MASTER'S THESIS | LUND UNIVERSITY 2015

Automated build service to facilitate Continuous Delivery

Ture Karlsson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2015-27

Automated build service to facilitate Continuous Delivery

Ture Karlsson
turekarlsson10@gmail.com

June 12, 2015

Master's thesis work carried out at IKEA IT AB.

Supervisors: Magnus Glantz, magnus.glantz2@ikea.com
Ulf Asklund, ulf.asklund@cs.lth.se

Examiner: Martin Höst, martin.host@cs.lth.se

Abstract

Continuous delivery can be seen as an evolution from agile software development methods and high demands to deliver working software quickly. It aims to always be able to deliver working and reliable software in short iterations by continuously integrate, build and test the software. This puts high demands on automation and the focus of this thesis is to automate the pipeline between source code and deployable software artifacts. The problem definition of this thesis is to improve and unify the deployment pipeline of software running on Linux at IKEA IT. The project resulted in a service that supports continuous delivery by providing automated building, testing, signing and deployment of software. It runs in production environment at IKEA IT and provides a high level of automation. It was evaluated with help from end users and the evaluation showed that the service is useful for the intended users and automate several steps they earlier have had to do manually.

Keywords: MSc, build service, continuous delivery, RPM Package Manager, DevOps

Acknowledgements

I would like to give a big thank you to Magnus Glantz at IKEA IT for all the inspiration and for giving me this opportunity. I would also like to thank Ulf Asklund and Martin Höst for their support and feedback on my work on this report. Also, the testers for their time and help in the evaluation.

This work was supported by Vinnova in the ITEA2 project 12018 SCALARE.

Contents

Chapter 1
Introduction

1.1 Background

Agile development methods are nowadays a common approach to software engineering. These methods promote many practices, e.g. close collaboration between programmers, short development iterations, early and continuous delivery. This puts high requirements on the development environments of big organizations. They need to be able to provide the developers with the correct tools to enable them to work in short iterations and deliver software easily. The release process profits from being automated in as many steps as possible since this makes the process faster and unified. It reduces the risk of failure in each release since it allows smaller changes between releases and also by abstracting the process away from human interaction.

1.2 Current conditions

In the network architecture of IKEA IT there are approximately 3,500 Linux servers distributed around the world on different central data centers, regional data centers and distributed sites. A typical example of a distributed site is typically a warehouse. To be able to administrate such a large amount of servers in a controlled way IKEA IT has developed a Linux Technical Standard that defines the configuration running on the servers. They use the operating system *Red Hat Enterprise Linux* (RHEL) together with *Red Hat Satellite* which is a system management platform to deploy and manage RHEL servers. The Satellite server contains information about the technical standard, all software packages needed and much more.

The servers run roughly approximated 200 different applications which are developed both by different vendors but also internally at IKEA IT. There is currently no standard on how these applications should be delivered which complicates the delivery process and

results in extra work for both developers and testers. The release process of an application may include many manual steps which heavily increases the release time. The release process differs much between applications depending to their characteristics. For example some of them has to go through multiple test environments, while others can be taken into production environment faster.

1.3 Problem definition

The problem definition of this thesis is to examine the possibilities to automate the process of building software, performing automated tests and deliver software to a provisioning service. The thesis work is carried out at the Linux department at IKEA IT and the goals of the thesis are to:

- explore different tools and techniques to automate the deployment pipeline of software that can be used to facilitate continuous delivery at IKEA IT.

- develop a service that works with their Linux platform and can be an effective tool in their every day work. The service should be able to package, perform automated tests, sign and deliver the package to a provisioning service. The goal is to unify how the applications are packaged and examine how much of the release process that can be automated. The product should be flexible so that it can be used for different types of applications.

- implement the service into the production environment at IKEA IT so that it is available for the intended users.

- evaluate the service in terms of usability and functionality to make sure that it meets the requirements above.

It focuses on the challenges and problems in automation of tasks that usually are performed manually, both from a command line, but also through GUI:s. Many command line tools require the user to accept different outputs, provide passwords etc. which is a problem that needs to be solved to be able to use these kind of tools without user interaction.

Another problem that will be treated in several steps is to figure out how one activity should trigger another in the best way. In this kind of systems it is important that one automated process can follow another automated process. This can be done in different ways, e.g. by sending some kind of signal or by letting the later process periodically check if the earlier has terminated. Both security and performance in context of time and robustness will be taken into consideration.

The service will also integrate with Red Hat Satellite 6.0 which is a new version of the Satellite platform. When this thesis work was done, Satellite 6.0 was just released and implemented at IKEA IT.

1.4 Related work

The idea of continuously deliver or deploy software has come from the increasing trend of agile development methods. One way to visualize the process and to break it down in stages is to see it as a *deployment pipeline*. This idea was first presented in an article by Humble, Read and North [17] (even though they called it deployment production line). The deployment pipeline as it is presented in [16] is shown in figure 1.1. It contains all the steps a software has to go through to be released, which can either be as packaged software or being deployed into production environment.

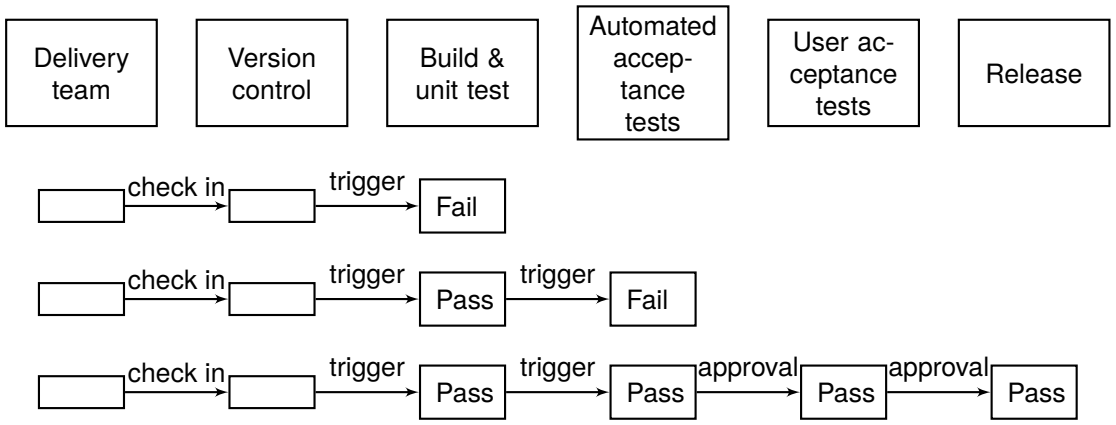

Figure 1.1: A deployment pipeline. A change to the software is checked into the Version Control System (VCS), which triggers a build and unit test. If it is successful the change continues through the different test stages of the pipeline before it is ready to be released. Each stage also provide feedback to the developers if it is successful or not.

The adoption of a deployment pipeline have multiple consequences. It guarantees that software is properly tested before release and the automation makes the process fast and reliable. The software can thereby be released more often which reduces the risk of each release. It does not imply that the release process is completely free from human interaction. For example the user acceptance test stage in figure 1.1 needs to be done manually to catch defects that has passed the automated tests. The goal is to automate as many steps as possible where manual interaction is unnecessary [16].

The definition of *Continuous Delivery* (CD) is to make bundled software available to be deployed by the push of a button. It is related to the term *Continuous Deployment* which differs from Continuous Delivery in an important way. The difference is that every build that passes automated tests are automatically deployed into production [16]. Both of the techniques benefit from the use of a deployment pipeline.

In a survey from 2015 about the state of Continuous Delivery, 50 % of the respondents said that they have implemented CD for some or all of their projects. But by sorting out the respondents that not did fulfill three key technical characteristics of CD only 18 % perform complete CD. In the same survey from the year before it was 8 % which shows how much growth there is in the area of CD [10].

The technical and social challenges a company faces when applying Continuous Deployment were examined by Claps, Berntsson Svensson and Aurum in [9]. The main technical challenges discovered were the following:

- Provide the proper hardware and software infrastructure. This will be the main focus of this thesis.
- Adopting the practice of developing in small batches.
- Seamless upgrades.
- Adopting the Continuous Integration process.

The four main social challenges discovered were the following:

- Pressure on the developers to always have code ready to be deployed.
- Process documentation.
- Lack of motivation in adopting the new process.
- Shorten customer feedback loops.

In [24], a "stairway" from Traditional Development to an Experimental System is presented containing the following staircases: A - Traditional Development, B - Agile R&D Organization, C - Continuous Integration, D - Continuous Deployment, E - R&D as an Experiment System. The goals of this thesis can be seen as providing hardware and software to enable IKEA IT to move from staircase B to staircase D (but with delivery instead of deployment) for the applications running on Linux.

The development process of four different companies are studied in [24] and the following key barriers in moving towards Continuous Deployment are identified in the case study:

- Transition towards agile development.
- To introduce Continuous Integration requires an automated test suite and a main branch where code is continually committed to.
- To introduce Continuous Deployment it is required that the whole organization are fully involved.

In one of the companies there were also noted many tools take long time to learn which can be a barrier.

1.5 Approach

To find the solutions to the problems the different steps of the deployment process will be treated step by step. The answers to what tools and techniques that is appropriate will be found in an exploratory fashion by studying them both in theory and practice. Also if and how they are currently used at IKEA IT will be taken into consideration.

The results found will be the basis for the solution to the problem which is the development of the software. The different parts of the software will be treated separately which means that each step will be explored separately and then implemented. Their relationships are shown in figure 1.2 below. The following steps will be performed:

- The functionality of a Continuous Integration (CI) server will be studied and set up with support for different Version Control Systems (VCS).
- The process of compiling and packaging software at IKEA IT will be studied, automated and integrated with the CI server.
- The possibilities for supporting automated tests by installing the software on a test server will be studied and implemented.
- Functionality for signing software packages that has passed testing will be implemented.
- 2 different ways of integrating with Red Hat Satellite will be evaluated. One of them will be implemented so that packages can be uploaded to the Satellite server.

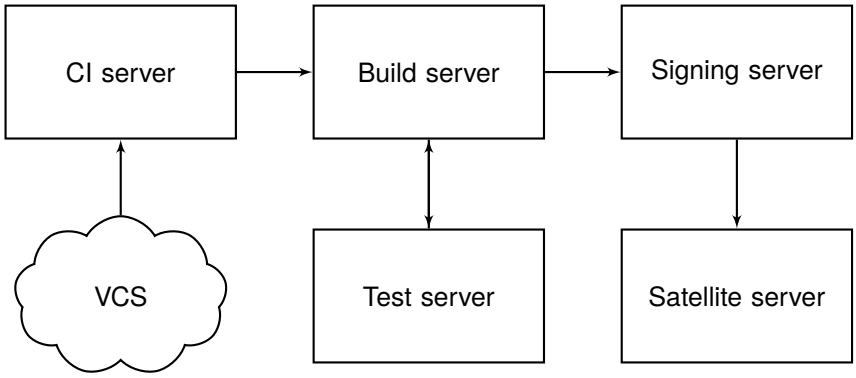

Figure 1.2: The idea of how the different components of the solution may be integrated.

To make the build service flexible for developers it should be possible to either use a VCS controlled by IKEA IT or be able to use a VCS of your own choosing. This means that the VCS entity in figure 1.2 and the division between what is IKEA and what is the vendor depends on what setup is preferred by the vendor.

