Autonomic Management via Dynamic Combinations of Reusable Strategies

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

ABSTRACT

Autonomic Management capabilities become increasingly important for attaining functional and quality goals in software systems. Nonetheless, successful Autonomic Management solutions must feature complex, adaptive behaviors, which remain difficult to conceive and control. This paper proposes a generic approach and a reusable framework for the construction of Autonomic Manager applications. The presented solution advocates creating Autonomic Management behaviors via the dynamic and opportunistic integration of individual management strategies. The provided framework proposes a general architecture and a common infrastructure for supporting the presented approach. A sample Autonomic Manager was built using the framework with several management strategies and was successfully tested in an experimental scenario.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*; D.2.13 [**Reusable Software**]: Domain engineering, Reusable libraries, Reuse models

General Terms

Design, Management

Keywords

autonomic management, dynamic composition, problem solving modules, framework, service-oriented computing

1. INTRODUCTION

In the present IT domain, dependable autonomic solutions for managing complex computing systems become increasingly critical for achieving and maintaining business success. Nonetheless, building Autonomic Management (AM) systems remains rather difficult and costly an undertaking, as pointed by multiple academic and industrial parties in relevant domains. Initial efforts made in the Autonomic Computing field indicate that for successfully administering complex software systems AM solutions must likewise become complex software systems. This is a natural consequence of the very purpose that AM systems must serve, which is to absorb the complexity of currently manual management tasks and leave simplified, intuitive and high-level interfaces for human administrators. Consequently, Autonomic Computing will actually increase overall system complexity, at the gained advantage of hiding complexity from external users. Hence, the complexity problem is initially exacerbated for the developers of systems incorporating autonomic elements. However, once implemented, autonomic solutions will profit system administrators via the simplicity of use and the reusability of thoroughly understood and tested management functions.

Several AM prototypes have been recently implemented as part of different academic and industrial efforts, for handling various administrative aspects of different system types (e.g. Jasmine[1], Jade[11], Rainbow[4]). While these solutions provide useful and important insights into the workings of AM solutions, their construction remains mostly ad-hoc and application-specific. Consequently, these contributions offer limited reusable architectures and/or development methodologies for the Autonomic Computing community. For the time being, the actual construction of AM applications remains a difficult and costly task.

Numerous functionalities studied and implemented in current autonomic frameworks could be reused across multiple AM solutions. However, at present, common functionalities are being implemented repeatedly and individually for different autonomic systems. Such management-specific functions include hardware and software probes (e.g. CPU, or response times monitoring), analysis and planning policies (e.g. threshold detection, resource expansion or limitation) and effector capabilities (e.g. parameter modification, or server instantiation). Considering the functional complexity and dependability requirements of AM application, the provisioning of common, reusable solutions becomes essential for facilitating the development of such applications. Reusable AM functionalities, interaction patterns and partial implementations should be made available for utilization across multiple AM implementations. In addition, in order to meet the adaptability and extensibility requirements of autonomic solutions, such reusable modules should be easily configurable and replaceable over time.

This paper proposes a generic solution and a reusable

---

[1]Jasmine project: http://wiki.jasmine.objectweb.org

framework for the construction of Autonomic Manager applications. The adopted approach is based on: i) the partitioning of AM behaviors into elementary functions, or management strategies; and ii) the dynamic activation and deactivation of different combinations of management strategies, as needed to respond to various situations, management contexts and administrative challenges. The goal of the presented work is twofold. First, it proposes a modular approach to building AM solutions, based on dynamic and opportunistic combinations of reusable management strategies. Secondly, it provides a reusable framework that supports the presented approach. The framework consists of a generic architecture and an implementation of the reusable architectural parts. An initial framework prototype has been implemented to offer support for the integration and execution of management strategies, as well as for their seamless administration during runtime. The framework implementation is based on a Service Oriented Component technology. The modularity and loose-coupling properties of such technologies provide the necessary technical support for implementing the presented approach. The completed experimental work targeted the pervasive computing domain and more particularly, ubiquitous home applications. Nonetheless, the proposed contributions can generally be reused across multiple application domains and system types (e.g. enterprise, or grid systems).

2. REQUIREMENTS AND CHALLENGES OF AM APPLICATIONS

AM applications must provide complicated behaviors for successfully administering complex software systems. Specifically, administrative functions must correctly and efficiently detect and react to various changes in the managed applications and their execution environments. Multiple management goals must be taken into account and possibly conflicting adaptations must be resolved. These requirements lead to a significant space of possible contexts that require different administrative behaviors. Consequently, the specification of an Autonomic Manager's overall behavior becomes a cumbersome endeavor.

In addition, an Autonomic Manager's behavior must be dynamically adaptable, in order to keep pace with runtime modifications in the managed application, execution environment and given management goals. For this reason, Autonomic Manager implementations must equally be administered, in order to ensure their efficiency and correctness in conditions that continuously evolve over time. Some of the main adaptability requirements envisaged for AM applications include the addition and removal of autonomic utilities, as well as the dynamic organization and coordination of concurrent utilities. One possible example involves the dynamic installation, activation, deactivation and removal of monitoring probes, for maintaining suitable instrumentation coverage over fluctuating resources. A second example consists of maintaining analysis and planning policies in sync with changes that occur in the managed system, or in the administrative goals. The dynamic removal of certain management behaviors becomes necessary in case they prove ineffective, or no longer applicable to an updated system. An Autonomic Manager should inactivate its faulty or deprecated functions and replace them with suitable versions.

