Journal of Universal Computer Science, vol. 20, no. 3 (2014), 431-460
submitted: 24/3/13, accepted: 4/2/14, appeared: 1/3/14 © J.UCS

Multilevel and Coordinated Self-management in Autonomic Systems based on Service Bus

Mohamed Zouari
(CNRS, LAAS 7 avenue Colonel Roche F-31400 Toulouse, France
Université de Toulouse, INSA, LAAS; F-31400 Toulouse, France
mohamed.zouari@laas.fr)

Codé Diop
(CNRS, LAAS 7 avenue Colonel Roche F-31400 Toulouse, France
Université de Toulouse, INSA, LAAS; F-31400 Toulouse, France
code.diop@laas.fr)

Ernesto Exposito
(CNRS, LAAS 7 avenue Colonel Roche F-31400 Toulouse, France
Université de Toulouse, INSA, LAAS; F-31400 Toulouse, France
ernesto.exposito@laas.fr)

Abstract: Modern dynamic distributed systems require to dynamically take into account at runtime the changes in users' needs and the execution environment variations in order to improve the quality of service. The evolution of distributed systems, through the smart management of their properties and the extension of the existing integration infrastructures, becomes a necessity. Autonomic computing allows the self-management of system properties at runtime, according to fluctuations in the environment and changes in users' requirements. However, the mechanisms for parallel and distributed execution of multiple self-management processes have not been addressed substantially. It is critical to coordinate the execution of several processes performed by different autonomic managers, while still guaranteeing specific and global goals achievement. We address this issue by proposing a software architecture that allows the coordination of multiple autonomic managers which handle several component-based and service-oriented collaborative software entities. This architecture offers a distributed cross-layer self-management solution through orchestration and choreography. Using both techniques, autonomic managers running on multiple locations and different layers will be able to achieve their goals in a consistent and cost-effective way. In this paper, we present a set of mechanisms intended to coordinate the distributed execution of a set of self-management processes in one or more layers. We have chosen an use case involving the self-management of autonomic data replication systems integrated via an autonomic service bus in order to illustrate our approach.
Key Words: autonomic computing, quality of service, enterprise service bus, distributed and coordinated management
Category: D.2.9, D.2.11, D.2.12

1 Introduction

The management of distributed applications and more generally distributed systems is a complex and critical process. In fact, these systems run on distributed infrastructures that are subject to important variations. For instance, telemedicine applications provide services to ensure remote health care delivery for patients and the collaboration of remote and mobile caregivers. The execution environment of these applications is characterized by the diversity of terminals, changes in computing and communication resources availability, and different network connection conditions. These kinds of applications imply also users that have different quality of service (QoS) requirements according to their profiles and preferences. In this context, the configuration change of application features (e.g., software components behaviour, their connections and their distribution) at runtime enables the system to deal with different fluctuations in available resources, to meet new user requirements, and to improve the QoS.

Moreover, these distributed systems are developed more and more following a service-oriented architecture (SOA) approach and the Enterprise Service Bus (ESB) is used to allow the integration of pervasive, distributed and networked systems, which are a composition of heterogeneous services and software components. Consequently, the ESB may integrate a large number of parallel and concurrent systems. In the context of very active and dynamic environments where new systems and services can be increasingly deployed within the integration environment, overload situations for the internal components and resources of the bus may occur. In such situations, the bus becomes a bottleneck that leads to QoS degradation. Thus, the bus needs to be able to deal with these issues by the dynamic reconfiguration and optimization of communication mechanisms.

In this context, the self-management property (i.e. self-configuring, self-optimizing, self-healing and self-protecting) represents a solution for software systems running in fluctuating and heterogeneous environments. In fact, the autonomic computing approach [Huebscher and McCann 2008] accelerates the accomplishment of more appropriate system configuration and ensures cost savings through the reduced dependence on human intervention. In general, an autonomic manager monitors the execution context (resources availability, terminal capabilities, connection quality, user activity, etc) in order to trigger dynamic reconfiguration whenever it predicts or detects an undesirable situation. It makes decisions regarding the target goal and controls the modification of the system components in order to achieve the appropriate configuration.

A single autonomic manager may satisfy the self-management requirements when the target application is composed of small number of nearby software components in homogeneous environments. However, when a decentralized application or several inter-dependent applications and services are running in heterogeneous environments, distributed management mechanisms are needed to improve their quality such as efficiency and scalability. The distributed management leads to the concurrent execution of multiple reconfiguration processes performed by different autonomic managers. Conflicting decisions and inconsistent changes may occur if the interdependencies among the applications' components are not considered. Therefore, coordinating the activities of multiple autonomic managers is critical in order to preserve the overall consistent state of the applications.

Currently, the self-management mechanisms in existing systems are generally implemented in ad-hoc way. The existing approaches do not allow to coordinate the activities of multiple autonomic managers based on distributed coordination. In addition, it is not possible to easily customize the coordination capabilities. Furthermore, the existing approaches do not enable to add autonomic capabilities in several layers like the business services layer and the integration layer with aim of reconfiguring the appropriate entities in one or more layers in a coordinated and cost-effective way. In order to overcome these problems, we propose an approach to define a software architecture for distributed and coordinated self-management mechanisms for dynamic software reconfiguration. This architecture supports both the orchestration and the choreography of multiple self-management processes.

We define a software architectural model of autonomic managers that allows the variability of their configuration and provides flexible coordination mechanisms. This model abstracts distributed and coordinated self-management mechanisms in a modular way and facilitates their specialization during the development process. We use this model to set up several autonomic managers that manage the business services and the bus communication services. In this way, it becomes possible to reconfigure dynamically the business services to improve their QoS and to rely on the autonomic capabilities of the bus in order to optimize and make scalable the communication mechanisms. Our approach permits to change the configuration of the business services, the bus configuration, or both of them imposed jointly. Our case study concerns the data management in medical environments for collaborative remote care delivery.

The remainder of the paper is organized as follows. Section 2 introduces our case study. Section 3 analyses related work. In section 4, we present our architectural model of distributed autonomic systems with focus on the coordination mechanisms. We introduce the approach to develop an autonomic replication system and we explain how to make an associated service bus autonomic in Section 5. A scenario of distributed self-management for QoS improvement is described in Section 6. Section 7 gives an overview of the prototype and its evaluation. Finally, we conclude and discuss future work in Section 8.

2 Case study

Our case study is related to the healthcare delivery within and across organizational boundaries and the solutions for evolving healthcare systems towards a

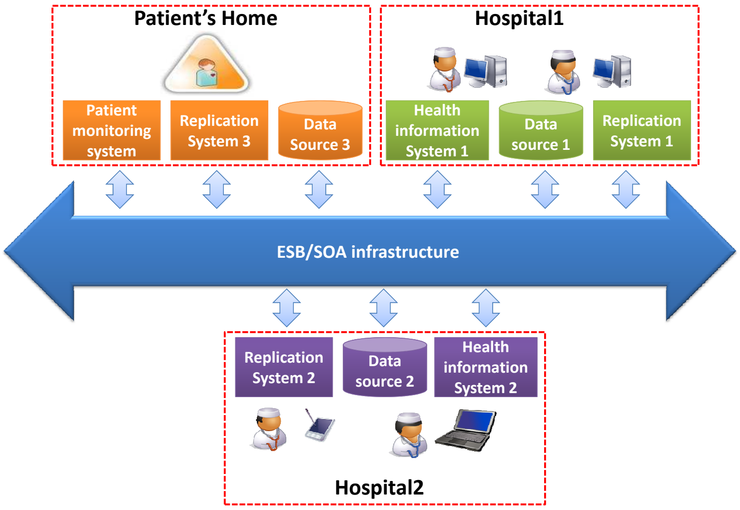

Figure 1: Integrated systems for remote collaborative health care delivery

service-oriented architecture using the enterprise system bus middleware technology for resolving integration and performance issues. We improve the availability of data and response times for data requests using data replication systems. We make the replication systems autonomic to improve their functionalities and their quality characteristics.

We consider a set of services for remote health care delivery. They enable timely remote access to the health status of patient having chronic disease and provide him with the appropriate interventions. In addition, they offer means for the collaboration of caregivers working at different locations (distant hospitals, mobile emergency team, etc). Successful health care relies on collaborative care that requires a broad network of collaborative interactions among geographically distributed caregivers. They may share information from different sources to treat a patient. Having accurate and timely information is vital to provide efficient care. Consequently, adding autonomic capabilities to replicated data management systems may lead to significant improvement of quality of service in terms of data availability and access latency.

