From mini-clouds to Cloud Computing

Boris Mejías, Peter Van Roy
Université catholique de Louvain – Belgium
{boris.mejias|peter.vanroy}@uclouvain.be

Abstract

Cloud computing has many definitions with different views within industry and academia, but everybody agrees on that cloud computing is the way of making possible the dream of unlimited computing power with high availability. However, being a cloud computing provider seems to be reserved to very large companies that can achieve having a huge data center. The rest of the companies and institutions have to play the role of cloud users. We propose an architecture to organize a set of mini-clouds provided by different institutions, in order to provide a larger cloud that appears to its users as a single one. Such architecture requires self-managing behaviour in order to deal with the complexity of matching cloud users requests with the computing utility that mini-clouds of institutions can offer.

Index Terms: Self-management, Cloud-computing, System architecture.

1. Introduction

Cloud Computing is an active area in the IT industry for a couple of years already, calling immediately the attention of the research community thanks to its possibilities and challenges. Projects such as Reservoir [14], NanoDataCenters [10], XtreemOS [18] and OpenNebula [4] are just examples of the interest of the research community. However, defining cloud computing is not that simple, because there are many interpretations within industry and academia. Interpretations go from seeing cloud computing as ubiquitous computing, or that any application provided as a web service is said to be living in the cloud. Consistently with Berkeley's view of Cloud Computing [2], and sharing some of the conclusions of 2008 LADIS workshop [3], we see Cloud Computing as the combination of hardware and software that can provide the illusion of infinite computing power with high availability.

Large companies provide this illusion of infinite computing power by having real large data centers with software capable to provide access on demand to every machine on the data center. Industrial examples supporting Cloud Computing are Google AppEngine [6], Amazon Web Services [1] and Microsoft Azure [9]. This three companies follows the architecture described in [2] where the base of the whole system is such a large company being the cloud provider. Cloud users are actually smaller companies or institutions that use the cloud to become Software as a Service (SaaS) providers. The end user is actually a SaaS user, which is indifferent to the fact that a cloud is providing the computational power of the SaaS.

We assume such architecture for Cloud Computing to develop our proposal, which focuses on the interaction between the cloud user and the cloud provider. We delegate the interaction between SaaS provider and SaaS user to the design of the SaaS application itself.

When a SaaS provider decides to move its service to Cloud Computing, it has to deal with the API offered by the cloud provider. Since every cloud provider has its own API, it is not trivial to migrate from one cloud provider to another one. Another limitation we see in Cloud Computing as it is currently developed, is that only very large companies have the resources to become cloud providers. Middle to large companies with their own data center have not enough resources to indefinitely scale up independently. This is not a problem except when the service experience high peaks on their demand curve. Most of the time those data centers remain not far from idle. This is why Cloud Computing appears as a interesting solution.

Instead of focusing on companies, we consider academic institutions, such as universities, for our case study. These institutions usually have clusters and servers that most of the time are running at a low percentage of their full capacity, but they could run into problems when students generate a large demand of university's services. The same situation can occur to middle size companies. We consider that these institutions could provide a mini-cloud that alone is not enough, but that in combination with other mini-clouds can provide scalable resources on demand. We define a mini-cloud as a set of computers linked together within an institution, with the ability to provide services such as web pages and storage. A mini-cloud can‘ be one or more clus-

ters, or several computers within the same local area network.

We propose on this work an architecture that combines several mini-clouds to become a large decentralized cloud provider. The new cloud is interfaced to the SaaS provider with a manager layer that transparently handles its resource demands as if it were a single cloud provider. With respect to computing power, it is not the goal to provide the same amount of resources as a large cloud provider, which is prepared to host general-purpose cloud services to many companies using the same data center. The goal is to allow institutions to share their resources so as to use them when their demand is higher that what their mini-clouds are able to handle. As a consequence, the illusion of infinite computing power is given to a certain amount of institutions, instead of to an unbound amount of SaaS providers. The difficulty of the approach lies on the management interface, which has to be able to make the system *scale up and down*, depending on the demand of the system. We believe that scalability can be achieved in two ways: with a hierarchical and centralized structure being organized by a scheduler as in grid computing, or with a decentralized peer-to-peer network that can handle churn and scalable storage.

