DRAAS: Dynamically Reconfigurable Architecture for Autonomic Services

Emna Mezghani, Riadh Ben Halima, Khalil Drira

DRAAS: Dynamically Reconfigurable Architecture for Autonomic Services

Emna Mezghani[1,2,3], Riadh Ben Halima[1,2,3] and Khalil Drira[1,2]
[1] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
[2] Univ de Toulouse, LAAS, F-31400 Toulouse, France
[3] University of Sfax, ReDCAD, B.P.W, 3038 Sfax, Tunisia
{emna.mezghani,khalil.drira}@laas.fr, riadh.benhalima@enis.rnu.tn

Abstract. The development and the provisioning of autonomic networked services are essential for enterprises and factories of the future. Endowing services with autonomic properties allows one to maintain at runtime the Quality of Service (QoS) including different parameters related to performance, availability and reputation such as response time and successful execution rate. Handling the autonomic properties requires the ability to deal with permanent requirement evolving and constraint changes. For instance, managing QoS degradation requires the capacity of identifying its possible or actual sources and the capacity of reconfiguration planning and execution. Dealing with these issues is especially challenging for web services since the autonomic solution has to be seamless for the service requesters, ensuring that Web Services are always usable under the different deployment constraints. To implement such autonomic systems, the literature provides different approaches, varying from the design to the full implementation of autonomic primitives. In this chapter, we present DRAAS: a Dynamically Reconfigurable Architecture for Autonomic Services able to provide autonomic properties for QoS management in web service-based distributed applications. DRAAS has been implemented and experimented successfully with different use cases. It covers the whole cycle of autonomic management including monitoring and analysis of QoS parameters , planning and execution.

1 Introduction

The important data flows, the frequent interactivity, the increasing number of connected devices, and the network unpredictability make critical the management of the new distributed software systems. In one hand, although the reform of verification and validation of software models hasn't ceased improving, components of the systems may still hide design faults resulting in system failures or come across deadlock that freezes the system. In the other hand, user′s requirements are evolving following the end-user technologies evolution as mobile phone emergence (Multimedia mobile and group-enabled application). In the same time, systems constraints are variable as unstable bandwidth and decreasing energy. As a result, autonomic computing paradigm is crucial for current systems in order to ensure QoS-aware execution.

More specifically, self-control systems such as elevator control systems or critical systems such as spacecraft navigational systems need robustness to detect anomalies and avoid them by reconfiguring the systems at runtime [1]. For these reasons, research actors are accelerating the work on autonomic systems. Such systems are capable to detect the problems and continue to operate by managing malfunctions without human intervention. Autonomic computing technology does not only reduce potential catastrophic errors, in critical systems for example, but it also minimizes the human intervention. It is applied when reliability and QoS are required. An autonomic system inspects and changes its own architecture and behavior when the evaluation indicates that the intended QoS is not achieved, or when a better functionality or performance is required.

The autonomic computing architecture is composed of four modules and a knowledge component that constitute a control loop namely MAPE-K [2]. *Monitoring* which monitors the data exchanged between the managed elements, *Analysis* which detects possible QoS or performance degradations, *Planning* that implements algorithms for selecting and scheduling appropriate elementary reconfiguration actions and *Execution* which performs them.

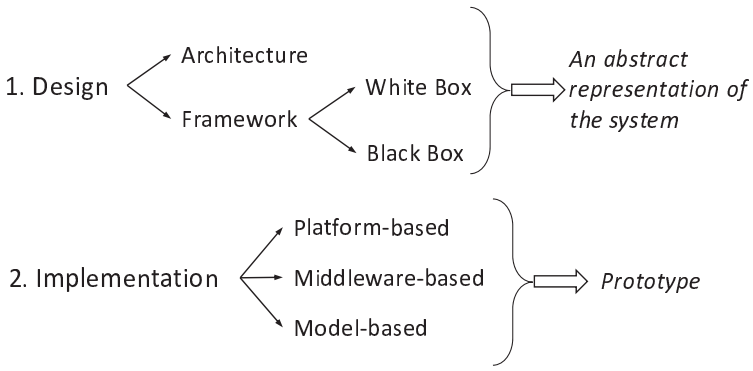

Fig. 1. Autonomic computing scope

This autonomic computing paradigm includes the design and implementation of computer systems, as shown in Figure 1. The first step focuses on establishing a detailed design from which results a framework or an architecture. Frameworks present the skeleton of an application that can be customized by the developer. We distinguish two types of framework [3, 4]: black box that does not need a deep understanding of the framework's implementation in order to use it and white box that requires the internal understanding of the framework to use it effectively. Architectures provide high-level abstraction of system components, while enabling easier understanding and interpretation. Furthermore, the architectural approach constraints can be expressed explicitly. The second

step concentrates on implementing the architecture or the framework through various techniques. The implementation may be classified into three categories: model-based, middleware-based and platform-based (see Figure 1). Model-based solutions [5–7] provide explicit implementation of all necessary actions for monitoring, analysis, planning and execution. In this category (model-based), we focus particularly on architectural approaches. Monitoring and analysis are made by testing if the running system conforms to a given architectural style or model. Middlewares, like Bionet [8], AgFlow [9], and OpenORB [10], support dynamic reconfiguration process by offering primitives (like interception) for all autonomic computing modules. Platforms [11–13] provide developers with already developed autonomic entities.

In this paper, we evaluate and classify a set of autonomic solutions according to criteria such as provided functionalities, managed autonomic steps, applied techniques, programming languages, etc. We aim to provide features which help and guide users to select a suitable solution for implementing autonomic services. However, it is usually difficult to select the appropriate approach to implement an application. We think that this choice depends on the size of the problem to solve, the architecture type (decentralized or centralized), the programming language, the application area (Server or Client, etc.), etc. Then, we propose our DRAAS architecture implementing the autonomic computing to ensure the dynamic reconfiguration of web service-based applications.

This paper is organized as follows. Section 2 gives an overview of the autonomic computing background. Section 3 details different ways to design autonomic system. Section 4 describes a taxonomy of dynamic reconfiguration implementation approaches focusing on the "model", "middleware" and "platform" categories. Section 5 presents our DRAAS architecture illustrated by a use case. Finally, section 6 concludes this paper and presents our future works.

2 Autonomic Computing Background

Autonomic computing constitutes an active research area in computer systems [14]. This paradigm, inspired from the human autonomic nervous system [2], has a mechanism that can trigger changes in the computing system structures and behaviors in order to bypass or correct them. Furthermore, it is a collection of autonomic components that the overarching goal is to manage themselves, so that systems will be dynamically reconfigured at run time, with minimum human intervention.

2.1 Self-* Capabilities

The principles that govern all autonomic computing systems, according to IBM, have been summarized in eight aspects [15]:

- Self-Configuring: the ability to dynamically configure components following high-policies in order to adjust the system. Such configurations could include

the deployment, the installation of new components or the removal of existing ones [16–18].
- Self-Healing: the ability of the system to perceive if it doesn't work correctly. It ensures the necessary adjustments to restitute it towards its normal state without human intervention [19]. By knowing about the system, it analyzes information, detects system degradations and initiates corrective actions without disrupting the system execution.
- Self-Optimizing: the ability of the system to continually enhance its performance. It is a proactive mechanism that detects performance degradation and acts intelligently such as in reallocating resources with minimal human intervention [16].
- Self-Protecting: the ability of the system to detect and protect its resources from internal and external attacks and maintain its security [2].
- Self-Awareness: the ability of the system to know itself and to be aware of its state and behaviors [15].
- Context Awareness: the ability of the system to know its execution environment and be able to react to its changes [15].
- Open: the ability of the system to work in a heterogeneous world and implement open standards. It should be portable across multiply hardware and software architectures [15].
- Anticipatory: the ability of the system to anticipate its needs and behaviors and to manage itself proactively [15].

