Autonomic Management using Self-Stabilization for Hierarchical and Distributed Middleware

Eddy Caron[†], Maurice Djibril Faye[*†] and Ousmane Thiaré[*]
[*]Université Gaston Berger. Sénégal
[†]LIP Laboratory, UMR CNRS - ENS de Lyon - INRIA - UCB Lyon 5668
University of Lyon, France
{Maurice.Faye, Eddy.Caron}@ens-lyon.fr, Ousmane.Thiare@ugb.edu.sn

***Abstract*—Dynamic nature of distributed architecture is a major challenge to avail the benefits of distributed computing. An effective solution to deal with this dynamic nature is to implement a self-adaptive mechanism to sustain the distributed architecture. Self-adaptive systems can autonomously modify their behavior at run-time in response to changes in their environment. This capability may be included in the software systems at design time or later by external mechanisms. Our paper describes the self-adaptive algorithm that we developed for an existing middleware. Once the middleware is deployed, it can detect a set of events which indicate an unstable deployment state. When an event is detected, some instructions are executed to handle the event. We have designed a simulator to have a deeper insights of our proposed self-adaptive algorithm. Results of our simulated experiments validate the safe convergence of the algorithm.**

***Keywords*—Middleware; Autonomic System; Self-stabilization; Simulation; Finite State Machine; Cloud; Distributed Computing**

I. Introduction

Distributed environments are complex systems. They are available in different flavors like Grid Computing, Cloud Computing, Desktop Computing, etc. Middleware are used to manage the complex access of the advantages which are provided by these distributes environments. Thus the deployment of the middleware is mandatory. Once the middleware is deployed, how it adapt the changes in dynamic environment? If the deployment is static, it may be necessary to redo all the deployment process, which is a costly operation. A better solution would be to make the deployment self-adaptive [9], [10].

This paper describes our work which aims to add self-adaptive capabilities to an existing middleware [2] so that its deployment becomes self-adaptive. The middleware is hierarchical and is composed of a finite set of component type. Thus, we have designed a self-adaptive algorithm that encapsulates the behaviour of each component instances with respect to component type. The algorithm is a set of rules. Each rule consists of an event (a guard) and a set of instructions. The detection of a rule event triggers the execution of the rule instructions. The middleware deployment is unstable when at least one rule is enabled, that is , the rule event is detected and the corresponding rule instructions are being executed. If no rule is enabled, the middleware deployment is stable. To have a deeper insight of the algorithm behaviour (for example the number of time units an unstable deployment takes to become stable), we have designed a simulator and we have used Finite State Machines (FSMs) [1] to model the middleware instances. The simulation results validate the safe convergence of the algorithm. This paper is organized as follows: Section II presents the architecture of the targeted middleware we rely on for this work. Section III presents the self-adaptive algorithm. The section IV describes the simulation (IV-A- simulator description, IV-B- simulator environment and simulation results). Section V presents related work and section VI presents conclusion and future work.

II. Architecture of the Targeted Middleware

This paper introduces an autonomic solution to maintain an existing middleware in a correct configuration. DIET (Distributed Interactive Engineering Toolbox) [3] is focused on the development of a scalable middleware with initial efforts relying on distributing the scheduling problem across a hierarchy of agents. The middleware is built for distributed environment from cluster to Grid and Cloud [2].

The DIET component architecture is illustrated in Fig. 1.

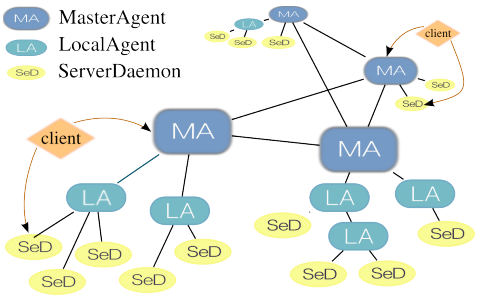


Fig. 1. DIET multi-hierarchy.