Our goal is to improve the exchange of electronic healthcare information and sharing information stored in various healthcare systems and hospitals while preserving specific goals of individual systems and also collaborating for the satisfaction of global health systems' goals. Figure 1 shows the integrated systems that provide the remote healthcare services. The ESB infrastructure integrates healthcare systems, the home patient monitoring system, the replication systems and the data sources. It provides the basis for the exchange of electronic health data and meets the integration needs. The replication system *Replication System 3* on the patient's home, replicates data provided by the monitoring system over the WAN. *Replication System 1* and *Replication System 2* replicate patient's data handled by the care givers respectively in hospital 1 and 2.

3 Related work

An important number of research works has been carried out in the area of self-management in distributed systems and services as well as the coordination in multi-agent systems.

3.1 Autonomic Computing and ESB

In the last few years, a number of surveys have evaluated the state of research efforts in the autonomic computing field [Salehie and Tahvildari 2005, Parashar and Hariri 2004, Nami and Bertels 2007, Khalid and al. 2009]. They provide a review of architectures, frameworks, techniques, and challenges in Autonomic Computing (AC).

One of the first and most important contribution is represented by the AC approach initially proposed by IBM in order to face the increasing complexity of manual management of information technologies [Kephart and Chess 2003, Horn 2001, Huebscher and McCann 2008]. The AC paradigm proposes a specialized architecture based on a set of well defined components and interfaces as well as a precise specification of the individual and collective expected autonomic behaviour of the diverse and potentially distributed system components. The most basic autonomic composition is based on one managed element (ME) associated with one autonomic manager (AM) in order to build one elementary autonomic element (AE).

For more complex compositions involving several potentially distributed ME, a hierarchical management model has been proposed in [Ac 2006] based on: (1) a single resource management where one AM manages one ME, (2) multi-properties management of a single resource where several AMs manage the various properties of one ME (i.e. self configuring, self-optimizing, etc), and (3) multi-elements management where one AM manages a homogeneous or heterogeneous set of MEs. Similarly, at higher levels of an autonomic system, hierarchical compositions of autonomic elements and autonomic managers can be defined. In these compositions, each managed AE offers the sensor interface and implements the effector interface. A higher level AM manages the lower level AE by implementing the MAPE (Monitor, Analyse, Plan, Execute) functional phases and communicating with the AE via the touchpoint interface (composition of both sensor and effector interfaces).These high level autonomic managers are called orchestrators. At lower or higher levels of an autonomic system, the overall self-management functions as well as the MAPE activities performed by basic autonomic managers orchestrators are fundamentally based on its knowledge base and the policies guiding its autonomic behaviour.

The generic architecture promoted by AC, has also been applied in the area of networked systems and in particular in the self-management of network resources and technologies. In this area, the autonomic networking (AN) approach is aimed at reducing the complexity involved on network resources and services management face to the increasing diversity of heterogeneous networked devices, distributed applications, components, services and network technologies [Strassner and al. 2006, Dobson and al. 2010]. [Exposito 2012] proposes in this domain 4 levels of autonomic orchestrators in order to provide the basic coordination functionalities required by an Autonomic Transport Layer: stream-level, application-level, system-level and group-level. For instance, at the application-level, managers aimed at orchestrating the several streams of the same application taking into account the application requirements or preferences. Likewise, [Strassner and al. 2009] proposes an autonomic orchestration based architecture for a Future Internet by mapping business goals to network services in order to dynamically adjust distributed services and resources. In the context of ESBs, [Morand and al. 2011] proposes an integration model called Cilia. The main goal of this approach is to make autonomic the integration process, with the aim to tackle scalability issues or execution contexts evolutions. The theory of control is used with a set of state variables and action variables in order to allow following the state of the integration contexts. Autonomic managers are introduced to adapt the integration process if needed. In [Gonzalez and Ruggia 2012], authors propose ESB-based solutions that can be applied at runtime to address some QoS issues in service-based systems. They propose an adaptive ESB infrastructure through which, for each mediation service, an adaptive service is proposed with a dynamic behaviour that depends on the runtime information. However, the orchestration technique may be not suitable in many cases, especially when (1) the management processes performed by autonomic managers may scale to a high number of autonomic managers (2) the opacity of management process details is desired in the management of systems belonging to different organizations, (3) the different management processes require their own customizations performed by experts in many application domains and technologies, and (4) the different self-management processes are highly dynamic or goal-seeking.

The choreography and orchestration technology within a SOA is a popular technique for composing services and ensuring the compatibility of interacting processes. Some works consider Autonomic Computing together with Web Services to avoid manual configuration in the process of composing services into complex applications. For instance, [Delamer and Lastra 2006] enables to automate the composition of service-oriented industrial applications in order to emancipate the factory floor from the timing and monetary constraints of manual programming efforts during deployment. Other approaches have focused on generating a distributed choreographed implementation from a logically centralised orchestration specification. [Mostarda and al. 2010] describes an approach where given a logically centralised service orchestration, it automatically generates a distributed implementation that correctly enforces the orchestration behaviour. A similar approach was implemented in CiAN, a choreography-based workflow engine [Sen and al. 2010]. [Geiger and al. 2011] deals with issues to check the conformance between ebBP-ST (a subset of ebBP) choreographies and corresponding WS-BPEL based implementations. Unfortunately, such approaches do not address the problems of distributed and coordinated self-management processes. Therefore, their interest for coordinating autonomic managers over heterogeneous environments or in several layers is limited.

3.2 Multi-agent negotiation mechanisms

Multi-agent systems have been widely used for the analysis, the modelling and the development of complex and distributed computer systems. An agent is an autonomous and intelligent entity that is capable of acting on itself and on its environment in a multi-universe agent and can also communicate with the other agents. Its behaviour is the consequence of its observations, its knowledge and the interactions with the other agents [Ferber 1995]. With regard to the distributed problem solving [Durfee and Lesser 1989], a multi-agent system is defined as a loosely coupled network of agents working together as a society to solve problems that are generally beyond the scope of any single agent. Recently, a growing number of multi-agent systems have been developed in different domains such as TRAINS [Allen and al. 1995], HOMEY [Beveridge 2001], PARMA [Greenwood and al. 2003], Workflakes [Valetto and Kaiser 2003], etc. Most of these systems are centralized in the sense that they are based on a mediation agent, which can cause a bottleneck. For instance, Workflakes has been developed as an execution environment for the system reconfiguration within the autonomous platform Kinesthetics [Valetto and Kaiser 2003]. Workflakes is constructed as an extension of the agents that execute programs called worklets which perform local reconfiguration. An engine allows the coordination of the agents in order to control the execution of the various reconfigurations.

The researchers agree on the purpose of the negotiations which is the result to a satisfactory common agreement. There are many similar definitions in this area. One of the most basic and succinct is made by Bussman and Muller [Muller 1996]: "Negotiation is the communication process of a group of agents in order to reach a mutually accepted agreement on some matter". All mechanisms for negotiation are based on the exchange of offers. Agents make offers that they find acceptable and respond to offers proposed to them. Several works have been proposed [Jennings and al. 2001, Rahwan and al. 2004, Ramchurn and al. 2003, Amgoud and al. 2004] in order to provide agents with the ability to hold such dialogues. The approaches rely on the game theory and the heuristics and are mainly focused on the numerical estimation of offers in terms of utilities. An important limitation of these works is that the utilities or preferences of agents are usually assumed to be fully characterized before the interaction. Thus, an agent is assumed to have a mechanism by which it can allocate and compare two proposals. In addition, it is hard to change the set of issues under negotiation, and goals of the agents are assumed to be fixed.

3.3 Synthesis

Distributed and multilayer systems with autonomic capabilities require decentralized coordination of their self-management processes, with local coordination rules between autonomic managers.

To the best of our knowledge, no systematic study has been performed on the choreography of several autonomic managers in order to provide the coordination function, in particular by enhancing AC architecture with the required components and logic to allow distributed self-management based on both orchestration and choreography. There is no clear view on what methods could be used to provide the self-management on distributed and multilayer systems without a central control actor.

Moreover, the configuration variability of collaborative autonomic managers were not considered. Consequently, there is a need for reducing the efforts of design, implementation and customization of mechanisms to support collaborative self-management processes involving multiple choreographed or orchestrated autonomic managers and especially their interactions seen from a global perspective.

4 Architectural model for distributed self-management processes

Our approach aims at facilitating the building of autonomic distributed systems. We design generic self-management mechanisms in a modular way and we permit their specialization according to the target application. We provide an architectural model that specifies types of software components composing the system, the variation in system configuration and constraints to be respected by all autonomic managers. It is difficult to identify all types of constraints that developers would have to respect. Nevertheless, we express several constraints enforcing architectural styles, design patterns and modelling rules. The constraints concern mainly the authorized connections among components, the exchanged data and the way to customize them.

