Scalability of the Snooze Autonomic Cloud Management System

E. Feller*, C. Morin*, M. Simonin*, A-C. Orgerie** and Y. Jégou*

INRIA*, CNRS**, Myriads project-team, Rennes, France

1 Introduction

Large-scale data centers enabling today's cloud services are hosting a tremendous amount of servers and virtual machines (VMs). Such data centers impose a number of important challenges on their cloud management systems. First, cloud management systems must remain scalable with increasing numbers of servers and VMs. Second, with the probability of server failures increasing at scale (e.g. see past Amazon outages), cloud management systems must be highly available in order to enable continuous cloud service operation. Finally, at scale server administration becomes a demanding task and thus cloud management systems must be designed to be easily configurable. Such challenges call for scalable autonomic cloud management systems which are self-organizing and self-healing.

We focus on Infrastructure-as-a-Service (IaaS) clouds as they serve as the building block to enable most of the available cloud services, ranging from scalable web deployments to parallel data processing. A number of IaaS cloud management systems such as CloudStack, Nimbus, OpenStack, OpenNebula, VMware DRM and Eucalyptus have been proposed in the past. However, the former five systems are based on centralized architectures thus limiting their scalability. Indeed, in [5] the authors show that VMware DRM does not scale beyond 32 servers and 3000 VMs. Eucalyptus improves the scalability via a hierarchical architecture. However, it is not designed to be self-organizing and self-healing. Moreover, all the aforementioned systems require active/passive servers to achieve high availability thus wasting resources. Finally, the scalability and usability limitations of the open-source cloud management systems have been identified in [7].

To tackle these limitations, we have proposed the Snooze [4] IaaS cloud management system. Unlike existing systems, for scalability, ease of configuration, and high availability, Snooze is based on a *self-organizing and self-healing hierarchical architecture* of system services. No active/passive servers are required to achieve high availability. For the challenge, we present an extensive scalability study of Snooze across over 500 servers of the Grid'5000 experimentation testbed [1]. We evaluate the Snooze self-organizing and self-healing hierarchy with thousands of system services. We also demonstrate the application deployment scalability across hundreds VMs on the example of a Hadoop MapReduce application. To the best of our knowledge this is the first extensive evaluation of the scalability of an autonomous cloud management system.

The remainder of this document is structured as follows. Section 2 briefly describe the Snooze cloud management system. Section 3 presents the evaluation results. Finally, Section 4 concludes the document.

2 Snooze System Description

The architecture of the Snooze cloud management system is shown in Figure 1. It is composed of three layers: computing, management, and client. At the computing layer each server is managed by a Local Controller (LC) system service. The LC performs server and VM resource (i.e. CPU, memory, and network) utilization monitoring. Moreover, it allows to control the VM life-cycle (e.g. start, shutdown). Each LC is assigned to a Group Manager (GM) at the management layer. A GM receives VM resource utilization data from its assigned LCs and implements algorithms to place VMs on the servers managed by the LCs. A Group Leader (GL) exists to manage the GMs. It is elected among the GMs during the system boot phase and after a GL failure. The GL receives aggregated resource utilization data from the GMs. It also accepts client VM submission requests and dispatches them among the GMs based on a given VM dispatching policy. Finally, the GL is in charge of assigning joining LCs to the GMs based on a given LC assignment policy. To enable system services failure detection and recovery, bi-directional heartbeat protocols are implemented in all layers of the system. LCs and GMs send unicast heartbeats to the GMs and the GL, respectively. This way LC and GM failures can be detected. GMs and the GL send multicast heartbeats in order for the LCs and the GMs to detect GM and GL failures, respectively. In order for the clients to discover the current GL, one or multiple system services, called Entry Points (EPs) exist which listen for GL heartbeats. While the self-organization and self-healing mechanisms enable autonomous hierarchy construction and recovery, it is up to the system administrator to decide on the initial number of LC, GM, and EP system services during the deployment. System services can be dynamically added and removed at runtime.