Another runtime requirement for AM solutions is the ability to reconfigure their existing functions and tune their administrative behaviors. Most autonomic functions can be continuously adjusted and optimized for better meeting their goals. Reconfigurations can help an AM solution reach an optimal behavior for managing an initially given system and environment. Additionally, it allows an AM solution adapt to continuous changes in its managed resources and execution environments. For example, various parameters used for the analysis and planning policies of an autonomic solution can be gradually adjusted, as progressive knowledge becomes available on the managed system, its surrounding environment and the efficiency of previously applied policies. Similarly, monitoring policies can dynamically be configured depending on current necessities, including the frequency and accuracy of extracted data, or the constraints imposed on resource consumption and induced delays.

Finally, AM applications must be able to detect and resolve management deadlocks, infinite control loops, oscillating states and inefficient analysis and/or planning processes. Most desirable characteristics of AM applications include the following.

Maintainability: allows Autonomic Managers to be partially or completely modifiable or replaceable, whether statically or during runtime. This is an important characteristic for removing erroneous functionalities, updating existing functions, or fixing securities issues.

Adaptability: allows Autonomic Managers to be context-aware. This allows managers to display different behaviors depending on their execution contexts and administrative goals. In this manner, an Autonomic Manager may employ accurate, resource-consuming strategies in a 'normal' context, while using strategies that trade preciseness for better efficiency in cases of emergency or limited resources.

Extensibility: enables introducing new AM capabilities without completely disrupting existing functions. For example, a managed application may evolve over time in unexpected ways (e.g. provide a new sensor). In this scenario, the Autonomic Manager should be easily extensible in order to take advantage of its new available capabilities without the need for completely rewriting, reloading, and rerunning the Autonomic Manager.

Dependability: represents the reliance that can be placed on the AM functions. This characteristic mainly results from the previous properties. Maintainability and adaptability allow administrators to correct faulty behaviors. Extensibility often leads to reusability of part of the code, reducing the chances of introducing possible errors.

3. PROPOSED SOLUTION

3.1 Management behavior development

Likewise the managed systems that they strive to administer, autonomic applications are also complex, large-scale, distributed systems featuring important dependability and adaptability requirements. To alleviate the difficulty of specifying complex management behaviors, the solution presented in this paper proposes to divide the overall administrative activities into elementary management functions, or *strategies* (Figure 1). It then promotes the formation of composite behaviors via the simultaneous activation and/or deactivation of different elementary strategies, during runtime. Examples of reusable management strategies include simple monitoring probes, threshold-detection analyzers, re-

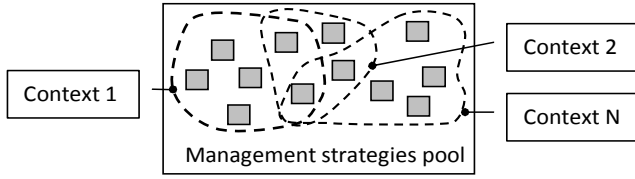active planning policies, or parameter-setting effectors.

Figure 1: Composing adaptable management behaviors from reusable strategies: different sets of management strategies are activated to handle diverse administrative contexts.

Multiple monitoring, analysis, planning, or execution strategies can be available for responding to various management requirements and contexts. The Autonomic Manager dynamically selects the strategies it needs to use from the available functional implementations, in order to deal with each given situation or challenge. In this approach, the occurrence of a certain situation (i.e. a problem to solve in a given context) maps to a certain set of strategies that are activated to handle that situation. Based on this principle, distinct sets of possibly overlapping strategies are active in different execution contexts, their combined effects generating various management behaviors. This paper proposes a framework that offers an extensible set of management strategies, as well as a reusable infrastructure common to all management solutions. The infrastructure provides the necessary support for integrating and interconnecting management strategies, coordinating their activities and organizing their produced results. This includes support for communication and data processing (e.g. aggregation and filtering).

Available strategies are categorized based on various criteria, expressed as strategy *attributes* and provided as strategy *descriptions*. Such strategy attributes include the managed resource type, functions performed, concerns addressed, applicable situation and predicted effects. A strategy description indicates the strategy's type, as well as the strategy's dependability, or approximate performance yielded. For the scope of the presented work, strategies were categorized based on the generic Autonomic Computing blueprint architecture[1] into monitoring, analysis, planning and effector strategies. In this manner, the respective functions of an Autonomic Manager are ensured via the activation of strategies from the corresponding type.

3.2 Management behavior adaptability

In order to meet the adaptability requirements of Autonomic Managers, this paper proposes a means of automatically changing management behaviors by dynamically modifying the sets of strategies that are to be activated in different situations. More precisely, the presented solution proposes the introduction of a higher-level management application that is in charge of administering the AM behavior at runtime.

The solution proposed in this paper introduces a layered architecture that implements the aforementioned concept. Specifically, a higher-level Autonomic Manager, called the *Manager Adaptation layer*, monitors and adapts the basic-level Autonomic Manager, called the *Resource Management layer*. To ensure that management behaviors remain within

a desirable space, administrative functions at the higher abstraction layer supervise and adjust the behavior of the Autonomic Manager at the lower layer. Supervising and adapting the AM functions helps prevent undesirable situations, such as deadlocks, or repetitive, inefficient and oscillating operations. This is achieved by monitoring management strategies and changing the active strategies when their behavior transcends acceptable boundaries. Monitored parameters may include the time taken by an active strategy to reach a useful conclusion, or the detected strategy activation patterns in the observed management procedures. The upper layer manager adapts the basic layer manager by adding, updating and removing management strategies as they become available or deprecated, respectively. Moreover, the higher manager reconfigures the lower layer modifying the sets of strategies to be activated in response to each situation. Reconfigurations are based on existing knowledge on the efficiency and success rates of the available strategies.

