Will Be Defined After the Brainstorming Phase*

Eirik Brandztæg[1,2] and Sébastien Mosser[1]

[1] SINTEF IKT, Oslo, Norway
[2] University of Oslo, Oslo, Norway
{firstname.lastname}@sintef.no

Built: February 15, 2012

Abstract. ∼150 words expected. Will also be defined after the brainstorming phase. Must mention *(i)*the problem, *(ii)*the actual contribution and *(iii)*the obtained results.

1 Introduction

I'll write the introduction afterwards, when the content of the paper will be fixed.

- Cloud–computing research field [2] Map with section 2 from mde
- Model–driven engineering applied to the cloud

2 Challenges in the cloud

To recognize challenges when doing cloud provisioning we used an example application [3]. The application (from here known as BankManager) is a featureless bank manager system written in Grails [9], it supports creating users and bank accounts, moving money between bank accounts and users. BankManager is designed but not limited to support distribution between several nodes. Some examples of provisioning topology is illustrated in Fig. 1, each example includes a browser to visualize application flow, front-end visualizes executable logic and back-end represents database. It is possible to have both front-end and back-end on the same node, as shown in Fig. 1(a). In Fig. 1(b) front-end is separated from back-end, this introduces the flexibility of increasing computation power on the front-end node while spawning more storage on the back-end. For application performing heavy computations it can be beneficial to distribute the workload between several front-end nodes as seen in Fig. 1(c), the number of front-ends can be linearly increased n number of times as shown in Fig. 1(d). BankManager is not designed to handle several back-ends because of relational model based database, but this can solved on a database level with master and slaves **source about master-slaves?** (Fig. 1(e)), although this is beside the

* This work is funded by the European commission through the REMICS project, contract number 257793, with the 7th Framework Program.

scope of this article but we wanted to show the possibility. We used bash-scripts to prototype full deployments of BankManager against Amazon AWS [1] and Rackspace [8] with a design with three nodes as shown in FIG. 1(c). From this prototype it became clear that there were several challenges that CloudML had to tackle:

- **Complexity**: The first challenge we encountered was to simply authenticate and communicate with the cloud. The two providers we tested had different approaches, AWS [1] had command-line tools built from their Java APIs, while Rackspace [8] had no tools beside the API language bindings. For Rackspace we decided to use the HTTP data transferring tool *curl* and communicate directly with their RESTful API
- **Technical competence**: Because of the complexity encountered in our scenario the second challenge was to understand how to operate against the command-line tools and public APIs. As this emphasizes the complexity even further it also stresses engineering capabilities of individuals executing the tasks to a higher technical level
- **Multicloud**: Once we were able to provision the correct amount of nodes with desired properties on the first provider it became clear that mirroring the setup to the other provider was not as convenient as anticipated. The first distinction was in the concrete differences between API callouts, even the node property structures and names were different
- **Reproducibility**: The scripts provisioned nodes based on command-line arguments and did not persist the designed topology in any way, this made topologies cumbersome to reproduce
- **Shareable**: Since the scripts did not remember a given setup it was very difficult to share topologies between coworkers, as this was important to exchange ideas
- **Robustness**: There were several ways the scripts could fail and most errors were ignored
- **Metadata dependency**: The scripts were developed to fulfill a complete deployment, and to do this it proved important to temporally save run-time specific metadata. This was crucial data needed to connect front-end nodes with the back-end node.

Our envision is to tackle these challenges by applying model-driven approaches and modern technologies. One example is to create a common model for nodes to justify ***multicloud*** *differences and at the same time base this on a human readable lexical format to resolve* ***reproducibility*** *and make it* ***shareable****. .*

3 Contribution

The concept and principles for CloudML is to be an easier and more reliable path into cloud computing for IT-driven businesses of variable sizes.

