Universiteit Antwerpen

Scarlet Nebula:
Cloud Management Toolkit in Java

Proefschrift voorgelegd op 30 mei 2011 tot het behalen van de graad van Bachelor in de Wetenschappen, bij de faculteit Wetenschappen, aan de Universiteit Antwerpen.

Promotoren:
Prof. dr. Jan Broeckhove
dr. Kurt Vanmechelen

Ives van der Flaas

CoMP Research Group Computational Modelling and Programming

Contents

Abstract

De beschikbaarheid van een bijna onbeperkt aanbod van (virtuele) computers die met een druk op de knop kunnen worden opgestart en vervolgens vanop afstand beheerd kunnen worden is een relatief nieuwe ontwikkeling in de computerwetenschap. De infrastructuur die nodig is voor deze zogenaamde Infrastructure as a Service *clouds wordt voorzien door verscheidene onafhankelijke bedrijven, zoals Amazon en Rackspace.*

Scarlet Nebula is een cross-platform Java applicatie die het mogelijk maakt om een willekeurig aantal computers in één of meerdere van deze clouds te starten en beheren. Scarlet Nebula biedt ook de mogelijkheid om deze computers te besturen via SSH of VNC en om de belasting per computer in het oog te houden.

Abstract

The availability of an almost unlimited supply of (virtual) computers that can be started by the push of a button and remotely operated is a relatively new development in the realm of computer science. The infrastructure needed for these so-called Infrastructure as a Service clouds is provided by several independent companies, such as Amazon and Rackspace.

Scarlet Nebula is a cross-platform Java based application that allows you to start and manage an arbitrary number of computing instances in one or more of these clouds. It also provides facilities for controlling instances through SSH and VNC connections and for monitoring the current load of instances.

CHAPTER 1

Introduction

1.1 Problem

The appearance of Cloud Computing has been a godsend for anyone interested in a financially interesting unlimited supply of computing power and storage capacity. The essence of Cloud Computing — the ability to treat computing power as a utility — dramatically decreases cost for many organisations, by making it unnecessary to purchase large amounts of expensive infrastructure.

Unfortunately, the tools for managing clouds did not evolve at the same speed the clouds themselves did. At the moment only proprietary or expensive tools for managing *Infrastructure As A Service* clouds exist. Scarlet Nebula aims to fill this gap by providing a cross-platform system that supports several clouds, allowing users to manage an arbitrary number of instances across clouds.

1.2 Goals and status

1.2.1 Goals

The main goal of Scarlet Nebula is to provide a cross-platform all-encompassing system that can be easily adapted to accommodate new clouds. Scarlet Nebula is not only able to manage instances in a cloud, it can also track the current load of instances and connect to the instances themselves, through both SSH and VNC connections.

1.2.2 Status

Scarlet Nebula is a stable Java application that fully supports Amazon EC2 and CloudSigma clouds out of the box, has an easy to use graphical user interface, and is easily adaptable to other clouds.

CHAPTER 2

Design

2.1 Requirements

The following features should be present in Scarlet Nebula:

- Starting, stopping, pausing, rebooting and terminating instances.
- Entering the credentials for a new account with a cloud provider.
- Creating and deleting firewalls, which instances can then use to protect themselves against unwanted traffic.
- Adding and removing rules to existing firewalls.
- Selecting a hard drive "image" from a library, on which a new computing instance will be based.
- Entering tags which provide meta-data for instances.
- Viewing an instance's current load.
- Starting an SSH connection to an instance.

2.2 Design

Scarlet Nebula is built up out of two major parts, the Core and the GUI.

2.2.1 The Scarlet Nebula Core

The Scarlet Nebula Core is the stateful system that communicates with a cloud provider and the computing instances themselves. The Scarlet Nebula Core keeps track of servers, cloud providers and SSH keys. It also allows a user to do a variety of things, going from simply starting a server to requesting to be updated when a new data point in a certain statistics data stream arrives. A simplified version of the way the Scarlet Nebula core is designed can be seen in Figure 2.1.

CloudProvider is the class that represents one of the most important concepts in Scarlet Nebula. A CloudProvider instance represents a connection with a certain cloud, like Amazon EC2, on a certain endpoint and with certain login credentials. Two sets of credentials for one cloud are handled as two distinct CloudProvider instances. A CloudProvider is responsible for keeping track of linked Servers and for managing Servers, firewalls, ... All CloudProviders are managed by a CloudManager, a singleton class.

