HAAGA-HELIA
University of Applied Sciences

Developing IT Infrastructure: Automated and Centralized System Configuration Management with Puppet

Armen Igitian

Bachelor's Thesis

Business Information Technology

December 2013

HAAGA-HELIA
University of Applied Sciences

Abstract

19.12.2013

Business Information Technology

Author Armen Igitian	**Group** BITE
Title of report Developing IT Infrastructure – Automated and Centralized System Configuration Management with Puppet	**Number of report pages and attachment pages** 33 + 22
Supervisor Juhani Merilinna	

An ability to precisely configure computer systems is needed in any IT infrastructure to ensure that each system serves its intended purpose. To centrally control configuration of a group of systems in an automated manner is invaluable.

The purpose of this thesis project was to enhance the development of an IT infrastructure by using a system configuration management framework called Puppet. This thesis project was assigned by Conformiq Software Oy.

This thesis contains definitions and underlying principles of automated system configuration management, an introduction to the Puppet framework, a set of objectives with their problematics, reasoning and implementation as well as a description of chosen research methods and working methodology.

The thesis indicated that Puppet is a powerful tool. It is used to keep diverse systems in certain configuration states and to orchestrate changes whenever necessary. Such general usage of Puppet for achieving concrete results in systems' configuration was one integral part of this project.

Powerful tools alone are just tools until they are included in a meaningful, safe and easy-to-use workflow. Hence, another crucial aspect of this project was establishing and implementing an advanced workflow to better support the process of developing an infrastructure with Puppet.

By the end of this thesis project, the configuration of the company's 20+ servers was entirely managed by Puppet. Moreover, the further development of the infrastructure with Puppet is now supported by the customized workflow and necessary documentation.

This report is primarily targeted at students and professionals in IT administration. However, software developers, managers and anyone else interested in the concept of system administration and automated system configuration management might also find it useful.

Keywords
system configuration, automation, system administration, puppet

Table of contents

1 Introduction

The role of system configuration management in an IT organization is difficult to overestimate. All companies relying on IT need to ensure that their computer systems are configured and their users are supported by these systems as expected. The extent of precision of systems' configuration depends on how much an organization relies on IT.

Conformiq Software Oy - the sponsor of this thesis project, is a small-sized software house in Espoo. It strongly relies on its IT infrastructure in general and particularly on a group of servers configured to support its everyday operations, which consist of developing, testing, benchmarking, and releasing its software products. The required precision of systems' configuration is high, especially for performing meaningful benchmarks and building releases since servers performing these purposes need to be identical in their state of configuration.

The value of automation and centralization of system configuration management lies in ability 1) to quickly deploy and redeploy diverse systems with precisely defined configurations, 2) to ensure that systems stay in the defined state of configuration as long as it is needed, 3) and to quickly and easily propagate any further changes required in their configuration.

Already before this project Conformiq Software had enjoyed some of the benefits of automated and centralized system configuration management. Conformiq's server farm had been deployed and controlled by one of the most powerful frameworks for system configuration management called Puppet. However, some features were missing.

Some of the required features referred to configuration of the existing Conformiq's Puppet farm, which included installation and configuration of new services as well as an improved calibration of some of the existing ones. Other features were targeted at establishing an infrastructure that would support a safe, smooth and sensible workflow for further development of the Puppet farm. This included fine-tuning of the framework itself as well as its integration with other tools in the infrastructure. The need for such workflow and its underlying infrastructure was mostly driven by an increasing number of Puppet contributors in Conformiq, so the development process had to be better supported and administered.

The main objective of this thesis project was to implement the missing features which included both – further configuration of the Puppet farm and establishing an infrastructure for further development with Puppet. The full list of features is presented in section 3.2.

Out of scope

This project is Linux centered and all concepts, findings and discussions imply Linux as the operating system whenever applicable. Considerations for Windows and other operating systems are beyond the scope of this project.

Conformiq Software started using Puppet to manage system configuration 1.5 years before this project. Reasoning for choosing Puppet (and not some other framework) is beyond the scope of this project. However, some principles of what Puppet is and how it works are presented in section 2.3 in order to familiarize the reader with this framework and to thus make the reading experience smoother.

Automation of repetitive tasks, quick deployments, smooth and safe development procedures can all be resulting in a decrease of human hours spent on these tasks. This thesis project, however, does not undertake any attempt to actually measure the direct monetary benefits provided by automated system configuration management for one reason – developing and propagating configuration changes through Puppet ensures that your infrastructure is sustainable, and because it is scalable of course it is cheaper than manual labor. However, there are other possible techniques to quickly develop and propagate configuration changes (e.g. image cloning) that are not as sustainable as with Puppet. So, comparing the Puppet way to manual configuration to find the monetary benefits and not considering these other techniques would not be extremely revealing. This could probably make a topic for a separate deeper research.

