Thinking autonomic for sensing devices

Rémi Sharrock*, Amin Cherbal†, Levent Gürgen‡, Thierry Monteil* and Shinichi Honiden§

* University of Toulouse, Toulouse, France
{remi.sharrock,monteil}@laas.fr
† LIG Laboratory, Grenoble, France
amin.cherbal@imag.fr
‡ CEA-LETI Minatec, Grenoble, France
levent.gurgen@cea.fr
§ National Institute of Informatics, Tokyo, Japan
honiden@nii.ac.jp

Abstract—This paper presents a middleware that aims to respond to the increasing management needs of networked sensing devices, such as remote software deployment, dynamic configuration and real-time performance monitoring and tuning. These are essential functions for building adaptable systems providing high quality of service. Furthermore, autonomic management bears particular importance for large scale systems having timely response requirements. Our middleware allows creating high-level policies that describe the actions to be autonomously performed when interesting events occur in the system. The policies are formulated as Event-Condition-Action (ECA) rules. They are created by using UML-like diagrams and can be deployed at any level of the distributed architecture of the system.

Keywords—autonomic; management; service-oriented; middleware; sensors; sensing devices; dynamicity

I. INTRODUCTION

With technological advances, tiny sensing devices (temperature sensors, audio-visual devices, chemical sensors, GPS devices, RFID readers, etc.) are increasingly present in applications from various domains such as industrial, medical and home automation. These applications have more stringent dynamic adaptability, re-configurability and quality of service requirements. Efficient remote management mechanisms are crucial to fulfill these requirements, enabling for instance remote deployment of software modules, (re-)configuration of device parameters or monitoring performance of devices. However, the large number and heterogeneity of sensing devices make it difficult to manage the system with human administrators using conventional management tools. Furthermore, the real-time nature of sensing systems imposes time critical management actions in these highly dynamic systems. Self-manageability is therefore an essential property that is still challenging to fulfill for networked sensing systems.

The autonomic computing vision [1] has been investigating new ideas in response to the increasing complexity of the management of large scale distributed systems. Indeed, the rapid technological evolutions (e.g., P2P, agents, grids, clouds) have increased the complexity, the cost and the number of errors when human administrators manage such IT infrastructures. Our work particularly focuses on sensing devices and attempts to apply this autonomic management vision to networked sensing systems.

In fact, there has been significant research in both autonomic computing and wireless sensor networks (WSN) domains, nevertheless, few of them tend to combine both research themes [2]. A recent and extensive survey on middleware challenges for WSN [3] showed that autonomic approaches could provide more robustness, reliability, and self-management. Impala [4] is the first attempt for autonomic management of sensing devices. However it is mostly used for network-level management (e.g., to switch between adequate protocols) and considers only the same type of sensors, thus it doesn’t support heterogeneity in terms of hardware platforms.

The next section introduces XSStreaMWare project and TUNe concepts to finally evolve towards a new approach for autonomic management of sensing devices.

II. BACKGROUND

A. Motivation

Our goal is to provide a generic middleware with autonomic management capabilities, independent of the functional area (software, configuration or performance management), the architecture level (application, system or network level) and the type of sensor used in the system. For this purpose, we adopt concepts of an existing autonomous management system TUNe [5], and apply its principles to our sensing device management middleware XSStreaMWare [6]. Both TUNe and XSStreaMWare provide abstract layers to access low-level features in a generic manner. The former uses the encapsulation mechanism using wrappers around software. The latter uses adapters that do necessary method, event and data format translations between sensor specific software and the generic middleware. This allows us to focus on more high level issues, such as the description of the management policies. We define high-level Event-Condition-Action (ECA) rules to realize these policies. At the occurrence of an event, if the event conforms to a given condition, then the corresponding management action(s) can be performed, as it is specified in the rule. We propose a Sensor Management Modeling
Language (SMML) that allows creating UML-like diagrams that represent the rules. We have developed ECA engines that autonomously evaluate rules at the different levels of the distributed architecture of XSStreaMWare.

B. XSStreaMWare

XSStreaMWare [6] is a service-oriented middleware for management of heterogeneous sensing devices. It allows a simple set of management operations to be performed on a generic data model, operations such as setting/getting device parameter values, installing/uninstalling applications or performing diagnostics. Thanks to its generic data model and loosely-coupled management services, XSStreaMWare can manage sensors of different types and from different providers. It provides a "plug&manage" facility thanks to its service-oriented approach. Arrival and departure of services at runtime, as well as their modification are intercepted, thus the adequate measures are taken immediately.

