5
SCS Deployment Infrastructure in Use

Currently, an increasing adoption of cloud computing resources as the base to build IT infrastructures is enabling users to build flexible, scalable, and low-cost production environments. For example, consumption of CPU processing and manipulation of large data sets are easy tasks using computing and storage services, respectively. Applications like MapReduce fulfill these characteristics because it requires the processing of large storage files as input data. Also, its distributed architecture allows us to experiment with different deployment configurations. This chapter focuses on the use of SCS Deployment Infrastructure using cloud infrastructures as the target environment. We deployed a SCS MapReduce application on the cloud where the MasterNode and each WorkerNode were executed on virtual machine instances. We describe in section 5.1 the architecture and main concepts of MapReduce programming model and describe the architecture of a SCS component-based MapReduce application. Section 5.2 explains the necessary steps to deploy a SCS MapReduce application on cloud infrastructures. Finally, section 5.3 describes some useful considerations that should be taken into account at the moment when an application is deployed.

5.1 MapReduce Application

MapReduce is a programming model for distributed and parallel processing, and it generates large data sets [Dean04]. MapReduce allows users to hide technical details of distributed processing, parallelization, fault-tolerance, data distribution, and load balancing, making it a popular library for programmers without skills in distributed and parallel programming. MapReduce defines two main functions *map* and *reduce*, both are written by the users. The map function processes a key/value pair as an input and produces a set of key/value pairs or intermediate values. The reduce function receives a key and merges all intermediate values associated with this key, generating a new list of values. MapReduce implementations are designed to be executed over a large cluster of physical machines, and should support many terabytes of input

data.

$$map :: (key_1, value_1) \rightarrow list(key_2, value_2)$$

$$reduce : (key_2, list(values_2)) \rightarrow list(value_3)$$

Figure 5.1 shows an architectural overview of MapReduce. First, the MapReduce framework splits the input files into M pieces of 16MB to 64MB, and then many copies of the program are made to the cluster. A copy called *Master* assigns tasks *map* or *reduce* for other copies called *Workers*. There are M map tasks and R reduce tasks. All Workers execute the map tasks by reading their input split assigned and generating intermediate data distributed into R regions on the local disk. Later, workers reduce over the intermediate data generating a set of output files.

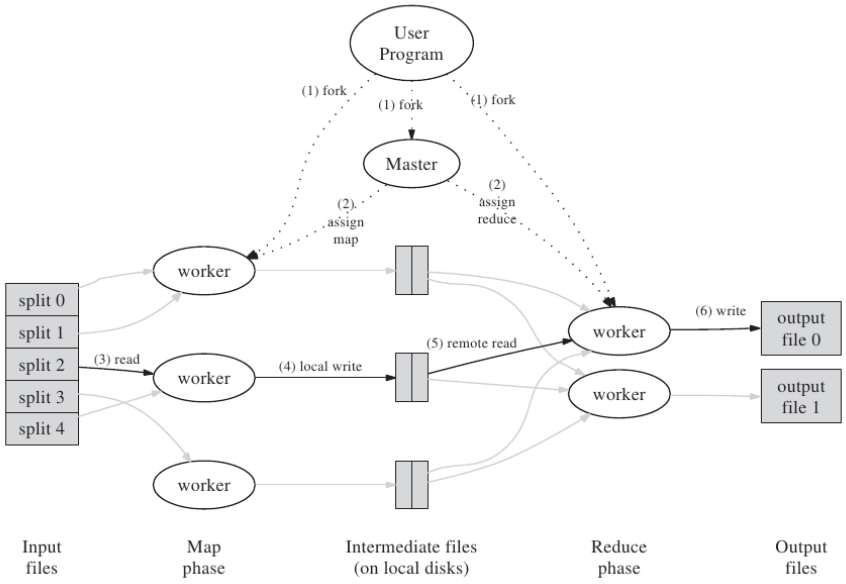

Figure 5.1: MapReduce Architecture

This work uses a *WordCount* application, this is an implementation of MapReduce architecture built using SCS components [Fonseca09]. This application allows us to count the number of words in an input set. Figure 5.2 displays its architecture, composed of *Master*, *Worker*, *Scheduler*, *Reporter*, and *Channel* components. Master controls all the application executions, it is connected with the Workers using a receptacle, and implements the facet *Master* used to submit jobs. Workers are connected to a channel, and their consumers are connected to the Master component. Workers use this channel to notify the final status of a task. Scheduler is responsible to select workers

from a set of machines, it uses a RoundRobin approach. Reporter component is used to register debugged messages.