2 Theoretical Background

2.1 Automation of system configuration management

In one of his books on system configuration management James Turnbull writes that "the lives of system administrators and in general individuals employed in IT's operational sector often revolve around a series of repetitive tasks: configuring hosts, creating users, managing applications, daemons, and services" (Turnbull 2007, 1). The repetitiveness of these tasks can be host-based, for example to ensure that same or similar configuration is applied to a number of hosts, and it can be time-based, i.e. "in the lifecycle of one host in order to add new configuration or remedy configuration that has changed through error, entropy, or development" (Turnbull 2007, 1).

Systems (desktops and servers) can, naturally, be configured ad-hoc by e.g. logging in to the machine and manually configuring system resources like users, packages, services, mount points etc. No matter if configuration is performed by executing commands one after another in a shell (e.g. bash or zsh) or using graphical user interface tools "these tasks can be an ineffective use of time and effort" (Turnbull 2007, 1).

"The usual first response to these tasks is to try to automate them. This leads to the development of custom-built scripts and applications" (Turnbull 2007, 1). Such scripts can be simple, designed to manage a single configuration resource on a host or on a group of hosts with sufficiently identical environment. They can also be complex and manage multiple resources on a group of hosts with heterogeneous environments. Simple scripts can contain a simple set of configuration procedures, while others - to support configuration of diverse systems, can also contain functions for getting necessary data from systems as well as necessary logic for applying procedures appropriate for systems in question; and depending on system administrator's intentions as well as on conditions [such as hardware, operating system and software tools] it is possible to create scripts which can bring systems to various (including identical) states of configuration.

There is a number of benefits that can be derived from automated system configuration management tools. First of all, they can make system administration more efficient by automatically applying repetitive configuration procedures on multiple nodes. Secondly, they can help ensure the quality of configuration by avoiding human configuration errors. They also can serve as technical documentation containing applied procedures and, therefore, describing the state of the configuration (at least the desired state). They can serve as backups of configuration e.g. if the machine needs to be reinstalled or if a new machine with a given configuration has to be deployed, i.e. they can provide increased efficiency and quality of deployments and redeployments.

2.2 Automation and centralization challenges

Over time as systems evolve the scripts can grow in size and complexity and can become more and more difficult to use and manage. "Custom scripts and applications rarely scale to suit large environments and often have issues of stability, flexibility, and functionality. [...]. This increases the time and effort required to develop and maintain the very tools you are hoping to use to reduce administrative efforts" (Turnbull 2007, 1). With this quote James Turnbull directly referred to the first challenge from the ones presented below. However, it can also be applied to the other two - pursuing automated configuration of a group of complex and diverse systems requires proper tools or frameworks to handle the task, otherwise, if development of tools starts taking too much time and effort the whole point of automation becomes secondary.

2.2.1 Support for multiple platforms and distributions

One of the problems of scripts not scaling well in large environments is that "such scripts tend to suit only one target platform, resulting in situations such as the need to create a user creation script for BSD, one for Linux, and yet another one for Solaris" (Turnbull 2007, 1).

This challenge is also applicable in cases when IT infrastructure contains various Linux distributions, i.e. when command-line tools and resource names or paths can differ between systems. For example, a system administrator needs to install Apache web-server on Debian and Red Hat systems, i.e. the challenges for automated system configuration management tool would be to find out which system is Debian and which is Red Hat, to apply different command-line tools (in this case package managers) like ‘aptitude’ for Debian and ‘yum’ for Red Hat and to give the administrator a way to separately define a name of the package for each system like ‘apache2’ for Debian and ‘httpd’ for Red Hat.

2.2.2 Offline nodes

The term “offline nodes” in this context refers to systems which are offline at the time when changes in system configuration are applied. For example, system administrator needs a new user to be created on 20 servers. He or she instructs the framework accordingly, but at the moment of applying these changes several machines were offline due to maintenance, i.e. the new user is created only on machines which were online. The challenge here is to be able to ensure that when the offline machines are back online they should also be instructed to create the new user.

2.2.3 Idempotence

Idempotence deals with ensuring that configuration stays intact if same configuration is applied repeatedly. For example, an Apache web-server’s configuration file is defined to have some specific content and whenever the content is changed the Apache service should be restarted. So, there must be a way for the framework to ensure that the configuration file has the defined content, that it changes if instructed, and that the service is restarted only when it is changed.

2.3 Introduction to Puppet

Puppet is an open source framework and toolset for managing configuration of computer systems (Turnbull, 2011, 1). It usually is deployed in a simple client-server model, where server is called “Puppet master” and clients - computers managed by the Puppet master are called “Puppets” or “nodes” (Turnbull, 2011, 2).

