Doing LSD with Splunk in the Cloud

SIMON HABTEMIKAEI

KTH Computer Science and Communication

Master of Science Thesis
Stockholm, Sweden 2009
Doing LSD with Splunk in the Cloud

SIMON HABTEMIKAIL

Master’s Thesis in Computer Science (30 ECTS credits) at the School of Computer Science and Engineering, Royal Institute of Technology year 2009. Supervisor at CSC was Henrik Eriksson. Examiner was Stefan Arnborg.

TRITA-CSC-E 2009:146
ISRN-KTH/CSC/E--09/146--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.csc.kth.se
Abstract

As networks grow, so does the time it takes to install and deploy software. And the more complex the software is the harder it is to deploy. In this report we present Splunk, a software product designed for network surveillance with a lot of different configuration options. The paper describes a tool, Aurora, we designed to facilitate large-scale deployments of software, in this case Splunk. We also present cloud computing – the benefits of it and how it can be used as a hardware platform for our large-scale deployments.
Referat

Storskaliga installationer av Splunk i molnet

I takt med att nätverk växer, växer även tiden det tar att installera mjukvara och ju komplexare mjukvara desto svårare blir installationen. I den här rapporten presenterar vi Splunk, en mjukvaruprodukt som är designad främst för nätverksövervakning och ger häpnadsväckande många konfigurationsmöjligheter. Vi presenterar ett verktyg, Aurora, som vi tog fram för att göra det lätt att göra mjukvaruinstallationer oberoende av storlek. Vi introducerar även datormolnet, vilka fördelar som finns med det och hur molnet kan användas som en hårdvaruplattform för våra stor-skaliga mjukvaruinstallationer.
Contents

1 Introduction 1
  1.1 Splunk 1
  1.2 Amazon WS 1
  1.3 Problem 1
  1.4 Acknowledgement 2
  1.5 How we divided the work 2

2 Splunk 3
  2.1 Splunk - What is it? 3
  2.2 Forwarding 4
  2.3 Indexing 4
  2.4 Round Robin 4
  2.5 Distributed search 5
  2.6 Deployment server 5

3 The Cloud 7
  3.1 Amazon Elastic Compute Cloud 7
  3.2 Amazon Images - AMI 7
  3.3 Amazon Simple Storage Service 8
  3.4 Authentication with AWS 9

4 Deployments 11
  4.1 Deployment vs. Large Scale Deployments 11
  4.2 Aurora 12
    4.2.1 Aurora Architecture 12
    4.2.2 Graphical User Interface 12
    4.2.3 Management Service 12
    4.2.4 Provisioning Service 13
    4.2.5 Deployment Service 13
    4.2.6 Persistence 13
    4.2.7 Clusters 13
    4.2.8 Recipe 14
    4.2.9 Template 16
Chapter 1

Introduction

1.1 Splunk

Not long ago data was stored in huge data centers with no easy way of browsing it but at least it was in one place. But in the 21st century with large networks, the need for SOX/PCI compliance, virtualization and the need to monitor networks for security threats the users are suddenly faced with data being more decentralized with an even greater need to browse it. This is where Splunk enters the picture. Splunk gathers data from the network, virtual machines and other types of log files it indexes the files and makes the data searchable through an web-interface.

Due to the different ways networks are set up Splunk has to be flexible in this respect. An Splunk instance can be configured to be either a forwarder (forwards data to another Splunk instance), an indexer (receives data and indexes it) or both at the same time. If you have a lot of data you can set up Splunk to balance the load between indexers. This way it does not choke on your logs.

1.2 Amazon WS

In the middle of July in 2002 Amazon launched it’s Amazon Web Services. AWS is a collection of web services you access through remote commands. AWS offers you the possibility to store your files online (S3), to allocate computing resources (EC2) and more.

1.3 Problem

To ensure high quality, Splunk Inc has to be sure that their product works well on different systems and under different network topologies. This means setting up a lot of different systems and maintaining them. Installing and configuring many different systems and making sure the systems run as they should is a very time-consuming matter. So they were looking for some way to automate this process.
So we were asked to investigate whether they could replace their own hardware dependency by using the cloud instead. To test if the cloud could be used as a viable testing ground for their product, especially for large-scale deployments.