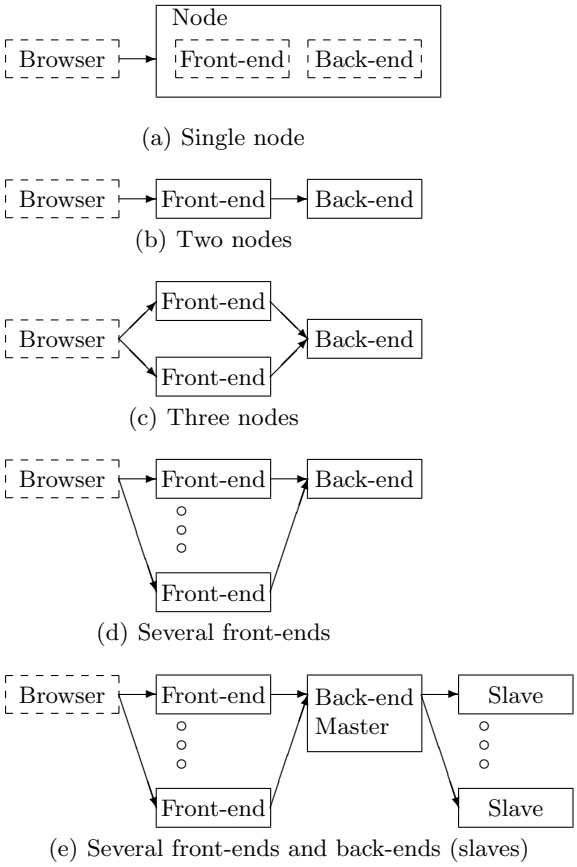

Fig. 1. Different architectural ways to provision nodes

CloudML is a lexical model-based language for cloud provisioning designed to solve the challenges presented in Sec. 2. The language has two main models shown in Fig. 2, *Account* and *Template*. The *Account* model is used for provider credentials to authenticate against a given provider such as Amazon or Rackspace. This model can be stored, shared and reused for the same provider several times. *Template* model on the other hand is used to define sets of resources **source** such as nodes or load balancers, also known as a *stack* **source**. This model can also be stored, shared and reused like the *Account* model, but it does not alternate between providers (*Accounts*), once a template is constructed it can be reused to create the same *stack* on other supported providers. It can even be reused to multiply a setup, without former knowledge of the system as a constraint which directly solved the problem of **reproducibility**. The two models use the same lexical syntax and therefore share the benefit of being **shareable** through mediums such as e-mail and can be properly used on version control systems such as Subversion or Git. The models are separated in the figure because combining them is semantically wrong, as they are only used together during building without any link to runtime models. Both models are regarded as building blocks in a model-based approach, given their distinct similarities to models described in *Virtual Deployment Model* by Konstantinou [7], only directed against provisioning instead of complete deployment. This model-based provisioning approach is beneficial in regard to the challenge of **technical competence**, since the models can be visualized with graphical models on tools such as whiteboards. This makes it easier for people with less technical knowledge to discuss, edit and even design propagation configuration. The template model of CloudML has a counter model, a model designed specifically for runtime environment. This model-at-runtime **source** provides information about nodes in form of properties, this data is essential to overcome the issue of being **dependant on data** from started nodes when performing cloud deployment.

CloudML is implemented as a proof of concept framework [4] (from here known as cloudml-engine*). Cloudml-engine is written in Scala, builds with Maven and can be used from Java. The templates from the contribution are constructed with JavaScript Object Notation (JSON). Clouml-engine use jclouds.org library to connect with cloud providers, giving it support for several providers out of the box to minimize* ***complexity*** *as well as stability and* ***robustness****. .*

4 Validation & Experiments

To validate how CloudML tackled the challenges we found in Sec. 2 we provisioned the BankManager application using different topologies Fig[1(a), 1(c)]. The implementation uses JSON to define templates (Fig. 2), so the lexical representation of Fig. 1(a) can be seen in Listing. 1.1. The whole text represents the *Tempalate* of Fig. 2 and consequently "nodes" is an array of Node from the model. The JSON is lexical **source: and human readable?** which makes it **shareable** as files, and once such a file is created it can be reused (**reproducibility**) on any given provider (**multicloud**).

