ICT-287510
RELEASE
A High-Level Paradigm for Reliable Large-Scale Server Software
A Specific Targeted Research Project (STReP)

D4.2 (WP4): Homogeneous Cluster Infrastructure

Due date of deliverable: 30th June 2012
Actual submission date: 16th July 2012

Start date of project: 1st October 2011 Duration: 36 months

Lead contractor: Erlang Solutions Ltd. Revision: 1.1

Purpose: To build and describe a framework that enables automatic deployment of an Erlang/OTP application into a given number of Erlang nodes running in a homogeneous cloud environment.

Results:

- We developed a Command Line Interface (CLI) which can be used to automatically deploy a given Distributed Erlang Application into a set of N machines running in the Amazon EC2 cloud environment
- We have run EC2 clusters of Erlang nodes within a secure environment, by using the Amazon *Virtual Private Cloud* (VPC) functionalities.

Conclusion: The approach we propose to automate deployment of an Erlang OTP application into a homogeneous cloud environment worked smoothly. We now face two major challenges, which will be tackled as part of the next deliverables: moving from a homogeneous environment to a heterogeneous one and adding cluster control and monitoring functionalities to the framework.

Project funded under the European Community Framework 7 Programme (2011-14)			
Dissemination Level			
PU	Public		⁕
PP	Restricted to other programme participants	(including the Commission Services)	
RE	Restricted to a group specified by the consortium	(including the Commission Services)	
CO	Confidential only for members of the consortium	(including the Commission Services)	

Homogeneous Cluster Infrastructure

Roberto Aloi <roberto.aloi@erlang-solutions.com>
Torben Hoffmann <torben.hoffmann@erlang-solutions.com>

Contents

1 Executive Summary 3

2 Introduction 3

3 A Scalable Virtualization Infrastructure 4

4 CCL /sısılı/ 5

5 Implementation 6
- 5.1 Amazon VPC . . . 6
- 5.2 erlcloud . . . 6
- 5.3 Road to a cluster of Erlang Nodes . . . 6
- 5.4 Amazon VPC Limitations . . . 8

6 Automatic Deployment 8
- 6.1 Installation . . . 9
- 6.2 Execution of user-defined commands . . . 10
- 6.3 Cluster Configuration . . . 11
- 6.4 Automatic Failover and Takeover Strategies . . . 11
- 6.5 Starting the application . . . 12
- 6.6 cloud-init Limitations . . . 12

7 Future Work 12
- 7.1 Leveraging the current architecture . . . 12
- 7.2 Planned Features . . . 12
- 7.3 Moving Towards a Heterogeneous World . . . 13

8 Conclusions 13

A The CCL Configuration File 15

B Moebius: Continuous Integration as a Service 16
- B.1 The Moebius Architecture . . . 16
  - B.1.1 Frontend . . . 16
  - B.1.2 Database . . . 17
  - B.1.3 Backend (Master) . . . 18
  - B.1.4 Agent . . . 19

B.2 Current Feature Set 19
B.3 Moebius vs Travis CI 22

Abstract

The main objective of this deliverable (D4.2) is to provide an infrastructure which, given a Distributed Erlang Application[1] *AppX*, allows the user to automatically deploy the application into a fully-connected cluster of N Erlang nodes running in a homogeneous cloud environment. For the sake of this deliverable, we assume that the Erlang Application uses the Erlang Distribution Protocol[4] as the main mean of communication between the Erlang nodes composing the cluster. We initially focus our attention uniquely on Erlang Applications built following the *OTP* principles and conventions[6] and based on the *rebar*[20] building tool. We utilize the Amazon EC2 web service[12] as the Cloud Service Provider for our homogeneous infrastructure.

This deliverable represents the first step towards a more generic broker layer capable of creating, managing and dynamically scaling heterogeneous clusters of Erlang nodes, based on capability profile matching, as pictured in the WP4 section of the Release project.

1 Executive Summary

The Statement of Aims for our Scalable Virtualization Infrastructure (*SVI*) is to help build, test, deploy and operate distributed scalable systems written in Erlang. This serves the needs of development teams building scalable systems and allows companies to improve their time-to-market, quality, customer engagement and overall development flow.

According to our original plan, we wanted to cover all the four stages of a systems life-cycle through our SVI: development, testing, deployment and operations. Therefore, we started developing *Moebius*, a tool aimed at providing large-scale Continuous Integration (*CI*) as a service. The simultaneous raise of *Travis CI* as the standard de-facto for *CI as a service* convinced us to slightly modify our original plan and to focus on the deployment and operations aspects of a scalable system. We decided to name our new infrastructure *Cloud Computing Lace* (in short, *CCL* /sIsIlI/).

As the first step towards a fully-functional Scalable Virtualization Infrastructure, we decided to design and build a tool which eases automatic deployment of an Erlang/OTP application into a given number of Erlang nodes running within a homogeneous cloud environment. We developed a Command Line Interface (*CLI*) which can be used to deploy and configure a given Distributed Erlang Application into a set of N machines running in the Amazon EC2 cloud environment. To ensure that the EC2 clusters of Erlang nodes run within a secure environment, we utilized the Amazon Virtual Private Cloud (*VPC*) functionalities. To interact with the Amazon Web Services (*AWS*), we used a modified version of the *erlcloud* libraries. The CLI offers several commands to the user who can, among the other things, execute bulk commands on the Erlang nodes composing the cluster or configure takeover and failover strategies.

Our approach to automate deployment of an Erlang OTP application into a homogeneous cloud environment worked smoothly. We now face two major challenges, which will be tackled as part of the next deliverables: moving from a homogeneous environment to a heterogeneous one and adding cluster control and monitoring functionalities to the framework.

2 Introduction

Deliverable D4.2 is part of the Work Package *WP4* of the *Release* project. The Work Package, titled *Scalable Virtualization Infrastructure*, is aimed at providing a broker layer capable of creating, managing and dynamically scaling heterogeneous clusters, based on capability profile matching. More specifically, deliverable D4.2 focuses on homogeneous environments, as opposed to the heterogeneous environments that will be analyzed as part of deliverable D4.3. It also assumes a static configuration for the cluster (e.g. in terms of nodes composing the cluster), rather than relying on a dynamic, capability-driven approach which will be considered during deliverable D4.4.

