Autonomic iPOJO: Towards Self-Managing Middleware for Ubiquitous Systems

Ada Diaconescu, Johann Bourcier, and Clement Escoffier

Abstract— **The recent proliferation of ever smaller and smarter electronic devices, combined with the introduction of wireless communication and mobile software technologies enables the construction of a large variety of pervasive applications, such as home supervision and alarm systems. The inherent complexity of such applications along with their non-expert clientele raises the necessity for Autonomic Management solutions. Nonetheless, such solutions remain difficult to conceive, as they must deal with the increased volatility, heterogeneity and distribution of the pervasive domain, while ensuring stringent performance and dependability requirements. This paper proposes that reusable support for Autonomic Management solutions be provided by middleware platforms, along with already existing middleware services, such as security and transactions. Following this approach, a service oriented component platform, iPOJO, was extended with elementary Autonomic Management capabilities. These include monitoring and effector touchpoints, as well as embedded Autonomic Management functions, such as service dependency management. IPOJO is an open source Apache project and has been successfully employed to implement several research projects in the pervasive domain. This paper presents iPOJO's extension with reusable Autonomic Management middleware services.**

Index Terms— **autonomic management, pervasive computing, middleware, service oriented components, software engineering**

I. Introduction

OUR present living environments are being increasingly populated with ever smarter and smaller electronic devices. The introduction of such communicating devices has already changed the way we interact with our social and physical environments. However, this seems to be a mere beginning, as devices continue to feature progressively more capabilities and to increasingly cooperate with each other for providing new, higher-level services [1]. Still, important scientific and technical challenges must be tackled before fulfilling the pervasive computing vision. The development of high-level services using heterogeneous, distributed and highly-dynamic appliances proves particularly complex a task. Dynamicity, in particular, is a key characteristic of the pervasive domain, raising serious difficulties for the development and management of pervasive solutions. The execution environment of pervasive applications is constantly evolving, as users change their social contexts, or physical locations. Continuous fluctuations in the availability of software services and electronic gadgets, such as mobile phones and PDAs, further aggravate the situation. In addition, local pervasive applications, such as pervasive home gateways, must occasionally access remote applications, in order to provide Video on Demand, or Weather forecast services, for example.

In this context, the *autonomy* of home and office applications becomes critical for their successful integration in our present society. Indeed, clients of such applications are typically unknowledgeable in the computer science domain. Consequently, pervasive applications must feature inbuilt properties such as safety (e.g. fault-tolerance and security) and self-adaptation to evolving execution contexts (e.g. self-configuration, self-optimization and self-repair). Nonetheless, the development and maintenance of autonomic solutions remains a difficult endeavour, as strict dependability and performance guarantees must be offered while transparently dealing with complex technical and social issues.

From a technological point of view, Service Oriented Computing (SOC) provides the basic characteristics necessary for building flexible and reusable applications with complex and adaptable functionalities. The modularity and loose-coupling properties inherent to the SOC paradigm offer the opportunity for ad-hoc service compositions and dynamic change. Nonetheless, available Service Oriented Architecture (SOA) platforms implementing the SOC paradigm currently lack the necessary intrinsic functions for performing concrete service compositions and runtime modifications.

This paper proposes extending SOA platforms with inherent self-management capabilities, in order to support the creation of autonomic pervasive systems. iPOJO[1] service oriented component platform was selected for following this approach. This paper presents how iPOJO platform was applied and extended for providing inherent self-management services. This allows pervasive applications built and deployed on iPOJO platforms to obtain self-management support from the underlying middleware runtime, hence requiring no extra

Manuscript received June 20, 2008. This work was carried out as part of the ANSO project, which was partially supported by the French Ministry of Industry under the European ITEA program.

A. Diaconescu is with the University Joseph Fourier, 38041 Grenoble, France (phone: +33-476-63-5566; e-mail: ada.diaconescu @ imag.fr).

