SOS Cloud: Self-organizing Services in the Cloud

Bogdan Alexandru Caprarescu[1], Nicolò Maria Calcavecchia[2],
Elisabetta Di Nitto[2], and Daniel J. Dubois[2]

[1] West University of Timisoara
Faculty of Mathematics and Computer Science
bcaprarescu@info.uvt.ro
[2] Politecnico di Milano
Dipartimento di Elettronica e Informazione
{calcavecchia,dinitto,dubois}@elet.polimi.it

Abstract. Cloud computing is becoming an interesting alternative as a flexible and affordable on-demand environment for deploying custom applications in the form of services. This work proposes a bio-inspired, self-organizing solution to support the allocation and deallocation of virtual machines and the deployment of services on virtual machines in a cloud infrastructure. The goal is twofold: to meet the service level agreements and to minimize the number of required virtual machines.

Keywords: cloud computing, self-organization, autonomic computing.

1 Introduction

Cloud computing is a distributed computing paradigm with the objective of leveraging economies of scale in order to offer on-demand, flexible virtual resources. The advantage of running applications in a cloud environment is the fact that their execution is charged for the amount of resources actually used, thus reducing the cost of the initial investment and allowing applications to scale up and down in response to changing computational requirements.

A list of obstacles to the massive adoption of cloud computing was identified by [3]. The top two obstacles are particularly relevant for our work. The first one is that specific adaptation has to be provided at the application level in order to allow services to scale with the traffic demand; the second one is that services are usually bound to a single cloud provider since there are no leading interoperable standards yet. Moreover, typical solutions are centralized and thus unsuitable in contexts characterized by high dynamism, a high number of applications instances, and constraints that require fast decisions.

The SOS Cloud project aims to provide robust and scalable solutions for service deployment and resource provisioning in a cloud infrastructure. The goal is twofold: to meet the service level agreements (SLA) and to minimize the required cloud resources (i.e., virtual machines). To meet both quality and functional goals we borrow inspiration from the self-organizing systems in nature (e.g., ant

J. Suzuki and T. Nakano (Eds.): BIONETICS 2010, LNICST 87, pp. 48–55, 2012.

colonies, flocks of birds, school of fish) which benefit from built-in robustness and scalability and their goals emerge from the interactions among a myriad of individuals. With this idea in mind, in our approach each virtual machine will be instrumented with an autonomic layer that, through cooperation, will make decisions whether to instantiate new nodes or remove existing ones, deploy services on nodes, and route requests. Same as in biological self-organizing system, the global goals of the system are expected to emerge from local actions.

Existing bio-inspired contributions [1], [2], [6] are limited to the dynamic deployment of a set of services in a cluster with a fixed number of physical servers. Consequently, their systems cannot accommodate high traffic demands due to the limited amount of physical resources. Our work targets cloud environments and aims to provide a self-organizing algorithm that, not only allocates nodes to services, but also dynamically adjusts the number of nodes to accommodate high fluctuations in the request rate.

The remaining of this paper is organized as follows. Section 2 describes the context of this work as well as our assumptions and notations. Section 3 proposes a self-organizing solution to this problem while some preliminary simulation results are described in Section 4. Finally, Section 5 concludes the paper.

2 Context

The context of our problem involves four different actors and it is depicted in Figure 1: (i) *cloud providers* offer their computing infrastructures under the Infrastructure as a Service (IaaS) paradigm, (ii) *service providers* develop services to be deployed in the cloud and give the SLA that should be respected, (iii) the *cloud broker* deploys the services owned by many service providers on virtual machines rented from the cloud providers, finally (iv) *clients* access the services under the Software as a Service (SaaS) paradigm. Basically, the cloud broker is a company that offers Service Optimization as a Service (SOaaS) to service providers. Our long term goal is to develop a solution that allows the broker to rely on several cloud providers. However, this is outside the scope of this paper where only one cloud provider is considered.

For simplicity, we assume stateless, computation-intensive services without any form of composition; each service offers exactly one operation. Services are deployed within virtual computing environments (e.g., Amazon EC2 virtual machines). We will use the symbols s for a certain service, S for the set of deployed services and $|S|$ to denote the number of deployed services belonging to the set S. Specifically, for a service s we assume that SLA consists of two thresholds: the *maximum response time* for each request $\overline{R}_s$, and the *maximum rejection rate* $\overline{P}_s$ (number of rejected requests per time unit). A service request is considered rejected whenever it is not satisfied within $\overline{R}_s$ time units; in this case the request is discarded before the processing. The estimated processing time for service s is denoted with D_s. Either this value is given at deployment time by the service provider or it is computed at runtime. Finally, the number of requests for a service per time unit is called *request rate* and denoted with λ_s.