2.2 Autonomic Computing Techniques

Autonomic computing is based on four main steps [2]: *Monitoring*, *Analysis*, *Plan* and *Execution*.

Monitoring techniques	**Analysis techniques**	**Plan&Execution techniques**
Interception [20, 21]	Architectural differentiation [22, 23]	Substitution[24, 10]
Assertion [25, 26]	Behavioral differentiation[27]	Wrapping[28, 29]
AOP [30–32]	QoS-contract (SLA)[33]	Load Balancing[34]
Reflection [10, 35]	QoS-aware (QoS historic)[36, 37]	Rollback [38]
		Redundancy & Duplication [39–41]

Table 1. MAPE-k loop techniques.

Monitoring is usually defined as the act of listening, carrying out supervision on, and/or recording the activity of a software entity for the purpose of maintaining system reliability and QoS. Monitoring can be ensured using the following techniques as listed in table 1:

- *Interception* which represents a hook into exchanged data between a client and a server allowing requests/responses supervision.
- *Assertion* is a set of code lines, introduced in a program, which enables to control and to constrain a program.
- *AOP (Aspect Oriented Programming)* which aims to verify system properties and also to configure scope/constraints of each function and discover even a tiny abnormal state.
- *Reflection* which enables to discover and to operate on fields and methods of an object at runtime.

Analysis is the process of detecting possible degradation of the system through the evaluation and the examination of monitored data. Analysis compares current system behavior and architecture with a reference model. The following techniques are used by the *Analysis* (see table 1):

- *Architectural Differentiation* refers to compare the obtained architectural model to the architectural style of the system in order to detect non-compliance.
- *Behavioral Differentiation* refers to map the behavior of an implementation to model behavior.
- *QoS Contract* which represents explicitly the system requirements under contract between clients and providers.
- *QoS Aware* which is based on the historic of the system state. It compares the current state with previous system states.

Plan&Execution are complementary. In fact, the plan presents a set of algorithms which refer to concrete reconfiguration actions enforced in the Execution module. While, the Execution refers to the act/the process of repairing or the condition of being repaired. Also, it may be defined as changes applied to a software entity so that it reaches a desirable state. In distributed systems, several techniques are used to achieve the repair process (see table 1):

- *Substitution* which replaces a system component by another.
- *Wrapping* which substitutes a system component by another enveloped which presents the same business logic.
- *Load Balancing* consists on distributing load on available components.
- *Rollback* allows the system to come back to the last stable state.
- *Redundancy* which repeats an action more than one time in order to achieve it.
- *Duplication (replication)* which involves addition of components representing similar functionalities.

3 Autonomic Design and Implementation Approaches

Software design is defined in IEEE610.12-90 as both "the process of defining the architecture, components, interfaces, and other characteristics of a system" and "the result of that process" [42]. It is the only way allowing an accurate

translation of requirements into a finished system. The design of architectures and frameworks is derived from the system specification.

From the implementation point of view, three main categories of solutions could be used to implement autonomic systems: the model-based category, the middleware-based category and the platform-based category. For the model-based implementations [7, 50], developers start from the scratch and should implement all actions related to autonomic computing modules. For the middleware-based implementations [8, 9, 51], developers build their solutions by adapting basic primitives to their application context. The provided APIs include primitives for monitoring, analysis, planning and possible reconfiguration actions. The platform category provides reusable components to implement autonomic computing strategies [11–13].

In the following, we present in detail the different approaches related to the design and the three categories implementing the autonomic computing. We also study the different factors that can guide to choose a specified approach rather than another.

3.1 Architecture design

The architecture models the structure of the system. It provides the global perspectives and a high-level abstraction enabling easier understanding of systems.

The work in [43] defines the architectural design as the high hierarchical structure of a system. It describes the overall design of the system that includes global control structure, communication protocol, data access, system components and their behaviors.

The authors of [44] define a derivative of the architecture, namely *dynamic architecture*. This type of architecture evolves and changes itself during runtime as the system changes. It aims to design an autonomic system in order to ensure the dynamic reconfiguration of system.

3.2 Framework design

Work in [45] defines a framework as a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact. The same author describes the framework in [46] as a combination of components and design pattern. CEYLON [47], SAFDID [48] and Cactus [49] are examples of frameworks implementing autonomic computing in order to dynamically reconfigure the system. The purpose of frameworks is to ensure reusability and extensibility, two main types of frameworks are distinguished [3]:

- ***White Box Framework***: In a white box framework, we usually extend behavior by creating subclasses, taking advantage of inheritance. A white box framework often comes with source code.
- ***Black Box Framework***: In a black box framework, the behavior is extended by composing objects together, and delegating behavior between objects. We have no idea about technical implementation only the functionality of the component and where components can be plugged into the framework.

Table 2 recapitulates the difference between black and white frameworks.

White Box Framework	Black Box Framework
Use subclassing	Use Composition and delegation
Inheritance	Polymorphism
Must know the internal structure	Must know interfaces
Simpler, easier to design	Complex, harder to design
Harder to program	Easier to program

Table 2. Comparison between White and Black Box Frameworks

3.3 Model-based implementation approaches

Model-based solutions implement all actions starting from the scratch. No primitives are offered. In this category, we focus particularly on architectural approaches in which constraints can be expressed explicitly. Management modules are generally proposed to ensure the dynamic reconfiguration process. These entities enable monitoring, implement analysis and planning, and enforce reconfiguration actions.

The work described in [7] provides "Kinesthetics eXtreme" (KX), which is a generic framework for self-healing based on lightweight middleware. It is a decentralized architecture using event-based system. It can be implemented at the middleware level or at the application level. The analysis is based on specific rules for error detections. KX approach follows the functional properties of an application in order (i) to semantically analyze exchanged messages; it plays the role of proxy for capturing message contents (source address, subject, message-ID) for detecting SPAM messages (ii) and to monitor the beginning and the end of method calls in order to detect services which run incorrectly. KX is scalable and can be applied to a composed system. However, it has limits: First, the scenarios is implemented with manually-written glue code for attaching the external autonomic infrastructure to the target system. Second, the probe deployment, the gauge derivation, and the construction of reconfiguration plans are performed manually.

Rainbow [6, 50] is a framework for self-adaptive systems. It is composed of two layers. First, the system layer, which collects information about the system and enforces reconfiguration plans. Second, the architecture layer, which reflects the current architecture model, checks constraint violations, and determines the required adaptation. The architecture and system layers interact through a *translation infrastructure*. However, experimental work [52] has shown that this externalized approach for self-adaptation causes a significant slowdown of the system behavior. Also, this approach supposes that the target system contains hooks for monitoring and management. The reconfiguration plan is built manually and integrated in the code. No evaluation or validation of this plan is provided.