A possible adaptation example initially involves the activation of strategies capable of dealing with an emergency. As immediate action is required, these strategies will include: i) extensive monitoring probes for providing an accurate view of the system's state; ii) efficient analysis and planning policies; and iii) "aggressive" effectors prioritizing management decision implementations over system availability. Once the emergency is solved, a different set of strategies is activated to handle the routine context. As minimum maintenance activity is required, these strategies will include: i) a reduced number of monitoring probes for supervising key system points; ii) broad analysis and planning strategies considering multiple management concerns; and iii) "cautious" effector strategies that may wait for an optimal moment before altering the system. In addition, individual strategies can be configured for different contexts. For example, the frequency at which a monitoring probe extracts and provides runtime data can be increased during an emergency and reduced during normal system functioning.

3.3 Advantages

The presented approach allows for different behavioral aspects, dealing with different management concerns, to be separately implemented, managed and reused. The proposed design clearly separates management strategies from each other, as well as from the management logic deciding which strategies to activate in which context. The main advantages of the proposed solution consist in its inherent modularity and flexibility. These properties facilitate the development and maintenance of AM solutions and render such solutions self-adaptable. In contrast to building centralized, monolithic management strategies, this paper proposes a decentralized design, where complex management behaviors are created dynamically and opportunistically by simultaneously activating simpler strategies. Separating AM behaviors into finer-grained strategies that implement specific management functions enhances development productivity and reusability. In addition, it facilitates the dynamic evolution of autonomic behaviors, as their various parts can be seamlessly added, configured, removed, activated, or deactivated at runtime. This allows AM applications to adapt their overall behaviors to fluctuating goals, running conditions and managed resources. Finally, it allows new management behaviors be dynamically discovered and tested as various strategies become available during runtime.

4. PROPOSED FRAMEWORK

4.1 Overview

The second main contribution of this paper consists in proposing a reusable framework that implements the presented solution. The framework defines a generic architecture and implements the reusable architecture infrastructure. This supports for the creation of application-specific Autonomic Managers by integrating and managing pluggable strategies. The framework proposes a layered architecture, where each layer administers a lower-level layer, while being administered by a higher-level layer. Two layers were defined so far for the proposed framework architecture: a *Resource Management layer* and a *Manager Adaptation layer*. The general framework architecture is presented in figure 2.
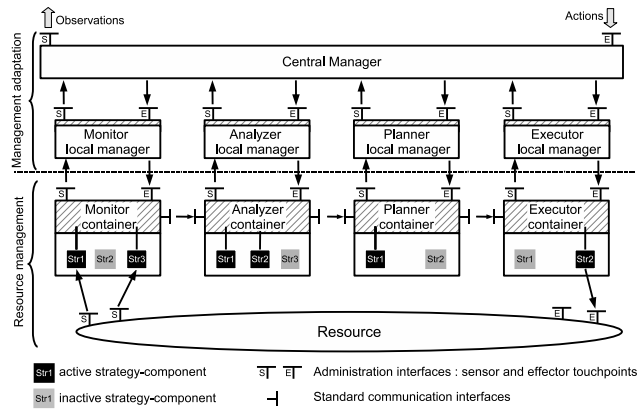

Figure 2: Layered framework architecture

Data is interchanged between the various framework entities in the form of *Reports* of different types. All Report types must comply with a general standard, which allows the framework's communication infrastructure to correctly identify the Reports and route them to their destination(s). Nonetheless, only the concerned parties have to understand the actual semantics of the Reports' contents.

The main roles, functions and relations between the two framework layers are described as follows. The *Resource Management layer* corresponds to the "classical" AM control loop, as it is commonly portrayed in the Autonomic Computing community [1][6]. This layer implements AM functions for administering the underlying system resources. The Resource Management layer is composed of two parts: a common container and a set of pluggable *strategy-components*. The common container is application-independent, while the strategy-components are application-specific.

The Resource Management *container* provides support for integrating strategy-components and ensuring their intercommunication and coordination at runtime. The container implements functions that are independent of the administrative goals and management techniques specific to each application. This aspect renders the container reusable AM solutions. At the same time, certain container parts can be configured to meet the specific requirements of each AM application. This includes strategy activation criteria and Report aggregation and filtering functions. The framework architecture uses strategy-components to encapsulate the management strategies of the presented solution. Strategy-

components implement the actual AM functions (e.g. monitoring, analysis, planning and effector functions). Strategy-components can be dynamically plugged into, or out of, the Resource Management container, as well as activated and deactivated depending on the current context and incoming Reports. All strategy-components of a certain type expose a standard interface that allows them to receive data Reports and return processed results (Figure 2). It is possible that multiple strategy components simultaneously receive the same data Reports for processing. Individual strategy-components are unaware of the execution of other strategies. The container collects results from all active strategies and processes them so as to obtain coherent output Reports.

Strategy-components can be categorized based on the management functions they perform. Therefore, the Resource Management layer was designed as a chain of interconnected mediation modules, where each module is responsible for a specific management function. The successive mediation modules of the Resource Manager layer are referred to as management steps. Each management step consists of a container instance and an associated set of management strategies (Figure 2). A widely accepted decomposition into management steps consists of a sequence of monitoring, analysis, planning and execution modules, which use a central knowledge base (i.e. the MAPE-K model[1]). This decomposition was selected as a starting point for the presented work. All strategy-components have access to a central knowledge base, which contains relevant management-related information (excluded from figure 2 for simplicity). The knowledge base's goal is to ensure that historical data is readily accessible and that information is preserved reliably over time.

The *Manager Adaptation layer* is responsible for controlling and administering the underlying Resource Management layer. It has a strong impact on the composition of the Resource Management behavior by specifying which strategy-components should be activated in various execution contexts. In addition, the Manager Adaptation layer can also modify container functionalities, such as the communication support, or the production of output Reports from multiple strategy results. The Management Adaptation architecture is further divided into two management levels: a local level - Local Manager Adaptation, for administrating individual management steps (e.g. monitoring, analysis, planning and execution), and a central level - Central Manager Adaptation, for administering the overall Resource Management layer. The central administration level ensures coordination between the local administration managers and provides support for communicating with external entities (e.g. other Autonomic Managers).