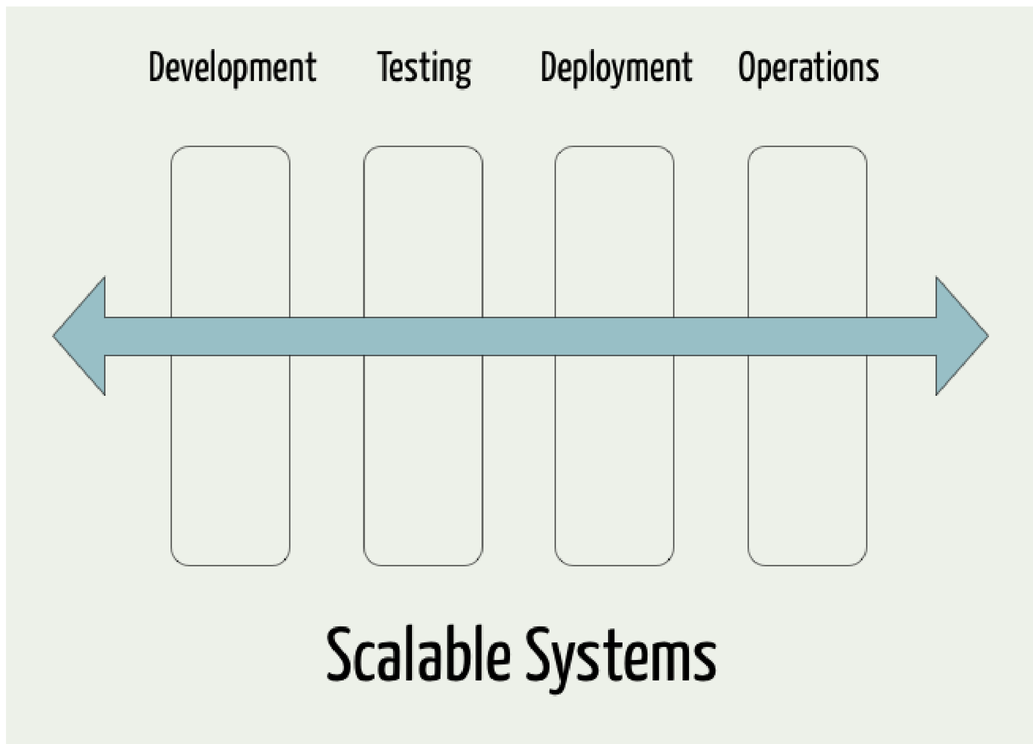

Figure 1: Scalable Systems

While working on the deliverable, involvement of project partners has been extremely precious. Worth to mention is the collaboration - mostly happened in the form of remote brain-storming sessions -, with the team from Heriot-Watt University. The collaboration mainly regarded the usage of *s-groups* within our *Scalable Virtualization Infrastructure*. During the meeting in St. Andrews (Scotland), progresses and future plans about *WP4* and the SVI have been presented to the Release partners, who provided invaluable feedback and revealed available for further collaborations.

3 A Scalable Virtualization Infrastructure

The Statement of Aims for the *Scalable Virtualization Infrastructure* discussed in WP4 is to help build, test, deploy and operate distributed scalable systems written in Erlang. Our Scalable Virtualization infrastructure aims at servicing the needs of development teams building scalable systems, allowing companies to improve their time-to-market, quality, customer engagement and overall development flow.

In figure 1 we list the four main stages of a system's life-cycle. Being able to build scalable systems implies that the required tools and flows are in place for all of the listed stages:

1. Development
2. Testing
3. Deployment
4. Operations

According to our original plan, we wanted to cover all the four stages of a system's life-cycle through our Scalable Virtualization Infrastructure. Therefore, we started developing *Moebius*, a tool aimed at

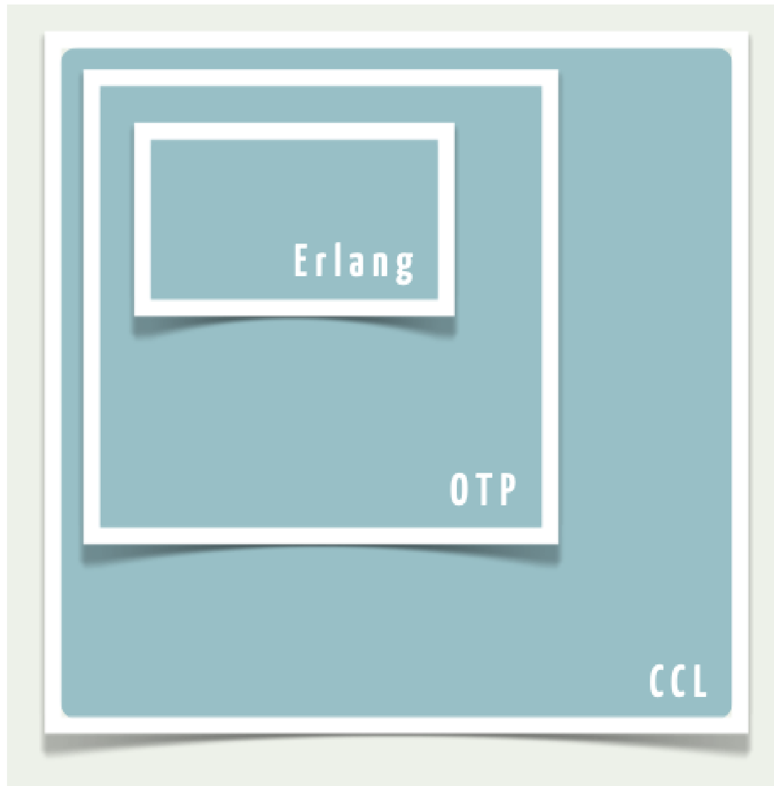

Figure 2: A Scalable Virtualization Infrastructure

large-scale Continuous Integration, where the main idea was to serve Continuous Integration as a service. Unfortunately (or fortunately) the simultaneous raise of Travis CI[1][7] as the standard de-facto for Continuous Integration (CI) as a service convinced us to slightly modify our original plan and to focus on the deployment and operations aspects of a scalable system. For more details about Moebius, its architecture and the current status of its development and for more information on how Moebius relates to Travis CI, please refer to Appendix B.

4 CCL /sɪsɪlɪ/

We decided to name our Scalable Virtualization Infrastructure *CCL*, pronounced /sɪsɪlɪ/, which stands for *Cloud Computing Lace*.[2]

CCL represents a layer which sits on the top of both Erlang and OTP, enabling the developer to build its distributed Erlang application without pain. When dealing with distributed applications, Erlang assists the developer at *language* level, providing programming paradigms and syntactical structures. OTP does the same, but it acts at a *node* level, with its libraries and patterns (i.e. *behaviours*). CCL tries to jump to an even higher level, allowing the developer to deploy his distributed applications and to manage *clusters* in a twist. Please note that, when we refer to a *distributed application*, we always assume to talk about a Distributed *Erlang* application, built according to the OTP[6] principles and conventions and based on the *rebar*[20] building tool.

[1] Travis CI is a hosted continuous integration service for open source projects. It supports Ruby, PHP, Python, Java, Node.js, and many more.

[2] Despite of the official acronym, CCL also refers to Aristophanes' play *The Birds*. During the play, the two major characters attempt to erect "a perfect city in the clouds", to be named "Cloud Cuckoo Land". That story didn't end too well and the characters were mutated into birds. We hope the same doesn't happen to us.

5 Implementation

5.1 Amazon VPC

The native Erlang Distribution mechanism has been entirely designed to run within trusted environments. The authentication model used is trivial and based on a shared secret, better known as the *Erlang cookie*. Erlang *nodes* exchange messages using mere TCP/IP sockets. All these reasons make the use of the Erlang Distribution mechanism *unsafe* in a cloud environment such as the Amazon EC2 one, where an Erlang cluster could be subjected to a variety of attacks, including forgery, interception and tampering of Erlang messages[19]. Even if several attempts have been conducted in the past decade to secure distributed communication in Erlang[9], none of these reached that level of maturity required by a production system. This is why we needed a way to isolate our cluster of Erlang nodes in a *corner* of the Amazon EC2 cloud.

Fortunately for us, Amazon has recently introduced a new important feature for its EC2 Web Service: *Amazon Virtual Private Cloud* (Amazon VPC), a tool which lets you provision a private, isolated section of the Amazon Web Services (AWS) Cloud where you can launch AWS resources in a virtual network that you define. With Amazon VPC, you can define a virtual network topology that closely resembles a traditional network that you might operate in your own datacenter. You have complete control over your virtual networking environment, including selection of your own IP address range, creation of subnets, and configuration of route tables and network gateways. Additionally, you can create a Hardware Virtual Private Network (VPN) connection between your corporate datacenter and your VPC and leverage the AWS cloud as an extension of your corporate datacenter[10]. This seemed exactly what we needed for our Erlang Scalable Virtualization Infrastructure.