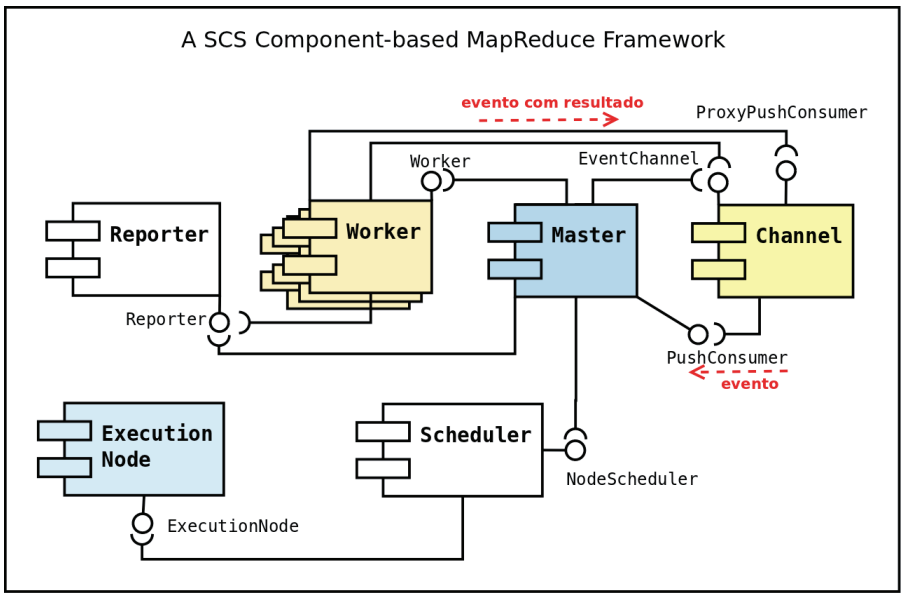

Figure 5.2: A SCS Component-based MapReduce Framework Architecture

5.2 Deploying a SCS-MapReduce on the Cloud

We explained the MapReduce application *WordCount*, therefore we have all elements to complete our main objective, that is, to enable the deployment of distributed component-based applications on cloud infrastructures. Figure 5.3 displays a flow diagram overview showing the components of the Deployment Infrastructure fulfilling the role of Platform as a Service(PaaS). The cloud API allows us to consume Infrastructure as a Service(IaaS), and the MapReduce application is intended to test our proposed architecture. The MapReduce application is represented by a *Master* node, *N Worker* nodes, and a *NFS shared storage*. To deploy the MapReduce application we use the *mapreduce-deployment* script. Also, there are some considerations to take into account before we execute an updated version of the *mapreduce-deployment* script. We need to check our architecture configuration, policies, and the cloud computing environment. An original version of the *mapreduce-deployment* script supporting a local-network as its target environment is displayed in appendix A.2.

Our first scenario for testing assumes the minimal configuration, as was explained in subsection 4.3.2. We use all policies studied in section 4.3, thus we use the cloud infrastructure policy displayed in 4.3, the platform policy specified in 4.4, the application policy 4.5, and the target environment policy defined in 4.6. The application policy is necessary to update the *deploy-*

