CEYLON : A service-oriented framework for building autonomic managers

Yoann Maurel, Ada Diaconescu and Philippe Lalanda
Laboratoire Informatique de Grenoble
F-38041, Grenoble cedex 9, France
(Yoann.Maurel, Ada.Diaconescu, Philippe.Lalanda)@imag.fr

***Abstract*—The important, ambitious goals of autonomic management applications require complex, adaptable reasoning capabilities that prove extremely difficult to conceive and implement. An increasing number of Autonomic Computing projects have provided partial solutions and studies that brought significant contributions to the understanding of this domain. At the same time, little support is currently available for facilitating the creation of complete autonomic management applications. This paper proposes a solution for the opportunistic integration of specialised autonomic management resources, conceived and implemented as services, so as to obtain complex, adaptable management strategies. The paper introduces an architecture that follows the proposed solution and provides a framework that implements this architecture. The solution's validity is indicated by experimental results obtained by testing the framework prototype on a home security application.**

***Keywords*-autonomic management; dynamic composition; problem solving modules; framework; service-oriented computing**

I. INTRODUCTION

The purpose of Autonomic Computing is to enable the self management of software systems and to minimize human intervention [8] [7]. In recent years, autonomic management solutions have become critical for business success in many domains [2] [3]. Not surprisingly, their complexity has notably increased in order to allow for the administration of more important and complex software systems. The significant complexity of autonomic management functions is directly linked to the ambitious goals they must attain. The monitoring, reasoning, decision and execution capabilities necessary for implementing autonomic management strategies involve multiple complicated tasks, at various abstraction levels. These include collecting important amounts of heterogeneous data, interpreting uncertain, incomplete, or contradictory information, diagnosing complicated problems and providing viable solutions while avoiding conflicting management plans. Additionally, autonomic management strategies must maintain a good balance among multiple, potentially conflicting business goals, including performance, reliability and security. Finally, autonomic managers must constantly adapt their administrative strategies, in order to react to changes in managed resources, accumulated knowledge, running environments and business objectives. Meeting such goals requires complicated reasoning capabilities, which are hard to design and implement.

An increasing number of Autonomic Computing projects have provided partial solutions and studies that brought significant contributions to the understanding of this domain. In general, available solutions focus whether on addressing specific management concerns (e.g. runtime monitoring, pattern recognition, optimal configurations, learning techniques, or dynamic resource modifications), or for administering a certain application, running on a given platform, with respect to a certain business goal [13] [6]. Nonetheless, little reusable support is currently available for integrating existing solutions and facilitating the development of complete autonomic management applications. Most systems are developed by engineers with a high expertise in the managed resources domain but not in the software engineering field. As a result, most developed autonomic solutions are hardly replicable and based on technical solutions that are difficult to test, reuse, and scale up. We believe that, in order to meet the usual standards of software quality for autonomic managers, we have to use well proven software engineering techniques. In particular, abstraction, modularity and separation of concern should be heavily used to build robust autonomic management solutions. In particular, a clear distinction should be made between expert code and reusable, non-functional, domain-independent code.

Another important issue, of course, comes from the challenging requirements to be met in order to build a generic enough framework for autonomic management. Defining reusable frameworks for guiding the design and implementation of Autonomic Computing solutions raises new challenging issues, not least because we are dealing with dynamic, non-deterministic reasoning processes. In particular, an autonomic solution must be able to adapt to various configurations. Different reasoning strategies have to be used depending on the autonomic management process state. For instance, when an autonomic configuration fails to solve a given problem, it is necessary to try out new options. Similarly, when a very specific problem occurs, it is often necessary to add specific management resources. We have worked on such an issue in a recent project[1] dealing with electrical distribution. A remote installation suffered from an electrical default every day at noon. The collected data were not sufficient for a (human or electronic) expert to detect the

[1] Autonomic Networks for SOHO users: http://anso.vtt.fi

problem. Hence, it became necessary to inject different data collectors and analyzers in order to cope with the problem.

Additionally, an autonomic solution is based on opportunistic reasoning. This indicates that the way a problem is detected, analyzed and solved is determined during runtime, depending on the current conditions and requirements. When autonomic managers must react to complicated and unpredictable scenarios, the space of detectable conditions and desirable decisions grows exponentially. In these cases, it becomes difficult, or impossible, to statically predict all possible situations and provide all necessary solutions in an autonomic manager. The approach where developers fully specify and control the overall application behavior is hard to apply.

In this paper, we advocate the adoption of a service-oriented approach for developing autonomic management applications. We propose to build complex administrative strategies from simple, specialized autonomic tasks conceived and implemented as service-oriented components. In this view, specialized tasks are dynamically identified and assembled for detecting and solving complicated, possibly unexpected problems. During runtime, the available autonomic tasks form opportunistic collaborations depending on the current situation, requirements and available information. A precise, exhaustive specification of monitoring, analysis, planning and execution directives is no longer required.