Figure 3 shows an example of *VPC*(1), which is an isolate portion of the AWS cloud. The VPC is connected to the Internet and to other AWS resources via an *Internet Gateway*(2). The VPC contains a *Subnet*, which is a segment of a VPC's IP address range. Subnets enable you to group instances based on your security and operational needs. *Routing Tables and Routing Rules*, represented by *R* in figure, enable traffic to flow between the Subnet and the Internet (4), while *Security Groups* (5)[3] control the inbound and outbound traffic. (6) represents an *instance* running inside the VPC. In addition to their private address, instances may have an *Elastic Public IP Address* assigned (7).

5.2 erlcloud

Sadly, the introduction of the Amazon VPC service was so recent that there were no Erlang libraries to interact with it at the moment of this writing. We therefore decided to use *erlcloud*, a popular open source Cloud Computing Library for Erlang[17], as a basis for our needs and to extend it to support the Amazon VPC functionalities. We've then contributed to the erlcloud project with about 500 lines of code, implementing a good section of the Amazon VPC APIs. Our changes are currently kept on a separate fork of the project, given that we focused only on those APIs which were strictly relevant to our use case, but we plan to complete the full set of VPC APIs and to push our contributions back to the original fork of the project very soon.

5.3 Road to a cluster of Erlang Nodes

The first step towards the programmatical creation of a cluster of Erlang nodes was to spawn a set of Amazon instances running inside the same VPC. This involved the following steps:

1. Sign up for Amazon VPC

[3] *VPC Security Groups* are more powerful than standard *EC2 Security Group*, since they offer some extra functionalities and more granular control. For an analysis on how VPC Security Groups compare to EC2 Security Groups, please refer to [14].

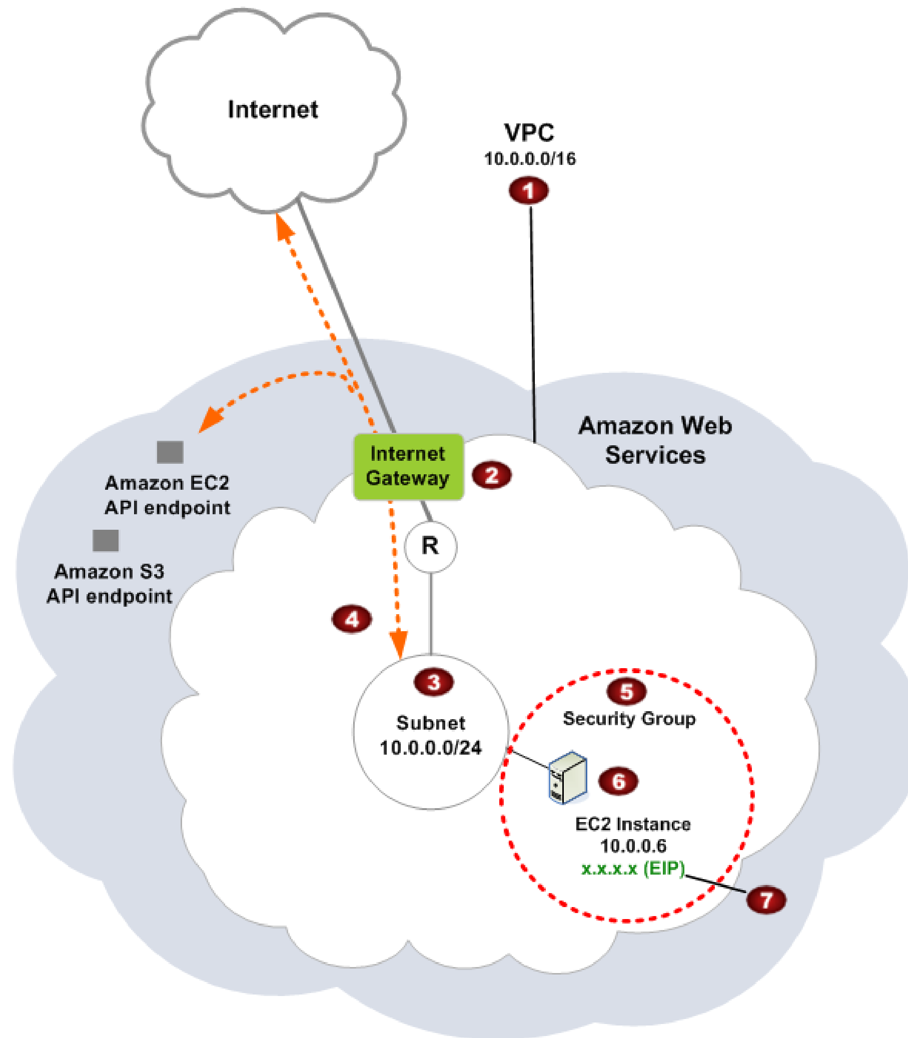

Figure 3: Amazon VPC

2. Set Up a VPC and an Internet Gateway
3. Set Up a Security Group
4. Launch one or more instances in the VPC
5. Assign an Elastic IP Address to one or more instances

Assuming that the user already owns a valid Amazon VPC Account, all the above steps can be achieved using CCL via the following one-liner instruction:

```
rel/ccl_cli/bin/ccl deploy --config my.config
```

All the information required for the deployment, such as the Amazon EC2 credentials for the user, the Amazon Image Id (AMI) and the dimension of the cluster can be specified in a configuration file (*my.config* in the above example). Let's consider the following configuration extract, which would be sufficient to spawn a cluster of *two* Erlang nodes within a newly created Virtual Private Cloud.

Figure 4: Creating a VPC in 5 steps

```
%% -*- Erlang -*-

%% EC2 Account Information
{ec2_hostname, "ec2.eu-west-1.amazonaws.com"}.
{ec2_accesskey, "YOUR_EC2_ACCESS_KEY"}.
{ec2_secretkey, "YOUR_EC2_SECRET_KEY"}.
{ec2_ami, "ami-123456"}.

%% CCL Configuration
{ccl_clustersize, 2}.
```

The above command, together with the given configuration, would produce an output resembling the following.

```
[...]
15:07:27.620 [info] ===> VPC created ("vpc-18282b71")
15:07:27.976 [info] ===> Subnet created ("subnet-1f282b76")
15:07:28.134 [info] ===> Internet Gateway created ("igw-10282b79")
15:07:28.299 [info] ===> Internet Gateway ("igw-10282b79") attached to VPC ("vpc-18282b71")
15:07:28.458 [info] ===> Route Table created ("rtb-12282b7b")
15:07:29.094 [info] ===> Security Group created ("sg-b5110fd9")
15:07:29.280 [info] ===> SSH Access allowed ("sg-b5110fd9")
15:07:29.896 [info] ===> Starting Instances (["i-7500643d","i-7700643f"])
15:08:22.372 [info] ===> Instance Created ("i-7500643d", "176.34.141.142").
15:08:31.479 [info] ===> Instance Created ("i-7700643f", "176.34.141.144").
[...]
```

Please refert to Appendix A for a list of all the available CCL configuration options.

5.4 Amazon VPC Limitations

Amazon Virtual Private Clouds come at no extra cost, exception made from when you need to setup VPNs to connect multiple VPCs. Nonetheless, there are some limitations regarding Amazon VPCs one should be aware of. We list the limitations in table 1. It is possible to request Amazon to increase most of these limits using the Amazon VPC Limits form[11]. Such limits need to be carefully considered at design time.

