensor data). Having a means to easily specify and automatically enforce them in the system is therefore desired.

Policy-based management approaches have often focused on such large-scale systems that constitute an enterprise network or span beyond the borders of it. Policies can be applied to large sets of managed resources in order to configure them uniformly and adapt their behavior to changes in business requirements [8]. On the other hand, scale, heterogeneity, and distribution of these systems is typically handled by the generic facilities provided by middleware. Therefore, combining the power of policies with middleware seems a promising approach for mastering such distributed systems as illustrated in the motivating example.

The sketched application scenario indeed clearly motivates the need for such a policy-driven approach. The network manager may receive a high-level policy file from the security company that expresses that the door/window monitoring system should be secured (i.e., these sensors may only be queried by the security company). Similarly, the factory operator may express that temperature data from its air conditioning sensors should be as accurate as possible (i.e., requests for temperature data should always be directed to the nodes in the network, instead of to a caching server in...
the back-end). Validating and enforcing these policies relies on a process involving multiple steps: context information gathering about capabilities and state of the infrastructure, conflict detection between and reasoning on policies, and finally enforcing the outcome of the reasoning process in the infrastructure. The focus of this paper is to first define these individual ‘policy-enabled’ building blocks and their relation with each other. Furthermore, the actual interpretation and enforcement of these policies will be discussed.

Section 2 gives a brief overview on state-of-the-art research on policies in traditional middleware and its open challenges related to policies in sensor networks. Section 3 zooms in on our architecture for a policy-driven middleware dedicated to manage these kinds of systems. Section 4 describes by means of a case study how the proposed architecture can be used for policy-based management. Finally, Section 5 summarizes our contributions.

2. POLICIES IN TRADITIONAL MIDDLEWARE

Policies provide abstractions that can be used for the configuration and management of resources (i.e. services and system resources), possibly at runtime, without having to change their implementation. They can declaratively specify both the functional and non-functional behavior of these resources in certain circumstances and can be found in many domains like security, QoS, network management, or user preferences. The level of abstraction a policy offers to its users can be very broad in the sense that policies may range from high-level management goals (e.g. the door monitoring system must be secure), which are easier to understand by humans, to low-level actions (e.g. enable AES-encryption on communication link), which can be interpreted directly by the underlying system. The level of abstraction a policy describes therefore depends on the degree of detail contained in its specification. A policy specification is a description containing (i) a policy model, consisting of technological aspects within the policy and relevant concepts from the broad domain of subject, and (ii) a language, to describe the desired behavior of these concepts from the model.

This section further zooms in on three aspects of the policy life cycle: policy specification, reasoning, and enforcement and integration into an application.

2.1 Policy specification

To make policy specification as effective as possible, targeting a policy language to a specific application domain usually makes policies easier to understand and simplifies their specification. This allows one to define the language and model using more higher-level familiar concepts from within the domain itself. Consequently, not only programmers but also ordinary end-users can understand, validate, modify and define domain-specific policies satisfying their (business) requirements. In this context, there has already been research done on policy specification in several domains like, e.g. security [10], QoS [16] or network management [7].

However, having a separate policy specification for each of these domains is impractical because their limited useability and the fact that one has to comprehend a different policy specification for each of these different areas to master the entire system. In order to be practical, there exists the need for one common general-purpose policy language that can be used for configuring services in each of these domains using primitives from the same core language. This common language offers a generic model that can be used for specifying concepts and rules that are common to all types of policies. Initial research in this context includes standardization efforts from the IETF [15] and DMTF [2] for such a general-purpose model, although their focus lies on the domain of network management. A drawback of these models is that, due to their generalization effort, they lack detail for fine-grained specification and therefore are mainly unused.

Recent research [14] however, has tried to bridge the gap between these general-purpose and domain-specific policy specifications. In this approach, a generic specification is defined based on a selected and refined subset of concepts from the existing general-purpose models. The resulting specification acts as an expressive, general-purpose language that can be used in many different domains. Hence, the core concepts from this policy model can be extended and refined with specific concepts from individual domains like e.g. security or QoS, but also elements from the application domain itself.

2.2 Policy reasoning

As illustrated in the motivating example from Section 1, distributed applications no longer offer services to a single user but they require the collaboration between many dif-
fferent actors. Because each of these actors are free to design their own policies for capturing their business requirements, the total number of policies in the system may become quite large. Moreover, these policies must be combined, possibly leading to conflicts (e.g. allowing vs. denying a request to the sensor network) and inconsistencies (e.g. differences between the number of allowed requests). In order to avoid these problems, policy reasoning is required to detect and resolve these anomalies.