Scalability is certainly not the only challenge presented in cloud computing. Avoiding *data lock-in* is also very important to allow SaaS providers to migrate from one cloud provider to another at a low cost. To prevent data lock-in, and independently of the chosen strategy for organizing the nodes on the cloud, we identify the need of designing the management interface layer as a self-managing set of components that can follow a plan, but that it is also capable of self-adapting the plan according to the state of the service. Components can be reconfigured according to the adaptation plan or they can simply be replaced by other components.

In the next section we will present the general architecture to provide Cloud Computing with a set of mini-clouds. Section 3 discusses our strategy to design the interfacing layer, and we conclude in Section 4.

2. Cloud Computing with mini-clouds

The most general architecture to represent how Cloud Computing is provided by large companies, such as Google, Amazon or Microsoft, is analyzed in [2]. We can observe that architecture in Figure 1.a. The *cloud provider* is at the base of the architecture offering *utility computing* to the *cloud user*. Utility computing can be understood as a certain amount of resources during a certain amount of time, for instance, a web server running for one hour, or several Tera bytes of storage for a certain amount of days. The cloud user, which is actually a *SaaS provider*, has a predefined utility computing request, which can vary enormously depending on its users demands. At the top of the architecture we find the *SaaS user* which requests services from the SaaS provider. The service that the SaaS provider offers to its users is usually presented as a web *application*.

There are basically two things that are important to the SaaS provider: get more resources when users' demand increases more than what the current resources can handle, and release resources when the SaaS users are not demanding too much from the services. The objective is to maximize the quality of the service, and minimize the cost of utility computing demanded to the cloud provider. The system has to be able not only to scale up, but also to scale down. Idle resources are an unnecessary cost to pay.

The analysis made in [2] identifies 10 challenges on Cloud Computing. This proposal focuses on three of them: *data lock-in*, *scalable storage* and *scaling quickly*. We will discuss the challenges related to scalability in Section 3. Now we will see how a possible solution to the first challenge can help us to introduce mini-clouds in the architecture.

2.1. Abstracting the cloud provider

Data lock-in refers to the problem of SaaS providers of not being able to easily migrate from one cloud provider to another. This is because there is no common API for different cloud providers, and it is unlikely to expect the main corporations to agree on something like that. The Reservoir project [14] introduces a managing layer between the cloud provider and the cloud user. This layer has its own API to be used by the SaaS provider. The request for utility computing is managed in this layer which is in charge of using cloud provider's API. This adapted architecture is depicted in Figure 1.b. By abstracting the cloud provider, the application runs independently of the cloud provider behind the interface layer, reducing the problem of data lock-in. The issue of data lock-in cannot be entirely removed because it also depends on the functionality that the cloud provider can offer with its API. Even though, having a layer between cloud provider and user it is a great advantage.

2.2. Gathering mini-clouds

After abstracting the cloud provider behind the interface layer, we can replace the base of the architecture with whatever is able to provide a similar functionality of a cloud provider. Our proposal is to gather several mini-clouds from different institutions willing to collaborate in order to achieve a large amount of resources that can provide Cloud Computing to those institutions.

