Towards a service-oriented component model for autonomic management

Yoann Maurel, Philippe Lalanda
Laboratoire Informatique de Grenoble
F-38041, Grenoble cedex 9, France
(yoann.maurel, philippe.lalanda)@imag.fr

Ada Diaconescu
departement INFRES, Telecom ParisTech
75013 Paris, France
ada.diaconescu@telecom-paristech.fr

***Abstract*—Modern applications are increasingly dynamic and heterogeneous and their lifecycle is more and more governed by autonomic managers that are also getting more and more complex. The purpose of this paper is to present a service-oriented framework that facilitates the development and management of dynamically extensible autonomic managers. More precisely, we propose an architecture based on the opportunistic collaboration of very specialized and coherent modules called administration tasks. The current framework prototype has been implemented as a specialized Service-Oriented Component Model. It allows the dynamic integration of autonomic tasks and their management based on contextual evolutions.**

***Keywords*-autonomic management;autonomic managers; service oriented computing;framework;administration tasks**

I. Introduction

A decade ago, IBM introduced the idea of Autonomic Computing in order to cope with the ever growing complexity of modern systems administration. The purpose was to hide administration complexity by incorporating it directly in the systems. The approach is very ambitious in the sense that complexity does not disappear; it has simply been moved from administration to design, from administrators to developers. Unsurprisingly, building autonomic systems turned out to be a challenge for several reasons. First, it implies to integrate a variety of techniques and tools in order to collect and filter information, reason about it, carry on time-constrained actions, evaluate their utility, learn about failures, etc. The risk, considering the amount of skills required, is that systems result from a disorganized tangle of poorly-mastered disparate methods and algorithms. Good practices, methodologies, patterns and architectures are required to assist developers in this overwhelming task. In addition, autonomic managers are also impacted by the computing environment dynamism. In practice, it is not possible to precisely define in advance all the management situations that may occur and some adaptation capabilities are needed at the autonomic manager level. Finally, autonomic systems must often deal with conflicting goals. For example self-protection often implies greater resources consumption, while self-optimization tends to reduce the use of resources. Decision-making process must be flexible as the priority between these context-dependant goals may change.

Our purpose is to assist the development of mid-size to large-size managers with complex behaviours and goals: typically complex application or large-system manager. Such managers proved to be hard to conceive for all the aforementioned reasons: integration of multiple technologies and tools, disparate and conflicting objectives, hard to predict evolving management situations. In this paper, we introduce a service-oriented component model to facilitate the design and implementation of these autonomic managers. To meet our objectives, we have defined the concept of administration task. An administration task is an independent, specialized and coherent element that achieves one or more management function. At runtime, these discoverable tasks opportunistically combine to form control loops. At any moment, the cooperation can be refined or changed by adding, updating or removing administration tasks using a service oriented approach. It is thus possible to change the overall behaviour or delegate new responsibilities to the manager. Selection mechanisms help to manage conflict between tasks and goals so as to ensure manager consistency. Our approach is implemented on top of OSGi for dynamicity and performance purposes.

This paper is organized as follows. Second section presents an overview of existing approach for building autonomic managers. Third section introduces our framework principle. Our architecture is composed of a management layer, a control layer and an administration layer described respectively from section 4 to section 7. Implementation of this approach is described in section 8 with an example.

II. Existing Work

A few projects have proposed generic frameworks for the development of autonomic managers. While some projects, including Rainbow[1], focus on a specific adaptation method such as the architecture adaptation, a large majority of these platforms concentrate primarily their efforts on the communication between managers[2] [3]. In practice, most projects offer little guidance on the design of the manager internal architecture. The actual architecture of most existing systems closely follows the MAPE-K[4] model, a logical architecture proposed by IBM. This architecture, based on management blocks, cannot always be used directly. Sometimes, an autonomic manager does not have to go through the whole MAPE-K loop because some of the blocks are too simple and skipped. Sometimes, these blocks are too complicated and end up as monolithic elements. This is the case, for instance, when many management concerns are mixed in a same management block. In addition, the MAPE-K architecture provides little guidance for the integration of several different, eventually conflicting, concerns within a single manager. Some projects, like Jade[5] for instance, suggest the implementation of these concerns in separate control loops, but do not propose clear solutions for the collaboration between these loops.