A Server is the entity that represents a single virtual machine. A Server can be paused, rebooted, terminated, etc. It can also create CommandConnections to that server, or build a StatisticsManager for that server.

CommandConnections are used to communicate to a Server, usually through an SSH connection. A CommandConnection can be used to execute a command on a server returning a continuous stream of output data, it can be used to execute a command and return a string containing the Servers response and it can also be used to establish an interactive terminal-emulation connection with a server.

A StatisticsManager uses a CommandConnection to retrieve statistics from a Server. A class can subscribe to a StatisticsManager when that class wishes to be notified when new Datastreams and Datapoints are received.

2.2.2 The Scarlet Nebula GUI

As the name says, the Scarlet Nebula GUI provides an easy to use graphical user interface that allows the user to utilize the Scarlet Nebula Core to its fullest.

Because the final version of the Scarlet Nebula GUI contains more than 90 top-level classes and another 80 nested classes, a class diagram wouldn't be very useful.

Packages

The GUI is built up out of several packages, all located in the package be.ac.ua.comp.scarletnebula.gui:

(Main GUI Package) This package contains all classes that don't merit their own package. These are mostly classes that derive from a Swing component, like ServerList.

windows The windows package contains every window and dialog in Scarlet Nebula. E.g. TaggingWindow, ProviderPropertiesWindow, ...

welcomewizard The welcomewizard package contains the different pages in the Wizard that is shown when a user starts Scarlet Nebula for the first time.

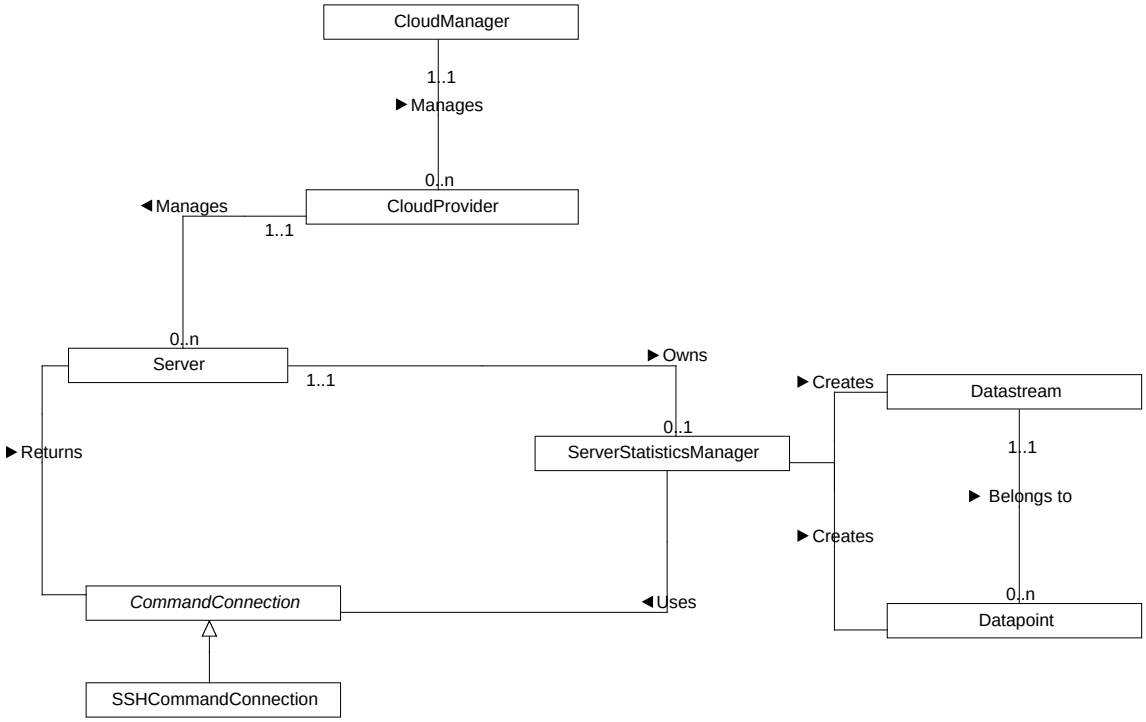

Figure 2.1: A simplified version of the Scarlet Nebula Core design.

keywizards The keywizards package contains the wizards and pages needed relating to adding, removing and importing SSH keys.

addproviderwizard The addproviderwizard package contains the pages and DataRecorder for the wizard that's used to add a new CloudProvider.

addserverwizard The addserverwizard package contains the pages and DataRecorder needed to start a new Server.

tests The tests package contains unit tests for classes related to the GUI, such as InputVerifiers.

