Non-intrusive autonomic approach with self-management policies applied to legacy infrastructures for performance improvements.

Rémi SHARROCK*
CNRS ; LAAS ; 7 avenue du colonel Roche, F-31077 Toulouse, France
IRIT ; CNRS ; 118 Route de Narbonne ; F-31062 Toulouse ; France
Université de Toulouse ; UPS, INSA, INP, UT2, ISAE ; LAAS ; F-31077 Toulouse, France

Thierry MONTEIL
CNRS ; LAAS ; 7 avenue du colonel Roche, F-31077 Toulouse, France
Université de Toulouse ; UPS, INSA, INP, ISAE ; LAAS ; F-31077 Toulouse, France

Patricia STOLF
IRIT ; CNRS ; 118 Route de Narbonne ; F-31062 Toulouse ; France
Université de Toulouse ; UPS, INSA, INP, UT2, ISAE ; IRIT ; F-31062 Toulouse, France

Daniel HAGIMONT
IRIT ; CNRS ; 118 Route de Narbonne ; F-31062 Toulouse ; France
Université de Toulouse ; UPS, INSA, INP, UT2, ISAE ; IRIT ; F-31062 Toulouse, France

Laurent BROTO
IRIT ; CNRS ; 118 Route de Narbonne ; F-31062 Toulouse ; France
Université de Toulouse ; UPS, INSA, INP, UT2, ISAE ; IRIT ; F-31062 Toulouse, France

ABSTRACT

The growing complexity of large IT facilities involves important time and effort costs to operate and maintain. Autonomic computing gives a new approach in designing distributed architectures that manage themselves in accordance with high-level objectives. The main issue is that existing architectures do not necessary follow this new approach. The motivation is to implement a system that can interface heterogeneous components and platforms supplied by different vendors in a non-intrusive and generic manner. The goal is to increase the intelligence of the system by actively monitoring its state and autonomously taking corrective actions, without the need to modify the managed system. In this paper, we focus on modeling software and hardware architectures, as well as describing administration policies using a graphical language inspired from UML. We demonstrate that this language is powerful enough to describe complex scenarios and we evaluate some self-management policies for performance improvement on a distributed computational jobs load balancer over a grid.

Keywords: Autonomic computing, policies, legacy, distributed, grid, performance.

INTRODUCTION

Autonomic computing principles

Autonomic computing aims to provide methods and tools to answer the exponentially growing demand in IT (Information Technologies) infrastructures. These IT systems are getting increasingly complex while using a wide variety of technologies. Huebscher (2008) compares the actual situation to the one experienced in the 1920s in telephony: automatic branch exchanges finally supplanted trained human operators. Nowadays, large IT facilities involve important time and effort costs to operate and maintain hardware and software. Numerous new technologies are emerging and they consume considerable human resources in learning how to run, tweak, or configure. One of the challenges facing large companies that use such IT infrastructures is that of reducing their maintenance and operating costs (David, Schuff & St. Louis, 2002) in order to increase their dependability (Sterritt & Bustard, 2003) and assurance levels to help them being more confident.

Here are some of the issues raised in this field of research:

First of all, managing large scale infrastructures requires describing the global system in a synthetic way. This involves describing a deployment objective, a picture of what to deploy and how to deploy it. This picture represents what the system should look like upon deployment, the intended construction. Indeed, it is necessary to have an automatically orchestrated deployment process due to the huge number of machines (at least hundreds, and up to thousands) that are potentially involved. It has been shown that large deployments cannot be handled by humans, because this often leads to errors or inconsistencies (Flissi, Dubus, Dolet & Merle, 2008).

Once deployed, the system has to be configured and started. These multiple tasks are ordered (some parts of the system have to be configured or started before others), which also imply an automatic process. Indeed, (Kon & Campbell, 1999) argue that it is hard to create robust and efficient systems if the dynamic dependencies between components are not well understood. They found common issues were some parts of the system fail to accomplish their goals because unspecified dependencies are not properly resolved. Sometimes, failure of one part of the system could also lead to a general system failure.

During runtime, human system operators tend to have slow reaction times and this can result in unavailability of critical services. For example, Gray, (1986) clearly shows that human mistakes made during maintenance operations or reconfigurations are mainly responsible for failures in distributed systems. The need here is to introduce a rapid system repair or reconfiguration so that critical services are kept at an acceptable level Flaviu, (1993). Moreover, new services (large audience services like social networks, video services) tend to follow heavy fluctuations in demands (Cheng, Dale & Liu, 2008). Thus, these services experience scalability issues and need fast rescale mechanisms.

Raised problematics

Exploring the autonomic computing principles lead us to the following questions:

How to describe deployment objectives, repairs, reconfigurations or rescaling? This raises the need for easy to use, high-level policies to describe the management of distributed systems.

How to administrate proprietary system components (also called legacy), even those that do not include their own built-in management capabilities, within another management system? This points out the need for a non-intrusive, generic framework that does not require modification of the proprietary components.

In this paper, we deal with these problems and present an approach for an autonomic computing platform that uses high level policies specification for dynamic reconfiguration of legacy software. Our work is based on an existing prototype called TUNe (Broto, Hagimont, Stolf, De Palma & Temate, 2008), implemented in java, and introduces dynamic performance adaptation capabilities.

This paper is structured as followed: first, we compare our approach to related works and introduce our prototype; we continue with the description of the hardware and software models, and the management policies model. To finish, we experiment the policies on a real system and some results are presented.

Principles of our approach

Our approach aims at managing in an autonomous way distributed systems composed of legacy software elements and probes (the environment) while optimizing simultaneously their performance. These elements are automatically deployed and configured on different hardware resources. During runtime, we enable the environment to evolve (for example, by adding/removing components, restarting, reconfiguring or repairing them). Our autonomic closed control loop therefore encompasses:

1. Event detection by generic probes, proprietary probes or even the software elements themselves if the legacy includes its own probing system. The detection of one or many events can generate one or more notifications depending on the probe's integrated intelligence.
2. Notification dispatching to the autonomic manager that gathers information (how to identify the faulty elements, probing metrics).
3. Reaction: the autonomic manager can then react on the system, given the high-level behavior specification (the intelligence). This description is both used to analyze the information and to carry on actions on legacy software pieces or probes.

Each administration facet (deployment, configuration, reconfiguration, optimization) has a specific description need. Four types of Domain Specific Modeling Language (DSML) are introduced for their specification: one for the Hardware Description (HD), one for the Software Description (SD), one for the Software Wrapper Description (SWD), and one for the Policy Description (PD). The DSML used for SWD is based on a simple XML syntax. The DSML used for HD, SD and PD are inspired from a subset of UML graphical language (Dobing & Parsons, 2006), and the resulting graphical diagrams are called Hardware Description Diagram (HDD), Software Description Diagram (SDD) and Policy Description Diagram (PDD). These diagrams describe the global view of the system and its management in a high level of abstraction. TUNe (Toulouse University Network) deploys applications in various domains: web architecture (like a J2EE service (Chebaro, Broto, Bahsoun & Hagimont, 2009), grid computing (like electromagnetic simulations (Sharrock, Khalil, Monteil, Stolf, Aubert, Cocccetti, Broto & Plana, 2009)), or middleware architecture (like a DIET service (Hagimont, Stolf, Broto, & De Palma,

(2009))). The input diagrams (HDD, SDD and PDD) may be created with graphical UML editors, are being automatically parsed by TUNe, and executed using a multithreaded deamon.

One possible use case for TUNe would be:

1. System preparation and description: creation of the diagrams, TUNe starting.
2. TUNe enters the initial deployment phase of the system: hardware and software are mapped together; software and input files are being installed on machines.
3. TUNe configures (creates the dynamic configuration files), initializes, and launches the system during the starting phase.
4. TUNe manages the system autonomously without any human intervention during the management phase.
5. When the entire system wants to stop or if the user orders to do so, TUNe enters the ending phase.