An administrator who is expert in the autonomic management, provides a description of the autonomic managers architecture in keeping with the model. The description specifies the instances of autonomic managers with the values of their configuration parameters, the connections among them, and the connections with the managed elements. This section gives a global view of our model and details the mechanisms for the execution of self-management process and for the coordination of autonomic managers activities: on one hand the orchestration/choreography of distributed decision making processes and on the other hand the choreography of the distributed reconfiguration control processes.

4.1 Overview of cooperative autonomic manager architecture

A software system can be seen as a set of collaborative components providing services. Some components may expose control interfaces that provide primitive operations to observe and reconfigure them (e.g. read/modify a configuration parameter value). The autonomic managers are connected with the system trough these interfaces.

In the original AC architecture [Horn 2001], an Autonomic Manager is a composite that implements 4 major functions: monitor, analyse, plan and execute. Our architectural model specifies the types of autonomic manager's components necessary to guarantee an efficient distributed management. A software system may include several instances of autonomic manager. The scope of every manager activity is limited to a subset of system components. A subset consists of components that collaborate with each other in order to provide one or several specific services and/or are placed close geographically. Our model supports the orchestration and choreography of self-management processes. For this purpose, we define interfaces and components to ensure the coordination of autonomic manager activities with homologous components. Figure 2 presents the architectural model of this type of component. For clarity, it does not show all the dependences among components and the connections to the knowledge base. As a consequence, self-management processes are created from composite services performed by these components. Orchestration represents control from one party's perspective and tracks the message exchange in specific process that an autonomic manager executes. This process can interact with external services performed by other autonomic managers. The interactions occur at the message level. They include management logic and task execution order. They can span applications and organizations to define a long-lived, transactional, multistep process model. This differs from choreography, which is more collaborative and allows each involved autonomic manager to describe its role in the interaction.

Actually, the management is expressed in choreography as a direct communication between services provided by autonomic managers without any central actor, making it scalable.

Our model specifies the types of sub-components for the system management: *Monitor*, *Decider*, *Planner*, *Executor* and *Knowledge Base*. An autonomic manager is a composite that contains a single instance of each type. In fact, a *Monitor* collects, interprets and aggregates some contextual data. The data is stored in the *Knowledge Base*. The collected information is about the application and its runtime environment and is provided by physical sensors (e.g. a camera to locate a patient at home) and the control interfaces of application components (e.g. an interceptor to calculate the data access frequency). The *Decider* is responsible of decision making (the analyse phase) and it returns a reconfiguration strategy (the goal). The strategy specifies the appropriate changes to the actual application configuration (e.g. changing the replica placement algorithm). As described below, the choice of strategy may involve other deciders and negotiators or not.

A publish/subscribe mechanism ensures asynchronous interaction: a *Decider* subscribes to symptoms that may need reconfiguration (*SubscribeItf* interface) and the *Monitor* notifies the subscriber when an appropriate change of context is detected (*NotifyItf* interface). Moreover, a decider can query a monitor for specific contextual information in request/response mode (*MonitorItf* interface). For instance, a monitor in a patient home could notify the symptom

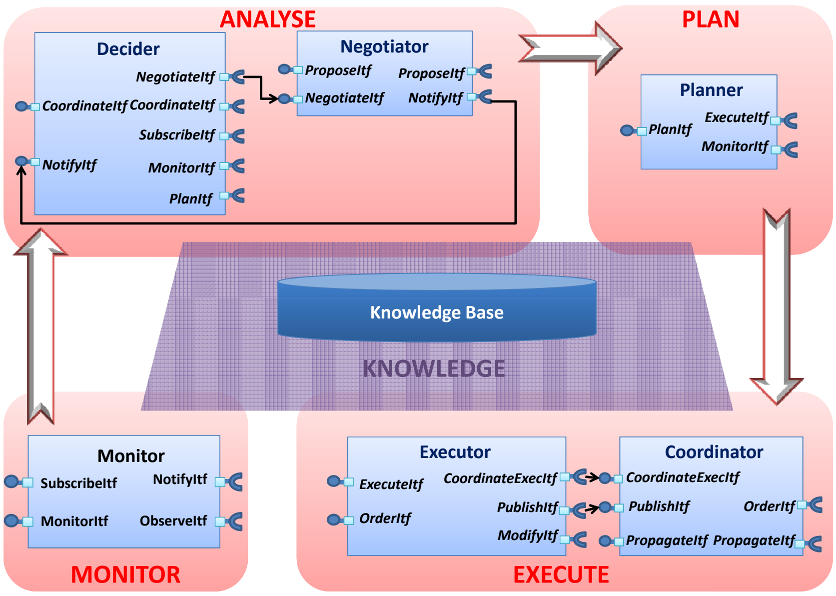

Figure 2: Architectural model of a cooperative autonomic manager

"EmergencyEvent" (*notify* operation) to a subscribed decider in the event that a patient is in need of emergency medical help (see Figure 3). In such a case, the

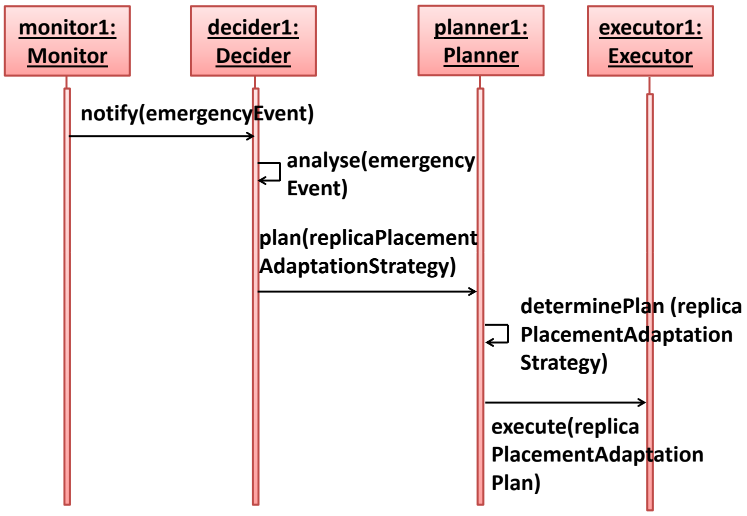

Figure 3: Example of reconfiguration strategy adopted when an emergency occurs

decider may choose (*analyse* operation) a strategy type that defines reconfiguration strategies for a component managing the placement of replicas (Figure 4). In this type, it is possible to define the name of the placement algorithm and its configuration parameters as well as the node that hosts the component instance. A reconfiguration strategy for applying a random placement algorithm with 10-fold replication, requires to initialize the attribute *placementAlgorithmName* with *RandomPlacementStrategy* and the attribute *algoParams* with *10*.

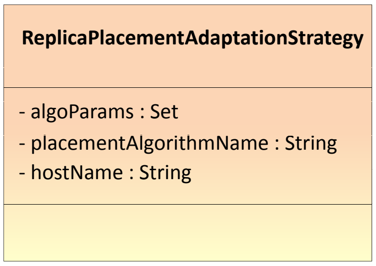

Figure 4: UML class representing a type of strategies for replica placement component

The achievement of the strategy is ensured by a *Planner* and an *Executor*. The first identifies a sequence of reconfiguration actions (e.g. replace a component, switch to new algorithm, allocate new resource, etc) in the form of a plan to apply the strategy chosen by the *Decider*. The second one controls the application of the plan that it receives (*ExecuteItf* interface). For that, it interacts with effectors associated with the managed elements through the touchpoints. For instance, a planner may select and configure a plan (*determinePlan* operation) that modifies the algorithm implemented by a placement component in a replication system (see Figure 3). Figure 5 shows this plan. It comprises a first adaptation action *modifyPlacementAlgorithm* to modify the algorithm and a second action *configureParameters* that configures its parameters (for example, the attribute that specifies the desired number of replicas).

Finally, the *Knowledge Base* stores all relevant information about the problem domain and provides the components with a solid basis of knowledge. For instance, it contains the measured monitor values (e.g. the network status, the performance indicators of the running services, the patient activity), the policies that customize the autonomic manager behaviour, parametrizable data exchanged between sub-components like the strategies and the plans, etc. The base offers basic querying capabilities. It exposes an interface to read, write, update, and delete the different types of data. This interface is used by each sub-component of the considered manager. The contextual dataset is captured by the sensors and then managed by the monitors. The monitored data is stored in the base. This data can be interpreted to generate more significant information and is accessed when necessary like during the evaluation of execution context to choose the reconfiguration strategy. In addition, the knowledge base is used to record history data about reconfiguration processes such as negotiation failure, detected symptom or reconfiguration delay. This data enables further processes to make more intelligent decisions or to manage events resulted from the self-configuration activities. On the other hand, a reconfiguration expert stores at deployment time the different policies and defines the possible strategies, plans,

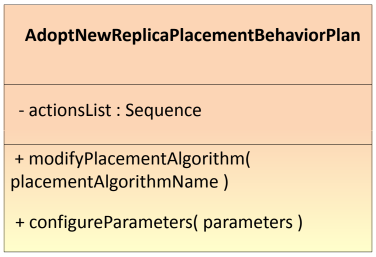

Figure 5: UML class representing an adaptation plan for replica placement component

and contracts. Then, the components access to them at runtime. For instance, the analyse process necessitates to interpret the reconfiguration policy. The decision making resulted in the instantiation of the appropriate type of strategy and the customization of the instance (defining the attributes values).