Other model-based approaches use architectural styles designed to enable autonomic computing. In fact, an architectural style represents a collection of design decisions that have already been made and can be reused. It consists in a few key features and rules for combining these features so that architectural integrity is preserved. From this category, we can mention Prism [22, 53] which is an autonomic approach based on component oriented architectures. It is composed of two layers. The first is the application layer which includes functional components and exchanges messages between them. The second represents the autonomic layer. At this layer, components act as effectors. They facilitate monitoring, enacting changes, planning and deployment. These effectors are aware of application-layer components and may initiate interactions with them, but not vice versa. The proposed style, even it is abstract, seems to be complete as it covers most requirements for autonomic system like adaptability, dynamics, awareness and robustness. It is one of the few approaches which considers mobility. However, this framework does not target a solution for specific application context, but it is a general approach for designing autonomic systems. Specific functionalities, like coordination and policies are not considered in this approach. The applicability of the style has been demonstrated.

Taylor and all present an architectural approach for autonomic computing. In [25], Taylor and Oreizy propose an approach for self-adaptive architecture. This approach is neither implemented nor applied to a specific application context. It focuses on system integrity which requires the management of consistency, correctness and coordination of reconfiguration actions. In [54], Taylor and Dashofy refine the architecture proposed in [25]. They exhibit a style-based approach for autonomic systems. They provide an infrastructure able to support the design, validation and execution of reconfiguration plans. The architecture is described using xADL 2.0 (XML-based Architecture Description Language). The infrastructure is mainly designed for event-based systems. However, this architecture is centralized for single process systems and no mechanism is considered for error detections.

3.4 Middleware-based implementation approaches

The middleware-based approaches provide primitives helping developers to implement autonomic computing system. In the following, we present middleware-based implementations. Details about each middleware are presented in table 3.

OpenORB *middleware* [10] is built on the basis of reflective technology. Its reflective architecture uses meta-object protocols to perform integrations that support dynamic adaptation at runtime. The meta-models allow monitoring and reconfiguration of misbehaviors in order to preserve the architecture style. Through the interception meta-model, it analyzes exchanged messages between components and client requests. The interface meta-model provides access to the component implementation while the architecture meta-model provides access to the object graph. The illustration prototype deals with the continuous media flows transmission quality while introspecting and reconfiguring available resources (CPU, network, etc.) in order to maintain QoS [55]. This case study

demonstrates that OpenORB provides sufficient support for small-scale multimedia applications. To secure the communication between peers, authors of [10] said that it may be done while using interceptors.

DynamicTAO [56] is a reflective ORB (an extension of the TAO ORB). It enables detecting changes in environment and reloading new component implementations which may be bound to the system at runtime. These features are achieved by the use of a collection of entities known as component *configurators*. These configurators maintain information about the dependencies between the components they manage. The *DynamicConfigurator* inspects component implementations (list_categories, list_components, domain_component, impl_info, comp_info, etc.) and reconfigures system on the fly while loading or removing implementations stored in a *Repository*. The scalability of DynamicTAO is not improved. However, it is only tested with a simple example (*getHello()*). The DynamicTAO infrastructure includes two management security services. The first is used to crypt/decrypt message contents and the second authenticates communication peers to control access. The security strategy can be loaded and bound dynamically to the system at runtime. This allows the use of a large range of security models.

Eternal [57] is a component-based middleware which provides fault tolerance to CORBA-based applications by replicating components. The autonomic computing aspect is developed as an external layer underneath the ORB layer. The monitoring is based on the middleware interception approach which is transparent to all ORB. Eternal intercepts client requests and transfers them to a replication manger in the case of misbehaviors. A simple test application is done in order to measure the performance of Eternal when a fail happen. It shows the recovery time required for reconfiguration while varying the size of the application state to transfer across the network. The drawn curve points out that the recovery time increases while the transferred data size goes up. Eternal uses CORBA security service (SecIOP) and integrates SSL to secure exchanged messages. Also, it implements a firewall in order to filter requests and accepts only authenticated clients.

Middleware	DynamicTAO	Eternal	OpenORB	JavaPod	CME
Monitoring	*Event Collector*: Observes behavior of components and generates relevant QoS events	*DynamicConfigurator*: Inspects component implementations	*Interceptor*: Uses three technics: checkpoint, pinging ("i-am-alive" periodic message), and logging	*Server Container*: No given details	*Connection Monitor*: Monitors network devices (as modem) and controls protocol entities (as routing table).
Analysis	*Monitor*: Collects QoS events and reports abnormal behaviors	*The user*: Interrogates the database, in order to inspect QoS values	*Fault Detector*: Communicates the occurrence of faults to the Fault Notifier	No given details	*Adaptation Trigger*: Triggers adaptation based on predefined context criteria
Planning	*Strategy Selectors*: Selects an appropriate adaptation strategy based on feedback from monitors	*The user*: Chooses the suitable implementation from each category	*Fault Notifier*: Distributes fault event notifications to Replication manager	No given details	*Adaptation Selector*: Chooses suitable adaptation approach
Execution	*Strategy Activators*: Implements a particular strategy, e.g. by manipulating component graph while preserving the architectural style	*DynamicConfigurator*: Loads and binds new service implementations (load, resume, suspend, remove, delete etc.)	*Replication Manager*: Transfers the system state towards a replica	*Composition*: Extends component with new implementations and wraps requests	*Adaptation Executor*: Executes commands (open and switch channels) and changes entity behaviors
External/Internal	External or Internal	External	External	External	External
Programming language	Python	C++ / Java	C++	ejava (extension of Java)	Java
Scalability	High	Low	Medium	Medium	Low
Security	Low	High	High	Medium	Medium
Application domain	Component/Web service	Component	Component	Component/ Mobile Component	Network devices (PC, PDA,...) and protocols (LAN, GPRS,...)
Reconfiguration Strategy	*Structural*: Modifies the architecture	*Behavioral*: Reloads a new component implementation	*Structural*: Connects clients to replica	*Behavioral*: Extends components with new implementations	*Structural/Behavioral*: Switching/Adjusting channels.

Table 3. Middleware features

JavaPod [58] is a reflective middleware which focuses on the separation of functional and nonfunctional properties in a distributed context. The adaptation is managed using object compositions. It is achieved while dynamically extending methods with new implementations. Authors developed a java extension to make easy implementation of this approach. They provide new protocols to manage connections and handle faults. For the security, JavaPod implements an access control list, which allows to manage access at the method level for each user. For the evaluation, an e-learning application, called *Baghera*, is used as a case study. The performance evaluation shows that an overhead is considerable due to the composition mechanism. But, this overhead is negligible compared to the communication time which is enhanced.

CME [59] (***C**onnection **M**anagement **E**ngine*) is a middleware for network applications. It manages logical connections (called channels) between two communication peers. It monitors channels to determine which stations communicate with each other, when connections begin and end, and how much information is exchanged. CME uses a policy mechanism to facilitate the Network management. Policies represent adaptation requirements to ensure at runtime. CME enables monitoring of security privileges by recording which stations communicate with each other. The scalability is not improved. The middleware is only tested with a prototype on PDA. CME acts on the network level. Consequently, it is above the Operating System and it can support many kinds of applications.

DynamicTAO, Eternal, JavaPod and CME are developed as an external layer to manage autonomic systems at runtime. However, OpenORB can be external or internal. External, in which the monitoring component is supported by applications as an external service. Internal, in which the monitoring component is injected into the application components to provide such service. With Eternal, the reconfiguration mechanism is limited to the replication. DynamicTAO does not provide entities for fault detections and analysis. The user has to inquire about the application health and to choose the suitable reconfiguration plan. JavaPod does not provide any entities for analyzing and planning. Meanwhile, all steps are automated with other middlewares.