6 Automatic Deployment

In software engineering, the term *deployment* refers to all of those activities which are required to make a software system available for use. According to our vision, most of these activities are mechanical and repetitive and they leave room to a good level of automation. This is particularly true for an

Component	Limit	Comments
Number of VPCs per region	5	
Number of subnets per VPC	20	
Number of Internet gateways per region	5	One per VPC
Number of virtual private gateways per region	5	One per VPC
Number of customer gateways per region	50	
Number of VPN connections per region	50	Ten per virtual private gateway
Number of route tables per VPC	10	Including the main route table
Number of entries per route table	20	
Number of VPC Elastic IP addresses per AWS account	5	You have one limit for VPC Elastic IP addresses (5) and another for standard EC2 addresses (5)
Number of VPC security groups per VPC	50	
Number of rules per VPC security group	50	
Number of VPC security groups a VPC instance can be in	5	
Number of network ACLs per VPC	10	
Number of rules per network ACL	20	
Number of BGP Advertised Routes per VPN Connection	100	

Table 1: Amazon VPC Limitations

Erlang OTP Distributed application. Table 2 summarizes the steps required to deploy an Erlang OTP Distributed Application onto a target machine. It also highlights the *parameters* required to automate each of these steps.

Step	Parameter(s)
Install required Erlang version in the target machine	Erlang Version Number
Copy the source code to the target machine	URL for the source code
Build the application on the target machine	Build steps
Configure cluster	Number of machines composing the cluster
Start the application on the target machine	Command(s) required to start the application

Table 2: Deployment steps and parameters

6.1 Installation

We provide to the final user a set of Amazon Machine Images (AMIs) which are pre-provisioned with an Erlang stack. This reliefs the user from having to manually install Erlang on the machines[4]. We then allow installation and configuration of a custom Erlang OTP application dynamically, at instance launch time. This is done exploiting the Amazon's *Instance Metadata Service*.

Amazon provides an *Instance Metadata Service* that can be accessed locally on the instance once it is running to access instance-specific metadata as well as data supplied when launching the instance (also known as *user data*)[13]. We send all the information required to fetch the code of the user application (the *fetch url*) and to build it (the *build steps*) via the *UserData* field. Canonical's *cloud-init*[16] package

[4]In future releases, we plan to give users the ability to customize the Erlang runtime system, in terms of building options[5], emulator flags, standard flags and plain arguments[2].

is then used to handle early initialization of the cloud instances using the *UserData* arguments. The interactions between the user, the *UserData* field and the *cloud-init* package are reported in figure 5.

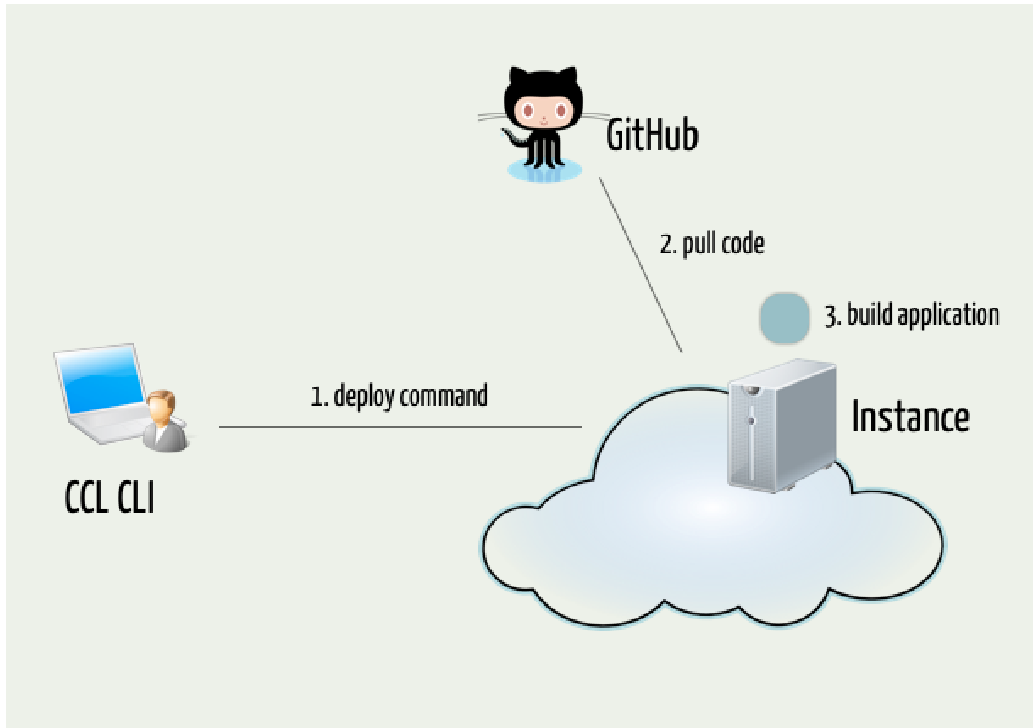

Figure 5: Automatic deployment of an Erlang OTP Application in CCL.

Instructions to build a specific Erlang/OTP application can be specified by the user via the CCL configuration file, in the form of a list of shell commands. We call these instructions *build steps*[5]. Using the same configuration file, the user also specifies the information (e.g. the GitHub credentials) required to access the repository hosting the source code for the application. All of these information are combined together by CCL and sent to the instance at launch time via the *UserData* field, in the form of an executable shell script. This file is automatically executed within the instance by the *cloud-init* script. cloud-init ensures that the script is executed only once, during the very first launch time. The source code for the application is automatically pulled from GitHub and built on the instance. The process is replicated for each of the machines composing the cluster.[6].

6.2 Execution of user-defined commands

CCL provides an *execute* command to run a shell command on each of the machines composing the cluster[7].

```
rel/ccl_cli/bin/ccl execute COMMAND
```

[5]Build steps are executed in parallel on each of the machines composing the cluster, whilst sequentiality is ensured within the context of the same machine.

[6]In future releases of CCL, we might introduce some form of caching for source tarballs (e.g. to avoid downloading multiple copies of the same package).

[7]Please note that, at this stage, we do not forward the output generated by user-executed commands to the client, so you might want to redirect the output (both stdout and stderr) to a file in the filesystem while executing commands on the hosts. This will simplify debugging, should things fail.

6.3 Cluster Configuration

After deploying an Erlang OTP application to a set of remote machines, you want to interconnect the machines - or actually the Erlang nodes running on those machines - together. In Erlang/OTP, this is typically achieved by connecting each of the nodes with the same node (e.g. the first one). The fact that Erlang connections between nodes are, by default, *transitive* does the rest. Say that you have a set of three nodes, A, B and C. You can simply connect B to A and C to A to obtain a fully-connected cluster of Erlang nodes. For example, to configure a *riak* cluster, you might want to execute the following command:

```
rel/ccl_cli/bin/ccl execute "rel/riak/bin/riak-admin cluster join riak@{{first}}"
```

The mustache-like[18] syntax allows you to parametrize commands. The {{*first*}} part will expand to the IP address of the first (in a sorted list) node of the cluster[8].