Dynamically modifying the management process is often ignored. It is nonetheless worth noting that some of them, including Automate[2] or Autonomia[3], allow the dynamic deployment of new autonomic elements. It is thus possible to evolve the overall system behaviour. However, the granularity of deployed elements is usually rather big and the changes are kind of abrupt and difficult to achieve in practice. Additionally, Automate is based on a rule system that can evolve over time. Nevertheless, if it is theoretically feasible to change these rules dynamically, this operation is complex considering the low granularity of the rules.

Both SOC and AC are recent and increasingly popular approaches that can be used together[6]. Usually the autonomic approach is used to manage service-oriented applications, for instance for dynamic service orchestration [7] [8]. More rarely, although suggested, SOC is used to bring dynamism to autonomic managers - but not at the same level of granularity as proposed in this paper. There are indeed a few frameworks for building managers (i.e the decision making process) using SOC while it is more commonly used for building autonomic element as a whole. Some approaches [9] propose to bring flexibility by using Web-services for building MAPE blocks. Apart from the considerable performance cost, the granularity is still important and they offer little help to manage potential conflicts.

The solution advocated here has similarities with blackboard system[10] or with the TAEMS project[11]. Regarding the level of abstraction, our administration tasks are relatively similar to blackboard's knowledge sources. A notable difference is that, in our solution, control and data are much more distributed. In particular, tasks communicate directly with their counterpart and the selection mechanisms are only involved when necessary. We also propose a component model that clearly separates the generic aspects from business aspects so that development is facilitated.

III. Principles

Central to our proposition is breaking down autonomic managers into a set of simple and specific behaviour that can be easily implemented and combined opportunistically depending on the context and goals. Our goal is then to define the appropriate behaviour *encapsulation unit* with *adequate granularity*, and to support their *opportunistic integration*.

With that purpose, our framework is structured around the notion of administration task. An **administration task** is a discoverable, specialized and independent **encapsulation unit** responsible for providing a specific and coherent management function. It offers an **homogeneous** integration model for management activities, by featuring the same architecture and by exposing the same API, irrespectively of the achieved control loop . The model homogeneity simplifies development by limiting the number of required technologies. Encapsulating management activities into functionally-independent and coherent units enables their reuse and hides the implementation heterogeneity. Administration tasks are typically *coarser grain than a Java-class or a basic rule*, which, if taken separately, makes little sense. They are intended to implement a complete and coherent function like information gathering, failure or faulty behaviours, plan proposal, learning, etc. They are generally *finer grain than any MAPE-K logical blocks*, as they focus only on a particular goal. Tasks can be implemented in various ways such as fuzzy-logic routines, set of rules, using a planner or simple procedures.

In our approach, the autonomic manager is composed of a number of tasks and control loops result from their dynamic and **opportunistic** collaboration (figure 1).

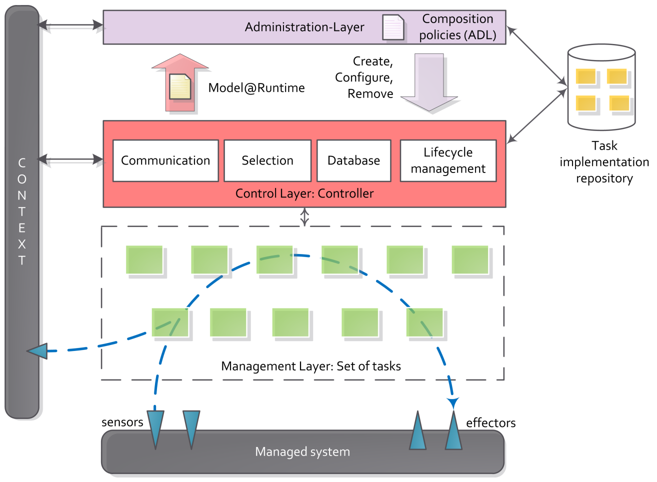

Administration-Layer
Composition policies (ADL)
Model@Runtime
Create, Configure, Remove
Communication
Selection
Database
Lifecycle management
Control Layer: Controller
Task implementation repository
CONTEXT
Management Layer: Set of tasks
sensors
effectors
Managed system

Figure 1. The control loops ensue from the tasks' collaboration.