CHAPTER 3

Implementation

As mentioned before, Scarlet Nebula can be divided into two major parts: the core and the GUI.

3.1 Scarlet Nebula Core

Instead of communicating with each individual cloud provider using their own proprietary API's, the Scarlet Nebula core uses a library called the Dasein Cloud API[1], which provides a unifying layer between the Scarlet Nebula core and the communication libraries provided by cloud providers.

The Scarlet Nebula GUI will communicate with cloud providers entirely through the Core, which does all of the the Dasein-related heavy lifting.

3.1.1 Communication through the Dasein Cloud API

As mentioned before, Scarlet Nebula communicated with Cloud Providers entirely through the Dasein Cloud API. The Dasein Cloud API provides a universal API and implementations for a lot of cloud providers. It does this by either using cloud provider's own communication libraries, or providing its own.

The Dasein Cloud API is backed by enStratus[1] and updated regularly. Unfortunately, documentation for the Dasein Cloud API is extremely limited, consisting of nothing more than a few pages and some partial API documentation.

As noted in [2], the Dasein Cloud API provides the following services:

- Access control services
- Accounting and Finance management services

[1] More information at `http://www.enstratus.com/`.

- Static IP address management services
- CDN management services
- Geographic/data center management services
- Virtual disk management services
- Firewall management services
- Machine image/template management services
- Virtual load balancer management services
- Push notification management services
- Auto-scaling management services
- Virtual server management services
- Cloud storage/blob storage management services

These interfaces can be used by querying their parent's interface. The service that was asked for will be returned if it exists for that cloud provider; if it does not exist `null` will be returned.

3.1.2 SSH Communication with Servers

Scarlet Nebula's main method for communicating with servers is an SSH connection, which takes the shape of a CommandConnection. To accomplish this, the JSch[2] SSH2 library is used. Although both code quality and documentation is mediocre at best, it does work pretty well. In the Scarlet Nebula GUI, a package called JCTerm[3] will be used in conjunction with JSch to display a graphical VT100 terminal emulator.

3.1.3 Statistics Subsystem

Although lots of cloud providers have facilities for monitoring server load, the extra associated cost and limited available data makes monitoring a server farm solely through the cloud provider an expensive and uncomfortable task. Because of this, Scarlet Nebula provides its own easily extendible statistics system.

Scarlet Nebula gathers statistics by running a program or script on each monitored instance through an SSH connection. This program or script will keep running as long as Scarlet Nebula wants to receive statistics and returns a continuous stream of newline separated JSON objects describing each datapoint. A new datastream is recognized when the first datapoint for that datastream arrives. Datapoints do not have to arrive in the same order, and the spacing between two datapoints from the same datastream can differ from stream to stream.

[2] JSch is availabe at `http://www.jcraft.com/jsch/` under a BSD style license.
[3] JCTerm is available at `http://www.jcraft.com/jcterm/`, under an LGPL license.

Data transmission format Several data formats were considered for transmitting statistics data. XML is structured and easily parsable in Java, but too verbose for efficient data transmission. A comma-separated format is also easy to parse, but not structured at all and harder to read from a human perspective. As a compromise between verbosity and structure, JSON[3] was chosen.

On the Scarlet Nebula side, JSON is deserialized into objects with the google-gson library[4].

More concretely, each JSON fragment should be valid with respect to the following JSON Schema:

```
{
    "name":"Scarlet Nebula Statistics Datapoint",
    "properties":
    {
        "datapointType":
        {
            "type":"string",
            "enum": ["RELATIVE", "ABSOLUTE"],
            "required":true
        },
        "datastream":
        {
            "type":"string",
            "required":true
        },
        "value":
        {
            "type":"number",
            "minimum":0,
            "required":true
        },
        "lowWarnLevel":
        {
            "type":"number",
            "minimum":0,
            "required":false
        },
        "mediumWarnLevel":
        {
            "type":"number",
            "minimum":0,
            "required":false
        },
        "highWarnLevel":
        {
            "type":"number",
            "minimum":0,
            "required":false
        },
        "max":
        {
```

[4] google-gson is available at `http://code.google.com/p/google-gson/` under an Apache License 2.0.

```
            "type":"number",
            "description":"Maximum allowed value for an absolute datastream",
            "minimum":0,
            "required":false
        }
    }
}
```

Newlines are used as separators for the JSON fragments and are therefore not allowed inside of a JSON fragment.