We notice that the monitoring mechanisms are not detailed in this paper since there is no need for coordination during this phase even if the monitoring is distributed.

Concerning the coordination capabilities, this architectural model supports the coordination of autonomic managers' activities: the decision making (Analyse feature) and the control of the configuration modifications (Execute feature). First, the coordination of decision making aims to enable a group of autonomic managers to make a collective decision in order to ensure non-conflicting and complementary decisions. Our architectural model defines specific interfaces for the component *Decider* and a new component type *Negotiator* so to enable a decider to take part in a coordination process. Second, the plan execution coordination addresses the problem of applying several plans in parallel by a group of autonomic managers when plans contain dependent reconfiguration actions. In such a case, the execution control of these plans must be coordinated in the sense that a specific sequence of their actions execution must be performed to achieve a consistent configuration of the managed elements. Each autonomic manager must include an additional sub-component called *Coordinator* and the *Executor* has optional interfaces to interact with its associated coordinator.

Some of these components expose configuration parameters. In fact, an administrator specializes the behaviour of the components *Decider*, *Planner*, *Negotiator*, and *Coordinator* by giving a value to parameter that references a management policy. Actually, we adopt a policy-oriented approach where the management logic is described as external policies separated from the services implementation. These policies are stored in the knowledge base.

Decoupling the analyse feature (resp. the execute feature) into two components can result in a number of residual benefits. First, the responsibility and dependencies of each component may be well defined and easily understandable. Second, the singularity of purpose of each component renders the overall system easier to customize by different policies and to evolve. Third, the focus on a single purpose leads to components that can be implemented differently in several contexts and managed by disparate development teams (e.g. a decider implemented as a learning system and a negotiator based on a heuristic solution).

The components type *Monitor*, *Decider*, *Planner* and *Executor* are designed as mandatory elements. However, the components type *Negotiator* and *Coordinator* are optional components. They are not instantiated if there is no need for the autonomic manager to coordinate its activities with other managers. Each managed entity is controlled by a single autonomic manager. The autonomic manager does not handle multiple self-management processes at the same time. As a consequence, the analyse process begins by a new detected symptom after the achievement of reconfiguration plan executed by the previous reconfiguration process.

In the following sections, we will focus on the coordination mechanisms of decision making activities.

4.2 Mechanisms of decision making coordination among autonomic managers

Making decisions collectively results in the determination of a global strategy. We define a *global strategy* as a set of strategies that are chosen and applied by several autonomic managers. The decision must be relevant to the current global situation and has to predict the effect of reconfiguration strategies to avoid anomalous situations.

Coordinated decision making is performed in three steps: (1) retrieve the required context information, (2) choose a reconfiguration strategy, and (3) initiate a coordination process. We assume here that one manager triggers the coordination process. It is called the *coordination initiator*. Step 3 involves the deciders of other managers called the *participants*. During this stage, deciders may need to retrieve context information and make a reasoning to determine the definitive global strategy.

The deciders can interact with each other. The interface *CoordinateItf* allows a decider to communicate messages for an arbitrary number of other deciders. The reconfiguration policy must define when the interaction occurs, the involved deciders and the message content. Moreover, an optional component type *Negotiator* can be instantiated and interconnected with the decider to allow the negotiation of strategy. The Decider and the Negotiator components enable the autonomic manager to apply three coordination patterns described below.

- *The master-slave pattern*: Within this pattern, the coordination initiator can behave as a master, and considers the participants as slaves which follow its orders. The master chooses individually a strategy. Then, it assigns the task of applying a specific strategy to each participant. The externally and internally message exchanges of each slave as well as its internal state are not visible to the master. The coordination leads to the peer to peer externally observable interactions that happen between the master and the slaves during a decision making phase.

- *The strategy publishing pattern*: This pattern is based on the notification by the initiator to the participants of the strategy it chooses for itself. Once a participant notified, it analyses its environment and selects an appropriate strategy, taking into account the decision made by the initiator. The chosen strategy must be consistent and complementary with the initiator strategy.

- *The strategy negotiation pattern*: In this case, a decider can initiate a negotiation using the interface type *NegotiateItf*. The components type *Negotiator* included in different autonomic managers are connected each other through interfaces type *ProposeItf*. During the negotiation process, the negotiator interacts with homologous components included in the involved autonomic managers (the participants) by exchanging contracts. A negotiation policy specializes the behaviour of the component by specifying the participants and control rules of the progress of each negotiation process.

  In fact, when a decider requests the negotiation of a strategy, the negotiator creates an adaptation contract and initiates the negotiation with a set of participants. The contract is an object that specifies the initiator, the participants and global strategy to be negotiated. This is specified as parameters where each parameter specifies a local strategy, the manager responsible for applying it and negotiators involved in the negotiation (see Figure 6).

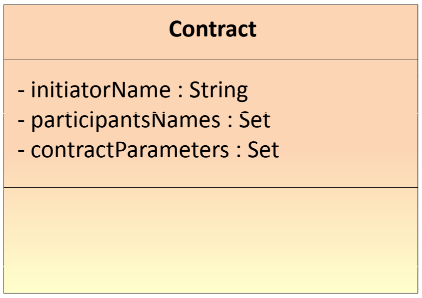

Figure 6: UML class representing the attributes of a negotiated contract

The diagram presented in Figure 7 describes the sequence of messages for negotiation of a contract between an initiator and participants. For clarity sake, only one participant is shown.

The initiator decider chooses a reconfiguration strategy. Next, it uses its *NegotiateItf* interface and asks the negotiator to negotiate the strategy that it has chosen. This negotiator constructs the contract. The initiator uses the appropriate *ProposeItf* interfaces to offer each participant the contract concerning it, in parallel.

Each participant negotiator receives the contract and interprets its policy to reason on its applicability. It can then accept, refuse or offer/request a contract modification and then, it responds to the initiator. When the initiator

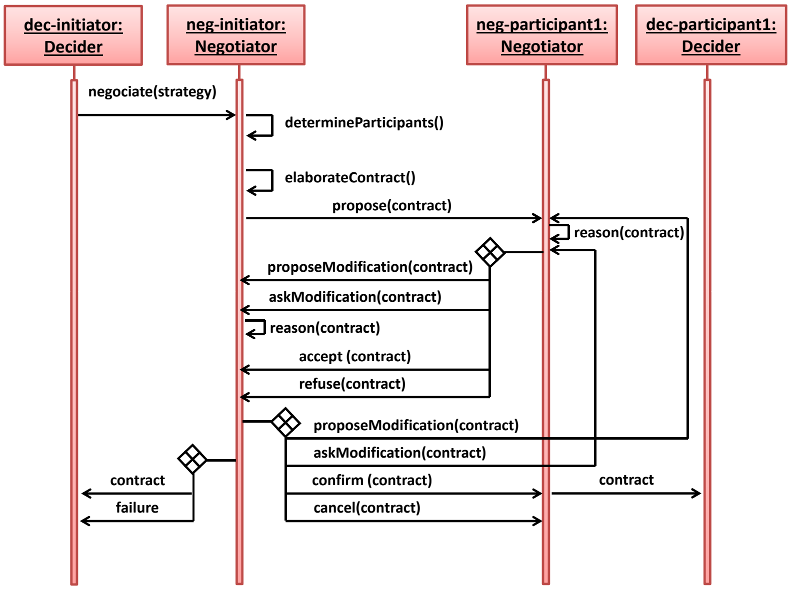

Figure 7: Sequence diagram of the negotiation between two managers

receives all the responses, it reasons on the acceptance and/or applicability of the modifications requested or proposed. When all participants accept the contract, the negotiation succeeds. In the opposite case, it detects and resolves the conflicts, and can then itself propose/request a contract modification. The negotiation process is stopped if a negotiator refuses a contract or if a stopping condition is verified. This condition can be related to the maximum negotiation time authorized, or the maximum number of negotiation rounds.

If the negotiation succeeds, the initiator negotiator returns the contract resulting from the negotiation to the initiator decider and sends also the final contract to each participant negotiator. Upon receiving this contract, the participant negotiator uses the *NotifyItf* interface to request the decider to apply the strategy resulting from the negotiation. In the opposite case, the initiator decider and the participants are informed of the failure of the negotiation. It is possible to consider that the reconfiguration strategy is not applicable and to stop the decision making process. Another alternative consists in considering this failure as a new symptom. So, a new management process is launched to find more relevant strategy. The choice of solution is done during the definition of policies that customize the behaviour of the components according to considered execution context. In general, there is a need to start a new management process if the modification of the current configuration is critical.