Service orientation brings the necessary flexibility, allowing reconfiguration and dynamic integration of management tasks. Service-Oriented Computing (SOC) promotes loosely-coupled architectures [12] [14]. It aims to reduce dependencies among composition units, letting each element evolve separately, so the application is more flexible than monolithic applications. It also promotes substitutability: a service can be transparently replaced by another service, as long as both services implement the interface defined in the provider-consumer contract. Implementation and platform heterogeneity are hidden from service consumers. Several implementations meeting the SOC principles have been proposed. Web Services [2] are the most prominent one. However, several other implementations currently exist, including UPnP [3], OSGi [4] and iPOJO [5] [4]. A salient aspect of SOC is the way in which service compositions are achieved. Service composition is essentially a matter of control, which can be extrinsic or intrinsic to services. In the first case, services are called in accordance with a process - an oriented graph of services. In the second case, service-oriented components[1] manage their own interconnections and interactions. This latter approach has been recently gaining an important momentum and is being used in various domains such as pervasive computing and enterprise systems.

[2]Web Services : www.w3c.org
[3]Universal Plug and Play : www.upnp.org
[4]Open Services Gateway initiative: www.osgi.org
[5]iPOJO: www.ipojo.org

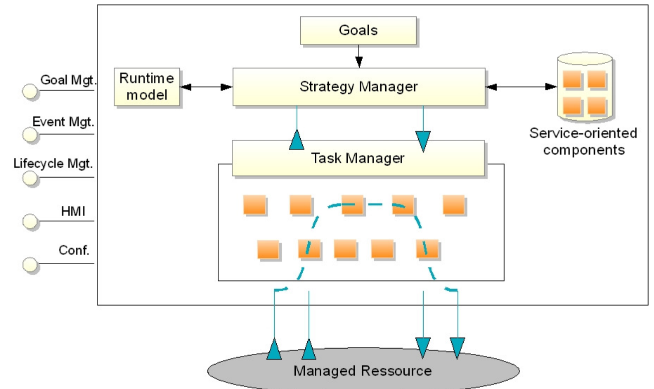

Figure 1. Proposed two-layer architecture

This paper proposes an architecture that follows the presented approach and provides a reusable framework, named Ceylon, that implements this architecture. The solution's validity is indicated by experimental results obtained by testing the framework prototype on a sample pervasive application.

II. Proposal

A. Overall presentation

Our approach is to implement an autonomic manager as an opportunistic composition of loosely-coupled service-oriented management components. Each management component implements a simple administrative task, such as monitoring a parameter, detecting a problem type, planning a specific solution or modifying a managed resource. Communication between management components is asynchronous and topic-based. A management component's execution is triggered by events from a predefined set of topics. In turn, a component's execution may generate a number of topic-based events.

In this manner, complex management strategies are developed through the integration of simpler administrative tasks. Composition is opportunistic in the sense that it is performed at run-time, depending on run-time phenomena. Depending on the problem solving state and on the earlier performances of the management components, different management components are chosen and connected. The set of available management components implementing simple administrative tasks can be modified over time in order to cope with evolving conditions and requirements.

As illustrated in Figure 1, we defined a two-layer architecture. A first layer, the resource management layer, consists of the available management components administered by a Task Manager. As we will see in more details, management components can be simple or composite (i.e. made of other management components). The purpose of the Task Manager is to provide a shared communication channel and to decide on the management components to be activated. Such

decisions are driven by higher-level management strategies, which are implemented via specific configurations of the Task Manager. A second layer, the manager adaptation layer, consists of a Strategy Manager, which supervises and reconfigures the lower, task management layer. Namely, the Strategy Manager observes the solving process executed at the task management level. It develops a runtime model of the autonomic management process, including information about the management strategies formed and the associated management components activated. As such, the runtime model contains information about the performances of the activated components and about the advances of the consequent management actions. Based on this information, the Strategy Manager may chose to dynamically modify the service composition process, in order to obtain an overall behavior that more closely corresponds to its high-level administrative goals. At present, the Strategy Manager has the ability to add, remove or reconfigure management components, as well as to reconfigure the Task Manager's service composition parameters. For example, a management component identified as faulty or inefficient can be dynamically replaced by an alternative component by correspondingly reconfiguring the Task Manager's communication channel.