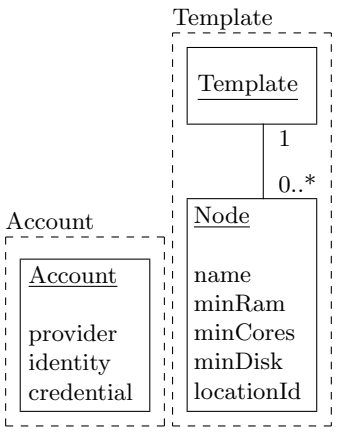

Fig. 2. Model of Account and Template

```
{
  "nodes": [ { "name": "testnode" } ]
}
```

Listing 1.1. One single node

More advanced topology such as FIG. 1(c) is represented in LISTING. 1.2, the main difference from LISTING. 1.1 is that there are more nodes with more properties. Traits of each node are carefully chosen based on each nodes feature area, for instance front-end nodes have more computation power, while the back-end node got more disk.

```
{
  "nodes": [
    { "name": "frontend1", "minRam": 512, "minCores": 2 },
    { "name": "frontend2", "minRam": 512, "minCores": 2 },
    { "name": "backend", "minDisk": 100 }
  ]
}
```

Listing 1.2. Three nodes

Provisioning nodes is by its nature an asynchronous action that can take minutes to execute, therefore we implemented the actors model [5] using Scala actors. With this asynchronous solution we get concurrent communication with nodes under provisioning. We extended the model by adding a callback-based pattern allowing each node to provide information on property and status changes. Developers exploring our implementation can then choose to "listen" for updating events from each node, and do other jobs / idle while the actors are provisioned. We have divided the terms of a node before and under provisioning, the essential is to introduce models-at-runtime to achieve a logical separation. When a node is being propagated it changes type to *RuntimeInstance*, which can have

a different *state* such as *Configuring*, *Building*, *Starting* and *Started*. When a *RuntimeInstance* reaches *Starting* state the provider has guaranteed its existence, including the most necessary metadata, and the task of provisioning is concluded. But when provisioning **here be talk about deployment?**.

According to Ilham [6] "java is gaining popularity in software development." and then continues to state that "it is widely used in network computing and embedded systems". Because of this we prototyped Java to interact directly with the Scala actors model and successfully confirmed this to be no obstacle, including the callback-based pattern.

5 Related Works

- AWS CloudFormation (Amazon only)
- jclouds (Only through (more advance?) code)
- libcloud (Only through (more advance?) code)
- CA Applogic (only graphical, and inhouse)

6 Conclusions

I'll write the conclusions afterwards.

References

1. Amazon: Amazon web services (2012), http://aws.amazon.com/
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia, M.: Above the Clouds: A Berkeley View of Cloud Computing. Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley (Feb 2009), http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html
3. Brandtzg, E.: Bank manager (2012), https://github.com/eirikb/grails-bank-example
4. Brandtzg, E.: cloudml-engine (2012), https://github.com/eirikb/cloudml-engine
5. Haller, P., Odersky, M.: Actors that unify threads and events. Tech. rep.
6. Ilham, A., Murakami, K.: Evaluation and optimization of java object ordering schemes. In: Electrical Engineering and Informatics (ICEEI), 2011 International Conference on. pp. 1 –6 (july 2011)
7. Konstantinou, A.V., Eilam, T., Kalantar, M., Totok, A.A., Arnold, W., E.Snible: An architecture for virtual solution composition and deployment in infrastructure clouds. Tech. rep., IBM Research (2009)
8. Rackspace: Rackspace cloud (2012), http://www.rackspace.com/cloud/
9. SpringSource: Grails (2012), http://grails.org