During the customization phase, specific roles (master, negotiator, etc) are assigned to autonomic managers through the specified policies. The manager may have a single role in all the self-management processes or may play multiple roles depending on the context changes and the selected reconfiguration strategies. The responsiveness of autonomic managers and the use of computing and network resources may be critical criteria to be considered when defining the different roles.

5 Mechanisms for self-management of distributed and integrated systems

5.1 Self-management mechanisms at the business level

We have designed an architectural model of replication systems that defines the types of components composing a data replication system, the message flows between them, their configuration parameters, and the control interfaces to observe a system and modify its configuration. The component-based approach leads to modular and evolvable software entities. So, the replication system enables fine-grained management operations. Moreover, the service orientation brings additional flexibility since components can be dynamically loaded, unloaded and interconnected. It is also possible to rely on the bus to reconfigure legacy replication systems by intercepting the data flows and make transformations or redirections. This approach facilitates the configuration management of such system and gives high flexibility. The same approach may be followed to manage other types of systems.

Our architectural model defines five component types for building a replication system (Figure 8). A replication system includes one instance of each type. First, the *PlacementManager* manages the replica placement. It receives requests from users for the creation/deletion of data. For each request, the component determines the replication scheme (number of replicas and nodes where the replicas are placed) and generates the necessary requests for the creation/deletion of replicas. Second, the *Repository* manages information on replicas metadata and makes them available to other components of the replication systems. It enables to store, retrieve, modify, and remove metadata. For instance, the placement manager stores the fixed replication scheme in the repository then, may modify or remove it. Third, the *AccessController* has two roles. The first one is related to replica management. It performs services to create/remove replicas, read/write replicas, and to apply updates. The second one deals with concurrency control. Fourth, the *UpdatePropagationManager* ensures the propagation of updates among replicas. It does not modify the replicas but is responsible for triggering the update and transporting the update requests. In fact, this component is notified about the data access attempts. Depending on the consistency protocols, such notifications can trigger the update of the selected replica before performing a read or a write operation. The component stores changes and decides when to send them to the other replicas of the data. Finally, the *RequestEngine* implements the data query function. It provides a service to receive the data access requests from the users. Then, it selects replica(s) to be used to answer these requests. The access operations as well as the update operation are transmitted to components of type *AccessController* that will apply them on the selected replica.

Each component of the replication systems includes some elements that are common to all replication systems and variation points. The values are fixed to these points during the deployment phase by an administrator. Moreover, these values may be modified at runtime by the autonomic managers. Each component is accessed and controlled through an associated touchpoint. The touchpoint is a building block that implements sensor and effector behaviour for the component. Each component has two variation point types: the algorithm applied by the component and the configuration parameters of the algorithm (e.g. the replica update frequency). In order to change the attributes values, the internal structure of a component is defined as the design pattern *Strategy*. This pattern allows defining a family of algorithms that are encapsulated in the same component and interchangeable according to the current execution context. The reconfiguration in this case is simpler and less expensive than a component replacement. We note here that there is no relation between this

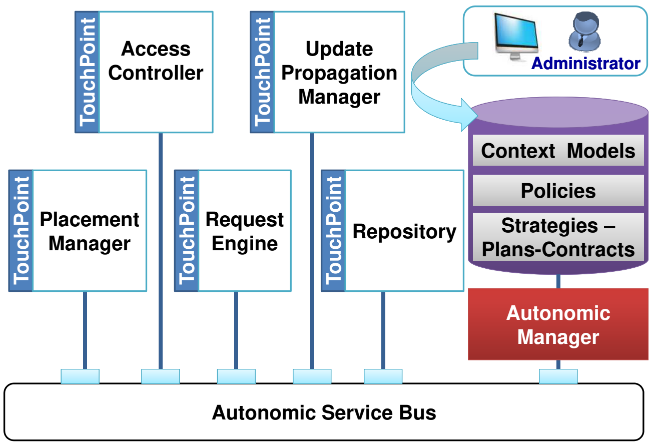

Figure 8: Architectural model of autonomic replication system

pattern which is a standard design in the field of software architecture and the reconfiguration strategy that represents the output of the analyse process. The aim of our approach is not to define new replication methods but to enable several interchangeable implementations of methods proposed in existing works. As a consequence, these mechanisms provide several ways to change the data replication strategies [Goel and Buyya 2006]: the placement method (placement manager), (2) the method for the selection of accessed replica (request engine), and (3) the consistency protocol (access controller and propagation manager).

The autonomic manager supervises and reconfigures the various replication system components. It observes the data management at specific business services using the sensor interfaces. It elaborates a runtime representation of the replication system processes, including information about the managed data, the replication management strategies and the performance indicators. Based on this information, the decider sub-component may choose to dynamically modify the replication system configuration in order to obtain an overall behaviour that more closely corresponds to its high-level QoS goals. For that, the decider interprets the reconfiguration policy provided by the administrator. For example, an update propagation manager component identified as faulty or inefficient can be dynamically replaced by an alternative component. As another example, the placement manager can switch from random replica placement algorithm to new algorithm that limits the replica access delay.

5.2 Self-management mechanisms at the integration layer

The high dynamicity of existing SOA environments requires more efficient ESB solutions to ensure a performance level consistent with user expectations. Our approach tackles the two main challenges confronting the world of performance guaranties. One is the scalability management of an ESB at the IT level in an autonomic way in order to cope with pervasive service integration. The second is related to the way an ESB can take into account the non-functional requirements of service consumers for the purpose of autonomously guarantee them when the service provider cannot do it.

The Autonomic Service Bus (ASB) [Diop and al. 2012] is an extended ESB with the required knowledge and abilities to achieve scalability and QoS management. This management can be done during the initial service integration or dynamically based on observed bus condition. Figure 9 presents the architecture of the ASB. The autonomic message router is used for synchronous and one to one communications and the autonomic message broker enables asynchronous communications or publication of messages that can be accessed by many subscribers. Traditional ESB components (binding components, service engines, message broker, messaging router) are extended with an interceptor, the needed QoS-oriented mechanisms and the required touchpoints in order to become managed elements. These managed elements are monitored and controlled by the Autonomic Manager via the touchpoints. The Semantic Service Repository contains the semantic characterisation of deployed services and QoS-oriented mechanisms. This semantic is expressed in terms of both functional and non-functional properties (e.g. performance, dependability, security, accuracy, trust, etc.). The Context Models describe well known execution environments and behaviour and the Management Policies represents rules to be applied in order to trigger actions implementing the self-management functions and the autonomic behaviour of the ASB according to the context. When a new request comes from a consumer for a specific provider, the request is intercepted and the Autonomic Manager is notified. This one using the knowledge base will check if the provider can satisfy the required non-functional properties or not. Two cases are considered. If the provider satisfies the non-functional requirements, the interceptor releases the request, which is directly routed. Otherwise, the Autonomic Manager of the ASB enforces the most adequate mechanism to deal with them. Once the integrations are established, the ASB needs to control the states of the communications, to collect the context information and to analyse them to predict or to detect anomalies or improvements. In the case of failures prediction and/or detection, corrective actions are performed. The goal is, for instance in best-effort service models, to react in front of congestion that can occur both on