RELATED WORKS

Autonomic computing approach

Some software programs have their own general-purpose facilities to automate problem detection and/or problem correction. For example, some new operating systems include engines to automate the collection of crash data (like windows XP and its crash report (Ganapathi, 2005)); other tools help detect anomalous behavior by monitoring system, network or application logs like nagios (Harlan, 2003) or ganglia (Massie, Chun & Culler, 2004). However, these tools do not analyze how and what the system should be doing (or not doing). They leave this task to human administrators, who must then determine if something is going wrong, to eventually plan and carry out a reconfiguration or repair process.

Several research works have been conducted in addressing the challenges of autonomic computing. In (Kephart & Chess, 2003), the authors call a system autonomic if it takes care of some “four-Self” concepts, like: Self-configuration, Self-optimization, Self-healing or Self-protection.

As a result, different autonomic systems have been developed. These systems can be broadly classified as

1. Systems that incorporate autonomic mechanisms for problem determination, monitoring, analysis, management (e.g. OceanStore (Kubiatowicz, Bindel, Chen, Czerwinski, Eaton, Geels & Gummadi, 2000), Oceano (Appleby, Fakhouri, Fong, Goldszmidt, Kalantar, Krishnakumar & Pazel, 2001), AutoAdmin (Agrawal, Bruno, Chaudhuri & Narasayya, 2006), QFabric (Poellabauer, Abbasi & Schwan, 2002)).
2. Systems that investigate models, programming paradigms and development environments to support the development of autonomic systems and applications (e.g. Kx (Kinesthetics eXtrem) (Kaiser, Parekh, Gross & Valetto, 2003), Astrolable (Van Renesse, Birman & Vogels, 2003), Autonomia (Dong, Hariri, Xue, Chen, Zhang, Pavuluri & Rao, 2003), AutoMate (Parashar, Liu, Li, Matossian, Schmidt, Zhang & Hariri, 2006)).

Our approach falls in the second category and Kx is closely related to it. We both provide autonomic capabilities onto legacy systems, by relying on specific wrapping mecanism (Effectors in Kx) to adapt a proprietary software. However, we will see that our approach is more high-level since it actually uses a graphical language and it hides the low-level difficulties to the

end user, especially the way to describe an execution flow, or the way to describe architectural modifications. Also, Autonomia implements the *self-configuring* and *self-healing* concepts with mobile agents; our approach implements the *four-self* concept, and is contextually aware by automatically deploying small agents on each nodes. Autonomia also provides check-pointing and migration solutions that are currently being developed in our tool.

Intrusive versus non-intrusive legacy management

Most approaches described in the literature for developing autonomic software are intrusive. Indeed, they use framework interfaces that need to be implemented within the system to be managed, like Autonomic Management Toolkit (Adamczyk, Chojnacki, Jarzqb & Zielinski, 2008), or Policy Management Autonomic Computing (Agrawal, Lee & Lobo, 2005). This means that the components have to be modified, becoming framework-dependent, and also implies that the administrators have access to all source codes to rebuild them. This assumes that the developer of the components will be willing and able to migrate to these frameworks, which is usually not an option because it costs too much (Claremont, 1992).

Some other solutions in the autonomic computing field have relied on a component model to provide a generic, non-intrusive system support (Bouchenak, De Palma, Hagimont & Taton, 2006; Garlan, Cheng, Huang, Schmerl & Steenkiste, 2004; Oreizy, Gorlick, Taylor, Heimbigner, Johnson, Medvidovic & Quilici, 1999). The idea is to encapsulate the managed elements (the legacy software) in software components (called wrappers) and to administrate the environment as a component architecture. This way, the source codes do not have to be modified and the wrapped elements are controlled via the wrappers interfaces.

However, we rapidly observed that the interfaces of a component model are too low-level and difficult to use. This led us to explore and introduce higher level formalisms for all the administration tasks (wrapping, configuration, deployment, and reconfiguration). Our main motivation was to hide the details of the component model we rely on, and to provide a more abstract and intuitive graphical specification interface.

DSML USED FOR THE DEPLOYMENT PHASE

Hardware Description Diagram

When deploying a distributed system, multiple sets of computers may be used, like directly accessible machines or managed resources (accessible via a resource scheduler). The latter include grid platforms and their Virtual Organizations (VOs) (Foster, Kesselman & Tuecke, 2001) or cloud computing platforms (Buyya, Yeo, Venugopal, Broberg & Brandic, 2009; Hayes, 2008). These represent the hardware layer (resource layer) of the system.

In our approach, the hardware platform is described with a DSML inspired from the UML class diagram. In the resulting HDD, each class represents a set of computers, making them members of a same family (called host-family). We eventually found that a minimum of two properties for a family have to be specified:

- How to get resources: direct access, or using a specialized resource scheduler.
- How to access them: which protocol to use to copy the files, execute commands.

These properties are defined using the attributes of the HDD classes. A typical HDD class would include a *user* attribute (the user to use on the machines), a *DirLocal* attribute (the automatically created working directory on the machines), a *javahome* attribute (the jdk location used by TUNe), a *type* attribute (either *local* or *cluster* that uses *nodefile* attributes pointing to a file containing all machines addresses, or a name of a TUNe scheduler plugin), and the *protocole* attribute (either *rcp*, *ssh*, or any other protocol developed in a TUNe protocol plugin). The actual HDD diagram do not support links between HDD classes, and we consider that all machines of all classes using the same protocol can communicate.

To illustrate our approach, we made some experiments using the french national grid Grid'5000 (Cappello, Caron, Dayde, Desprez, Jegou, Primet & Jeannot, 2005). Grid'5000's architecture is distributed among 9 sites around France (dispatched in all major cities), each one hosting several clusters. Grid'5000 software set provides, among others, a reservation tool: OAR (Capit, Da Costa, Georgiou, Huard, Martin, Mounié & Neyron, 2005), and a deployment tool: Kadeploy2 (Georgiou, Leduc, Videau, Peyrard & Richard, 2006). Resource allocation is managed at two levels: a cluster level (with the *oar* command) and a grid level (with the *oargrid* command), and provides most of the important features implemented by other batch schedulers such as priority scheduling by queues, advanced reservations, backfilling and resource match making.

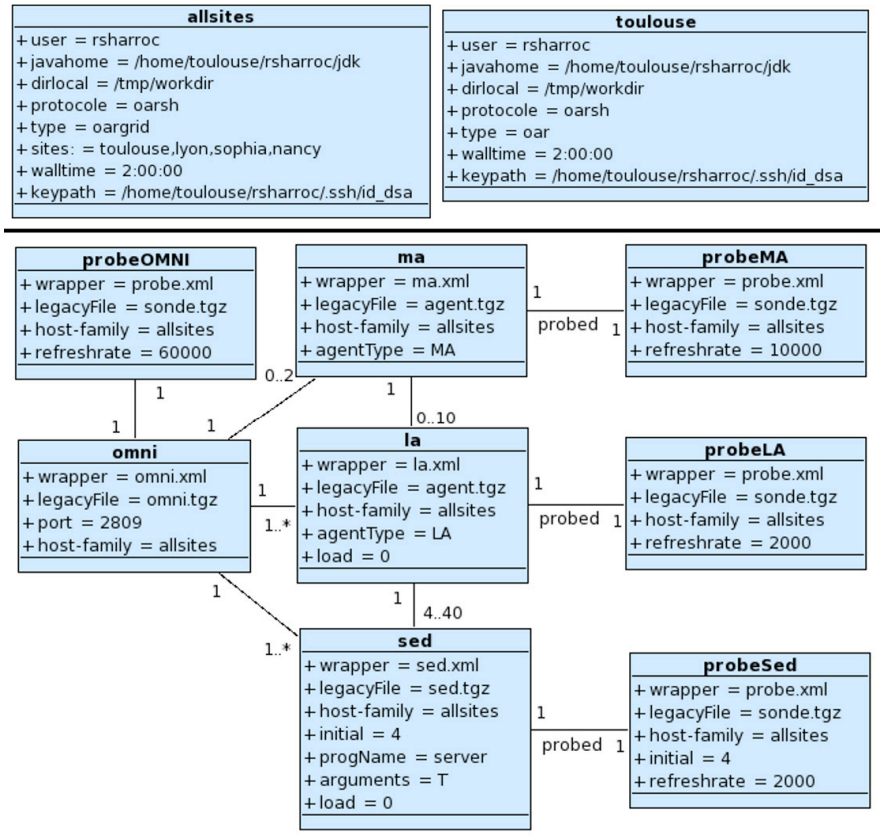

Figure 1: (a) up: HDD for Grid'5000 (b) down: SDD for DIET.