The DIET framework is composed of four element types: The first element is a **Client**, an application that uses the DIET infrastructure to solve problems using a GridRPC approach [4]. The second is the **SeD** (**Server Daemon**) which acts as the service provider, exposing functionalities through a standardized computational service interface; the third element of the DIET architecture are the **agents** which facilitate the service location and invocation interactions between clients and SeDs. Collectively, a hierarchy of agents provides higher-level services such as scheduling and data management. These services become scalable by distributing them across a hierarchy of agents composed of one or more **Master agents (MA)** and several **Local agents (LA)**.

III. SELF-ADAPTIVE ALGORITHM

The self-adaptive algorithm described below allows a self-adaptive behaviour of a deployment. Only events related to the middleware components are managed (joining of new isolated instances, deletion of existing instances, link lost between to instances, etc.).

The self-adaptive algorithm is defined by a set of rules. From the middleware perspective, the rules we defined below can also detect inefficient deployments (when at least one rule is enabled). Each time a rule is enabled, the corresponding rule actions are executed. The objective of a rule actions is to lead the deployment through a stable state. An efficient and stable middleware deployment is one in which no rule is being executed. A set of rules are defined for each middleware component type.

Client rules: 3 rules are defined for the Client:

Client rule R1 defines how a Client instance reacts when it detects a connection lost with a MA instance when at least one other MA instance exists.

Client rule 1: R1

1 **if** $Client \ \wedge \ (MA\_lost == True) \ \wedge \ (\#MA > 0)$ **then**
2 | select one MA and connect;
3 **end**

Client rule R2 defines how a Client instance reacts when it detects a connection lost with a MA instance and there is no other MA instance.

Client rule 2: R2

1 **if** $Client \ \wedge \ (MA\_lost == True) \ \wedge \ (\#MA == 0)$ **then**
2 | create one MA as the client Child ;
3 **end**

Client rule R3 defines how a Client instance reacts when it detects a connection lost with a SED instance.

Client rule 3: R3

1 **if** $Client \ \wedge \ (SeD\_lost == True)$ **then**
2 | submit the request again ;
3 **end**

MA rules: 5 rules are defined for the MA:

MA rule R4 defines how a MA instance reacts when it detects that it has no child and knows that there is at least one other MA instance than itself.

MA rule 4: R4

1 **if** $MA \ \wedge \ (\#MA\_children == 0) \ \wedge \ (\#MA \ > \ 1)$ **then**
2 | $\#MA = \#MA - 1$;
3 **end**

MA rule R5 defines how a MA instance reacts when it detects that it has neither child nor other MA instance.

MA rule R6 defines how a MA instance reacts when it detects that it has a unique agent child. The $Merge(x, y)$ function (line 2, R6) remove y and connects children of y as new children of x.

MA rule 5: R5

1 **if** $MA \ \wedge \ (\#MA\_children == 0) \ \wedge \ (\#MA \ == \ 1)$ **then**
2 | add a SED as MA child ;
3 **end**

MA rule 6: R6

1 **if** $MA \ \wedge \ (\#MA\_children == 1) \ \wedge \ (MA\_child\_type == (MA \ \vee LA))$ **then**
2 | $Merge(MA, MA\_child)$;
3 **end**

MA rule R7 defines how a MA instance reacts when it detects that there is at least another tree with a MA instance as root but the two trees are not connected.

MA rule 7: R7

1 **if** $MA \ \wedge \ (\{\#father \ / \ fatherType = MA\} == 0) \ \wedge \ (\{\#tree \ / \ treeRootType = MA\} > 1)$ **then**
2 | select one tree root of type MA as MA father ;
3 **end**

MA rule R8 defines how a MA instance reacts when it detects it is overloaded.

MA rule 8: R8

1 **if** $MA \ \wedge \ (MA\_load \ \geq \ MA\_load\_threshold)$ **then**
2 | divide the MA children in two sets A, B:
| $\mid card(A) - card(B) \mid \ \leq \ 3$;
3 | create one agent as father of all the instances in each set ;
4 | the roots (2 agents) of newly created trees becomes MA Children
5 **end**

LA rules: 6 rules are defined for the LA.