6.4 Automatic Failover and Takeover Strategies

In a distributed system with several Erlang nodes, resources might go down for a limited or extended period of time. OTP allows the developer to setup failover and takeover strategies for when this happens, so that applications can be "moved" from failing machines to working machines[3]. These strategies are typically achieved by manually configuring the *kernel* application for each of the nodes composing the cluster[8]. When dealing with cloud service providers, an extra level of difficulty is added, since some of the required information to setup the failover and takeover strategies, such as the IP addresses of the machines composing the cluster, may not be known at deployment time. Moreover, manual configuration of a huge numbers of machines is indeed a tedious and error-prone process. Failover and takeover strategies can be specified using the CCL configuration file:

```
{ccl_failovernodename, "NODE_NAME"}.
{ccl_failovernodes, BACKUP_NODES}.
```

NODE_NAME represents the name of the Erlang node (without the host) which will run on each of the machines composing the cluster, whilst *BACKUP_NODES* is the number of backup nodes we would like to have in our cluster.

Assume you want to deploy your application *my_app* to a cluster of three machines and you want one machine acting as the master node and two machines acting as backup nodes. You can achieve this with the following setup:

```
{ccl_clustersize, 3}.
{ccl_failovernodename, "my_app"}.
{ccl_failovernodes, 2}.
```

CCL will take care of configuring the *kernel* application for you on each of the machines the application was deployed to. The configuration for the master node might look something like:

```
[...]
{kernel,[{distributed,[{my_app,5000,
                         ['my_app@10.0.0.188',
                          {'my_app@10.0.0.189','my_app@10.0.0.187'}]}]}]},
         {sync_nodes_mandatory,['my_app@10.0.0.189','my_app@10.0.0.187']},
```

[8]During deployment, configuration files (e.g. vm.args or vars.config) are scanned for occurrences of *127.0.0.1*, which are replaced with the actual hosts values.

```
          {sync_nodes_optional,[]},
          {sync_nodes_timeout,30000}]}
[...]
```

Where *my_app@10.0.0.188* represents the *master node* and *my_app@10.0.0.189* and *my_app@10.0.0.187* are the two backup nodes.

6.5 Starting the application

You can use the *execute* command (see section 6.2) to generate an Erlang release and to start an Erlang application.

```
rel/ccl_cli/bin/ccl execute "./rebar generate"
rel/ccl_cli/bin/ccl execute "rel/my_app/bin/my_app start"
```

6.6 cloud-init Limitations

There are two important limitations about using *cloud-init* for the installation and configuration of an Erlang Application on a EC2 instance. In fact, the *cloud-init* executable script is passed to the instance via the Amazon EC2 *UserData* field, which has a maximum dimension of 16Kbytes. Also, the *UserData* are immutable by definition.

A possibility to overcome both limitations is not to pass the build steps and the other required configuration parameters directly via the *UserData* field, but to pass the location where the real data is kept, for example, using the Amazon S3 storage service. This way one could make changes to the configuration without having to change the location of the configuration. Please note that the Amazon S3 service allows creation of buckets which are accessible only from a specific VPC, which fits our scenario perfectly.

7 Future Work

7.1 Leveraging the current architecture

At the moment, the communication with the Erlang nodes running in the cluster happens through a tiny software layer which, acting as an abstraction over SSH, allows the user to distribute commands to the EC2 instances and to the respective Erlang nodes. From an EC2 instance perspective, this is translated into a couple of pre-installed shell scripts triggered by CCL. We plan to move from the current architecture to one where an *esl-core* application runs on each of the EC2 instances (probably on a secondary, hidden Erlang node). *Esl-core* would represent an *Operations and Maintenance* application, decoupled from the user-application running on the main Erlang node, but offering some API, in case the user would like to rely on some of its features at run-time. We devise to implement a *riak-core*[21] application as the first *esl-core* prototype.

7.2 Planned Features

We have a huge list of features which we would love to add to the CCL framework in the next few months:

- Simplified upgrades
- Automatic Dimensioning
- Automatic Scaling

- Enhanced Monitoring
- Tools and Patterns
- Advanced Clustering

Let's see them in detail.

Simplified upgrades. After you deploy an Erlang/OTP application to a cluster of Erlang nodes, you might want to upgrade them to a newer version. This could happen because you discovered a bug in your current implementation or simply because you wanted to add a new feature to it. Erlang/OTP already offers mechanisms, such as *code replacement* and *release handling* which simplify the task. CCL would leverage that, automating procedures and minimizing down time.

Automatic Dimensioning. What are the requirements for your Erlang application in terms of Erlang nodes, memory or storage? What kind of Amazon EC2 instances do you need to avoid your application to be overloaded? How can you avoid over-expending by renting resources you do not actually need? CCL should be able to help you saving money and time, by automatically selecting the best option to your needs.

Automatic Scaling. It is not always possible to estimate exactly the amount of resources required by an Erlang application. CCL Auto-Scaling, built on the top of Amazon EC2 AutoScaling, would allow you to scale the capacity of your Erlang application up or down, according to conditions you define.

Enhanced Monitoring. Monitoring a cluster of Erlang nodes is crucial to keep a system healthy and balanced. Overloaded and failing nodes should be easy to identify and traffic should be kept uniform across the cluster.

Tools and Patterns. Erlang Solutions has a great knowledge in terms of libraries, tools, patterns and architectures. Load balancers, logging systems, tunnelling and queing applications - to name a few - should be easy to setup and configure using CCL.

Advanced clustering. For a big number of Erlang nodes, the Erlang philosophy and the concept of *transitive connections* between Erlang nodes can be problematic, since a fully-connected graph of Erlang nodes could be too chatty, therefore reducing throughput and efficiency. To overcome this limitation, a technique is being currently developed as part of the Release project. According to this strategy, Erlang nodes would be grouped in sets of nodes, known as *s-groups* (or *Scalable Groups*). Groups would have transitive connection within the same group, but non-transitive connections with other groups, allowing the overhead due to inter-node communcation to be heavily reduced. We are evaluating the possibility to support such an experimental feature by making it available for CCL users.

7.3 Moving Towards a Heterogeneous World

As you might have already realized, the current implementation of CCL is strictly tightened to the Amazon EC2 Web Services offering. The major challenge in the next phase (deliverable D4.3) will be to move away from this assumption and to support clusters which are non necessarily Amazon-based. For this reason, we are looking at the *OpenStack*[15] software.

8 Conclusions

Our approach to automate deployment of an Erlang OTP application into a homogeneous cloud environment such as the Amazon EC2 one have been pretty straightforward. Moving from an homogeneous world to an heterogeneous reality represents a major challenge for us, given the intrinsic differences charcterizing the different cloud providers and the difficulties of implementing a common cloud broker which allows to use the providers' services in a transparent way.

We listed many features which we plan to add to the Scalable Virtualization Infrastructure, ranging from *simplified upgrades* to *automatic scaling*. Implementing these feature will leverage our Scalable Virtualization Infrastrucure, bringing it to the next level. This will allow companies who use the SVI to improve their time-to-market, quality, customer engagement and overall development flow.

A The CCL Configuration File

A sample *CCL Configuration File* follows, accompanied by a brief explaination for each of the listed parameters.

```
%% -*- Erlang -*-
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% EC2 Account Information %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
{ec2_hostname, "ec2.eu-west-1.amazonaws.com"}.
{ec2_accesskey, "YOUR_EC2_ACCESS_KEY"}.
{ec2_secretkey, "YOUR_EC2_SECRET_KEY"}.
{ec2_ami, "ami-123456"}.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Github Information      %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
{github_username, "prof3ta"}.
{github_repository, "m8ball"}.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% CCL Configuration       %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
{ccl_clustersize, 3}.
{ccl_failovernodename, "m8ball"}.
{ccl_failovernodes, 2}.
{ccl_buildsteps, ["./rebar compile",
                  "./rebar generate",
                  "rel/m8ball/bin/m8ball start"]}.
```

.