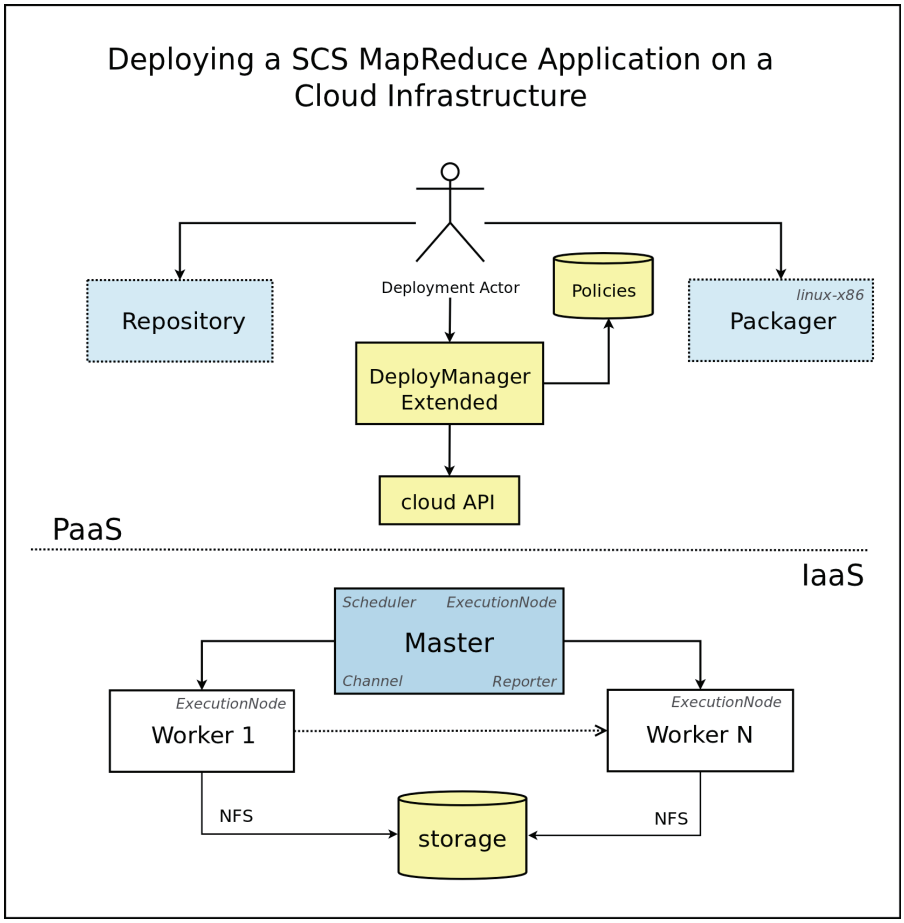

Figure 5.3: Deploying a SCS-based MapReduce Application on a Cloud Infrastructure

ment_script parameter with the *mapreduce_deployment.lua* script. To start deploying the SCS MapReduce application, we assume that the Deployment Infrastructure is running, including the CIS server. We use Amazon Web Services as the cloud infrastructure for our experiments, but it is nccessary to update the security parameters located in the application policy. We use an updated version of *mapreduce-deployment* script, which loads the policies and defines the *Master* and *Worker* nodes. List 5.1 shows a segment code that operates the options from *ec2_instances* policy, i.e., getting or running virtual machine instances. List 5.2 displays the creation of the workers and their containers. The complete code of our first test is showed in A.3.

Our second scenario assumes the Multi-Deployment Architectural configuration explained in subsection 4.3.2. Two or more *deployment actors* execute their own Deployment Infrastructure(PaaS) and Target Environment(IaaS). However they share the same Cloud Infrastructure Service requiring cloud resources. We use the same policies as our first scenario, except the *application*

Listing 5.1: Getting or Running Virtual Machine Instances Metadata

```
1  ...
2
3    -- Instantiating a cloud_engine
4    cloud_engine = CloudEngine{}
5    -- One Master and Two Workers
6    local num_instances = 3
7
8    if policy == 'getting' then
9      -- Getting VM Instances
10     vmInstances = cloud_engine.
11       instances[ cloud_engine:get_cloud_deployment_model() ].
12       cloud_connection:get_instances(num_instances,num_instances,policies)
13
14   elseif policy == 'running' then
15     -- Starting VM Instances
16     vmInstances = cloud_engine.
17       instances[ cloud_engine:get_cloud_deployment_model() ].
18       cloud_connection:run_instances(num_instances,num_instances,policies)
19   end
20
21 ...
```