Possible divergences between conflicting strategies may be addressed at both the Resource Management and Manager Adaptation layers. A careful configuration of the Resource Manager container prevents the simultaneous activation of conflicting strategy-components. This option was adopted for the current framework prototype and presented experimental work. Additionally, a special-purpose mechanism can be implemented at the Resource Management container level to inhibit certain strategy-components when a conflicting strategy is activated. Finally, a data-mediation function can be employed at the container level to aggregate and/or filter conflicting results into a single coherent output Report. Alternatively, the Manager Adaptation layer can decide to map conflicting strategy-components to completely different

contexts, in order to avoid their concurrent activation.

4.2 Resource Management layer

The Resource Management layer was split into several management-steps, in order to separate the different concerns involved in the management process. Each management-step consists of a generic *management container* and a set of specific *strategy-components*. Consequently, the Resource Management layer consists of a linked set of management containers for monitoring, analysis, planning and executing functions, respectively (Figure 2). Accordingly, available monitoring, analysis, planning and executing strategies are assigned to the corresponding containers.

Information interchanged between management-steps is encapsulated into Reports of different types. With the exception of the first and the last management steps, each step transforms the set of incoming Reports produced by the previous step into a set of Reports forwarded to the subsequent step. Each Report contains meta-information such as a unique ID, a production date, a producer name and the *category* of the management step that produced the Report. In addition, a Report *type* is provided to indicate the sort of content available in the Report. A Report's content consists of a set of property-value pairs, depending on the management step that created the Report. Hence, monitoring Reports contain measurements extracted by probes and planning Reports contain solutions for detected problems.

The management *container* implements application-independent functions, including communication, strategy-component administration and Report processing. The main container roles and functionalities include enabling communication amongst active strategies of different types and directing incoming Reports to the active strategies that must process them, under each particular context. The container subsequently collects, aggregates and filters Reports from active strategies and sends the results to interested parties. In addition, the container provides high-level information on the functioning of the Resource Management layer to the Manager Administration layer. Finally, containers reconfigure strategy-components, based on indications received from human administrators and/or from the Manager Adaptation layer.

In order to meet these functional requirements, the container architecture provides four main subcomponents: *Receiver*, *Bridge*, *Data-Processing Chain* (DPC) and *Emitter* (Figure 3). All subcomponents provide touchpoints that allow the container (and indirectly the Manager Adaptation layer) to administer them. The container itself provides monitoring and effector touchpoints for its own management.
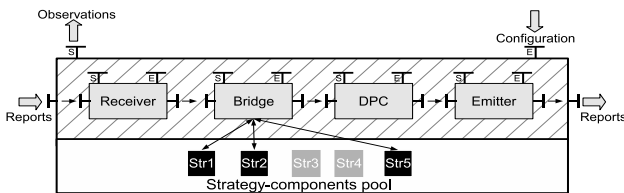

Figure 3: Manageable container

The container subcomponents and their interactions are defined as follows. Container Receivers and Emitters en-

sure the communication between management-steps. Each container holds both a Receiver and an Emitter, with the exception of the first and last management-step containers. The first step does not require a Receiver as information are gathered directly by monitoring strategy-components. Similarly, the last step does not use an Emitter as the actions are directly performed by the execution strategy-components.

The container Receiver forwards incoming Reports to the container Bridge subcomponent (Figure 3). The Bridge subcomponent is responsible for managing the container-specific *strategy-component pool*. Strategy-components implement various management strategies and expose standard interfaces that allow the Bridge to manage them. A strategy-component transforms an incoming set of Reports into an outgoing set of Reports. For example, an analyzer strategy expects to receive monitoring Reports, which it processes in order to produce analyzer Reports. The Bridge manages pluggable strategy-components and distributes incoming Reports to the relevant strategies. It uses a mapping table for determining which Report types to send to which strategy-components. The Local Manager Adaptation layer can alter the Bridge mapping table via the container's administration interface. In turn, the Bridge collects information on the active strategy results and forwards high-level Reports to the Local Manager Administration component. These Reports include information on the active strategies' success rates and computation times, allowing to determine the strategies' performance and dependability parameters in various contexts.

The Bridge collects the Reports produced by active strategy-components and transmits them to the *Data Processing-Chain* (DPC) subcomponent (Figure 4). The DPC subcomponent consists of a flexible set of linked Data-Processing Elements (DPE), including data *aggregators* and *filters*.
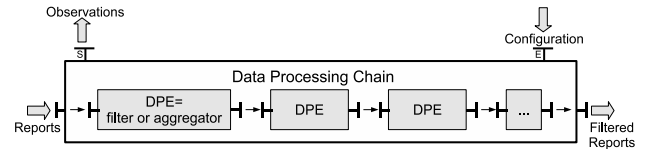

Figure 4: Data Processing Chain subcomponent

Generally, the DPC becomes necessary when different strategies are activated concurrently to process the same set of incoming Reports. The simultaneous use of multiple strategies may result in an inconsistent Report set, providing contradicting or missing information. The goal of the DPC is to eliminate unwanted data or possible contradictions and/or to complete the produced Reports. The DPC arbitrates amongst the solutions proposed by several active strategies and selects the most adapted ones for forwarding to the next management step. For instance, considering the case where two planning strategies are activated simultaneously to provide a solution for a given problem. The DPC must process the individual solutions proposed by the two strategies and determine a unique, coherent plan. The DPC may simply select one of the proposed solutions (e.g. the fastest one), or merge the two solutions into a single one, if no conflicting actions are obtained in result. The DPC forwards the resulting Report set to the Emitter subcomponent, which sends it to the subsequent management step in the chain.