The LA rules are almost the same as the MA rules because both are agent and play almost the same role. LA has 6 rules instead of 5 like the MA. This additional rule for LA is LA rule R13 which illustrates the case when a LA instance detects it has no father and gets the information there is no MA in the system. The others LA rules, R9, R10, R11, R12, R14 can be respectively interpreted as MA rules R4, R5, R6, R7, R8 by replacing MA by LA and agent (which may be a MA or a LA) by LA.

LA rule 9: R9

1 **if** $LA \ \wedge \ (\#LA\_children == 0) \ \wedge \ (\#LA \ > \ 1)$ **then**
2 | $\#LA = \#LA - 1$;
3 **end**

LA rule 10: R10

1 **if** $LA \ \wedge \ (\#LA\_children == 0) \ \wedge \ (\#LA \ == \ 1)$ **then**
2 | add a SED as LA child ;
3 **end**

SED rules: 3 rules are defined for the SED.

The SED rule R15 illustrates the reaction of SED which is not currently executing a job, has no father, included in a deployment with zero agent.

LA rule 11: R11

1 **if** $LA \wedge (\#LA\_children == 1) \wedge (LA\_child\_type == LA)$ **then**
2 | $Merge(LA, LA\_child)$;
3 **end**

LA rule 12: R12

1 **if** $LA \wedge (\#LA\_father == 0) \wedge (\#\{tree : treeRootType = (LA \vee MA)\} > 1)$ **then**
2 | select one tree root (of type MA or LA) as LA father;
3 **end**

LA rule 13: R13

1 **if** $LA \wedge (\#LA\_father == 0) \wedge (\#\{tree : treeRootType = (LA \vee MA)\} == 1)$ **then**
2 | add one MA as LA father ;
3 **end**

LA rule 14: R14

1 **if** $LA \wedge (LA\_load \geq LA\_load\_threshold)$ **then**
2 | divide the LA children in two sets A, B:
| $\mid card(A) - card(B) \mid \leq 3$;
3 | create one LA as father of all the instances in each set ;
4 | the roots (2 LA) of newly created trees becomes LA Children
5 **end**

SED rule 15: R15

1 **if** SED $\wedge (\#SeD\_father == 0) \wedge (is\ computing == False) \wedge (\#\{MA,\ LA\} == 0)$ **then**
2 | add one MA as SeD_father;
3 **end**

The SED rule R16 illustrates the reaction of SED which is not currently executing a job, has no father, included in a deployment which contains at least one agent.

SED rule 16: R16

1 **if** SED $\wedge (\#SeD\_father == 0) \wedge (is\ computing == False) \wedge (\#\{MA,\ LA\} > 0)$ **then**
2 | select one agent (MA or LA) as SED father ;
3 **end**

The SED rule R17 illustrates the reaction of SED which has no father but is currently executing a job.

SED rule 17: R17

1 **if** SED $\wedge (\#SeD\_father == 0) \wedge (is\ computing == True)$ **then**
2 | compute for max of T (user defined parameter) units of time after which the current computation is supposed to be over.
| `/* there is no unbounded computation: all */`
| `/* are terminated after a unit of time */`
3 **end**

The effects of each of the above rules are summarized in Table I.

TABLE I. RULES EFFECTS

Element	RULE Id	RULE Effects
Client	R1	#tree - 1
	R2	#MA + 1
	R3	re-submit request
MA	R4	#MA - 1
	R5	#SED + 1
	R6	#agent - 1
	R7	#tree - 1
	R8	#agent + 2
LA	R9	#LA - 1
	R10	#SED + 1
	R11	#LA - 1
	R12	#tree - 1
	R13	#MA + 1
	R14	#LA + 2
SED	R15	#MA + 1
	R16	#tree - 1
	R17	T unit of time computing

IV. SIMULATION

A. Simulator description

We have designed a simulator for the purpose of validating the self-adaptive algorithm and especially study the convergence. The simulator is coded with the Erlang programming language [7].