We were also asked to investigate if it was possible to design a tool to design arbitrary complex Splunk deployments and utilize the cloud as a deployment ground.

1.4 Acknowledgement

This investigation part and the design of the application was a collaboration between me and Daniel Norberg. For a complete picture of this project you should take the time to read his thesis[9] as well.

I would like to thank my thesis partner Daniel, without him I do not think this project would have been as fun. I would also like to thank my chief Boris Chen at Splunk for the interesting project and for the opportunity to go to San Francisco, my advisor Henrik for all his encouragement and wise words and my examiner Stefan for his constant support.

1.5 How we divided the work

The design was a collaboration between me and Daniel but when it came to the actual system development I focused on the parts that related to the cloud (provisioning of the machines, os and such) and the deployment of the actual software. Daniel built the user interface and the recipe-generator.
Chapter 2

Splunk

2.1 Splunk - What is it?

A lot has happened in the IT world the last fifteen years. IT has gone from being a necessary evil to one of the core instruments in most corporations. The web has exploded and grown far more than anybody could have imagined, people are starting to use the web for more than just surfing around and email, now people run software on the web (SaaS, software as a service), they listen to music (MaaS, music as a service) and a lot more. But as the need for IT grows so does the need for surveillance. This is what Splunk gives you. Splunk is an application designed to help the user get an overview over what has happened with his systems. Splunk takes log data in different forms, indexes it and makes it searchable. It is designed to run on a network of computers so it has many different configuration parameters; by changing the parameters\(^1\) the user can get his splunk instance to assume the role he feels is more fitting.

\(^1\)For a list of all the configuration options available the reader should read the documentation[10])
2.2 Forwarding

When a Splunk instance is configured to forward data, it collects the data available on the server and it transfers the data to an indexing instance of Splunk. You can set it up either to index locally or if you just want it to forward all the data to the indexer without indexing it locally. The user also has an option to forward different types of data to different forwarders. An example of this is if the user for example runs a J2EE-server and wants to forward that type of data to one indexer and his syslog data to another indexer. For an example see figure 2.1.

![Figure 2.1. Indexer](image)

2.3 Indexing

An indexer receives data either from forwarders or from the TCP/UDP ports. The instance then parses the data, extracts relevant information and puts it into an index.

2.4 Round Robin

To prevent the indexers from clogging when you have a high load there is a load balancing technique built into Splunk. The forwarders can distribute the data in a round-robin fashion to different indexers, either depending on data or just round-robin everything. What round robin distribution entails is that the instance transfers one event each to every indexer, then starts over again. This way it is less likely for the indexers to clog if a couple of forwarders have too much data to process. For an example see figure 2.2.
2.5 Distributed search

If the user has set up a Splunk topology with a couple of forwarders, a couple of indexers perhaps using the round robin discussed in the previous section and he now wants to search the data, how will he do it? One way is to access every individual indexing instance using their UI and search for what is needed. This is hardly practical, even with as few as two instances this will not be something to look forward to. That is why Splunk has an option called distributed search. The user just enters the hostnames for the running indexing servers into the instance he wants to use. Now when the user runs a search, that search request will be executed on every instance we listed and the result will be propagated back to the instance that executed the search request. It will combine the result into an ordered list. For an example see figure 2.3.

2.6 Deployment server

Setting up a complex Splunk topology can be a tedious task. To facilitate the setup of the Splunk instances the user is provided with a deployment server. The user creates configuration bundles and tells the server which instance should get what bundle (multiple instances can use the same bundle). All the instances connect to the deployment server and initiate a hand-shake. Afterwards the deployment server pushes out the configuration bundles to the clients. So the user just has to create the bundles and configure the deployment server, and is saved from the task of logging in on every instance and configuring it from scratch. For an example see figure 2.4.
CHAPTER 2. SPLUNK

Figure 2.3. Distributed Search

Figure 2.4. Deployment Server
Chapter 3

The Cloud

The standard model of buying hardware and software for a data center has changed in recent years. In the old model, hardware and software was seen as any other resource that you bought at a fixed cost, but this has slowly started to change with high-speed internet connections being more easily available. Software-as-a-service, where software is subscribed to instead of bought, has started to emerge as a replacement for the old model. Cloud computing\(^1\) is the next step in the as-a-service direction, where you not only rent the software but the computing resource\(^2\) as well. Some people refer to this type of service as Everything-as-a-Service (EaaS).