3.5 Platform-based implementation approaches

In this section, we present the main suggested platforms employed for developing autonomic applications. The evaluation is based on criteria including provided self-aware, used components and the architecture types supported by these platforms.

***Unity** [11, 60]*: The Unity project is looking for how component behaviors and relationships can support self-management of computing systems. The Unity project implements a prototype of autonomic systems, designed to show the feasibility and to validate the dynamic reconfiguration of the environment. During runtime it reallocates and reconfigures resources to optimize its behavior according to specified policies. In this approach, every component incorporates an autonomic part in a way that it becomes autonomic. The different components of Unity are:

- "Application environment manager": which is responsible for the management, communication between components, and predicting the resource availability.
- "Resource arbiter": which manages sharing and allocation of resources.
- "Registry": which allows locating components.
- "Policy repository": which represents administration interfaces.
- "Sentinel": which is used by a component to monitor the functioning of another.
- "Solution manager": which is responsible for the reconfiguration and the maintenance.

The monitoring is enabled by all components, including defective components (if they exist) which can cause system damages. They have to be sure about monitored Data. They should add policies in order to filter gathered data.

***PAC-MEN** [12]: **P**ersonal **A**utonomic **C**omputing **M**onitoring **EN**vironment:* PAC-MEN provides concepts and techniques for a range of platforms including PCs, mobile laptops, PDA etc. It is based on '*reflex reaction*' in order to respond to threats, and '*vital signs*' to assess operational health (inspired from human mechanisms). In this approach, every peer in the system is an element:

- Each peer is responsible for its own internal behavior.
- Each peer may be extended to include shared monitoring of the external environment to inform group members of events that may require individual action.

PAC-MEN approach proposes to set up a management server in each peer called "Autonomic Manager" (AM), which shares data and management decisions between other *AMs*. This allows collaboration of *AMs*. However, this style is more dynamic and decentralized.

***CODA** [61]: **C**omplex **O**rganic **D**istributed **A**rchitecture*: CODA applies concepts such as self-organization, self-regulation and viability to derive an intelligent architecture. It reacts to operation failures and proactively searches for successful patterns of behavior. CODA is a layered approach. It contains five layers:

- "Operations": which represents business operations of a system.
- "Monitor Operations": which performs internal monitoring.
- "Monitor of the Monitors": which performs external monitoring.
- "Control": which learns about faults and predicts reconfiguration actions.
- "Command": which recognizes threats and makes decisions.

Platform names	Unity	PAC-MEN	CODA	Cactus	MAIS	Jade	SAFDIS
Centralized/ Decentralized	Decentralized	Decentralized	Centralized	Decentralized	Centralized	Centralized	Decentralized
Architecture type	Client/Server	peer-to-peer, Grid	Client/Server	Client/Server	Client/Server	Client/Server	Client/Server
Human intervention	Minimal	No	No	No	Interaction	No	No
Self-aware	Self-configuration, self-optimizing, self-protecting, self-healing	Self-healing, self-awareness, self-monitoring	Self-organization, self-regulation, self-monitoring	Self-configuration, self-adaptation	Self-adaptation, self-optimizing	Self-adaptation, self-optimizing	Self-adaptation, self-optimizing
External/ Internal	Internal	Internal	Internal, external	External	Internal, external	External	External
Programming language	Java	Java	Java	C/C++/Java	Java	Java	Java
Presentation	GUI	No	No	Monitoring interface	No	Monitoring visualizator	No
Monitoring	Sentinel	Vital signs	Monitor operations, Monitor of the Monitors	Event Handlers	Diagnoser and Inspector	Monitor	Event Manager
Execution	Solution manager	Reflex reaction	Command	Event Handlers	Recovery actions	Connector	Migration of Services
Level	Application	Application and device	Application	Network	Application, network and device	Application	Application
Web services	Yes	No	Yes	No	Yes	Yes	No

Table 4. Platform features.

***Cactus** [49]*: Cactus provides support for dynamic adaptation and offers a potential solution for building autonomic software in networked systems. A service in Cactus is implemented as a "set of handlers" which reacts when an event occurs in order to manage QoS (reliability, timeliness, performance, and security). In the case of CORBA based application, the service is a communication protocol that resides in the protocol stack on top of a lower level communication service such as UDP. Handlers react when a message is exchanged between the client and the server. Cactus proposes several handler kinds. Each handler does not need to know about other one and we can choose the desired handler for each service. This may return Cactus more configurable and more adaptable.

***MAIS** [13]: **M**obile **A**daptive **I**nformation **S**ystems*: The MAIS project studies adaptability at all levels in information systems, from application level to network and device levels (PCs, laptops, palmtops, cellular phones, and so on). Several levels of adaptability are considered: the upper level (Application level), the middle level (web service level) and the bottom level (Infrastructure & Middleware level). MAIS provides an environment to run composite, multi-channel, mobile, and context-aware web services in an adaptive way. The MAIS architecture implements a runtime service-oriented fault analysis and recovery actions. It detects faults by inspecting request and response messages and analyzing them through a diagnoser component. This architecture provides four modules to handle reconfiguration actions, namely: "reallocation", "substitution", "wrapper generator" and "quality modules".

***Jade** [62]*: Jade is an autonomous administration platform for software infrastructure. It provides an abstract view of the application and acts when a failure occurs on a part of the system. It uses duplication to maintain the service availability and to handle the resource allocation according to the load variation in order to manage scalability. Jade is composed of two parts:

- *Managed Element*: which wraps each software and provides an administration interface;
- *Autonomic Manager*: which implements the administration management policies (repair and optimization). It monitors and acts on system through the *Managed Elements* interfaces.

***SAFDIS** [48]*: **S**elf-**A**daptation **F**or **DI**stributed **S**ervices, enables the dynamic management of service-based architectures. The implementation is built for the OSGi platform, using iPOJO to manage the life-cycle. The different components are:

- *Adaptation Manager*: its role concentrates on ensuring the communication between the different components and services responsible of the adaptation of the system.
- *Event Manager*: it collects events from monitors, composes them and keeps a local view of the system. It is the supervisor of the whole system.
- *Analyst*: it is a distributed and a decentralized process. It identifies, analyzes the system changes and decides if an adaptation is needed. Then, it makes an adaptation decision when a need arises. Furthermore, this component is composed of:

  - *Decision Maker*: it listens to events coming from the event manager and sends them to the reasoners for analysis.
  - *Negotiator*: it is composed of a back end and a front end connected to a remote negotiator of another SAFDIS instance.
  - *Negotiation Manager*: it is responsible of managing the multiple negotiations that can happen at the same time.
- *Planner*: it is composed of a set of *Planning Algorithms* and a *Manager* component. According to objectives and constraints, the *Manager* produces simple orderings of actions to reconfigure system.
- *Execution Engine*: it is called to perform planned actions. In SAFDIS, the reconfiguration action moves a service from an execution node to another,which is called the migration of services.

There are other platforms like in [63] and [64]. The work described in [63] presents a reflexive and dynamically adaptable execution environment which allows building dynamic, reflexive and flexible application and middleware. Authors of [64] propose a mechanism for adding and adapting services based on an adaptable reconfiguration language.

In table 4, we present properties of each platform. Most platforms do not allow any interaction between human or administrator and application except for Unity and MAIS. The programming language supported by all platforms is Java. But, there is another Cactus version that supports C and C++. CODA, Unity and MAIS support web services based dynamic reconfiguration. The others may investigate to support them. Cactus integrates the dynamic adaptation in the level of the exchanged data. In order to support web services, it may extend SOAP protocol to take into consideration new handlers for QoS preservation.