The Simulator can:

Create a predefined or a random deployment; Generate simulation events which can trigger self-adaptive behaviours (rules) of the instances; Display the global state of the deployment (if the deployment is stable or unstable) with the number of stable and unstable instances; count the number of hops (the deployment takes to recover after a simulation event). The simulator is composed of 3 main parts: A **deployment server** which serves as an oracle and acts as a resource discovery service. It keeps a runtime image of the deployment. A **stability detection server** which serves to detect the global state of a deployment, i.e, if the deployment is stable or unstable. A description of the stability detection algorithm is given at Section IV-A4 A **Deployment** which Consists of an hierarchy of instances linked between them. The hierarchy is created by the simulator randomly or from a predefined description.

1) Middleware objects representation:

As described in section II, the middleware is composed of four types of basic components (Client, MA, LA, SED). We used **Finite State Machine (FSM)** [1] to model each of these four elements. A FSM contains a finite number of states and produces outputs on state transitions after receiving inputs.

2) FSM state management:

The internal state of a FSM is described by a set of local variables which are used to calculate the FSM state. A state calculation is triggered by the reception of a **state calculation message**. A state calculation message can only be sent by an instance to itself. A state calculation message can be sent after a periodic test (neighbours link) or after the management of an incoming message which may modify the instance state. Fig. 2 highlights the behaviour of a generic FSM instance and the kind of states transition a FSM instance can do. The value of k depend on the element type and is equal to the number of rules defined for this element (each rule detects an unstable state). We have used a non-deterministic FSM model. When a process is in one of the k unstable states, it can make a transition from its current state to the same unstable state (statu quo) or to another unstable state. it can also make a transition to the stable state.

3) Deployment stability definition:

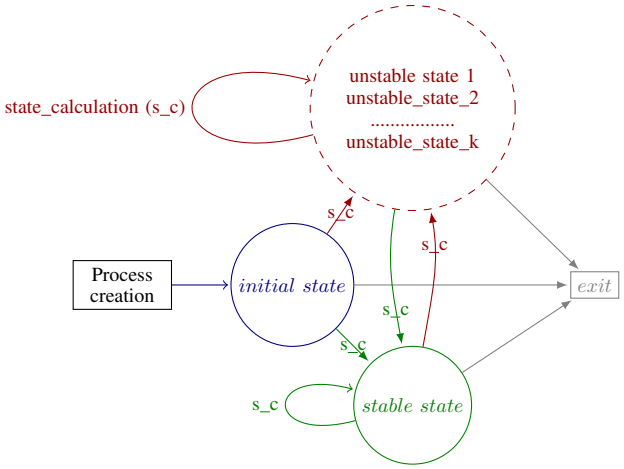


Fig. 2. A generic FSM states transition

A deployment corresponds to an hierarchy of instances (modeled as FMSs), deployed on Erlang nodes. A **deployment is stable** when each instance is in its stable state (e.g., is not executing a rule actions). In this case, $\#\{unstable\ instances\} = 0$. Once the deployment is stable, it remains stable and its number of instances is constant in the absence of external events.

4) Deployment stability detection:

For simulation purposes, we need to be able to detect a stable deployment. A stable deployment is a global state, and the deployment stability detection can be understood as a termination detection algorithm [8]. Termination detection determines whether a distributed algorithm has terminated. We want to detect the self-adaptive algorithm termination, which corresponds to a stable deployment. To detect a stable deployment, we used a stability detection server (Section IV-A). It has two internal variables which are updated as descibed in the Listing 1, where: **UnstableInstanceCount** and **StableInstanceCount** are positive integer variables, initialized to zero when the server is launched (before the creation of any instance). These two variables are intended to count respectively the current number of unstable and stable instances. The server reads periodically the values of these two variables and display the global state of the deployment (stable when $UnstableInstanceCount = 0$, unstable otherwise).

```
1  case {previousState, currentState} of
2  /*case state transition*/
3  /*transition from stable state to one of
4  the unstable states*/
5
6  {stable, unstable}: UnstableInstanceCount++;
7                      StableInstanceCount--;
8
9  /*transition from one of the unstable
10 states to stable state*/
11
12 {unstable, stable}: StableInstanceCount++;
13                     UnstableInstanceCount--;
14
15 /*dying unstable state. instance in its
16  termination function*/
17
18 {unstable, dying instance}: UnstableInstanceCount--;
19
20 /*dying stable state. instance in its termination function*/
21
22 {stable, dying instance}: StableInstanceCount--;
23
24 /*transition from initial state to
25 one of the unstable states. first state
26 calculation after initialization state*/
27
28 {initial_state, unstable}: UnstableInstanceCount++;
29
30 /*transition from initial state
31  to stable state. first state calculation
32  after initialization state*/
33
34 {initial_state, stable}: StableInstanceCount++;
35
36  end;
```

Listing 1. Stability detection method

Each time an instance calculates its state for any reason, it records its previous state and after calculating its current state, it compares the two states. If a state variation (a transition from state s_i to s_j with $s_i \neq s_j$) is noticed, the instance sends an update message to the stability detection server. If $s_i = initial\_state$, it means that the instance recalculates its state for the first time after its initialization. The message depends on the variation type, it can lead to the increment and/or decrement of **UnstableInstanceCount** and/or **StableInstanceCount**. When the test is done inside the termination function (the last function a dying instance can execute), an update message is sent to the stability detection server according to the last state the instance has before it died.

B. Results

All simulations were performed on a computer with the following information: Debian GNU/Linux 7 (wheezy) operating system, Intel(R) Xeon(R) CPU X5570 @ 2.93GHz with 16 cores and 33 GB of RAM, Erlang R15B01 (erts-5.9.1).

For all simulations, we used 4 Erlang virtual machines (Erlang nodes), deployed as an Erlang cluster on one physical machine. Each Erlang Node is connected to all others nodes in a fully connected graph.

4 simulations were performed.

The first simulation always starts with a stable system with six (6) elements: One MA and five SEDs. From this stable system and for each $X \in \{5, 10, 50, 100, 200, 500, 600, 700\}$, X new SEDs are added in the system (the newly created SEDs are unstables and will try to become stables) and we counted the number of hops the system takes to recover a stable state. For each X, the simulation is executed 5 times and the Mean of the 5 values (number of hops) is computed. The Fig. 3 shows the ratio between the Mean value of hops and the corresponding X.

For the 2^{nd} simulation, we tried to have a stable system with a quiet large number of SEDs. For this purpose, this simulation always starts with a stable system obtained by adding 500 SEDs to an predefined stable system with six (6) elements: 1 MA and 5 SEDs. The resulting stable system is not always the same (even if two stable systems have the same number of objects, their topologies may be different), but it is obtained by the same operations. When a stable system is reached, then for each $X \in \{5, 10, 50, 100, 200, 500\}$, X SEDs are killed (randomly chosen in the system) and we count : the number of hops the system takes to recover a stable state,