The general problem solving method of the thesis will be *action research* [18] which contains three parts: *observations*, *solution* and *evaluation*. In the observations part the current situation at IKEA IT is observed and compared with practices found in literature. This is done to identify the problem that needs to be solved. In the solution part a solution is proposed and also implemented according to the different steps listed above. The last part of the action research methodology is to evaluate the result. This is done by letting future user test the solution and analyze the result. The evaluation part result in possible future improvements that can be applied to the solution. In the evaluation part the project will also be applied to a model developed in the SCALARE project where the different drivers, abilities and transformations made in the project are visualized on a canvas.

Therefore, the following three chapters which form the result part of this thesis are named Observations, Solution and Evaluation.

Chapter 2
Observations

This chapter gives an overview of the different practices of *Continuous Integration* (CI) and package management found in literature and compared to how they are currently used at IKEA IT. CI is a prerequisite for CD [16] and is a natural part of a deployment pipeline. IKEA IT had set up a rudimentary CI server in the test environment before this thesis was started, which was used as a starting point. The pipeline also needs a unified way of packaging the software. It was decided to use RPM [11], since it is the package management system developed by Red Hat. RPM also provides an easy way to handle dependencies between packages which is very useful. It is also the goal of IKEA IT to package all Linux software as RPMs.

This chapter contains both the underlying theoretical foundations together with how it is currently realized at IKEA IT. The problem(s) observed are presented in the end of the chapter.

2.1 Continuous Integration

The key purpose of CI is to make bundled software artifacts available for testing environments at any time [25]. The goal is to integrate the different parts of the software to make sure that they work together. The advantage of doing this is that it reduces the risk of ending up in a situation where integration of different parts of a software becomes hard because it has not been done in a long time. If the code is integrated and tested often and feedback is provided to the developers, bugs can be found quickly.

2.1.1 Version Control Systems

To use a Version Control System (VCS) is essential to CI. A VCS enables multiple developers to collaborate on the same project. It also allows you to keep multiple versions of your files so that you are always able to access old versions of files after someone has

modified them [16]. The files are kept in a *repository* and to modify them the user makes a local copy of that repository. After the modifications are done to the local files they are pushed to the repository to make the changes visible for the other users. This is known as a *check-in* or *commit*.

There are a number of different popular implementations, e.g. Subversion (SVN) [1], Git [4] and Concurrent Versions System (CVS) [3]. The Linux department at IKEA IT has set up an SVN server that contains many projects that run on Linux. It will be used later on in this thesis for putting up test projects etc. The projects in this repository contains files that are compiled and ready to be installed.

2.1.2 CI Server

A CI server is used to manage the integration and can be configured to run different scripts triggered by events specified by the user. A CI server is typically configured to run scripts that compile, package and test a software project, also known as a *build* of the software [25]. In the CI server each project is setup with the URL of the VCS repository so that the server can access the source code when it is going to build the software. The build can be triggered by different events, such as periodically or when a commit has been made to the repository. The best CI practice is to make a new build of the software at every commit [25].

IKEA IT has tested using Jenkins CI Server [7] which is an open source implementation. It basically supports the functionality described above. It is possible to add a new project and provide a URL to its repository, add build scripts and configure what should trigger a build. Support for different VCS can easily be installed as plug-ins. The Jenkins server keeps track of all builds and provides feedback on their status and results through a web interface. Each build contains a log file with the outputs from the build scripts. This can be used to get feedback on why a build has failed and what files the build has produced. Each build gets its own number to be able to separate them [7]. The start page of Jenkins is shown in figure 2.1 below.

2.2 RPM Package Manager

RPM Package Manager is a packaging software created for use in RHEL and the goal of IKEA IT is to use RPM to package all software running on Linux. It is used to build `.rpm` files that contain software that is ready to be installed. The biggest benefit of using a packaging tool like RPM is that you have full control over the files that are installed by the package. Consider the installation of a normal software where many files are installed in different locations of the operating system. If no package management system is used you have no control over the files that are distributed, which makes it very hard to upgrade or uninstall the software. But with a package manager you have control over what files are installed. If a software package needs updating or uninstalling, RPM has control over what files that should be effected. Therefore, when targeting a single operating system, it is considered best practice to use the package manager of the operating system [16].

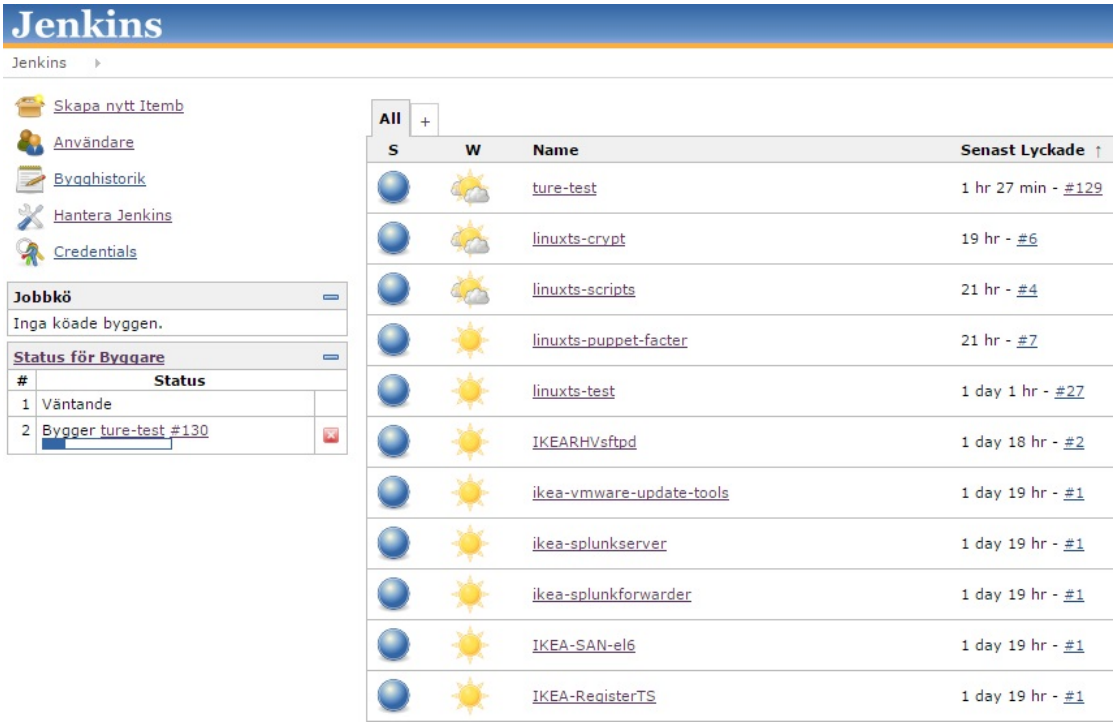

Figure 2.1: The dashboard of Jenkins CI server in the web interface. To the left is a settings panel and below that one can see ongoing builds. The list to the right contains the projects and gives a good overview of their condition.

2.2.1 The `.spec` files

To be able to pack an RPM a `.spec` file needs to be written. It contains all information necessary to pack, install and uninstall the package. A project named `ture-test` will be used as an example. The spec file begins with a section of package information in a straight forward manner (all of the most common information tags are listed in appendix A):

```
Packager: Ture Karlsson <ture@ikea.com>
Name: ture-test
Version: 1.0.1
...
```

After the initial package information there are a couple of sections starting with a `%` character. Except for the first one, they contain shell scripts that is run in the different steps of the installation process:

`%description` This section allows for a longer description of the package. There is an information tag `Summary:` that is used to give a one line description, but this section can be of arbitrary length.

`%prep` This section specifies the commands that need to be run before the build, e.g. extract the software.

`%build` This section specifies what build script(s) that should be run if the software needs to be compiled. This section can for example contain a call to e.g. `make`. This section will not be treated much in this thesis, since all the software that the SVN server at IKEA IT currently contain are compiled before commited.

`%install` This section contains the commands needed to install the software.

`%clean` This section specifies what clean-up that needs to be done after the packaging is done, e.g. deleting temporary files.

`%post` In this section configuration can be done. It can be used to start some service or add users etc., which could be very useful for automation purposes.

`%files` This section lists all the files that are packed in the RPM and will be installed. It can also specify their attributes.

`%changelog` In this section the changes of the file are documented.

RPM allows use of macros and variables in spec files. This can make them harder to read at first, but provides a lot of flexibility. E.g. the `Release:` field in the package information should increase with one every time the software is built. This is because every RPM should be unique and be able to be connected to a certain build. Instead of modifying the spec file before every build, the field can be defined as

```
Release: %{?BUILD_NUMBER}
```

The build number can then be provided as a parameter at the command when the build is started instead [11].

A full example of a spec file for the project `ture-test` is given in appendix A. This file can be used as a template for packages that only installs files and scripts that does not need compilation or any specific privileges. The most commonly used information tags are also listed.

2.2.2 Packaging

An RPM package (i.e. an `.rpm` file) is built with the `rpmbuild` command. To manage the building process six directories are created:

`BUILD/` In this directory the configuration scripts are run and the software is built. The build command changes working directory to this directory when the scripts in `%prep` and `%build` are run.

`BUILDROOT/` The content of this directory is a copy of what files and directories that should actually be installed.

`RPMS/` This directory contains the results of the builds, RPMs.

`SOURCES/` This directory contains the source files to the RPMs. Prior to the packaging the files that should be contained in the RPM is put here.

`SPECS/` This directory contains the spec files described above. The spec file is given as a parameter to the `rpmbuild` command to select the correct package.

`SRPMS/` This directory contains specific source RPMs. These contain the source files of the corresponding binary RPMS in the `RPMS/` folder [11].

Before the build command can be run there are a few things that needs to be put in place:

- `SPECS/` must contain a spec file with the same name as the project we want to build.
- The source files specified in the spec file must be put in `SOURCES/`. They can be placed there directly, but the best practice is to create a tarball (`.tar.gz`) that contains all the files and place it in `SOURCES/`.

We are now ready to build the RPM. The command for this is:

```
rpmbuild -ba SPECS/ture-test.spec
```

The first option means "build all" and can be altered to stop the build at a certain stage. Then the path of the spec file is provided which depends on your current working directory. If the build is successful an RPM is created in `RPMS/` and a source RPM is created in `SRPMS`. They are in the following convention:

```
ture-test-<version>-<release>.<architecture>.rpm
```

This means that the first build of this project would produce two files:

```
ture-test-1.0.0-1.noarch.rpm
ture-test-1.0.0-1.src.rpm
```

The RPM can then easily be installed with the following command:

```
rpm -I ture-test-1.0.0-1.noarch.rpm
```

All the installed files are as easily removed by uninstalling the package:

```
rpm -e ture-test
```

To be able to separate RPMs of the same version they include the version and build number in their file names. In this way an RPM can easily be connected with a certain build log from Jenkins. Both the spec file, the `rpmbuild` and the `rpm` commands have lots of options that are thoroughly described in [11].

