StratusLab

Enhancing Grid Infrastructures with
Virtualization and Cloud Technologies

Software Integration Final Report

Deliverable D4.6 (V1.0)
28 May 2012

Abstract

This document reports on the integration activities performed during the whole StratusLab project, focusing on lessons learned and identifying the areas of improvement, throughout the different releases of the StratusLab software, including the process used during the development, integration and test. The goal of this document is also to provide lessons that could be applied as the project transitions to an open source consortium, such that it continues to improve its performance.

StratusLab is co-funded by the
European Community's Seventh
Framework Programme (Capacities)
Grant Agreement INFSO-RI-261552.

Contributors

Name	Partner	Sections
Marc-Elian Bégin	SixSq	All
Louise Merifield	SixSq	All

Document History

Version	Date	Comment
0.1	5 May 2012	Initial version for comment.
0.2	23 May 2012	Completed statistics and updates figures.
1.0	28 May 2012	Integrated review comments from David O'Callaghan.

Contents

List of Figures

List of Tables

1 Executive Summary

StratusLab is a two year project, with a challenging programme of work. The project exists in the context of cloud and grid computing, a fast and very dynamic field in distributed software engineering. In order to manage this challenging situation, the project decided to adopt an agile software development process, called Scrum, alongside a set of engineering practices, some taken from eXtreme Programming (XP).

This report describes the strategy and techniques used during the two-year project to deliver the required system incrementally, building on the strength of the previous increment, generating feedback from users, and feeding this back into the prioritisation of the functionality released with every iteration.

While the Agile eco-system is rich and diverse (e.g. Extreme Programming - XP, Scrum, Lean Development, Kanban), StratusLab chose to focus its organisational structure on Scrum. Scrum has a clear model and is prescriptive of the dynamics between the customer and the provider, the emergence of functional specification (i.e. Product Backlog) and the rhythm of development (i.e. sprints). While other methods would also apply, such as XP (as well as derivatives like Test-driven development - TDD), we focused our effort on the Scrum part of the development process.

The Scrum process described early in this document focuses on the management of functionality and prioritization of features for each StratusLab release. This report then changes its focus towards the engineering practices used in the integration and test activities required to deliver high-quality StratusLab software.

To deliver a quality solution the StratusLab project employs a set of practices of continuous automated integration and deployment in the course of the development process. The set of practices (properly pipelined, automated and frequently executed) allows for rapid feedback on the quality of the software stack at all levels as well as its delivery models.

The project uses Git (a distributed revision control system) as its code repository. The build procedures themselves in StratusLab are implemented using Maven2. This tool is the de-facto standard in the Java world, but also provides support for other languages. Maven2 provides a rich set of plugins, capable of performing complex tasks.

In order to guarantee that everything is working as we move forward with new features, improvements and fixes, the project uses a continuous integration (CI) infrastructure, composed of a main server and a number of dedicated 'slave' machines.

Having reviewed and explained the methodologies and processes used to integrate and test StratusLab, the last part of this report brings together important lessons learned. These include a wide range of lessons:

- Improve, as well as simplify, the continuous integration system
- Automate further the creation of release candidates and repositories (e.g. snapshot, certification, release)
- Documentation updates for every development and integration task
- Further automation of machine re-imaging to ensure clean installations using Quattor
- Push patches back to external code providers faster to reduce flakiness in our build system
- Encourage working on HEAD/master branch whenever possible
- Encourage committing on a regular basis
- Encourage a stop-the-line culture regarding Hudson jobs to avoid committing code on a broken system
- Improve and increase variety of modern operating systems support for StratusLab user tools and cloud system.
- Support manual and automated installation capabilities
- Engage with third party providers of important software dependencies to avoid unexpected integration effort

This short project was very productive with several releases and numerous sprints delivering significant functionality towards the project goal of a fully functional high quality cloud distribution. The build and test activities summarised in this report were also a great source of lessons learned, which we are convinced will help the project improve even further as it becomes a consortium that will depend on a community of open source developers.

2 Introduction

StratusLab is a two year project with a challenging programme of work. The project exists in the context of cloud and grid computing, a fast and very dynamic field in distributed software engineering. In order to manage this challenging situation, the project decided to adopt an agile software development process, called Scrum, alongside a set of engineering practices, some taken from eXtreme Programming (XP).

Figure 2.1 shows the high-level architectural vision of StratusLab v2.0. This report describes the strategy and techniques used during the two-year project to deliver the required system incrementally, building on the strength of the previous increment, generating feedback from users, and feeding this back into the prioritisation of the functionality released with every iteration.

As can be seen in this diagram, StratusLab is composed of a number of components and services, each integrated and tested individually and as a whole, using an automated procedure and on a regular basis (i.e. generally several times per day). This report describes how the development, integration and test activities are orchestrated and synchronised to produce the StratusLab distribution.