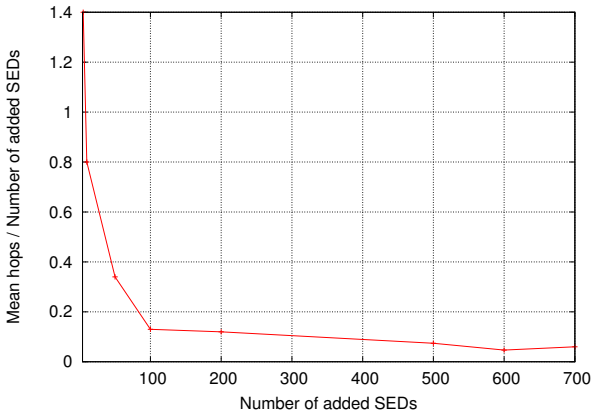


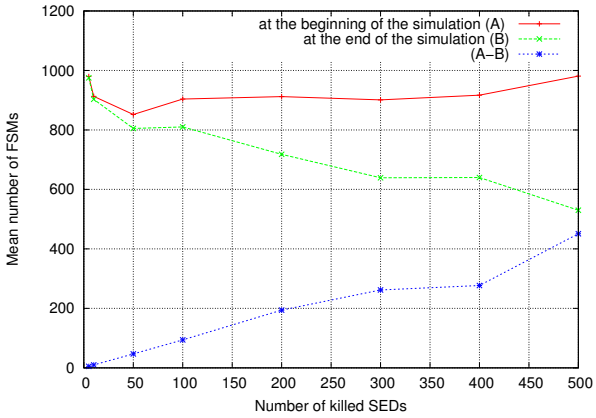
Fig. 3. Add X SEDs simulation: Ratio between Mean hops and X.

the total number of instances (also called FSMs in this paper) before the "kill *X* SEDs " action, the total number of instances when the system recovers its stable state after the "kill *X* SEDs " action (the end of the simulation). For each *X*, the simulation is executed 5 times and the Means of the 5 values (number of hops, number of FSMs at the beginning of the simulation, number of FSM at the end of the simulation) are computed. The Figure 4 show the results of this simulation. The curves in Fig. 4 (a) represent the mean of FSMs at the beginning of the simulation, at the end of the simulation, and the difference between these two values for each *X*. The curve in Fig. 4 (b) represents for each *X*, the ratio between the mean of hops and *X*.

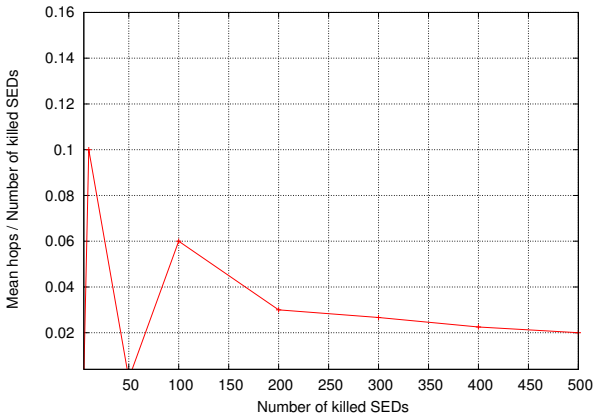
For the 3^{th} simulation, we start with a stable system obtained by adding 250 SEDs to an predefined stable system with six (6) elements: One MA and five SEDs. The resulting stable system have 408 instances. From this stable system, we kill 100 SEDs (randomly chosen) and wait until the system becomes stable again, then we add 100 SEDs, and we repeat these two simulation events one after another and always when the system is stable. The number of stable instances, the number of unstable instances are recorded every unit of time (hop). Fig. 5 shows the results of this simulation. The curve in Fig. 5 shows that, after every simulation action ("kill" or "add"), the system recovers a stable state, that is, the number of unstable instances is equal to zero.

The 4^{th} simulation is like the 2^{nd} with the following differences: Instead of killing *X* SEDs, it is *X*% of SEDs which are killed and $X \in \{5, 20, 25, 35, 40, 50, 65, 75, 80, 100\}$. For each $X \in \{5, 20, 25, 35, 40, 50, 65, 75, 80, 100\}$, *X*% of SEDs are killed; the simulation is executed 5 times and the following values are recorded: The number of hops the system takes to become stable; the total number of instances (FSMs) before the "kill *X*% SEDs " action; the total number of instances when the system recovers its stable state after the "kill *X*% SEDs " action.For each *X*, the mean of the 5 values obtained during the 5 simulations for each of the recorded parameters ((number of FSMs at the beginning of the simulation, number of FSMs at the end of the simulation)are computed.

Fig. 6 shows the results of this simulation.



(a) Mean of FSMs at the beginning of the simulation and at the end of the simulation.