Policy reasoning has always been an important topic in the context of policy-based management. An important aspect of this policy reasoning is that the process may require additional information from the system. For example, a classical Event-Condition-Action (ECA) policy rule (e.g. if ‘door opened event’ & ‘evening’ then signal alert) requires additional contextual information to check if the condition (‘evening’) holds. This information might be extracted by additional sensors or simply a clock. Existing research on reasoning focuses on the use of logic [1], event calculus [4], or exploiting relationships between policies [13] to reason on policies and resolve potential conflicts.

2.3 Policy enforcement and integration

A common used approach [15] for enforcement and integration of policies into an application is based on a policy engine that evaluates policies against decision requests. The approach uses fixed enforcement points which are integrated into an application and contact the engine whenever a policy needs to be evaluated during the normal flow of the application. This policy engine has access to all stored policies and needs to be evaluated during the normal flow of the application. The engine then returns the outcome of the policy evaluation to the enforcement point which is then responsible for enforcing this result.

Ideally, adding such policy blocks to an application should not require radical changes to the existing application code and should be cleanly modularized as a separate concern to achieve maximal reusability. Existing techniques like aspect-oriented programming [6] or simply hardcoding these blocks are generally used to integrate them in the application. Other approaches propose policy blocks to be enabled at fixed points in the infrastructure, for instance, inside messaging middleware like event and service busses [4].

2.4 Limitations and challenges of traditional policy-based middleware

A number of challenges and limitations arise when one would simply apply traditional policy-based middleware for the management of a real world business application involving sensor networks:

1. The scale, heterogeneity and distribution of applications involving sensor networks is generally of a much larger order than in any traditional distributed system. The concept of applying policy-based management for these kinds of applications seems promising, however current approaches definitely do not scale well to them as traditional policy-based management is typically a heavyweight process involving intense system monitoring, policy reasoning and enforcement.

2. Applications using sensor networks are more subject to environmental dynamism and are more integrated with the physical world when compared to traditional Internet-based systems. The latter kind of systems execute in a typically more static environment which makes it easier to predefined policy interception and enforcement points at fixed locations. The dynamism in distributed applications involving sensor networks where, besides unreliability and uncertainty of the infrastructure, changes in requirements are common, may require one to (dynamically) adjust the configuration from time to time in order to satisfy the current and newly specified policies. This type of dynamism causes that one cannot predefine these points beforehand at fixed locations anymore.

3. Dealing with this unreliability and unpredictability of the infrastructure is a serious challenge in the context of sensor networks for real world business applications. These uncertainties require that the entire infrastructure should also be monitorable after the policy enforcement phase. For instance, one may want to express that the collected data must be retrieved in the most efficient manner. The reasoning process may then decide that several nodes in a room have to be transformed into dedicated aggregation nodes to which sensor data is forwarded to. However, depending on the state of the network, the corresponding configuration for this policy may have to change in time. In this case, whenever nodes and connections fail, the system may not be collecting its data in the most efficient way anymore. Alternate paths must therefore be sought or additional data collection points must be installed in the affected network regions. Having the means to express, reason, and detect violations on these kinds of non-functional requirements in such a dynamic environment becomes even more important than in traditional systems.

3. POLICY-DRIVEN MIDDLEWARE

3.1 Focus on challenges

Having these challenges in mind, this section proposes a generic middleware architecture that tries to tackle each of them. An approach to handle the traditional heavyweight process of traditional policy-based management, is to modularize the entire process into smaller interoperable building blocks and maximally exploit the heterogeneity and distribution of the environment for hosting them. These individual and cooperating ‘policy-enabled’ blocks can then be independently deployed according to their host’s capabilities.

Secondly, due to the various sources of dynamism in the environment and application (i.e. changes in requirements), policy-based management can neither be of a static nature. Therefore, the architecture must (i) support dynamic points for monitoring, policy interception and enforcement, and (ii) having a means to decouple them from the underlying system is strongly desired. Both criteria can be achieved by adding a middleware solution that implements the concept of a bus. This bus mechanism then enables independent components to publish and subscribe to any kind of information. The decoupling from the underlying system is then supported by introducing adapters that link the components one wishes to manage to this bus.