2.3.1 Puppet client-server workflow

A typical workflow of Puppet deployment is presented below in Figure 1.

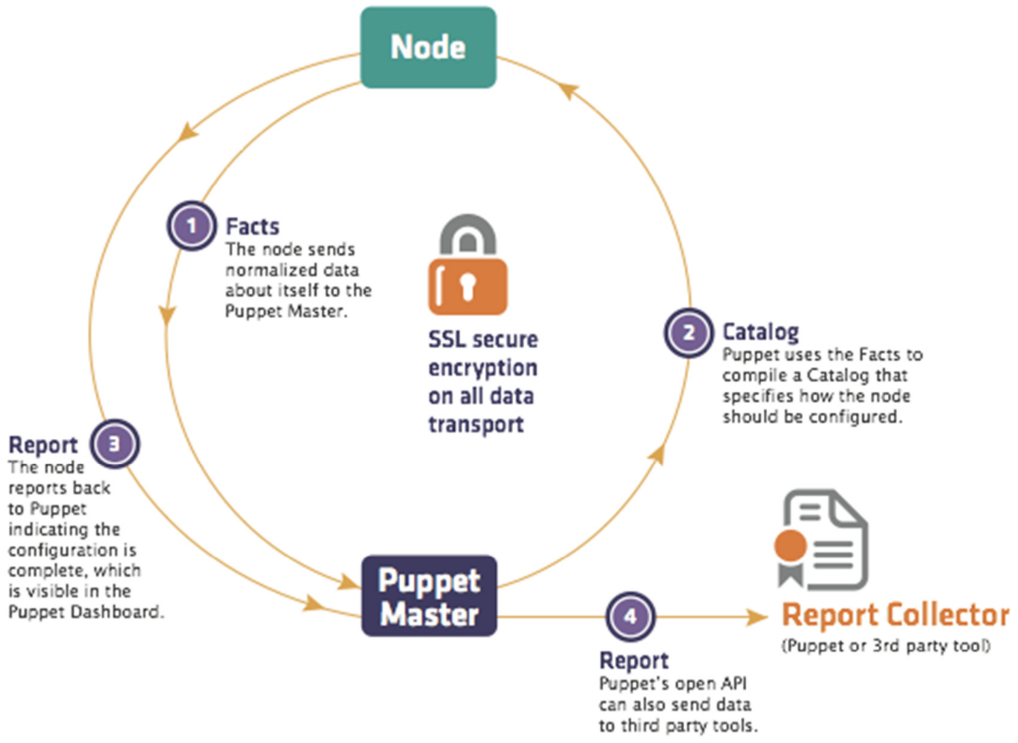
Node
Facts
The node sends normalized data about itself to the Puppet Master.
SSL secure encryption on all data transport
Catalog
Puppet uses the Facts to compile a Catalog that specifies how the node should be configured.
Report
The node reports back to Puppet indicating the configuration is complete, which is visible in the Puppet Dashboard.
Puppet Master
Report
Puppet's open API can also send data to third party tools.
Report Collector
(Puppet or 3rd party tool)

Figure 1: Puppet client-server workflow model (source: http://www.aosabook.org/en/puppet.html)

The communication starts with Puppet nodes connecting to master by sending data about themselves and asking if there is any configuration they should apply. The fact that [typically / at least optionally] Puppet nodes are initiating the communication addresses the “offline nodes” challenge described in 2.2.2. Nodes can be configured to ask for new configuration at any time interval (default is 30 minutes) as well as at boot time, which ensures that any new configuration will be applied with a maximum delay of the defined time interval or if the node is offline - then whenever it is back online.

All the required configuration for Puppet nodes is defined on the master. When a node requests for new configuration the master checks what configuration is defined for the node, compiles a configuration catalog and sends it to the node, the node then applies the configuration and reports back to the Puppet master. When no new configuration is defined - nothing has to be done. This workflow is supported by the core of Puppet

– its transactional layer, which allows configurations to be created and applied repeatedly on the host which is called to be idempotent, meaning that multiple applications of the same operation will yield the same results (Turnbull, 2011, 5). In practice it means that "Puppet configuration can be safely run multiple times with the same outcome on your host and hence ensuring your configuration stays consistent" (Turnbull, 2011, 6), which addresses the challenge of idempotence presented in section 2.2.3.