2.3 Observed problem

The problem that the Linux administrators at IKEA IT face is that the delivery of applications are done in different ways depending on the application. It is desired that all applications should be delivered as RPMs, but many of the applications are not. This is because that the packaging procedure requires many manual steps and insight in how RPM works. As one can see in the previous section there are a couple of steps when building

RPMs that are easy to get wrong. Files need to be correctly written, placed or archived in the correct locations or the build will fail. It is also time consuming to do these steps by hand for every build that should be done. This results in that the software does not get built often, which means that every update results in a higher risk of error, since it contains bigger changes. This process should be automated so that every change to the source code starts an automated flow that builds, tests and eventually delivers the software.

Jenkins can take care of a couple of steps that otherwise would be manual, such as checking out the content of the repository and start the build process. The rudimentary Jenkins server IKEA IT has set up in their test environment can build RPMs. But the builds are started manually and the SVN server needs to contain a spec file for the package. The build script need to be more flexible and the software should be delivered to the correct location. It should also be possible to verify the correctness and integrity of the software packages which can be achieved by testing and digital signatures. This will result in a more secure, faster and unified delivery process and also take the responsibility of the packaging away from the developers.

Chapter 3
Solution

This chapter contains the description of the solution that was developed and begins with a short overview and figure.

A deployment pipeline will be implemented with focus on automation. The deployment pipeline used is slightly different from the one presented in [16] due the release processes used at IKEA IT. The pipeline is presented in figure 3.1 below. It is started by a check-in or commit from a developer. The commit should be noticed by a CI server that triggers the build. If the build is successful it will go into the testing phase where the build is installed on a test server and automated tests are performed. If the tests are successful the idea is that the content should be signed so that its integrity can be verified. The final step is to upload the software into the Satellite. The deployment pipeline guarantees that all software artifacts that ends up in the Satellite have gone through all these stages.

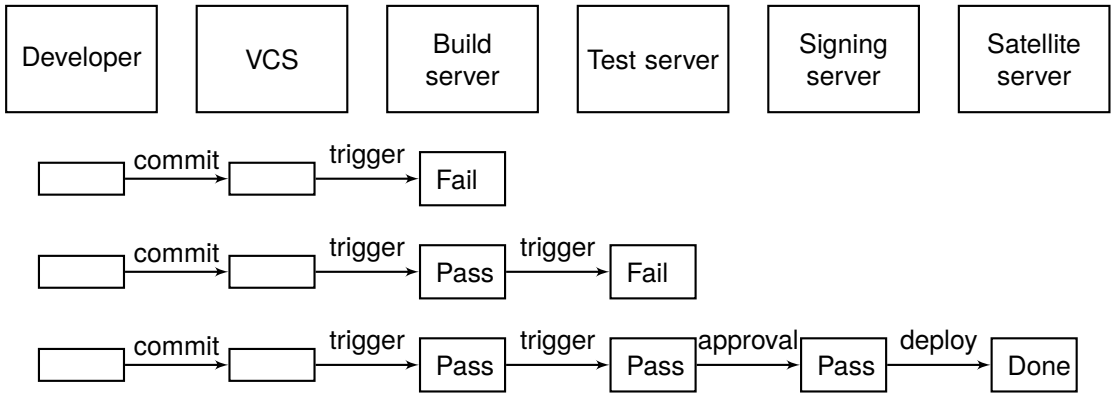

Figure 3.1: The deployment pipeline that will be used. If the build or the test phase fails the rest of the deployment is canceled and the developer gets feedback of what went wrong. If the build passes test the package is signed and deployed to the Satellite server.

The steps of the solution are presented in the different sections of this chapter. The first section covers the Jenkins server which will be used to provide CI. Jenkins was chosen because it is open source and IKEA IT has already tried using it, but there are many other alternatives, e.g. AnthillPro, Apache Continuum, Bamboo and CruiseControl [25].

3.1 CI Server

The Jenkins software was installed to a newly deployed virtual server. As mentioned earlier a development team will either be able to save manage their artifacts on the SVN server hosted by IKEA IT or provide a SVN or Git server of their own. Jenkins provides a useful web interface which makes CI trivial and it is easy to configure a new project. Basically, the user needs to provide the name of the project, what VCS that is used and the URL to the repository. Then provide build triggers that defines what should trigger a build [7]. It can be done periodically (e.g. once every day) but one practice of CI is to build at every commit [25].

A project called `ture-test` was created and committed to the SVN server and a new job was created in Jenkins to manage the build of the project. Jenkins is configured to poll the SVN with a certain interval to see if a new revision is available. This is accomplished with schedules similar to the ones used by Cron [2]. They contain 5 fields specifying MINUTE HOUR DAY-OF-MONTH MONTH DAY-OF-WEEK. A * represents all valid values. A couple of schema examples and their interpretations are given below.

* * * * *	every minute, every hour, every day of the month, every month, every day of week
*/5 * * * *	every fifth minute, every hour, every day of the month, every month, every day of the week
*/15 9-16 * * 1-5	every 15 minutes, between 9 pm and 16 pm, every day of the month, every month, mondays through fridays

Table 3.1: 3 different Cron schedules and their interpretations. Cron is often used in Linux environments to schedule scripts that should be run periodically.

A reasonable schema could be every fifth minute. But consider a Jenkins server running hundreds of projects. If every project is configured like this there would be a lot of polls every fifth minutes and very quiet in between. Therefore it should be changed to "H/5 * * * *". "H/5" means "some time every fifth minute" (the H is actually a hash of the project name and a random function).

Another approach that can be used is that instead of letting Jenkins poll the VCS periodically, the VCS can wake up Jenkins and tell it that a new revision is available. This approach will make the build start (marginally) faster, but it requires some kind of plugin to the VCS and differ between e.g. SVN and Git. The periodic approach will be used in this thesis, but the other approach with the correct configuration is possible as well.

The last thing that needs to be provided is what build scripts that should to be run. It can be called with some environment variables provided by Jenkins, e.g. `${JOB_NAME}` provides the name of the project and `${BUILD_NUMBER}` the number of the current

build. In the description of the spec file in section 2.2.1, the release number of an RPM was specified as this variable. In this way the release number of the RPM can automatically be specified as the build number of the project without user interaction [8].

As discussed previously, IKEA IT has an SVN server where developers check in compiled software that is ready to be built to RPM packages. This is a limitation that removes the responsibility of compiling code from the build server and means that a developers has to compile their code in their development environment before it is checked in to this SVN. Though, one can still get the build server to do the compiling if the compilation step is defined in the spec file of the project. But the problem is that if the build server should be able to compile all kinds of projects, it has to have all compilers and libraries installed which could potentially be very many.

3.2 Build script

This section describes the build script that was developed. It solves the problems with packaging of RPMs described in the observations chapter. The build script was written in Python instead of for example a Bash script. This decision was made due to the extensive support from its standard library, the easy and minimalistic syntax and the fact that it is an interpreted language which makes it suitable for scripting. Also it has an API to RPM called `rpm-python`.

A first version of the build script was developed to be able to build a project locally. It required that the source files and a spec file was provided in the correct directory structure. The `rpm-python` API mentioned above turned out to be useless since it only had functionality for querying RPM packages and printing information about them, but did not support the actual building of packages [11]. The functionality that was implemented was to pack the source files into a `.tar.gz` archive (a tarball) and then build the RPM with the `rpmbuild` command. The script worked as it was supposed to and placed a binary RPM in the `RPMS/` directory and a source RPM in `SRPMS/`. The script was named `rpmbuilder.py`

Even if the script worked as supposed it needed some more functionality and flexibility to be able to automatically build the projects from Jenkins:

1. Jenkins needed to be able to call it.
2. It needed to be able to use the source files that Jenkins checks out from VCS.
3. It needed to be able to give different options on how the `.spec` file should be provided.
4. It needed to make the resulting RPMs available for the next step in the deployment pipeline.

All commands made by Jenkins are performed by a user called `jenkins`. To make it able to call the command in an easy manner the script was placed in the `PATH` environment variable of the Jenkins user. This means that no path to the script needs to be provided to call it. Also the first line was changed to:

```
#!/usr/bin/python
```

This means that this script can be run by only specifying its name and not that it should be run as a python script. The script was also changed to accept two parameters; project name and build number. This means the script can be called from Jenkins in the same way for every project:

```
rpmbuilder.py ${JOB_NAME} ${BUILD_NUMBER}
```

Jenkins automatically provides the name of the project as `${JOB_NAME}` and the number of the current build as `${BUILD_NUMBER}`.

As a solution to the second item on the list above a couple of paths that the script uses needed to be specified. When Jenkins starts a new build it is configured to check out the content of the repository to a path:

```
.../jenkins/jobs/<project name>/workspace
```

The script was therefore modified to set up a separate directory for each project from where the `rpmbuild` command is called:

```
.../jenkins/jobs/<project name>/rpmbuild
```

which contains the directories explained in 2.2.2. Before the build is done the script copies the content of `workspace/` to `rpmbuild/SOURCES/` and creates a tarball of the content. In this way the build script will always use the latest checked out content of the repository.

As described earlier, every rpm build needs a spec file. These can contain a lot of instructions on what should be done before and after the installation of the RPM, which can be used to automate many steps that otherwise would have to be done manually. A different script called `rpmspec.py` was developed and called from the build script. Its responsibility is to provide a spec file in the `SPECS` directory. The script searches the files checked out in the workspace to see if it contains a spec file. If that is the case, it is copied to the `SPECS` directory of the project. If not, a rudimentary spec file is created from scratch (much like the one in appendix A). This gives the developers the choice to either provide a spec file or to let the build service generate one. If the spec file should contain custom instructions it is preferable to have a spec file in the repository that can be edited by the developers to fit their needs. But if a project just contains files that needs to be put in the correct directories, the automatically generated spec file is sufficient.

The last step of the build process was to make the RPMs available for the next step in the pipeline; testing. This was realized by copying them to the web root directory `/var/www/html/rpms`. If the server is running Apache HTTP Server the files in `/var/www/html` will be published as a file system on the web page of the server. Each project was provided with its own directory which means that the latest RPMs of a project will be available for from `servername.com/rpms/projectname/`. Figure 3.2 below shows how Apache presents the web root in a browser.

The Jenkins web interface displays every build of a project in a list. If a build has failed it is marked red which gives the user an easy overview of them. Each build contains a log with the output from the build scripts. The build script gives output of its process to give the user enough feedback to be able to debug the build if it fails.

Since the build script is run locally on the Jenkins server the approach sketch in figure 1.2 is not accurate. In the next section a new virtual machine will be deployed to handle

Figure 3.2: The standard way Apache presents files in the web root.

the testing. A more accurate figure of the solution with a VCS, the Jenkins server and a test server so far is shown below.

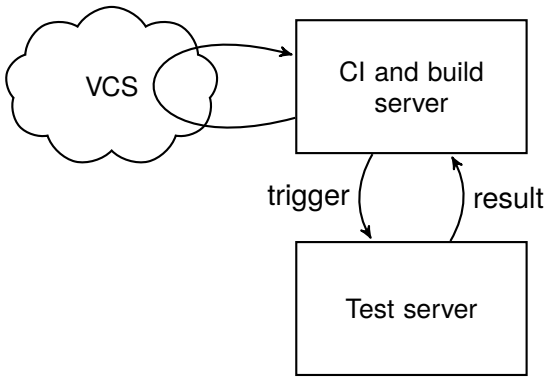

Figure 3.3: An overview of the solution so far with the test server introduced in the next section included.

