# Towards introspectable, adaptable and extensible autonomic managers.

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*—In this paper, we propose an architecture for building adaptable, extensible and introspectable autonomic managers. In that purpose, we introduce the concept of administration tasks: very specialized components that are opportunistically assembled into autonomic control loops. We discussed particularly the possibility of monitoring and modifying the managers' behaviours at runtime. Our architecture has been implemented as a service oriented framework using iPOJO/OSGi technologies and tested on a sample application pertaining to home automation.**

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

## I. Introduction

Autonomic solutions [1] will play an increasing role in the development of modern systems. Central to most autonomic systems are autonomic managers responsible for the self-management [2]- i.e implementing the adaptation process. In our view, autonomic managers must be:

- **Administrable/Introspectable**: managers are complex systems and their behaviour should be administrable. Self-management is the result of a succession of specialized activities (monitoring, inferences, aggregation, problem detection ...). Their implementation requires expertise and is often an overwhelming task. Our goal is to ease the implementation and the evaluation of these tasks and favour the reuse of specialized activities.
- **Adaptable**: managers' behaviours should evolve in the light of past performances/evaluations. Managers should display various behaviours depending on the evolution of context and goals.
- **Extensible**: managers should be able to adapt to different management situations and configuration. Establishing an exhaustive description in advance is hardly possible - particularly for pervasive environments [3]. One solution is to build extensible managers so as to take new situations into account.

In that purpose we propose an architecture and a supporting domain-specific **service-oriented component model** for building autonomic manager. We decompose managers' behaviours into specialized management functions implemented by discoverable components called **administration tasks** (figure 1). The main advantages include:

- Allowing the reuse of existing administration tasks - i.e. autonomic management functions;
- Enabling the easy and fast assembly of such reusable administration tasks into autonomic control loops;
- Enabling the dynamic evaluation of these tasks;
- Supporting the dynamic adaptation and extension of existing autonomic control loops;
- Supporting redundancy of administration tasks, which will whether work in parallel (if not in conflict) or will be selected by a specialized arbiter (if in conflict).

This paper first provides an overview of our framework and administration tasks and then explains how the manager is made introspectable and adaptable. Finally it presents implementation details and some applications.

## II. Framework Overview

Our framework is divided into three layers (figure 1) :

1) **the management layer**, composed of a set of administration tasks, managing the system;
2) **the control layer** managing tasks lifecycle and organising the tasks collaboration;
3) **the administration layer** providing the tools - HMI, API, ...- for building and managing the manager;

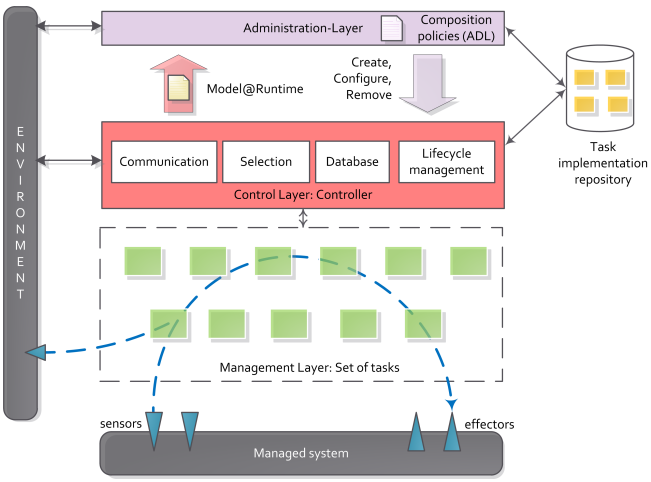


Figure 1. Architecture of a manager.

The system is managed by the **management layer** composed of a set of **administration tasks**: highly specialized

coherent component, performing a specific activity that advances a control loop's execution (e.g. monitoring a value, aggregating information or detecting a specific problem). Tasks are discovered/deployed at runtime - using a service oriented approach - and then **opportunistically** combined into autonomic control loops. These loops are thus not defined in a static way but are the result of the tasks' collaboration. Tasks may work in parallel so that several loops - dealing with specific concerns - may coexist at runtime. Depending on the context, the set of working tasks is refined dynamically by adding, updating or removing tasks. This way the manager behaviour can be adapted or extended.

Tasks offer an **homogenous integration model** for management activities, by featuring the same architecture and by exposing the same API, irrespectively of the control-loop phase they achieve. We propose a component model that establishes a clear separation between the concerns. A task is made of several modules (figure 2). The communication is realized by one input and one output communication port. The activation and triggering conditions are analyzed and managed by the scheduler. The coordinator avoids conflicts between tasks by communicating with a selection process provided by the control layer. The specific task algorithm is implemented by the processor. Two modules are for administrations purposes: the statistical module gathers information on task execution and the administration module exposes unified administration APIs.