policy, because the current configuration is intended to be utilized byt two *deployment actors*. They require deploying their applications on the same cloud infrastructure. Code 5.3 and 5.4 show the application policies for the *WordCount* application, deployed by *user a* and *user b*, respectively. To avoid security problems each *deployment actor* needs to check and update the security parameters from the *ec2_creds* policy, The cloud API manages the connections with the cloud infrastructure. Specifically, the EucaEngine allows the *deployment actors* to set different *access* and *secret* keys. Finally, both *deployment actor A* and *deployment actor B* are ready to execute the same *mapreduce-deployment* script as the first scenario.

5.3 Final Considerations

We have a set of metadata from a pool of virtual machine instances as a consequence of the adoption of cloud infrastructures as target environments. However, currently we have not fully this metadata. Therefore, we could take advantages of this metadata in several ways. For example, we could use this information to obtain customized target environments, to address non-functional application requirements, etc.

Our tests generate a set of log information, such as logs files from the cloud infrastructure, cloud API, deployment infrastructure, and applications. The log files of cloud API usually contain many useless data, especially when *deployment actors* are using the cloud resources. Thus, to make the debugging simple in advanced configurations of our architecture the support of other monitoring tools is necessary.

Listing 5.2: Creation of Containers and Workers

```
...

  -- Workers
  workers,containers = {},{}
  for i=2, num_instances do
    containers[i] = plan:create_container("java")
    containers[i]:set_instance( exNodes[i] )
    containers[i]:set_property( "classpath",
      SCS_HOME.."/libs/jacorb/\*:"..SCS_HOME..
      "/libs/\*:"..SCS_HOME.."/libs/luaj/\*" )
    workers[i] = plan:create_component()
    workers[i]:set_id( workerId )
    workers[i]:set_container( containers[i] )
    workers[i]:set_args( {SCS_HOME ..
                          "/scripts/execute/mapReduce.properties",
                          exNodes[i]:get_host().ip} )
    workers[i]:add_connection("Channel", channel, "EventChannel")
    workers[i]:add_connection("Reporter", reporter, "Reporter")
    master:add_connection("WorkerServant", workers[i], "WorkerServant")
  end

...
```

Listing 5.3: Application Policy to deploy WordCount by Deployment Actor A

```
-- MapReduce Application
mapreduce_app = {

  name = "WordCount", version = "1.0", author = user_data.user_a,
  deployment_script = "/home/usera/projects/scs-deploysystem/src/lua" ..
                      "/scs/demos/deployer/mapreduce-deployment.lua"
}

-- Deployment Actor A
deployment_actor = {

  -- User Data
  user_data = user_a,

  -- AWS Credentials for User A
  ec2_creds = { ... },

}

-- Target Environments
target_environment = {

  policy = 'cloud_infrastructure',
}

-- Applications to Deploy
applications_to_deploy = {

  -- Application 1
  application_1 = {

    application        = mapreduce_app,
    deployment_actor   = deployment_actor,
    target_environment = target_environment,
  },
}
```

Listing 5.4: Application Policy to deploy WordCount by Deployment Actor B

```
1 --- MapReduce Application
2 mapreduce_app = {
3
4   name = "WordCount", version = "1.0", author = user_data.user_a,
5   deployment_script = "/home/userb/projects/scs-deploysystem/src/lua" ..
6                       "/scs/demos/deployer/mapreduce-deployment.lua"
7 }
8
9 -- Deployment Actor B
10 deployment_actor = {
11
12   -- User Data
13   user_data = user_b,
14
15   --- AWS Credentials for User B
16   ec2_creds = { ... },
17
18 }
19
20 --- Target Environments
21 target_environment = {
22
23   policy = 'cloud_infrastructure',
24 }
25
26 --- Applications to Deploy
27 applications_to_deploy = {
28
29   --- Application 1
30   application_1 = {
31
32     application        = mapreduce_app,
33     deployment_actor   = deployment_actor,
34     target_environment = target_environment,
35   },
36 }
```