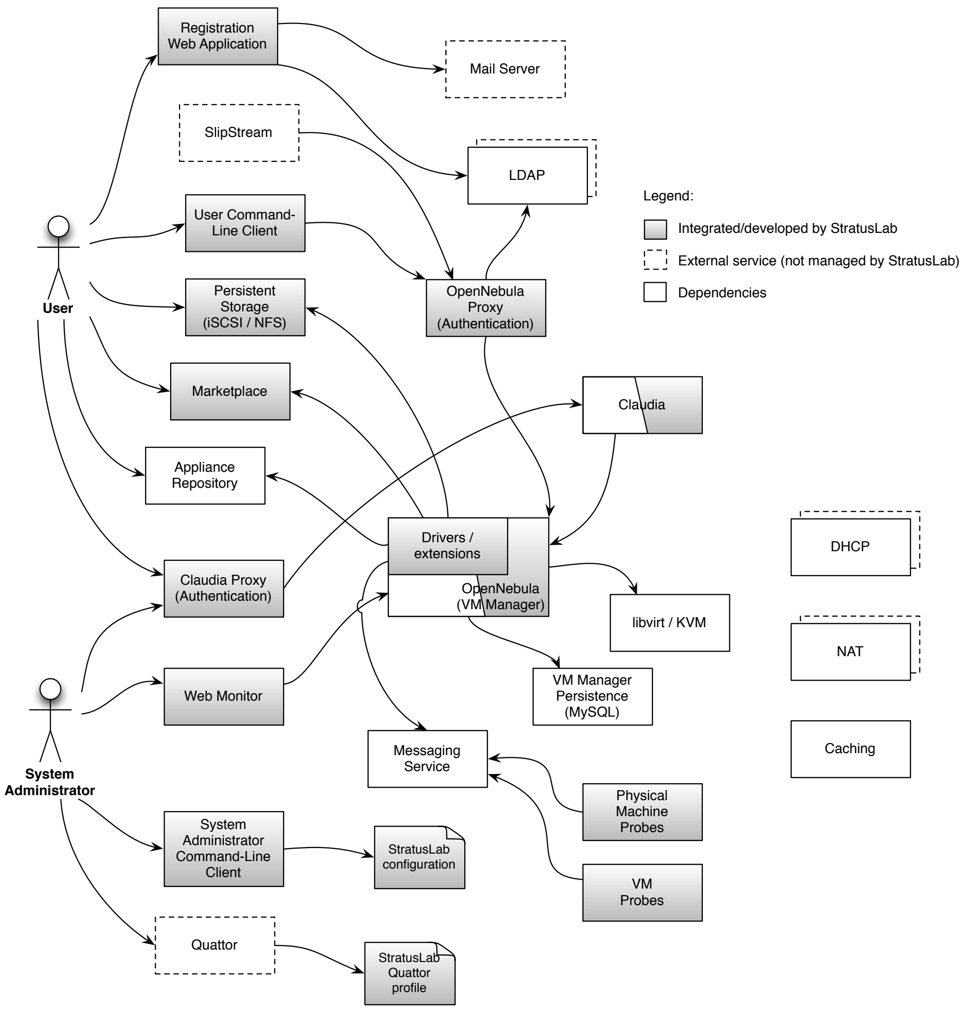

Figure 2.1: *StratusLab v2.0 Architectural Vision*

3 Agile Process: Scrum

3.1 Agile Overview

While the Agile eco-system is rich and diverse (e.g. Extreme Programming - XP, Scrum, Lean Development, Kanban, Core Principals), StratusLab chose to focus on organisational structure on Scrum. Scrum has a clear model and is prescriptive of the dynamics between the customer and the provider, the emergence of functional specification (i.e. Product Backlog) and the rhythm of development (i.e. sprints). While other methods would also apply, such as XP (as well as derivatives like Test-driven development - TDD), we focused our efforts on the Scrum part of the development process.

The main reasons for choosing Scrum were:

- Client-centric approach encouraging convergence towards fulfilling the real needs
- Improved project visibility
- Promotes trust between all stakeholders
- Controlled incorporation of changes
- Continuous inspection and improvement of the process
- Higher quality software
- Simpler solutions, easier to maintain and evolve
- More predictable delivery of functionality

While a complete description of Agile and Scrum is outside the scope of this report, the following explains the main artefacts and processes involved in StratusLab's implementation of Scrum. In Scrum, each iteration (called a "sprint") starts with a planning meeting. During this meeting, high-priority requirements are reviewed, analysed, decomposed into tasks and selected for the next sprint. The duration of each sprint varies normally between one and four weeks, with a trend towards shorter sprints. StratusLab decided to standardise at around three weeks, to take into account the integration overhead and the distributed nature of