Option	Description
ec2_hostname	Endpoint for the Amazon EC2 Web Services
ec2_accesskey	Alphanumeric text string that uniquely identifies your EC2 account
ec2_secretkey	The *password* for your EC2 account
ec2_ami	A special type of pre-configured operating system and virtual application software which is used to create a virtual machine within the Amazon Elastic Compute Cloud (EC2)
github_username	The GitHub account owning the application that you want to deploy
github_repo	The name of the GitHub repository that contains the application that you want to deploy
ccl_clustersize	The number of machines composing the cluster
ccl_failovernodename	The name of your Erlang node (in case of automatic failover/takeover strategies). Nodes running on different machines will have the same *name*, but different *hosts*
ccl_filovernodes	The number of Erlang nodes acting as backup nodes (in case of automatic failover/takeover strategies)
ccl_buildsteps	List of shell commands which will be run during deployment

Table 3: The CCL Config File Parameters

B Moebius: Continuous Integration as a Service

As already explained in the introduction, we originally wanted to cover all the four stages of a system's life-cycle through our *Virtual Scalable Infrastructure*. Therefore, we started developing *Moebius*, a tool aimed at large-scale Continuous Integration, where the main idea was to serve Continuous Integration as a service. In this appendix we outline the architecture, the current feature set and the development status of Moebius.

B.1 The Moebius Architecture

Moebius has been designed as a distributed system composed of modular components. Each component provides mostly generic APIs allowing other components to communicate with it. Moreover the architecture is split into layers representing the different roles of components in the overall systems. The following diagram depicts the layered architecture.

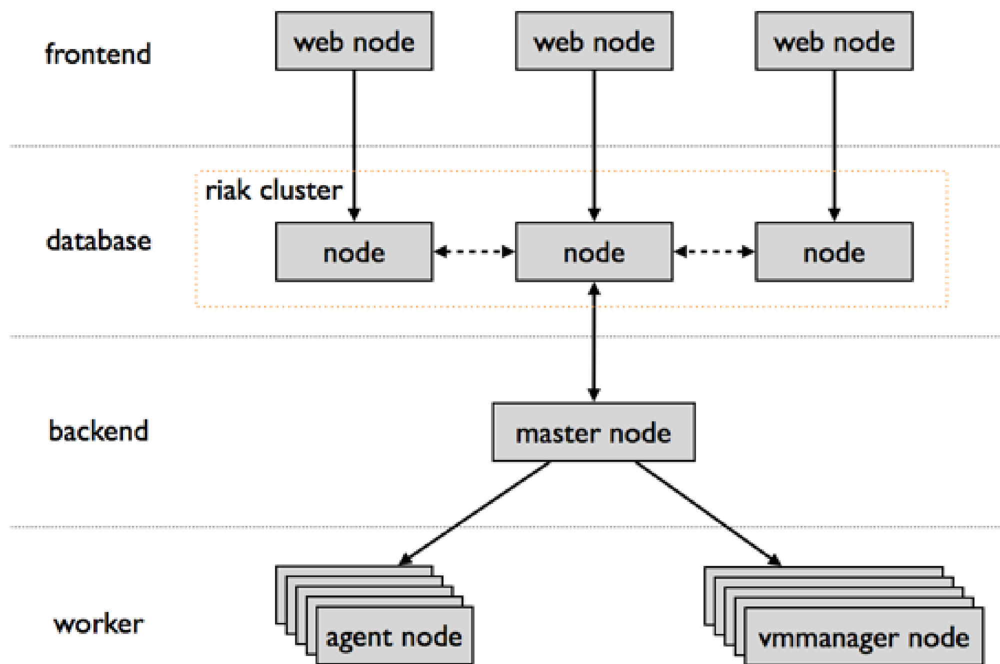

Figure 6: The Moebius Architecture

B.1.1 Frontend

The frontend layer is responsible for serving user requests through HTTP. Two content formats are supported.

1. HTML for browsing via a web browser
2. JSON for programmatic access to data

Irrespectively of the chosen format, the system uses HTTP Basic authentication to identify and authorize user requests. Each frontend node is a independent Erlang node, keeping a connection to a single node in the Riak cluster to read, query and write data. Therefore no data is shared

directly between frontend nodes. The following components are used by the Erlang nodes internally to implement the aforementioned functionality:

- *Mochiweb*: serves HTTP requests
- *Webmachine*: provides abstraction around the HTTP state model
- *ErlyDTL*: rendering of HTML template files

Moreover the HTML content delivered to a web browser uses client-side rendering to allow the dynamic update of content utilizing the JSON API without having to reload whole HTML pages. The following libraries are used to facilitate the client-side rendering:

- *Twitter Bootstrap*: used for HTML layout styling
- *JQuery*: used for general Javascript functionality
- *PureJS*: used for client-side rendering

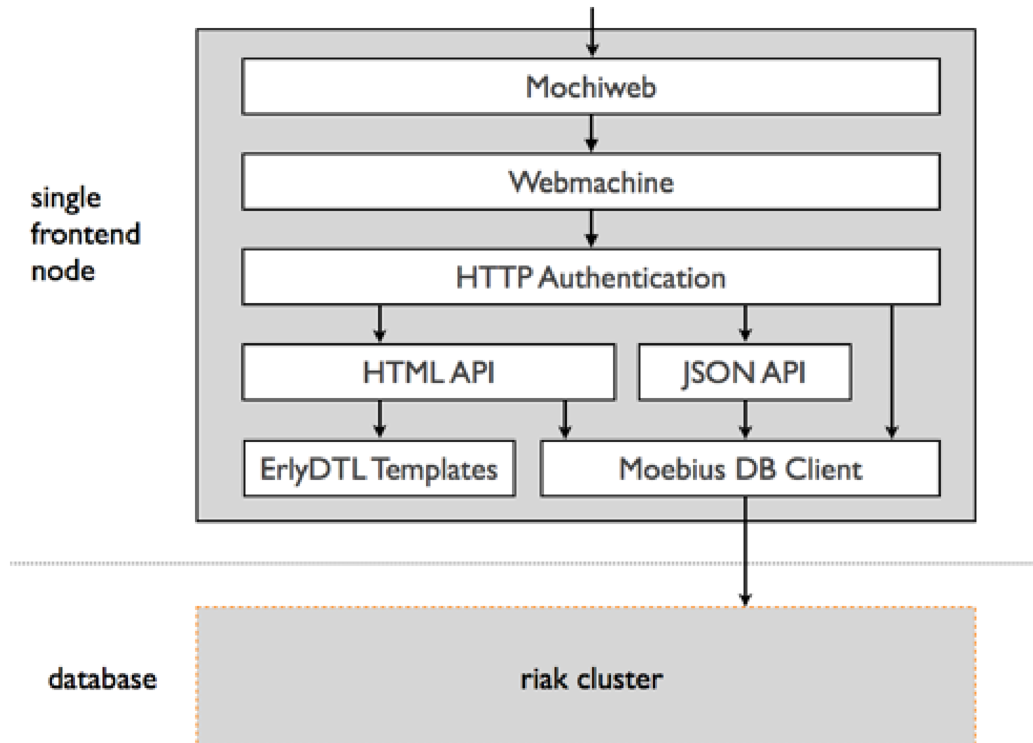

Figure 7: The Moebius Frontend

B.1.2 Database

All communication with Riak is encapsulated in a single component which exposes useful APIs such that one doesn't need to use the Riak client application directly. This component is used by both the web front-end as well as the master. The component, named *moebius_db_client*, consists of 3 layers:

Models. Each model is represented as an Erlang record and a respective module which can be used throughout the system to operate on the entity without directly accessing Riak. Most modules