B. Resource Management layer

1) Autonomic management components: As indicated, our approach to developing autonomic management is to specify and implement administrative operations as loosely-coupled service-oriented components. In this approach, an autonomic manager is made of several such management components interacting through an event-based protocol. The component-based approach leads to modular, evolvable code and allows the reuse of fine-grained management operations. Using service orientation brings additional flexibility since service components can be dynamically loaded, unloaded, adapted and interconnected.

We defined an autonomic management component via the following interfaces:

- Goal management interface. This interface allows specifying the component's objectives. Goals are translated by the component into specific actions to be undertaken or into specific algorithm parameterizations.
- Event management interface. This interface is dedicated to event reception and transmission. As we will see in section 2.3 this interface exposes publish-subscribe methods.
- Lifecycle management interface. This interface allows manipulating or getting information on the component's state. A component can be installed, started, stopped or resumed.
- Human-Machine Interaction (HMI) management interface. This interface defines methods that provide user-friendly information about the component's purpose and state during runtime.
- Configuration management interface. This interface allows the selection and configuration of non-functional features, such as persistency or security.

Our component model follows a fractal approach in the sense that a management component can be simple or composite. A simple management component implements an administrative operation (or a management activity) via a receiveEvent method, which is part of the Event management interface. This method takes a set of events as input and delivers a set of events as output. It performs a basic management action related to monitoring, analysis, planning or execution, or to any combination of these preoccupations. A composite management component replicates the architecture presented in Figure 1: it contains a set of management components, a Task Manager and a Strategy Manager. Simple and composite management components expose the same interfaces and are manipulated in the same way. Specifically, a Task Manager does not know the nature of its management components (simple or composite).

This structure allows our solution to scale to large numbers of management components. In simplest cases, a flat architecture made of a small number of components will be sufficient. In more complex systems, possibly distributed, it is of major importance to group management components into a same composite in order to be able to manage correctly the reasoning process (more precisely, to be able to express the way to control components). Such grouping can be driven by similarities in matters of concerns, location, response time, etc.

Autonomic management tasks are generally difficult to develop because they require different skills. First, developers must be experts in the underlying managed system in order to collect the right data at the right time, decide when enough data is received, perform relevant analysis, and decide on the right actions to be performed and so on. Only domain experts can have such capabilities. In addition, developers also need specific skills in component-based software engineering, in order to design and develop the set of interfaces and non-functional mechanisms presented before. For instance, event management requires mastering synchronization issues, which are highly error-prone in most languages. Such thorny non-functional features demand complicated code to be delivered. Experience shows that such double expertise is almost impossible to achieve or find.

For these reasons, we developed a container-based approach for designing and implementing our service-oriented component model. Our purpose was to clearly separate the code related to domain-specific autonomic tasks (e.g. monitoring or planning) from the code dedicated to generic component-based software issues (e.g. dynamicity, late binding, synchronization or error handling). The idea is to allow domain experts to focus on familiar concepts and ignore, as much as possible, complex technical code related to software

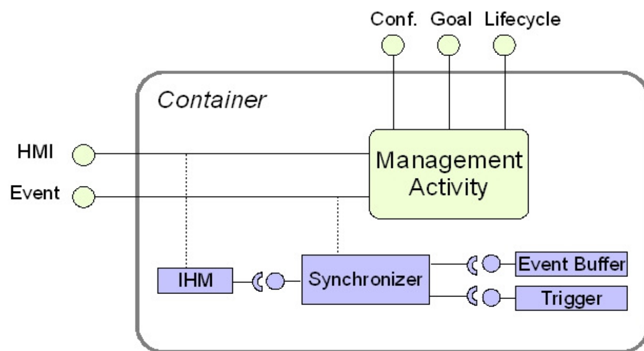

Figure 2. A container handles non-functional aspects like synchronization or HMI

issues. A container can be seen as an envelope that manages the execution of domain-specific code. It intercepts incoming and outgoing calls in order to insert domain-independent technical code, as needed for the execution of domain-dependent operations. The container can be parameterized via a configuration file.

As illustrated in Figure 2, the purpose of the container is to add technical code related to the engineering of service-oriented component-based systems. In particular, it deals with event communication, administration, HMI and lifecycle. The container acts as a black-box to the domain expert. Hence, from the domain expert's point of view, a management component is defined by an autonomic task and by a configuration file. The configuration file provides all the necessary information to activate and insert the management component in a larger reasoning context. It specifies the events of interest to the component and the component's triggering condition. A triggering condition specifies the situation(s) under which a component will execute. It is a predicate based on data (events) that the component receives over a given period. Specifications in the configuration file are used to generate code in the container. At the same time, it is important to note that the configuration file is presented in domain-oriented terms so that domain experts need not be aware of all the technical implications.