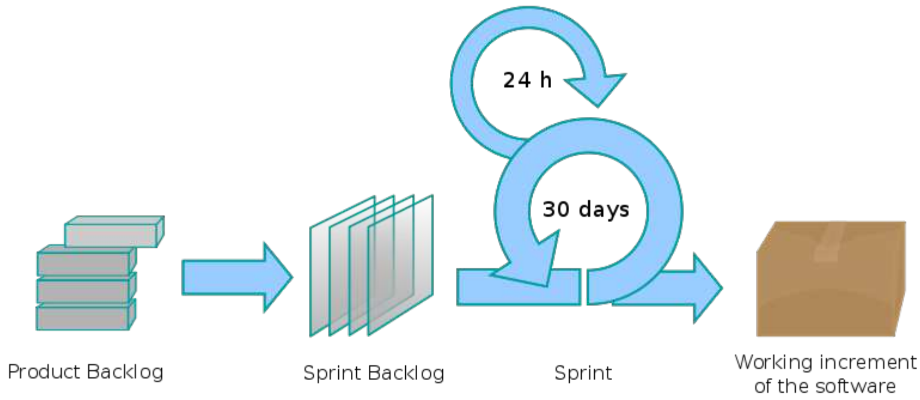

Figure 3.1: *Scrum Overview*

our teams. Each sprint ends with a sprint review, composed of two events: a retrospective and a demo. The objective of the retrospective is to review the past sprints' performance, with a focus on emulating what was particularly effective, while eliminating impediments. For the demo, the team assembles and deploys the software and presents the implementation of the requirements selected for that sprint. Different stakeholders can be invited to this event, if necessary, including for example end-users. This is an important generator of feedback and insights, which can be fed into the requirements, and provide new data for prioritisation. The process is illustrated in Figure 3.1, with the distinction that StratusLab adopted a three week sprint duration instead of the four week illustrated on the figure.

Another important Scrum event (but also required by most agile methods) is the daily meeting, daily stand-up or daily Scrum. The sole purpose of this meeting, which should not exceed 15 minutes, is to foster continuous and fluid communication between all teams. Stand-up meetings focus on reporting what has been accomplished since the last meeting, what is planned until the next meeting and identification of any impediments. Further, to alleviate the fact that for StratusLab, stand-up meetings cannot be performed face-to-face, a new item to the standard topics is a mention of the state of the Hudson continuous integration server, such that any failures are highlighted to all and corrective actions are immediately scheduled. StratusLab stand-up meetings take place Monday to Friday at 10:30 sharp, Paris time.

Scrum also defines three key roles:

Product Owner The person (or representative) responsible for maintaining the Product Backlog by representing the interests of the stakeholders.

Scrum Master The person responsible for the Scrum process, making sure it is used correctly and maximizing its benefits.

The Team A cross-functional group of people responsible for managing itself to develop the product.

These three roles are mapped differently to each project structure, depending on the project, its stakeholders, contractual setup, distribution of skills and locations across participants, etc. In StratusLab, the role of Product Owner is fulfilled by a group composed of all work package leaders and senior technical members of the project. Charles Loomis, the project director, arbitrates and helps focus the group regarding prioritisation. The role of Scrum Master is generally held by Marc-Elian Bégin, WP4 leader, but is regularly filled by other members when required.

Each sprint is concluded with a demo. This is the opportunity to show the project members, via a alive demo, the completion of each user story, improvements and bug fixes. Over the course of the project, the definition attached to completing an item of work (definition of 'done') evolved and is expected to continue evolving. This natural evolution also follows the maturity of our build and test infrastructure, discussed in more detail in Chapter 4.5. This improvement to the doneness of our tasks is important the reduce the risk that problems are discovered during deployment by WP5, or worse that we break a feature already delivered in a previous version (also called regression).

An additional reason for choosing relatively short three-week sprints is that, assuming we can release often enough, bug fixes can be released with normal releases, as opposed to requiring special bug fixing releases. While this policy might have to be revisited if critical problems are detected to which an urgent fix is required, it offers a much simpler rollout model.

3.2 StratusLab Scrum Measurements

The main artefact driving the Scrum process is the 'Product Backlog'. For StratusLab, the Product Backlog is captured using JIRA (and the GreenHopper plugin).

While the planning meetings, demos and daily stand-up events, described in the previous section, provide the 'tactical' process for efficient project execution, we also need a longer term and strategically focused think tank. This important function is fulfilled by the Technical and Scientific Coordination Group (TSCG), chaired by Ruben Montero from UCM, to which all work package leaders and senior project members contribute. From the set of priorities defined by the TSCG, the product backlog is maintained (i.e. items added and removed), including clear priorities. In effect, the TSCG in StratusLab is the custodian of the product backlog. Charles Loomis, as the project director, present at most planning meetings, has de-facto fulfilled the role of Product Owner.

Figure 3.2 illustrates the virtuous cycle feeding the StratusLab agile process. The diagram clearly shows the important role and actions each activity has and performs. This feeds into the feature prioritisation performed by the TSCG, as well as the architectural vision required to provide a solution. Using our agile approach we can then more easily follow and adapt to deliver these features.

As mentioned above, the StratusLab Product Backlog is captured with JIRA

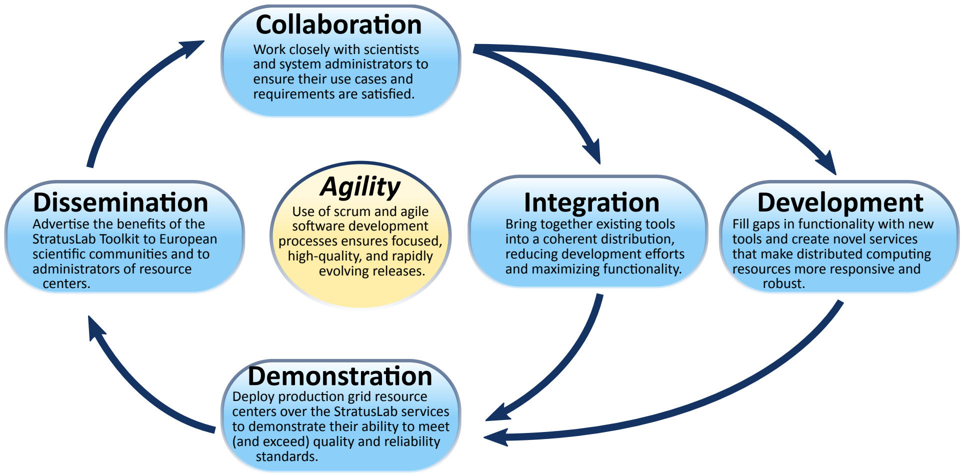

Figure 3.2: *Activities Interactions*

using four types of tickets:

Epic A high-level feature description, which is normally decomposed into specific user stories. Epic can also be seen as small vision statements, useful when expressing vague or higher-level objectives.