(b) Ratio between the Mean hops and X

Fig. 4. Kill X SEDs simulation

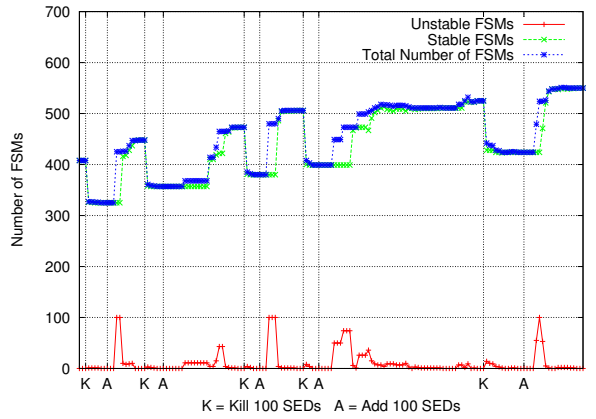


Fig. 5. Applying the two following actions: kill 100 SEDs and add 100 SEDs, one after another. After each action, wait until the system recovers before applying the following action.

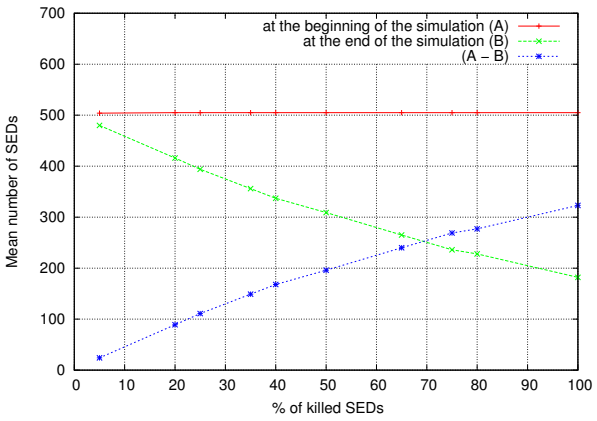


Fig. 6. Kill X percent SEDs simulation: Mean of SEDs at the simulation Start and at the simulation end.

V. Related work

Distributed software systems like middleware are usually deployed on distributed and dynamic infrastructures. Once a software system was deployed, then how it will adapt the changes from the infrastructure? A solution to overcome this challenge is to make these software systems autonomic (at design time or later by external mechanisms). Autonomic computing [9] is a vision to make a software self-adaptive [10]. A self-adaptive system have the capability to autonomously modify its behaviour at run-time in response to changes in its environment. The enforcement of this vision for software systems raises some challenges [11]. Like our algorithm which is intended to maintain a topology, many self-stabilizing topology control algorithms were proposed to construct and/or maintain differents topologies. We can cite [12] for graphs, [13] gives a solution for Delaunay graph construction, skip graphs are solved by [14], skip list in [15], trees are done in [16], hyperTree in [17], spanning tree [18] or ring [19].

Our work takes benefit of these previous works and adapt the methodology to a given middleware. Beyond the contribution for the DIET middleware, the reader should see this paper as a proof of concept and a way to begin from theoretical work and usage in a real environment.

VI. Conclusion

In this paper, we addressed the problem of adding self-adaptive capabilities to an hierarchical grid and cloud middleware. For this purpose, we have proposed a distributed self-adaptive algorithm. The algorithm objective is qualitative and effective in maintaining a stable deployment. The processes which execute the algorithm are able to self-adapt themselves when a faulty state is detected. To validate the self-adaptive algorithm, a simulator was designed to simulate the dynamic behaviour. The simulations show that the algorithm is self-adaptive with respect to the set of faults we used for simulation. Currently, processes rely on their local knowledge (as far as it is possible) to self-adapt themselves. However, they sometimes need to interact with the deployment server, which acts as a centralized resource discovery service. As future work, we plan to integrate a distributed resource discovery, for example , using gossiping algorithms. In addition to the simulations, which give us an idea of the algorithm convergence, we will prove that the algorithm is self-stabilizing [6], that is, following the defined rules, an unstable deployment can achieve stable state within an arbitrary but finite time.