The container is itself made of several service-oriented components. They are invoked when specified component methods are called. A Synchronizer service is an important part of the container. It is called whenever an event is received. It is made of complex Java code dealing with synchronization issues. Specifically, it deals with the treatment of concurrent events and with the memorization of received events. Event storage follows a periodic strategy: all the received and emitted events are persisted in a local file and preserved for a configurable duration. Additional events, considered to be of major importance, can be kept in the local storage for longer than the configured duration. At each event, the Synchronizer calls a triggering function specified by the domain expert in the configuration file. As soon as the triggering predicate is achieved, the management component is turned to a ready state (via the component Life-cycle interface).

We developed a number of additional container components. For instance, a service-oriented component has been developed to provide administrative information from the container (i.e. the HMI service). As indicated on Figure 2, this component has a dependency on the Synchronizer service. The container also deals with the dynamic aspects of the service-oriented components (e.g. automatic service discovery and binding based on specified dependencies).

2) Task Managers: In the proposed architecture, autonomic management components are grouped into a problem-solving space controlled by a Task Manager. The purpose of the Task Manager is twofold. First, it has to ensure the effective communication between individual management components. Second, it has to coordinate the global problem-solving process. Management components communicate through events transmitted via a shared publish/subscribe Event Manager, which is part of the Task Manager. The role of the Event Manager is to distribute the events to the right parties (i.e. impacted management components). Events are organised into topics of interest. A topic of interest (or topic) corresponds to a certain management concern, or preoccupation. For example, a topic may represent the state of a certain monitored resource, the presence or absence of a certain problem type or the proposal of a certain kind of management plans. For efficiency reasons, management components are only aware of topics where data they can interpret may occur. Therefore, the Event Manager uses a publish/subscribe model, which delimits the area of interest of each management component. Hence, management components subscribe to topics they can interpret and publish their results to topics affected by the data types they produce.

Management component activation is the second function assigned to the Task Manager. The flexibility of CEYLON relies on the ability to dynamically orchestrate the management tasks under its responsibility. As illustrated by figure 3, a special-purpose Controller is notified as soon as the triggering condition of a management component is satisfied. In that case, the Controller must decide on the activation of the management component and possibly on its launching time. Conflict resolution is an important aspect treated by the Controller. This function prevents the activation of conflicting management components when events triggering their simultaneous execution occur. Such situation may arise if alternative or redundant management components were available for addressing the same management concern. For example, multiple planning components may be introduced to deal with the occurrence of the same problem type. A different planning component may be preferable in diverse execution contexts or when different business goals have priority.

There are several ways to organize tasks activation and

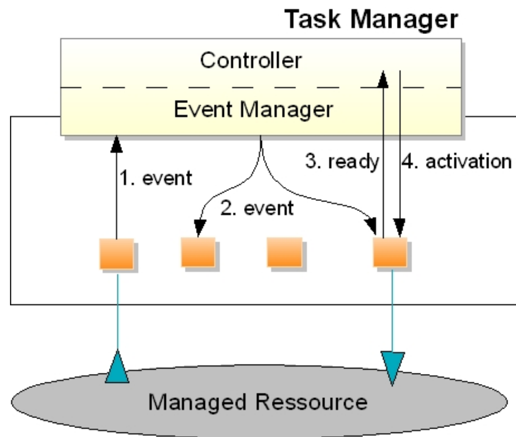

Figure 3. The Task Manager ensures communication and the opportunistic activation of management components.

thus different associated conflict resolution methods: it depends on the manager requirements and complexity. In simple case, one may, for instance, use a token-based algorithm to obtain a turn by turn behaviour. If more flexibility is required an algorithm based on inhibition and/or activation might be required. In the default implementation, conflict resolution is based on an inhibition mechanism: the activation of a certain management component inhibits the activation of conflicting management components. There are several ways of implementing such method from using activation lease time to using more bio-inspired algorithm based on the tasks excitation for instance.

The accuracy of the conflict resolution mechanism relies on the amount of information available at a given moment. One classical solution to solve such problem is the use of a temporal window which represents a fixed time period. The size of the window depends on the manager requirements. During such a period, the Controller may receive several notifications of readiness. At the end of the period, it uses a decision function to select the best component(s) to be used, depending on the problem solving state. When there is no conflict between potential autonomic components, the size can be shortened. The Controller is not used in this situation, as management components do not have to ask for an authorization to start. We readily acknowledge that using a temporal window may delay important decisions. On the other hand, however, deciding on the activation of a component right away frequently leads to wrong decisions. Conflicting or corroborating events often occur very close in time and they have to be all considered in order to properly solve a case.