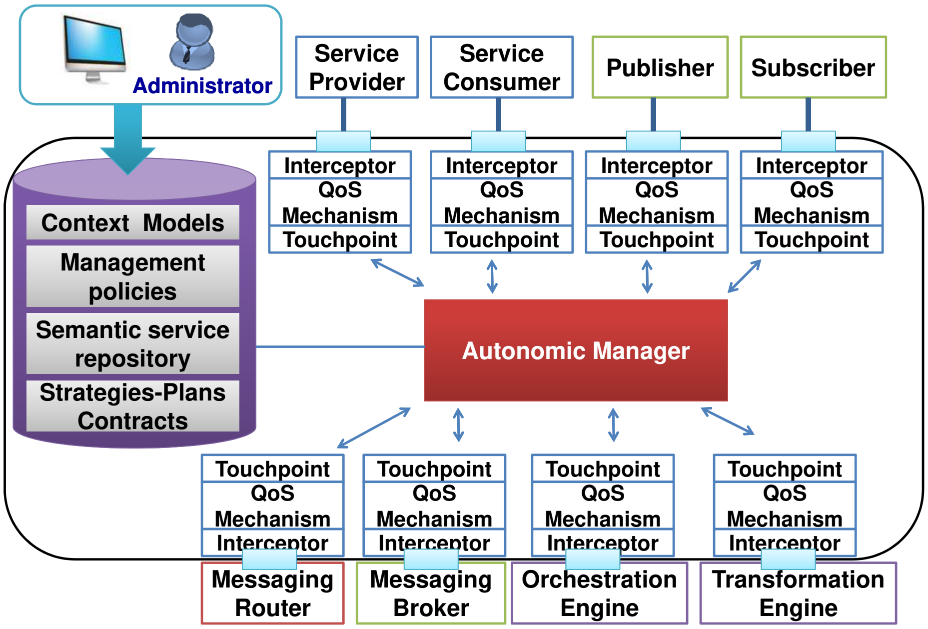

Figure 9: Autonomic Service Bus Infrastructure

service providers and on the internal bus components. Differentiated or guaranteed QoS-oriented models can be implemented by mechanisms able to manage priorities between service consumers or providers if needed. The ESB resolves prioritization based on semantic characterization of the client requirements and the request tags. For instance we offer a best effort service by avoiding that the ESB becomes a bottleneck. We manage its scalability at the IT level by taking advantage of the rapid elasticity concept of cloud computing. A virtual machine is created on a server and the capabilities of this machine are elastically provisioned. We provision or release CPU attributed to this virtual machine according to the traffic through the ESB. The resources allocated to the virtual machine are extended when the ESB meet scalability issues. When the limit of extension is reached, a new ESB instance is deployed on another virtual machine to have a highly available networked service bus. ESB instances will be dynamically clustered and a load balancer is used to share the mediation requests according to the load of each one. The Autonomic Manager, dealing with the CPU usage, evaluates the available resources during the monitoring phase. The Monitor component observes both the ESB performance and the CPU of the machine hosting it. It measures the Key Performance Indicators as the CPU load, computes the values and compares them to two defined thresholds (*maxUsageCPUThreshold* and *minUsageCPUThreshold*). These thresholds will guide the reconfiguration. When the *maxUsageCPUThreshold* value is reached, the Monitor component sends a degradation alarm to the Decider. The optimization alarm means that the CPU usage is lower than the fixed *minUsageCPUThreshold.* Once the Decider receives the alarm, it selects an adjustment of the CPU resource strategy. It sends the strategy to the Planner that defines the plan which is sent to the Executor component. Finally, the Executor enforces the new plan that achieves reconfiguring the virtual machine resources. Similarly, other QoS attributes are considered like the heap memory and the number of concurrent requests.

6 A scenario of self-distributed management processes in integrated healthcare systems

In this section, we present a self-management scenario of the integrated systems presented in Figure 1. The autonomic system includes several autonomic managers (AMs) since the managed elements are running in heterogeneous and distant environments. An AM is associated with each replication system and is specialized differently depending on the environment specificities. Thus, each AM manages a specific network domain: hospital 1, patient's home, and hospital 2 (Figure 10.A). In addition, an AM is associated with the bus to manage the communication and integration services. These AMs coordinate their activities in order to reach their goals in consistent and cost-effective way. For instance, during the management of changes in the behaviour of update propagation managers, the autonomic managers may negotiate the data consistency protocol. Each one includes a component type *Negotiator* connected to others. The implementation of the components is described in Section 7.

Lets consider the configuration of the replication system where an optimistic protocol [Goel and Buyya 2006] (propagate the updates in background, discovers conflicts after they happen, and reach an agreement on the final content) is chosen for a particular data group. We will present a reconfiguration scenario to switch to a pessimistic protocol [Goel and Buyya 2006] (synchronous replica update). When an emergency occurs, *AM3* detects this context change. Its monitor, in the patient home network notifies the symptom to the subscribed decider. This decider interprets the reconfiguration policy and finds that a pessimistic protocol should be adopted. This strategy should then be accepted by deciders associated with the other replication systems by negotiation. The autonomic managers *AM1*, *AM2* and *AM3* play the role of negotiator in this coordination process (Figure 10.B).

The initiator starts the negotiation process by proposing the chosen strategy in form of a contract to the two participants. The contract specifies the strategy (i.e. the name of strategy as well its parameters: the protocol name "primary copy protocol" and the concerned data group "group1") and that the coordination process involves *AM1*, *AM2* and *AM3*. The negotiator associated with *AM1*

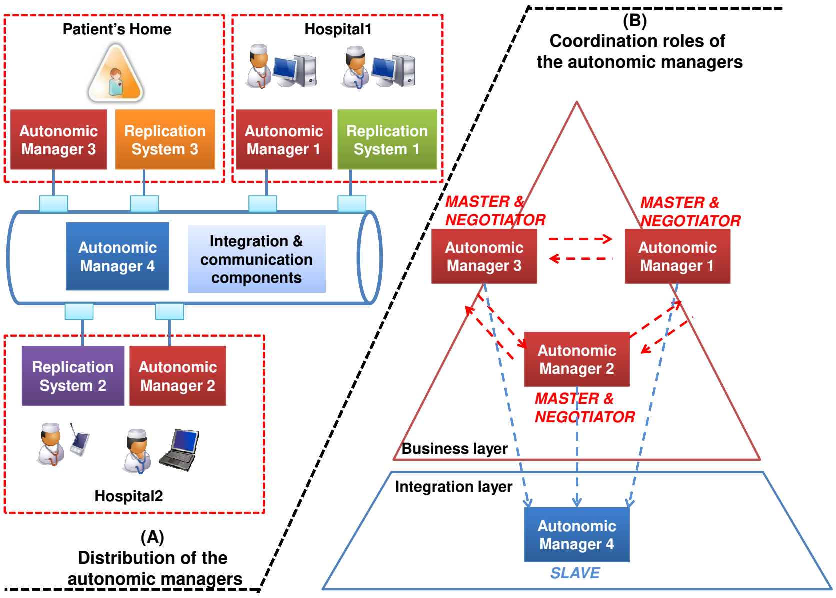

Figure 10: Distribution and coordination roles of the autonomic managers

proposes another protocol (”quorum based protocol”) for performance reasons. A second round of negotiation begins and both *AM2* and *AM3* accept the modification of contract. The negotiation succeeds and each negotiator notifies the final contract to the decider associated with it. Each decider informs the planner about the strategy. The planner constructs the plan and inserts the appropriate coordination actions since it detects that the same strategy will be adopted by the other updates managers.

Figure 11 shows the reconfiguration plan built by each planner. The UML class defines the possible actions (the operations that the class can take) and some attributes that are necessary to parameterize a plan. The planner creates an instance and configures its attributes using the planning policy and the information included in the reconfiguration strategy.

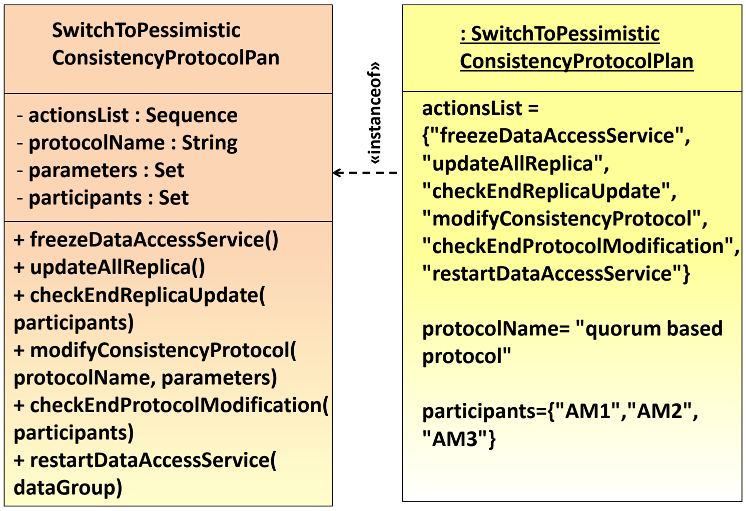

Figure 11: UML class of adaptation plan and its instance

The plan for this scenario contains four composite adaptation actions and is applied concurrently by the involved executors. Firstly, the action ”freezeDataAccessService” activates the interception and the process of freezing data access requests received by the components type *AccessController* into the autonomic managers ”sphere of control”. Then, the ”updateAllReplica” action forces the update of all replicas that are managed by the components type *UpdatePropagationManager* controlled by the adaptation manager. Indeed, the missing updates and the potential conflicts between updates will not be manageable by the new behaviour of the update propagation managers after reconfiguration. The ”modifyConsistencyProtocol” action changes the behaviour of the access controller and the update propagation manager that are into the autonomic managers "sphere of control". The action switches the algorithm applied for the concerned data. Finally, "restartDataAccessService" action reactivates the data access service and launches the processing of the waiting data access requests. In addition, two composite coordination actions are required. The action "checkEndReplicaUpdate" coordinates the end of the update of all the replicas since it needs the collaboration of the update propagation managers before switching to the new behaviour. The action "checkEndProtocolModification" allows each executor to verify that all the update managers and the access controllers have been modified. We will go into details on how this second coordination action is executed. When an executor reaches the "checkEndProtocolModification" coordination action, it uses the interface type *PublishItf* to execute a first primitive action *publish (subject="endProtocolModificationConfirmation", content="true")*. This action enables to broadcast the information specifying that the local protocol modification has finished. A request is sent to the coordinator which transmits this information to the involved coordinators. Then, a second primitive action *coordinate (subject="endProtocolModificationCoordination", participants="all")* is launched using the interface type *CoordinateExecItf* in order to ask the coordinator to manage the achievement of the protocol modification. A delay is specified for each primitive reconfiguration action. Once the delay has expired, the reconfiguration is cancelled and the states of the managed entities are restored.