The tasks are dynamically discovered and assembled for the detection and resolution of complex problems (possibly unpredicted). Composition is opportunistic in the sense that it depends on the runtime conditions. Because the system can adapt during execution it is then no longer necessary to anticipate in a timely fashion all the management situations. It is simply necessary to specify the triggering conditions of the management tasks. Several loops may coexist at any time, each responsible for achieving a particular management goal. Tasks can be deployed, discovered, installed, updated or removed during runtime allowing changes in the manager configuration and strategies. This is made possible by the use a **service-oriented approach**: each task is implemented as a service-oriented component. The system is event-based: tasks asynchronously exchange typed data and are endowed with the capacity to analyse external or internal event to decide when to be triggered.

The framework is organized into three layers dealing with different concerns. The **management layer** includes the administration tasks and is responsible for the management of the system. The **control layer** is in charge of the tasks organisation, conflict management and tasks lifecycle. Interchangeable control mechanisms are provided to cope with conflicts between tasks. Finally the **administration layer** offers all the necessary mechanisms to administer manually or automatically the manager configuration. These three layers are described in the coming sections.

IV. Management Layer : Administration Tasks

A. *Administration Task Architecture*

By providing a homogeneous task model we intend to clearly separate the concepts that generally belong to autonomic managers from the application-specific aspects that are developed case by case. To do so, we have broken task into a number of modules, *each dealing with a specific concern* (figure 2): input and output ports deal with communication, the triggering mechanism specify when a task can be executed, the coordinator prevents conflicting tasks from being concurrently activated, the processor implements task functionality, a statistic module computes usage statistics dedicated to task evaluation, and ,finally, an administration module is in charge of configuration and lifecyle management.

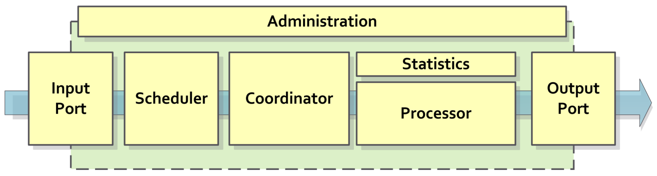


Figure 2. Task's sub-modules

Some mechanisms are optional. For example, by default, the scheduler and the coordinator let all information pass through unfiltered. Statistics calculation can be disabled. A minimal configuration should enable the creation of a manager that requires minimal interaction with the control layer. We aim to provide various implementations of these mechanisms to render development more flexible. Developers can adapt or rewrite them to address specific requirements. This will consequently lead to the creation of a library and favour the reuse of redundant functionalities.

1) Communication: Tasks receive and send data via their **input and output ports**. These ports are not mandatory: typically straight monitoring tasks or action-based tasks do not use these ports. Current implementation favours direct communication using a service-oriented approach on top of OSGi for performance reasons. Input Ports discover and subscribe to output ports via a registry. Output Ports then send data directly until the registry notify task disappearance. Assigning communication to dedicated modules facilitates its evolution - *e.g.* to address distribution-related concerns, without having to systematically rewrite all the code related to communication. Developers are free to modify the default behaviour by providing their implementation: using a centralized event dispatcher, using Web Service to deal with heterogeneous technologies, or using a specific MOM technology such as Joram.

2) Activation: Task activation depends on the **scheduler**. Collected data, if any, are analyzed by this module, which filters them and plans task processing. This module consists of two parts: a buffer and several triggering conditions. The **buffer** stores a copy of the data for as long as they haven't been processed. Frequently, it purges obsolete data considering their type and receipt date. The **triggering conditions** are checked whether regularly, by their own initiative, when new data arrives or as context changes. They evaluate data relevance and decide if and when they should be processed. They can equally observe information and subscribe to context-related events (*e.g.* current date or administrative objectives), as well as calculate information (*e.g.* number of calls, time since last activation, fixed time interval, data quantity). The framework provides several generic and reusable triggering conditions that can be rewritten or extended as needed.

After the evaluation process, the scheduler produces a request in the form of a combination of logical operator (*e.g.* OR, AND) that defines all the possible sets of data the task plan to process. This expression is used to express data dependencies as some of them are not dissociable. This allows the conflict management algorithm to authorize a subset of the request only.

3) Concurency: The planned requests execution depends on the **coordinator** agreement. Many tasks are potentially eligible to handle data at any time. The coordinator guarantees that task is authorized to execute by consulting the control-layer selection mechanism. Selection is solely performed when necessary to ensure good reactivity and avoid excessive communication costs. For this purpose, the expert configures coordinators with the list of conflicting data types (or another discriminator depending on the coordinator). When receiving a request, the coordinator removes references to non-conflicting data from it to obtain a negotiated request. It then sends the latter to the selection mechanism and the task's description. The description contains meta-information set and used by the selection mechanism to keep information from previous selection. Once selection completed, each task receives a list of authorized data and a new state. Concerned data, if any, are removed from scheduler buffer and are sent to the processor.