J. Bourcier is with the University Joseph Fourier, 38041 Grenoble, France (phone: +33-476-63-5568; e-mail: johann.bourcier @ imag.fr).

C. Escoffier is with the University Joseph Fourier, 38041 Grenoble, France (phone: +33-476-63-5568; e-mail: clement.escoffier @ imag.fr).

effort from service developers or system administrators. iPOJO is an open source Apache project and has been successfully employed in the development of several pervasive computing research projects (e.g. ANSO European Project and HOMEGA middleware platform [2]). The main contribution of this paper is to propose an extensible SOA platform, iPOJO, offering inherent self-configuration, self-repair and self-optimisation capabilities. The provided platform consequently facilitates the creation of autonomic pervasive solutions.

II. Pervasive Computing and Autonomic Management

A. *Requirements for Successful Pervasive Computing*

The success of any pervasive system highly depends on several key elements, including provided functionalities, Quality of Service (QoS) and affordability. In short, pervasive applications must offer services that are useful, or somehow interesting to the user. In addition, provided functionality must be associated with domain-specific QoS guarantees, such as performance, dependability and usability. System performance allows users to experience natural, real-time interactions with the pervasive environment. Dependability ensures service reliability and security and implies that the same behaviour is experienced every time the system is run in similar execution scenarios. Application usability implies ease of service exploitation, with no expert knowledge required and with minimal maintenance effort necessary for modifying and evolving the system. Finally, the overall utility of provided services must overcome the effort required to acquire and maintain the corresponding pervasive systems. Affordable pervasive solutions imply that clients are willing to invest the required resources in exchange for offered functionalities, both in the short term (e.g. acquisition and installation) and over long durations (e.g. maintenance).

Our previous work on the iPOJO framework focused on providing development support for pervasive application functionalities (e.g. [2], or [3]). This paper addresses QoS and affordability requirements by adding Autonomic Management (AM) capabilities to the existing middleware platform.

B. *Pervasive Applications Characteristics*

Pervasive computing systems generally consist of various electronic devices and software entities capable of communicating with one another. Different types of software-equipped appliances may be available for a variety of purposes, such as interacting with the real environment, providing control and display services, or exposing data and application interfaces to other devices and applications. The main challenge of the pervasive computing domain is to provide coherent pervasive environments, offering useful applications and services, based on an entanglement of heterogeneous, distributed and dynamic devices and software services, communicating via various technologies and protocols. In this context, several characteristics specific to pervasive equipments make this domain appealing from a business perspective, while raising difficult problems for system development and maintenance. Such device properties include:

- *Distribution*. Devices are typically scattered across the physical environment and accessible via diverse protocols, generally over a wireless communication support.
- *Heterogeneity*. A vast range of appliances, software technologies and communication protocols are available in the pervasive computing domain. A consensus on uniform and compatible implementations is not presently foreseen.
- *Limited resources*. Resource availability is generally scarce on the physical execution platforms employed in pervasive systems. In the pervasive home context, software applications typically run on a small gateway, with little memory space and low processing capabilities.
- *Dynamism*. Device availability is by far most volatile in pervasive systems with respect to other computing system types. This is due to several facts, including: i) users may freely and frequently change their locations and hence the locations of equipments they carry; ii) users may voluntarily activate and deactivate devices, or devices may unexpectedly run out of battery. This directly impacts on the availability of services running on these devices; iii) users and providers may periodically update deployed software services.

In addition to hardware and software dynamism, pervasive systems are constantly confronted with changes in their execution contexts. This may include modifications in the user's current behaviour, social circumstance, location, mood, or general routine, as well as changes in other software applications' availability and behaviours.