We consider the following scenario to motivate the possibilities of such system. Our university has a web service for students and the academic personnel to organize the ma-

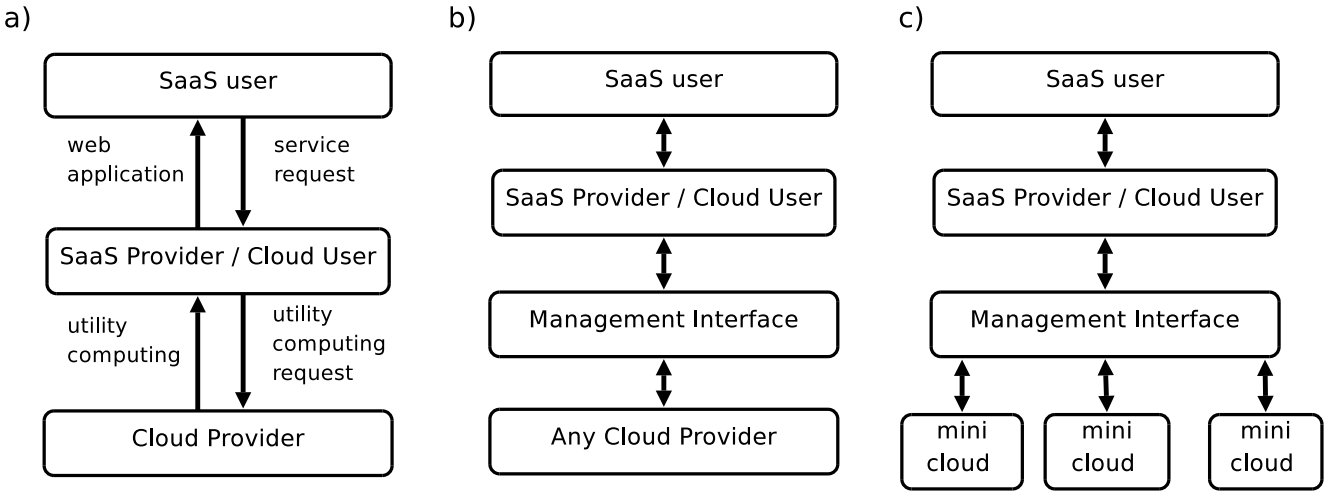

Figure 1. a) General Cloud Computing architecture with a single large cloud provider. b) Adding a managing layer that can interface any single large cloud provider. c) Replace the cloud provider with many mini-clouds.

terial and projects of every course. Having independent pages for every course has the inconvenience that each one of them has different layout scheme, navigation map, and different support for student collaboration. The web service instead, provide a platform to host every course with an equivalent scheme and functionality, so students can navigate and use it more efficiently. But having one single platform increases a lot the size of the system and the amount of users. We know that most of the time students make a light use of the service, but there are very identifiable peaks of use. For instance, there are more students visiting the courses at the beginning of the semester, but even more at the end, during the period of exams. There are small peaks when the deadline of a project is approaching and many students want to submit their files at same time. All these characteristics reflect that our scenario can be seen as regular web service that needs to optimize its resources to be able to handle peaks on demand, and to minimize the use of resources the rest of the time, when the system load is minimum. We assume that many universities have implemented their own platform to provide an equivalent service with equivalent characteristics. Since every university has already acquire the hardware to host these services, each of them can be considered a mini-cloud with limited capabilities to scale. Therefore, combining these mini-clouds to emulate a large cloud provider can increase the benefits of the everyone's infrastructure.

Figure 1.c depicts our proposal where the cloud provider is replaced by several mini-clouds. Comparing Figure 1.b with Figure 1.c we observe that it is indifferent to the SaaS provider what is providing the utility computing. Therefore, it is also possible to migrate not only from one cloud provider to a different one, but also from cloud provider to mini-clouds and vice-versa, without changing the SaaS application.

3. Self-managing interface

As we saw in the previous section, abstracting the cloud provider with an interface layer reduces the problem of data lock-in, and it allows us to introduce mini-clouds to behave as a cloud-provider. The new issue now is to deal with the higher complexity of designing the management layer with have to interface different APIs from different cloud providers or mini clouds. Using mini clouds raises also the issue of organizing distributed resources. Working with one single cloud provider is simpler because management can be done in a centralized manner, and the resources are usually in the same location, but our challenge is to organize the set of mini-clouds.