4.3 Manager Adaptation layer

The Manager Adaptation layer ensures the correct functioning of the Resource Management layer and adapts its characteristics to changing execution contexts. The main role of the Manager Adaptation layer is to observe the activities of the various strategies active in the Resource Management layer and to modify their associations to various contexts. The Manager Adaptation layer is not concerned with the implementation details of each strategy. It is only aware of the strategies' types and general attributes and keeps a history of the strategies' performances when previously activated in different contexts. In this manner, the Manager Adaptation layer is able to detect and correct inefficient strategies, management conflicts, deadlocks, oscillating states and infinite control loops. Initially, system administrators manually specify mappings between contexts and the set of strategy-components used in those contexts. The Manager Adaptation layer can also set in place these initial mappings, based on the strategy-components' types and advertised attributes. During runtime, the Manager Adaptation layer reconfigures the Resource Management containers in order to update their mapping tables and modify the strategies used in each context. Reconfiguration decisions are based on the strategies' descriptions and previous results, as well as on the system state and management goals.

4.3.1 Local Manager Adaptation

The Local Manager Adaptation layer administers the Resource Management layer at the local, concern-specific level. In the presented framework, one Local Manager Adaptation instance is present for each available management-step (Figure 2) (i.e. monitoring, analysis, planning and execution managers). Local managers collect information on active strategy-components and dynamically determine viable container implementations for the current context. The goal is to maintain a correct, dependable and possibly optimal behavior for each management step, in conditions of varying execution contexts and management goals. Lack of local management adaptation would result in the continuous adoption of a default administrative behavior, predefined at the AM system's startup.

Adapting a management-step's behavior is achieved by configuring the Bridge subcomponent of the corresponding container. Specifically, the local manager adjusts the Bridge's mapping specification between Report types and active strategy-components. For this purpose, local managers require functional and non-functional information on the available strategies. This information is initially provided by strategy-component descriptions and progressively completed with runtime information. Local managers complete the strategy-components' profiles by merely observing their applicability for performing a certain task and without necessarily 'understanding' how the strategy operates. The current framework prototype uses static strategy descriptions and does not yet update them with runtime information. A local manager can also reconfigure the DPC container subcomponent, influencing the manner in which it aggregates and filters conflicting Reports. The DPC configuration is tightly-coupled with the Bridge configuration. Finally, the local manager is responsible for the dynamic discovery of strategy-components. Support for this capability mainly depends on the underlying framework technologies. A possible approach involves the availability of a central registry providing strategy descriptions for the existing strategy-components. This solution was retained for the current framework implementation.

4.3.2 Central Manager Adaptation

The Central Manager Adaptation component globally coordinates the individual actions of Local Manager Adaptation components. Exclusively localized adaptation without coordination may prove insufficient to attain an Autonomic Manager's high-level goals. For this purpose, the Central Manager Adaptation component performs all administrative actions that require a global view of the Resource Management layer. In addition, the central manager mediates communication between local managers, as well as with external entities, such as other Autonomic Managers and system administrators.

The central manager receives high-level administrative goals from system managers and/or other Autonomic Managers and accordingly configures each local manager in order to attain these goals. High-level goals may concern the managed system, as well as the functioning of the Autonomic Manager itself. Such administrative goals may include the Autonomic Manager's reaction times and resource consumption. In this case, the central manager observes the behavior of local managers and takes appropriate action in order to regulate their overall resource consumption. In addition, the central manager must ensure that the choice of strategies activated in the different management steps results in an overall coherent autonomic behavior. It must also verify that the Report types circulated between successive management steps are understood mutually by the strategies activated in those steps.

Another important role of the central manager is to disseminate and filter information among the local managers. For example, an 'alert' signal can be distributed to all local managers indicating the occurrence of an emergency. Consequently, all local managers update the strategy mappings in the associated containers, activating those strategies that are most suitable for handling the emergency. This ensures a coherent, uniform reconfiguration of the entire Resource Management layer.

4.4 Meeting AM Requirements

Table 1 summarizes the mapping between discussed properties in section 2 and how the framework addresses them.

Properties	Addressed by framework by way of
Maintenability	loosely coupled, dynamically updatable strategies with separated concerns; independently updatable, reusable and application-independent framework components; Standard administration interfaces.
Adaptability	Dynamically adaptable set of activated strategies; replaceable framework components; changeable communication between components.
Extensibility	extendable strategy pool via discoverable strategies
Dependability	reusable and replaceable(to fix errors) strategies and framework components

Table 1: Mapping between desirable characteristics and proposed solution

5. SOLUTION IMPLEMENTATION

5.1 Service-Oriented Computing technology

A framework prototype was implemented and tested for managing a sample application. A Service-Oriented Computing (SOC) approach was selected for the framework implementation, as it largely facilitated the development of pluggable strategies. This approach uses services as first-class elements for building software applications. It advertises modularity and loose coupling as its core principles. Services can be supplied by multiple service providers, featuring various implementations, and are discovered and bound at runtime via a discovery mechanism. These characteristics are particularly interesting when implementing pluggable strategies as services. Based on this approach, local managers in the proposed framework have the possibility to dynamically discover strategy services and use them at runtime. This facility gives local managers the means of evolving their administrative behaviors in order to handle unforeseen situations and challenges.

The prototype was developed based on the OSGi[2] service platform and more precisely on the iPOJO[3][3] service component runtime extending the OSGi platform. The OSGi framework is a well-known service execution platform, which supports dynamic reconfiguration and provides a service runtime environment. iPOJO simplifies service development by providing a *component container* that is responsible for managing all SOC-related aspects, from service publication to service dependency management. Moreover, iPOJO offers, via the use of aspect-oriented manipulation, the possibility of adding non-functional behavior to deployed services(e.g. logging, administration). This ability was used when implementing the presented framework, for reducing the complexity of strategies implementations. Finally, iPOJO provides an Architecture Definition Language(ADL) for describing service-based applications.