On Puppet master the configuration is defined in manifests. Manifests can contain definitions for various configuration resource types, e.g. user, package, file, service, exec (executing commands), mount (mount points) etc.; the number of officially supported resource types manageable by Puppet at the moment of writing is 48 (Puppetlabs, Docs: Type reference). Resources that are related to each other can be grouped into manifests, e.g. the simplified snipped below defines that Postfix is installed and is always running:

```
class postfix {
   package { "postfix":
       ensure => installed,
   }
   service { "postfix":
       ensure    => running,
   }
}
```

Manifests can be organized into modules for better grouping of related resources; for example a "webserver" module might include everything necessary to be a webserver such as Apache configuration files, virtual host templates, and the Puppet code necessary to deploy these (Arundel 2011, 62). Modules are then applied for specific nodes, which actually propagates the configuration to the specified nodes.

2.3.2 Declarative language

To define resources Puppet uses its own declarative language (Turnbull 2011, 3). Declarative nature of the language, as opposed to imperative or procedural, allows defining the state of system configuration as it should be, rather than how it should be done (Turnbull 2011, 3). In the snippet above a package "postfix" is declared to be present on a system, "Puppet handles the "how" by knowing how different platforms and operating systems manage certain types of resources" (Turnbull 2011, 4). According to Turnbull for the "package" type alone Puppet has more than 20 providers covering a variety of tools including yum, aptitude, pkgadd, ports, and emerge (Turnbull, 2011, 4). For example, the above declaration of package "postfix" to be installed would be applicable for nodes running on Debian, Ubuntu, Red Hat, CentOS and other systems. Puppet's ability to handle various abstracted resources by invoking appropriate providers partly addresses the multi-platform / multi-distribution configuration concern outlined in section 2.2.1.

2.3.3 Facter

To find out which provider to use Puppet uses a tool called Facter (Turnbull, 2011, 4), which is an independent, cross-platform Ruby library that gathers basic node information about the hardware and operating system (http://puppetlabs.com/facter).

Facts provided by Facter can also be used by a developer for further overcoming the multi-platform / multi-distribution challenge outlined in 2.2.1. Installing Apache on Debian and Red Hat (an example from 2.2.1) can be accomplished by using the "operatingsystem" fact for manually parameterizing the name of the Apache package:

```
case $operatingsystem {
       debian: { $apachepackage = "apache2" }
       redhat: { $apachepackage = "httpd" }
}
package { $apachepackage:
  ensure => present,
}
```

Facts can also be powerfully used in ERB templates, which are served as configuration files to multiple nodes. For example, Postfix's main.cf should contain a setting called "myhostname" with a value of machine's fully qualified domain name; to serve all nodes with a customized configuration file from a single template it is enough to use the Facter's "fqdn" fact in the template:

```
myhostname = <%= @fqdn %>
```

Such customized Puppet modules can be grouped into specific server roles (in a class-like style) and then these server-role classes can be applied for specific nodes. So, adding a new node to a specific server-role would simply require one definition that new node needs to include the class of the specific-server role. Adding a new module to specific server-role(s) would similarly need inclusion of this module into a class which is known to be applied by the server-role(s) in question. In the snippet below there are definitions of three nodes, all of them need Postfix installed and only one needs Apache; there are several ways to define it, this is just one of them:

```
class base { include postfix }
class webserver {
    include base
    include apache
}
node1 { include base }
node2 { include base }
node3 { include webserver }
```

This should demonstrate how serving new configuration or new nodes is quick and simple.

For more information on how Puppet works the reader is advised to visit Puppetlabs' thoroughly-documented website at http://puppetlabs.com/.

3 Project Background

3.1 Environment

Conformiq Software Oy is a small-sized software developing company located in Espoo, Finland. Its main operations consist of development and technical support of several Conformiq Software tools specialized for automated model-based software testing. The company's two core teams (R&D and Customer Success) as well as system administration and most of company's computing resources are located in Espoo. Currently there are 20 employees working in the Finnish office.

Over the past two years Conformiq's IT infrastructure has been growing in size and complexity - the number of servers running inside the corporate network has more than tripled and servers' roles and configuration have become more diversified. The main driver for undergoing this change was the need to better support company's operations in developing, testing, benchmarking and releasing Conformiq Software products by increasing processing capacity to avoid bottlenecks in continuous integration system orchestrating test-, benchmark- and release builds.

Development and maintenance of such growing infrastructure was posing specific challenges. Various server roles used in the continuous integration farm implied that configuration of machines have to be precisely calibrated to ensure that various server roles' are serving their intended purposes and that servers within each role definition have identical configuration. Thus, development and maintenance of this infrastructure using same tools and processes as before the expansion would require more time and effort to be spent to manually configure each server. Quality of configuration would also be more difficult to ensure due to human errors / negligence.

These challenges have posed a need for a system capable of automating and centralizing management of systems' configuration. Puppet framework was chosen to manage system configuration on most of Conformiq's continuous ingratiation farm's members. In summer 2012 while planning automated deployment of new servers as well as redeployment of the old ones system I have developed a set of Puppet modules (collections of Puppet manifests), which would help bring the new and the old servers to the desired states of configuration in an automated and centralized manner. Subsequent configuration changes to these and other nodes were implemented through the existing and new Puppet modules.