XSStreaMWare is based on a hierarchical distributed architecture composed of 4 main layers (see Figure 1). A control site is the entry point to the system and constitutes the highest level of the hierarchical architecture. It is responsible for an entire environment usually composed of several regions. Each region is managed by a gateway that hosts several adapters that interface sensor specific proxy software. Finally, sensing devices are physically distributed in a given region and they form the lowest level in the hierarchy.

C. TUNe

Toulouse University Network (TUNe) is a prototype for autonomic management of proprietary distributed software. It is based on high-level descriptions for software and hardware architectures, as well as management policies. It has been successfully tested over the french national research grid Grid5000 [7] to help electronic engineers simulating 3D electromagnetic fields [8]. By automatically deploying, configuring and starting the simulator on hundreds of nodes, it has dramatically simplified the use of the grid by hiding the underlying complexity. Furthermore, it has increased the robustness of the application execution by automatically managing failures during runtime.

TUNe intends to show the feasibility of building an autonomic policy engine that can help users managing complex distributed software. The main idea is to create graphical diagrams, initially based on the Unified Modeling Language (UML), to describe the user’s management intentions. The internal engine processes the intentions and tries at its best to achieve it, according to the environmental changes (machine and network failures, overloads). Designing generic high-level languages for management is challenging in a sense that proprietary softwares use their own interfaces, functions and starting or configuring mechanisms. Therefore, to hide the complexity for the final user and to provide a generic framework, TUNe encapsulates the software into a component model using wrappers. The wrapper describes management methods (starting, creating configuration files, re-configuring) that are used in the graphical diagrams.

In our work, we chose to adapt the concept of using a graphical language to define ECA rules. In the next section, we introduce a new language to design a workflow of actions and conditions that describes a management policy for sensor devices. This language has a new syntax and includes specific actions for sensors management, we only reuse TUNe’s graphical framework. We also use parts of the internal TUNe’s engine to parse and execute these diagrams.

III. Middleware for Autonomic Management of Heterogeneous Sensing Devices

A. Global architecture

A promising approach is to introduce a new distributed autonomous management service within the XSStreaMWare middleware. As shown by figure 1, this service is composed by multiple ECA Engines dispatched at the three levels of the architecture, on the different system elements: Control-site, Gateway and Adapter/Proxy levels, making them context-aware and autonomous at a local level. Indeed, this allows to differentiate autonomous sites that are controlled by their own ECA Engine, and allows to manage one site even if the control site or any other part of the management system goes down. At the upper level in the architecture, the Control-site offers a service to the end-user to register ECA rules. External applications may also subscribe to the service to receive events. The user can deploy the rules using a graphical client interface that connects to this service. The rules are created using the Topcased graphical modeling tool [9], and are either registered at the Control-site, or propagated to the concerned Gateways or Adapters. For now on, sensors do not have their own ECA engine, but considering the technological advances, we plan to develop specific integrated ones in the near future. Once registered, the rules are autonomously evaluated when corresponding events occur in the system. Using this distributed hierarchical architecture makes the system more scalable (less network traffic), and enables the middleware to react quickly: the closer the ECA Engines are to the sensors, the faster the reactions are executed. The internal structure of ECA Engines is described in the next sub-section.
B. ECA Engines architecture

Figure 2 introduces the internal ECA Engines mechanisms and the different steps of the ECA cycle. Each ECA Engine is divided in two parts: the Event Engine and the Rule Engine. The Event Engine is responsible of the monitoring task and catches all the system events (SE). The Rule Engine is responsible of analyzing the events and executing actions on the different system elements. The overall process of the ECA cycle begins with the registration of the event types, stored in the Event Repository, and the rules, stored in the rule repository. We first describe the overall process of one ECA Engine, then we define the event types, and we finish with the introduction of the new language for describing the rules.

As shown by figure 2, the overall process of the ECA cycle begins with the registration of event types and rules (step 0) that are stored in repositories. The Event Listener is in charge of the detection process (steps 1–4) and instantiates Management Events depending on all the registered event types. Further on, they are dispatched (step 5) to the Rule Manager and to the subscribed ECA engines at a higher level in the architecture (see the figure 1). The corresponding registered ECA rules are evaluated (step 6–8) and the associated actions are executed on the system.