In addition, the policy of *AM3* specifies that an additional coordination process is needed. *AM3* communicates with *AM4* since the situation requires to reduce the delay of the patient information transfer. *AM3* will play the role of master and *AM4* will be the slave (Figure 10.B). *AM3* sends to *AM4* an order with the goal to assign high priority to the message flows related to the patient. In this case, *AM4* selects the actions plan that consists of configuring the mediations components to manage the concurrent and parallel requests with different priorities as they share the ESB communication resources (Differentiated service model). In this way, the QoS offered to the urgency exchange and more specifically to requests coming from the patient home is improved. The management process is achieved successfully. Few minutes later, the new applied protocol and the use of the ESB by other systems generate communication overload. *AM4* detects the change and new management process starts. In this case, the decider chooses a strategy that does not requires coordination processes. In fact, an elasticity mechanism will be applied to add a new CPU and to allow the ESB to support the load.

7 Implementation and evaluation

A first prototype has been developed focusing on the self-management mechanisms. An experimental application was used for simulating the health delivery services. The application was implemented to supervise a patient home and notify caregivers when an urgency situation is detected. Several sensors are employed for capturing the patient activities and its health status.

For the communication infrastructure, we deploy and enhance a set of WSO2 products [WSO2 2013]. We have implemented the replication components based on the Fractal component model [Bruneton and al. 2006]. This model has been chosen since it provides flexible and extensible control capabilities that make easy the touchpoint implementation. Proxy services that define virtual services are hosted on the ESB that can accept requests, mediate them, and deliver them to replication services. These proxy services perform transport or interface switching and expose different semantics than the actual service, i.e., WSDL, policies and QoS aspects. Endpoints are used as specific destination for messages when designing proxy services.

The WSO2 ESB implementation is used and interconnected to the WSO2 Business Activity Monitor (WSO2 BAM) for the monitoring purpose. WSO2 BAM is a tool designed to perform Business Activity Monitoring. Data collected during the monitoring phase, are processing using WSO2 Complex Event Processor (CEP). Once expected events are predicted or detected, a Decider implemented as Event-Condition-Action (ECA) engine, uses an ontology to infer the reconfiguration strategy. Similarly, the Negotiator, Planner and Coordinator are implemented as ECA engines. The reconfiguration actions involve web services deployed on the ESB so as to change the configuration of this one and the configuration of the mediations strategies deployed on it. The actions may be sent to the services exposed by touchpoints to manage replication services. Each AM becomes aware of other AMs by interpreting the policies that customize its behaviour. Is is envisaged to use a protocol dedicated for the discovery of other managers in dynamic way. This is suitable especially when the number of managers is high and new managed entities may become involved in coordinated self-configuration processes since the systems could evolve (e.g. adding new component, defining new requirement, etc.). Moreover, such a mechanism could make easier the customization of the managers. In addition, we are studying approaches to deal with interruptions and latency in communication in order to avoid stopping or cancelling the self-configuration when a problem occurs.

We focus here on experiments that are intended to measure the impact of distributing the decision making on performance, and more precisely, its responsiveness. To this end, we used the reconfiguration scenario of the consistency protocol described in the previous section to evaluate the negotiation overhead. We measured the time between the reception of the symptom and the choice of

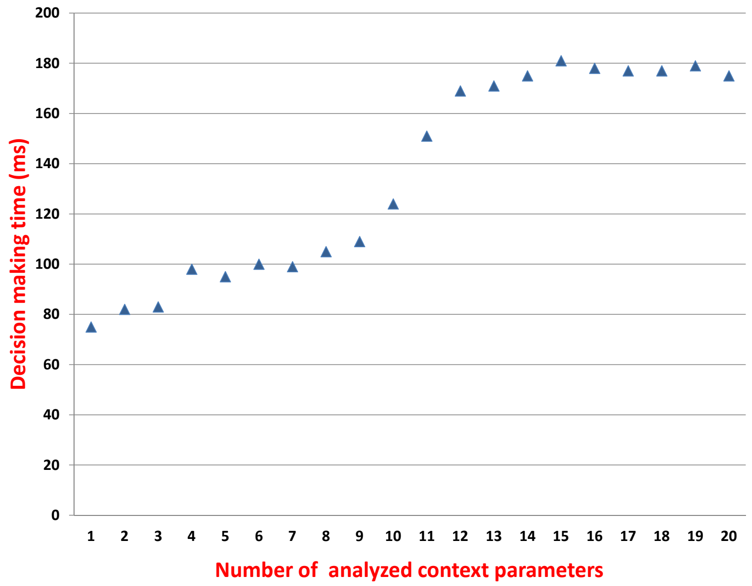

Figure 12: Distributed decision making time

the reconfiguration strategy including context information retrieval. We varied the number of the analysed context parameters to increase the complexity of the decision process. Figure 12 shows the decision making time. We notice that the distributed decision-making process gives an additional time due to the time spent in the two rounds of negotiation. However, the distribution and parallel execution of context evaluation gives performance improvement when the number of parameters becomes high. This proves that the negotiation pattern may be appropriate especially when the analysis of context parameters is costly.

The second experiment concerns the execution time of the plans. In this case, we measure the time between the reception of the plan and the end of its execution. We increase the number of replicas to update in order to increase the number of reconfiguration actions to perform. We note that these actions are costly in time compared to other actions such as switching to another algorithm. Clearly, the distributed management of execution results in faster execution despite the coordination overhead thanks to the parallelism in the plans execution and the decrease of communication latency between each executor and the effectors it controls. In this experiment, the average of time with distributed execution management is 700 milliseconds without high variation despite the number of replicas increase. The distribution of execution management is suitable for costly adaptation plans that contain actions requiring high execution time and which necessitate intense interactions between the executor and the effectors.

Concerning the determination of one plan, fast execution of the process happens since the planners do not need to interact together. In fact, it turned rapidly, spending less than 20 milliseconds for all the considered reconfigurations in the scenario.

Regarding the experiments related to the scalability of the ESB, we focus on the way to trigger the self-management process in order to manage the scalability of the ESB. It is driven by the number of concurrent requests handled by the ESB. Using an emulator, we get offline the maximum of concurrent requests that the ESB is able to handle for a specific configuration (Figure 13). It is possible to have information about this number and to allocate at runtime more resources when the ESB gets near reaching it. The Figure 13 shows that when we configure

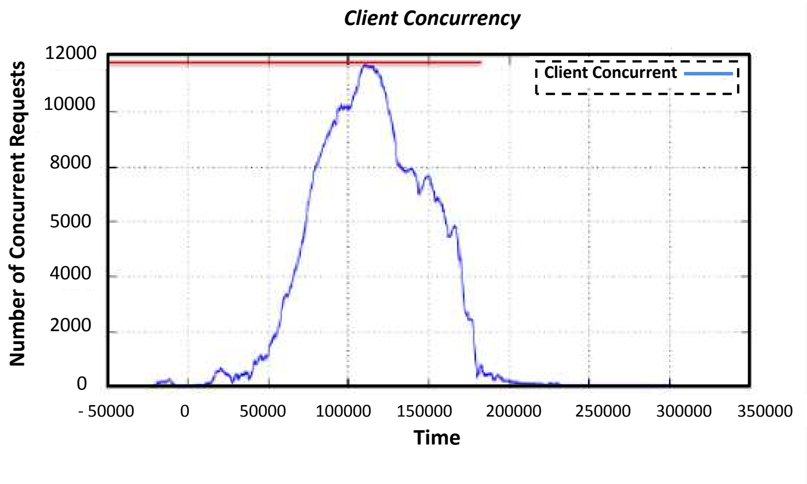

Figure 13: Scalability of the service bus

our ESB in a pass through mode with 1 Gb of memory, the maximum number of concurrent requests is less than 12000.

The execution times of analyse, planing, and execution processes are stored in the knowledge base. As a consequence, the history of execution time can be taken into account during the analyse phase in order to determine if the reconfiguration cost in terms of time is acceptable or not.

Moreover, the management policies could be not easy to develop, particularly if they require different skills (several systems, layers, services, etc). Our approach makes possible the participation of several experts in the underlying managed software components and services in order to define appropriate data to be collected, relevant analysis processes, and suitable actions to be performed.