Before this project started I have already been using Puppet in Conformiq to manage configuration of 15 servers, a dozen of virtual machines and 1 user workstation, i.e. about 20+ nodes running 4 different distributions of Linux spanning 7 different server roles. Among the manageable resources were system configuration, user management, package management, and service management.

3.2 Objectives

As mentioned above, at the beginning of this project Puppet was already used to manage system configuration of a group of Conformiq servers and although this has been seen as a significant development in system administration in Conformiq there still is a lot of room for improvement. This project's aim is to create new definitions for further configuration of the Puppet farm nodes and also to establish a more favorable infrastructure for further development of the Puppet farm using an improved workflow for this process. Below are objectives of this project named and described as features of the desired Puppet-related functionality:

- Configure Puppet reports to ensure that changes in configuration served by Puppet are reported to system administrator.
- Integrate development with Puppet and Conformiq's version control system (SVN) to make sure that all changes first go to version control and from there are deployed onto Puppet server. Automated Puppet syntax check before committing considered as a bonus.
- Ensure packages' versions stay in sync across all nodes.
- Install and configure a local Debian package repository for distributing custom Debian packages.
- Centralize and automate user management - minimize number of actions needed for adding and editing user accounts served by Puppet; simplify the

procedure for users to edit their accounts (e.g. changing their passwords and default shell)

- Develop an environment for changes in configuration served by Puppet - create an environment for safe testing of new or edited manifests and modules before using them in production.

4 Methods

This chapter describes the research methods used in the project for gathering information, development methodology applied in the project, as well as working process and tools used thought the project.

4.1 Research methods

All the necessary material used during this project can be divided into three groups: books, documentation and various IT-related sources in the internet. Books are mostly Puppet related and have been used to gather higher-level ideas of what Puppet is and what it can be used for. Documentation has been used to solve more practical issues related to Puppet, Subversion (SVN), Python, Bash, Vagrant etc., i.e. issues dealing with [lower-level ideas for] implementing functionality and addressing e.g. language semantics and syntax. By online IT-related resources I refer to various questions-and-answers web sites where technical solutions are suggested for specific technical issues e.g. serverfault.com (for system administration) and stackoverflow.com (for programming) as well as various public wikis, forums, blogs, mailing lists etc.

These research methods were used throughout the project, each serving a different purpose - specific chapters in books were mostly used for gathering and analyzing requirements and designing solutions on conceptual level. Documentation and other online resources were found useful for looking for technical insights and solutions mostly while implementing solutions, but also during design (on technical level) and test phases. While documentation has been the primary source for authoritative technical insights, other internet resources despite their "anonymity" and non-authoritativeness also proved to be extremely helpful when looking for quick solutions to known technical problems.

4.2 Development methodology

This thesis project has been conducted in a fashion of agile development [as opposed to the traditional "waterfall" model]. Main rationale for choosing agile methodology was that development of Puppet infrastructure just like the rest of system administration in Conformiq is feature-driven - new functionality is requested as features or tasks with various levels of priorities / severities, which can change. Therefore, to be able to adapt to the potentially changing environment these features and tasks were modularized into independent **tasks** which could be worked on independently of each other. The order of implementing tasks was dictated by tasks' priorities and when priorities were changing the order was changing as well (see Project management for details).

It is worth noting that 'officially' no specific agile development methodology was adopted for this project. For example, agile development usually implies a cooperation of a group of people with diverse technical know-how, or short daily progress meetings none of which were applicable since the project was conducted by one person only.

4.3 Working process and tools

In this project each feature implementation was in itself an iteration of traditional development stages. Following workflow and tools have been applied:

Gathering and analyzing requirements for each feature started as a ticket in Conformiq's bug tracking system (Trac). Puppet related tickets were commonly opened by either the system administrator or someone from R&D. Progress of development of the feature was reported to the ticket. Puppet related tickets were public for anyone in Conformiq, i.e. apart from reporter of a ticket, its owner and who is deliberately added in CC, anyone could see the progress of the ticket through search or timeline.

In Conformiq's system administration design is often split into conceptual and technical sides, while conceptually solutions are described in a related Trac ticket the technical side usually involves a prototype in the production environment or a replicated production environment for development. Replication of the production environment has been achieved through such virtualization tool as VirtualBox.

Implementation and testing here are referred to the actual implementation and testing of a feature in the production environment. Most common artifacts of this phase were Puppet modules and manifests, Puppet configuration files as well as other scripts and configuration files. All implementation artifacts have been committed to Conformiq's version control repository.