3.1 Amazon Elastic Compute Cloud

EC2\(^1\) (Elastic Compute Cloud) is Amazon’s way of offering EaaS. Their version of the cloud uses virtualization to offer computing resources on-demand. They offer a couple of different machine-types depending on your needs. When the user virtualizes a system, he needs an image containing operating system and other optional software. The only thing he has to do to make an allocation request with EC2 is to specify what machine instance-types he wants and what image he wants them to run and within minutes he will have the computing resources he requested. This makes it easy to scale up and down his computing power depending on the stress of the system (this technique is discussed in [5]). A simple illustration can be seen in figure 3.1.

3.2 Amazon Images - AMI

When the user makes an allocation request, he can specify what machine-type he wants but more importantly he can specify the image to be loaded and run on

\(^1\)For some interesting thoughts about the implications of cloud computing the reader should read [8]

\(^2\)The reader should read [7] for some interesting thoughts on cloud-computing
the virtual machine. The user has the choice of either using one of Amazon’s pre-configured images or he has the option of creating his own. This makes it possible for the user to specify exactly what software he wants his machine to run at boot-up, but what is even more important is that this makes it really easy to scale up and scale down his deployment depending on server-load.

3.3 Amazon Simple Storage Service

Amazon S3[2] is a simple storage service. It is designed to be easy to use. The user should never have to think about buying and setting up a reliable storage solution. The idea is really simple, the user creates a bucket (their version of a folder) to store his files, he uploads it to the bucket and sets the correct permissions on the file. Now the file can be accessed through a simple URL.

Amazon S3 is well suited when the user wants his EC2 instances to have high-speed access to a file or if he wants to use a custom image.
3.4 Authentication with AWS

When the user does allocation requests he must provide Amazon with the right credentials. Every user is allotted an Access Key and a Secret Access Key. The Access Key is to identify the unique account and the Secret Access Key is used for the authentication.

The system is designed like a typical signature-schemes. When the user makes a request he generates a signature with his Secret Access Key, Amazon looks up the user’s Secret Access Key and generates a signature from the key, compares the two signatures and if they match the request is processed.
A software deployment includes all the steps that are needed to make software available for end-users. In this chapter we will discuss large-scale deployments. We will also present Aurora, a system designed to make it easy to do software deployments independent of the deployment size.

4.1 Deployment vs. Large Scale Deployments

So what is the difference between a regular and a large scale deployment? Is there any? If the user just looks at the essential problem the answer is no, but there is a difference when the user actually has to do the deployment. Then there is a number of differences like the number of machines and the time it takes to setup (both installation and configuration) the software on the machines.

Usually, when the user does a deployment, he spends the majority of his time configuring the system. Installing most systems are trivial but configuring them to fit the needs of the user can take a lot of time. If the user wants a software deployment of 5 machines where every installation takes 30 minutes, then he will spend 2.5 hours installing and configuring the machines. What happens when the load increases and he needs more machines? If the user wants a software deployment on 100 machines? Then he will have to spend 50 man-hours installing and configuring the software, plus the added complexity of configuring that big a system. Fortunately some software comes with a built-in deployment server, so some of the configuration time can be lowered when this is the case. But the majority of the software out there does not come with a deployment server so the user can not rely on this functionality to always be there. To make the transition to large deployments the user needs some kind of software framework to configure and then deploy software on an arbitrary number of machines.
4.2 Aurora

Aurora is the framework that we created for doing exactly this. With the help of Aurora the user can create arbitrary complex deployment topologies. The user tells Aurora how many machines he wants and what type of operating systems he wants the machines to run and what type of software he wants on the machines. The software-configuration part of Aurora is designed around a plug-in model. If the user of the system wants to deploy some software not currently supported by Aurora, he just creates his own plugin consisting of some configuration templates for the software. The UI for that plugin will then be automatically generated from that template. This way Aurora can support any existing software.