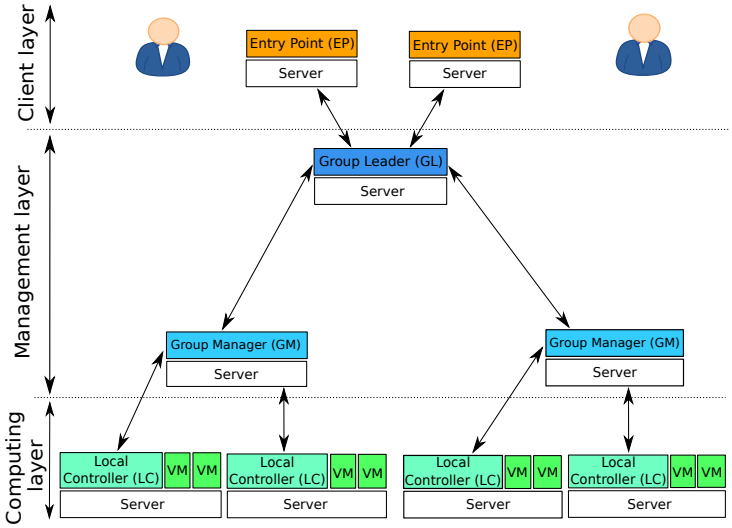

Figure 1: High-level system architecture

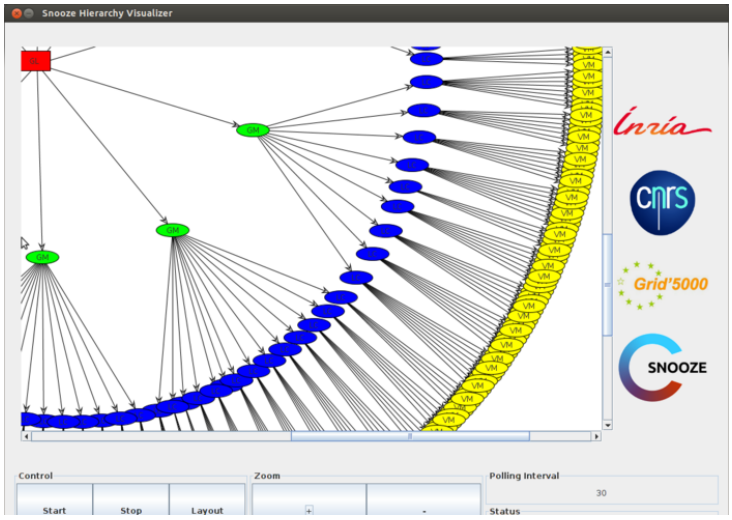

Figure 2: Real-time Graphical User Interface

3 Snooze Scalability and Autonomy Evaluation

We have deployed Snooze across five geographically distributed sites of the Grid'5000 experimentation testbed. The clusters were located in Rennes, Sophia, Nancy, Lyon and Toulouse. They were interconnected using 10 gigabit Ethernet links. To create the illusion of all the servers belonging to the same cluster, a global Virtual LAN (VLAN) was set up. Given the limited number of servers, to evaluate the scalability of the system we have deployed multiple system services per server. We evaluated the system setup time, self-organization and self-healing protocols, system services resource consumption, and the application deployment scalability.

3.1 System Setup Time

Evaluating the system setup time highlights the scalability of the deployment scripts. The system setup time is composed of two steps: (1) deployment of a Debian base image; (2) Snooze package installation, configuration and start. We used 538 servers to deploy Snooze with 1 EP, 40 GMs (including the GL) and 497 LCs. One system service per server was used. The deployment of the Debian base image was done using the Kadeploy3 [6] tool and took 30 minutes for 538 servers. The Snooze system services installation, configuration, and start were supported by the parallel remote execution tool called TakTuk [3] and took 15 minutes. This way 45 minutes were required to have a running Snooze deployment. A Snooze deployment is considered as running when all the system services are started.