Documentation phase included publishing wiki pages (hosted at Conformiq's Trac) with information on each feature's functionality. This information has been written as user guidelines and has been targeted at both - mostly at developers who work with Puppet infrastructure but also at users who are included in this infrastructure indirectly (e.g. by having a user account managed by Puppet). References to Trac tickets and SVN revisions were made in wiki documentation when applicable.

5 Project

This chapter contains results of the implemented features as well as some facts related to project management.

5.1 Results

This section discusses implemented features, it describes the need for each feature as well as the process of implementation. Where applicable it presents alternatives that I faced while during implementation and, where applicable, my analysis for choosing one alternative over another.

5.1.1 Automated reports of Puppet service runs

Puppet service (agent) had already been configured to run every 60 minutes on all the Puppet slaves. During each run an agent contacts Puppet master to ask if there are any changes in the configuration it should apply. If there are no changes nothing needs to be done. If there are changes Puppet master instructs the agent on what changes should be applied and how. The problem is that all the applied changes are documented in logs and the system administrator has to manually log in to each agent and see if Puppet runs were successful.

This feature's main point was to automate reporting, i.e. to have the results of Puppet service runs emailed to the system administrator. Initially it was planned to configure it so that only reports of runs which contained errors should be emailed, but while designing the solution another option appeared more superior: emailing all the reports if there were any changes in the configuration. This option was preferred to only sending reports with errors for following reasons:

- I as a system administrator stay in control by being aware of all configuration changes distributed through Puppet by others as well by myself
- I stay in control by being informed of any unplanned changes in state of any server, e.g. some service is stopped instead of running

The downside of this option is that more reports would be emailed, but since the changes distributed by Puppet master are not too frequent and the reports are concise no data overflow is expected.

The functionality of this stage was implemented by modifying an already existing module "puppet_conf" (Appendix 2), which is used to distribute agent's configuration, to report results of Puppet service runs to Puppet master and by configuring Puppet master (Appendix 1) to email these reports to system administrator.

After later implementing another feature (SVN post-commit hooks) email notifications of Puppet service runs became disrupted - post-commit hook was restarting the Puppet master in a way that would fail to properly construct email headers ("From" field would contain a non-RFC-compliant address, for some reason the Puppet could not correctly resolve its own host name). Whist if Puppet master was restarted manually reporting would work as expected. Some investigation was made to find out how these two ways would differ (e.g. if effective user id was different from real user id), but the cause was not found. However, to fix the issue it was sufficient to explicitly define it in the Puppet master's configuration settings file (Appendix 1).

5.1.2 Local repository for custom Debian packages

In order to avoid building some software packages from source on each Puppet node one can build them once into a Debian package, include it into a local repository, and instruct all Puppet nodes to install the package from the repository.

The local repository for custom Debian packages was installed by creating a Puppet module "reprepro" (Appendix 3) which ensures that: a package "reprepro" – a tool to manage a repository of Debian packages (SPI Inc, Reprpepro) is installed, was required directory structure is created, configuration files with defined content were ensured to be present in appropriate locations, "apache" service was configured to be restarted whenever there is a change in the repository-related apache configuration file.

The “reprepro” module was applied to the company’s internal file server, after that all Debian servers’ sources lists had to also include the location of the new local repository. This was implemented by modifying an existing Puppet module “apt” to distribute an updated sources.list configuration file to all the machines (Appendix 4, see inclusion of a URL specific to Squeeze – the custom packages are currently needed on Debian Squeeze machines only).

Because installing and configuring reprepro is a one-server operation one might suggest doing it manually, e.g. to save time and effort spent on automation. However, I see value of doing it with Puppet for several reasons: if the fileserver ever needs to be redeployed or if reprepro ever needs to be deployed elsewhere there will be no need to install and configure it again, instead the existing module can be used to deploy it automatically.

5.1.3 Keep packages’ versions in sync

Making sure that all necessary packages exist on all production servers has already been implemented before this project by the Puppet module “packages”, through which packages are installed to appropriate servers. However this did not ensure that packages’ versions were all constantly up-to-date and in sync. This has been implemented by writing two new modules: “unattended-upgrades” (Appendix 5) and “postfix” (Appendix 6). The “unattended-upgrades” module now ensures that each servers’ package manager periodically checks for updates and applies them when they are available. The “postfix” module ensures that all the servers have a mail transfer agent installed and configured in order to be able to email the system administrator results (logs) of the unattended-upgrades runs.

The choice of unattended-upgrades involved a compromise. The main disadvantage of this tool is that it can only perform upgrades equivalent to “apt-get upgrade” and cannot perform a “deeper” upgrade (with installing new packages as dependencies) equivalent to “apt-get dist-upgrade”. There was an alternative to unattended-upgrades – a more configurable cron-apt tool. However, since cron-apt is the non-recommended way for automatic upgrades (SPI Inc, Cron-apt) the decision was made to go with the unattended-upgrades. The missing functionality of performing an equivalent to apt-get dist-upgrade was solved by writing a custom function for Fabric (a python based automated configuration management tool) where all the servers and commands are defined, and manually executing this function against the bunch of servers at once.