Figure 1 (a) shows a HDD example with two classes representing two families of nodes on the Grid'5000 platform. Specialized attributes are used to fit the OAR resource allocation manager. For this specific grid, the *type* is either *oar* for cluster-level (*toulouse* class), *oargrid* for multiclusters on multi-sites reservations (*allsite* class) or *kadeploy* for specifying what operating systems to deploy during reservation. The *sites* is a scheduler-specific parameter representing the city names and the number of nodes to reserve for each site, and the *walltime* is the duration of the nodes reservation, here 2 hours. The *keypath* facilitates the remote login with ssh keys on the grid to avoid password typing. Finally, there is a specific *protocol* to access the nodes called "oarsh" within Grid'5000, instead of the standard *ssh* protocol.

Other specific properties for managed machines are described in the same way. For instance, the user is able to describe hardware-specific constraints like the minimum amount of memory, CPU speed, or minimum disk space within specific attributes.

Software Description Diagram

The software infrastructure (legacy layer or application layer) is also composed by multiple sets of software pieces. TUNe provides generic software probes that the user can attach to its application, and it is possible to develop proprietary probes as long as their notifications follow a specific syntax (reaction identifier and a set of arguments: names of faulty elements, names and values of probing metrics). Usually, the overall design of a software organization can be described using architecture schemas. They outline the *intended* structure to be deployed initially, but one important aspect of autonomic computing is that the system may evolve during runtime. Indeed, certain parts of the system may grow following the demands, according to the self-optimization recommendation (for example, adding new servers to fit the growing incoming flows of requests). Therefore, it is necessary to introduce a framework for the evolution of the system.

As for the HD, the intended initial deployment of the application and its evolution framework is described using a DSML inspired from the UML class diagram.

As an example, we chose to deploy and manage a middleware named DIET (Caron & Desprez, 2006). DIET stands for Distributed Interactive Engineering Toolbox and is used as a hierarchical load balancer for dispatching computational jobs over a grid. DIET architecture consists of a set of agents: some Master Agents (MA) are linked to Local Agents (LA) that manage a pool of computational SErver Deamons (SED). These servers can achieve specialized computational services. Communications between agents are driven by the omniORB system (OMNI). MAs listen to client requests and dispatch them through the architecture to the best SED that can carry out this service. We attached generic probes to the OMNI, MA, LA and SED to monitor the CPU load average and if the process is alive. For single-CPU systems that are CPU-bound, the CPU load average (output of the *uptime* command) is a percentage of system utilization during the last minute. For systems with multiple CPUs, the probe divides by the number of processors in order to get a comparable percentage.

Figure 1 (b) is the SDD for this architecture where each class corresponds to a software which can be instantiated in several instances (components). Also, maximum and minimum cardinalities between each class have been introduced to limit the number of instances. Predefined attributes are used for the initial deployment, like the *initial* number of instances (1 by default if it is not set), which set of computers to use (*host-family*), the archive containing the

software binaries and libraries (*legacyFile*, TUNe comes with scripts to automatically construct these archives). The *wrapper* attribute gives the name of the SWD which interfaces the software (see Figure 1 (b)). This SDD shows that initially, one MA, two LA and four SED are deployed, as well as their respective attached probes. Moreover, cardinalities show that one MA can be linked up to 10 LA, and each LA can be linked up to 40 SED. Note that the link with the probes is named *probed* to facilitate the use of generic method calls in the SWD. Also, specialized attributes have been introduced to configure the agents (*agentType*, *progName*, *arguments*), and one attribute will be used to store a dynamic values for a metric called (*load*).

Software Wrapper Description

In our approach, each managed software is automatically wrapped into a Fractal component (Bruneton, Coupaye, Leclercq, Quéma & Stefani, 2006). To simplify and hide the complexity of this underlying component model, the wrapper is described using a SWD based on the XML syntax, which contains methods defining how to start, stop, configure or reconfigure the software deployed. TUNe comes with a predefined set of generic SWD containing start, stop and configure methods that can be reused. Most of the needs should be met with this provided finite set of generic methods. Furthermore, other legacy specific methods or proprietary operations can be defined according to the specific management requirements of the wrapped software.

Here is an example for the SWD of the SED in the DIET architecture:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>

<wrapper name='sed'>

 <method name="start" key="extension.GenericUNIXMethods"
    method="start_with_pid_linux" >
  <param value="$dirLocal/$progName $dirLocal/$srname-cfg $arguments $srname"/>
  <param value="LD_LIBRARY_PATH=$dirLocal"/>
  <param value="OMNIORB_CONFIG=$dirLocal/$omni.srname-cfg" />
 </method>