Acknowledgement

The authors of this paper would like to thank the African Center of Excellency (CEA-MITIC) Project supported by World Bank and managed by University Gaston Berger through UFR SAT. Thanks to Pushpinder K. Chouhan for the review.

References

[1] E. Vidal, F. Thollard, C. De La Higuera, F. Casacuberta, and R. C. Carrasco, "Probabilistic finite-state machines-part ii," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 27, no. 7, pp. 1026–1039, 2005.

[2] E. Caron, F. Desprez, D. Loureiro, and A. Muresan, "Cloud Computing Resource Management through a Grid Middleware: A Case Study with DIET and Eucalyptus," in *CLOUD 2009: IEEE International Conference on Cloud Computing*, IEEE, Ed., Bangalore, India, September 2009, published In the Work-in-Progress Track from the CLOUD-II 2009 Research Track.

[3] E. Caron and F. Desprez, "DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid," *International Journal of High Performance Computing Applications*, vol. 20, no. 3, pp. 335–352, 2006.

[4] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova, "A GridRPC Model and API for End-User Applications," in *GFD-R.052, GridRPC Working Group*, jun 2007.

[5] E. W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control," *Communications of the ACM*, vol. 17, no. 11, pp. 643–644, 1974.

[6] S. Dolev, *Self-Stabilization*. Cambridge, MA, USA: MIT Press, 2000.

[7] F. Cesarini and S. Thompson, *Erlang Programming*. O'Reilly Media, Inc., 2009.

[8] D. Dhamdhere, S. R. Iyer, and E. K. K. Reddy, "Distributed termination detection for dynamic systems," *ParallelCompufing*, vol. 22, pp. 2025–2045, 1997.

[9] J. Kephart and D. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[10] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel *et al.*, "Software engineering for self-adaptive systems: A second research roadmap," in *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 1–32.

[11] L. T. MAZEIAR SALEHIE, "Self-adaptive software:landscape and research challenges," 2009.

[12] J. Ben-Othman, K. Bessaoud, A. Bui, and L. Pilard, "Self-stabilizing algorithm for efficient topology control in wireless sensor networks," *Journal of Computational Science*, vol. 4, no. 4, pp. 199–208, 2013.

[13] R. Jacob, S. Ritscher, C. Scheideler, and S. Schmid, "A Self-stabilizing and Local Delaunay Graph Construction." in *ISAAC*, ser. Lecture Notes in Computer Science, Y. Dong, D.-Z. Du, and O. H. Ibarra, Eds., vol. 5878. Springer, 2009, pp. 771–780.

[14] R. Jacob, A. Richa, C. Scheideler, S. Schmid, and H. Täubig, "SKIP+: A Self-Stabilizing Skip Graph," *J. ACM*, vol. 61, no. 6, pp. 36:1–36:26, Dec. 2014.

[15] T. Clouser, M. Nesterenko, and C. Scheideler, "Tiara: A self-stabilizing deterministic skip list and skip graph," *Theoretical Computer Science*, vol. 428, pp. 18–35, 2012.

[16] E. Caron, F. Chuffart, and C. Tedeschi, "When self-stabilization meets real platforms: An experimental study of a peer-to-peer service discovery system," *Future Generation Computer Systems*, vol. 29, no. 6, pp. 1533–1543, 2013.

[17] S. Dolev and R. I. Kat, "Hypertree for self-stabilizing peer-to-peer systems," *Distributed Computing*, vol. 20, no. 5, pp. 375–388, 2008.

[18] A. Kravchik and S. Kutten, "Time optimal synchronous self stabilizing spanning tree," in *Distributed Computing*. Springer, 2013, pp. 91–105.

[19] F. Ooshita and S. Tixeuil, "On the self-stabilization of mobile oblivious robots in uniform rings," *Theoretical Computer Science*, 2014.