3.3 Testing

The purpose of the test phase is to perform build verification tests [16] which test if the package is correctly built and possible to install and if its dependencies to other packages can be resolved. These tests are also known as smoke tests and are performed to give developers early feedback on the build and to avoid deploying bad builds into further test environments. The test phase was also integrated with a custom test suite called `linuxts-test` that is newly developed at IKEA IT. It tests functionality on the host server specified by the application in order for it to be able to run. It can test if certain services are running, certain RPMs are installed, if communication to remote servers are working, etc. IKEA IT has a goal to have more automated tests for the applications running

on Linux. The integration with `linuxts-test` provides an easy way for the automated tests to be run, which can influence developers to create more automated tests.

A new virtual server was deployed which was used as a test server during the development. This was done to separate the testing from the build process and to test the builds in a clean environment. Two different ways to start the testing was identified. Either to periodically check if there are any new builds that should be tested or to trigger the testing as soon as a build is complete. Both approaches were tried and the results are presented below.

3.3.1 Check for new builds periodically

A first approach to the testing was to monitor the web root of the build server and trigger a test every time a new RPM is published. A script was developed to parse the HTML page of the build server. As mentioned earlier each project was provided with its own directory and since Apache provides a field "last modified" to every file and directory that is put in the web root the script can see if any changes has been made since it was last run. Each directory of `servername.com/rpms/` are presented in the following way in the HTML source code:

```
...<a href="project1/">project1/</a>...11-Mar-2015 14:01...
...<a href="project2/">project2/</a>...09-Mar-2015 09:58...
```

where `project1` and `project2` are two directories followed by their corresponding date of last modification. The HTML page was parsed to figure out which projects had new RPMs since the last time the script was run and that should thereby be tested. And the RPMs could be downloaded from this page as well. To run the script periodically can be accomplished either by adding it as a cron job (which means that it will be called with a given period) or it could be programmed as a daemon (a service that always runs in the background on the system).

During the development some drawbacks with this method was found. The first one was that the parsing of the HTML file becomes very static and error prone, since it has to look for specific key words. Secondly, since the tests will be run in a periodic fashion, there will take longer time until the tests are performed than if they in some way were triggered by the build itself. And lastly, there is no trivial way to provide feedback to the developers on the testing. A corresponding way of communicating the feedback to the Jenkins server would have to be setup, which would result in even more time delay and a lot work for a small reward.

A new approach needed to be found that could trigger the test directly and be able to provide feedback in an easy manner.

3.3.2 Trigger test at every build

A better approach would be to trigger the test phase for a project every time the build script has successfully terminated. The RPMs would need to be transferred to the test server and the output from the tests should be returned to the build server. In that way both the feedback from the build script and from the test can be presented at the same place, namely in in the build log of the project in Jenkins.

This was accomplished by using the remote login program `ssh`, a secure copy program `scp` and RSA encryption. The commands that need to be run on the test server are called over `ssh` from the build server. To enable this, encryption keys were created for the `jenkins` user on the build server and a user called `rpmtest` was created on the test server. To avoid having to provide passwords at every remote call the public key of the `jenkins` user was saved as an authorized key for the `rpmtest` user. In this way the user `jenkins` can perform operations at the test server as `rpmtest` without providing its password. The syntax used to call commands over `ssh` is the following:

```
ssh remote-user@remote-host command
```

If no public key encryption scheme is in place like the one presented above, `ssh` will ask for the password of the remote user before the command can be executed.

This approach is better than the previous one in multiple ways. It makes the test procedure faster because it starts as soon as the build is done which can provide the developers with faster feedback on the build. Since the test script is called from Jenkins the feedback from the build script and the test can be presented in the same log. Because of the same reason, if the test fails, the build can be marked as failed, which would be harder to accomplish with the previous approach. This also makes the service more flexible in an other way. With just a small modification, it is possible to let the user provide a test server of its own choosing as a parameter to the build. In this way the service can be used to deploy the software into test environments where all kinds of tests can be performed.

3.3.3 The testing procedure

The following steps were implemented in the test script:

Copy The RPMs are copied to the test server. To copy files over `ssh` the program `scp` is used with the following syntax:

```
scp /local/file remote-user@remote-host:/remote/file
```

Install The RPM is installed on the test server with a program called `yum` instead of the `rpm` command. `yum` has got a couple of benefits and features when installing software that the `rpm` command lacks. For example it resolves dependencies between packages which is wanted in this context. If the RPM that is going to be tested has specified any dependencies in its spec file (the `Requires:` tag) those packages get installed as well.

`linuxts-test` If the RPM contains any test files for the `linuxts-test` suite they are run. The tests are specified with a simple syntax and can test if certain services are running by writing their names in a file `appname.services`, if something is listening to local ports by putting the port numbers in `appname.ports`, certain RPMs in `appname.rpms`, communication to remote servers in `appname.firewall` and check if resources are managed by Puppet in `appname.puppet` (not relevant at the moment). If these files are installed in `/linuxts-test/test.d/` the tests can be performed by calling `appCheck appname`.

Uninstall After the tests the RPM is uninstalled using `yum`. If all these steps are successful the test phase is complete.

If any of the steps fail the whole build gets marked as failed in Jenkins and the user may look in the build log to see in what stage the build that failed. An example of the output from the test phase is shown in figure 3.4 below.

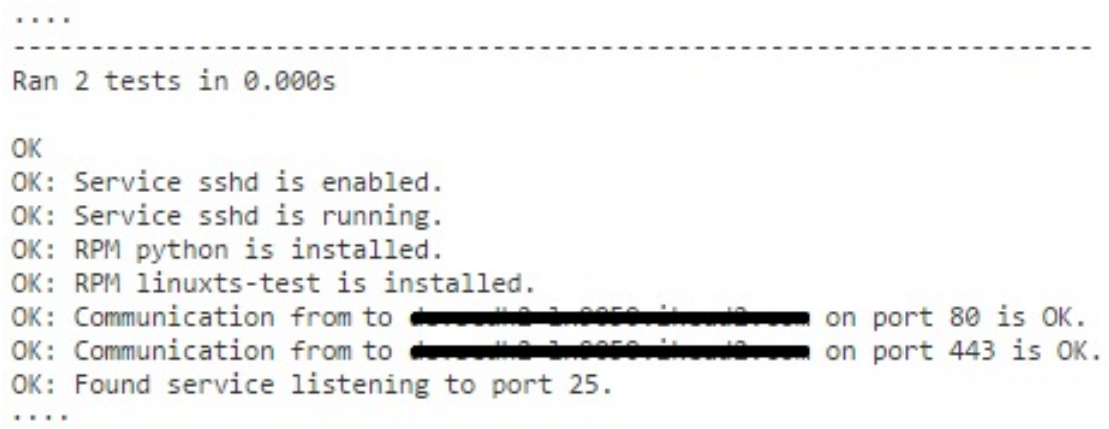

Figure 3.4: The output of the test phase for a build of a project called `tutorial-test`. The first two tests are Python unit tests and the rest are functional tests for the `linuxts-test` suite.

3.3.4 Improvements

At this point the test script worked in the following way: The newly built RPM was copied to a certain dedicated test server where it was installed, tested with `linuxts-test` and then uninstalled. This is not always a good solution for the developers and testers. A reasonable setting could be the following: An application should be built and then deployed at a test server where the testers of the application want to perform various kinds of tests. A previous build of the application is probably installed on the test server which means that when the new build is deployed it is an upgrade. When an upgrade has been made, it is possible to perform a `yum downgrade`. This is a feature in `yum` that makes it possible to roll back to a previously installed build. After the upgrade, the `linuxts-test` tests should be executed. If any of the tests fails, the software should be downgraded to the previously working build. If it succeeds subsequent tests can be performed to determine if the build meets the requirements for production environments.

The test script was altered due to these insights. It was modified to accept another parameter that specifies the name of the test server. This enables the developers to specify a test server of their own where the tests should be performed. It was also modified to not uninstall the application after the tests. Instead, when a new build is going to be tested, the last stable build is upgraded to the new build. If the tests then fail, the application is downgraded to the previously stable build. Only builds that pass the automated tests are passing the test phase and go on to the next stage in the deployment pipeline.

3.4 Signing

This section describes how and why the RPMs should contain digial signatures [14] before they are deployed intot the Satellite server. Another server was set up with the single purpose of signing the RPMs and uploading them to the Satellite server. An updated overview of the whole solution is shown in figure 3.5 below.

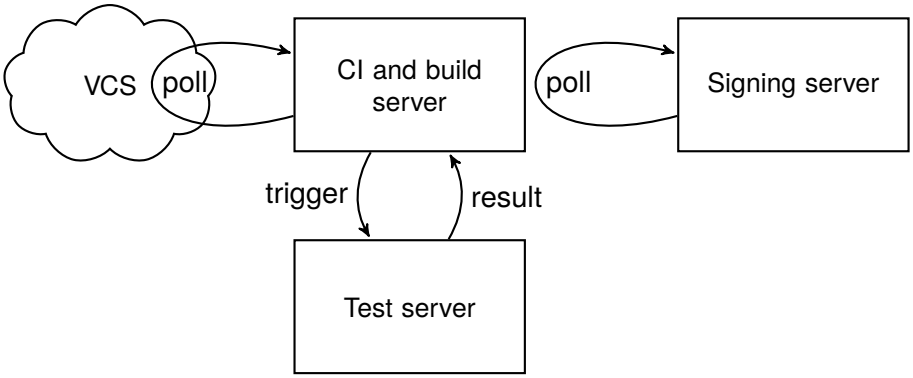

Figure 3.5: An overview of the solution so far, which now includes another server that fetches and signs RPMs that have passed testing.

The motivation behind the signing is to provide a way to verify the integrity and origin of the RPMs. If an RPM is signed by the signing server it is possible to verify that the RPM has been built by this service, has passed the automated testing phase and has not been altered. RPM has built in support to sign RPMs with GNU Privacy Guard (GPG) which is an open source implementation of Pretty Good Protection (PGP). In other words, GPG implements PGP. Confusing as these abbreviations are, the only command that needs to be used for the signing is `gpg` [11].

3.4.1 Signing to verify test

The first thing that was done in the implementation of the signing process was to make Jenkins sign every RPM that has passed the testing phase. In this way the signing server can later verify that the RPM has been built by Jenkins and that it has passed the testing. The file structure of the web root of the build server was changed so that each project directory contains one directory with the latest built RPMs and one with the latest tested and signed. A GPG key pair was generated for the Jenkins user. This is done with the command `gpg - -gen-key`. An empty passphrase was used to avoid having to store a passphrase in the code. A couple of macros needed to be configured so that the RPM command knows what key to use. This is done in a file called `.rpmmacros` in the home directory of the current user. After that an RPM can be signed with the command `rpm -addsign filename.rpm`.

A time consuming setback occurred in the implementation of the signing. The command to sign the RPMs needed to be provided with the empty passphrase as a response, which seemed to be impossible with the Python standard library. The problem was that when RPM asks for the passphrase of the GPG key, it uses a function called `getpass`

[5]. `getpass` is built to make it difficult to provide passwords in any other way than with human interaction at the command prompt. After looking at different possibilities for a workaround an approach with an Expect [23] script was tried. Expect scripts are used to automate command line programs that require interaction with the user and provides a trivial syntax for writing scripts that are based on an expected behavior. The following short Expect script that takes an RPM file as an argument solved the signing problem:

```
#!/usr/bin/expect -f

spawn rpm --addsign {*}$argv
expect -exact "Enter pass phrase: "
send -- "\r" # Just "\r" because an empty passphrase is used.
expect eof
```

Instead of making the Python script call `rpm -addsign`, it calls this Expect script which performs the signing. A call to the signing script was added in the end of the test script to sign RPMs that has passed testing with the key of the Jenkins user.