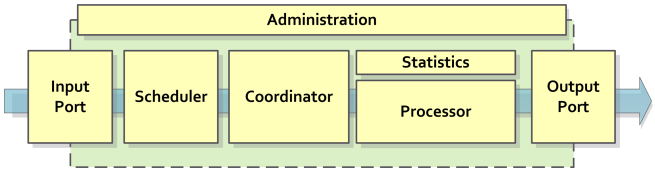


Figure 2. Task's sub-modules

Encapsulating management activities into functionally-independent homogeneous and coherent units favours their reuse and hides the implementation heterogeneity of the underlying collection, inference and execution methods. Proposing a component model allows the developer to extend or reimplement existing modules for specific purposes. Our framework provides fully functional reusable modules.

## III. Administration and Introspectability

To improve maintainability, our framework allows the dynamic monitoring of the manager at runtime and proposes a model-driven approach for building managers.

To allow their evaluation, each task is endowed with a statistics module. It executes at each task call, provides information on the task state (configured, valid/invalid, waiting, active or blocked) and calculates usage statistics (e.g. the number of task executions, the quantity and type of processed/produced data, the average task execution time, the number of times a task was considered blocked). The default statistics module can be extended to provide new types of information.

These statistics are part of the information gathered by the control layer to build a model@runtime[4] of the current manager execution. The model contains task information (identifier, type, 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. At any time the framework can provide a full description of the task activity.

We establish a clear separation between task type, task implementation and task instances. The implementation defines the implementation and the combination of all sub-modules. This concept corresponds to the component type or class concept. The same task implementation may have multiple task instances at runtime. The task instance concept corresponds to the component instance concept (or to the object/instance concept in object-oriented programming). At the same time, a given functionality can be realized in several manners (e.g. language, algorithms or targeted system). For this reason we introduce the task type concept; it enables us to standardize the use, the configuration and the data consumed/produced (data are typed too) by task categories that accomplish the same functions. Each task implements a specification that describes its utilization and the operation it realizes. Two tasks of the same types are substitutable.

The administration layer is at the origin of the creation and configuration of the manager. The manager is described in a targeted manager model. This description in terms of tasks types describing task's functionality is fully independent from the implementation. At runtime, a construction module is responsible for instantiating this model - by selecting task implementation - and synchronizing both model@runtime and targeted manager models. The administration layer offers all the necessary tools to monitor, build and modify the manager. In particular, it offers an API, an ADL interpreter and a GUI (figure 3)

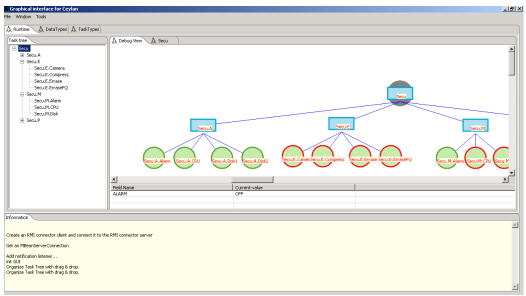

Figure 3. HMI allows to modify the manager at runtime.

## IV. Adaptability and Extendibility

One key strength of our framework is that the combination of tasks is opportunistic and may be refined during runtime. Adaptation and extension are made possible for the following reasons.

First, tasks are free to choose the appropriate moment to activate. Each task is endowed with a specific module, the scheduler, that determines when to activate. The triggering condition can depend on the collected data (type or quantity) as well as on external events (e.g. time). They are part of the model@rutime and their configuration can be changed dynamically at runtime.

Second, the framework supports redundancy of administration tasks, which will whether work in parallel or will be selected for by a specialized selection mechanism. One important adaptation in our system concerns the configuration of this mechanism provided by the control layer. By modifying the configuration it is possible to change the manner in which concurrent tasks execute and thus to change the overall manager behaviour. Having multiple tasks competing enables the management process to explore various possible solutions. We use two approaches for solving conflicts. First we use a synthesis task which will receive the contribution of each executed task and decides on an appropriate result. This solution is interesting for testing several solutions in parallel - however it might be difficult for the synthesis task to rollback some actions as this method intervenes after tasks' execution. Second, the control layer proposes one or more dedicated specialized components - the arbiters - that will be called by conflicting tasks to select which tasks can execute and which tasks cannot. These arbiters are implemented as services and can be replaced or reconfigured anytime during runtime to adapt the manager behaviour. Selection is solely performed when necessary: the expert configures tasks with the list of conflicting data types. We provide several arbiter implementations embedding different arbitration algorithm (e.g. data based, token based or priority based). Developers are free to extend them.