3.2 Scalability of the Self-organization Mechanisms

We evaluate the hierarchy construction time and demonstrate the upper bound on the number of GM and LC system services Snooze can manage. We distinguish between three scenarios: (1) scalability of the GL; (2) scalability of a GM; (3) scalability of the whole hierarchy. To evaluate the first scenario we have deployed up to 5000 GMs on 100 servers with each server hosting 50 GMs. A similar experiment was done for the second scenario with up to 5000 LCs with each server hosting 50 LCs. In both experiments we measured the number of GMs (resp. LCs) which could join the system. In the third scenario we measured the time to construct the Snooze hierarchy with 1 GL, 1000 GMs, and 10000 LCs. LCs were assigned to GMs in a round robin fashion. We deployed 20 servers hosting 50 GM each, and 200 servers hosting 50 LC each. A dedicated server was hosting the GL. Table 1 depicts the number of system services which could successfully join the hierarchy in a given period of time. As it can be observed GMs join faster than LCs. This can be explained by the fact that the LC join procedure requires to contact both, the GL and the newly assigned GM. We observed that our prototype could handle up to approximately 4600 GMs and 4300 LCs per GL (resp. GM). Beyond this scale the GL and GM system services have experienced internal exceptions and networking timeouts. Finally, when considering the scalability of the whole hierarchy, all of the 1000 GMs and 10000 LCs could successfully join the hierarchy in 20 minutes.

3.3 Scalability of the Self-healing Mechanisms

Our evaluation focuses on the hierarchy recovery time. We distinguish between two scenarios: (1) GL failure; (2) GM failures. We first measured the GL election time and investigated the impact of the number of GMs on the GL

Topology / Time since start up	1 GL 5K GMs	1 GL - 1 GM 5K LCs	1 GL 1K GMs - 3K LCs	1 GL 1K GMs - 10K LCs
30 seconds	1485 GMs	509 LCs	980 GMs - 474 LCs	-
1 minutes	3861 GMs	1043 LCs	1000 GMs - 767 LCs	979 GMs - 482 LCs
3 minutes	4656 GMs	2520 LCs	1000 GMs - 2134 LCs	983 GMs - 1492 LCs
10 minutes	4689 GMs	2633 LCs	1000 GMs - 2657 LCs	1000 GMs - 7436 LCs
15 minutes	4645 GMs	4283 LCs	1000 GMs - 3000 LCs	1000 GMs - 9593 LCs
20 minutes	4629 GMs	4300 LCs	1000 GMs - 3000 LCs	1000 GMs - 10000 LCs

Table 1: System services join time

election time. Snooze was deployed with up to 1000 GMs on 20 servers each hosting 50 GMs and a GL failure was injected. The results show that the GL election time is about 10 seconds and does not depend on the number of GMs (see [4] for more details on GL election). GMs rejoin the new GL in less than 2 minutes (see Figure 3). To evaluate the hierarchy recovery time in the event of GM failures we deployed Snooze with 1 GL, 1000 GMs, and 3000 LCs. We used 20 and 150 servers with each of them hosting 50 GMs and 20 LCs, respectively. Failures were injected by terminating the GM services on up to 19 servers. We then measured the number of LCs on the last server hosting GM services. Figure 4 depicts the GM failures as seen by the GL and the number of LCs rejoining the remaining GMs. We observe that all the LCs rejoin the hierarchy quickly even when 95% of the GMs become unreachable in a short period of time. This result shows that Snooze is able to tolerate a large number of simultaneous GM failures. Indeed, after 20 minutes all LCs have rejoined the hierarchy.