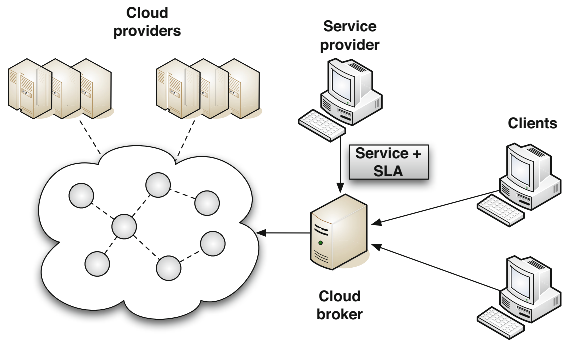

Fig. 1. Context of the SOS Cloud project

Each service may be deployed on several nodes. In this situation, we say that an instance of that service is running on each of these nodes. The cloud broker targets services with high request rates for which dozens of service instances are usually required. We assume that all nodes provided by the cloud infrastructure are homogeneous (e.g., same hardware capabilities) and that any service can run on any node. In the following, the symbol n will be used to denote a certain node. For security reasons (the services belong to different organizations) only one service is deployed on a node at a time.

3 A Self-organizing Solution

The cloud broker needs a system capable of continuously adjusting the number of nodes, the number of instances of each service, and the allocation of service instances to nodes in order to satisfy the SLA of each service and to minimize the number of nodes. Building such kind of systems that monitor and manage themselves according to certain objectives represents the goal of autonomic computing. As highlighted in our previous work [4], engineering robust and scalable autonomic systems continues to remain challenging. This is because many autonomic architectures employ a central manager that acts as a single point of failure and becomes a scalability bottleneck for large systems.

An approach to design robust and scalable large systems is to take inspiration from the self-organizing systems in nature like ant colonies. These systems are highly decentralized and their global goals emerge from the local interactions among a myriad of individuals. In our past work [5], we argued that self-organization can be successfully applied to build robust and scalable autonomic systems. We also observed that the systems composed of a high number of identical components are particularly suitable for being designed in a self-organizing manner. The broker's autonomic system exhibits this characteristic as it is composed of a large number of homogeneous nodes. Therefore, instead of having one or a few components that manage the whole system, we provide each node with an *autonomic layer* that applies local changes. The nodes self-organize and their autonomic layers collaborate to perform local optimization. The emerging global solution is usually not the optimal one, but the high robustness and scalability of the architecture pay the trade-off. In this section, we briefly describe the architecture and the self-organizing algorithm of the broker's autonomic system.

Like ants, our nodes have only a local view of the overall system. In other words, each node knows about and communicates with a limited number of other nodes in the system (called from now on the *neighbors* of that node). The nodes form $|S| + 1$ overlays: one overlay for each service and a system overlay which connects all nodes. Consequently, each node is simultaneously part of its service overlay and the system overlay. For example, Figure 2 depicts the overlays of a system consisting of two services (named *s1* and *s2*) deployed on nine nodes. The number of neighbors of each node in an overlay is the same and is called the degree of the overlay. As described below, the service overlays are needed for request routing and node provisioning while the system overlay is used only for service switching (i.e., the current offered service of a node is changed with another one without restarting the virtual machine). In the following, by *neighborhood* of a node we refer to the set composed of the node itself and its neighbors in a given overlay. As a node is part of two overlays, it is also part of two neighborhoods: service neighborhood and system neighborhood.

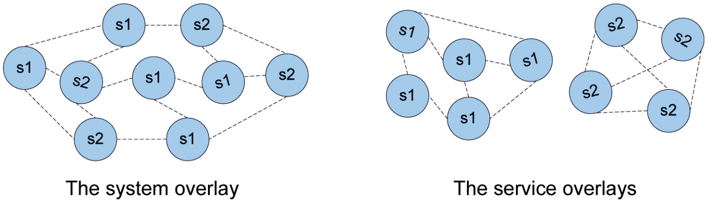

Fig. 2. Example of system and service overlays

From the architectural perspective, a node runs a service and is instrumented with an autonomic layer that executes three decentralized mechanisms: service optimization, overlays management, and request routing. Service optimization is the most important function of the autonomic layer. To fulfill it, the autonomic layer uses a feedback loop with three main activities: monitoring, analysis and decision, and execution. The autonomic layer stores performance data for the current node and for its neighbors in the service and system overlays. This data is then periodically analyzed and a decision is made whether a modification is needed. There are three possible modifications: allocate a new node in the cloud, remove the current node from the cloud, and change the service of the current node. In the following, the three feedback loop activities are described.