4.2.1 Aurora Architecture

Aurora is designed as a collection of services: Management Service, Provisioning and Deployment Service (see figure 4.1). All these services are controlled from a Graphical User Interface. Each of these services has its own responsibilities and runs independent of the others. They communicate with each other using asynchronous message passing; this is to prevent unnecessary hold-ups (the provisioning in particular may take some time).

4.2.2 Graphical User Interface

The UI is web-based interface. It is built in the Django and uses ExtJs for presentation. The user interface presents the user with the option of designing his own deployment, what the clusters should look like, how many machines every cluster should have, how the software, in this case Splunk, should be configured and what other software packages should be included in the deployment. If the user wants, he can set an expiration time on the deployment. The expiration time is a safety mechanism in the EC2 case. Without it the user’s bills may rack up if he forgets to turn the machines off when he is done with them. When the user is done with designing his deployment he saves it as a deployment-recipe. This way the deployment can be reused an infinite number of times. All the communication between the user Interface and the Management service is done through a REST-interface.

4.2.3 Management Service

It is the Management service that gets the allocation requests from the user interface, it passes it along to the Provisioning service. When the Provisioning service has provisioned the machines it returns a list of the hosts to the Management service. Management then generates a recipe with the deployment information and sends it to the Deployment service.

When the user is done with his deployment or if the running time expires, Management sends a termination request to Provisioning.
4.2. AURORA

4.2.4 Provisioning Service

The Provisioning service either waits on new allocation requests or it updates the status of running requests. This means that it is the Provisioning service that returns a list of viable hosts to the Management service after an allocation request. When the Provisioning service gets a termination request it is also responsible for terminating the machines and alerting the Management service that the machine is now terminated.

4.2.5 Deployment Service

The Deployment service gets the deployment requests from the Management service. The request is in the form of a recipe that is parsed and executed on. If any of the files in the recipe are marked as assisted (see 4.2.13) the Deployment service downloads the files itself and then uploads it to the correct host. This is to make it possible to deploy files that are only available on the intranet. It also uses a file distributor to decrease deployment time if necessary, but more about this later.

4.2.6 Persistence

What happens if the Provisioning service crashes just before an allocation request? Or in the middle of a request? This can be costly if you rent the computing resource (as is the case of EC2). Or what happens if the Management service crashes? Will all the information about current deployments disappear? Neither of these are acceptable scenarios for the user. To provide persistence Aurora uses a database to store the information about the deployments and the allocation requests. Even if one of the services crashes in the middle of an allocation request, it will start again once the service is restarted.

4.2.7 Clusters

Normally when a user has to deploy software on multiple machines, he has an repetitious assignment in front of him. There are of course shortcuts for this, an example is to create an image of the system and then installing the image on the computers you want to run the new software. However, this does not leave the rest of the system intact, instead the operating system and all the previous software will be replaced. Aurora takes a different approach to this problem.

When the user designs his deployment in Aurora the user defines clusters of machines where operating system and software configurations are the same, he then specifies how big he wants his cluster to be. The clusters are a recursive structure so the user can define clusters in clusters (if A is a cluster and B is a subcluster of A then that would mean that everything A has, B has as well, in addition to that B can have an extra software package). With this cluster-model the user creates configurations for each cluster of machines instead of having to do it for every
individual machine. This way the user can configure a large number of machines very quickly.

When the user decides what software packages he wants to run on each cluster he can choose if he wants to deploy a local file (local for the Aurora instance) or a file that is available remotely. If the file is available remotely Aurora opens up an SSH-connection to the computer we want to deploy the package on and uses Wget/cURL to download the package.

The observant reader will see a problem with the remote-deployment method. What happens when the package we want to deploy is available on the intranet but not available for the computer we want to deploy the package on? Aurora has a solution to this problem and it will be discussed later.

4.2.8 Recipe

When the user is satisfied with the deployment topology he saves it and then tells Aurora to deploy it. Aurora will then generate what we refer to as a recipe.

The recipe that is generated contains the whole deployment and is saved in
4.2. AURORA

YAML (rhymes with camel), a data serialization format. There is an example of what a recipe looks like in Appendix B.

Here is a brief description of the keys in the recipe:

**Filedistributor** Aurora uses Amazon S3 as a filedistributor, but it can be replaced with any filedistributor as long as it uses the same interface as the s3distributor.