3.6 Concluding Remarks

Internal vs. External autonomic computing All cited implementations have as goal the dynamic reconfiguration of the system which allows it to evolve incrementally from one state to another at run-time in order to accommodate to changes [12]. The dynamic reconfiguration activities, based on autonomic computing, can be carried out either internally or externally to the application. In internal, codes responsible of the reconfiguration are merged with the application codes, while in external, they are separated from the application codes [65].

In an *Internal* autonomic mechanism, it is difficult to add a new code or a new strategy to a black-box component; we must know about the component design in order to govern it. The Unity [11] and PAC-MEN [12] projects present a prototype enabling dynamic reconfiguration, based on internal mechanisms.

External mechanism is appropriate when it is so difficult to modify application codes. We generally deploy components in heterogeneous context; therefore, if we use an internal strategy, we have to develop a new component version (with specific self-healing mechanism) for each context. Also, externalized mechanisms allow the reuse of autonomic components and make easy their update, since they are localized [66]. In addition, external mechanism allows us to divide the task

of the application implementation between the component developers and managers. Kinesthetics eXtreme [7] and Rainbow [6] built systems based on external mechanisms.

Each component may include autonomic mechanisms in order to heal itself. Designed systems have to inquire into problems and ask components to reconfigure their structure or behavior. Furthermore, dynamic reconfiguration strategy must not cause significant slowdown to the execution process and especially for the real time application. It must react in order to repair crashes while the variance of global system response time is kept in limited bounds. In order to reach a suitable and adaptable system which makes system resilient to faults, we have to apply autonomic computing techniques which cover all levels: hardware and software. But this solution may be very expensive and it requires combination of various mechanisms.

Behavioral vs. Structural autonomic computing We can distinguish two strategies of the dynamic reconfiguration in the Execution level. In fact, reconfiguration actions act on the system either behaviorally or structurally.

Following the first strategy , it is related to the **behavioral** dimension of the system in general; otherwise, it focuses on its internal behavior. Indeed, we speak about such approach when components behavior is customizable or modifiable. So, when degradation is detected, the installed reconfiguration infrastructure is brought to repair the process at runtime, by applying the reconfiguration actions to the concerned components. This reconfiguration is considered as a direct adjustment, because its actions are supposed to modify at once the internal composition of system components in order to correct it further to a problem. Eternal [57], CEYLON [47] and JavaPod [58] change the behavior of components in order to dynamically reconfigure the system.

Following the second one, it is related to the **structural** dimension of the system. So that systems components are observed during the execution of this later. Several symptoms are stored before taking the decision of activating or not reconfiguration actions. In this case, the reconfiguration is done by applying basis actions such as adding or removing components or their connections. The structural adaption referred to as "run-time" when the reconfiguration is scheduled during execution. DynamicTAO [56] and OpenORB [10] implement the structural reconfiguration. While, CME [59] implements both structural reconfiguration by switching and behavioral reconfiguration by adjusting channels.

4 DRAAS: Dynamically Reconfigurable Architecture for Autonomic Services

In this section, we present our approach, DRAAS which manages QoS of web service-base applications at runtime. It is based on monitors, able to extend SOAP messages exchanged between the service requester and the service provider (Web Service), and a dynamic connector which is used to redirect requests to concrete providers offering the same business logic.

4.1 DRAAS architecture

DRAAS provides the management of QoS by implementing virtualization and the different components of the autonomic computing architecture reference (MAPE K-loop): The first step is Monitoring. It corresponds to the supervision of the application. It observes flows and stores the value of the monitored data. Second, the Analysis detects the QoS degradation. If detected, an alarm signal will be sent to the planning module. Third, the Planning identifies the origin of the QoS degradation and calculates the new reconfiguration. Fourth, the Execution module executes the reconfiguration actions.

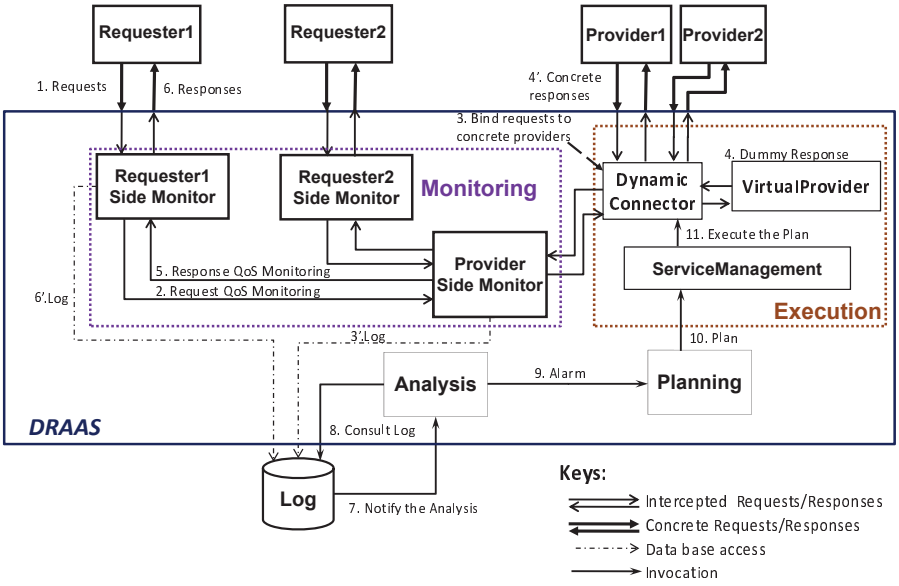

Fig. 2. DRAAS architecture

The figure 2 presents an overview of DRAAS deployed between *Requesters* and *Providers*. Software entities composing our architecture are:

- In the Monitoring module:
  - RSM: Requester Side Monitor, associated to each requester, is responsible of intercepting inflow/outflow (Request/Response) in the requester side.
  - PSM: Provider Side Monitor, associated to all providers, is responsible of intercepting inflow/outflow (Request/Response)in the provider side.
- Analysis for detecting QoS degradation.
- Planning for calculating the new reconfiguration.
- In the Execution module:

- VirtualProvider, is the initial destination of requester requests.
- Dynamic Connector, redirects/binds requester requests to concrete providers according to the reconfiguration plan.
- ServiceManagement, executes the new reconfiguration.

According to the comparative study established in the previews sections, DRAAS is a decentralized architecture thanks to Web Services, based on client/server communication, implements the structural strategy by redirecting requests. It is an externalized approach for dynamic reconfiguration.

4.2 Illustration: Data Load Use Case

In this section, we illustrate the DRAAS architecture within the *Data Load* use case. It consists in transferring files from the client side to a Load Repository. Transferring files is provided by providers (Web Services) which offer the LoadTransfer service and each file is associated to a request.

A prototype of DRAAS is implemented enabling the balance of load among available providers in order to manage QoS such as response time which corresponds to minimizing the transfer time. Balancing requests is the task of the dynamic connector. According to the DRAAS architecture, presented in figure 2, we distinguish these actors:

- Requester $\longrightarrow$ Client
- Provider1 $\longrightarrow$ LoadTransferWS1
- Provider2 $\longrightarrow$ LoadTransferWS2
- VirtualProvider $\longrightarrow$ LoadTransferVirtualWS

Initially, the client sends files to the *LoadTransferVirtualWS*. Each one is encapsulated in a request. Each request is intercepted twice, first by the *RSM* and second by the *PSM*. The *Dynamic Connector*, associated to the *LoadTransferVirtualWS*, balance the load (requests) by redirecting them to *LoadTransferWS1* or *LoadTransferWS2*. Each web service provider (*LoadTransfer*) transfers a file per request. If the transfer of each file is successfully done, each response is also intercepted twice as the request but inversely: first by the *PSM* and second by the *RSM*. All monitored data is stored in the log. If the *Analysis* detects an increase of the transfer time, it sends an alarm to the *Planning* in order to calculate a new reconfiguration. In this case, the Planning decides to activate an available *LoadTransferWS3* to participate in the next transfer. It sends this decision to the ServiceManagement to perform it. The execution of this decision will be caught by the *Dynamic Connector*. It will be taken into account for the next load transfer.