C. Manager Adaptation: The Strategy managers

The purpose of the Strategy manager is twofold. First, it has to transform high-level goals specified by a human manager into decision functions used by the Task Manager. At present, decision functions are expressed as configuration tables specifying the mapping between topics of interest and management components, as well as preferred management components in different contexts. The Strategy Manager has also to dynamically supervise and reconfigure the Task Manager in order to achieve its goals. s indicated in Figure 1, the Strategy Manager develops a runtime model of the autonomic management process. A runtime model, as defined in [5], presents views of various aspects of an executing system and is hence an abstraction of runtime phenomena. Such a model is useful to support dynamic adaptation of software-based systems. In our case, the runtime model includes:

- Management component profiles, describing the components' characteristics in different contexts. Profiles may include: performance characteristics - response times or consumed resources; the efficiency for solving different problems; or dependability characteristics - reliability or safety. A management component may feature different characteristics in different contexts. Additionally, management components may be profiled in isolation or as part of larger collaborations with other components.
- The current state of activate management components (e.g. which components are presently active and for how long have they been executing?). This information is used for dynamically detecting and replacing inefficient or blocking management components. It is also used for updating the management component profiles.
- A history of signaled problems and consequently activated solutions. This information serves to evaluate the efficiency of current management strategies. It can be used for detecting repetitive patterns in the autonomic system behavior, which may indicate the ineffectiveness of current strategies.

Based on the collected information, the Strategy Manager has the ability to change the way the integration of management components is conducted at the Task management layer. Two main options are open. First, the Strategy Manager can change the Task manager's decision tables. Thus, alternative reasoning paths can be explored. Otherwise, the Strategy Manager has the ability to add, remove or reconfigure management components. Such components are available in a dedicated repository. They can searched, selected, instantiated and deployed in the workspace controlled by the Task Manager.

D. Framework implementation

To implement our approach, we have used an extensible service-oriented component model, namely iPOJO [9] [10], which is available from the Apache foundation and is based on OSGi. OSGi supports the dynamic deployment of services without requiring any prior knowledge of the service implementation. In addition, it provides infrastructural

services such as a service repository and a lightweight, local communication support.

IPOJO provides important OSGi extensions, including distributed communication and automatic service discovery and binding for dynamically resolving service interconnections. An iPOJO component provides server and client interfaces exposing its functionalities and dependencies, respectively. At run-time, an iPOJO component is managed by a container, which injects non-functional facilities into the application component as necessary. IPOJO functionalities are implemented following the service orientation paradigm: default facilities include service provisioning, service dependency and lifecycle management. Once an iPOJO component is deployed, the component's provided functions are published and made available as services, in conformance with the SOC paradigm. In order for a component's services to become valid, all the component's dependencies must be resolved. IPOJO is seamlessly extensible with new functionalities: each container can be configured with a different set of services, implemented as handlers. Handlers are themselves service oriented components with dependencies. Dependencies are resolved at run-time, following the SOC philosophy. These services can be in the container or in the run-time platform. These properties made iPOJO a natural choice for the current framework implementation. Hence, each autonomic management component is implemented as an iPOJO service component. For this purpose, we have developed a number of additional handlers in order to define our domain-specific, service-oriented model.

The Task Manager represents another, composite iPOJO service. Autonomic management components are connected to the Task Manager's communication channel via dynamic iPOJO bindings. The initial composition of a framework instance is specified via a special-purpose XML file (e.g. management component types, instances and initial subscriptions to topics of interest). The specification language is technology-independent, so as to facilitate possible migrations to other Service Oriented Component platforms. The strategy management layer supports simple administrative operations, including the observation of active management components, the detection of repetitive, inefficient, or faulty activations, the dynamic addition or removal of management components and the reconfiguration of component priorities and topic associations.

III. Experiments

A first prototype of CEYLON has been developed focusing on the conflict resolution mechanisms. An experimental application was used for validating the framework prototype. The application was implemented based on our team's experience with the ANSO Project. It supervises a home and notifies a security company when an intrusion is detected. Several video cameras are employed for capturing images from different rooms. They communicate with the application via specific drivers. Images are sent to a motion-detector, which searches for differences that may indicate movement. In parallel, images are sent to a persistence service, which saves them to a local file system. Guards notified by alarms may first analyse the available images before intervening. The application runs on a home gateway executing multiple applications. The goal of the autonomic framework is to ensure the functioning of the security application despite fluctuating gateway resources. Hence, tested scenarios focus on the video cameras and image storage service management, depending on the current application state (i.e. normal or alarm) and on the available disk and CPU resources.