 <method name="configureOmni" key="extension.GenericUNIXMethods"
      method="configure_plain_text">
  <param value="$dirLocal/$omni.srname-cfg"/>
  <param value=" = "/>
  <param value="InitRef:NameService=corbaname::$omni.nodeName:$omni.port"/>
  <param value="DefaultInitRef:corbaloc::$omni.nodeName" />
 </method>
…
```

This SWD shows the methods *start* and *configureOmni* for the SED SDD class. The *key* determines a package location that contains implemented methods. For each method, parameters are given with the *param* tag. The character *$* allows to get the different attribute values from the SDD, and the character . allows to navigate through the links between SDD classes. Thus, values

are dynamically interpreted for each instance of one class of the SDD. Specific keywords allows to get the name of the instance (*srname*, which by default is the name of the SDD class followed by an incremented number), or the node name on which it is deployed (*nodeName*).

DSML USED FOR THE POLICIES DESCRIPTION

DSML principles and definitions

Each characteristic of the four-self concept has a particular need on the managed system. At deployment time, self-configuration of the elements needs a contextually aware environment. Indeed, some elements need to know the configuration of others elements to auto-configure. Self-healing needs a probing system to detect failures, and a way to diagnose and repair them (an example is given in (Broto, Hagimont, Stolf, De Palma, & Temate, 2008). Self-optimization needs a way to adapt the system (modifying the number of instances running) and to apply reconfiguration procedures. Finally, self-protection involves two points of view: external protection to defend against malicious attacks, and internal protection to maintain the system's consistency and to avoid its explosion. In our approach, we consider the "four-self" entirely, but we limit the self-protection to the internal protection.

Policies for the autonomous management of the system are modeled using a DSML inspired from UML activity diagrams (part of the behavioral set of UML). These diagrams are created using the Topcased editor (Farail, Gaufillet, Canals, Le Camus, Sciamma, Michel & Crégut, 2006) that creates uml files following the UML 2.0 standard description in XML. Our tool parses these files and creates runnable objects that are executed following the sequential or parallel flow descriptions of the diagram.

In the following section, we focus on the meta-model for this DSML. We also give examples for the *self-configuring*, *self-optimizing* and *self-protecting* mechanisms.

DSML meta-model for the policy description diagram

Figure 2 illustrates our DSML meta-model for the policy description diagram. This meta-model is composed by multiple *nodes* and eventually some input/output *parameters*. *Nodes* are linked with *edges* also called transitions. We can see that our DSML consists of two different stereotypes for the *Node* metaclass, namely *Action* and *Control*. The stereotype *Action* describes the specific tasks related to the managed system while the stereotype *Control* is used to control the global execution path of execution for the PDD.

The *structural modification* extends the *Action* metaclass, meaning that it has an impact on the managed system. Indeed, creating or destructing a component modifies the running architecture. The same goes for link creation/destruction actions to create or delete bindings betweens components. Moreover, *SWD method call* are used to call SWD methods like starting, stopping or reconfiguring the components during runtimes, thus modifying the system's behavior. The *Component modification and selection* stereotype provides actions to create or modify attribute values with the *Attribute value creation* and *Attribute value modification*. The *Component selection* filters lists of components depending on the values of their attributes. The *PDD reference* action executes another PDD (references another PDD).

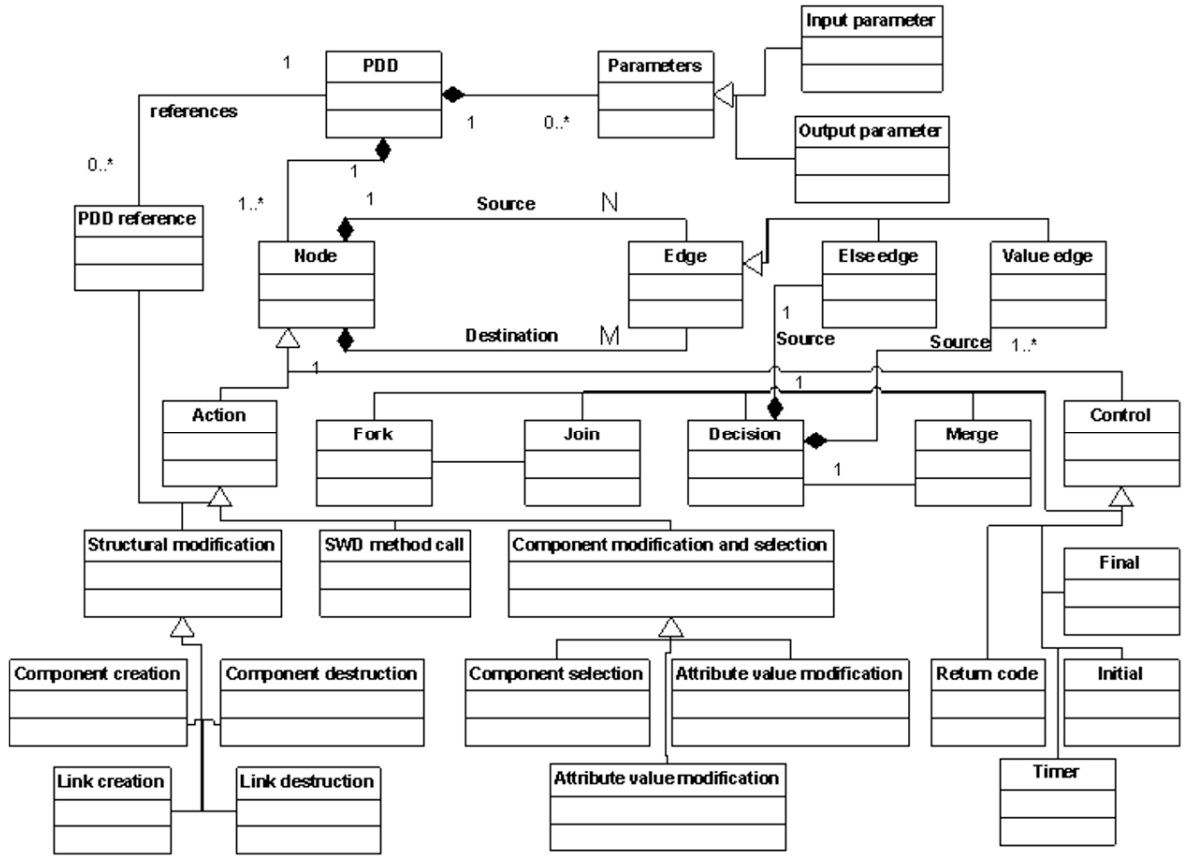

Figure 2: Metamodel of the DSML for the PDD.

The stereotype *Control* can be classified as *Fork* / *Join* nodes, they are used to parallelize the PDD execution flow by creating Threads and synchronizing them. Furthermore, within one PDD, there is a unique entry point which is called the *Initial node* and one unique ending point called the *Final node*. Eventually, at the end of the execution, the diagram can return a code with a *return code* node. The *timer* controls the execution flow by pausing the diagram. Finally, *Decision* and *Merge* nodes are used to create mutual exclusive paths crossed under conditions, and to merge them before the end of the PDD. Note that a Decision node is a source for some *Value edge* and a source for a minimum of one *Else edge*. Indeed, if all conditions of the *Value edge* are not satisfied, this forces the PDD to continue the execution on the *Else edge* to avoid a dead lock. The cardinality between one node and one edge (N,M) depends on the type of the node: Fork(2..*,1), Join(1,2..*), Decision(2..*,1), Merge(1,2..*), Action(1,1), Initial(1,0), and Final(0,1).

Each of the actions is represented by a node that contains its particular expression following a specific syntax. Here are some of these syntaxes, written in the Extended Backus-Naur Form (Scowen, 1993):

component creation = list , "=" , SDD class name , "++" , ["[" , number of instances to create , "]"] ;
PDD reference = [outs] , "=" , PDD name , "(" , [ins] , ")" ;
outs = out , { "," , out } ;

out = list , { " " , variable } ;
ins = in , { "," , in } ;
in = list , { " " , input } ;
input = variable | attribute ;