User Story A feature requiring several tasks to complete, possibly involving several partners, but sized such that it can be implemented within a single sprint.

Bug A bug, requiring a simple fix, normally affecting a single component

Improvement A small upgrade in current functionality, not requiring a new User Story

This level of granularity allows us to manage all items during sprint execution.

During the whole project, we have completed 27 sprints (including sprint 0). Figure 3.3 gives an overview of the number of items (i.e. user stories, bug fixes, improvements) completed for each sprint (up to sprint 24). We can also correlate important project events on this picture. For example, the dip in sprints 7 and 8 correspond to the winter and new year break, where several team members were on holiday, reducing the effort available to complete items. The surge in bugs in sprint 3 corresponds to our first release (v0.1) where certification in view of production deployment identified issues. From sprint 10, the number of 'improvements' grew, corresponding to feedback from usage, expressed as small improvement requests, as opposed to new functionality. The spikes of user stories for sprints 17, 20 and 23 correspond roughly to releases. The level of development activity remained high all the way to the end of the project, despite a significant amount of effort diverted to the production of deliverables towards the end of Quarter 8.

During the entire project duration, 12 releases (see Table 3.2) were performed, including two major releases–v1.0 and v2.0.

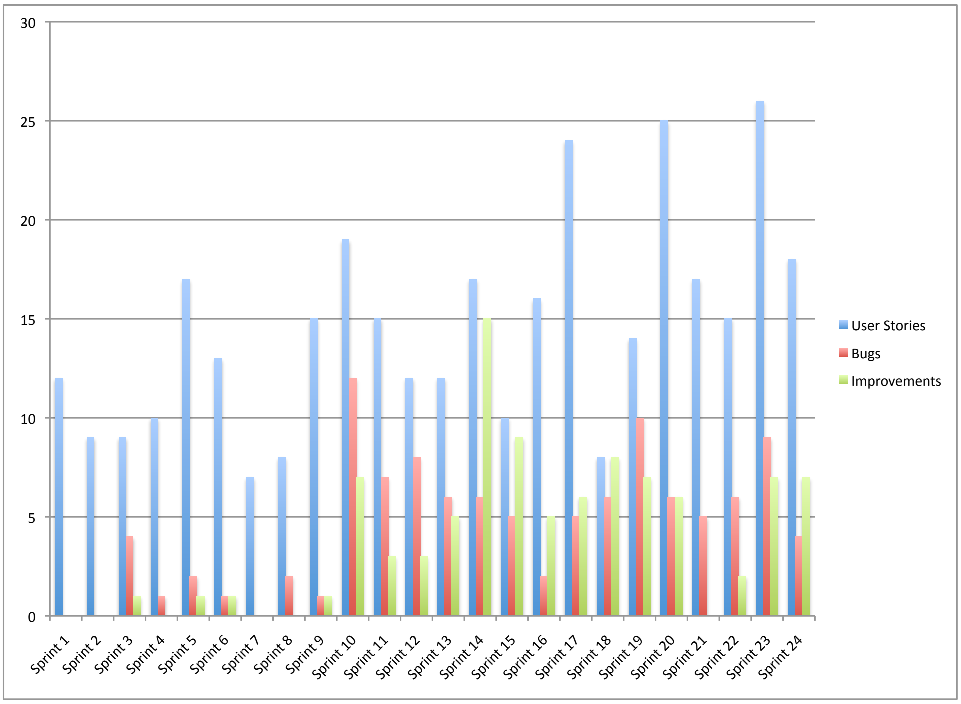

Figure 3.3: *Sprints Overview*

Table 3.1: *StratusLab Completed Tickets Summary*

Completed user stories	408
Fixed bugs	140
Completed improvements	118

During the project, as summarized in Table 3.1 we completed 408 user stories, fixed 140 bugs and implemented 118 improvements on already delivered features.

Each release was tested and deployed on dedicated test machines, prior to being released to WP5.

Table 3.2: *StratusLab Releases*

v0.1	9 November 2010	Sprint 5
v0.2	23 December 2010	Sprint 7
v0.3	15 March 2011	Sprint 10
v0.4	9 June 2011	Sprint 13
v1.0	4 July 2011	Sprint 15
v1.1	16 September 2011	Sprint 17
v1.2	19 December 2012	Sprint 20
v1.3	30 January 2012	Sprint 21
v1.4	28 February 2012	Sprint 22
v1.5	27 April 2012	Sprint 24
v1.6	21 May 2012	Sprint 25
v2.0	25 May 2012	Sprint 26

4 Engineering Practices

The Scrum process described in Chapter 3 focuses on the management of functionality and prioritization of features for each StratusLab release. This chapter focuses on the engineering practices used in the integration and test activities required to deliver high-quality StratusLab software.