5.1.4 Integrate Puppet and SVN

The common way to develop Puppet manifests at Conformiq was to work directly in Puppet master's working directory, i.e. to work with manifests in production. This was resulting in occasional syntax errors breaking manifests as well as inability to simultaneously edit manifests by more than one user. Integrating Puppet with company's version control repository was aiming to solve these problems by having developers each work with Puppet in their local checkouts and commit their code when done, and automatically checking syntax error in a pre-commit hook and deploying new code to Puppet production in a post-commit hook.

Both SVN pre- and post-commit hooks have already been used by the company and in order to implement the desired functionality without disrupting current SVN services the development and testing was performed in a replicated environment, and only later implemented in production.

SVN pre-commit hook (Appendix 7) was configured to listen to Puppet-related commits and to check new Puppet manifests for syntax errors before actually committing them to SVN; commits with errors in manifests are cancelled and the committer gets notified of a syntax error and its possible location. Implementation of this functionality required a bash for-loop which runs Puppet's tool for validating syntax (puppet parser validate) against each Puppet manifest committed.

SVN post-commit hook (Appendix 8) was implemented so that each Puppet-related commit would deploy a fresh copy of manifests to the Puppet master's working directory. Deployment was split into several steps: exporting Puppet configuration and manifests from SVN to a temporary directory, stopping Puppet master service, replacing old Puppet master working directory with the new copy, and starting the Puppet master. The strict order of these steps ensures that at any point of time when Puppet master is compiling a catalog for some slave the state of its manifests is clearly defined and that it can only serve a catalog based on a definite configuration.

Stopping and starting puppet master during post-commit hook execution posed a problem - since the commit hook is actually run as the same Unix user who makes the commit (over svn+ssh://) the user needs elevated privileges to stop and start the puppet master service. There were several options to deal with the issue, for example, to give all users rights to run /etc/init.d puppetmaster in /etc/sudoers, or write a C wrapper which would stop and start the service, or to not stop and start the service during the post-commit deployment at all.

All of these options had their downsides. Reason for not taking the last options (not stopping and starting the service at all) was described in the above paragraph - Puppet master can compile configuration based on undefined state of its codebase in working directory if deployment is in progress, which is not acceptable; reason for not implementing the first option was that even with all users having privileges to stop and start puppet master there would be conflicts on the file permissions level - after every deployment the ownership of puppet master working directory needs to be writable so that the next Puppet-related commit can succeed, however, leaving the manifests writable through any other means than SVN was regarded as highly undesirable. The downside of the C wrapper implementation was that it required a binary to be owned and executed by "puppet" user (with setuid flag on) and setuid flags are never desirable.

Out of three options the C wrapper was chosen (Appendix 9). It appeared equally bad and equally probable to have a hostile Unix-user with either write access to Puppet working directory or a possibility to become "puppet" user through abusing the suid flag. However, it also appeared much less probable that someone would accidently abuse the suid-enabled binary, become "puppet" user, and for example work with

Puppet manifests in production as opposed to a more likely scenario where a user with writable rights to Puppet manifests in production accidentally edits these manifests.

5.1.5 Centralize and automate user management

Objective of this feature was to further centralize and automate user management on all Puppet nodes.

First of all this meant centralizing user management. Functionality of two existing modules which were taking care of defining users and users-specific data folders were merged into one module (Appendix 10). Creation of users-specific data folders was defined as a function of creation of users, i.e. no separate definitions of such folders' creation is anymore needed.

Secondly, creation and management of user definitions was automated. This was implemented by two scripts run as cron-jobs. One of the scripts (Appendix 12) uses puppet tools to generate user definitions from unix users existing in the system (same system where Puppet master runs), and the second script (Appendix 11) compares these generated definitions with the ones in production; when they differ the ones in production are replaced with the fresh definitions. This way if a new user has to be added it is enough to create him/her on the system with e.g. "adduser" and add to a specific user group, the script then analyzes users of this group and creates the definitions.

Also, now if a user wants to change password or default shell on all Puppet nodes there is no need to edit manifests (as before) - instead it is enough to log in to the central server, make the changes there (e.g. with passwd or chsh -s) and just wait for these changes to be propagated automatically.

There are two scripts instead of one for a reason: processing text (output from puppet tools) was easier to implement with Python, while running Bash command is more natural with a Bash script, it might be beneficial to merge them into one script at some point.