Even though the complexity is increased with the interface layer, it also gives other possibilities, specially with respect to scalability, which is part of the focus in our research. We have extensively studied structured overlay networks in the Selfman project [16], where peer-to-peer networks can scale well and quickly. Some of the networks developed in Selfman, Beernet [13, 8] and Scalaris [15, 12], provide not only self-organization of peers to deal with churn, but they also provide self-managing replicated storage. These networks are prepared to deal with unanticipated churn, because it cannot be known in advance when peers are going fail, join or leave the network.

Working with the cloud presents an important advantage with respect to churn. It is the cloud manager who decides when are the new nodes going to be aggregated, and when nodes can leave the network in order to released resources. Failures are obviously still unpredictable, but they can be more accurately detected, because the available resources are known in advance. Therefore, building a peer-to-peer network with cloud resources provides a self-organizing system with controlled churn, which can help to deal with the two of the challenges mentioned in [2]: scalable storage and scaling quickly.

3.1. Three-layer architecture

Due to the complexity of the interface layer, we identify the need for self-management in the design of it. For our proposal we use the three-layer architecture presented in [7], which is an adaptation of [5] applied to software design.

The architecture is depicted in Figure 2. At the bottom we find the *component control* layer, which communicates directly with cloud resources. This layer consists of components in charge of monitoring resources and triggering actions on them. This layer, and actually the whole architecture, is full of feedback loops [17] that constantly monitor the mini-clouds, analyze the information and decide on actions to affect the state of the cloud in order to achieve predefined goals. If we choose for a peer-to-peer architecture to organize the resources, peers are living on this layer.

The state of components running at the bottom of the architecture is reported to the *change management* layer. The interaction between these two layers can be seen as a meta feedback loop. Change management is constantly monitoring the component control to introduce changes whenever is needed. For instance, if a failure detector seems to trigger false suspicions too often, it could be reconfigured or replaced by another failure detector.

To analyze the top layer of the architecture we come back to our scenario of the web service provided by the university to administrate courses. Having logs of the web service it is possible to create a predefined plan of requesting and releasing resources from the cloud. This pre-planning would consider the schedule of the students on a daily basis, and it would take into account the yearly academic agenda to include exams periods on the demand of resources. The task of the *goal management* layer is to guarantee that the plan is going to be followed. Since pre-planning cannot be perfectly conceived, the layer must constantly monitor the system, being able to change the plan to deal with unexpected demand from the users.

Previous experience on adaptive planning systems gives us the intuition that the goal-management layer can be conceived with constraint programming. If the layer is to be

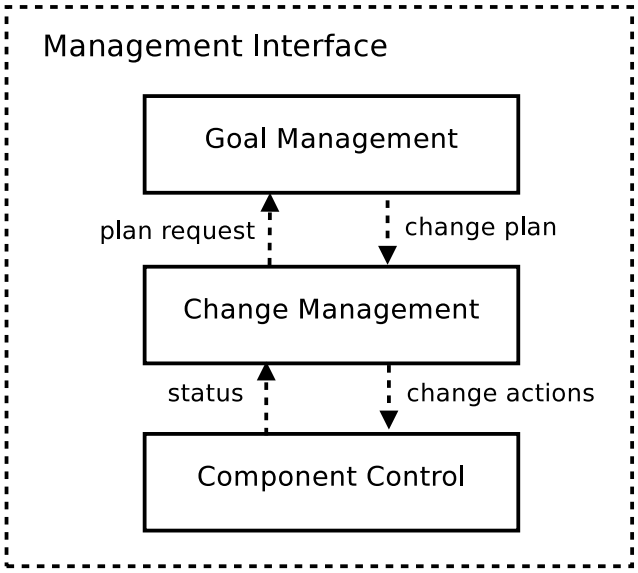

Figure 2. Three-layer architecture for a self-managing interface layer.

applied on a pay-as-you-go scheme, either for commercial cloud providers or as a strategy for fair use of mini-clouds, we can imagine many constraints such as the amount of *money* that can be spent, the maximum allowed delay to provide the service, or the resources that can be provided. Satisfying all these constraints is a constraint satisfaction problem (CSP). Trying to find an optimum way of satisfying such constraints is a constraint optimization problem (COP). Because our case study involves many entities that do not have a central point of control or global state, this lead us to a distributed constraint satisfaction problem (DCSP) and distributed constraint optimization problem (DCOP). We believe that DCSP and DCOP are the right paradigms to address the design of the goal-management layer on this three layer architecture. Furthermore, the system constantly changes the number of participants of the problem scaling up and down, and therefore, we can model it as Dynamic DCOP [11].