4.1 Continuous Integration and Deployment

To deliver a quality solution the StratusLab project employs a set of practices of continuous automated integration and deployment in the course of the development process. The set of practices (properly pipelined, automated and frequently executed) allows for rapid feedback on the quality of the software stack on all of its levels as well as its delivery models.

To facilitate continuous integration effort the StratusLab project uses Hudson.

Build Automation Build automation is handled using Apache Maven. All separate software components of the StratusLab project (identified as separate projects in the git repository) are Maven projects. This allows for a common build interface and specification of concise instructions for the components testing, building and upload to the project's distribution repositories.

Building all commits to baseline To reduce the number of conflicting code changes, regular commits (many times a day by a single developer) to the code baseline are encouraged and being practiced. The commits are immediately unit-tested, built, deployed and functionally tested to reduce the window between commit and the feature/fix actual functional usage/testing. The latter facilitates an immediate awareness by the developers of any possible failures caused by the code changes. In case of failures, responsible people are notified immediately by email.

Deployment, Integration and System Testing A special infrastructure comprising physical nodes as well as virtual machines was deployed by the project to resemble as much as possible a scaled production environment. The infrastructure is used to exercise per-component feature testing. Most important, this includes the execution of integration and system tests simulating user interactions on a deployed system built from the latest snapshot versions of the software.

Release Automation To ease the software stack release procedure special Maven build targets were defined for each component and per-component release jobs were created in Hudson. Also, a common catch-all release job per supported Linux distribution triggering the per-component release jobs was created. The latter allows the release of the StratusLab software stack with virtually a single click.

4.2 Configuration Management and Version Control

The project uses Git (a distributed revision control system) as its code repository. The project's Git repository contained the following logically identified subprojects (called repositories in Git) each often composed of more than one submodule, from which the build procedure produces a number of software packages.

stratuslab-authn Authentication proxy for OpenNebula and Claudia, providing support for local or LDAP managed username/password, as well as X.509 and grid certificates

stratuslab-benchmarks A set of standard benchmarks for the cloud installation

stratuslab-claudia Claudia system

stratuslab-client End-user client for remote creation and management of virtual machines, system administrator installation and configuration tools, webmonitor for simple monitoring of physical nodes and virtual machines.

stratuslab-image-recipes Image recipes for creating base virtual images from standard operating systems

stratuslab-marketplace Marketplace providing a service for query and registration of virtual appliances metadata (both machine and disk images), with cryptographic support for integrity checking and endorsement of the images.

stratuslab-one StratusLab custom OpenNebula build, integrating the latest upgrades, patches and customisation for StratusLab

stratuslab-quattor Set of Quattor templates for automated installation and configuration of StratusLab

stratuslab-registration A registration web application providing confirmation workflow for users of a specific StratusLab installation.

stratuslab-storage A persistent storage rest application for the creation and management of persistent disk for virtual image instances and extra storage.

project-documents All project documents, including deliverables and dissemination material

Using a common Git store, all code contributors can share code. Like all good version control systems, Git provides a merge feature, ensuring that conflicting commits can be resolved without loss of information. All team members are encouraged to commit (and push) their code often.

An important component in StratusLab is OpenNebula. In order to better support StratusLab, the OpenNebula team at UCM developed new features and provided bug fixes requested and reported by StratusLab integration and operation activities. OpenNebula has a release cycle that does not directly match StratusLabs rapid incremental release strategy. To resolve this mismatch the UCM team created a clone Git repository (OpenNebula also uses Git for managing its code) dedicated to StrartusLab. This allows UCM to rollout directly to this repository code so that StratusLab can release, independently from OpenNebula releases. In order to avoid the risk of divergence between the master OpenNebula code base and StratusLab's custom version, UCM regularly merges back to the master code base. With StratusLab version v2.0 beta, we now are again synchronized on the production Open Nebula v3.2, since this version of Open Nebula was already released and only maintenance patches are being applied.

The same applies for patches. StratusLab team members regularly modify OpenNebula extensions (e.g. to fix bugs, extend functionality). This is done by defining patch files that are applied by the Maven build procedure (see next section for detail). These patches have the advantage of providing immediate fixes, but over time can become expensive to maintain as the code they patch changes over time. To mitigate this situation, OpenNebula regularly integrates these patches into the mainstream code base, so that these patches can be removed.

Preparing for the transition towards an open source community driven consortium, the Git repositories have been migrated to GitHub[1].

4.3 Building Software

The build procedures themselves in StratusLab are implemented using Maven2. This tool is the de-facto standard in the Java world, but also provides support for other languages. Maven2 provides a rich set of plugins, capable of performing complex tasks. Here are the operations that we execute using Maven2 when building StratusLab packages:

- extract and install dependencies
- build binaries
- execute unit tests
- package
- install

[1] See http://github.com/StratusLab/.