C. *Service Oriented Computing Platforms*

Service-Oriented Computing (SOC) is a recent popular technology that uses services as first-class elements for building software applications. In the SOC context, a service represents an abstract resource described by a contract. Contracts specify a service's interfaces, general semantics and QoS properties, while not referring to the service implementation. Consequently, services can be supplied by multiple service providers and feature various implementations. Available services are registered into one or more service registries where they can be subsequently found and recuperated by service consumers. An important consequence of this interaction pattern is that SOC technologies support dynamic service discovery and lazy inter-service binding. Such characteristics are essential when building applications with strong adaptability requirements.

iPOJO (e.g. [4], [5]) is a service component runtime that aims to simplify the development of applications on top of OSGi[2] SOC Platforms. iPOJO allows the straightforward development of application logic based on Plain Old Java Objects (POJO). iPOJO subsequently *injects* non-functional facilities into the application components, as necessary. Such

[1] iPOJO service component framework: www.ipojo.org

[2] OSGi Alliance : www.osgi.org

facilities include service provisioning, service dependency and lifecycle management. In addition to providing a reusable set of non-functional capabilities, iPOJO is seamlessly extensible to include new middleware functionalities.

The iPOJO framework merges the advantages of component and service oriented paradigms. Specifically, iPOJO application functionalities are implemented following the component orientation paradigm. Each component is fully encapsulated, self-sufficient and provides server and client interfaces exposing its functionalities and dependencies, respectively. As many component-oriented platforms (e.g. Java EE and .NET), iPOJO separates a component's application-specific business logic from its application-independent functions. As such, iPOJO components consist of a component implementation that is managed by a reusable *container* (Figure 1). iPOJO containers provide common middleware services to the component implementations they manage (e.g. distributed communication and lifecycle management). Each component container can be configured with a different set of middleware services, implemented as *handlers*. Once an iPOJO component is deployed, its provided functions are published and made available as services, in conformance with the SOC paradigm. In order for a component's services to become valid, all the component's dependencies must be resolved. For this purpose, available services corresponding to a component's required (or client) interfaces must be found and connected.

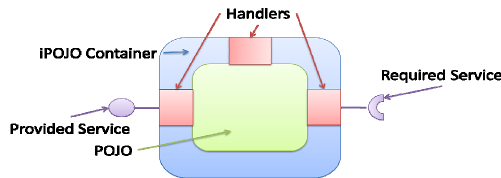

Figure 1: Internal design of an iPOJO component

Finally, iPOJO components can expose internal properties, which can subsequently be interrogated and modified during runtime. Among others, service properties enable clients to evaluate and select the services used when multiple service providers are available. Each iPOJO instance can be created with a different configuration of property values.

D. *Autonomic Management Benefits and Challenges*

Successful pervasive applications must fulfil specific functionality, QoS and affordability requirements (subsection A), while dealing with the important technical challenges associated (subsection B). Autonomic Computing[3] proposes constructing software systems capable of managing themselves, so as to self-optimise, self-configure, self-repair and self-protect themselves with minimum requirements for human intervention. Hence, Autonomic Computing promises affordable solutions to pervasive systems' distribution, heterogeneity, resource constrictions and dynamicity, ensuring system correct functioning, performance and dependability.

The general architecture currently adopted by the Autonomic Computing community consists of a management control-loop with the following main functions [6] Figure 2): i) *monitoring*, for extracting runtime information from the managed resources and their execution environments; ii) *analysis*, for detecting system anomalies based on the monitoring data; iii) *planning*, for finding solutions to the detected problems; and iv) *execution*, for putting the planned solutions into practice into the running system. In addition, a common *knowledge base* is shared by all functional modules in the control loop, for providing collective information such as accumulated system data, or historical records.

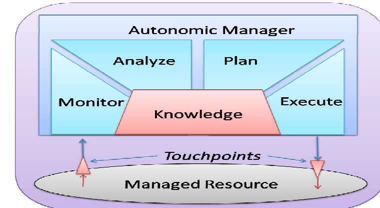

Figure 2: Autonomic Management architecture

This logical architecture may be implemented to various degrees of complexity, depending on the management level required. Basic reactive management solutions may employ minimal monitoring probes linked to system effectors via simple event-condition-action (ECA) policies. Sophisticated management solutions may require more complicated analysis and planning capabilities, for detecting a larger spectrum of anomalies (e.g. congestion tendencies, or oscillating system states) and for considering multiple, possibly conflicting management concerns (e.g. performance, repair and security).