3.4.2 Signing server

The purpose of the signing is to make it possible to verify that the RPMs has been built by the service and thereby passed through the pipeline. IKEA IT has a key that is used to sign packages that are going to be used in production. This makes the security around this key very delicate and the signing server has to store this key to be able to automatically sign RPMs that has passed testing. Just as with the test script, a decision had to be made to either trigger the signing process by the previous stage of the pipeline or to periodically check if any packages from the previous stage has been made available. As figure 3.5 above reveals, the signing server was developed to periodically check if the build server stores any RPMs that has not yet been signed. At the test stage the time factor is of bigger concern, but at this stage the security is more important. This approach means that it is possible to make it much harder to access the signing server by turning off `ssh`. Another difference from the test stage is that the signing server does not need to provide any feedback directly to each build in Jenkins. If a build has passed the test phase it will eventually be signed by the signing server. In other words, a build is considered successful after the test phase and will be signed and deployed into the provisioning service.

This functionality was implemented in a similiar fashion to the first approach to the testing at section 3.3.1. The benefits in form of security are superior to the drawbacks in form of time and parsing html, since the html page always looks the same. A signing script was written that parses the html page to see if any projects has been changed since last time the script was run. If so, it checks if there are any new RPMs in the directory named `tested/`. New RPMs are first queried to see if they are signed by Jenkins and thereby are really built and tested by Jenkins. If so, they are signed by the production key that certifies that the package is ready for production.

3.5 Upload RPMs to Satellite server

The last stage of the pipeline will be to upload RPMs to a repository at a Red Hat Satellite 6 server. The pipeline will provide Continuous Delivery when this functionality is in place. It will not provide Continuous Deployment, since it does not automatically install new packages into production. This limitation to the service emerged from the scope of the thesis but also from IKEA IT. Continuous Deployment is often a goal, but it is not suitable for all companies [16]. At IKEA IT there is currently no goal to reach Continuous Deployment. Continuous Delivery provides the necessary level of automation but persist the manual control of when to deploy a new version of a software package, which is needed due to change management etc.

Red Hat Satellite is a life-cycle management platform for RHEL. It provides administrators tools to manage large systems and is used for deployment of physical and virtual servers, subscription and repository management etc [22]. The latest major release, Red Hat Satellite 6, was released during the fall of 2014. It was implemented at IKEA IT during the time of this thesis.

The Satellite server stores all RPMs in repositories that are used by the servers. This is where the RPMs should be deployed. The Satellite server at IKEA IT has different repositories for RPMs created for different major releases of RHEL. This forced a little adjustment to the pipeline to make it possible to specify what major release an RPM should be built for. It was needed because an RPM should be tested at the same major release of RHEL as it is built for.

Satellite 6 has a web UI that enables the user to control it and use all its functionality. It also provides two other ways of communication, namely a Representional State Transfer (REST) API and a CLI tool called Hammer. Both of these can be used to perform the task of deploying the RPMs to the correct repositories and are examined separately below.

3.5.1 RESTful API

REST [12] is an architectural style that is implemented in interfaces to many web services. A RESTful API is an API that follows the principals of REST. The main constraints put on these API:s are that they often use existing technology such as HTTP for communication. They are also using a client-server architecture to separate the responsibility of e.g. user interfaces and data storage between clients and servers. The most important constraint is probably that the communication should be *stateless*. This means that each request from a client to a server should contain all information that the server needs to perform the action.

The API to Satellite 6 follow these constraints. A client can send HTTP requests such as `GET`, `POST`, `DELETE` etc. to the server in a stateless manner. As described in this guide [20] it is possible to write Python scripts that connects to the Satellite API using JSON [6].

3.5.2 Hammer CLI

The second approach to the deployment is to use Hammer CLI which is a command line interface that also can be used for scripting most of the different functionalites of the web

UI [21]. It was also used in the transition from Satellite 5 to Satellite 6 at IKEA IT.

After seeking advice from the Linux team at IKEA IT and also a Red Hat consultant it was decided to use Hammer rather than the RESTful API. This decision was made because the task of deploying an RPM can be done with a single command in Hammer given the correct parameters. Hammer CLI can also handle bigger files without having to split them into chunks. Also, the Linux team at IKEA IT are used to Hammer from the transition. If the build service would have needed to send many different kinds of requests to the Satellite server, the API would possibly be a better approach. But Hammer was better suited for this single task.

The request to the Satellite server using Hammer was implemented on the signing server. The request to the Satellite is basically to upload the specified content to the specified repository. It needs to specify the following parameters:

- URL of the server
- Username and password for the user performing the upload
- Organization name
- Product name
- Repository name
- Path of the file to upload

The product name is the technical standard for Linux at IKEA IT and it contains 3 different repositories for the 3 latest major releases of RHEL (5, 6 and 7). Because of this, it was realized that it has to be possible for the end user to specify what major release a build is intended for. Otherwise it will not be possible for the service to know which repository the RPM should be uploaded to. It was therefore decided that the major release needs to be included in the release number of each RPM so that the full name of an RPM with version number 1.0, release number 4 that is built for RHEL 7 (`el7`) gets the following name:

```
package-name-1.0-4.el7.noarch.rpm
```

The major release can therefore be provided in the call to the build script like this

```
rpmbuilder.py ${JOB_NAME} ${BUILD_NUMBER}.el7
```

The Hammer command called has the structure showed below.

```
hammer -s <server address> -u <username> -p <password>
repository upload-content
--organization <default organization>
--product <product name>
--name <repository name>
--path <path to the rpm>
```

It was implemented in the signing script so that after an RPM is signed with the production key it is uploaded to the correct repository with this single command. The pipeline is now complete and a figure of it is shown in figure 3.6 below.

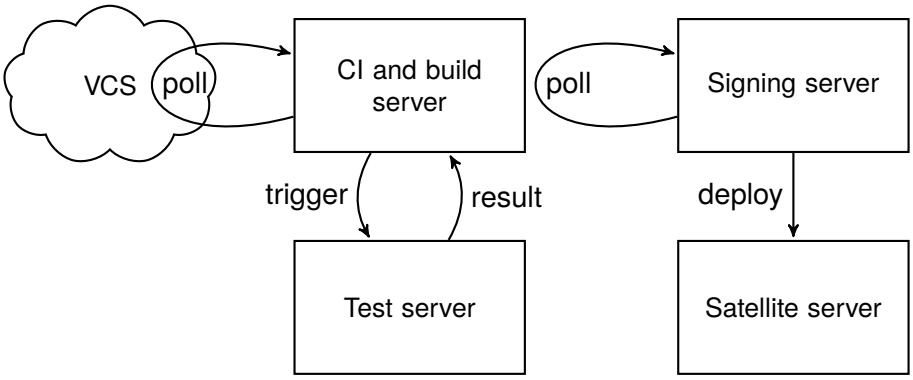

Figure 3.6: An overview of the complete solution with the last step implemented, which was to deploy signed RPMs to Red Hat Satellite 6.

3.6 Deliver the product

This section describes the process of making the different parts of the service into packages that contains everything that is needed to install the service on new servers. The service was developed and test run in test environment, but had to be correctly packaged in order to be able to be installed in production environment. Documentation for installation and test that follow the guidelines at IKEA IT were written simultaneously. Since the product is a build service, it was a suitable first test to let it build it self.

3.6.1 Packaging

To prepare for the delivery and allow for future changes, all the components needed to be put under version control. This was done in the SVN server and modifications to the source files were from now on checked in there. All the source code was checked in, which means that it is possible to continue the development. The complete product will consist of 3 different packages:

`linuxtp-buildservice` Contains everything that needs to be deployed to the build server. This includes the following:

- The scripts `rpmbuilder.py`, `rpmspec.py`, `rpmtest.py`, `rpmsigner.exp`
- Test cases for the `linuxts-test` suite
- A Jenkins template job with the standard settings that can be copied when creating new jobs
- An initiation script Jenkins
- An `index.html` page that is used to direct users between the Jenkins page and the web root page where the RPMs that the service has built can be found.
- A spec file that specifies what other packages that is required to run the service, what files it installs and a post installation script that enables services and settings that are vital for the build service (like Jenkins and Apache). This spec

file is shown in appendix A.3 and is a good example of a little more complicated spec file than the one discussed in 2.2.1.

`linuxtp-buildservice-testintegration` Contains what needs to be configured on a server that will perform the automated tests. It sets up a user that will be used to perform the commands at the test machine. It also installs the public key of the Jenkins user so that the build server can control the test server without any password.

`linuxtp-buildservice-signer` Contains the scripts needed to perform the signing and deployment to Satellite 6, namely `rpmsigner.py`, `rpmsigner.exp` and `rpmsigner-lastcheck`. It also contains a Cron tab (see table 3.1) for the root user which contains:

```
* * * * * /usr/bin/rpmsigner.py
```

This means that the script `rpmsigner.py` will be run by the root user once every minute if this package is installed.

The latest version of Jenkins was downloaded and added to the Satellite. This version of Jenkins will be installed when `linuxtp-buildservice` is installed, since it has `Requires: jenkins` in its spec file. The 3 projects were set up as jobs and built by the version of the service running in the development environment.

3.6.2 Testing

The packages were built and tested on a test server. During the testing some complications were found. It was mainly settings made to other services on the build server, such as `ssh` and `httpd`. These settings were automated as much as possible by adding them to the spec file of the package, but some settings will have to be done by hand after installation. They were documented in a document describing installation and configuration that has to be written for every product at IKEA IT.

The `linuxtp-buildservice` RPM contains test cases for the `linuxts-test` suite to test some core functionalities on the server it gets installed on:

```
linuxtp-buildservice.ports:
   443
   80
linuxtp-buildservice.rpms:
   java-1.6.0-openjdk
   jenkins
   python
   expect
   rpm-build
linuxtp-buildservice.services:
   jenkins
   httpd
```

The last time the RPM was built, installed on a test server and tested before it was deployed into production all the test cases passed with the following output:

```
OK: Service jenkins is enabled.
OK: Service jenkins is running.
OK: Service httpd is enabled.
OK: Service httpd is running.
OK: RPM java-1.6.0-openjdk is installed.
OK: RPM jenkins is installed.
OK: RPM python is installed.
OK: RPM expect is installed.
OK: RPM rpm-build is installed.
OK: Found service listening to port 443.
OK: Found service listening to port 80.
```

3.6.3 Installation

The build service were now ready to be installed in the production environment. The RPM for the build server was installed which caused the installation of Jenkins and the other dependencies. Jenkins was up and running within a minute and contained the template project which was packaged in the RPM.