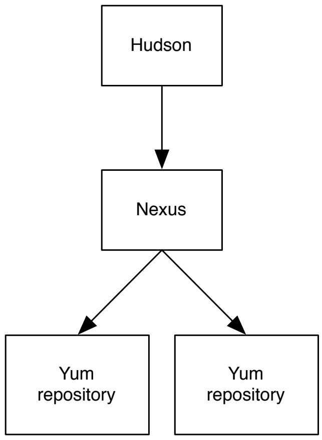

Figure 4.1: *Packaging Publishing*

- configure
- deploy to remote server
- release

While the details of every step listed above goes beyond the scope of the current report, here are a few important aspects worth mentioning regarding these steps. The packages are currently only created in the RPM format. While we have partial support for Debian packages (compatible with Ubuntu), we are not currently able to release a full distribution in .deb package format.

Having said that, we also release the end-user command-line client in tarball format, which is tested on CentOS 6.2, Fedora 14 and 16, Ubuntu and Windows.

Maven2 generates the packages and deploys them to a Maven2 server-side component called Nexus. The Nexus server provides a convenient way to manage dependencies. This is important so that developers can very quickly create a development environment using Maven without having to install software from different sources.

Maven is also used to generate binary packages (e.g. RPM) which are published and uploaded by the deploy target to Nexus. From there, RPMs are then registered to package repositories, such as YUM repositories for RedHat-derived distributions. This process is illustrated in Figure 4.1. The packages generated by Maven are also configured to include any runtime dependencies, such that the packaging system can pull all required dependencies during installation.

Further, the same process is used to handle snapshot, certification and release packages. This means that we can have a clear separation between the packages that are produced with every build (i.e. snapshots), and the packages that are used for certification and eventually released and therefore visible by our users.

For certification packaging, the Maven2 release plugin is used, which automatically adjusts the version number of every package and incrementing the number, with Git commits at every stage, such that the process is reversible and fully traceable. Once a set of packages are certified, the project can decide to promote these packages to the release repository, making them the latest released version.

4.4 Hudson

In order to guarantee that everything is working as we move forward with new features, improvements and fixes, the project uses a continuous integration (CI) infrastructure, composed of a main server and a number of dedicated 'slave' machines.

The project chose Hudson as the CI server. Hudson is open source and the de-facto standard CI solution at the moment. Hudson executes a number of 'jobs', from triggers ranging from code commits in Git, on a schedule, or from the successful completion of a parent job. The jobs comprise the following actions: code checkout, unit-testing, build, deployment, functional multi-service testing as well as release of all the StratusLab services and components on the supported platforms.

For example, every day, we test a complete installation of the latest snapshot version of StratusLab. At 02:00, two machines are re-imaged (i.e. operating system reset) with a pristine operating system installation. This re-imaging process is managed by Quattor at LAL. At 3:00, a Hudson scheduled job triggers and executes the job workflow illustrated in Figure 4.2. After the re-image, the machine is upgraded such that it contains the right dependencies, as specified in the online documentation. Then the StratusLab front-end (including OpenNebula) is installed and configured. From that point, several jobs are executed, testing features and/or adding supplementary software before testing it. For this process to work, two Quattor controlled machines are used, one for the front-end, the other for the node.

Another important workflow is the building of each package. This process is illustrated in Figure 4.3, in this case for releasing the packages. From a single Hudson job trigger, we are now able to preform a release of the complete StratusLab distribution. Each package can also be released individually, by directly triggering the corresponding job.

We currently have a significant Hudson infrastructure, comprising of eight physical servers and four virtual machines. We try to use as many virtual machines as possible, instead of physical servers. However, since several of the Hudson jobs use and test virtualisation, we require a physical server, since for example it is not possible to run a virtual machine inside another virtual machine (unless para-virtualisation is used inside full-virtualisation, which adds its own complexity and biases). Newer version of KVM also allow to run KVM inside KVM, but our target operating systems (e.g. CentOS 6.2) do not run this newer kernel.

We did not construct our build and test infrastructure overnight. On the contrary, we have built it over successive sprints, growing it as needs were identified, such that we could release software very early after project kickoff, carefully trading off effort invested in the build and test infrastructure and tools from integration and development tasks.

The Hudson server polls the Git repository regularly, checking for recent commits, and triggers build and test jobs. This means that developers receive feedback a few minutes after each commit, if their change has introduced a regression. This near immediate feedback gives the developer an opportunity to fix the problem while the problem domain is fresh in his/her mind, as opposed to being told that a problem occurred following a change committed days or even weeks before. Finally, assuming that all tests were green (passed) before the commit, the sources of errors are much narrower, compared with integration activities trying to integrate several changes at once.

4.5 Installation Strategies

StratusLab supports two installation strategies:

Manual Installation This method consists of a set of commands that perform the necessary installation steps based on information in a single configuration file.

Quattor Installation This method supports an automated installation of machines, which can scale to very large deployments. This method is supported by Quattor, for which StratusLab maintains specific profiles available in the standard StratusLab distribution.

Both methods are used in our automated build and test procedures, such that they are both tested regularly in order to identify any discrepancies that could occur between the two systems. This ensures that whichever the method chosen by system administrator, the resulting StratusLab deployment will be nearly identical.