Types of streams Two types of datastreams exist: *relative* and *absolute* datastreams. Whereas in a relative datastream each value is in the range $[0, 1]$, absolute datastreams have an unlimited range when no `max` value is set for this stream and a range of $[0, \max]$ when a max value is set. The difference between the two types of datastreams will be essential later on, when displaying graphs in Section 3.2.3.

Warning Levels As previously mentioned when describing the data format, the optional fields lowWarnLevel, mediumWarnLevel and highWarnLevel can be present in a datapoint. When one or more of these fields are present, Scarlet Nebula remembers them, making it unnecessary to put them in each datapoint. When a datapoint's value exceeds the low-, medium- or highWarnLevel, that datastream is in what is called the respective low-, medium- and high warning zone.

Server-side Implementation After starting a new server, an attempt is made to run the default server-side command. The default command is a Bash-script, which runs an infinite loop that uses the Linux /proc pseudo-filesystem to gather data about the system.

Users that want to gather statistics from non-Linux systems, such as FreeBSD or Windows, or simply receive more advanced or specific statistics, can write and install a program or script on each server and then configure Scarlet Nebula to execute it. Command line parameters entered in Scarlet Nebula can be used to change the server-side program's behaviour, for example to change the frequency data is transmitted.

Data Usage When running the default server-side command, which returns a CPU measurement every five seconds, a memory measurement every 50 seconds and a measurement of the number of open TCP connections every 5 seconds, around 6.6kB is used per minute of statistics, per server. This can definitely be improved to around 4kB per minute per server, without any loss of resolution by not transmitting the low- medium- and high-warning levels every datapoint. When less datapoints are required, the default server-side command can easily be adjusted.

Examples The following are three examples of datapoints. Note that the newline in the first datapoint is purely for layout reasons and *may not* appear in the actual JSON.

```
{"datapointType":"RELATIVE","datastream":"CPU","value":0.09,
     "lowWarnLevel":0.5,"mediumWarnLevel":0.85,"highWarnLevel":0.95}
{"datapointType":"ABSOLUTE","datastream":"MEM","value":959,"max":1001}
{"datapointType":"ABSOLUTE","datastream":"Open connections","value":5}
```

3.2 Scarlet Nebula GUI

The Scarlet Nebula GUI is based on the Swing API. Swing is a lightweight GUI library that is built on a Model-View-Controller architecture. Unlike the Abstract Window Toolkit, which can be considered Swing's predecessor, Swing does not use the underlying operating system's widget system but instead draws its own widgets. However, Swing does provide a *Look and Feel* that makes applications look similar to native applications.

Swing is very versatile and provides many frequently used components, but also makes it easy to construct your own. A good book on Swing is [4].

3.2.1 VNC Communication with Servers

The Scarlet Nebula GUI uses a modified version of version 1.3.10 of the well known TightVNC library[5]; later versions were no longer written in Java.

TightVNC needed to be modified because

1. It was scattered with `System.out.println()` and `System.exit(1)` statements.
2. The TightVNC user interface looked a lot better after converting large portions from AWT to Swing.
3. All of the TightVNC code was in the anonymous top-level package, which is inaccessible from packages in newer Java versions.

3.2.2 Wizard Framework

One of the major concepts in the Scarlet Nebula GUI is the concept of a 'wizard'. A wizard guides the user through a sequential series of pages to accomplish a goal, such as starting a new instance. A wizard framework is not included in Swing, and to the best of my knowledge, no generally accepted third party packages exist.

After trying several open-source wizard frameworks, I decided to implement one myself. This wizard framework has the following main features:

[5] TightVNC is available under a GPL license at `http://www.tightvnc.com/`.

Figure 3.1: A page rendered using a SimpleWizardTemplate demonstrating the wizard framework's ability to determine the next page at run-time.

- *Extreme flexibility.* The current page can decide what page will be the next page after the "Next" button was clicked. This feature was necessary for complicated wizards (e.g. the SSH-key related wizard that is shown when creating a new cloud provider, see Figure 3.1) and was not to be found in any of the wizard frameworks I tried.