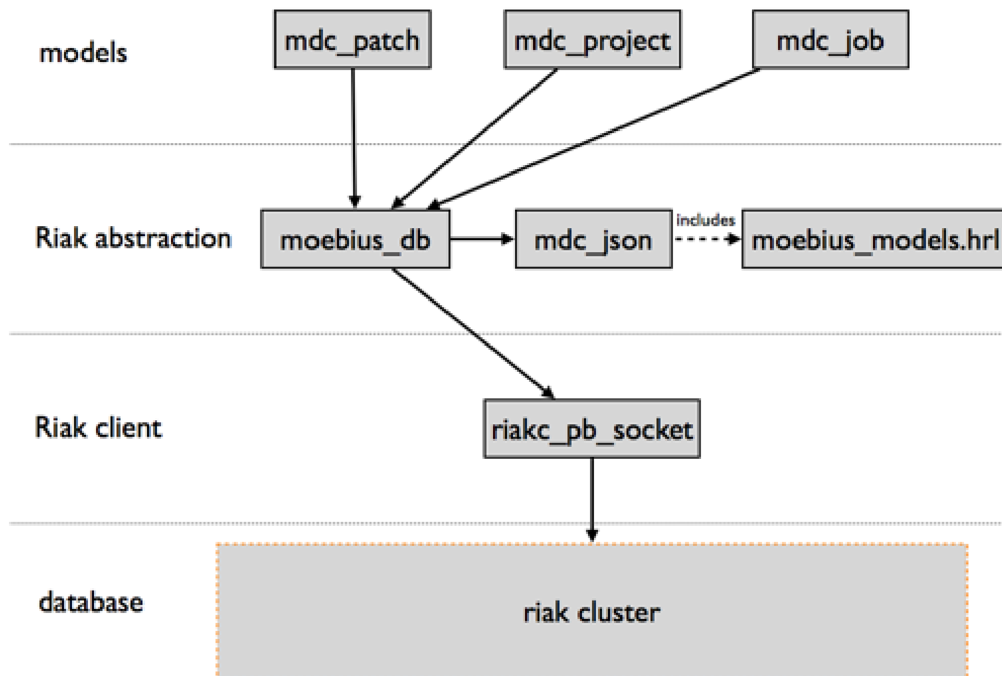

Figure 8: The Moebius Database

provide *CRUD*[9] functionalities together with some more specialized API functions which are specific to a certain entity. Almost all API functions have side-effects since they perform database operations internally. All models use the database abstraction *moebius_db* instead of the Riak client directly. All model records are defined in *moebius_models.hrl*.

DB Abstraction. The database abstraction *moebius_db* implements common operations done in Riak using default parameters where appropriate. Furthermore, it performs the marshalling and unmarshalling of entities using *mdc_json*. All data is stored in JSON format in Riak, thus *mdc_json* translates Erlang records to JSON and vice-versa, transparently to the developer. Finally, the abstraction layer provides a common, uniform *error handling* behaviour when accessing the database.

Riak client. The official Erlang Riak client application implements the communication protocol and messaging with Riak. It is used directly only by the database abstraction.

B.1.3 Backend (Master)

The backend, also called *master*, co-ordinates all the work in the system[10]. Its main tasks are:

- project monitoring
- job coordination
- job execution

For each project which is hosted on the system, a *project_worker* monitors its SCM repository for changes. By default, the worker periodically polls the repository for changes. All workers are supervised

[9] Create, Read, Update and Delete.

[10] In the current implementation of Moebius, the master is represented by a single Erlang node. To avoid having a single point of failure (SPOF) we were planning to convert it into a proper Erlang distributed application.

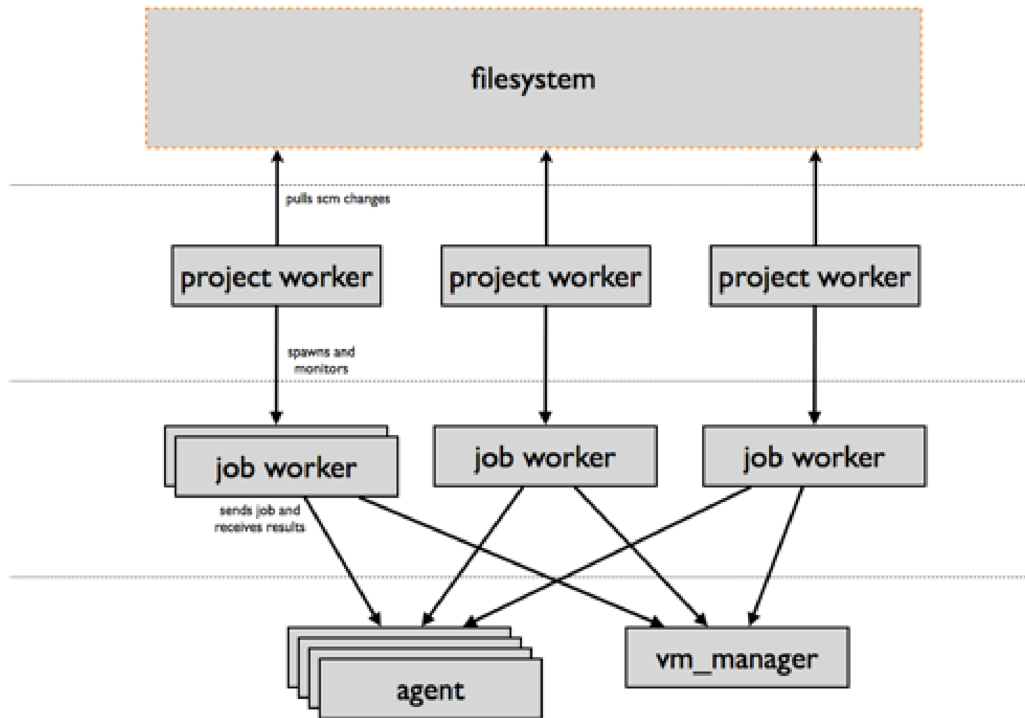

Figure 9: The Moebius Backend (Master)

by a *project_manager* which is merely responsible for cleanly restarting workers in the case of failures. Whenever a change is detected for a project, the respective *project_worker* creates a *job_worker* which is responsible for co-ordinating the activities for the job, acquiring resources as well as communicating with agents. Each *job worker* spawns several *job_runners*, one per platform. A *job_runner* acquires an assigned *platform* from *moebius_box* then starts sending the job data to the respective agent. The agent streams back all the results, such as logs and artifacts, which are stored in the database. Job runners terminate on completion. Likewise, once all the job runners spawned by a job worker terminate, the worker itself dies, thus all the processing in regard to the job is done.

B.1.4 Agent

The agent is a fairly trivial component which, deployed on a platform, exposes an HTTP API to the master. The API endpoint is used to receive new jobs. These jobs are transformed into a internal data format before they are sent to the job runner. The HTTP connection is kept open until the job runner returns the job results, which are then streamed back to the original caller. After the job has finished and all results are sent, the HTTP connection is closed.