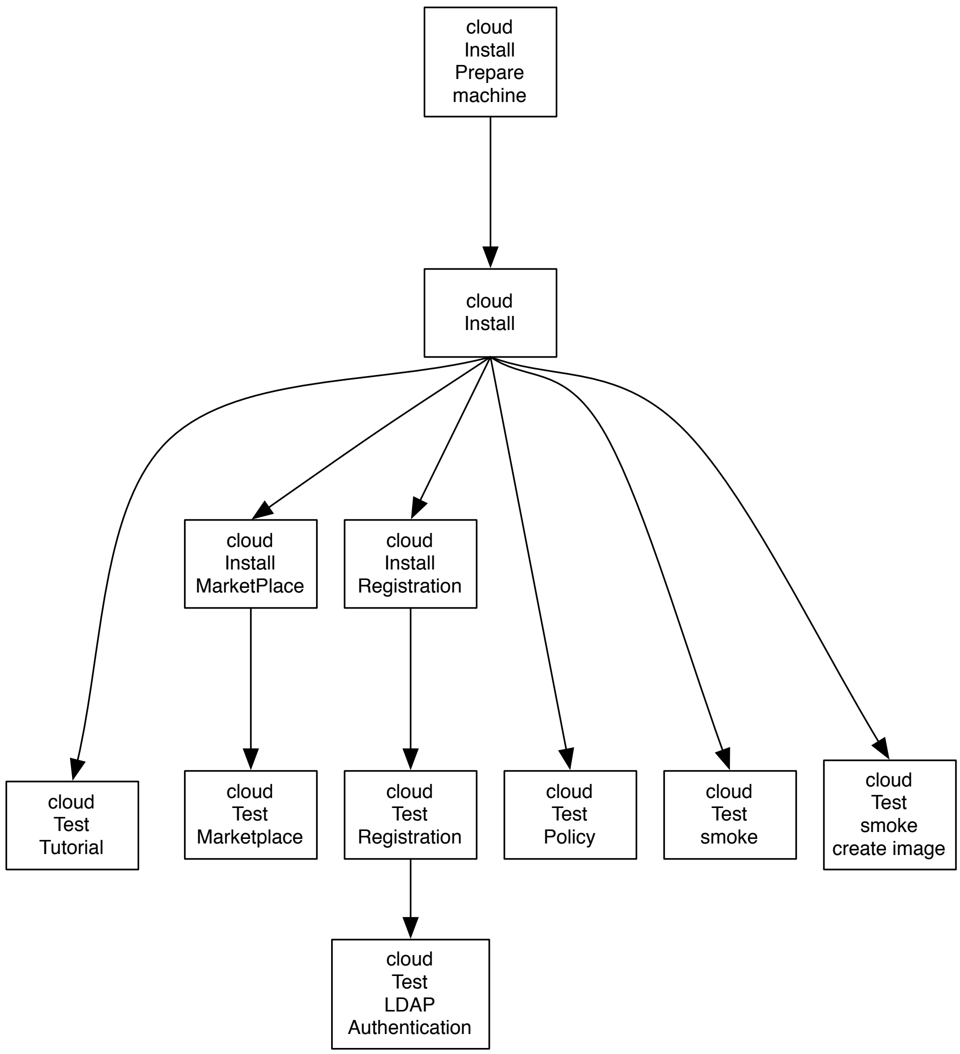

Figure 4.2: *Hudson Jobs for Daily Smoke Tests on Full Installation*

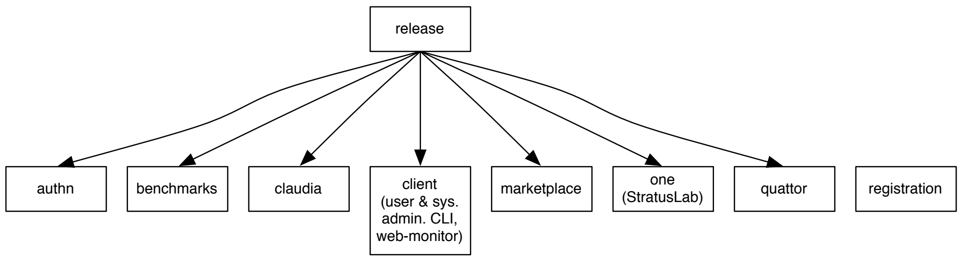

Figure 4.3: *Hudson Release Jobs*

5 Lessons Learned

Having reviewed the agile process the project uses to manage requirements and development, followed by the engineering practices used to develop, integrate and test the StratusLab software, we will now summarise lessons learned, with which we hope to improve further as we transition towards a community-based consortium.

1. Throughout the project, we took the opportunity improve and reshape the Hudson server and redefine the job topology, making it more flexible and efficient. We also put the Hudson job definition under version control, to avoid catastrophic loss of information as we experienced during the first year of the project. This allows us to roll back to previous configuration more easily if and when required.

2. Despite our best efforts, several production system upgrade were not completely straightforward, with longer than expected down-time. Although a significant improvement was made during the second year of the project, we can still improve significantly. The Persistent Storage Service can be configured in different ways, allowing system administrators to take advantage of different storage strategies, such as iSCSI or distributed file systems. The test, certification and migration of production systems using different backends is a challenge and requires sophisticated tooling. Although we made great progress in automating the certification of StratusLab in representative deployment contexts, we can still improve. For this, we need to take advantage of parameterised test jobs in Hudson, such that the same job definitions can be applied to different deployment models.