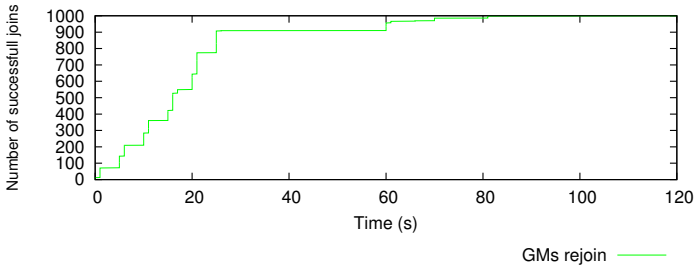

Figure 3: GMs rejoining after a GL failure

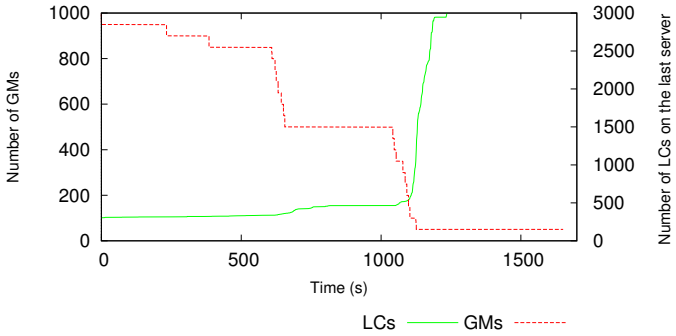

Figure 4: GMs failure and LCs rejoin over the time

3.4 System Services Resource Consumption

We evaluate the GM resource (CPU, memory, network) consumption as it gives an upper bound on the number of manageable LCs and VMs. The evaluation focuses on the resource consumption with increasing number of LCs and VMs. To evaluate the former aspect, we increased the number of LCs up to 1000 and measured the resource consumption at the GM server. We used 20 servers with each of them hosting 50 LCs. Figure 5 shows the GM resource consumption when increasing the number of LCs incrementally (1 LC per second) with 1 GL and 1 GM in the system. As it can be observed the GM network received traffic is proportional to the number of LCs. The send traffic remains low since it does not depend on the number of LCs. Only small spikes can be observed during the LC join period. Ultimately, only a fixed amount of aggregated resource consumption data along with heartbeat messages is sent every 3 seconds to the GL. No significant variation in CPU consumption is visible. Memory usage increases to about 220 MB. We believe that the memory usage increase is related to the memory and thread management of the Java Virtual Machine (JVM). The same kind of experiments were performed to evaluate the scalability of the GL in terms of the number of GMs it can manage. Similar observations were made on the GL when increasing the number of GMs. To evaluate the resource consumption on the GM with an increasing number of VMs, we have deployed Snooze with 1 GL, 1 GM and 20 LCs, each on a dedicated server. VMs were configured with 1 VCORE, 256 MB of RAM, and 5 GB of disk space. We used QCOW2 disk images which were hosted on a Network File System (NFS). We started the VMs in groups of 10 incrementally up to 1000 (see Figure 6). Each LC hosted approximately 50 VMs. As it can be observed the GM received traffic increases proportionally with the numbers of VMs, while the send traffic remains low. The CPU utilization remains under 1%. The memory usage increases up to 500 MB. Indeed, currently monitoring data is stored in-memory using a per VM ring buffer. Thus after some time oldest monitoring values are overwritten resulting in a flat memory usage curve for a fixed number of VMs.