  One page can also be in several Wizards if both wizards derive their DataRecorders from the same abstract base class or pass data purely through constructor parameters.

- *Fully skinnable.* Wizards created by the wizard framework are fully skinnable. A wizard displays itself by using an instantiation of a class derived from the abstract class WizardTemplate.

3.2.3 Statistics Subsystem

When a new server is linked into the system, the GUI subscribes to all of the relevant statistics data streams provided by the server, as described in section 3.1.3. This data is then passed on to the JFreeChart[6] library, which renders the actual graph.

If a server is displayed in the main server panel, and this server is currently providing statistics, a compact graph indicative of a single datastream is displayed in the serverlist itself. Because of space restrictions, which make it impossible for axes to be displayed, the datastream represented in the main server list has to be a relative datastream, like the CPU stream which is displayed in Figure 3.2.

All of the available datastreams for an instance can be viewed by right clicking a server and choosing "View Statistics". A window containing these datastreams will be shown, as can be seen in Figure 3.3.

[6]The JFreeChart project is licensed under the LGPL and can be found at `http://www.jfree.org/jfreechart/`

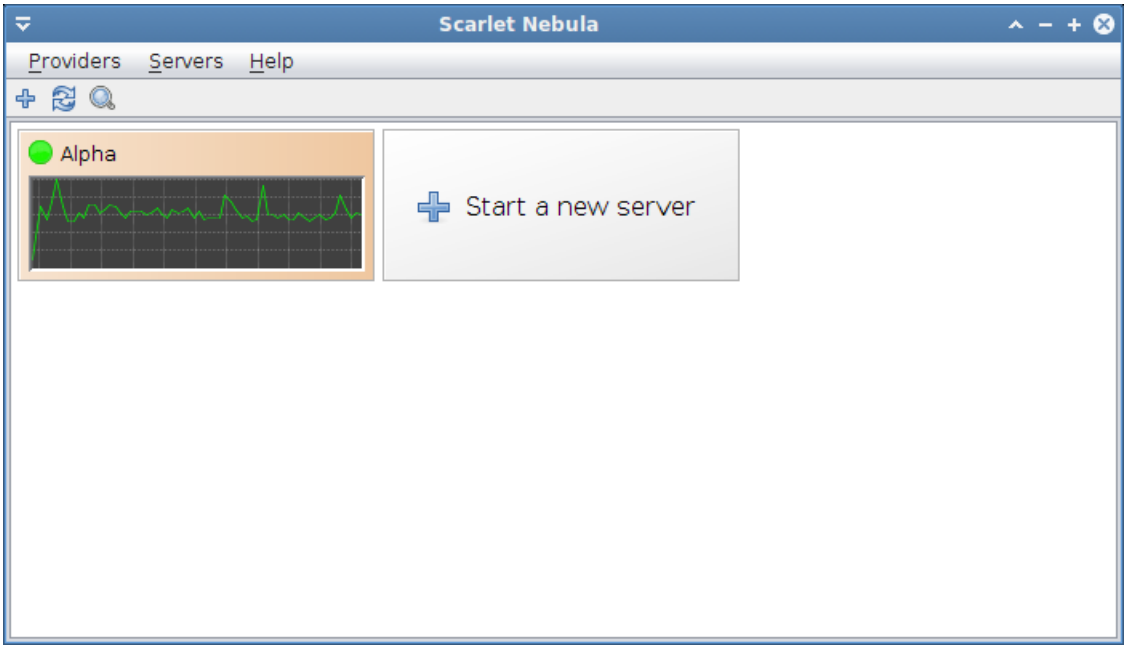

Figure 3.2: Scarlet Nebula managing a single server called 'Alpha', which is currently on a low warninglevel (indicated by the panel's color).

3.2.4 Filter Syntax

An advanced filtering syntax, similar to the one used by Google Mail, is available in Scarlet Nebula. Applying a filter to the main instance panel will limit the instances shown to those that pass the filter.

If a server is displayed after a filter is applied, this means it matches *all* of the terms without a negation qualifier and that it does not match *any* of the terms with a negation qualifier.

Filters can be written in either the "formal" syntax, in the "informal" syntax or in a mix of both.