Third adaptation is made possible by using a service-oriented approach for task discovery. Tasks are implemented as services and the communication between them is direct and data/event-driven. Data producers and data consumers discover dynamically at runtime and it is possible to add, remove or update tasks dynamically during runtime. This way human or auto-adaption modules can extend the manager behavior using the aforementioned administrations tools.

In particular, the adaptation/extension of the manager can be automated by self-adaption modules using the model@runtime to identify the new requirements. This automation doesn't need to be complex for being useful. In particular we used such modules to replace blocked tasks or trying new task implementations when an implementation was judged faulty.

## V. Implementation and Application

Our architecture has been implemented in JAVA on top of the popular OSGi [1] framework. Service orientation brings the necessary weak coupling and dynamism for our approach. We based our implementation on Cilia[5], a service-oriented mediation framework implemented with iPOJO[6]. We extend it significantly to develop our component model. In particular, we derived the Cilia ADL to specialize it by hiding all the service references and iPOJO related technical details so as to raise the abstraction level.

To test our approach, we implemented a manager responsible for administrating a residential surveillance application. This application is made of several video cameras used to monitor an house. Captured images are used by an OSGi application running on a dedicated platform. Several services including motion detector are used to trigger alarm. We used this application to experiment a lot of different management situations -managing batteries, disk, CPU, alarm ... -and we showed how to implement these different concerns in isolation and then how to build a coherent manager in [7].

In this paper, we will solely focus on auto-adaptation scenarios. We select some basic scenarios - focussed on the disk and CPU management - and show how auto-adaptation can be performed. For convenience, we used a batch of precaptured images in the following scenarios. In the following graph (figure 4 and 5), the x-axis represents the time in s, the y-axis represents the usage in percentage of CPU (dotted blue line) and memory (solid red line).

In the first scenario, we try to adapt the framework when a task is detected blocked. The framework detect blocked task by checking that the average execution time is not going beyond a predetermined threshold. We integrate an automatic module in the administration layer responsible for auto-replacing blocked task implementation by an alternative when possible. This is what happened in figure 4, the first task is replaced by an alternative at 150s. Hopefully an implementation providing best result is found. When no alternative task is found -or when too much changed have been done in a short delay implying an unstable state- the administrator is asked to manually modify the manager. This scenario shows how a simple auto-adaptation module can be used to improve manager dependability.

The second graph (figure 5) shows the usage of arbitration. We develop three different algorithms - one erasing images and two different compression algorithms - for solving the same problem (avoiding disk saturation). Compression is always a better choice because it is not destructive, however it is only efficient for a short period because the number of images to be compressed diminish over time (as the proportion of compressed images increases). To solve that problem, we use a generic token-based arbiter. Each task has a certain maximum amount of token to spend on a given

[1] http://www.osgi.org/Main/HomePage

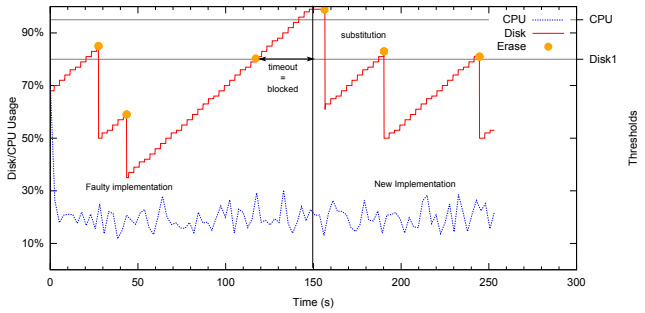


Figure 4. Substitution of a blocked task.

period of time - token are regained at regular interval. This is why on the first part of the graph (150-250s), one of the compression tasks (5 tokens) is used five times more than the erase task (1 token). Erase is called when the compression algorithm uses its token too quickly. This way the system is able to choose the non-destructive tasks first - and they are often sufficient. The two compression tasks are grouped into a composite task (basically a task composed of other tasks with its own selection mechanism). A second arbiter uses an estimation of CPU and compression efficiency of each task (this information is whether provided by implementations' meta-information or calculated by a special-purpose statistic module). When configuring the arbiter, the expert indicates what should be considered first (CPU or compression). At runtime it is possible to change the configuration to adapt the behaviour. This is why the compression algorithm is changed after the reconfiguration happening at 265. The new algorithm is more efficient (the memory curve descends more) but consumed more CPU (for the same number of images to compress the CPU usage is higher at 320s than at 180s).To ease comparison between the two tasks we have manually erased compressed images between 260 and 270s. These scenarios show how competing tasks can be used together to create a complex behaviour and how the manager can be reconfigured depending on goals.