Having selection expressed separately allows it to evolve seamlessly. The reasoning of the mechanisms we have used is mainly based on data types, task priorities and task states. We should note here that several selection strategies are possible - *e.g.* maximise the number of active tasks or maximise the number of reports processed by high-priority tasks. In the selection process, the coordinator can play a passive role (as described here) or a more active role by effectively participating in the selection process. When an active coordinator is involved the selection process becomes a negotiation.

4) Processing: At this point, data have been collected, filtered and selected and authorized. At the task's core, it is the **processor** that actually processes the data. Most tasks adopt one or more of the four MAPE roles: collecting, looking for problem, finding solutions and applying them. Additional tasks can play a role in facilitating and coordinating management actions. For instance, mediation tasks transform produced data without modifying their semantics and synthesis tasks can be employed when executing tasks are competing and a compromise must be found. The former offers the possibility to explore concurrently different solutions and to choose the best or fastest solution. A task may implement a complete or a large portion of the management loop. A compromise must be found between high flexibility provided by functional decoupling; and, functional-assembly difficulties and performance costs induced by too high fragmentation.

The processor is based on three data flows. The first data flow consists of typed and standardized data provided by others tasks via communications ports as explained here before. A second data-flow exists between the managed system's probes or effectors and the processor. Monitoring probes and executors rely on the processor for collecting data and modifying the managed system. The exchanges between these touch-points and the processor are not standardised. They depend on the processor algorithm and implementation and various communication means can be used (*e.g.* JMX, sockets, or Web Services). The last data-flow consists of the set of data the processors read and write to/from a shared database. This database corresponds to the Knowledge block of the MAPE-K architecture. It enables tasks to exchange persistent information, such as the activation of an alarm. Hence, this reduces the amount of necessary task communication. Data stored in the shared database are typed.

The algorithm implemented by a task's processor is relatively independent of the other tasks. It certainly depends on the available data but has no functional dependence on other tasks. In particular, it has no knowledge of the number, the implementation and even the objective of tasks that provide it with data. The manager assembler and the other tasks are oblivious of the algorithm's implementation or technology used. Still, as with any implementation, functional dependencies may exist towards libraries, towards services deployed on the platform or towards services provided by the platform. The processor is not necessarily implemented in a single module. On the contrary, it is desirable for the processing algorithm to be developed based on existing, well-tested blocks or applications. For this purpose, the task can whether include, encapsulate and hide such existing blocks or simply represent a reference towards existing services. In all cases, complexity is hidden from the expert that selects the tasks and assembles the manager.

B. Task Lifecycle

The task's **administration module** provides unified interfaces to manage task lifecycle, obtain its description, manage its configuration via exposed properties, and provides access to usage statistics. This information is used to build a *model@runtime*[12] detailed further.

The description of a task includes its state and some meta-information. The lifecycle is made of three states: **configured** when a task has a configuration but is not started or has been stopped, **invalid** when data dependencies or functional dependencies are not satisfied, or **valid**. The valid states is refined into three more states: when no request is being processed the task is **waiting**, as soon as the task is processing it becomes **active** and finally if the processing takes too long the task is considered to be **blocked**. The latter is the more interesting as it gives the administration layer a chance to replace malfunctioning tasks. This state is determined using a maximal execution delay given by the task developer or administrator. As explained before the task description is completed by meta-information set by the selection mechanism to keep selection-related information such as a task priority.

To allow task evaluation, a **statistical module** is launched at each processor call and calculates information on the task activity. Hence, the control architecture or the administrator can decide to replace certain tasks or to modify the selection mechanism. The selection mechanism can be updated by changing task priorities, so as to prioritise the execution of certain tasks. Developers can seamlessly rewrite the basic statistical algorithm the framework provides by default. The statistical algorithm collects various information. The current algorithm calculates the number of task executions, the quantity of received/processed data per types, the average task execution time and the number of times a task was considered blocked.

C. Task types and data types

To ensure that tasks have minimal understanding about the nature of the exchanged data, **data are typed**. A data type includes a unique name and a description. It defines a set of attributes, where each attribute has a unique name and a default value. Data are encapsulated into messages, which are made of two parts: a header and the actual data. Headers are used to transmit meta-information that can be used to refine the binding process as well as providing tracking information used to determine control loops.