The current conflict resolution mechanism is based on an inhibition mechanism with a very short temporal window. The Task Manager currently uses three tables for dispatching data to tasks. First, a Topic Mappings Table defines classical publish/subscribe topic subscription. Second, a Conflict Mapping Table describes couple of conflicting tasks. An expiration duration states the time that a task should be inhibited for when its conflicting counterpart is activated. Finally, an Inhibition Status Table shows the inhibition status of each task. When a task is inhibited, priority and expiration time are stored in this table. At the same time, more than one inhibition may concurrently exist for a single task entry. It occurs when different tasks place inhibitors with overlapping periods; in that case the highest priority is considered at each moment. Mapping tables are initially set by system administrators and refined by the Adaptation Layer at runtime. This refinement is the key of the opportunistic behavior of the manager. The Task Manager uses the Topic Mapping Table to forward data to subscribing tasks. For each of these tasks, the Task Manager uses the Inhibition Status Table to verify whether subscribing tasks are blocked. If the task is currently blocked, the priority of the incoming data is compared with the priority of the blocked task inhibitions. If the incoming data has a higher priority, the inhibition is ignored. Otherwise, the affected task is discounted. The conflicting tasks are blocked, in accordance with the Conflict Mapping Table.

The framework instantiation consisted of four Composite management components linked in a "classic" control-loop:

- The monitoring Composite uses three management components observing the alarm service state, the CPU utilization and the disk consumption levels. These components produce data of types: monitor.alarm, monitor.CPU and monitor. disk.
- The analysis Composite uses four management components for processing monitored data. The first component detects the presence or the absence of alarms. The others detect the crossing of different predefined thresholds by the consumptions of disk and CPU. These components are instances of the same implementation but are configured with a different threshold and activated

by different topics. One is activated by monitor.CPU with a threshold set to 95% of the CPU capacity. The other two analyzers are activated by monitor.disk data with thresholds set at 80% and 90% of the disk capacity. The four analysis component produce results of different data types and priorities: analysis.alarm (priority 2) and analysis.non-alarm (priority 4), analysis.CPU (priority 3), analysis.disk1 (priority 1) and analysis.disk2 (priority 1).
- The planning Composite contains four management components: Cam Planner, Standard DiskPlanner, Alarm DiskPlanner, AlarmCPU DisPlanner.
- Four execution management component were necessary for the execution Composite for managing camera activations, deleting a certain quantity of old images, erasing one image out of every given number and compressing recorded images.

The Cam Planner MR decides to switch on/off cameras located in rooms with no access points, depending on the alarm state. This planning task is activated by the analysis.alarm or analysis.non-alarm topics and issues plan.camera-on and plan.camera-off. The Standard DiskPlanner component, activated by analysis. disk1 topic, decides to delete 40% of the stored image. Alternatively, The Alarm DiskPlanner task can be activated by analysis.alarm, analysis.disk1, or analysis.disk2 topic. Its decision depends on two possible data type combinations. When analysis.alarm and analysis.disk1 data types occur simultaneously, this component orders registered images compression. However, when disk usage approaches the maximum, the planner tries to save the most recent images.

Therefore, when activated by the concurrent occurrence of analysis.alarm and analysis.disk2 data, the same Alarm DiskPlanner task decides to delete 40% of the oldest image records. The Alarm DiskPlanner task only proposes a disk management action in case an alarm is present; this planner remains inactive in the absence of an alarm. The default inhibition state of the Alarm DiskPlanner was set to 2. This enables data of type analysis.alarm (of priority 2) to activate this component. However, it prevents analysis.disk1 and analysis.disk2 (of priorities of 1) to activate this component alone. When an alarm activates Alarm DiskPlanner, the component also receives disk related data. The second Ceylon prototype currently under development has been extended, so as to support logical expressions simplifying these settings.

The last alternative is AlarmCPU DiskPlanner activated by analysis.alarm, analysis.disk1, analysis.disk2 or analysis. It remains inactive as long as there is no CPU overload, but once activated it receives all data affecting its decisions (i.e. CPU, disk and alarm). For this reason, this planner's default inhibition state was set to 3 (priority of analysis.CPU topic). When activated, this planner orders the deletion of one in three recorded images. This planner preserves CPU consumption avoiding image compression. When the second disk threshold is crossed, this component orders the deletion of 40% of images.

Some planning component decisions are conflicting. These conflicts are managed by the aforementioned conflict tables. The Cam Planner is not in conflict with any of the other planners and never gets blocked. The three other planning components try to mutually inhibit each other, whenever active, using the maximum priorities of the data that activates them. Based on analysis data priorities, the AlarmCPU DiskPlanner can inhibit the Alarm DiskPlanner, which can inhibit the Standard DiskPlanner.

The fully implemented application works with real video cameras. However, for better controlling the scenarios, experimental drivers simulate the existence of multiple cameras. The results are displayed in Figure 4. This graph shows the percentages of CPU and disk consumption, the starting and stopping of alarms and the alternate activation of the four execution management components.