5.2 Framework implementation

The proposed framework offers a set of reusable libraries and services for building Autonomic Managers. The different architectural modules are implemented as separate iPOJO services, which are replaceable during runtime. In iPOJO, services are created using instance factories. Factories are specialized services that facilitate component instantiation. The current prototype provides a default instance factory for each architectural component, from the central manager to the management step containers.

In the current framework implementation, a system administrator must explicitly instantiate the management strategies used. Future extensions will allow local managers to automatically discover factories (based on their descriptions) and obtain strategy instances as necessary. The current framework instantiation procedure involves the hierarchical instantiation of the framework components, starting with the central manager and ending with the management strategies. Each parent component is responsible for the instantiation and maintenance of their direct children. The first service to be created is the central manager. It creates the different local managers in conformance with a given description. Each local manager instantiates the management container they will administer and each container creates its internal subcomponents and initial strategies.

The Resource Management layer was fully implemented and tested. The default Receivers and Emitters ensuring interactions between management steps are based on OSGi platform's support for service communication. The container-level mappings between Report types and management strategies are described via regular expression-based filters. The DPC component was developed to meet the requirements of the initial experimental scenarios. Several management strategies were provided for each management step, as necessary for the administration scenarios tested.

An important part of the Manager Adaptation layer was implemented and tested. Local managers were conceived following the MAPE-K model. Their implementation consists of two parts. First, a generic part manages interactions with the central manager and with the underlying container. It also provides an administration interface (i.e. sensor and effector touchpoints) for the central manager. Second, local managers contain an application-specific part, consisting of a *statistics* service, an *anomaly detection* service and a *strategy selection* service. These services respectively correspond to the knowledge base, analyzer, and planner in the MAPE-K approach. This division separates concerns and increases local managers' maintenance and reusability. A central service registry enables the discovery of strategies by local managers. The registry is provided by the OSGi service platform and contains information such as service implemented algorithms and specific parameters. An initial implementation was completed for the Central Manager Adaptation component and its functionality will be tested in future example scenarios.

5.3 Autonomic Manager description

The proposed framework offers the possibility of specifying Autonomic Managers using XML files. The goal is to describe the Autonomic Manager in a technology-independent way, in order to facilitate its specification and improve its portability. Although iPOJO does provide a general ADL, its use for specifying complicated applications with specific requirements can quickly become cumbersome. The important number of services used renders the production of a complete application description in iPOJO ADL too verbose and laborious to verify. Therefore, a specific ADL was defined to describe instances of the presented architecture, appreciably reducing verbosity for the prototype's specification. The XML description of a framework instance is divided into three parts: a factory description, an instance description and an autonomic manager description. The factory and the instance description parts are largely inspired by the iPOJO ADL. The factory description includes factory specifications for the DPEs and strategy-components. Descriptions of strategy factories provide information on the strategy implementation classes and default parameters. The instance description part contains the description of service instances, including the instance name and properties values. The autonomic management part contains specific information about the autonomic manager, such as the factories to use for each component, as well as the local and central-managers descriptions. The specific XML description is translated into the iPOJO description language at compile time, by means of an XSL[4] transformation.

---

[2]OSGi Alliance: www.osgi.org
[3]iPOJO project: felix.apache.org/site/ipojo.html

[4]eXtensible Stylesheet Language: www.w3.org/TR/xsl/

6. EXPERIMENTS AND RESULTS

6.1 Testing scenario

The presented framework prototype was tested in an experimental scenario pertaining to the field of home-automation. Home-automation aims at providing easy-to-use automatic applications for non-expert users. The current framework implementation was employed to manage a common home application: a video camera-based intrusion detector and elderly person supervision. The application consisted of four services: a simulated camera driver, a database, an intrusion detector, and a household accident detector. For simplicity, this application can be generalized to a classical producer-consumer scenario that uses an intermediate buffer for data mediation. In this scenario, the video camera service is considered as the producer, the database service as the intermediate buffer and the intrusion detection and person supervision services as different consumers.

Figure 5 shows the generalized application with one producer and two consumers and depicts the Autonomic Managers employed to administer each service. Autonomic Manager instances communicate via predefined events, disseminated through a common communication channel. The producer stores generated data into the database, along with meta-information such as production dates and used configurations. The consumers retrieve data from the database as required for achieving their functional goals. The role of the database is to store information reliably, until all consumers manage to recuperate and process it.
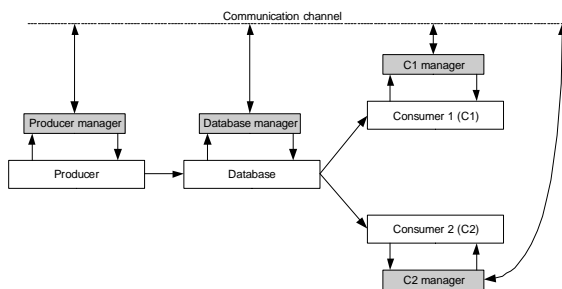

Figure 5: Sample application

The producer offers an administration facility for tuning its data output frequency. Considering the concrete intrusion-detection scenario, this facility is used to set the transmission rate of the video camera, increasing it for example when suspect activities are detected. Consumers operate independently, recuperating and processing data from the database at different frequencies. Consumer frequencies can change over time, depending on execution contexts. Data are maintained in the database until acquired by all registered consumers. This may eventually cause database saturation in cases where the producer's frequency exceeds the retrieval frequency of the slowest consumer. Special management action is required to avoid loss of information when the database reaches its full capacity. For that reason, one Autonomic Manager instance was associated with each application service and was given specific administrative goals with respect to that service. This paper focuses on the database Autonomic Manager and its corresponding strategies.