Finally, having policies as a means to describe and enforce non-functional business requirements that deal with unreli-
These policies will be mapped into more detailed policies, which are illustrated in Listing 2. As automatical translation of these high-level goals into more concrete policies is a very difficult problem, one has to define these lower-level policies too. The reasoning process is then assigned to connect the high-goals with applicable, non-conflicting, concrete ones.

### 3.3 Managed resources

The connection between our middleware approach and the underlying system leverages on the concept of managed resources. A managed resource in our middleware is a software object that is transparently linked to real resource such as software service or a system resource. A managed resource provides two types of management interfaces: a monitoring interface to gather information from the actual resource, and a control interface, to adapt the behavior of the actual resource. The monitoring interface offers support to query the state of the actual resource and to register for any events of interest. The control interface provides the means to manage the resource by e.g. offering an interface to adjust QoS or security settings of the actual resource.

### 3.4 System monitoring

To keep track of the managed resources on a node, we connected a system monitoring service with each of them to retrieve contextual information. This information includes the resource’s current state, supported operations on its interfaces, and any events of interest. The monitoring service offers a means to place any real resource under policy-based control by linking the real resource with a corresponding managed resource object.

System monitoring can be selective and hierarchical in the sense that (i) not all available resources of a node should be monitored if not required, and (ii) a node should be able to monitor a group of other nodes (e.g. a gateway that monitors all network traffic passing through it, or a gateway that keeps track of the state of the installed services inside its network). The advantage of selective monitoring is that it is less resource demanding, which is important in the context of sensor networks. Also, the frequency of monitoring resources can be adapted to the execution context: e.g. managed resources can be polled periodically from a node which lies logically higher in the infrastructure to determine their state and detect changes. Finally, the monitoring service provides preprocessing functionality which can be used to process the collected information (e.g. filtering certain events, combining events to form a higher-level event, or directly annotating context information by timestamping it).

### 3.5 Policy interpretation and reasoning

To realize the interpretation and reasoning of both functional and non-functional policies, we provide a policy engine component. This component is charged with interpreting and transforming statements from these policy files into concepts from the corresponding policy model. After policy interpretation, the engine has to detect any possible conflicts with already applied policies, which are stored inside a policy repository. Examples of possible conflicts are opposite actions like enabling and disabling the same service, or assigning different priorities to the same alert-event. This detection process may use one of the techniques from Section 2.2. In case of no conflicts, the engine reasons on the policy model and selects the appropriate low-level policies.

---

Figure 2: Policy-driven middleware architecture

---

Listing 1: Examples of high-level policies

```plaintext
ensure door_monitoring data is secured
ensure door_monitoring data is prioritized
```

Listing 2: Examples of more concrete policies

```plaintext
if event=door_sensor then event.priority=1
enable AES-encryption on door_sensors
enable Access control on gateway
```
to generate a valid configuration. Otherwise, the engine signals the conflict to the user or resolves it by proposing an alternative outcome based on additional criteria like e.g., the prioritization of certain policies.

The generation of an appropriate configuration for a business requirement may vary in complexity. Simple functional policies, often specified in ECA rules, may be directly applied in the network, whereas complex (non-functional) policies, often specified as management intentions, require intense reasoning. To enable this, the policy engine can implement a goal-based approach [1] to refine policies into more specific actions. In this case, the policy engine will then express its outcome as a set of adaptation strategies. For instance, ensuring the alerting system means: deploying an access control security component on the gateway and enabling encryption on door monitoring sensors.

However, as the underlying infrastructure is subject to dynamism, it might happen in time that the result of an already applied policy does not satisfy its policy anymore. For instance, the reasoning process for a policy specifying that the temperature must be retrieved in the most efficient manner, has decided that only a few fixed number of nodes are responsible to determine the average temperature in a warehouse. Whenever the number of available nodes drops below an acceptable level, this situation should be detected and the policy engine must automatically notified. The engine in turn then has to reason again on the current network state in order to determine which nodes will be (re)charged to retrieve the temperature.

3.6 Policy enforcement

Finally, the outcome of the policy engine’s reasoning process must be enforced in the infrastructure. As already mentioned, this outcome contains several adaptation strategies or actions that have to be interpreted by the policy enforcement service. This service translates the adaptation strategies into specific operations on the affected managed resource objects. The policy enforcement service then has to invoke the control interface on all affected managed resources in the infrastructure. Finally, the managed resources then translate the operation into device-specific operations on the actual underlying resources.