For example, the component creation is a SDD class name followed by ++ and optionally the number of instances to create in brackets. Another example is the PDD reference consisting of optional input and output parameters (*ins* and *outs*): *ins* are a succession of *in* with a , separator, *in* is a list name followed by a succession of *input* with a space separator, and *input* is either a variable name or an attribute. In the next section, we introduce a simple PDD example consisting of only actions on components (SWD method calls) and standard fork and join nodes. The grammar of the SWD is given below:

method call = list | SDD class name , "." , SWD method name ;

PDD for Self-configuring and starting of DIET

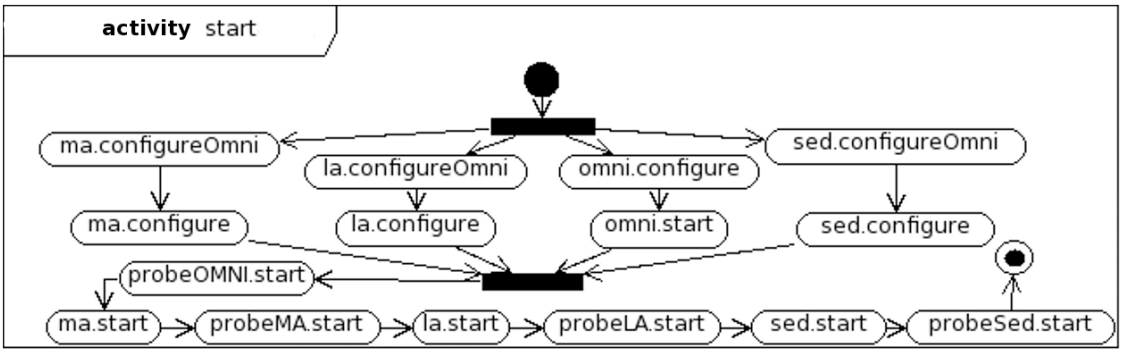

Figure 3: PDD for self-configuring and starting of a DIET architecture.

The Figure 3 represents a particular PDD for starting the DIET software architecture. This PDD is named *start* and is automatically executed upon the starting phase (see the principles of our approach section). To simplify, we do not consider starting errors in this case, and introduce failure cases with the self-protecting case in the section related to self-protection. We can see that actions on components used for starting and configuration (SWD method calls like *ma.configure*) are executed in parallel thanks to fork nodes creating different threads. Moreover, SDD class names are used which means that methods are invoked on all instances of the SDD class, also in a parallel way.

Threads are then synchronized with the join node and the execution follows a specific sequential flow. Indeed, it is necessary to start all MAs before all LAs, so that they can connect to their parent MAs. The same goes for the SEDs that have to be started after their parent LA, and probes that have to be started after the element they are probing.

PDD for Self-optimizing of the DIET architecture

The DIET architecture is designed to be a multi-client platform, meaning that clients can arrive at any time asking for a computational service. Thus, the SEDs may encounter heavy fluctuations of their load that depends, among others, on the number of clients they have to treat in parallel. Furthermore, the LAs dispatching clients to SEDs may also encounter heavy loads that depend, among others, on the clients' requests rate, and the number of SEDs they are in charge of.

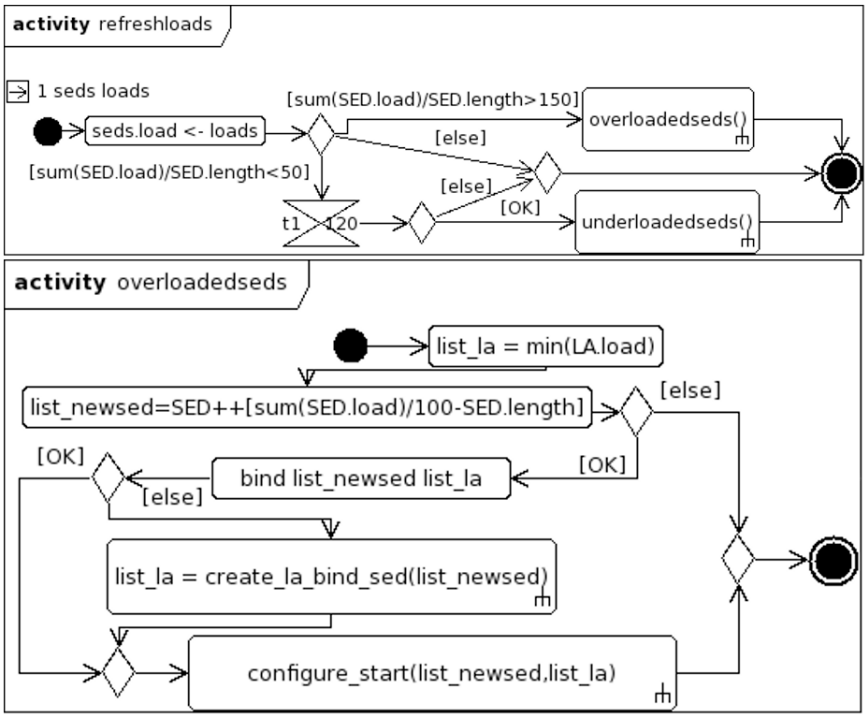

Figure 4: (a) up: global PDD for the experiments (b) down: PDD for self-optimization of DIET.

For this example (Figure 4 (b)), the PDD describes how overloaded SEDs are managed (the average load of all SEDs exceeds a maximum threshold). The goal is to create a calculated number of new SEDs, using the following equation (second action of the PDD):

$x = \left\lceil \frac{\sum_{i=1}^{n} C_i}{T} - n \right\rceil$ with *x*: the number of SEDs to create, *Ci*: the load of SED number *i*, *n*: the actual number of SEDs, and *T*: the targeted average load.

In Figure 4 (b), the targetted average load is the optimum uptime: 100%. The new SEDs are connected to the appropriate LA. Indeed, *list_la = min(LA.load)* creates a list of filtered LAs minimizing their CPU load. As multiple LAs could have the same minimum CPU load value, at binding time (*bind list_newsed list_la*) the ones that are already binded to a minimum of SEDs are chosen. If all LAs have reached their maximum cardinality (Figure 1 (b)), then some others are created (the bind action fails and continues on the *else* transition). All the new SEDs are binded to the newly created LAs in the *create_la_bind_sed* referenced PDD. Finally, all new components are configured (dynamic configuration files are created), and started using the *configure_start* referenced PDD.

Self-protecting within the autonomic manager

Structural modifications of the managed architecture raise difficult problems to maintain the system's consistency, thus our approach also includes static and dynamic verification mechanisms. Indeed, adding and removing components or bindings between them could lead to system inconsistencies, like exceeding the cardinalities between classes in the SDD (Figure 1 (b)). The SDD introduces an architectural pattern and all reconfigurations during runtime must follow the pattern constraints. This ensures the internal protection mechanism, thus making the global system *self-protected*. At first, the autonomic manager includes a static consistency verifier that checks HDD, SDD, SWD and PDD consistencies. These include:

- The DSML consistency, each diagram/SWD must follow the correct DSML syntaxes and constraints.
- The initial deployment consistency, meaning that the initial number of components is consistent with cardinalities between classes in the SDD.
- The management policies consistency, meaning that all PDD actions refer to existing SDD classes and links between classes, that input/output parameters can be mapped within PDD reference calls, and that the execution flow is correct: fork nodes are joined and decision nodes are merged before the final node, and decision nodes placed in-between fork and join nodes are merged before the join node.
- Various consistencies, like the return codes use, and all the expression syntaxes conformity within PDD actions.

If the static consistency verification fails, then the initial deployment will not occur and the diagrams/SWD files have to be modified.

The dynamic consistency verifier acts during runtime, checking if during the PDD execution, the modifications lead to a system inconsistency. For instance, when a new component is created and linked to another, cardinalities between the corresponding SDD classes must be verified.

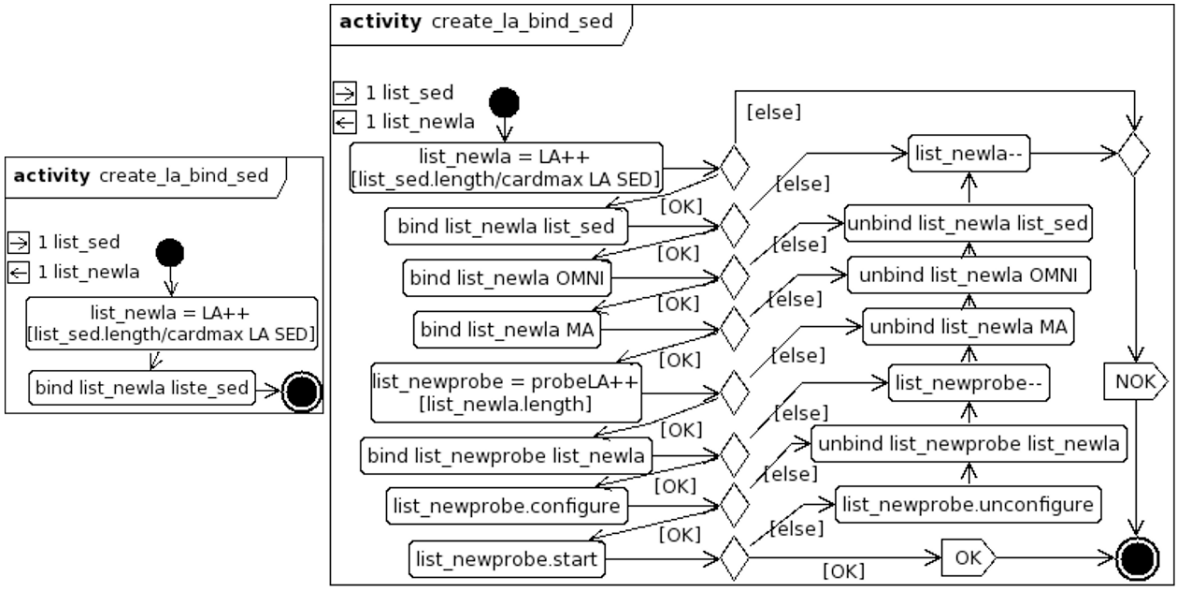

Figure 5: PDD for creating some LA (a) left: user PDD (b) right: visible automatic actions

As shown by Figure 5, three automatic processes are activated to maintain the system consistency. These processes are hidden to the end user, which makes the final PDD much lighter (Figure 5 (a) compared to (b)), but the user is still able to describe manually any other scenarii, overwriting some of the automatic processes. These processes are:

- Automatic binding creation process (the *bind list_newla OMNI* and *bind list_newla MA* actions in Figure 5 (b)): this process is activated at two levels. The first level is when a reconfiguration action needs to navigate through bindings that do not exist. That is, when a reconfiguration action needs to know a configuration parameter of another component it is linked to, but the binding does not exist yet, the binding is automatically created. The second level is when the PDD finishes, and some components do not verify the minimum cardinality of the SDD.
- Automatic component creation process (all actions related to the *probeLA* or the *list_newprobe*): this process is activated when the second level of the automatic binding creation process fails. This means that a component needs to be linked to another type of component but all instances of that type already have reached their maximum capacity defined by the maximum cardinality. Then some new instances are created, binded, configured and started automatically, but only if these new components do not involve another creation, thus avoiding creation loops.
- Automatic rollback process (actions on else edges in Figure 5 (b)): this process is activated when all of the above fail. In the worst case, and if the system is still inconsistent, then all previous actions are cancelled in the exact reverse order. If parallel actions (executed in thread after join nodes) have to be cancelled, the absolute time of execution is considered. This is achievable only for actions that have their reverse action. This is the case for component creation/destruction, link creation/destruction, and generic actions on components like starting/stopping or configuring/unconfiguring.
- At the end of the PDD, return codes like OK or NOK are automatically propagated to the caller PDD.

EXPERIMENTS RESULTS

Experiments preparation

For these experiments, we use and run the policies described by the PDD for self-optimizing. The generic probes send the load of the probed elements periodically (*refreshrate* attribute of Figure 1 (b), in ms) in a *refreshloads* notification. The global policy described with the corresponding PDD (See Figure 4 (a)) is executed when TUNe receives this notification. The first input parameter *1 seds loads* contains the name of the probed element(s) and the new value(s) for their load. The attribute value modification action *seds.load <- loads* updates the load attribute of the components with these fresh values. If the average load of all SEDs exceeds 150, the *overloadedseds* PDD is executed. If it goes under 50, the t1 timer is started for 120 seconds and the *underloadedseds* PDD is executed. The timer ensures that the *underloadedseds* PDD is executed once every 120 seconds at maximum, otherwise the execution continues on the else edge after the timer action.

Underloaded and overloaded experiments

We created a load injector that sends computational client requests to the DIET architecture like matrix multiplications. To simplify, we fixed here one request every fourteen seconds, and each request takes about 2 minutes of CPU time (with two Quad Core Intel Xeon 2.83 GHz CPUs).

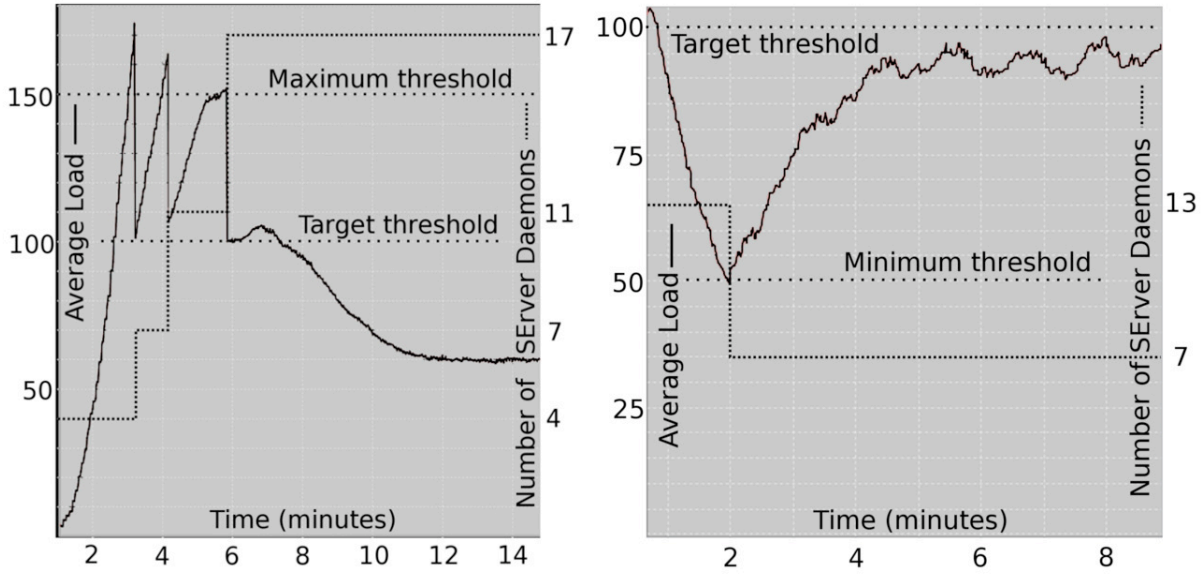

Figure 6: Results after applying the PDD (a) left: overloadedseds (b) right: underloadedseds

Figure 6 (a) shows how the average load of all SEDs if affected when the overloadedseds policy is applied. During the first six minutes, the average load of all SEDs is increasing rapidly and exceeds the maximum threshold three times. As shown by Figure 4 (b), a calculated number of new SEDs is created so that the average load targets the optimum uptime of 100%. Three new SEDs are created the first time, four the second time and six the third time. We can see that the average load goes a little bit over the maximum threshold because it takes a few seconds to create and start all the new SEDs. From t=7 to t=15 minutes, seventeen servers are running and absorbing the client requests, and their load average is stabilizing around 60% at t=10 minutes. The little overshoot of the target threshold at six minutes is due to the fact that the default internal DIET scheduling is a round robin that do not prioritizes newly created SEDs. Thus, some requests are still dispatched to the loaded SEDs until the round robin reaches the new ones.

Figure 6 (b) shows an example of the underloadedseds PDD execution. We first loaded thirteen SEDs and then decreased the number of requests so that their average load goes under the minimum threshold (one request every second with a calculation of about six seconds of CPU time). The number of SEDs to disconnect is calculated the same way as for the *overloadedseds* PDD, but rounded to the smaller integer, here six SEDs are disconnected. When a SED is disconnected, it does not receive more requests and finishes the running ones. Once all requests are finished, TUNe stops all processes, cleans files and makes the hardware available again. We see that the average load stabilizes around 95% from t=2 to t=4 minutes. Indeed, the requests are redirected to the remaining SEDs, and this load transfer and stabilization could take more time depending on the request rate and the calculation time. The t1 timer (Figure 4 (a))

waits two minutes for the next *underloadedseds* PDD execution if the average load stays under the minimum threshold (which is not the case here).

Constrained load balancing experiments

For this experiment, we deployed a DIET architecture with two LAs (added attribute initial=2 in Figure 1 (b) or the LA SDD class). We forced the first Local Agent (LA_1) to be placed on a fast computer, and the second one (LA_2) on a slow computer (Intel Xeon EM64T 3GHz versus Intel Xeon 5110 1.6 GHz). To do this, we created a new host-family in Figure 1 (a) with fixed values for the two computer addresses, and we changed the host-family attribute for the LA SDD class in Figure 1 (b).