IKEA IT has hard requirements on security and has to be PCI-DSS compliant (Payment Card Industry Data Security Standard). PCI-DSS compliance put high demands on who should be able to access and modify systems, as well as on logging etc. Also, anyone that should have access to the build service should not be able to take control over the server running the build service. It was realized that this is a risk that is exposed if anyone would be able to add or change configuration of a job. This is because the configuration page gives the user the ability to execute arbitrary commands in the build script section which could have serious consequences. This forced some security hardening of the service. The first thing that was put in place was user login. IKEA IT has a server that provides LDAP which can be used with a Jenkins plugin to provide username and password authentication [13]. This plugin was set up and configured. The next step was to decide who should have what privileges on the Jenkins server. Administrators were given the authority to edit all kinds of settings. Then it was decided that other users will be divided into two groups: power users that has access to the configure page of projects and normal users that only can start builds and see the output from them. This decision was made to ensure that the service does not give any users more privileges to the server than they had before. Users that will be able to execute arbitrary commands at the server through Jenkins would be able to do it by other means as well.

Another Jenkins plugin called Audit Trail [15] was also installed to provide logging of every change to Jenkins settings and project configurations etc. This is to ensure that every change is traceable.

The service was now ready to be used. The projects that make up the build service were added and built successfully. Also the `linuxts-test` project was added.

Three different video tutorials were recorded to be able to spread information about how the service is working. One shows how to set up a project on the SVN server and

add `linuxts-test` cases. The next showed how a new project is created in Jenkins and how to set up the build and test scripts with the correct parameters. The last tutorial shows how builds can be started either manually or by commits and also how to read the output of the build. The tutorials were made available on the intranet of IKEA IT so that everyone that should use the service will have access to them.

After the last modifications the standard settings for starting and performing builds are shown in figure 3.7 below.

Figure 3.7: This figure shows how Build Triggers and Build Scripts can be specified for a project in Jenkins. Jenkins will poll the repository once every minute to see if there is any commits the last minute. If it is the RPMs will be build with the `rpmbuilder.py` script where the build number will be increased automatically. Then tested with the `rpmtest.py` script when the test server is specified.

The complete process for an application can now in words be summarized like this: The development team sets up a project on a VCS server. The files that should be packaged are commited as well as automated test cases for the `linuxts-test` suite. A spec file can be added as well if it needs to contain custom configuration that is not included in the automatically generated (as the one in appendix A). A new project is then created in Jenkins and set up similiar to figure 3.7 and connected to the VCS server. A new build is then started either manually or by a new commit. The RPMs are built and tested. If the tests are successful the RPMs are signed by the signing server and are then uploaded to the correct repository at the Satellite server. Then, if potential manual tests are successful, a new Content View [19] is created in the Satellite server that includes the new RPMs. The RPMs can then be installed to every server that has `yum` connected to that repository by calling e.g.

```
yum install project-name
```

Chapter 4
Evaluation

The solution needs to be evaluated as the last step of the action research methodology. This was done with help from various end users of the service. Three persons were contacted. One that had tried using the test Jenkins server described in section 2.3 and one that normally packages, tests and signs RPMs manually. The third tester is one of the Linux administrators at IKEA IT that among other things works much with the `linuxts-test` suite. They were asked to watch the tutorials and then configure and build a couple of their projects with assistance. They were then asked a couple of questions to see if the service fulfilled the goals set up in the problem definition in section 1.3. The questions asked were the following:

1. How do you find the service in context of usability? Is it easy to learn and easy to use?
2. How do you find the service in context of functionality? Is it robust and working as expected?
3. How does it effect the way you package and deliver software? Has it solved any problem you have had?
4. Have you noticed anything that could be improved?
5. Have you noticed any features that you think the service is lacking?

4.1 Test person 1

This tester is responsible for development of multiple software packages containing NetScaler Load Balancing and NetScaler Support Servers. It is a team of 5 developers but the person testing the service is currently responsible for the packaging. This is because of the complexity according to the tester. These packages have lately been built with the test Jenkins

server described in section 2.3, but without any automated testing etc. The other developers send their changes to the tester who then commits the changes to the SVN server. The builds are then started manually after version number and release number have been updated manually in the spec file.

To show how the new build service works, the tester was guided through the steps of setting up and configuring new projects. The process of starting builds and how the testing phase works were then also described. All actions were performed by the tester, but with guidance. The first impression was that the service felt familiar since the tester had used Jenkins before, even if it was an older version. The tester had not used `linuxts-test` before but quickly realized how it could effectively be used together with the build service. The part where a new Content View needs to be created was considered to become an obstacle, but after explaining that otherwise untested RPMs could be automatically exposed to servers in production, the tester realized that it was reasonable.

The tester were asked to answer the questions above after the testing and demonstration and the answers are presented below.

1. "The service is very usable at a top level, however there are many features and much flexibility that is not obvious and therefore not accessible to an average user.

   I would like to see a system like Jenkins be more 'wizard' driven, inviting users to learn the options through description offered at the point where the feature is an option."

2. "The service is 'fit for purpose' and indeed, yes, robust!"

3. "During my time working with Application Delivery I have always worked closely with Magnus Glantz and followed his guidelines to build and delivery application to the Glinux supported platform. Therefore this newer version of the platform does not solve any problems. The main issue that my department, Application Delivery, faces is the speed of (or rather lack of) delivery of the packages to the Servers via the Satellite mechanism.

   Now we package applications and then manually/directly transfer the same to the target server and install with YUM by reference to the package file local to the disk. This has proved effective however individual file transfer can be time consuming across the 14 servers I manage."

4. "I would prefer that once a package is "passed" by my department, it is made available for general release to my servers. Simply no one other than my immediate team can claim whether the packages are "right" or "wrong" anyway..."

5. "I would like to be able to identify "files that contain local variables".... then when the package is installed, YUM identifies if the files exists, if it does it does not overwrite. If it is not there, then a default is installed.

   I am sure that this feature probably exists in YUM and/or Jenkins. I am also sure that I could script the same however I am sure this is a common requirement for the types of packages that I am working with..."

The tester set up all the projects of his team after the demonstration and they are now building all their projects in the build service.

4.2 Test person 2

The second tester is responsible for delivering and configuring IBM WebSphere middleware platform at IKEA IT. Every component of this platform is currently built, deployed and tested by hand which can be automated by the build service. The source files are in version control at the SVN server, but are structured as in section 2.2.2 so that they can be checked out and built manually. One of the packages were chosen to start with and the files were moved around so that it worked for the build service. The tester had spec files that needed some modification before they worked with the build service. The RPMs were now built successfully and the test script could also be run since the tester had test servers available. The RPMs were successfully installed on the test server. The test script resolved the dependencies to other packages successfully, but no tests were run since none existed. The tester explained that some manual configuration is currently needed before the platform is running and tests can be performed. It was also discussed what should happen with the build number when the version number is increased. As the build service is working currently, the build number increases for every build regardless of version number, but the tester would like to be able to reset the build number to 1 when the version number is increased.

This tester gave the following written answers:

1. "After a walkthrough and hands on its easy to use. Starting without this I will think it's hard to know what to do and how to get started. (Note that I have not read any user guide or similar)"

2. "Hard to give an answer right now. Will have to use it for a while to be able to have a correct opinion."

3. "Same as above."

4. "Build_number automation should be cleared when going for higher release number."

5. "Same as above."

4.3 Test person 3

The last tester is one of the Linux administrators at IKEA IT who are usually building RPMs by hand or lately by the test Jenkins server described in 2.3. This tester is responsible for the packaging and testing the `linuxts-test` suite that has been used throughout this thesis, but also some other packages. The tester found the service useful, especially the testing phase, since the `linuxts-test` suite needs to work on all different major releases and all different versions of the Linux Technical Standard at IKEA IT. But during this testing it was realized that it would be a very useful feature to be able to test an RPM on several test servers at each build. With the current functionality, the tester needs to change the settings of the project in Jenkins for every version the tests should be run on. The answers to the questions are presented below.

1. "I find the service simple to use, especially since I've used Jenkins before."
2. "My experience with the service is that it is stable and works as expected."
3. "It has simplified the testing of RPM:s that I've built. Previously I had to download, install and test the built RPM manually."
4. "Not that I can think of right now."
5. "It would be useful to be able to test the RPM on several servers every build, since you sometimes want to make sure an RPM works on several different platforms."

4.4 Summary

The evaluation of the service confirmed that the service fulfills the goals that were set up in the problem definition and also resulted in a couple of useful ideas for how the service could be further improved:

- The service could provide a way to make the delivery of applications from the Satellite server to the servers in production environment easier.
- Some people may want to be able to reset the build number for an application when the version number is increased.
- It would be useful to be able to build and test an application for different major releases of RHEL and versions of the Linux Technical Standard at them same time.

The evaluation also gave the testers a thorough introduction and hands-on experience with the service.

4.5 SCALARE canvas

As a second part of the evaluation the whole problem and solution treated in this thesis were applied to a model developed by the SCALARE research project presented in figure 4.1 below.

The different drivers of the project are identified and placed in the top of the canvas. Then the current abilities that were present before the project was started are placed to the left. These are dependent on the "as is" organization, processes and products. In the middle the transformations are presented which are the solutions to the problems. To the right the different advantages of doing the transformations are presented. The different entities are linked together with the arrows in different colours.

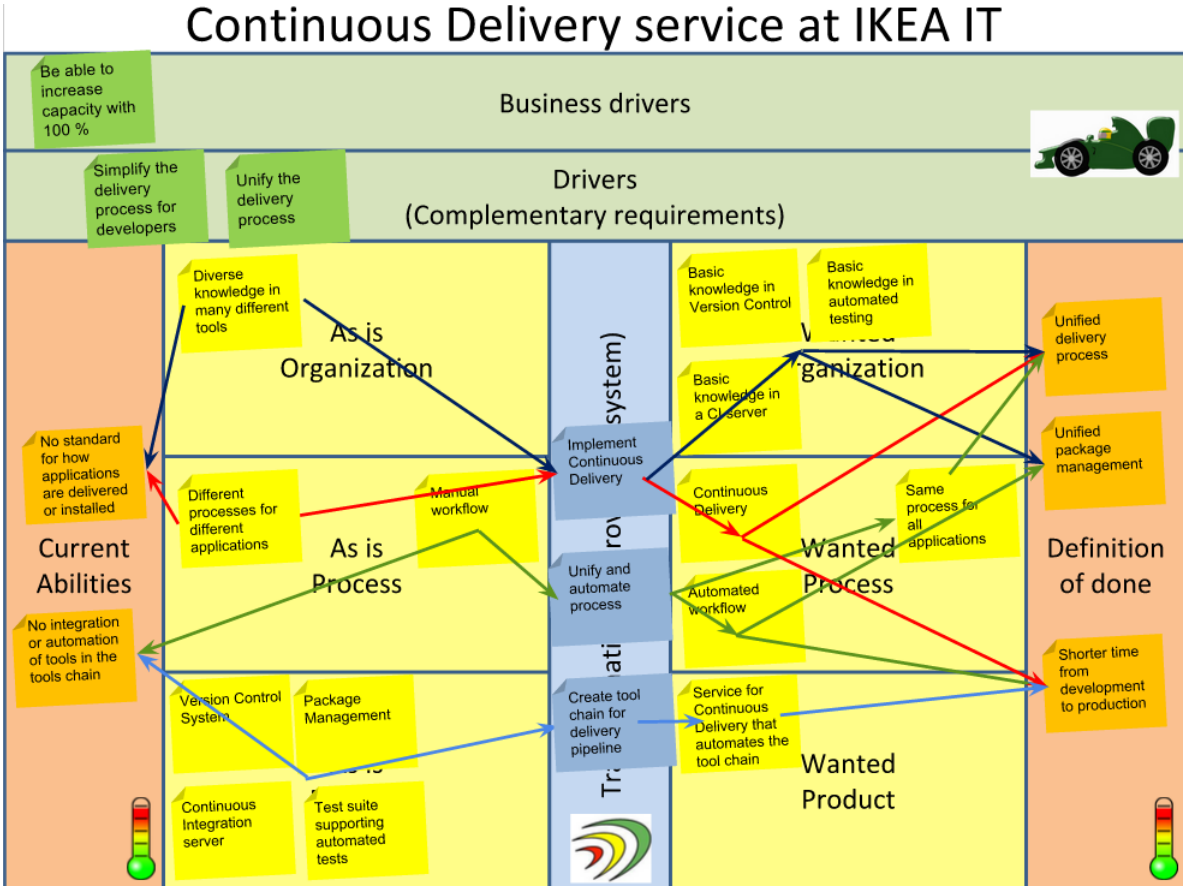

Figure 4.1: The project applied to the canvas developed in the SCALARE project.