Reusable tools and platforms are stringently needed today for assisting in the development of AM solutions. The current lack of such reusable tools means that each AM solution is built in isolation and in an ad-hoc manner (e.g. Jasmine[4], Jade [7], or Rainbow [8]). Developing such solutions from scratch requires non-negligible efforts and costs, which are then largely replicated for different AM applications.

III. AUTONOMIC IPOJO PLATFORM

A. *Solution Overview*

The development of dependable and efficient AM applications remains a difficult and costly a task. However, most elementary autonomic functionalities are applicable across multiple AM solutions. Such functionalities include monitoring and effector touchpoints accessible via standard communication protocols, such as JMX, or Web services. They also include simple anomaly detection functions, such as noticing when a monitored value crosses a predefined threshold, or when an inter-service binding is broken. Finally, common decision policies may be specified for solving most frequent anomalies, such as instantiating new servers to resolve resource contention, or finding alternative service providers when previous ones become unavailable.

This paper argues that common autonomic functions should

[3] Autonomic Computing: www.research.ibm.com/autonomic

[4] Jasmine project: wiki.jasmine.objectweb.org

be provided as reusable middleware services, offered by middleware platforms. Most recent component platforms already provide reusable middleware services such as security, or transactions (e.g. Java EE[5] and .NET[6]). This paper proposes extending such platforms with AM specific services. This solution enables software applications based on such platforms to obtain inherent AM support. In this context, provided AM services must be seamlessly configurable, in order to conform to the particular administrative goals of each application.

The proposed approach facilitates the construction of AM solutions starting from elementary AM functions, provided by the middleware platform. Reusable AM middleware services offer a common base for building more complicated management behaviours. Eventually, middleware platforms are progressively enhanced and extended to provide increasing AM support, with growing functional complexity. The work presented in this paper was carried out to follow this approach. It consisted of extending a service oriented component platform – iPOJO, with common AM capabilities. These include various monitoring and effector *touchpoints*, as well as several *AM services*, completely integrated into the iPOJO platform. Figure 3 positions the AM facilities introduced with respect to the iPOJO component infrastructure. The figure shows how AM services are integrated into the iPOJO component membranes, just like any other middleware service that iPOJO provides. As any other iPOJO handlers, AM services can interact and collaborate with each other in order to combine their behaviours. Figure 3 shows how monitoring and effector touchpoints are provided at the component membrane level. Touchpoints obtain information and act upon a component's services, as well as on its AM service facilities.

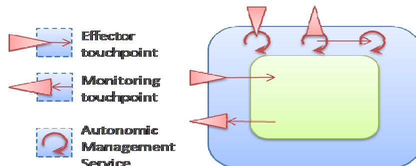

Figure 3: AM extensions for iPOJO components

Finally, Figure 4 shows how the AM facilities provided at iPOJO platform level are employed by external applications. Such applications may consist of administrative consoles, offering various runtime views of managed applications and the means to perform necessary reconfiguration operations. Provided iPOJO AM extensions are described in further details over the following subsections.

B. *iPOJO Monitoring Touchpoints*

In iPOJO applications, services may expose internal properties, whose values are used for service configuration purposes. Monitoring such properties allows notifying administrators of service configuration changes. For this purpose, iPOJO was extended with a *property monitoring touchpoint*, enabling the remote observation of iPOJO property values. iPOJO offers standard JMX[7] support for remote access to property values and update notifications.