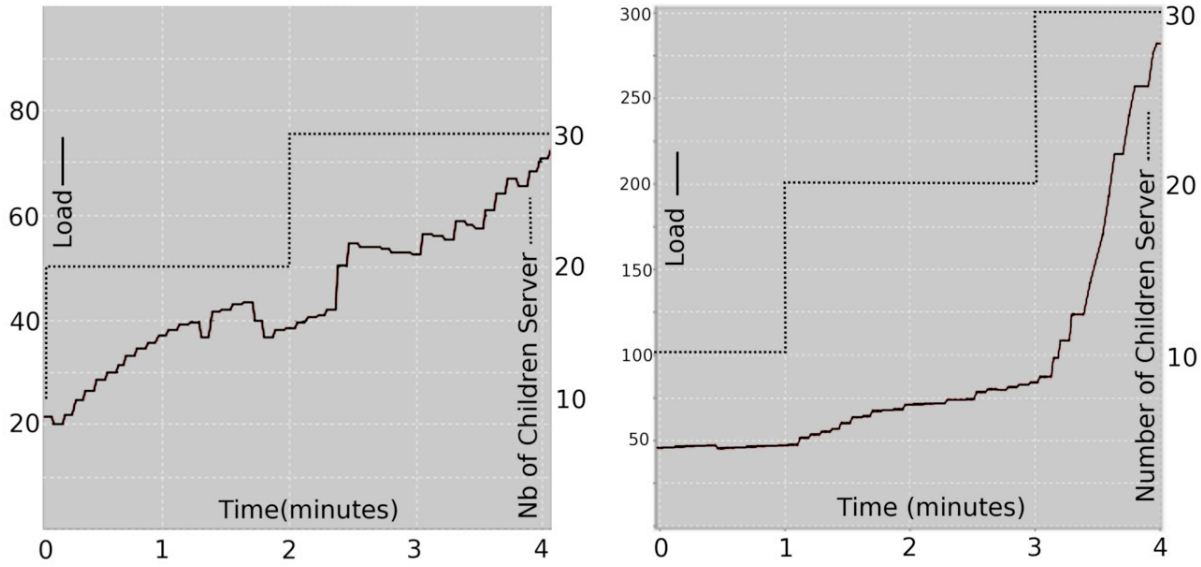

Figure 7: Numerical load balacing of SEDs under LAs (a) left: Local Agent 1, fast computer (b) right: Local Agent 2, slow computer

The first experiment consists in creating ten new SEDs every minute, with a numerical load balancing: the SEDs are connected to the LA with the lowest number of SEDs it manages. To do this, we replaced the first action of the PDD for self-optimization of the DIET architecture with *list_la = min(LA.SED)* (Figure 4 (b)) Figures 7 (a) and (b) demonstrate that a numerical load balancing is not always optimal. Indeed, the LA on the slowest computer cannot handle the incoming request rate (fixed at twenty requests per seconds) when the number of children SEDs reaches thirty (its load overpasses 100% and explodes), whereas the LA on the fastest computer is loaded around 70%. The second experiment constrains the SED binding by choosing the LA that minimizes its CPU load (first action of Figure 4 (b)).

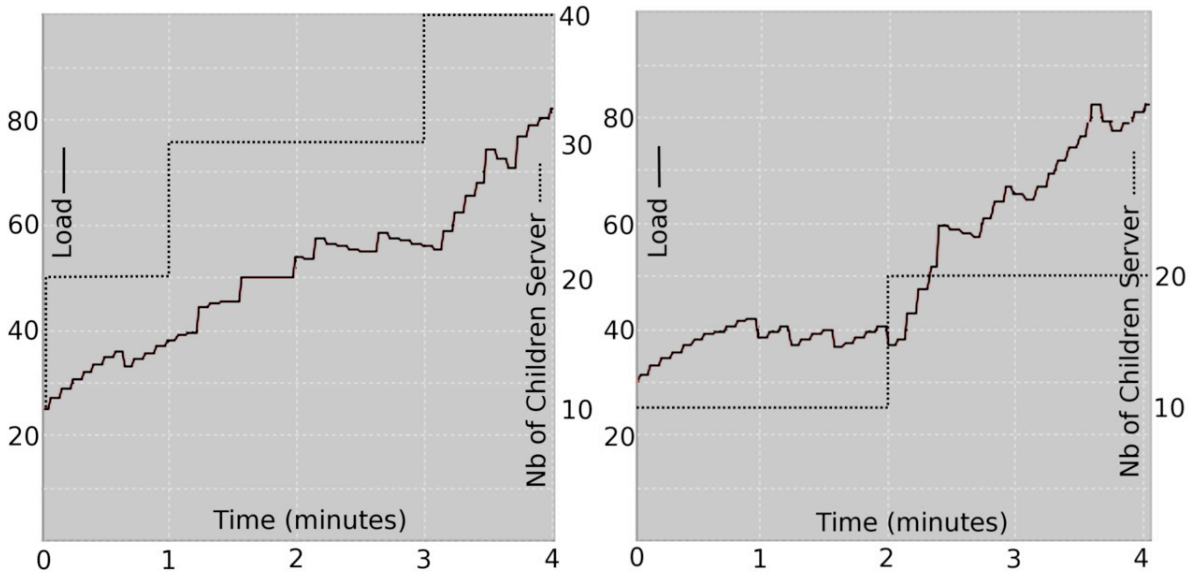

Figure 8: Constrained load balancing of SEDs under LAs (a) left: Local Agent 1, fast computer (b) right: Local Agent 2, slow computer

Figures 8 (a) and (b) show that, as a result, the LA on the fastest computer has to manage more SEDs than the one on the slower computer. Indeed, both LA reach an average load of 80%, but the first one manages forty SEDs, whereas the second one manages twenty SEDs. This demonstrates that, in this case, a constrained balancing for the number of SEDs under LAs is more efficient than a more natural numerical load balancing.

CONCLUSION AND FUTURE WORKS

Large IT facilities are increasingly complex and difficult to manage. They consume a lot of human resources that tend to have slow reaction times. To address this issue, much research projects related to autonomic computing give a new approach in designing distributed architectures that are able to manage themselves. However, existing systems do not necessary follow this new approach. Also, autonomic frameworks and APIs used for developing autonomic software are intrusive. We proposed an approach that is non-intrusive using automatic wrapping of software in a component architecture. This approach takes into account the hardware description, software description and autonomic policies description. These descriptions are modeled using Domain Specific Modeling Languages (DSMLs) inspired from UML, this high level of abstraction hides the usual complexity of the underlying component model. The Toulouse University Network (TUNe) implemented prototype was introduced, as well as connectable generic probes and generic default wrappers. These collections of probes and wrappers, and the reusability of the hardware, software and policies descriptions make the deployment and management of distributed legacy systems a lot easier for the final user. Indeed, TUNe takes into account the entire life-cycle of the software by autonomously reserving hardware resources, mapping software and hardware, deploying, starting, configuring and managing (optimizing, repairing) the software. This prototype implements the four-self concept, and some static and dynamic verifiers with automatic processes that ensure the system

consistency (internal self-protection) during runtime. We demonstrated the effectiveness of some self-management policies for dynamic performance improvement on a distributed computational jobs load balancer (DIET) over a grid (Grid'5000), with experimental results.

As a perspective, we would like to work on a specific extension for TUNe to manage Quality of Service (QoS) within datacenters. Indeed, management needs of networks and applications in datacenters are increasing while customers are asking to meet more QoS needs. This extension would dynamically reconfigure network-level hardware like routers or switches, thus enabling TUNe to manage not only the software layer but also the hardware layer during runtime.

Acknowledgments: The work reported in this paper benefited from the support of the French National Research Agency through projects Selfware (ANR-05-RNTL-01803), Scorware (ANR-06-TLOG-017), Lego (ANR-CICG05-11) and the French region Midi Pyrenees. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

REFERENCES

Adamczyk, J., Chojnacki, R., Jarzqb, M., & Zielinski, K. (2008). Rule Engine Based Lightweight Framework for Adaptive and Autonomic Computing. *Lecture Notes in Computer Science*, *5101*, 355–364.

Agrawal, D., Lee, K. W., & Lobo, J. (2005). Policy-based management of networked computing systems. *IEEE Communications Magazine*, *43*(10), 69–75.

Agrawal, S., Bruno, N., Chaudhuri, S., & Narasayya, V. (2006). AutoAdmin: Self-Tuning Database Systems Technology. *IEEE Data Engineering Bulletin*, *29*(3), 7–15.

Appleby, K., Fakhouri, S., Fong, L., Goldszmidt, G., Kalantar, M., Krishnakumar, S., Pazel, D., et al. (2001). Oceano-SLA based management of a computing utility. In *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management* (Vol. 5). Seattle , WA , USA.

Bouchenak, S., De Palma, N., Hagimont, D., & Taton, C. (2006). Autonomic management of clustered applications. In *IEEE International Conference on Cluster Computing*. Barcelona, Spain.

Broto, L., Hagimont, D., Stolf, P., de Palma, N., & Temate, S. (2008). Autonomic management policy specification in Tune. In *ACM Symposium on Applied Computing*, 1658-1663. Fortaleza, Ceara, Brazil.

Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., & Stefani, J. (2006). The FRACTAL component model and its support in Java. *Software: Practice and Experience*, *36*(11-12), 1257-1284. doi:10.1002/spe.767

Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., & Brandic, I. (2009). Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, *25*(6), 599–616.

Capit, N., Da Costa, G., Georgiou, Y., Huard, G., Martin, C., Mounié, G., Neyron, P., et al. (2005). A batch scheduler with high level components. *Arxiv preprint cs/0506006*.

Cappello, F., Caron, E., Dayde, M., Desprez, F., Jegou, Y., Primet, P., Jeannot, E., et al. (2005). Grid'5000: a large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, 99–106. Seattle,

Washington, USA.

Caron, E., & Desprez, F. (2006). DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid. *International Journal of High Performance Computing Applications,* 20(3):335-352.

Chebaro, O., Broto, L., Bahsoun, J. P., & Hagimont, D. (2009). Self-TUNe-ing of a J2EE clustered application. In *Proceedings of the 2009 Sixth IEEE Conference and Workshops on Engineering of Autonomic and Autonomous Systems*: *Vol. 00*. 23–31. San Francisco, CA, USA.

Cheng, X., Dale, C., & Liu, J. (2008). Statistics and social network of youtube videos. In *IEEE 16th International Workshop on Quality of Service*. Enschede, The Netherlands.

Claremont, B. (1992). *Understanding the Business Aspects of Software Migration*. Migration Specialties.

David, J. S., Schuff, D., & St. Louis, R. (2002). Managing your total IT cost of ownership. *Communications of the ACM*, *45*(1), 101–106. doi:http://doi.acm.org/10.1145/502269.502273

Dobing, B., & Parsons, J. (2006). How UML is used. *Communications of the ACM*, *49*(5), 113.

Dong, X., Hariri, S., Xue, L., Chen, H., Zhang, M., Pavuluri, S., & Rao, S. (2003). Autonomia: an autonomic computing environment. In *Proc. of the 2003 IEEE International Performance, Computing, and Communication Conference*, 61–68.

Farail, P., Gaufillet, P., Canals, A., Le Camus, C., Sciamma, D., Michel, P., Crégut, X., et al. (2006). The TOPCASED project: a toolkit in open source for critical aeronautic systems design. In *European Congress on Embedded Real Time Software*, 54-59. Toulouse, France.

Flaviu, C. (1993). Automatic reconfiguration in the presence of failures. *Software Engineering Journal*, *8*(2), 53-60.

Flissi, A., Dubus, J., Dolet, N., & Merle, P. (2008). Deploying on the Grid with DeployWare. In *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)* (p. 177-184). Présenté au 2008 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid), Lyon, France. doi:10.1109/CCGRID.2008.59

Foster, I., Kesselman, C., & Tuecke, S. (2001). The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, *15*(3), 200.

Ganapathi, A. (2005). *Why does Windows Crash?* Berkeley university.

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

Georgiou, Y., Leduc, J., Videau, B., Peyrard, J., & Richard, O. (2006). A tool for environment deployment in clusters and light grids. In *Second Workshop on System Management Tools for Large-Scale Parallel Systems, Vol. 4*. Rhodes Island, Greece.

Gray, J. (1986). Why do computers stop and what can be done about it. In *Symposium on reliability in distributed software and database systems, Vol. 3*. Los Angeles, California, USA.

Hagimont, D., Stolf, P., Broto, L., & De Palma, N. (2009). Component-Based Autonomic Management for Legacy Software. *Autonomic Computing and Networking*, 83-104.

Harlan, R. C. (2003). Network management with Nagios. *Linux Journal*, *2003*(111), 3.

Hayes, B. (2008). Cloud computing. *Communications of the ACM Journal*, *51*(7), 9-11. doi:10.1145/1364782.1364786

Huebscher, M. C., & McCann, J. A. (2008). A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys*, *40*(3), 1-28. doi:10.1145/1380584.1380585

Kaiser, G., Parekh, J., Gross, P., & Valetto, G. (2003). Kinesthetics eXtreme: an external

infrastructure for monitoring distributed legacy systems. In *Autonomic Computing Workshop Fifth International Workshop on Active Middleware Services*, 22–30. Seattle, Washington, USA.

Kephart, J. O., & Chess, D. M. (2003). The vision of autonomic computing. *Computer*, 41–50.

Kon, F., & Campbell, R. H. (1999). Supporting Automatic Configuration of Component-Based Distributed Systems. In *USENIX Conference on Object-Oriented Technologies and Systems*, *Vol. 5*, 13). San Diego, California, USA.

Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., et al. (2000). Oceanstore: An architecture for global-scale persistent storage. *ACM SIGARCH Computer Architecture News*, *28*(5), 190–201.

Massie, M. L., Chun, B. N., & Culler, D. E. (2004). The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, *30*(7), 817–840.

Oreizy, P., Gorlick, M. M., Taylor, R. N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., et al. (1999). An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, *14*(3), 54–62.

Parashar, M., Liu, H., Li, Z., Matossian, V., Schmidt, C., Zhang, G., & Hariri, S. (2006). AutoMate: Enabling Autonomic Applications on the Grid. *Cluster Computing*, *9*(2), 161-174. doi:10.1007/s10586-006-7561-5

Poellabauer, C., Abbasi, H., & Schwan, K. (2002). Cooperative run-time management of adaptive applications and distributed resources. In *Proceedings of the tenth ACM international conference on Multimedia*, 402–411.

Scowen, R. S. (1993). Extended BNF-a generic base standard. In *Software Engineering Standards Symposium*, *Vol. 3*, 6–2.

Sharrock, R., Khalil, F., Monteil, T., Stolf, P., Aubert, H., Cocccetti, F., Broto, L., & Plana, R. (2009). Deployment and management of large planar reflectarray antennas simulation on grid. In *Proceedings of the 7th international workshop on Challenges of large applications in distributed environments*, 17–26. Munich, Germany.

Sterritt, R., & Bustard, D. (2003). Autonomic Computing - a means of achieving dependability? In *10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems,* 247–251. Huntsville, Alabama, USA.

Van Renesse, R., Birman, K. P., & Vogels, W. (2003). Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems (TOCS)*, *21*(2), 164–206.