Chapter 5
Discussion

5.1 Continuous Delivery

Continuous Delivery has shown to provide benefits for many companies and there are many companies that currently move in that direction, as the white paper from CloudBees [10] show. Except for the technical challenges that has been treated in this thesis, company culture and other processes may need to be changed as well. Especially if the goal is to reach Continuous Deployment. The scope of this thesis was limited to make a service that upload RPMs to the repositories at the Satellite server. Then, to expose an RPM to production servers a new Content View needs to be created in the Satellite server. A Content View can be seen as a filter that decides what versions of different softwares that should be exposed to the servers. This becomes an obstacle in the complete delivery pipeline of an application but it is necessary. It should not be possible for all applications to go into production without going through more than automated tests.

A deployment pipeline turned out to be a good approach. It provided an intuitive abstraction and made it easy to divide the different stages. Another thing that is clear now is that the different stages of the development of the pipeline naturally resulted in 3 different RPMs:

`linuxtp-buildservice` This RPM contains the Continuous Integration and build server.

`linuxtp-buildservice-testintegration` This RPM corresponds to the test phase and is installed on every test machine.

`linuxtp-buildservice-signer` This RPM contains the signing and uploading functionality.

Another aspect of Continuous Delivery is how to be able to deliver applications that run on multiple platforms. As for IKEA IT the biggest applications run across multiple

platforms. As an example there can be applications that run databases on AIX, back-end on Linux and front-end on Windows. This kinds of setup makes the problem much more complex, but a build service like this one is essential if Continuous Delivery for these kinds of applications should be possible.

5.2 The solution

The development of the service was successful and according to the evaluation it fulfilled the goals that was set up. It was a good decision to build the service around a Jenkins server, since it provides a user interface which would take long time to develop. Jenkins is easy to modify in many different ways and since it is open source there is a lot of useful plugins available. All code that was written during this thesis was in Python except for the 5 lines long Expect script in 3.4.1. It was a real pleasure to learn and work with Python and it turned out to fit well for the task. Other alternatives would be Ruby or Perl, but I do not think it would have made any significant difference.

As one of the persons who helped out with the evaluation pointed out, it could have been a good idea to be able to reset the build number so that it starts at 1 for every new version of the software. The build service does not support this, but it can however be solved by a little trick. If a copy of the project is made in Jenkins and the old one is deleted, the build number will be reset to 1. It is not a very good solution, but it can be done in just a couple of mouse clicks. But this removes the ability to trace an RPM to a certain build in Jenkins since it removes all the history of that project from Jenkins. The build number should therefore be unique.

The testing part integrated well with the `linuxt-test` suite. This became very powerful, since it is possible to append all kinds of automated tests in your project and they will be run at every build. One issue with the testing part was to make it possible at arbitrary servers running different version of the Linux standard. But the idea is to make the automation work for all configurations inside the scope, so this problem was solved with the `linuxtp-buildservice-testintegration` RPM.

The last step of the development was the script that signs and uploads the RPMs to the Satellite server. There was some problem with the signing, as described in the solution chapter. I wanted to only use Python code and not involve the Expect script, but it was not possible. I suspected that more tools than the Python standard library would be needed, but I tried to keep it at a minimum. The uploading of the RPMs to the Satellite was done with a command to Hammer which was easy to use because most of the surrounding functionality was already in place.

The service as a whole provides a high level of automation. By using it, it is easier to work in short iterations because it is not time consuming to integrate and test a project often. One thing that has not been treated by this thesis is the compilation of code. This is because the files checked in to the SVN server at IKEA IT are already compiled or do not need compiling. This means that the files that need compiling are compiled locally by the developers, which is not optimal according to [25]. It is possible to make Jenkins do the compilation by specifying the compilation steps as the `%build` script in the spec file of a project. But the drawback with this is that the server running the build service would need to have all compilers and libraries etc installed. This does not scale well if the number of

projects increase. As for projects containing e.g. Python code like the ones developed in this thesis or PHP like the one in the NetScaler projects described in the evaluation chapter, no compilation is needed anyway.

Chapter 6
Conclusion

The overall questions that was asked in this thesis was how to best automate the delivery of software running on Linux at IKEA IT. It boiled down to a delivery pipeline that provides continuous integration, automated building, testing, signing of RPMs and delivery to a central repository. It fulfills the goals set up in section 1.3 and achieved the level of automation that was desired. The problem with realizing the flow through the pipeline was solved in different ways depending on the stages. The test stage is triggered by the build because it should provide as fast feedback as possible. But the signing was realized with a periodic approach since it provides a higher level of security. The complete deployment pipeline allows developers to work in short iterations by automating many steps that developers earlier were forced to do manually.

Another more general conclusion that can be drawn from this thesis is that these kinds of processes are closely connected to automation and benefits from being automated in as many steps as possible. Even if automation takes some work to achieve, it can save much time for a lot of people, which in the end profits the whole organization.

6.1 Future Work

One way to improve the service would be to develop a plugin to Jenkins that supports the same functionality. In that way it would be easy to make changes to the configure page of Jenkins. For example it would be possible to manage a spec file for the project directly in Jenkins instead of on the SVN server. It would be possible to add new features and connect them to the GUI in Jenkins to make it much easier to handle user settings. It would for example be fairly easy to include a feature that enables the user to reset the build number when the version number is increased.

There are some possible ways to increase the speed of builds. One way of scaling is to explore different ways of speeding up Jenkins. It is possible to modify how many "builders" Jenkins use, which is basically the number of possible concurrent builds. This

is set to 1 from the start but was increased to 10 when the service were installed in production. This was not examined very scientifically, but the limit of how many builders that is suitable is ultimately set by the hardware of the host server. If the build service would need to scale up even more, it is possible to connect build slaves to Jenkins. Build slaves are machines that Jenkins control and distribute the work load on.

One new feature that would probably be the first one to be implemented if there was time would be to make the build service able to perform parallel tests on different servers. Many of the projects at IKEA IT need to be tested for several major releases of RHEL and also different versions of the Linux Technical Standard. A simplified model of this is shown in figure 6.1 below where a single package is built and then tested on 3 different RHEL servers and then uploaded into their corresponding repository in the Satellite server.

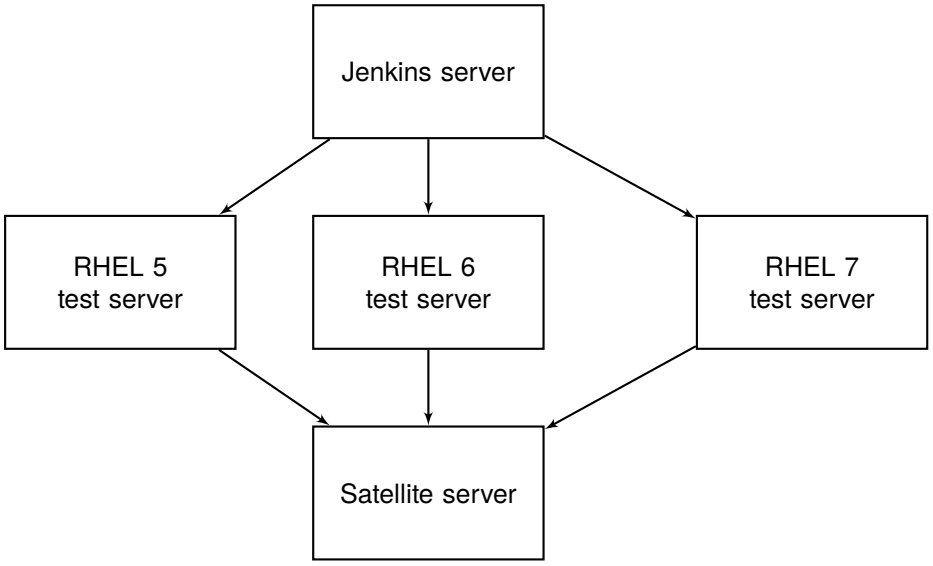

Figure 6.1: A simple figure of how it would be possible to simultaneously test at different platforms. If the tests pass, the RPMs are uploaded into the corresponding repository at the Satellite server for each of the test machines.

Bibliography

[1] Apache subversion. https://subversion.apache.org/, accessed 2015-05-13.

[2] cron(8) - linux man page. http://linux.die.net/man/8/cron, accessed 2015-05-13.

[3] Cvs - concurrent versions system. http://www.nongnu.org/cvs/, accessed 2015-05-13.

[4] Git version control system. http://git-scm.com/, accessed 2015-05-13.

[5] The gnu c library: getpass. http://www.gnu.org/software/libc/manual/html_node/getpass.html, accessed 2015-05-13.

[6] Introducing json (javascript object notation). http://json.org/, accessed 2015-05-13.

[7] Jenkins ci server. http://jenkins-ci.org/, accessed 2015-05-13.

[8] Jenkins wiki - use jenkins. https://wiki.jenkins-ci.org/display/JENKINS/Use+Jenkins, accessed 2015-05-13.

[9] Gerry Gerard Claps, Richard Berntsson Svensson, and Aybüke Aurum. On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software Technology*, 57:21–31, 2014.

[10] CloudBees DZone. Research partner spotlight. *The Guide To Continuous Delivery, vol II*, 2015.

[11] Eric Foster-Johnsson. *Red Hat®RPM Guide*. Wiley Publishing, 2003.

[12] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Doctorial dissertation. University of California, 2000.

[13] Jesse Glick. Jenkins ldap plugin. https://wiki.jenkins-ci.org/display/JENKINS/LDAP+Plugin, accessed 2015-05-13.

[14] Dieter Gollmann. *Computer Security 3rd Edition*. John Wiley & Sons, 2011.

[15] Alan Harder. Jenkins audit trail plugin. https://wiki.jenkins-ci.org/display/JENKINS/Audit+Trail+Plugin, accessed 2015-05-13.

[16] Jez Humble and David Farley. *Continuous Delivery: reliable software through build, test, and deployment automation*. Pearson Education, 2011.

[17] Jez Humble, Chris Read, and Dan North. The deployment production line. *In Proceedings of the conference on AGILE 2006*, pages 113–118, 2006.