3. We believe that releasing more often, closer to sprint cycles, would improve our ability to better upgrade, with minimal disruption for users, while providing new features more often. As more sites deploy StratusLab however, we have to be careful not to alienate system administrators with too frequent releases. Further, we need to ensure that upgrade paths are tested and solid, such that no loss of data occurs when upgrading to a new release. Finally, we could consider releasing different components are different rates, for example releasing end-user client more often to provide new features more regularly.

4. Another aspect with which we must make a conscious effort is to ensure that documentation is updated as we go along and that this update is added to the definition of done when completing any relevant task. This is important for all stakeholders of the project, such that documentation is available to support integration and test activities, but also for (pre-)production deployment and user evaluation.

5. The introduction of Quattor controlled re-imaging of test machines had a significant positive impact on stabilising and improving the code quality, test coverage and reliability of our installation and configuration tools. As we move forward and support more operating systems, it will be critical that Quattor controlled machines are provided for all of these, such that we can extend our current testing strategy to these.

6. Our quick patching mechanism works, but must be integrated on a regular basis by the developers of the patched system to avoid escalation in patch maintenance effort. This is now working very well for OpenNebula, and we need to continue ensuring that patches found in the project are considered for integration directly in the original code base, and removed as patches.

7. Significant integration effort had to be spent to workaround OpenNebula releases that introduced backward incompatible changes, with respect to the way StratusLab uses OpenNebula. Using a higher level of abstraction would help mitigating changes in underlying dependencies.

8. Our policy of encouraging development on the master branch (or HEAD in other version control system) is paying off. The classic alternative is for each development to take place in separate branches and merging the branches when ready for integration or certification. This can cause headaches and conflicts which can be a significant source of risk and hidden delays. Since our sprints are relatively short and user stories specifically designed to be short and focused, it means that we can afford to develop directly on the master branch, where each commit is automatically built and tested by our Hudson system, including function, system and end-to-end tests.

9. A cousin to this technique of working on the master branch is to commit (and push) often. This requires discipline, where simple and small steps are made when developing, yet this is largely done by all teams, with few exceptions. This technique also has the advantage of being able to back track quickly when jobs turn red after committing faulty code and makes troubleshooting simpler.

10. As more features were integrated and the number of services and components grew in StratusLab, so did the number of jobs in Hudson, as well as the number of resources required to run these. Sometimes, several jobs remained red (broken) for several days, or were not executed for several days,

while team members were busy working on their sprint tasks. This meant that the work required to bring all jobs back to green was long, difficult and often boring. To address this issue, we decided to introduce a Lean technique called 'stop the line'. This meant that we agreed that as soon as a job failed, the person or team responsible for this job would stop their current activity and fix the problem. Within a single sprint, the Hudson server became a lot more stable, a red is now the exception instead of the rule, which was becoming too often the case. We need to ensure that this reflex is maintained, encouraged and enforced.

11. During the two-year project, we decided to switch our baseline operating system from CentOS 5.5 to Fedora 14, and then from Fedora 14 to CentOS 6.2, while also adding OpenSuSE and Fedora 16. This was a big move, requiring changes to both software and infrastructure. Our ability to deliver this transition within a single sprint is remarkable and probably shows good command of our code base and its dependencies on runtime environment and operating systems. This should make supporting more operating systems relatively straightforward, assuming that our build and test tooling can follow.

12. Continuing to support manual and automated installation capabilities is an important asset of the project, giving choices to system administrators willing to deploy StratusLab on their resources. The effort made to synchronise the behaviour of the two system is also paying-off such that both systems use the same assumptions resulting in almost identical setups.

The two-year project was very productive with several releases and numerous sprints delivering significant functionality towards the project goal of delivering a fully functional high quality cloud distribution. The build and test activities summarised in this report are also a great source of lessons learned, lessons we are convinced will help the project as it transitions to an open source consortium.

Glossary

Appliance	Virtual machine containing preconfigured software or services
Appliance Repository	Repository of existing appliances
CI	Continuous Integration
DCI	Distributed Computing Infrastructure
EGEE	Enabling Grids for E-sciencE
EGI	European Grid Infrastructure
EGI-TF	EGI Technical Forum
Front-End	OpenNebula server machine, which hosts the VM manager
GPFS	General Parallel File System by IBM
Hybrid Cloud	Cloud infrastructure that federates resources between organizations
IaaS	Infrastructure as a Service
Instance	see Virtual Machine / VM
iSGTW	International Science Grid This Week
Machine Image	Virtual machine file and metadata providing the source for Virtual Images or Instances
NFS	Network File System
NGI	National Grid Initiative
Node	Physical host on which VMs are instantiated
Public Cloud	Cloud infrastructure accessible to people outside of the provider's organization
Private Cloud	Cloud infrastructure accessible only to the provider's users
Regression	Features previously working which breaks in a new release of the software containing this feature
TDD	Test Driven Development
Virtual Machine / VM	Running and virtualized operating system
VM	Virtual Machine
VO	Virtual Organization
VOBOX	Grid element that permits VO-specific service to run at a resource center
Web Monitor	Web application providing basic monitoring of a single StratusLab installation
Worker Node	Grid node on which jobs are executed
XP	eXtreme Programming