4. Conclusions

Research interest on cloud computing is constantly growing with different views within industry and academia. This work shares the view of a global architecture where a SaaS provider requests utility computing to a cloud provider. Such view presents many challenges from which we focus our proposal on three of them: reduce data lock-in, provide quick scalability and provide scalable storage. We start our proposal from the architecture that introduces a management interface layer between the SaaS provider

and the cloud provider, in order to make the application independent of the cloud provider. This abstraction allows as to replace the cloud provider with a set of mini-clouds that can be provided by different institutions for a global benefit.

To deal with the complexity of the middle layer we take inspiration from three-layer architecture to provide self-management. By combining these two ideas we believe that it is possible to get cloud computing out of mini-clouds. Since the resources of mini-clouds are distributed, we identify the need for decentralized management with self-organization, which can be provided by the inclusion of structured overlay networks. Our system becomes a scalable peer-to-peer network with controlled churn. We also propose the use of constraint programming solvers to achieve adaptable planning respecting the constraint on resource usage and quality of service.

5. Acknowledgments

The authors would like to thank Luis Quesada for helpful discussion on the architecture. This work has been supported by project SELFMAN.

References

[1] Amazon. Amazon web services. `http://aws.amazon.com`, 2009.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, Feb 2009.

[3] K. Birman, G. Chockler, and R. van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, 2009.

[4] Distributed Systems Architecture Research Group at Universidad Complutense de Madrid. Opennebula. `http://www.opennebula.org`, 2009.

[5] E. Gat. On three-layer architectures. In *ARTIFICIAL INTELLIGENCE AND MOBILE ROBOTS*, pages 195–210. AAAI Press, 1997.

[6] Google Inc. Google app engine. `http://code.google.com/appengine/`, 2009.

[7] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[8] B. Mejías and P. V. Roy. The relaxed-ring: a fault-tolerant topology for structured overlay networks. *Parallel Processing Letters*, 18(3):411–432, 2008.

[9] Microsoft Corporation. Azure service platform. `http://www.microsoft.com/azure/`, 2009.

[10] Nanodatacenters Partners. Nanodatacenters EU FP7 project media distribution. `http://www.nanodatacenters.eu`, 2009.

[11] A. Petcu and B. Faltings. Optimal solution stability in dynamic, distributed constraint optimization. In *IAT*, pages 321–327. IEEE Computer Society, 2007.

[12] S. Plantikow, A. Reinefeld, and F. Schintke. Transactions for distributed wikis on structured overlays. pages 256–267. 2007.

[13] Programming Languages and Distributed Computing Research Group, UCLouvain. Beernet: pbeer-to-pbeer network. `http://beernet.info.ucl.ac.be`, 2009.

[14] Reservoir Consortium. Reservoir: Resources and services virtualization without barriers. `http://www.reservoir-fp7.eu`, 2009.

[15] T. Schütt, F. Schintke, and A. Reinefeld. Scalaris: reliable transactional p2p key/value store. In *ERLANG '08: Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, pages 41–48, New York, NY, USA, 2008. ACM.

[16] Selfman Partners. Self management for large-scale distributed systems based on structured overlay networks and components. http://www.ist-selfman.org, 2009.

[17] P. Van Roy. Self management and the future of software design. In *Formal Aspects of Component Software (FACS '06)*, September 2006.

[18] XtreemOS Partners. XtreemOS: Enabling Linux for the grid. `http://www.xtreemos.org`, 2009.