**Hosts** A list of hosts, every host contains data about the deployment

**address** The hostname/IP for the host.

**deploy** Contains a list of files that we want to deploy with a set of parameters for every file.

**action** This key tells the deployer how it should treat the file, if it needs to be extracted or not.

**destination** Where on the host should the deployer upload the files.

**method** Tells the deployer how the file should be uploaded (sometimes the file has to be downloaded to the deployer and then uploaded from there).

**source** From where should the file be uploaded.

**id** Id to set the hosts apart.

**name** Name of the host.

**parameters** Contains information about cluster-id, amazon-image and what machine-type the EC2 instance is running.

**post-exec** A list of commands that should be run after the files has been deployed

**pre-exec** A list of commands to be run before the files are deployed

**provider** Contains information about hardware architecture, instance-id, where the amazon-image is located and what reservation this instance belongs to.

**ssh-key-name** Filename of the RSA-key the deployer should use when opening up a connection to the host.

**ssh-key-path** Path to the RSA-key the deployer should use.

**state** Contains information about the allocation state of this instance.

**user** The username the deployer should use when it is connection to the host.

Along with the recipe, configuration files for the software (Splunk in this case) is generated to go along with the deployment. To generate these configuration files Aurora uses the Django templating system. There is an example of what a template looks like and the result afterwards in Appendix C and D.
4.2.9 Template

To generate configuration files for the software packages Aurora uses the built-in templating system in Django. An example of a configuration template can be seen in Appendix C. The brackets enclose the part of the configuration files that should be generated. In Appendix C we have the configuration file inputs.conf, a file that describes the input-rules for a Splunk instance.

Appendix D describes what the configuration file looks like after it has passed through the templating system. In Appendix D we can see that all the text enclosed by brackets has been replaced with real values or in some cases entire parts of the configuration file has been removed.

By comparing Appendix C and D it is possible to see that the configuration file generated will monitor /var/log for files and listen for syslog data on port 9998 using the TCP-protocol and receive data from another Splunk instance on port 9997 using TCP.

4.2.10 Boto

Boto[3] is a lightweight open source Python package which provides interfaces for a number of Amazon’s Web Services. Aurora uses Boto for its AWS communication.

Boto is designed to be easy to use. Here is an example of an resource allocation:

```python
> import boto
> ec2_conn = boto.connect_ec2(AWS-ACCESS-KEY, AWS-SECRET-KEY)
> images = ec2_conn.get_all_images(image_ids=['ami-b111f4d8'])
> images[0]
Image:ami-b111f4d8
> reservation = images[0].run(1,1,'aurora-keypair')
```

This will result in an allocation request with one instance of the smallest type (if the user does not specify machine-type he will automatically get the smallest) running the image ‘ami-b111f4d8’ and using the key pair called ‘aurora-keypair’. The arguments written in capital letters are variables containing your access-key and your secret-access-key.

4.2.11 Authentication

Usually when the user uses RSA authentication over SSH, he generates a public/private key pair and uploads the public part of the key to the remote host, then when he tries to connect he authenticates using the normal RSA scheme. The crucial part here is that the remote host has to have access to the public key when the user tries to connect to it. This means that if the user wants to connect to an EC2 instance he needs to provide it with the public key of our key pair. Fortunately Amazon provides us with a way to do this. Using interfaces provided by them the user can create and name a key pair, Amazon will then retain a copy of the public

16
part of the key pair. When the user does an allocation request, he just has to specify
the name of the key pair and they will provide the hosts with the public part of the
key, the reader can see an example of this here.

Aurora doesn't provide the user with a way to create his own key pair, the user
has to provide his own EC2 registered key pair.

Aurora doesn't provide the user with a way to create his own key pair, the user
has to provide his own EC2 registered key pair.
4.2.12 How the deployment works

When the deployment service receives the recipe it goes through the list and connects to the hosts in the list. As you can see in the example provided in the appendix the recipe is divided into different hosts where every host contains all the relevant information for the deployment for that specific host.

Aurora uses SSH to connect to the hosts and SCP to upload the payload to the hosts (This is true as long it is not a remote file. If that is the case Aurora opens a SSH connection to the host and uses Wget/cURL to download the file). Once we are connected to the remote host, we uncompress the required files in the payload and run the commands specified in the deployment recipe. Figure 4.2 shows the flow of a deployment.