We identified several event types for both system or sensors elements. The first predefined type is HELLO, and occurs when a new sensor enters the system. The BYE type is used when a sensor wants to leave the system in a proper manner. The UNAVAILABLE type is used whenever a sensor is still unreachable after several tries or after a time deadline. The PERIODIC type refers to an event periodically generated (by a timer), and the MODIFIED type is used to specify a modification of a sensor parameter value. These types are predefined and can be used directly, but the user is able to define its own specific event types, using a generic event description. This description specifies parts of the characteristics of the event (system properties, value and date ranges or other specific properties). Whenever the Event Listener catches a System Event (SE), it verifies if it matches a registered type in its Event Repository and instanciates a Management Event (ME). The table 1 describes all attributes of the Management Event. The TARGET attribute could be extended to a list of targetted elements. For the DATAMODEL attribute, we reuse the datamodel format introduced by XSSstreamWare to store all parameters values issued from the sensor, so that they can be accessed in the same generic way.

After describing the event types, we introduce a new graphical language called SMML to define the ECA rules. SMML stands for Sensor Management Modeling Language, it is based on the graphical framework used by TUNE, but uses different syntaxes and actions, and also introduces the conditions definition. This graphical language is based on the well known UML, and makes it easy to learn for the final user. We chose to introduce specialized activity diagrams, that are typically used for modeling the logic of use cases or scenario. The overall diagram is a workflow of actions and conditions. The conditions are described using a combination of the standard operators and some values or variables. The operators are those used for comparison (≤, ≥, =) or for logics (AND/OR). The values and variables are either numbers or percentages for threshold definitions, strings for software version definition, or one of the following keywords :YES, NO, ELSE. There are three types of actions possible. First, it is possible to create variables using the variable creation action. These variables may contain system element references or parameter values. Second, the generic operations introduced by XSSstreamWare can be invoked using operation actions. There are three operations available: GET is used to retrieved a sensor parameter, SET is used to change a sensor parameter, and ACT is used to invoke a specific command on the sensor. The third and last type of action is used to access the datamodel attribute of the event. This is usefull when a sensor sends some of its parameter values. Indeed, the sent parameters are stored using the datamodel format, and may be accessed using the same generic way introduced by XSSstreamWare.

C. Real management policies examples

In this section we illustrate the features of our middleware with two concrete examples.

- **Autonomic software management.** Sensing devices join and leave the system dynamically. We want to be sure that all devices are using the latest version of firmware/software, in a complete transparent way. When a new sensor is detected, the management system has to check if the version of the
Autonomic configuration and performance management is easy to read for the final user. It shows how straightforward a rule can be created, and how it is undeployed and the new one is deployed. This example database. If the application needs to be updated, the old version number but we plan to check it with a synchronized application module is up to date. Here the latest version is a fixed communication with the sensor to check if the application containing this reference. Then the GET action initiates a target attribute. The first action is to create a SENSOR variable HELLO event. This event contains the sensor’s reference in the detected, the event manager notifies the rule manager with a hence it is applied to the whole system. When a new sensor is new sensor is reconfigured. This event also contains a generic datamodel instantiation (with the values sent by the sensor), that is accessed via the e.datamodel action. If the battery level is between 10% and 50%, then the sampling rate is decreased to 10 seconds with a SET action. If it is less than 10%, the sensor dumps all its logs and shuts down.

IV. Conclusion and Future Work

The ever increasing number of sensing devices makes it impossible for human to administrate them, and imposes more autonomic management solutions. We developed a specific hierarchical autonomic management service within the XSSStreamWare middleware, that is distributed among multiple managed sites. This service provides a generic management interface for sensing devices, independent of the functional area, the architecture level, and the type of sensor used in the system. We reused the graphical management concepts of TUNE, and adapted them to describe autonomic management policies based on ECA rules. We introduced a hierarchical ECA Engines architecture that defines autonomous regions and improves reaction times. We plan to develop ECA Engines within the sensors themselves, and to improve the SMML language to support not only system elements, but also external elements like software databases for autonomous updates. We also plan to define more event types, and to use multiple targets in order to take into account more complex reconfiguration scenarios. Finally we plan to define higher goals that will use multiple ECA rules.

- Autonomic configuration and performance management.

The energy of wireless sensors is an important resource that should be used efficiently. For instance, when a sensor battery level approaches a certain threshold, adequate measures should be taken such as reducing the data sampling rate to save energy (the kind of reconfiguration we use in our examples).

The figure 4 shows a SMML example for power management that is deployed at proxy level (see figure 1). For this example a PERIODIC event is used, which is created when the sensor periodically sends its battery level. As for the first example, the target attribute of the event is a reference to the sensor that has to be reconfigured. This event also contains a generic datamodel instantiation (with the values sent by the sensor), that is accessed via the e.datamodel action. If the battery level is between 10% and 50%, then the sampling rate is decreased to 10 seconds with a SET action. If it is less than 10%, the sensor dumps all its logs and shuts down.

References