4. CASE STUDY

We will show by means of a case study how the proposed middleware architecture can be used for the policy-based management of the scenario sketched in Section 1. In particular, this section describes the configuration of and interactions within the architecture for two types of policies: (i) a functional policy, describing that all alert notifications must have absolute priority, and (ii) a non-functional policy describing that the alerting system should be secure (i.e., its sensors can only be used by the security company). Figure 3 illustrates our policy-based management approach for both scenarios. For simplicity, we assume that the distributed system only consists of four categories of devices (i.e., four tiers): simple sensor nodes serving for fire detection, more advanced nodes equipped with door and window monitoring sensors and used for aggregation, one or more gateway devices inside a factory building, and back-end systems.

For both scenarios, several managed resource objects have been instantiated and connected to their corresponding real resource. As the fire detection nodes are resource-constrained and the nodes used for aggregation are relatively powerful, the default configuration of the middleware is therefore as follows: fire detector sensor nodes are managed by nodes used for aggregation. This means that the monitoring, policy engine and enforcement services are deployed on the latter type of nodes. These services directly monitor and control the fire detection sensor nodes’ managed resources.

4.1 Supporting functional policies

The functional policy from the first scenario is specified in terms of ECA-semantics using concepts of the application domain (i.e., ‘door opened’ and ‘fire detected’ events). Enforcing this specification, using our middleware, is relatively simple in this case. The policy engine at the network manager’s back-end first checks if the given policy is not in conflict with any existing policies residing inside the policy repository. In case of conflicts, the conflict is signalled or resolved by the policy engine. In case of no conflicts, the policy itself is distributed to the policy engine on the aggregation node, where it can directly be interpreted whenever it matches against the occurrence of its associated events.

Each time a sensor detects the opening of a door or fire, its corresponding managed resource object captures this event and delivers it through the communication bus to the associated monitoring service on the aggregation node. The monitoring service then dispatches it to the policy engine. Since the policy engine has a corresponding ECA-rule installed, it can directly interpret the action associated with the rule. This means that the event’s priority is adjusted to the maximum priority and the event is delivered back to the managed resource object of the communication stack for transmission.

4.2 Supporting non-functional policies

Supporting non-functional policies is challenging as they generally describe high-level intentions that need to be refined in more concrete actions. Enforcing these policies requires that they are also formalized using primitives from the host policy language. Reasoning on these policies requires information retrieval from diverse monitoring services, deployed at various points in the infrastructure. In this scenario, where the security company has specified that the alerting system must be secure, several nodes will be affected by the outcome of the reasoning process. The policy engine therefore, after checking for consistency, has to query the different monitoring services of diverse nodes to gather information about the affected nodes’ configuration (i.e., which nodes are involved, and what resources they manage).

In this case, the policy engine on the back-end is informed that node 1, 2, the gateway, and the back-end system itself are involved and that they all possess a security service that allows encryption or access control. The outcome of the reasoning process will then be the formulation of adaptation strategies, affecting the configuration of the security service on the involved nodes. This means that the request for encryption will be evaluated by the corresponding policy enforcement services and that this instruction will be translated into a device-specific operation on the actual resources by the corresponding managed resource objects (i.e., enabling AES-encryption on the nodes 1 and 2, configuring the security service to decrypt any relevant incoming message on the back-end, and installing or enabling the access control component at the gateway).
5. CONCLUSION

This paper proposes a policy-based approach to manage distributed sensor applications. We have motivated the use of such policy-based management approach with a realistic application scenario and described how our approach adds an extra level of abstraction on top of a current middleware. The motivation of this middleware layer is to easily describe and enforce both functional and non-functional business requirements of different end-users.

We have started from existing research on policies in traditional middleware to find limitations and challenges when applying them directly in the context of sensor applications. Furthermore, we propose the design of a policy-driven sensor middleware, which leverages on the one hand on current state-of-the-art techniques for system monitoring, specification, reasoning and policy enforcement, while on the other hand provides an answer to these identified challenges.

Finally, we describe by means of a case study from the motivating example how our architecture can be used to manage such distributed sensor applications. We briefly show that both functional policies and non-functional policies are supported, and describe the resulting configuration and interactions within the architecture.

Acknowledgment

We are very grateful to Sam Michiels and Christophe Huygens for interesting discussions and their comments on the paper. Research for this paper was sponsored by IBBT, the Interdisciplinary institute for BroadBand Technology, and conducted in the context of the IBBT-DEUS project [9].

6. REFERENCES