The autonomic layer monitors two parameters: the CPU utilization L_n and the number of rejected requests P_n. The CPU utilization is computed as the percentage of the number of requests processed by the application server in a time frame (called monitoring time frame) out of the total number of requests that could have been processed by the application server in the monitoring time frame (computed at its turn based on the estimated service processing time D_s). The number of rejected requests is computed over the same time frame. The autonomic layer also maintains one management table for each overlay. Thus, for each neighbor in one overlay, the management table of that overlay stores an entry with the following information: data needed to communicate with that neighbor (e.g., IP address and port number), CPU utilization, number of rejected requests, and the last update time. The purpose of the management tables is twofold: to maintain the overlays topology and to store performance data of neighboring nodes. In order to keep the management tables updated we use a gossip protocol based on T-Man [7]. More specifically, we customized T-Man in two ways. First, in order to increase the stability of the topology, when updating a management table we try to keep many of the existing neighbors. Second, we make sure that a new node is inserted in the service neighborhood of the node that decided to create it.

The autonomic layer needs a way to figure out how well the service neighborhood is performing with respect to the goals of the system. For doing that, a common practice in autonomic computing is to define a utility function that realizes a trade off among goals. The first goal of our system is to respect SLA, that is to minimize the number of rejected requests. In order for this goal to be measurable we define a reward function U^{SLA} that associates a numeric reward to each node based on the maximum allowed number of rejected requests defined in the SLA and the monitored number of rejected requests.

The second goal is to minimize the number of nodes. As a rule, the more efficiently the hardware resources of each node are utilized the less nodes are required. In this paper we assume that services are computation-intensive and focus on optimizing only one resource, namely the CPU. Normally, the minimum number of nodes is achieved when the CPU of all or most of the nodes is fully utilized. However, it is not a good practice to keep the nodes overloaded because, in case of a sudden increase in the request rate, the system would reject many requests before the autonomic layer has the chance to allocate new nodes. Therefore, we define a desired CPU loading L^{des} and a reward function U^{CPU} that associates a numeric reward to a node based on how close the current loading is from the desired loading. We say that a node is lightly utilized if the current loading is lower than the desired loading. Otherwise, it is heavily utilized.

The utility function can be further defined as a weighted average of the reward functions as shown in equation (1).

$$U(n) = w^{SLA} \cdot U^{SLA}(n) + w^{CPU} \cdot U^{CPU}(n), \text{ where } w^{SLA} + w^{CPU} = 1 \quad (1)$$

The utility can be computed not only at node level, but also for a neighborhood or even for the entire system as the average utility of all nodes in that neighborhood or system, respectively. However, optimizing the utility of a large system at the global level is usually an inefficient solution that may prevent the system from responding to traffic fluctuations in a timely manner. Therefore, we opted for a self-organizing algorithm where each node makes those decisions (i.e., add a new node, change its service or remove itself) that maximize the average utility of its service and system neighborhoods. To avoid conflicts, a dynamic and decentralized election algorithm allows only one member of a service neighborhood to execute changes at a time. In other words, while a node is effecting a change its service neighbors are forbidden to also execute changes.

Depending on the level of node utilization, a different algorithm is used for the analysis, decision, and execution activities of the autonomic layer. Thus, if the node is heavily utilized, the current average utility of the service neighborhood is compared with the predicted utility in the situation that a new node is added. If the predicted utility is higher, then a new node is allocated in the cloud. The prediction algorithm assumes that the new node will work at desired loading and will take over an equal amount of traffic from each existing neighbor.

The autonomic layer of a lightly utilized node estimates the utility of the neighborhood in the case the current node is removed. The estimation is done based on the assumption that the traffic processed by the removed node is equally distributed to its service neighbors. Here, by amount of traffic we mean the sum of processed and rejected requests. If the estimated utility is higher than the current utility, then, before removing itself, the node tries to switch to another service. The service switching mechanism works in the following way: for each service that is being run by its system neighbors, the autonomic layer estimates the utility of the system neighborhood in the event it would switch to that service. The node switches to the service that maximizes the predicted system neighborhood utility. But, if all predicted utilities are lower than the current utility, the node is deallocated from the cloud.

For request routing we use the simple algorithm proposed by [1]. First of all, we assume that the requests for a service are randomly distributed to the nodes of that service overlay. Then, once a request arrives at a node, based on the current loading and the estimated time to process the request D_s, the autonomic layer decides whether the SLA's maximum response time $\overline{P}_s$ can be met. If it can, then the node schedules the request on its internal queue. Otherwise, the request is forwarded to the least loaded neighbor in the service overlay. Finally, a request is rejected if there is no chance to handle it in the required time even by an unloaded node because of the time lost in the routing process.