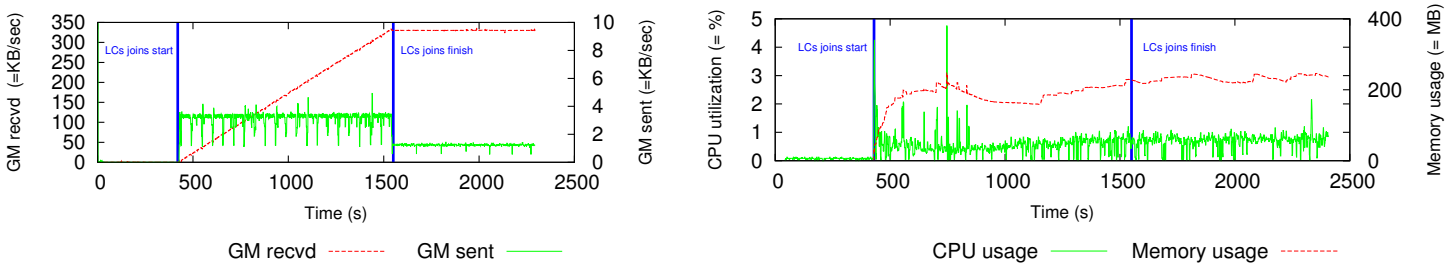

Figure 5: Resource consumption on the GM while increasing the number of LCs

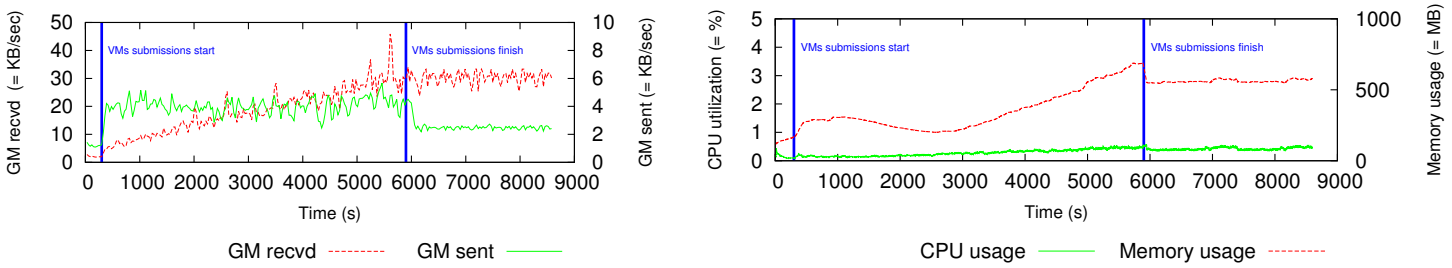

Figure 6: Resource consumption on the GM while increasing the number of VMs

3.5 Application Deployment Scalability: the Hadoop MapReduce Case

A key application of IaaS clouds is to enable Big Data analysis. Our evaluation focuses on three aspects: (1) Hadoop VM submission; (2) Hadoop MapReduce configuration; (3) Hadoop MapReduce application execution. Hadoop VM submission captures the time to propagate VM images to the servers and the time for the VMs to become accessible. We deployed Snooze with 1 EP, 3 GMs and 497 LCs (i.e. 3476 cores, 10 TB of RAM) each on a dedicated server. We then deployed 1000 VMs (3 VCORES, 8 GB of RAM and 50 GB of disk space) in five steps of 200 VMs. We used QCOW2 disk images which were propagated to the LC servers using SCP Tsunami [2]. We used Hadoop version 1.0.4. MapReduce was configured with two maps and one reduce slots per VM. HDFS block size was set to 128 MB. The VM submission took 30 minutes. At the end 917 out of 1000 VMs were submitted due to the heterogeneity of the servers. Hadoop MapReduce was configured on the 917 VMs using our scalable Hadoop configuration script. The configuration took 10 minutes. Ultimately, we executed the TeraGen and TeraSort benchmarks to measure the execution time. It took 3 minutes and 10.2 minutes for TeraGen to generate 100 GB and 1 TB of data, respectively. TeraSort required 9.5 minutes and 84 minutes to sort 100 GB and 1 TB of data, respectively.

4 Conclusions

We have experimentally validated the Snooze scalability. The results show that the resource consumption of the Snooze system services is bounded both during the hierarchy construction and system operation. We also show that Snooze prototype implementation is robust enough to manage thousands of servers and hundreds of VMs. Moreover, its autonomic behavior allows to achieve high availability in the presence of a large number of simultaneous system services failures. Indeed, as long as at least two GMs remain operational the system remains alive. Finally, Snooze has been successfully used to run data-intensive Hadoop MapReduce applications. Snooze software is available in open-source at `http://snooze.inria.fr`. It can be used either as IaaS cloud management system or a research testbed for VM management algorithms. In the demonstration we will show the Snooze self-organization and self-healing features in real-time using the GUI (see Figure 2) while executing a Hadoop MapReduce application.

References

[1] The Grid'5000 experimentation testbed. `http://www.grid5000.fr/`, 2013.
[2] The scp-tsunami tool. `http://code.google.com/p/scp-tsunami/`, 2013.
[3] B. Claudel, G. Huard, et al. Taktuk, adaptive deployment of remote executions. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09. 2009.
[4] E. Feller, L. Rilling, et al. Snooze: A Scalable and Autonomic Virtual Machine Management Framework for Private Clouds. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '12. 2012.
[5] A. Gulati, G. Shanmuganathan, et al. Cloud-scale resource management: challenges and techniques. In *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing*, HotCloud'11. 2011.
[6] E. Jeanvoine, L. Sarzyniec, et al. Kadeploy3: Efficient and Scalable Operating System Provisioning. *USENIX ;login:*, 38(1):38–44, Feb. 2013.
[7] K. Yelick, S. Coghlan, et al. The Magellan Report on Cloud Computing for Science. Tech. rep., U.S. Department of Energy Office of Science Office of Advanced Scientific Computing Research (ASCR), Dec. 2011.