Besides service properties, accurate runtime models of an application's architecture constitute a critical facility for system management. Nonetheless, the runtime model of an iPOJO application cannot be statically inferred from the individual implementations and configurations of its constituent components. The main reason is that the iPOJO platform resolves service dependencies during runtime (subsection II.C) and the resulting service interconnections are difficult to predict beforehand. Moreover, an iPOJO application's architecture may repetitively change during runtime, as various service providers become available or unavailable. For these reasons, an *Architecture touchpoint* was provided to allow system administrators collect information on iPOJO applications' runtime architectures (i.e. iPOJO instances and their interconnections). The Architecture touchpoint enables iPOJO components to expose their runtime states and current dependencies. For each component dependency, the published information includes: i) the dependency state (i.e. resolved, or unresolved); ii) the dependency configuration (i.e. mandatory or optional, simple or aggregated, using a certain service filter or binding policy); and iii) information on the currently used service provider (i.e. provider name, or location). This data provides sufficient information for inferring an application's runtime interconnection graph. The Architecture service communicates with other iPOJO middleware services, or handlers, in order to collect and disseminate information on their internal states.

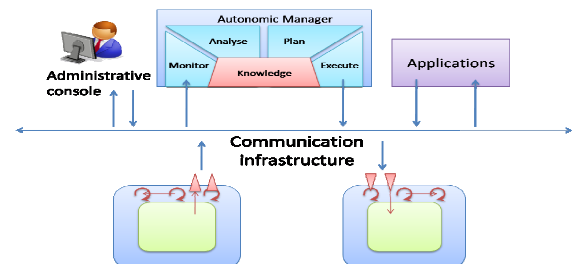

Figure 4: Utilisation of iPOJO AM facilities

C. *iPOJO embedded AM services*

Service dependency management is the most important AM service that iPOJO provides. A service dependency implies that the service declaring the dependency requires a provider that fits the dependency's specification (i.e. service type). If a service's dependency is mandatory, then the service cannot be validated until bound to a service provider that corresponds to the dependency requirements. iPOJO provides an embedded AM service for managing service dependencies. This service provides several autonomic functions, such as the dependency's initial configuration, runtime repair and dynamic optimisation. These functions are described as follows.

Initially, the dependency management service of an iPOJO component discovers the service providers that match the

[5] Sun Microsystems' Java EE technology: java.sun.com/javaee
[6] Microsoft's .NET Framework: msdn.microsoft.com/en-us/netframework
[7] Java Management Extension Technology: java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement

component's dependency requirements. Then the dependency manager selects one of the suitable providers and creates a binding to it, hence resolving the service dependency. During service execution, the dependency manager detects broken service bindings (e.g. the bound service provider becomes unavailable) and repairs them by discovering and binding to an alternative provider. The service discovery, selection and binding process (whether initial, or following a binding fault) is based on the following criteria.

Only service providers that meet the dependency's service specification can be bound for resolving a dependency. If multiple providers meet this criterion, the dependency manager selects one of these providers, depending on the dependency's configuration. Several parameters are available for tuning the selection procedure. A dependency's parameters may include a *filter* (LDAP-style property-value pairs), a *comparator* (an application-specific ranking function) and a *binding policy* (taking into account previous provider selections).

Besides the initial binding configuration and repair facilities, the dependency management service can be configured to continually optimise service dependencies during runtime. This is achieved via a special dependency filter, called *context filter*, which allows the use of context parameters in its specification. This filter is updated each time the corresponding context parameters are modified, consequently triggering the service dependency re-evaluation. This enables services to automatically optimise their dependency bindings with respect to their changing execution environments. This means that a service provider that was optimal in a certain context may be automatically replaced by a different service provider, which becomes the optimal choice in a new execution context. The context filter can be specified to trigger the dependency optimisation each time a new service provider with the required specification becomes available.

D. iPOJO effector touchpoints

iPOJO effector touchpoints are provided for modifying iPOJO components, as well as iPOJO AM services (Figure 3). The most important effectors are as follows. First, a *property effector touchpoint* enables the dynamic modification of iPOJO property values. iPOJO provides support for remotely accessing this touchpoint via JMX. This effector allows reconfiguring iPOJO instances during runtime. Next, a *dependency management effector* enables modifying any dependency parameter (e.g. optional/mandatory, aggregate/simple, binding policy and filter), with the exception of the dependency's service specification. For example, when a display service is required, users may prefer providers offering maximum display sizes and resolutions when they are alone (e.g. LCD screens) and providers ensuring maximum privacy when in a social context (e.g. PDAs). This can be achieved by dynamically changing the dependency filter of the service requiring the display, so as to follow the user's social context. Care must be taken when performing dependency modifications, in order to ensure service configurations remain coherent with service implementations. For example, a service that expects to use a single service provider may not function properly when configured to use an aggregated dependency.