4 Preliminary Results

A custom simulator was implemented in Java. The decentralized functions of each node (service optimization, request routing, and topology maintenance) are carried out by a couple of threads. A custom, asynchronous, message-based mechanism is used for implementing the inter-node communication. The simple utility function used in the simulator is shown in equations (2), (3), and (4). In equation (3), the definition of U^{SLA} uses the constant C to provide a dominant penalty in the situation the current number of rejected requests becomes higher than the double of the maximum allowed number of rejected requests. Other parameters of the simulator include the monitoring time frame (10 seconds) and the desired CPU loading ($L^{des} = 80\%$). The analysis is done every 0.5 seconds and both system and service overlay degrees are set to 10.

$$U(n) = \frac{U^{SLA}(n) + U^{CPU}(n)}{2} \tag{2}$$

$$U^{SLA}(n) = \begin{cases} \frac{\overline{P}_s - P_n}{\overline{P}_s} & \text{if } P_n < 2 \cdot \overline{P}_s \\ C & \text{otherwise} \end{cases} \tag{3}$$

$$U^{CPU}(n) = \begin{cases} \frac{L_n}{L^{des}} & \text{if } L_n < L^{des} \\ \frac{100 - L_n}{100 - L^{des}} & \text{otherwise} \end{cases} \tag{4}$$

Our preliminary tests addressed the provisioning of nodes as the effectiveness of a self-organizing service selection was proved by [1]. A simple test configuration of one client and one service deployed on one node was instantiated and the client was instructed to progressively send more requests. Figure 3 shows the variations of the system average utility as a response to traffic bursts. Thus, the client began by sending 500 requests per minute at a constant rate while a node at desired loading can process 96 requests per minute. The system responded by adding 4 new nodes, which raised the utility over 90. After 20 seconds, the request rate was increased to 2500 requests per minute. The number of nodes reached 25. Finally, after 60 seconds from the beginning, the request rate was set to 5000 requests per minute. As the end, 52 nodes were count.

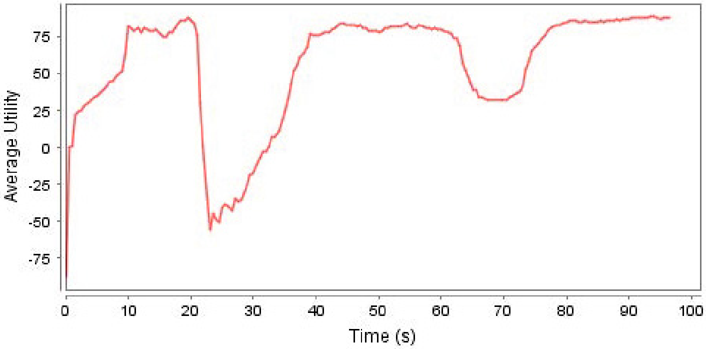

Fig. 3. System average utility (normalized between −100 and +100)

In conclusion, it was observed that the system is stable at constant request rates. When a traffic burst occurs, the utility initially drops, but quickly recovers after the allocation of new nodes. It is important to be aware that the decision of adding a new node can be made by any existing node. Moreover, while an existing node is adding a new node, its service neighbors are locked in the sense that they are not allowed to add other nodes.

5 Conclusion

In this paper, we have proposed an innovative self-organizing approach for service deployment and resource provisioning in a cloud infrastructure. The additional benefit of our approach, compared to other existing approaches, is the fact that we take advantage of the "elasticity" of the cloud in the sense that new resources may be allocated and deallocated to help services respect contractual SLAs. The proposed solution has been simulated and the preliminary results have shown that, after a transitory, services comply with their SLAs and the number of cloud nodes is comparable to the one obtained in the optimal solution (i.e., the solution that maximizes the utility function). Future work will include more precise and accurate simulations as well as a complete description of the architecture and algorithms of our system.

Acknowledgments. This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227077-SMScom, and grant FP7-ICT-2009-4-246839 (SPRERS).

References

1. Adam, C., Stadler, R.: A Middleware Design for Large-scale Clusters Offering Multiple Services. IEEE Transactions on Network and Service Management 3(1), 1–12 (2006)
2. Andrzejak, A., Graupner, S., Kotov, V., Trinks, H.: Algorithms for self-organization and adaptive service placement in dynamic distributed systems. HP Laboratories Palo Alto, HPL-2002-259 (2002)
3. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. Communications of the ACM 53(4), 50–58 (2010)
4. Caprarescu, B.A.: Robustness and scalability: a dual challenge for autonomic architectures. In: Proceedings of the Fourth European Conference on Software Architecture: Companion, pp. 22–26 (2010)
5. Di Nitto, E., Dubois, D.J., Mirandola, R.: On exploiting decentralized bio-inspired self-organization algorithms to develop real systems. In: Proceedings of the International Workshop on Software Engineering for Adaptive and Self-Managing Systems, pp. 68–75 (2009)
6. Jamjoom, H., Jamin, S., Shin, K.: Self-Organizing Network Services. University of Michigan, CSE-TR-407-99 (1999)
7. Jelasity, M., Montresor, A., Babaoglu, O.: T-Man: Gossip-based fast overlay topology construction. Computer Networks: The International Journal of Computer and Telecommunications Networking 53(13), 2321–2339 (2009)