The full search syntax is described in EBNF:

```
filter       = { term };
term         = formalterm | informalterm
informalterm = [ negation ] , argument;
formalterm   = [ negation ] , field , ':' , argument;
negation     = '-';
field        = 'inname' | 'tag' | 'size' | 'status' | 'inprovider';
argument     = quotedarg | unquotedarg;
quotedarg    = '"', { character | digit | ' ' | '\"' } , '"';
unquotedarg  = ( character | digit | '\"' | '\ '),
                    { character | digit | '\"' | '\ '};
character    = "A" | "B" | "C" | "D" | "E" | "F" | "G"
                    | "H" | "I" | "J" | "K" | "L" | "M" | "N"
```

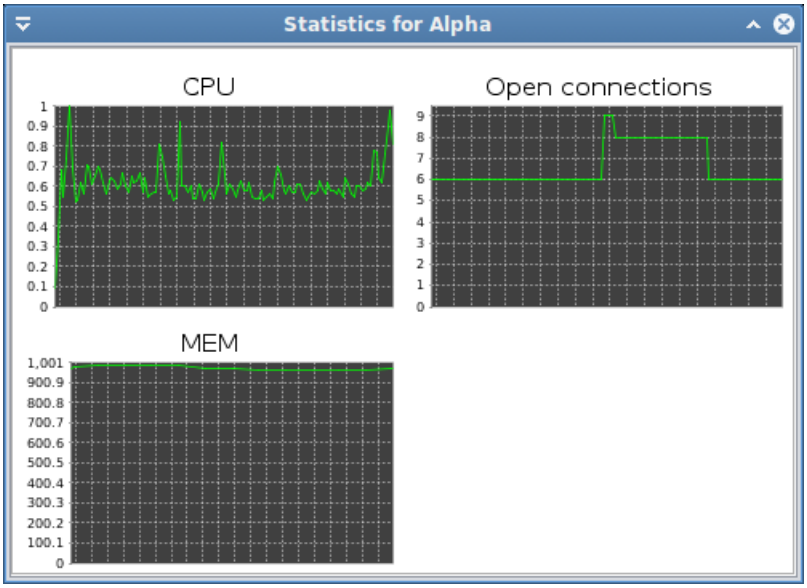

Figure 3.3: The statistics window for a server providing three datastreams. In this case a relative datastream for CPU load, an absolute datastream for the number of open TCP connections and another absolute datastream for total RAM memory usage.

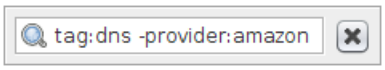

Figure 3.4: The filter field containing an example filter that shows all instances with a "dns" tag, except for those in Amazon's cloud.

```
                  | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
                  | "V" | "W" | "X" | "Y" | "Z" ;
digit         = "0" | "1" | "2" | "3" | "4" | "5"
                  | "6" | "7" | "8" | "9" ;
```

Formal Syntax

In the formal syntax, a filter is composed out of a list of space separated terms. Each of these terms contains information about what field should be filtered on, an argument and whether or not this search term is negated.

The following fields exist in the formal search syntax:

inname The argument must be contained in the instance's name.

tag The argument must be one of the instance's tags.

size The argument must be equal to the instance's size.

status The argument must be equal to the instance's current execution status. Possible arguments are `RUNNING`, `PAUSED`, `STOPPED`, `TERMINATED` and `RESTARTING`.

inprovider The argument must be contained in the instance's cloud provider's name.

Informal Syntax

Whereas in the formal syntax each of the filter terms had to have a field descriptor that determined which of the instance's attributes should be filtered on, the informal syntax does not. In the informal syntax, a search term consists only out of an optional negation qualifier and the actual argument. The argument of an informal term will be checked against *all* of the possible fields.

E.g. an informal search term "RUNNING" will be checked not only against the instance's current execution status, but also against the instance's name. If *any* of the fields match, this informal search term matches the server.

Examples

Example 1

To display only running servers:

```
status:RUNNING
```

Example 2

To display all paused servers for a provider called "Amazon EU".

```
status:PAUSED inprovider:Amazon\␣EU
```

Or with slightly different syntax:

```
status:PAUSED inprovider:"Amazon EU"
```

Example 3

To display all servers with a tag "ftp", except for those who also have a tag "web".

```
tag:ftp -tag:web
```

CHAPTER 4

Time Schedule

A more comprehensive version of this schedule can be found on the Scarlet Nebula Blog[5].

4.1 October

The first month was spent mostly by reading up on literature involving clouds, such as [6], [7] and [8], but also researching possible communication libraries with cloud providers. After researching, I decided to use the Dasein Cloud API as the library of choice for communicating with cloud providers. A lot of time was spent going through the extremely limited API documentation and Dasein Cloud API source code, trying to get a grip on the API.