Finally, a *lifecycle effector* enables operations such as service deployment, starting, instantiation, stopping and undeployment. This effector is provided at iPOJO platform level, while the property modification and dependency management effectors are implemented at the component container level. The lifecycle service allows the dynamic update of a service's implementation, if the service's undeploy and deploy operations are sequentially called. Client bindings to the updated service are automatically repaired by the clients' corresponding dependency managers.

IV. RELATED WORK

Most component and service technologies are beginning to offer monitoring and effector facilities via special-purpose interfaces, accessible via standard protocols (e.g. JMX), or via proprietary administrative consoles. In the SOC context, the importance of introducing management facilities in service applications is reflected by the publication of specific Web services standards, such as Web Service Distributed Management (e.g. [9]). Several research projects aim at enhancing existing component technologies in order to provide accurate architectural information during runtime (e.g. JADE [7] for the Fractal[8] technology, and [10] for Java EE servers). To the authors' knowledge none of the existing component or service technologies offers inherent support for dynamic composition and dependency resolution. Although certain technologies such as Jini[9], OSGi[10] or Web services were conceived to support such tasks, the actual implementation for supporting them is largely left to application developers.

Research work in the areas of multi-agent systems, self-organisation and emergent behaviours is the closest to address autonomic management of functional dependencies during runtime. For example, the Jack-in-the-Net [12] (or Ja-Net) framework uses multi-agent concepts and biology-inspired mechanisms to create adaptive services in large-scale, open network environments. In Ja-Net, self-organising Cyber Entities dynamically create emergent services, depending on network conditions and user preferences. Such approaches are complementary to the work presented here and can be considered for extending iPOJO technology. Nonetheless, many of the implementations available in these areas are currently in the prototyping phase, while iPOJO is a mature, extensible technology, available as an open source platform.

Certain researches on reusable middleware platforms for pervasive computing have similar objectives with the presented work. For example, [11] proposes a reconfigurable, context-sensitive middleware for component-based pervasive applications. This platform exclusively focuses on context-awareness and ad-hoc communication aspects. It presents interesting concepts that could be considered for enriching

[8] Fractal Project : fractal.objectweb.org
[9] Jini Technology : incubator.apache.org/river
[10] OSGi Alliance : www.osgi.org

iPOJO containers with self-configurable, context-aware AM services. Nonetheless, the research presented here has a larger scope, envisaging the integration of multiple Autonomic Computing concerns into iPOJO technology. iPOJO is a highly extensible and configurable platform, allowing the introduction of new middleware services and the creation of customised component containers.

V. Discussion on Social Impact

The development of pervasive computing applications and their adoption by the general public has an increasingly strong impact on our social environment. Autonomic Computing offers the key for rendering pervasive applications accessible to a large public, requiring minimum administrative effort and least technical knowledge. It aims at allowing non-expert users seamlessly install and administer pervasive applications, as well as create new applications from existing functionalities. This means that computing systems will no longer remain restrained to expert engineering environments and will instead become available to the general public. Service oriented technologies with inherent AM functions provide the essential support for implementing this vision. This section describes several sample scenarios that indicate the type of interactions supported by pervasive environments with AM capabilities.

Several service types are generally employed for assembling a large range of pervasive applications. Such applications typically include input services (e.g. sensors, consoles, or remote Web services), data-processing services (i.e. services that implement application-specific business logic) and output services (e.g. LCD displays, alarms, or messaging services). Based on such services, AM platforms allow the seamless creation, control and evolution of context-aware applications and the integration of virtual information into the real world. iPOJO provides AM support by handling inter-service dependencies, service reconfiguration, repair and optimisation.