[18] Martin Höst, Björn Regnell, and Per Runesson. *Att genomföra examensarbete*. Studentlitteratur AB, Lund, Sweden, 2006 (in Swedish).

[19] Red Hat Inc. Red hat satellite 6 user guide - content views. https://access.redhat.com/documentation/en-US/Red_Hat_Satellite/6.0/html/User_Guide/chap-Using_Content_Views.html, accessed 2015-05-13.

[20] Red Hat Inc. Red hat satellite 6.0 api guide. https://access.redhat.com/documentation/en-US/Red_Hat_Satellite/6.0/pdf/API_Guide/Red_Hat_Satellite-6.0-API_Guide-en-US.pdf, accessed 2015-05-13.

[21] Red Hat Inc. Red hat satellite 6.0 transition guide. https://access.redhat.com/documentation/en-US/Red_Hat_Satellite/6.0/pdf/Transition_Guide/Red_Hat_Satellite-6.0-Transition_Guide-en-US.pdf, accessed 2015-05-13.

[22] Red Hat Inc. Red hat satellite 6.0 user guide. https://access.redhat.com/documentation/en-US/Red_Hat_Satellite/6.0/html/User_Guide/index.html, accessed 2015-05-13.

[23] Don Libes. Expect - home page. http://expect.sourceforge.net/, accessed 2015-05-13.

[24] Helena Holmström Olsson, Hiva Alahyari, and Jan Bosch. Climbing the "stairway to heaven" – a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. *In proceedings of the 38th Euromicro Conference on Software Engineering & Advanced Applications*, pages 392–399, 2012.

[25] Paul M. Duvall and Steve Matyas and Andrew Glover. *Continuous Integration: improving software quality and reducing risk*. Pearson Education, 2007.

Appendices

Appendix A

RPM

A.1 exampleproject.spec

```
Packager: Ture Karlsson <ture@email.com>
Summary: This is a test project
Name: ture-test
Version: 1.0.0
Release: %{build_number}
Group: Applications/System
BuildArch: noarch
BuildRoot: %{_tmppath}/%{name}-%{version}-root
Source0: %{name}.tar.gz
License: Commercial

%description
This section contains a longer description of the project
and can be written on multiple lines.

%prep
tar -xvzf %_topdir/SOURCES/%{name}.tar.gz

%install
rm -rf $RPM_BUILD_ROOT
mkdir %{buildroot}
cd %{name}
cp -Rp * %{buildroot}/

%clean
```

```
rm -rf $RPM_BUILD_ROOT

%post

%files
%defattr(-,root,root)
/usr/local/bin/rpmqa.py
/usr/local/bin/info.txt

%changelog
* Fri Feb 27 2015 Ture Karlsson
- Changed version number and removed file list
* Thu Feb 26 2015 Automatic Packager
- File created.
```

A.2 Package information fields

```
Packager:
Summary:
Name:
Version:
Release:
Epoch:
Group:
BuildArch:
BuildRoot:
Source0: can be followed by Source1, Source2, ...
Patch0: can be followed by Patch1, Patch2, ...
License:
Group:
Distribution:
Vendor:
URL:
ExcludeArch:
ExclusiveArch:
Excludeos:
Exclusiveos:
Requires:
Provides:
Obsoletes:
Conflicts:
PreReq:
BuildRequires:
BuildConflicts:
BuildPreReq:
```

A.3 linuxtp-buildservice.spec

```
Packager: Ture Karlsson <tukar3@ikea.com>
Summary: Build service for applications running at Linux
Name: linuxtp-buildservice
Version: 1.2.0
Release: %{build_number}
Group: Applications/System
BuildArch: noarch
BuildRoot: %{_tmppath}/%{name}-%{version}-root
Source0: %{name}.tar.gz
License: Commercial
Requires: java-1.6.0-openjdk, jenkins, python, expect,
rpm-build, httpd

%description
Build service to package project to RPMs, smoke test them
and sign if they are OK.
It is running through Jenkins CI server.
Create a new project in Jenkins and add the following as
build script:

# Calls the build script to build the RPMs.
# Example: rpmbuilder.py <job name> <build number>
(optional other parameters)
rpmbuilder.py ${JOB_NAME} ${BUILD_NUMBER}

# Calls the test script to test the last successfully
built RPM.
# Example rpmtest.py <job name> (optional <remote host>)
rpmtest.py ${JOB_NAME}

%prep
tar -xvzf %_topdir/SOURCES/%{name}.tar.gz

%install
rm -rf $RPM_BUILD_ROOT
mkdir %{buildroot}
cd %{name}
cp -Rp * %{buildroot}/

%clean
rm -rf $RPM_BUILD_ROOT

%post
mv /etc/rc.d/init.d/jenkins.rpmsave /etc/rc.d/init.d/jenkins
```

```
setcap cap_net_bind_service+ep /usr/lib/jvm/jre-1.6.0-
openjdk.x86_64/bin/java
rm /etc/yum.repos.d/jenkins.repo
passwd -l jenkins
service jenkins start
chkconfig jenkins on
service httpd start
chkconfig httpd on
sed -i 's/#   StrictHostKeyChecking
ask/StrictHostKeyChecking no/g' /etc/ssh/ssh_config

%files
%defattr(-,root,root)
%dir %attr(755, jenkins, jenkins) /var/www/html/rpms
/etc/rc.d/init.d/jenkins.rpmsave
/usr/bin/rpmbuilder.py
/usr/bin/rpmsigner.exp
/usr/bin/rpmtest.py
/usr/bin/rpmspec.py
/linuxts-test/test.d/linuxtp-buildservice.ports
/linuxts-test/test.d/linuxtp-buildservice.rpms
/linuxts-test/test.d/linuxtp-buildservice.services
%dir %attr(755, jenkins, jenkins) /var/lib/jenkins/
jobs/template/
%attr(755, jenkins, jenkins) /var/lib/jenkins/
jobs/template/config.xml
%dir %attr(755, jenkins, jenkins) /var/lib/jenkins/
jobs/template/builds/
%attr(755, jenkins, jenkins) /var/lib/jenkins/
jobs/template/builds/lastSuccessfulBuild
%attr(644, jenkins, jenkins) /var/lib/jenkins/
jobs/template/builds/lastFailedBuild
%attr(755, jenkins, jenkins) /var/www/html/index.html
%attr(644, jenkins, jenkins) /var/www/html/trofast.jpg
%attr(644, jenkins, jenkins) /var/www/html/krister.jpg
%attr(755, jenkins, jenkins) /var/lib/jenkins/jobs/
template/builds/lastStableBuild
%attr(755, jenkins, jenkins) /var/lib/jenkins/jobs/
template/builds/lastUnstableBuild
%attr(755, jenkins, jenkins) /var/lib/jenkins/jobs/
template/builds/lastUnsuccessfulBuild
%attr(755, jenkins, jenkins) /var/lib/jenkins/jobs/
template/scm-polling.log
%attr(755, jenkins, jenkins) /var/lib/jenkins/jobs/
template/nextBuildNumber
%attr(755, jenkins, jenkins) /var/lib/jenkins/jobs/
```

```
template/config.xml
%attr(755, jenkins, jenkins) /var/lib/jenkins/jobs/
template/builds/legacyIds
%attr(755, jenkins, jenkins) /var/lib/jenkins/jobs/
template/subversion.credentials

%changelog
* Wed Apr 15 2015 Ture Karlsson <tukar3@ikea.com>
- Removed signing scripts and enabled services.
* Tue Apr 14 2015 Ture Karlsson <tukar3@ikea.com>
- Changed description.
* Wed Apr 01 2015 Ture Karlsson <tukar3@ikea.com>
- Added some more information and dependencies.
* Wed Apr 01 2015 Automatic Packager
- File created.
```

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2015-06-04

EXAMENSARBETE Automated build service to facilitate Continuous Delivery
STUDENT Ture Karlsson
HANDLEDARE Ulf Asklund (LTH), Magnus Glantz (IKEA IT)
EXAMINATOR Martin Höst (LTH)

Ny tjänst gör att utvecklare kan leverera uppdateringar kontinuerligt

POPULÄRVETENSKAPLIG SAMMANFATTNING **Ture Karlsson**

IKEA IT har många underleverantörer av programvaror som kör på deras Linux-plattform. För att få alla att leverera pålitliga uppdateringar på ett enhetligt sätt och dessutom tillåta dem att arbeta i korta iterationer har en tjänst utvecklats under detta examensarbete som tillhandahåller *Continuous Delivery*.

Ett problem som Linuxadministratörerna på IKEA IT ställts inför är att det inte finns någon standard för hur applikationerna som kör på deras servrar ska levereras. Detta innebär att personerna som är ansvariga för att paketera och leverera applikationerna gör detta manuellt och i olika format, vilket är tidskrävande och innebär en stor risk för misstag. Det gör även att uppdateringar görs sällan och innehåller större förändringar.

För att lösa detta problem har en tjänst utvecklats som paketerar applikationerna på ett enhetligt sätt i det format som passar bäst för operativsystemet. Så fort förändringar görs i applikationens källkod byggs ett nytt programvarupaket automatiskt utan mänsklig inblandning. Paketet installeras i en testmiljö där automatiserade tester utförs på applikationen. Om testerna är lyckade signeras paketet varefter det laddas upp i ett centralt lagringsutrymme. På så sätt garanteras både paketets korrekthet och integritet och det kan sedan installeras på de servrar i produktionsmiljö där de ska användas.

Dessa stegen har automatiserats, vilket gör att en ny förändring i programvaran startar ett flöde som paketerar, testar och levererar programvaran automatiskt. Detta medför att utvecklare kan arbeta i korta iterationer och kontinuerligt få återkoppling på sitt arbete genom att programvaran kontinuerligt testas för att upptäcka brister som senare kunnat orsaka problem i produktionsmiljö. Tekniken kallas *Continuous Delivery* och innebär att man alltid har senaste testade programvarupaket redo att sättas in i produktion. Tjänsten bidrar med följande fördelar för IKEA IT:

- Enhetlighet: Alla applikationer kan paketeras och sedan även installeras på samma sätt.
- Automation: Tjänsten har automatiserat många steg som annars behövt utföras manuellt, vilket alltid medför en risk för misstag.
- Kortare iterationer: Tjänsten medför att uppdateringar kan ske med tätare intervaller, vilket gör att mindre uppdateringar kan införas i produktion i stället för stora uppdateringar som medför en högre risk för fel.

Personer som är ansvariga för leverans av applikationer kan nu lägga till sitt projekt i den nya tjänsten. Så fort förändringar görs i källkoden kommer ett nytt flöde sparkas igång varefter alla de automatiska stegen utförs. Detta gör att utvecklarna inte behöver paketera, testa, signera och ladda upp sina applikationer manuellt, vilket sparar dem mycket tid.

Tjänsten kör nu i produktion hos IKEA IT. För att sprida kunskapen om den och för att få människor att börja använda den har användarinstruktioner spelats in i videoformat samt har några av slutanvändarna testat och låtits utvärdera tjänstens funktionalitet. Målet är att alla applikationer som körs på Linux hos IKEA IT ska använda sig av tjänsten.

Även ett annat stort företag har visat intresse för tjänsten och i framtiden planeras att tjänsten ska göras om till ett öppen källkodsprojekt för att underlätta att fler kan använda den.