4.3 Experimentation

To show the feasibility of DRAAS, we have carried out experiments on the Data Load. We present in the sequel hardware architecture and tools used for these experiments.

Hardware Architecture and Tools All test scenarios (to be presented in the next section) are assessed under this configuration:

- Operating system: Windows 7, 32 bits
- Processor: Intel Core(TM)2 Duo CPU T5800
- RAM: 2Go

Our implementation is built of the Web Service technology. Analysis, Planning and ServiceManagement are Web Services, while monitors and Dynamic Connector are based on handlers. In the following, we cite the technical choices for our implementation:

- Web service container: Axis2 1.5
- Web server: Tomcat 6.0.30
- Programming language: Java 1.6
- Monitors & Connectors: Axis2 Handlers
- Communication level: SOAP
- Logging: MySQL DBMS

Assessment To assess DRAAS performance, we have fixed the global size of the files to be transferred (T=32Mo) and we have prepared six scenarios for testing. All scenarios focused on varying the number of files while maintaining the global size. We present in table 5 the different scenarios used to evaluate DRAAS performance.

	Number of files	***File Size***
Scenario1	1	{32 Mo}
Scenario2	2	{17Mo; 15Mo}
Scenario3	3	{10Mo; 11Mo; 11Mo}
Scenario4	4	{9Mo; 8Mo; 8Mo; 7Mo}
Scenario5	8	{3,7Mo; 4,3Mo; 3Mo; 5Mo; 4Mo; 4Mo; 4,2Mo; 3,8Mo}
Scenario6	10	{3,2Mo; 2,8Mo; 3Mo; 2,8Mo; 3,2Mo; 3,2Mo; 3,2Mo; 3,2Mo; 3,7Mo; 3,7Mo;}

Table 5. Load transfer scenarios

In order to show the benefits of DRAAS, we have distinguished two cases for the *Data Load* use case: First, the transfer of file is accomplished without load balancing. Second, deploying DRAAS in order to maintain the QoS management, such transfer time, at runtime. Therefore, without applying DRAAS to the *DataLoad*, the client is connected only to the *LoadTransferWS1* Web Service, even if there is another available *LoadTransferWS2* Web Service providing the same business logic, and all files are transferred through it. If the *LoadTransferWS1* Web Service shows a QoS degradation, expressed by an increase of the transfer time, this degradation affects the Data Load application.

However, when we integrate our DRAAS prototype as described in the previews section, the load will be balanced on available Web services offering the load transfer service.

We have carried out each scenario experiments at least 5 times. According to obtained values, results are shown in table 6, where the average equals to the sum of values obtained by tests divided by the number of tests.

	Response Time (ms)					
	Single Web Service			*Two Web Services(DRAAS)*		
	Min	**Max**	**Avg**	**Min**	**Max**	**Avg**
Scenario1	4565	5141	4812,6	3748	5168	4700,6
Scenario2	6210	7669	6717,44	4067	5357	4918
Scenario3	7780	8228	7920	4520	5583	5039,4
Scenario4	8790	9609	9153,6	4090	5362	4600
Scenario5	14592	16326	15451,6	7611	9631	8324,6
Scenario6	18313	25147	20113,8	8123	8500	8510,4

Table 6. Performance measurement

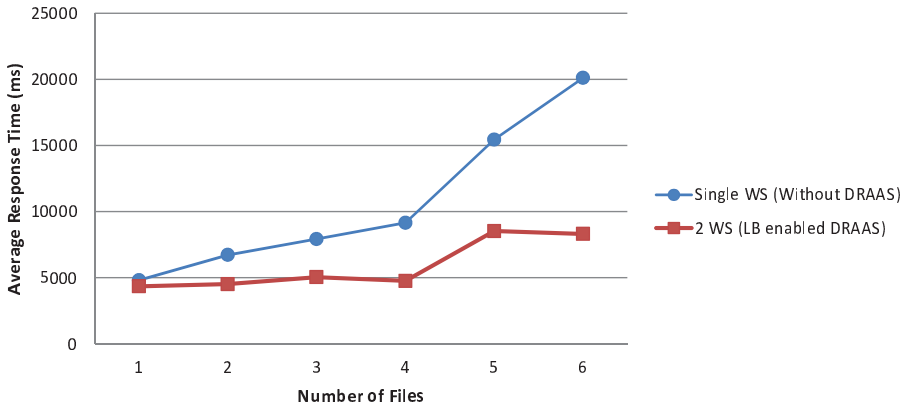

Fig. 3. Average response time of DRAAS prototype

Our experiments provided the curves shown in Figure 3. The blue curve (-●-) describes the average response time related to transferring a variable number of files with a single Web Service where the global size is maintained constant and equal to 32Mo. However, the red curve (-■-) describes the same parameters but while using two Web services and enabling the load balancing.

It is obvious that transferring files within our DRAAS prototype, using two Web services, is more efficient in term of transfer time than using a single Web service. We have noticed that the transfer time (response time) depends on the number of files. In fact, without DRAAS, increasing the number of files while maintaining the overall size leads to increase the response time. However, the DRAAS curve presents a critical point having the following coordinates (4, 4600) for which the average response time is optimal. Moreover, the deployment of

DRAAS with a single provider causes the increase of the response time due to an added delay ϵ, epsilon, caused by the virtualization. This ϵ has no impacts on response time since both the number of requests and the number of providers have exceed two. Therefore, the deployment of DRAAS is based on a necessary and sufficient assumption which is the presence of at least two providers offering the same business logic. Indeed the presence of a single provider does not allow the dynamic reconfiguration, which is currently based on load balancing.

5 Conclusion

In this paper, first,we have presented a classification and a comparative study of existing architectures and frameworks implementing autonomic systems. Different implementations are provided. For example, a model-based solution is usually suitable for a small system. Platform-based solutions are appropriated for systems in which only generic QoS properties are required. The new objectives are oriented towards the deployment and the execution of distributed applications on heterogeneous platforms (PC, smart devices, Smart card, etc).

Second, we have proposed our DRAAS architecture to bring dynamic reconfiguration capabilities to distributed web service-based applications. A prototype of DRAAS has been implemented to assess the applicability of the monitoring and reconfiguration within the designed architecture. The repair enactment is based on the architectural reconfiguration providing load balancing for web services at the origin of the QoS degradation.

We aim to improve our DRAAS architecture's to support new reconfiguration actions such as substitution. Moreover, we target to manage dynamically MAPE-K loop components while enabling flexibility by changing their behaviors at runtime in order to include new features, such as new monitors or new analysis algorithms.