A simple service reconfiguration example may consist of a lamp that modifies its light colour depending on weather information from a remote Web service. In this example, iPOJO automatically finds and connects the lamp driver to a weather service, locates alternative weather providers in case the initial connection is broken and provides the means of modifying the value of the lamp colour parameter. In a more sophisticated example, a video camera, image-recognition service and sound system may be combined with various specific services for creating different applications. Resulting services may involve playing ambient tunes when a home owner enters the living room, or sounding an alarm when a suspect intrusion is detected. In this scenario, customers purchase the various devices and software packages from different providers. Based on these services they create the two applications by simply deploying all services on an autonomic SOA platform, such as iPOJO and specifying their high-level preferences. Intuitive administrative consoles allow non-technical users to add and remove services from the underlying platform and to express their preferences with respect to service behaviours. Considering the alarm application example, a deployed video driver automatically finds and connects to an intrusion-detection service, which discovers and binds to a sound system driver. The local iPOJO platform automatically manages service dependencies and may periodically connect to external provider websites for updating relevant driver services [3].

Over time, customers may decide to replace certain devices and services of their applications (e.g. purchase a better video camera, or a bigger LCD screen). iPOJO automatically updates the concerned applications in order to use newly detected devices and optimise applications with respect to user preferences (e.g. use the biggest display appliance available). In case a new device breaks down and becomes unavailable, iPOJO automatically repairs concerned applications by switching back to previous configurations (i.e. employing the old device models). The administrative operations described require minimal human intervention and no specific development effort from service providers. iPOJO platform offers the necessary AM services by default.

iPOJO is a service component platform offering built-in AM support. This paper described the most important autonomic features that iPOJO currently provides. As an extendible platform, iPOJO will be progressively enhanced with additional AM features, so as to render application creation and management easier, faster and more dependable.

References

[1] Mark Weiser, “The computer for the 21st century”, *Scientific American*, 265(3):66-75, September 1991

[2] C. Escoffier, J. Bourcier & P. Lalanda. “Toward an Application Server for Home Applications”, *5th IEEE Consumer Communications and Networking Conference (CCNC’08),* Las Vegas, USA, 2008.

[3] A. Bottaro, J. Bourcier, C. Escoffier and P. Lalanda, “Context-Aware Service Composition in a Home Control Gateway”, *4th IEEE International Conference on Pervasive Services*, Turkey, July 2007

[4] C. Escoffier, R. S. Hall & P. Lalanda. “iPOJO An extensible service-oriented component framework”, *IEEE International Conference on Service Computing (SCC'07),* Salt Lake City, USA, 2007

[5] C. Escoffier & R. S. Hall. “Dynamically adaptable applications with iPOJO service components”, *6th Conference on Software Composition (SC’07)*, Braga, Portugal, 2007

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

[7] S. Sicard, F. Boyer, and N. D. Palma, “Using components for architecture-based management: the self-repair case”, in Proceedings of the 30th *International conference on Software engineering (ICSE ’08),* Leipzig, Germany, May 2008, pp 101–110

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

[9] H. Kreger and T. Studwell, “Autonomic computing and Web Services Distributed Management”, IBM online article: www.ibm.com/developerworks/autonomic/library/acarchitect, 2005

[10] Trevor Parsons and John Murphy: "Detecting Performance Antipatterns in Component Based Enterprise Systems", in *Journal of Object Technology*, vol. 7, no. 3, March - April 2008, pp. 55-90

[11] S. S. Yau et. al, “Reconfigurable Context-Sensitive Middleware for Pervasive Computing”, *IEEE Pervasive Computing*, 2002, pp 33-40

[12] T. Itao et. al, “Service Emergence based on Relationship among Self-Organising Entities”, in Proceedings of *IEEE Symposium on Applications and the Internet*, January 2002, pp 194-203