![Figure 4.2. The flow of a deployment](image)

4.2.13 Assisted deployments

In some cases the user is interested in deploying files that are available on the intranet but not accessible from an outside source. When this is the case the URL in the recipe will not be accessible to the host. To solve this problem we designed
4.2. AURORA

a feature called assisted downloads into Aurora. What this means is that Aurora will download the file to the deployer and after that upload the file, either directly or through a file distributor, to the host.

4.2.14 Going Large Scale without Increasing deployment time

One thing to consider when we scale-up the deployments it that the deployment time will increase a lot. The most time-consuming matter is the distribution of the payloads. There are a couple of tricks you can use to decrease the deployment time. The problem is that the deployer gets stuck with doing all the uploads, so the big question is; how do we relieve the deployer? One way is to make every cluster responsible for its own files (see figure 4.3). This is done by selecting one host in every cluster and uploading the payload to that host then either let that host transfer the payload to the other hosts in the cluster or deploy to some hosts and let them do some deployment themselves. The problem with using this method is that the hosts we want to connect to do not have access to the private-part of the key pair. So if we want to connect to another host from one of our hosts we will have to upload the private part of our key pair to the first host. This is not a recommended approach, the user should never transmit his private key over the net.

When we designed Aurora, we had to come up with a way to decrease the deployment time but avoid sending our private key around. For this we introduced the concept of filedistributors, or more specifically in our case, Amazon S3. When the Deployment Service gets the deployment recipe it goes through it and looks for files that are repeated multiple times. When it knows what files are going to be deployed to multiple hosts it creates a bucket for the files on S3, it then uploads the files to that bucket and then rewrites the recipe to match the new deployment (see figure 4.4). When the deployment is done the Deployment Service empties the bucket.

```python
> import boto
> from boto.s3.key import Key
> conn = boto.connect_s3('aws-key','aws-secret-key')
> bucket = conn.create_bucket("aurora-bucket")
> key = Key(bucket)
> key.key = "splunk.tgz"
> key.set_contents_from_filename("splunk.tgz")
```
Figure 4.3. Distribution of Files
4.2. AURORA

Figure 4.4. Distribution of Files using S3
Chapter 5

Conclusion

5.1 Summary

In this paper we have compared regular deployments with large-scale deployments and talked about why large-scale deployments are hard to do.

We began with presenting Splunk, a data analyzing tool with an immense number of configuration options. We presented it as a software product where large-scale deployments are complex and time-consuming.

The concept of cloud computing was introduced and Amazon EC2 was used as a real case. We discussed the cloud and what it can be used for, how the computing-by-demand could be used as an testing environment for our large-scale deployments. Together with EC2 another Amazon service was presented, Amazon S3. Amazon S3 is a storage service we used as way to distribute files quickly. By uploading files to Amazon S3 we could replace the users own bandwith (except for when you had to upload files to S3) with Amazons almost endless bandwidth. By using Amazon S3 our deployment time went down considerably when doing large deployments.

We presented Aurora, the software framework we designed to make it easy to do deployments, regardless of size. Aurora lets the user design his deployment by dividing the topology up in clusters, where he specifies the hardware architecture, operating system, software to run and the number of machines in that cluster. He also gets to configure the software. When he is satisfied with his deployment he presses deploy and a Aurora provisions the machines from a predefined provider, EC2 in this case. When the provisioning is done Aurora generates configurations for the software and a deployment recipe that is used by the deployer. The deployer reads the recipe and deploys the specified files to the hosts listed in the recipe. Now the user has a fully-functional deployment. When we tested Aurora, both me and Daniel were surprised by the result. The system worked better than we imagined. A deployment of 20 machines took minutes, where the majority of the time would be spent waiting on EC2 to provide the machine, and not hours.

\footnote{There are many different architectures techniques that we have not mentioned but are discussed in [11]}
Doing large-scale deployments is only as hard as the tools the user chooses to use. If he tries doing them without any help he is in for some time-consuming tasks including a lot of debugging, but with the right tools he has the opportunity to do large-scale deployments fast.