References

1. Shin, M.E.: Self-healing components in robust software architecture for concurrent and distributed systems. Journal of Science of Computer Programming **57**(1) (July 2005) 27–44
2. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1) (2003) 41–50
3. Ciupa, I.: Study on whitebox frameworks in java. (2003)
4. Conte, A., Anquetil, L.P.: A black box framework for an application protocol stack. In: Proceedings of the 3rd IEEE Symposium on, Application-Specific Systems and Software Engineering Technology, 2000, IEEE Computer Society (2000) 96 – 101
5. Gurguis, S.A., Zeid, A.: Towards autonomic web services: achieving self-healing using web services. In: DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software, New York, NY, USA, ACM Press (2005) 1–5

6. Cheng, S.W., Garlan, D., Schmerl, B.R.: Making self-adaptation an engineering reality. In: Self-star Properties in Complex Information Systems, Conceptual and Practical Foundations [the book is a result from a workshop at Bertinoro, Italy, Summer 2004]. Volume 3460 of Lecture Notes in Computer Science., Springer (2005) 158–173
7. Wile, D.S., Egyed, A.: An externalized infrastructure for self-healing systems. In: WICSA '04: Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04), Washington, DC, USA, IEEE Computer Society (2004) 285
8. Suzuki, J., Suda, T.: A middleware platform for a biologically inspired network architecture supporting autonomous and adaptive applications. In IEEE Journal on Selected Areas in Communications (JSAC) **23**(2) (February 2005) 249–260
9. Zeng, L., Benatallah, B., Ngu, A.H., Dumas, M., Kalagnanam, J., Chang, H.: Qos-aware middleware for web services composition. IEEE Trans. Softw. Eng. **30**(5) (2004) 311–327
10. Blair, G.S., Coulson, G., Blair, L., Duran-Limon, H., Grace, P., Moreira, R., Parlavantzas, N.: Reflection, self-awareness and self-healing in openorb. In: WOSS '02: Proceedings of the first workshop on Self-healing systems, New York, NY, USA, ACM Press (2002) 9–14
11. Chess, D.M., Segal, A., Whalley, I., White, S.R.: Unity: Experiences with a prototype autonomic computing system. In: 1st International Conference on Autonomic Computing (ICAC 2004), 17-19 May 2004, New York, NY, USA, IEEE Computer Society (2004) 140–147
12. Sterritt, R., Bantz, D.F.: Pac-men: Personal autonomic computing monitoring environment. In: 15th International Workshop on Database and Expert Systems Applications (DEXA 2004), Zaragoza, Spain, IEEE Computer Society (2004) 737–741
13. Cappiello, C., Missier, P., Pernici, B., Plebani, P., Batini, C.: Qos in multichannel is: The mais approach. In: Engineering Advanced Web Applications: Proceedings of Workshops in connection with the 4th International Conference on Web Engineering (ICWE 2004), Munich, Germany, 28-30 July, 2004. (2004) 255–268
14. Paulson, L.D.: Computer system, heal thyself. Computer **35**(8) (2002) 20–22
15. Parashar, M., Hariri, S.: Autonomic computing : An overview. (2005) 247–259
16. White, S.R., Hanson, J.E., Whalley, I., Chess, D.M., Kephart, J.O.: An architectural approach to autonomic computing. In: 1st International Conference on Autonomic Computing (ICAC 2004), 17-19 May 2004, New York, NY, USA, IEEE Computer Society (2004) 2–9
17. P.K., S., S., S.: Secured remote tracking of critical autonomic computing applications. published in IEEE E-Tech, Karachi, Pakistan (2004)
18. Charles Gouin-Vallerand, S.G., Abdulrazak, B.: Toward a self-configuration middleware for smart spaces. In: FGCN '08: Proceedings of the 2008 Second International Conference on, Future Generation Communication and Networking, 2008. Volume 2., IEEE Computer Society (2008) 463 – 468
19. Riadh Ben-Halima, Khalil Drira, M.J.: Survey a qos-oriented reconfigurable middleware for self-healing web services. In: ICWS '08: Proceedings of the 2008 IEEE International Conference on Web Services. Volume 1., IEEE Computer Society (2008) 104 – 111
20. Tosic, V., Pagurek, B., Patel, K., Esfandiari, B., Ma, W.: Management applications of the web service offerings language (wsol). In: Advanced Information Systems Engineering, 15th International Conference, CAiSE 2003, Klagenfurt, Austria, June

16-18, 2003, Proceedings. Volume 2681 of Lecture Notes in Computer Science., Springer (2003) 468–484
21. Chang, F., Karamcheti, V.: Automatic configuration and run-time adaptation of distributed applications. In: HPDC '00: Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00), Washington, DC, USA, IEEE Computer Society (2000) 11
22. Medvidovic, N., Mikic-Rakic, M.: Programming-in-the-many: A software engineering paradigm for the 21st century. In: Workshop on New Visions for Software Design and Productivity: Research and Applications, Nashville, Tennessee (December 2001)
23. Cheng, S.W., Garlan, D., Schmerl, B.R., Sousa, J.P., Spitnagel, B., Steenkiste, P.: Using architectural style as a basis for system self-repair. In: WICAS3: Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture, Deventer, The Netherlands, The Netherlands, Kluwer, B.V. (2002) 45–59
24. Schmidt, H.: Trustworthy components-compositionality and prediction. Journal of Systems and Software **65**(3) (2003) 215–225
25. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems **14**(3) (1999) 54–62
26. Guinea, S.: Self-healing web service compositions. In: ICSE '05: Proceedings of the 27th international conference on Software engineering, New York, NY, USA, ACM Press (2005) 655–655
27. Richters, M., Gogolla, M.: Aspect-oriented monitoring of uml and ocl constraints. In: In AOSD Modeling With UML Workshop, 6th International Conference on the Unified Modeling Language (UML. (2003)
28. Sridhar, N., Pike, S.M., Weide, B.W.: Dynamic module replacement in distributed protocols. In: Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on, IEEE Computer Society (June 2003) 620 – 627
29. Bouchenak, S., Boyer, F., Krakowiak, S., Hagimont, D., Mos, A., Jean-Bernard, S., Palma, N.d., Quema, V.: Architecture-based autonomous repair management: An application to j2ee clusters. In: SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems, Orlando, Florida, USA, IEEE Computer Society (2005) 13–24
30. Yoo, G., Lee, E.: Monitoring methodology using aspect oriented programming in functional based system. In: Advanced Communication Technology (ICACT), 2010 The 12th International Conference on. Volume 1., IEEE Computer Society (April 2010) 783 – 786
31. seong Lee, K., Lee, C.G.: Model-driven monitoring of time-critical systems based on aspect-oriented programming. In: Secure Software Integration and Reliability Improvement (SSIRI), 2011 Fifth International Conference on, IEEE Computer Society (August 2011) 80 – 87
32. Mdhaffar, A., Ben-Halima, R., Juhnke, E., Jmaiel, M., Freisleben, B.: AOP4CSM: An Aspect-Oriented Programming Approach for Cloud Service Monitoring. In: Proceedings of the 11th IEEE International Conference on Computer and Information Technology, IEEE Press (2011) 363 – 370
33. Mostafaei, F.S., Amani, N., Hajipour, P.: Proposing a new qos/sla management model by regulatory authority. In: Telecommunications (IST), 2010 5th International Symposium on, IEEE Computer Society (2010) 508 – 512
34. Kandula, S., Katabi, D., Sinha, S., Berger, A.: Dynamic load balancing without packet reordering. ACM SIGCOMM Computer Communication **37** (April 2007)