High-level management goals were statically set for each Autonomic Manager in the application. All Autonomic Managers were given the goal of maintaining memory consumption under a given threshold, on the shared execution platform. Apart from this common objective, each manager has specific administrative goals. The producer manager is in charge of adjusting the producer's output frequency, based on requests from the database and consumer managers. Similarly, consumer managers are in charge of determining the best data acquisition frequencies for the actual consumers they administer. Consumer managers may request producer managers to increase capture frequencies, when consumers require more data in order to work properly. The producer manager considers multiple requests from peer consumer managers and tries to reach an optimal compromise for adjusting the producer's frequency.

The database manager administers the memory resource allocated to the database. Its goal is to reliably preserve relevant data and to ensure that the database capacity does not exceed the system's memory limits. The database service was implemented as a data buffer and allocated a certain number of data units that simulate memory consumption. The database manager employs different management strategies at each management step, by activating the corresponding strategy-components available. Implemented strategy-components include several monitoring strategies for collecting system properties (e.g. available system memory) and database state (e.g. current database size and capacity). The database capacity represents the maximum amount of data that can be stored in the database. The database size indicates the current amount of data that is stored in the database. In the current implementation, monitoring strategies use JMX[5] touchpoints for collecting information from the managed resources.

Two analysis strategies were implemented for analyzing monitoring Reports on the database's size and capacity. One strategy detects when the database size exceeds 90% of the database's capacity, or when it goes below 60% of the database's capacity. When either of the two cases is detected, this strategy issues a database 'overflow' or 'underutilization' event, respectively. The second analysis strategy compares the database's capacity with the maximum system memory that can be allocated to the database (i.e. 25% of the memory). The analysis step activates both of these analysis strategies each time it receives a monitoring Report.

Three planning strategies were implemented and tested for dealing with database saturation. The first strategy uses a "smart data deletion" algorithm for selectively removing existing data from the database and limiting memory consumption. The idea is to drop irrelevant data based on provided application-specific heuristics (e.g. deleting successive, similar images). The second planning strategy increases the database capacity provided this does not exceed the system memory limits (i.e. "capacity increase" strategy). Rather than modifying the database, the third strategy tries to solve the problem by demanding the producer to decrease its data output frequency. More precisely, this strategy sends a `"too_high_frequency"` event to the producer's manager, via the communication channel. The producer manager subsequently reduces its output frequency, provided that none of the consumers requires a high fre-

---

[5]Java Management Extensions: http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/
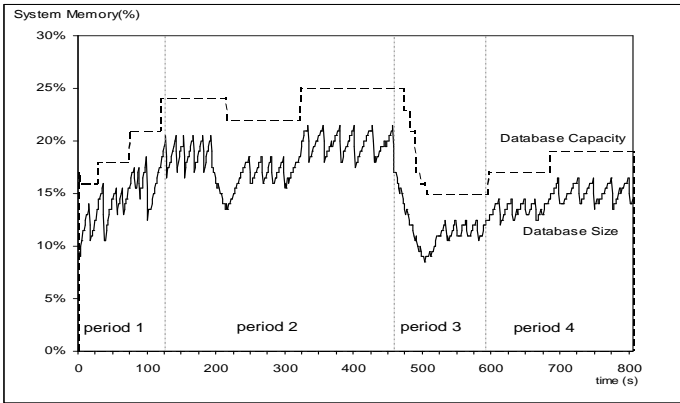
Figure 6: database management

quency. Otherwise, it ignores the `"too_high_frequency"` message received. While the three planning strategies presented handle database saturation situations, a fourth planning strategy was implemented to handle database underutilization. When receiving an analysis Report indicating capacity underutilization this planning strategy decrements the available system memory allocated to the database.

The local planning manager uses its statistics service to log information on the use of each planning strategy. Based on these statistics, the local manager detects when an active strategy repeatedly fails to meet its goals over a determined period. When this occurs, the local manager considers the strategy as faulty, or inefficient, and uses its selection algorithm to choose another strategy to activate. The planning selection algorithm specifies an ordered list of available strategies and the number of times each strategy must prove unsuccessful before selecting the next strategy in the list. Each planning strategy contains an ordered list of actions, which execution strategies can interpret and implement.

6.2 Experimental results

The graph in Figure 6 illustrates the manner in which the presented management strategies are employed and the effects they have on the database size and capacity. The graph shows the evolution in time of the database capacity (dotted line) and the database size (solid line). The goal of the database Autonomic Manager is to keep the database capacity at a minimum and to make sure it does not exceed 25% of total system memory. The management scenario depicted in the graph began with an augmentation in the database size (i.e. the initial part of 'period1'). This was induced by an increase in the producer's data output frequency, due for example to a request from one of the consumers. When the database size exceeded 90% of the database capacity, the analysis management step signaled a database saturation situation. The database-planning step consequently employed the available planning strategies for handling the problem. The graph in Figure 6 shows the results of alternating the activation of available planning strategies, as dictated by the local planning manager.

The first planning strategy the database manager employs is the "selective data deletion" strategy. However, this strategy can only provide a temporary solution and hence the database becomes repeatedly saturated. The repeated use of this strategy causes the database size fluctuations observable during 'period 1', as well as over all subsequent periods. As a result, the local database manager detected the frequent use of this strategy and decided to additionally activate the "capacity increase" strategy for solving the problem. The results of activating this second strategy appears several times in the graph, as the capacity curve increases in steps (i.e. three times during 'period 1', twice during 'period 2' and 'period 4').