A given functionality can be implemented in several ways. First, from a technical perspective, tasks can be implemented based on a "classical" programming language, on rules or on an existing application (e.g. a planner). It makes sense to use several algorithms concurrently, planning algorithms for instance, by introducing them in different tasks, so as to be able to dynamically select them depending on current context and requirements. For that reason, we established a clear separation between the functionality description and the functionality implementation. **The task behaviour is contractually defined by a task type** that does not refer to any kind of implementation. It specifies the task functionality, the exchanged information and the configuration properties. It is up to the adaptation layer or the administrator to choose the best implementation depending on the context. Providing typing mechanisms is fundamental to ease the understanding of manager and for adapting it automatically

V. Control Layer: Tasks Organization

The purpose of the Control Layer (figure 1) is to handle the pool of administration tasks and to control their context-aware execution.

A. *Controller architecture*

It is based on a controller, made of several specialised modules:

- the **lifecycle controller** is responsible for the discovery of task implementations and for the creation, configuration, and destruction of task instances. This mechanism builds a task execution model than enables it to observe the manager behaviour.
- the **communication support** (or communication manager) enables tasks to exchange information. The task communication ports directly rely on this support.
- the **selection mechanism** (or selection manager) determines the tasks to activate in case of a conflict. Each task coordinator directly communicates with the selection mechanism.
- the **database** allows information sharing. Specifically, the database can be employed for storing and maintaining the processing history. The selection algorithm can equally use the database to get context-related information. Storing shared information is a facility provided to the developer. Although direct exchange is more efficient, practice showed that a standard method for exchanging and maintaining information simplifies developers' work. Certain situations, such as the presence of punctual events, naturally call for the introduction of a centralised database.

B. *Model@runtime*

The lifecycle manager offers monitoring interfaces for observing the manager behaviour. It produces a **model@runtime** [12] of the executed architecture and keeps it up-to-date. To build its model, the lifecycle manager relies on the administration modules of each task. The model contains task information (identifier, type, statistics and configuration including scheduler, coordinator and ports configuration) as well as controller configuration (selection mechanism configuration and general configuration properties). This model is used by the administration layer to evaluate and adapt the manager's behaviour.

C. *Conflicts management*

The **selection manager** is not mandatory. In most cases, developers can build their managers without using a selection mechanism as soon as there is no conflict management to be done. However opportunism is at the core of our approach and the ability to explore and compare different solutions is important when working with open environments. Thus, it is possible to deliberately create a system in which multiple tasks are competing for processing the same information. In such cases, the choice of which tasks to execute significantly depends on the context and on the objectives set by the administrator.

Tasks selection can be implement is many ways. A weak form of selection is to create a specific task which purpose is to receive the results of two or more competing tasks and to select one of them. Another solution is to implement a direct negotiation between tasks' coordinator expecting them to elect the best one. Finally, since the latter approach can be tedious, the solution our framework implements by default is the use of an arbiter. Here, all task coordinators (i.e. one coordinator per task) communicate with a centralised arbiter, which is part of the framework controller. Each task coordinator submits a request expressing the task's activation requirements. The arbiter receives all the requests and decides which tasks can activate and which tasks must remain inactive. We designed the arbiter to be *extensible*. The selection algorithm is proposed as a service than can be *easily rewritten* or *extended*. The framework provides several algorithms - inhibition based, token based, and priority based - but developers are free to use theirs.

The ability to change dynamically control is an important feature. It is thus possible to evolve the control from a weak exploratory solutions to a more robust and static one. Exploration can be reserved for testing purpose or specific time-period while robust static control will be deduced from statistics and learning and use on a day-to-day basis.

D. *Composite Tasks*

The framework offers the possibility to create **composite tasks**. Composite task *behaves exactly as a normal task* and can be used seamlessly. A composite is an encapsulation unit grouping coherent tasks - e.g. tasks that play the same role, tasks that are in conflict or tasks that handle a specific problem. They are made of a set of a tasks and dedicated controller and communication ports.