35. Grace, P., Blair, G.S., Samuel, S.: Remmoc: A reflective middleware to support mobile client interoperability. In: On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE - OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003, Catania, Sicily, Italy, November 3-7, 2003. Volume 2888 of Lecture Notes in Computer Science., Springer (2003) 1170–1187
36. Ben-Halima, R., Drira, K., Guennoun, K., Jmaiel, M.: Non-intrusive qos monitoring and analysis for self-healing web services. In: First IEEE International Conference on the Applications of Digital Information and Web Technologies(ICADIWT 2008), Ostrava, Czech Republic, IEEE Computer Society (August 4-6 2008)
37. Truong, H.L., Samborski, R., Fahringer, T.: Towards a framework for monitoring and analyzing qos metrics of grid services. In: e-Science and Grid Computing, 2006. e-Science '06. Second IEEE International Conference on, IEEE Computer Society (Decembre 2006) 65 – 73
38. Zhang, H.Y., Urtado, C., Vauttier, S.: Connector-driven process for the gradual evolution of component-based software. In: Software Engineering Conference, 2009. ASWEC '09. Australian, IEEE Computer Society (June 2009) 246 – 255
39. Diaconescu, A.: A framework for using component redundancy for self-adapting and self-optimising component-based enterprise systems. In: OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2003) 390–391
40. George, S., Evans, D., Marchette, S.: A biological programming model for self-healing. In: SSRS '03: Proceedings of the 2003 ACM workshop on Survivable and self-regenerative systems, New York, NY, USA, ACM Press (2003) 72–81
41. MOO-MENA, F., DRIRA, K.: Reconfiguration of web services architectures: A model-based approach. In: Computers and Communications, 2007. ISCC 2007. 12th IEEE Symposium on, IEEE Computer Society (August 2007) 357 – 362
42. McBride, Matt: Software architecture and design. Technical report, IEEE EDUCATIONAL COURSES,Developed exclusively for IEEE eLearning Library (2011)
43. Hai-Shan, C.: Survey on the style and description of software architecture. In: Proceedings of the 8th International Conference on, Computer Supported Cooperative Work in Design. Volume 1., IEEE Computer Society (2004) 698 – 700
44. Yang Qun, Y.X.c., wu ;, X.M.: A framework for dynamic software architecture-based self-healing. In: Systems, Man and Cybernetics, 2005 IEEE International Conference on. Volume 3., IEEE Computer Society (2006) 2968 – 2972
45. Johnson, R.E.: Components, frameworks, patterns. ACM SIGSOFT Software Engineering Notes **22**(3) (1997) 10–17
46. Johnson, R.E.: Frameworks = (components + patterns). Communications of the ACM **40**(10) (1997) 39–42
47. Yoann Maurel, A.D., Lalanda, P.: Ceylon : A service-oriented framework for building autonomic managers. In: EASe'10: Proceedings of the 2010 Seventh IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems, IEEE Computer Society (2010) 3–11
48. Gauvrit, G., Daubert, E., André, F.: Safdis: A framework to bring self-adaptability to service-based distributed applications. In: SEAA'10: Proceedings of the 2010 36th EUROMICRO Conference on, Software Engineering and Advanced Applications, IEEE Computer Society (2010) 211 – 218
49. Hiltunen, M.A., Schlichting, R.D., Ugarte, C.A., Wong, G.T.: Survivability through customization and adaptability: the cactus approach. In: DARPA Information Survivability Conference and Exposition. (1999) 294–306

50. Cheng, S.W., Huang, A.C., Garlan, D., Schmerl, B.R., Steenkiste, P.: An architecture for coordinating multiple self-management systems. In: 4th Working IEEE / IFIP Conference on Software Architecture (WICSA 2004), 12-15 June 2004, Oslo, Norway, Washington, DC, USA, IEEE Computer Society (2004) 243–254
51. Huebscher, M.C., McCann, J.A.: Adaptive middleware for context-aware applications in smart-homes. In: Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing, New York, NY, USA, ACM Press (2004) 111–116
52. Garlan, D., Cheng, S.W., Schmerl, B.R.: Increasing system dependability through architecture-based self-repair. In: WADS. Volume 2677 of Lecture Notes in Computer Science., Springer (2002) 61–89
53. Mikic-Rakic, M., Mehta, N., Medvidovic, N.: Architectural style requirements for self-healing systems. In: WOSS '02: Proceedings of the first workshop on Self-healing systems, New York, NY, USA, ACM Press (2002) 49–54
54. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: Towards architecture-based self-healing systems. In: WOSS '02: Proceedings of the first workshop on Self-healing systems, New York, NY, USA, ACM Press (2002) 21–26
55. Limon, D.: A Resource Management Framework for Reflective Multimedia Middleware. PhD thesis, Lancaster University, UK (October 2001)
56. Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, L.C., Campbell, R.H.: Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In: Middleware 2000, IFIP/ACM International Conference on Distributed Systems Platforms, New York, NY, USA, April 4-7, 2000, Proceedings. Volume 1795 of Lecture Notes in Computer Science., Springer (2000) 121–143
57. Narasimhan, P., Moser, L.E., Melliar-Smith, P.M.: Eternal: a component-based framework for transparent fault-tolerant corba. Softw. Pract. Exper. **32**(8) (2002) 771–788
58. Bruneton, E., Riveill, M.: Experiments with javapod, a platform designed for the adaptation of non-functional properties. In: REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, London, UK, Springer-Verlag (2001) 52–72
59. J., S., J., T., J., S.: Cme: a middleware architecture for network-aware adaptive applications. In: 14th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications. Volume 1., Beijing, China, IEEE Computer Society (2003) 839–843
60. Tesauro, G., Chess, D.M., Walsh, W.E., Das, R., Segal, A., Whalley, I., Kephart, J.O., White, S.R.: A multi-agent systems approach to autonomic computing. In: 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004), 19-23 August 2004, New York, NY, USA, IEEE Computer Society (2004) 464–471
61. Ribeiro-Justo, G.R., Karran, T.: Modelling organic adaptable service-oriented enterprise architectures. In: On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops, OTM Confederated International Workshops, HCI-SWWA, IPW, JTRES, WORM, WMS, and WRSM 2003, Catania, Sicily, Italy, November 3-7, 2003, Proceedings. Volume 2889 of Lecture Notes in Computer Science., Springer (2003) 123–136
62. de Palma, N., Bouchenak, S., Hagimont, D., Sicard, S., Taton, C.: Jade : Un Environnement d'Administration Autonome. Techniques et Sciences Informatiques **27**(9-10) (2008) 1225–1252
63. Ogel, F., Folliot, B., Piumarta, I.: On reflexive and dynamically adaptable environments for distributed computing. In: ICDCSW '03: Proceedings of the 23rd

International Conference on Distributed Computing Systems, Washington, DC, USA, IEEE Computer Society (2003) 112
64. Hachichi, A., Martin, C., Thomas, G., Patarin, S., Folliot, B.: Reconfigurations dynamiques de services dans un intergiciel à composants corbacm. In: 1ère Conférence Francophone sur le Déploiement et la (Re) Configuration de Logiciels, Grenoble, France (October 2004)
65. Qun, Y., Xian-Chun, Y., Man-Wu, X.: A framework for dynamic software architecture-based self-healing. SIGSOFT Softw. Eng. Notes **30**(4) (2005) 1–4
66. Garlan, D., Schmerl, B.: Model-based adaptation for self-healing systems. In: WOSS '02: Proceedings of the first workshop on Self-healing systems, New York, NY, USA, ACM Press (2002) 27–32