4.2 November

In November I considered different graphics frameworks like the Abstract Window Toolkit and Qt Jambi, but finally settled for Java Swing. I also created a class diagram for the Scarlet Nebula Core and I spent some time on compiling and configuring the Dasein Cloud API. Later on, I implemented a few simple cloud provider operations such as listing servers.

4.3 December, January, February

I created the first version of the GUI. This includes the wizarding system, a lot of wizards, an initial version of the main serverlist, ... I also wrote a lot of code for the Scarlet Nebula core.

4.4 March

When writing the code that lists the machine images a provider has available, it became clear this functionality was not available in the version of the Dasein Cloud API I was using at the time. Luckily, I was able to upgrade the system to a new version of the Dasein Cloud API that included this functionality.

Furthermore, I implemented the new serverlist that shows a gridview in which each server is displayed as a rectangle containing its name, tags, and later on a graph. This is the final version of the serverlist as can be seen in Figure 3.2.

4.5 April

In April I implemented the entire statistics subsystem, including the server-side application and the Scarlet Nebula-side code. I also tried out different ways of indicating to a user that an asynchronous operation was running and implemented this for several lengthy operations.

4.6 May

I created the screen which makes it possible to add, remove and configure firewalls when starting a server. I also fixed a large number of bugs, finished the import key wizard, made a properties screen for CloudProviders and greatly improved the screen used to choose a machine instance type.

Later on, I added support for CloudSigma clouds, added a built-in VNC viewer based on TightVNC[1], and started writing this document.

[1] Available under the GPL on `http://www.tightvnc.com/`.

CHAPTER 5

Conclusions

In this section some of the larger choices taken in the development of Scarlet Nebula will be reflected upon. It will also contain my vision of the future of the project.

5.1 Decisions

5.1.1 Dasein Cloud API

Even with its severly lacking documentation and rapidly changing API, choosing the Dasein Cloud API over my own unification layer was still a good decision. It allowed me to think about communication with clouds at a higher level and freed me from a lot of (relatively small) problems, e.g. HTTPS implementations.

5.1.2 Swing

Choosing Swing as the GUI framework used in Scarlet Nebula proved to be an excellent choice. Swing was pleasant to work with, has a good architecture and made it easy to construct custom components. I did however on several occasions use a small addition to Swing called SwingX[1] for its `Painter` support in standard components.

[1] SwingX is provided by Oracle Corpration, more information is available at `http://swinglabs.org/`.

5.2 Future

The final version of Scarlet Nebula is stable and fully functional. It supports Amazon and CloudSigma clouds and can manage both Unix and Windows based instances.

However, a large product like Scarlet Nebula is never finished and some possible additions include:

1. Support for more cloud providers, for example Rackspace. Although Amazon EC2 and CloudSigma are very different from a functionality point of view, adding support for CloudSigma was still only a few hours work. Adding support for new cloud providers should be easy and painless.

2. Support for load balancers.

3. Built-in remote desktop viewer for better compatibility with Windows (at the moment instances running Windows can only be administered using the built-in VNC viewer or external software).

4. Support for uploading your own machine images, when the underlying cloud supports it. This would be extremely easy from a Core perspective and only slightly harder to implement in the GUI.

Bibliography

[1] G. Reese. The Dasein Cloud API. [Online]. Available: http://dasein-cloud.sourceforge.net/

[2] ——. (2010, April) Dasein Cloud Overview. [Online]. Available: http://mesh.dl.sourceforge.net/project/dasein-cloud/Overview.pdf

[3] D. Crockford, "RFC 4627 - The application/json Media Type for JavaScript Object Notation (JSON)," IETF RFC, JSON.org, Tech. Rep., 2006. [Online]. Available: http://tools.ietf.org/html/rfc4627

[4] M. Hoy, D. Wood, M. Loy, J. Elliot, and R. Eckstein, *Java Swing*, 2nd ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002.

[5] I. van der Flaas. The Scarlet Nebula Blog. [Online]. Available: http://scarletnebula.blogspot.com/

[6] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 25, 2009.

[7] R. H. Katz, "Tech titans building boom," *IEEE Spectrum*, vol. 46, no. 2, pp. 40–54, 2009.

[8] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, Feb 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html