The starting and stopping of alarms was sensed by the alarm monitoring component and triggered the alarm analysis component. This component produced analysis.alarm data which triggered the Cam Planner and resulted in data of types plan.alarm-on. The graph shows the activation of the camera almost superposed with the starting and stopping of application alarms (e.g. at times 120s and 380s for starting; and 210s and 460s for stopping). When more cameras are activated (at 120s and 380s), the slope of the disk consumption curve increases: more images are stored. When the alarm was off, disk overloads were handled by the Standard DiskPlanner ordering deletion (times 215s, 275s and 330s).

On alarm (analysis.alarm) and first disk threshold (analysis.disk1), the Alarm DiskPlanner was activated. The planner ordered compression, by activating the compression execution component. The activation of this execution resource occurred 5 times during the first alarm period, between 120s and 210s. This strategy became less and less efficient as the sizes of already compressed images could no longer be reduced. On alarm and second disk threshold (analysis.disk2) the Alarm DiskPlanner order deletion instead of compression. This activation is shown at the approximate time of 180s. The same component was activated at time 410s, but this time ordered by AlarmCPU DiskPlanner. This occurred when data of type analysis.alarm, analysis.disk2 and analysis.cpu was simultaneously present in the communication channel. However, when the disk consumption was in-between the two thresholds (i.e. presence of analysis.alarm, analysis.disk1 and analysis.cpu data), the AlarmCPU DiskPlanner chose the selective image deletion strategy instead, activating the corresponding execution component (plan.step-delete). This scenario occurs twice on the graph, during the second alarm period, when the CPU threshold is also crossed (i.e. at times 390s and 425s). The CPU overload

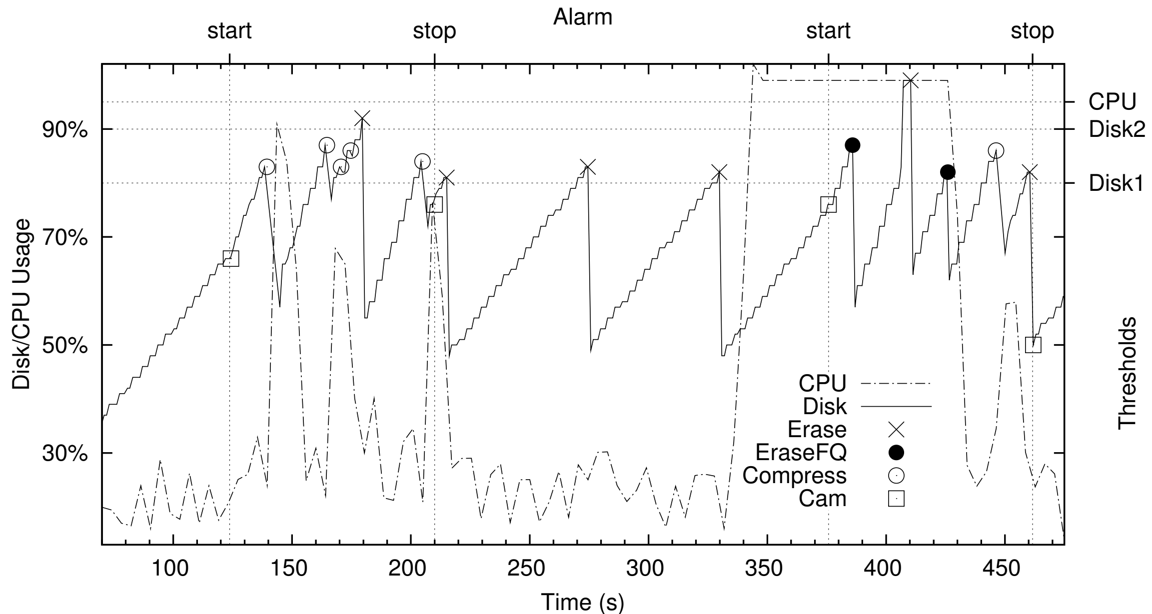

Figure 4. The Task Manager ensures communication and the opportunistic activation of management components.

was induced by a processor-intensive application.

These results show how the proposed framework can handle various administrative scenarios by dynamically creating management strategies from components of various types. The flexibility of the adopted solution allowed the seamless definition of behaviours for handling diverse combinations of external conditions.