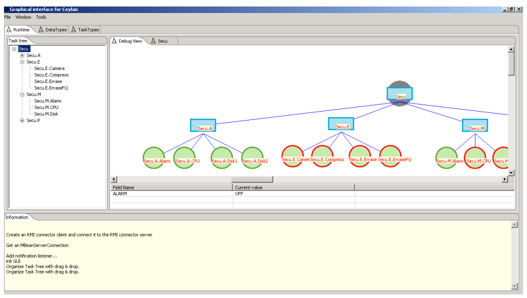
Enabling the framework to scale gracefully is the first reason behind the introduction of composite tasks. When the number of tasks increases significantly or when tasks have to handle various complex problems, the number of exchanged data messages may congest the communication support. Composites handle that problem by limiting messages scope. In addition, sharing information among multiple tasks by using a single database can become complicated. Once again composites limit the scope of shared data by providing a hierarchical implementation of database.

As each composite has its own selection mechanism, another important interest of composites is the ability to implement *different forms of control* for different subsets of tasks. For example a weak exploratory control can be used for specific tasks (e.g. for new tasks) while stricter one will be used for well-proven solutions.

VI. Administration layer: manager adaptation

The administration layer (figure 1) allows the modification of the overall behaviour of the manager by providing necessary tools. The modification of architecture and tasks configuration is accomplished in two ways :

- **Manually**: the framework provides administration tools to allow autonomic system expert reconfigure the combination policy of tasks. Particularly an **HMI** (Human-Machine Interface) offers a global view of the manager using the model@runtime as well as ways to modify its architecture (figure 3).
- **Automatically**: a administration module observes and analyses the functioning of the manager thanks to the model@runtime provided by the task manager.



Field Name	Current-value
ALARM	OFF

```
Create an RMI connector client and connect it to the RMI connector server

Get an MBeanServerConnection

Add notification listener...
init GUI
Organize Task Tree with drag & drop.
Organize Task Tree with drag & drop.
```

Figure 3. HMI allows to modify the manager at runtime.

In our approach, management policies results from the action of the tasks deployed, from their configuration and from the configuration selection mechanisms. They are written using an **Architecture Description Language** (ADL) in a description depicting the desired manager. This description is then compared with the *model@runtime* and the framework tries to keep both model synchronized using a model driven approach.

The administration can be automated using a *self-administration module*. This module based its reasoning on the model@runtime provided by the control layer and deduces the functioning of the control loop. It may then add, remove or re-configure tasks as well as reconfiguring the parameters of selection mechanisms. For example, a management component identified as defective or inefficient can be dynamically replaced with an alternative by re-configuring the tasks that work in concert with him. This results in an overall behaviour corresponding more accurately to its high-level goals. Auto-adaptation of autonomic-managers is of course complicated; we only experiment basic but realistic examples such as detecting and replacing improperly functioning tasks.

VII. Implementation and Application

A. *Implementation*

We have implemented the described architecture using SOC technologies and provided different extendable implementation for each task's modules. OSGi[1] was chosen for performance reason and because it provides a full administrable environment for deploying, installing, configuring, removing components at runtime. It allows direct communication between services (java direct call) once discovery has been done.

We implemented tasks as service-oriented components in order to ensure weak coupling. They can thus be deployed, installed, used and destroyed without impact on other tasks. Moreover, we rely on the service oriented component model iPOJO [13] to bring modularity to task modules. On top of these technologies, we based the task implementation on a dynamic mediation framework named Cilia[2]. It offers an ADL for the construction of mediators that we derived to propose a specific ADL in XML (*e.g.* figure 4).

In our approach the processor is implemented as a POJO (Plain Old Java Object) while the other modules are implemented as iPOJO handlers. Handlers are independent components that can be plugged to the component container during runtime to add new functionalities. This brings experts the possibility to develop and to extend task modules without modifying the processor. Many different task types can be then described on the mere basis of an XML file describing the composition and configuration of the modules without any modification to the code. The controller, and especially the selection mechanisms, has been implemented

[1]http://www.osgi.org/Main/HomePage
[2]http://wikiadele.imag.fr/index.php/Cilia

as a service-oriented component to allow modification at runtime. Their configuration and composition are also described using XML files. Managers are administrable using a GUI, JMX API as well as a basic scripting language. Each of these methods lets developers create and modify manager's configuration at runtime. With nearly 16000 lines of code the implementation is consequent. Further information are discussed in our preceding papers[14].

```
<task type="Plan.Alarm" implementation-name="Alarm-plan1" >
    <input-port ref="ceylan-direct-input"/>
    <scheduler ref="ceylan-scheduler-base">
            <exp value="ALARM^$base(RES="320-240-24), NONALARM^..."/>
    </scheduler>
    <coordinator ref="ceylan-direct-output">
            <forbiden>ANALYSE.ALARM,ANALYSE.DISK.D1</forbiden>
    </coordinator>
    <processor ref= "Alarm">
            <optional-input>ALARM</optional-input>
            <property name="delay" value="10000"/>
            […]
    </processor>
    <output-port ref="ceylan-direct-output"/>
</task>
```