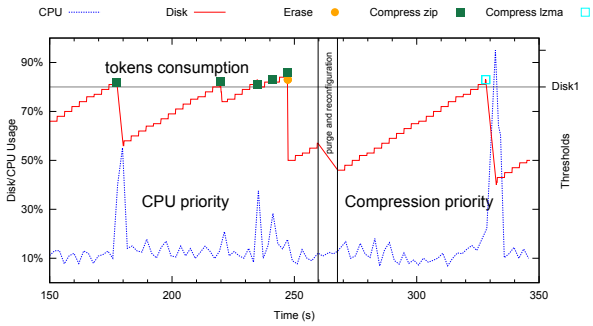


Figure 5. Using arbitration to select competing tasks.

## VI. Related Work

Autonomic Computing is based on an excellent logical architectures such as MAPE-K for Monitoring, Analyzing, Planning, Executing and Knowledge[2]. However this architecture is sometimes hard to implement and sometimes too constraining: why not merging Analysis and Planning for instance? The architecture blocks are coarse-grained and the different concerns are often mixed (one block for monitoring the whole context). They are frequently implemented using rules or coarse-grained component with negative consequences on maintenance or dynamism. Finally, as concerns are mixed, managers are hard to extend and reuse. In practice, most projects offer little guidance on the design of the manager internal architecture or are specialized in a specific adaptation method (e.g. Rainbow [8] and architecture adaptation). Tasks are finer grained than MAPE-K blocks and coarser-grained than rules.

The possibility of changing dynamically the internal architecture of managers is often ignored. MAPE-K does not provide guidance on how to manage dynamism between modules. When possible the modification often relies on rules or coarse-grained components (e.g. [9] [10]). The problem is that rules are fine-grained and often hard to predict, understand and maintain. Autonomic computing is often used for managing SOC applications ([11] [12]) but rarely to bring dynamism or building managers. [13] proposes to use Web Services but granularity is important and WS have a considerable performance cost. The solution advocated here has similarities with blackboard system[14] or with the TAEMS project[15]. However, in our solution control and data are much more distributed (tasks communicate directly with their counterparts and the selection mechanisms are only involved when necessary). Furthermore, we offer a complete framework and a component model with administration tools to observe and modify the behaviour.

## VII. Conclusion

Having adaptable, extensible, administrable managers avoids the need to provide an exhaustive description of the behaviors. Administration tasks provide homogeneous model for the integration of autonomic functions with an adequate granularity to build such managers. Using SOC, we are able to add, remove or update the behavior at runtime. In this paper, we show that simple and useful adaptation can be performed on managers and provide examples: 1) in manager internal state when tasks were blocked, a simple auto-adaptation module is able to replace them; 2) when the priorities of the manager change with the use of different compression algorithms based on indications provided by the administrator. Future works include a) evaluating performances - preliminary results shows that the performance cost is between 5% and 20% depending on the complexity and arbitration; b) distributing tasks across different systems; c) developing an IDE.

## REFERENCES


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

[2] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, pp. 41–50, January 2003. [Online]. Available: http://dx.doi.org/10.1109/MC.2003.1160055

[3] M. Weiser, "The computer for the 21st century," *Scientific American*, pp. 94–104, 1991.

[4] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 37–54. [Online]. Available: http://dx.doi.org/10.1109/FOSE.2007.14

[5] I. N. Garcia Garza, G. Pedraza, B. Debbabi, P. Lalanda, and C. Hamon, "Towards a service mediation framework for dynamic applications," in *Services Computing Conference (APSCC)*. Hangzhou: IEEE Asia-Pacific, December 2010.

[6] 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.

[7] Y. Maurel, A. Diaconescu, and P. Lalanda, "CEYLON : A service-oriented framework for building autonomic managers," in *Proceedings of the IEEE Conference and Workshops on Engineering of Autonomic and Autonomous Systems (EASe 2010)*, Oxford, England, March 2010.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] S. A. Gurguis and A. Zeid, "Towards autonomic web services: Achieving self-healing using web services," in *DEAS'05*, 2005.

[14] 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.

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