IV. Related work

The importance of applying the service-oriented paradigm to autonomic management applications is reflected by the publication of specific Web services standards, namely the Web Service Distributed Management [9]. In this context, Miller [10] also shows the advantages of standardizing the interfaces of autonomic management elements, as it allows the creation of autonomic applications from individual services developed by multiple providers. The availability of such interfaces is vital for building adaptable autonomic managers with dynamically interchangeable elements, as in the case of our approach. In the autonomic computing field, several projects have started to develop generic architectures, engineering principles and execution platforms with reusable capabilities that facilitate the creation of autonomic management applications (e.g. IBM Autonomic Computing Toolkit [6] , Autonomia [7] , AutoMate [8] , BioNets [9] , Amorphous Computing [10] , Autonomic Networked Systems [11] or ROC [12]). These projects have reached different maturity levels and propose autonomic management functions and infrastructures that are mostly complementary with the proposed solution. Therefore, the capabilities of existing platforms and their possible integration with the proposed framework will be continually studied and evaluated.

Other research areas relevant to our proposal are concerned with the development of automatic reasoning functions. Such areas include Artificial Intelligence, Robotics and Automated systems. Most significantly, concepts related to Multi-Agent Systems and Blackboard architectures [11] seem most tightly related to our approach. Nonetheless, in Multi-Agent Systems, agents are autonomous entities capable of identifying and of negotiating with peer agents in order to form necessary collaborations. In our approach, collaborations emerge from the simple reactions of management resources to the occurrence of data they can interpret. This behaviour closely resembles that of the Blackboard model and consequently features similar advantages and difficulties. In contrast to agents, the autonomous capabilities of individual management resources are quite reduced, with the connecting channel providing rather basic communication facilities.

[6]Autonomic Computing Toolkit: www.ibm.com/developerworks/autonomic/overview.html

[7]Autonomia (University of Arizona): www.ece.arizona.edu/hpdc/projects/AUTONOMIA

[8]AutoMate (Rutgers University): automate.rutgers.edu

[9]The Bio-Networking Architecture (University of California Irvine): netresearch.ics.uci.edu/bionet

[10]Amorphous Computing: swiss.csail.mit.edu/projects/amorphous

[11]Autonomic Networked Systems (ANS) (Imperial College): www.doc.ic.ac.uk/asher/ubi/ansproj

[12]Recovery Oriented Computing (ROC) project (Berkeley and Stanford Universities): roc.cs.berkeley.edu

V. CONCLUSION

We believe that the service-oriented computing approach brings the opportunity to reconsider the way we design and implement many systems, in very different domains. Its inherent modularity and dynamicity allow meeting stringent requirements in an affordable way. This is the case of autonomic managers, which are so far implemented using particular, hardly replicable techniques.

In this paper, we have presented a framework for building autonomic managers based on service-oriented components. We believe that the major contributions of this work are the following:

- The design and implementation of autonomic tasks as services;
- The implementation of different management concerns in isolation;
- The dynamic, opportunistic integration of available autonomic tasks so as to obtain more complex behaviors allowing the adaptation to fluctuating, unpredictable conditions, as adaptable solutions are dynamically created;
- The possibility to add, update or remove autonomic tasks and modify their collaboration logic at run-time;

In addition, the approach allows domain expert to focus on the management tasks of the monitored systems. Management tasks are encapsulated in a dedicated component; the component's container taking care of most of the software engineering issues.

REFERENCES

[1] H. Cervantes and R. Hall. Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model. In *ICSE*, pages 614–623. IEEE Computer Society, May 2004.

[2] A. Diaconescu, Y. Maurel, and P. Lalanda. Autonomic management via dynamic combinations of reusable strategies. In *Second International Conference on Autonomic Computing and Communication Systems, Autonomics 2008*, Turin, Italy, September 23 - Sep 25 2008. ICST.

[3] A. Diaconescu and J. Murphy. Automating the performance management of component-based enterprise systems through the use of redundancy. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, page 53. ACM, 2005.

[4] Escoffier, Hall, and Lalanda. iPOJO: an extensible service-oriented component framework. In *IEEE SCC, Salt Lake City, USA*, 2007.

[5] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.

[6] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[7] M. C. Huebscher and J. A. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Comput. Surv.*, 40(3):1–28, 2008.

[8] D. M. Kephart, Jeffrey O. et Chess. The vision of autonomic computing. *Computer*, 36, 2003.

[9] H. Kreger and T. Studwell. Autonomic computing and web services distributed management, 2005. www.ibm.com/developerworks/autonomic/library/ac-architect/.

[10] B. Miller. The Standard way of autonomic computing, 2005. www-128.ibm.com/developerworks/autonomic/library/ac-edge2/.

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

[12] M. P. Papazoglou and D. Georgakopoulos. Service Oriented Computing. *Communications of the ACM*, 46:25–28, October 2003.

[13] S. Sicard, F. Boyer, and N. D. Palma. Using components for architecture-based management: the self-repair case. *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 101–110, 2008.

[14] J. Yu and P. Lalanda. Integrating UPnP in a development environment for service-oriented applications. In *IEEE ICIT*, 2008.