Nevertheless, there is a possibility of failures due to undesirable circumstances (connection failures, errors, etc) in case of complex choreography. In such context, a choreography should support exception handling.

Finally, the autonomic capabilities of both business services and service bus enable in some situations to minimize the management cost in terms of time to take a decision, stopped-time delay of services necessary for configuration modification, and resource consumption for reconfiguration achievement.

8 Conclusion

We presented in this paper the basic concepts related to our service-oriented approach to the management of distributed autonomic systems. The service-oriented architecture enabled us to reconsider the way we design and implement autonomic systems. Our main contribution is to define an approach for building distributed and coordinated self-management mechanisms. We were particularly interested in the definition of an architectural model and of a flexible and generic set of mechanisms for coordinating self-management processes performed by several autonomic managers. The flexibility of our model allows to specialize the behaviour of collaborative autonomic managers based on orchestration and choreography techniques. Our approach offers several ways to coordinate the decisions making and the execution plans made collectively by several managers. In particular, the negotiation of strategies enables distributed and parallel decision making, the automatic resolution of conflicts among managers and the guarantee of their scalability. Moreover, we enable the QoS management at several layers in a coordinated and optimized way.

There are several possible directions for future work. We are studying approaches to deal with interruptions and latency in communication in order to avoid stopping or cancelling the entire self-reconfiguration process when a problem occurs. A solution will be defined in the near future. We are also interested in facilitating more the customization of the self-management processes. Currently, human actors may spend considerable efforts to make decision related to the deployment strategy of the autonomic managers and to provide management policies. There is a need for new techniques to help experts identify and generate the appropriate deployment strategy and the policies. Moreover, we are exploring the connections between SOA and Cloud Computing. Our vision is to generalize our approach for distributed autonomic platforms in the Cloud (i.e. at Platform as a Service level).

References

[Ac 2006] An Architectural Blueprint for Autonomic Computing; IBM white paper, June, 2006

[Allen and al. 1995] Allen, J., F., Schubert, L., K., Ferguson, G., Heeman, P., Hwang, C., H., Kato, T., Light, M., Martin, N., G., Miller, B., W., Poesio, M., Traum, D., R.: The TRAINS Project: a case study in building a conversational planning agent. Journal of Experimental and Theoretical Artificial Intelligence, 7-48, 1995.

[Amgoud and al. 2004] Amgoud, L., Prade, H.: Reaching agreement through argumentation: A possibilistic approach. Proceedings of the 9 th International Conference on the Principles of Knowledge Representation and Reasoning, KR2004, 2004.

[Beveridge 2001] Beveridge, M., Milward, D.: Definition of the high-level task specification language, Technical report, Deliverable D11, EU HOMEY Project, 2001

[Bruneton and al. 2006] Bruneton, E., Coupaye, T., Leclercq, M.: The FRACTAL component model and its support in java; Softw, Pract. Exper, 1257-1284, 2006.

[Delamer and Lastra 2006] Ivan Delamer, M., Jose Martinez Lastra, L.: Self-Orchestration and Choreography: Towards Architecture-Agnostic Manufacturing Systems: Proceedings of AINA, 573582, 2006.

[Diop and al. 2012] Diop, C., Exposito, E., Chassot, C., Jlidi, D.: QoS-aware and ontology-driven autonomic service bus; Proceedings IEEE WETICE 2012, France, 417422, 2527 Juin, 2012.

[Dobson and al. 2010] Dobson, S., Sterritt, R., Nixon, P., Hinchey, M.: Fulfilling the Vision of Autonomic Computing; IEEE Computer, 43, 1, 2010.

[Durfee and Lesser 1989] Durfee, E., H., Lesser,V.: Negotiating task decomposition and allocation using partial global planning, In L. Gasser and M. Huhns, editors, Distributed Artificial Intelligence Volume II, Pitman Publishing: London and Morgan Kaufmann: San Mateo, CA, 229-244,1989.

[Exposito 2012] Exposito, E.: Advanced Transport Protocols: Designing the Next Generation; Wiley-ISTE Ltd, ISBN: 9781848213746, 320, 2012.

[Ferber 1995] Ferber., J.: Les systmes multi-agents. Vers une intelligence collective. InterEditions, Paris, 1995.

[Geiger and al. 2011] Geiger, M., Schnberger, A., Wirtz, G.: Towards Automated Conformance Checking of ebBP-ST Choreographies and Corresponding WS-BPEL Based Orchestrations; Proceedings of SEKE, 566571, 2011.

[Goel and Buyya 2006] Goel, S., Buyya, R.:Data Replication Strategies in Wide Area Distributed; In Qiu R (ed), Enterprise Service Computing: From Concept to Deployment. Hershey, United States, 211241, 2006.

[Gonzalez and Ruggia 2012] Gonzalez, L., Ruggia, R.: Addressing QoS Issues in service Based Systems through an Adaptive ESB Infrastructure; Proceedings of ACM MW4SOC, Lisbon Portugal, 12 Dec, 2011.

[Greenwood and al. 2003] Greenwood, K., Bench-Capon, T., McBurney, P.: Towards a computational account of persuasion in law, Proceedings of the Ninth International Conference on AI and Law (ICAIL-03), USA, 2003

[Horn 2001] Horn, P.: Autonomic Computing: IBMs Perspective on the State of Information Technology, IBM Research; 15 October, 2001.

[Huebscher and McCann 2008] Huebscher, M., McCann, J.: A survey of autonomic computing – degrees, models, and applications; ACM Comput. Surv. 40, 3, 2008.

[Jennings and al. 2001] Jennings, N., R., Faratin, P., Lomuscio, A., R., Parsons, S., Sierra, C., Wooldridge, M.: Automated Negotiation: prospects, methods and challenge, International Journal of Group Decision and Negotiation (GDN), Vol 10, 99-215, 2001.

[Kephart and Chess 2003] Kephart, J., Chess, D.: The Vision of Autonomic Computing; IEEE Computer Magazine; 36, 1, 2003.

[Khalid and al. 2009] Khalid, A., Abdul Haye, M., Jahan Khan, M., Shamail, Sh.: Survey of Frameworks, Architectures and Techniques in Autonomic Computing; Proceedings of ICAS09, 220225, 2009.

[Morand and al. 2011] Morand, D., Garcia, I., Lalanda, P.: Autonomic enterprise service bus; Proceedings of IEEE 16th conference on ETFA, 59 Sept, 2011.

[Mostarda and al. 2010] Mostarda, L., Marinovic, S., Dulay, N.: Distributed Orchestration of Pervasive Services: Proceedings of AINA, 166173, 2010.
[Muller 1996] Muller, J., H.: Negotiation Principles, Foundations of Distributed Artificial Intelligence, John Wiley and Sons, 211-229, 1996
[Nami and Bertels 2007] Nami, M.R., Bertels, K.: A survey of autonomic computing systems; Proceedings of ICAS07, 26, 2007.
[Parashar and Hariri 2004] Parashar, M., Hariri, S.: Autonomic Computing: An Overview; UPP, Springer, 257269, 2004.
[Rahwan and al. 2004] Rahwan, I., Ramchurn, S., D., Jennings, N., R., McBurney, P., Parsons, S., Sonenberg, L.: Argumentation-based negotiation, 2004.
[Ramchurn and al. 2003] Ramchurn, S., D., Jennings, N., Sierra, C.: Persuasive negotiation for autonomous agents: a rhetorical approach. In IJCAI Workshop on Computational Models of Natural Arguments, 2003.
[Salehie and Tahvildari 2005] Salehie, M., Tahvildari, L.: Autonomic computing: emerging trendsand open problems; SIGSOFT Softw. Eng. Notes, 30(4): 17, 2005.
[Sen and al. 2010] Sen, R., Roman, G., D., Gill, Ch.: CiAN: A Workflow Engine for MANETs; Proceedings of COORDINATION08, 280295, 2008.
[Strassner and al. 2006] Strassner, J., Kephart, J.: Autonomic Systems and Networks: Theory and Practice, Network Operations and Management Symposium; Proceedings of 10th IEEE/IFIP, Vancouver, BC, 37 April, 2006.
[Strassner and al. 2009] Strassner, J., Leon, M., Donnelly, W., Meer, S.: Autonomic Orchestration of Future Networks to Realize Prosumer Services, Future Networks; Proceedings of International Conference on Future Networks, 52156, 2009.
[Valetto and Kaiser 2003] Valetto, G., Kaiser, G.: Using Process Technology to Control and Coordinate Software Adaptation, Proceedings of 25th International Conference on Software Engineering, IEEE Computer Society, USA, 262-272, 2003.
[WSO2 2013] Home page for WSO2 products: http://wso2.com/products.