B.2 Current Feature Set

User multi-tenancy

- allows many users to register and use the same instance of Moebius, independently from each other

Project multi-tenancy

- allows many projects to be hosted on a single Moebius instance

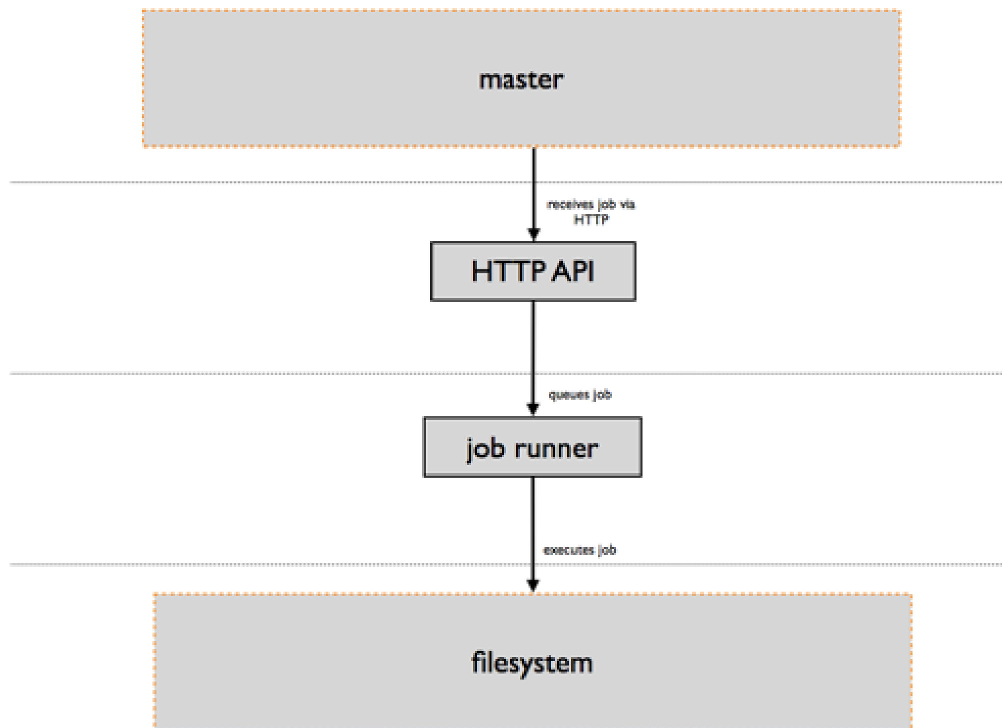

Figure 10: The Moebius Agent

- allows a user to manage many projects

Project Privacy

- a project can be public, thus e.g. test results can be seen by anybody, or private, only allowing admin users to access results

User Signup

- a user can create an account through the web front-end
- signups can be disabled to prevent users from registering (useful for closed testing)

Multiple Infrastructure Provider Support

- Amazon EC2
- VirtualBox
- dedicated servers

Multiple Platform Support

- a project can be assigned to many platforms, allowing builds or tests to run simultaneously on those

Web Front-End

- allows users to manage projects as well as view results

Integrated Documentation

- the web front-end includes a documentation system
- user documentation is work in progress

Json API

- provides the same functionality as the web front-end for managing projects and retrieving results
- uses the same access control mechanisms as the web front-end

Artifact Storage

- users can define regular expressions to select generated files to be stored in the database
- users can retrieve stored artifacts

VCS Support

- projects using any of the supported VCSs can be managed via Moebius
- Git
- Subversion

3rd Party Integration

- event-based internal integration API allow rapid integration with other services
- Gerrit (beta) - builds or tests can be triggered by new Gerrit reviews, results will be posted as comments to such reviews

Erlang-Independent Build/Test Commands

- can be any sequence of shell commands

Log Storage

- any output from builds/tests is stored and can be viewed by users later

Cloud Instance Instantiation

- creates new VM instances for each job and destroys these once a job has finished
- allows execution of many jobs in parallel

Project Dependencies

- projects can depends on other projects
- (successfully) finished builds/tests can trigger jobs for dependent projects (e.g. a successful build of Erlang/OTP can trigger a Riak build/test suite)

Puppet-based provisioning support for cloud instances

- allows the use of bare-bone VMs
- when a instance is launched it is provisioned using Puppet scripts provided by the user
- only available for VirtualBox as of now

B.3 Moebius vs Travis CI

While we were actively working on Moebius, a new little gem took shape in the open-source community: Travis CI, a hosted Continuous Integration service for open source projects. Like Moebius, Travis wanted to provide *Continuous Integration* as a *Service* to the final user, in a distributed fashion. Like Moebius, Travis wanted to be integrated with GitHub. Travis CI has been crowd funded and supported by some major sponsors. It soon became a *super-set* of Moebius, in terms of offered features. The development effort which has been put on Travis by the open-source community made him reach a stability and a level of maturity superior to the ones provided by Mobius. All of these reasons convinced us to stop the development of Moebius, before it was too late.

Neverthless, we did not throw away all the job done with the project, but we have re-used as much as possible from that experience, both in terms of acquired knowledge and actual code. Specifically, the following Moebius' *Building Blocks* have already been (or are going to be) migrated to the new framework:

- Interaction with GitHub APIs
- Webmachine RESTFul APIs
- Building workflow
- Artifacts browsing

References

[1] Ericsson AB. *Distributed Erlang.*
http://www.erlang.org/doc/reference_manual/distributed.html.

[2] Ericsson AB. *erl: The Erlang Emulator.*
http://www.erlang.org/doc/man/erl.html.

[3] Ericsson AB. *Erlang Distributed Applications.*
http://www.erlang.org/doc/design_principles/distributed_applications.html.

[4] Ericsson AB. *Erlang Distribution Protocol.*
http://www.erlang.org/doc/apps/erts/erl_dist_protocol.html.

[5] Ericsson AB. *Installing Erlang.*
https://github.com/erlang/otp/blob/maint/INSTALL.md.

[6] Logan at Al. Erlang and OTP in Action. Manning.

[7] Travis CI. *Travis CI: Distributed build platform for the open source community.*
http://travis-ci.org.

[8] Fred Hebert. *Distributed OTP Applications.*
http://learnyousomeerlang.com/distributed-otp-applications.

[9] Bertil Karlsson. *Secure Distributed Communication in SafeErlang.*
http://www.erlang.se/publications/xjobb/Secure_Dist_Comm_in_SafeErlang.pdf.

[10] Amazon Web Services LLC. *Amazon Virtual Private Cloud.*
http://aws.amazon.com/vpc/.

[11] Amazon Web Services LLC. *Amazon VPC Limits Form.*
http://aws.amazon.com/contact-us/vpc-request/.

[12] Amazon Web Services LLC. *Amazon Web Services.*
http://aws.amazon.com/.

[13] Amazon Web Services LLC. *Bootstrapping Applications With AWS Cloud Formation.*
https://s3.amazonaws.com/cloudformation-examples/BoostrappingApplicationsWithAWSCloudFormati
pdf.

[14] Amazon Web Services LLC. *VPC Security Groups.*
http://docs.amazonwebservices.com/AmazonVPC/latest/UserGuide/VPC_SecurityGroups.
html.

[15] Openstack LLC. *OpenStack: The Open Source Cloud Operating System.*
http://openstack.org/software/.

[16] Canonical Ltd. *Cloud Init.*
https://help.ubuntu.com/community/CloudInit.

[17] Gleb Peregud. *Erlcloud.*
https://github.com/gleber/erlcloud.

[18] Tom Preston-Werner. *Mustache for Erlang.*
https://github.com/mojombo/mustache.erl.

[19] Dave Smith. *Securing Distributed Erlang.*
http://dizzyd.com/sdist.pdf.

[20] Basho Technologies. *Rebar.*
https://github.com/basho/rebar.

[21] Basho Technologies. *Where to Start with Riak-core.*
http://basho.com/blog/technical/2011/04/12/Where-To-Start-With-Riak-Core/.