Figure 4. Task's implementation definition

B. Application

We considered a residential surveillance application and particularly the part responsible for monitoring intrusions. It is composed of a set of services that allow remote monitoring of an apartment and the signalling of suspicious events. Users subscribe to the home security service. The application triggers an alarm in the home and informs the home security service. To do so, cameras are arranged in the rooms and a video recorder captures and stores images. The application runs on a OSGi-based home gateway. For convenience, we used simulated cameras in the following scenarios.

The figure 5 illustrates the behaviour of the application while using our manager. The x-axis represents time, while y-axis represents the CPU, memory and battery usage. The solid line represents the memory, while the dotted line represents the CPU. Our purpose was to create a manager responsible for different management concerns and to implement all this concerns progressively. For each one, we implemented several tasks for monitoring and acting on various aspects of the application: disk spaces, CPU usage, user's presence using its phone IP, camera battery usage, alarm.

First, we dealt with image storage space: the amount of possible images is limited by the size of the hard disk. By default, no management is done and disk becomes saturated. We implemented several planning tasks to manage storage depending on alarm. When no alarm is present (i.e. range 0-50, 180-292, and 363-450s) we alternate between compression (square) and removal (cross) of old images. When compression is used too often and is then not efficient we switch to removal using a token-based arbiter. We also try to takes CPU into account. During an alarm, if CPU is too high, the compress solution is replaced by a smart selection of images to implement deletion based on movement quantity. This is the why when CPU is high we use selection at 310s and 318s, and compression at 358s when it comes back to normal. When dealing with an alarm, if for some reason, the memory size crosses a 90% disk threshold, we use the removal task (at 342s). Of course, the way tasks are selected can be changed dynamically at runtime by modifying the configuration of the selection mechanisms.

Second, we managed the frequency and resolution of image taken by cameras depending on alarm. We take account the layout of the house to extend or limit the ranges of the cameras in rooms where an intrusion is unlikely (typically upper floor cameras). This is why, during alarm, the disk consumption grows significantly faster. The activation of the camera-management task is represented by a triangle.

Third, we managed to limit CPU consumption by using different movement detector implementation. When there is no alarm (i.e. range 0-50, 180-292, and 363-450s) we use an imprecise but CPU friendly algorithm and when there is an alarm we use a more precise one. The rhomb show the activation of the decision task responsible for switching between implementation. This is why, apart from the period between 250s and 340s where we simulate a huge CPU usage, the CPU usage is around 15% when no alarm and 40% when alarm.

We also used this approach to build other control loops. In particular, we implemented camera calibration, automatic switch-off of the alarm when user's PDA's mac-address is detected, camera battery management, and automatic choice of compression algorithm depending on CPU and efficiency. The use of our approach has permitted to develop each concern separately. Actually, we have developed many sub-managers. We were able to re-use a certain number of monitoring and execution tasks along with threshold detection to construct each of these aspects. We were also able to separately test and evaluate parts of the control loop. We showed that this set of tasks could be used simultaneously to obtain a globally complex behaviour of the manager. Thanks to selection mechanisms it is possible to achieve a globally coherent behaviour.

VIII. Conclusion

In this paper, we presented a Service Component Model for developing autonomic managers. We propose to build autonomic managers via the opportunistic composition of coherent and specialized modules called administrative tasks. Our purpose is to obtain an homogeneous and dynamic model for the integration of autonomic functions. This approach facilitates the evolution and extension of global management strategies by supporting the addition, the updating and the deleting of tasks, including the modification of the collaboration logic. Importantly, it allows the evolution of combination policies from very exploratory solutions to