5.2 Improvements

A possible improvement for Aurora would be to make it easy to gather data files from the hosts that the user has deployed on. This would make it easier to run some real tests on the machines. Another possible improvement would be to make it easier to combine different types of providers, so that the user has the option to deploy on a couple of local machines and some machines in the cloud at the same time.

Another possible add-on would be to add quotas to different groups with deployment scheduling depending on how many machines are running and how many are requested at a given time.
Appendix A

Dictionary

**Django**[4] is a open-source framework for web development. It is designed to make it easy to create database-driven websites.

**ExtJs** is a Javascript library for building interactive web-sites.

**REST** is a software architecture model where resources are made available through a global identifier (like a URI in HTTP). REST provides the user with a way to transfer data over HTTP without having to use a technique like SOAP. REST was proposed by Roy Fielding [6]

**YAML** is a human-readable data serialization format. A more human-friendly version of XML.
Appendix B

Recipe

file_distributor:
  module: services.ec2provider.s3distributor
  parameters: {account: 1}

hosts:
  - address: ec2-75-101-196-12.compute-1.amazonaws.com
    deploy:
    - {action: extract, destination: ~/splunk/, method: url-assisted,
      source: 'http://releases.splunk.com/released_builds/3.4/
      linux/splunk-3.4-Linux-i686.tgz'}
    - {destination: ~/splunk/, source: /var/1_NewCluster/
      0_ec2-174-129-147-44.compute-1.amazonaws.com/0_splunk/splunk}
    id: 0
    name: ec2-75-101-196-12.compute-1.amazonaws.com
    parameters: {cluster: 1, 'image-id': ami-205fba49, type-id: m1.small}
    pass: null
    post-exec: ['~/splunk/splunk/bin/splunk start --accept-license']
    pre-exec: []
    provider: {arch: i386, instance_id: i-40e05829,
      manifest: ec2-public-images/fedora-core4-i386-base-v1.07.manifest.xml,
      reservation_id: r-6454f80d}
    ssh-key-name: id_rsa_aurora
    ssh-key-path: /Users/simonh/id_rsa_aurora
    state: Allocated
    user: root
  - address: ec2-174-129-147-63.compute-1.amazonaws.com
    deploy:
    - {action: extract, destination: ~/splunk/, method: url-assisted,
      source: 'http://releases.splunk.com/released_builds/3.4/
      linux/splunk-3.4-44873-Linux-i686.tgz'}
    - {destination: ~/splunk/, source: /var/aurora_20081218.230828SvJUEK/
APPENDIX B. RECIPE

1_NewCluster/1_ec2-174-129-143-80.compute-1.amazonaws.com/0_splunk/splunk
id: 1
name: ec2-174-129-147-63.compute-1.amazonaws.com
parameters: {cluster: 1, image-id: ami-205fba49, type-id: m1.small}
pass: null
post-exec: [-/splunk/splunk/bin/splunk start --accept-license]
pre-exec: []
provider: {arch: i386, instance_id: i-41e05828,
manifest: ec2-public-images/fedora-core4-i386-base-v1.07.manifest.xml,
    reservation_id: r-6454f80d}
ssh-key-name: id_rsa_aurora
ssh-key-path: /Users/simonh/id_rsa_aurora
state: Allocated
user: root
Appendix C

Configuration template

{% if receive %}
{% for port in receive %}
[splunktcp://{{port}}]
disabled = false
queue = parsingQueue
sourcetype = tcp-{{port}}
{% endfor %}
{% endif %}

{% if tcpinput %}
{% for cfg in tcpinput %}
[tcp://{{cfg.port}}]
disabled = false
queue = parsingQueue
sourcetype = {{cfg.sourcetype}}
{% endfor %}
{% endif %}

{% if file_input %}
{% for cfg in file_input %}
[monitor://{{cfg.path}}]
disabled = false
sourcetype = {{cfg.sourcetype}}
{% endfor %}
{% endif %}
Appendix D

Generated configuration

[splunktcp://9997]
disabled = false
queue = parsingQueue
sourcetype = tcp-9997

tcp://9998]
disabled = false
queue = parsingQueue
sourcetype = syslog

[monitor://var/log]
disabled = false
host = possimpble
Bibliography