At the beginning of 'period 2', the database capacity reached its maximum memory allocation allowed. At this point, the "selective data deletion" strategy, still active, was the only one preventing the database from overflowing. The "capacity increase" strategy could no longer succeed since the database maximum capacity had already been reached. The local database manager detected that this strategy failed to complete and decided to activate the third strategy. This strategy asked the producer to reduce its data production frequency, by sending a `"too_high_frequency"` event on the communication bus. The consequence of applying this strategy can be observed in the graph as the database size curve descends during 'period 2'. In the presented experimental setting, the producer was implemented to lower its frequency more and more each time it receives a demand over a certain period. This explains why the database size decreased more when the third strategy was applied the second time, at the beginning of 'period 3'. When the database usage dropped, the database manager consequently detected the database underuse and lowered the database capacity. For experimental purposes, the producer was implemented to automatically re-increase its transmission frequency, over time, in order to periodically induce the activation of managers.

The presented experimental results show that the use of different basic strategies for building complicated administrative behaviors is an efficient way to develop Autonomic Managers. Specifically, strategies proved to be easy to implement, since their scopes were reduced to limited concerns. The use of decoupled strategies offered the opportunity for their seamless combination into different behaviors. Nonetheless, the provided flexibility and extensibility is counterbalanced by possible conflicts that might occur between concurrently active strategies. Part of the complexity of strategy coordination and conflict handling is delegate to local managers. This design provides a better separation of concerns than if the strategies and the logic deciding which strategies to employ were mixed in a monolithic manager.

7. RELATED WORK

The solution proposed in this paper presents strong similarities with the Blackboard Architecture [2] and consequently features similar benefits and difficulties. In a blackboard system, a set of problem solving modules (i.e. knowledge sources) share a common database (i.e. blackboard) and collaborate in an opportunistic and dynamic manner to solve a given problem. BB1[5] extends this general approach introducing a layered architecture for modifying the actual control logic of the solving modules.

[7] introduces self-management capabilities at the application architectural level. It proposes a three-layered architectural model for handling component reconfigurations, local adaptation behavior management and global goal management. This is similar to the layered architecture proposed in this paper, while being more theoretical and remaining at a higher abstraction level. The ACCORD framework [9] aims at facilitating the development of autonomic managers

from reusable autonomic elements, which can be replaceable during runtime based on dynamically injected rules. Similarly, [12] proposes a further application of architectural approaches to autonomic computing. It defines the interfaces, communication and behavioral requirements of application components to enable the flexible composition of autonomic managers from reusable components.

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 [8]. In this context, [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.

Several research projects are currently developing general architectures, engineering principles and runtime platforms, for providing reusable autonomic capabilities to large-scale, distributed software systems (e.g. IBM Autonomic Computing Toolkit[6] , Autonomia[7], AutoMate[8] , BioNets [9], Amorphous Computing[10] , Autonomic Networked Systems[11] and Recovery-Oriented Computing [12]). These projects propose autonomic functions and infrastructures that are complementary to the work presented in this paper. Therefore, the capabilities of existing platforms will be studied and considered for extending the presented autonomic framework.

8. CONCLUSIONS AND FUTURE WORK

This paper proposed a solution for developing complex and adaptable Autonomic Managers via the dynamic and opportunistic integration of management strategies. In addition, the paper presented a reusable framework that implemented this solution. A service-oriented prototype was implemented and successfully tested for managing a sample application.

Two main aspects must be considered when building AM systems. The first aspect is concerned with the construction of individual Autonomic Managers responsible for the administration of a certain resource (e.g. a machine's CPU, a software component, an entire application, or platform). The second aspect is concerned with the organization and coordination of multiple AM instances, as necessary for meeting the business-level goals of an overall system. This paper focused on the development of individual AM solutions. However, the authors do recognize the importance of considering both presented aspects simultaneously. Indeed, building a single Autonomic Manager in isolation provides limited guarantees on the successful extensibility of this solution for supporting Autonomic Manager collaborations. Therefore, although outside the scope of this paper, the interconnection and communication amongst multiple managers was taken into account into the presented solution.

While the proposed framework enables the creation of complex AM solutions with extensive adaptability capabilities, it equally allows for the construction of lighter AM solutions when simple, efficient solutions are needed. Indeed, the framework can be partially instantiated to only provide the basic Resource Management functions, hence reducing the AM solution's runtime complexity and resource consumption. Local managers and/or the central manager can be progressively added depending on the level of adaptability required of the AM behaviour. The gained flexibility, resilience and efficiency of the proposed solution come at the cost of losing some of the absolute control on the application's runtime behaviour. This is because the combined results of independent strategies may become difficult to predict and debug. Further work is necessary for coordinating the activities of strategies and aggregating potentially conflicting results into coherent management actions.

9. REFERENCES

[1] An architectural blueprint for autonomic computing. Technical report, IBM, June 2005.

[2] B. Draper, R. Collins, J. Brolio, A. Hanson, and E. Riseman. Issues in the development of a blackboard-based schema system for image understanding. In R. S. Engelmore and A. J. Morgan, editors, *Blackboard Systems*. Addison Wesley, 1988.

[3] C. Escoffier and R. Hall. Dynamically Adaptable Applications with iPOJO Service Components. In *SC2007*, March 2007.

[4] 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.

[5] B. Hayes-Roth. Bb1: an architecture for blackboard systems that control, explain, and learn about their own behavior. 1984.

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

[7] Kramer and Magee. Self-managed systems: an architectural challenge. *fose*, 00:259–268, 2007.

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

[9] H. Liu and M. Parashar. Accord: a programming framework for autonomic applications. 36(3):341–352, May 2006.

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

[11] 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.

[12] S. R. White, J. E. Hanson, I. Whalley, D. M. Chess, and J. O. Kephart. An architectural approach to autonomic computing. *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 2–9.

---

[6]Autonomic Computing Toolkit (IBM developerworks): 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 (Berkley and Stanford Universities): roc.cs.berkeley.edu