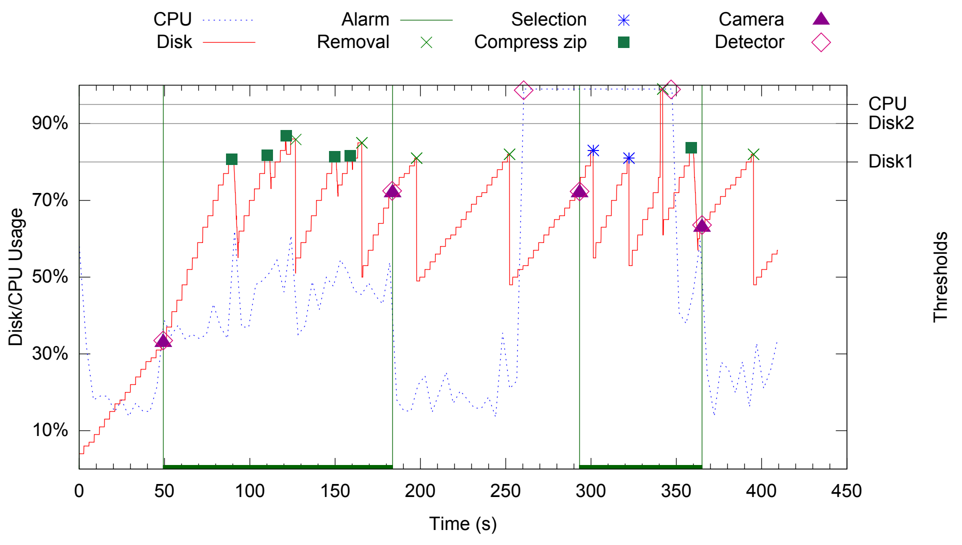


Figure 5. Using several tasks to manage application

more rigid but controlled ones. It is thus possible for a testbed to evolve towards an improved and reliable solution.

This framework has been fully implemented as a component model on top of proven service-based open source technologies, including OSGi/Felix, iPOJO and Cilia. These service-oriented technologies bring the necessary lazy binding and dynamism heavily used by our approach. Our framework has been used in collaborative projects in order to build robust, self-managed pervasive applications.

References

[1] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste, “Rainbow: Architecture-Based Self-Adaptation with reusable infrastructure,” *Computer*, vol. 37, no. 10, p. 46–54, 2004.

[2] H. Liu and M. Parashar, “Accord: A programming framework for autonomic applications,” *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 36, no. 3, p. 341–352, 2006.

[3] X. Dong, S. Hariri, L. Xue, H. Chen, M. Zhang, S. Pavuluri, and S. Rao, “Autonomia: an autonomic computing environment,” in *2003 IEEE International Performance, Computing, and Communication Conference*, 2003, p. 61–68.

[4] J. O. Kephart and D. M. Chess, “The vision of autonomic computing,” *Computer*, p. 41–50, 2003.

[5] N. D. Palma, S. Bouchenak, F. Boyer, D. Hagimont, S. Sicard, and C. Taton, “Jade: un environnement d’administration autonome,” 2007.

[6] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, “SERVICE-ORIENTED COMPUTING: a RESEARCH ROADMAP,” *International Journal of Cooperative Information Systems*, vol. 17, no. 02, p. 223, 2008.

[7] P. Deussen, M. Baumgarten, A. Manzalini, C. Moiso, M. Mulvenna, and E. Ho?fig, “Componentware for autonomic supervision services: The CASCADAS approach,” *International Journal On Advances in Intelligent Systems*, vol. 3, no. 1+2, p. 87–105, 2010.

[8] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz, “Music: middleware support for self-adaptation in ubiquitous and service-oriented environments,” *Software Engineering for Self-Adaptive Systems*, p. 164–182, 2009.

[9] S. A. Gurguis and A. Zeid, “Towards autonomic web services: Achieving self-healing using web services,” in *DEAS’05*, 2005.

[10] H. P. Nii, “Blackboard systems, part one: The blackboard model of problem solving and the evolution of blackboard architectures,” *AI Magazine 7(2)*, pp. 38–53, 1986.

[11] K. Decker and V. Lesser, “Task Environment Centered Design of Organizations,” *AAAI Spring Symposium on Computational Organization Design*, January 1994.

[12] N. Bencomo, G. S. Blair, and R. B. France, “Summary of the Workshop Models@run.time at MoDELS 2006,” in *Lecture Notes in Computer Science, Satellite Events at the MoDELS 2006 Conference*. Berlin, Heidelberg: Springer-Verlag, October 2006, pp. 226–230.

[13] C. Escoffier, R. S. Hall, and P. Lalanda, “iPOJO: an extensible service-oriented component framework,” in *IEEE International Conference on Services Computing, 2007. SCC 2007*, 2007, p. 474–481.

[14] Y. Maurel, A. Diaconescu, and P. Lalanda, “CEYLON : A service-oriented framework for building autonomic managers,” March 2010.