Users have also been split into logical groups for easier and clearer realization of these users on various Puppet nodes - users are now realized in these groups and the groups are included as classes in the nodes' definitions. This resulted in cleaner nodes' definition file (previously all users were realized there) and clearer and easier separation of which user should exist on which group of servers.

5.1.6 Environment for testing new configuration

The need for a testing environment for new configuration served by Puppet comes from the fact that one cannot revert to previous configuration, at least not trivially. If developer accidentally instructs Puppet to partition a hard drive or to upgrade the operating system to the next release version - there will be no easy way back, no matter if all the configuration was stored in version control – some data might get lost and the amount of extra time spent and the scale of the problem might depend on how soon the unwanted change is uncovered.

Implementation of this feature made use of Puppet's native functionality which enables defining various environments. In essence, if a Puppet node's configuration file states that it belongs to a specific environment, then Puppet master will look for modules specified for this environment. The snippet below shows this distinction in Puppet master's definitions for modulepaths.

```
[production]
modulepath = $confdir/modules

[testing]
modulepath = $confdir/$environment/modules:$confdir/modules
```

Production machines are served modules from default location, whilst machines from testing environment are first served from their specific environment module-path, and only then from the default location. This way, one can copy a module from the production module path to testing, Puppet master will then be serving this module to testing environment nodes from testing environment and the rest of the modules will be served from production. This ensures that modules in production keep served to machines in production without interruptions while a copy of one of the modules in under test. After successful testing the module can be moved back to production.

Provision of the actual test-beds had two alternatives. Either to have virtual machines running in the network dedicated for testing. Or to provision such virtual machines on individual basis. The latter option was found to be much more superior because the point of these test-beds was to test new configuration and to possibly be breaking them, so having possibly broken dedicated machines for testing seemed senseless.

Since Conformiq had already been using VirtualBox as the virtualization solution it was logical to employ a popular front-end for VirtualBox called Vagrant, which can provision pre-configured virtual machines from some location on the network directly to developer's workstation "with just a few keystrokes" (Mitchell Hashimoto, VagrantDocs: Packaging).

Further implementation of this feature consisted of installing virtual machines which would be replicas of server-roles from Puppet farm, packaging them into Vagrant boxes, putting them to specific place in the network, and documenting how further Puppet development can incorporate these boxes. Documentation can be seen in Appendices 13 and 14 (one is for using Puppet and Vagrant,while the the other is solely for Vagrant usage).

By default all Puppet nodes are requesting configuration from Puppet master automatically, but for these test-beds it was decided to have the Puppet service not started by default in order to give developers a better control of the environment. This was implemented by changing a template which serves this setting to all nodes. Adding the snippet below to a specific template resulted in all host whose hostname ends on "-v" not having Puppet agent started by default:

```
<% if @hostname =~ /.*-v$/ then %>
# do not start Puppet on this node by default
START=no
<% end %>
```

Similar conditional statement in another template defines these machines configuration as the test-beds

```
<% if @hostname =~ /.*-v$/ then %>
# this node is operating in testing environment
environment = testing
<% end %>
```

5.2 Project management

This thesis project was started in June of 2013 (week 24) and ended in December 2013 (week 51). Throughout the project I have spent a total of 332 hours, out of which:

- 45.5 hours were spent on project management related issues, such as writing and updating project plan, preparing status reports for project meetings, the actual project meetings, and writing meeting minutes
- 151.5 hours were spent on the research, design and implementation, as well as testing and documentation of the features.
- 135 hours were spent on compiling this thesis report

In relation to how resource allocation was planned it is both, close and not. Total amount of time is close to the planned 318 hours. The amount of time spent on project management is also close to the planned 40 hours. What differs much is the amount of time spent writing the thesis report, 135 hours against the 60 hours planned. The 151.5 hours spent on implementation is actually a bit more than was planned to be spent on these features against the planned 134.

Throughout this project the order of features' implementation was changing together with priorities for the features. Features which received higher priority (as e.g. happened with the need for local Debian repository) was implemented without waiting for its turn. Implementation of features which appeared less acute were postponed to make sure that important stuff (including writing of the report) gets done first. This way, there was no time left for implementing two features. One of which I mention in the section 6.4 (visualization of Puppet dependencies) and the other one (configuration of staged deployments) was dropped out completely due to the consideration that it would not bring a significant value.

Throughout this project there have been 5 project meetings, including the starting meeting and the closing meeting. All the minutes of the meetings as well as status reports are submitted to the supervisor of this thesis.

6 Discussion

This chapter presents a discussion of the results of this project - the value of change brought by the implemented features are discussed .

6.1 Reporting

Configuring Puppet master to send automated reports with results of Puppet service runs on all nodes was an important first step in this project - it was much easier to develop Puppet manifests having results of all the service runs available at immediately in one location - in this case my email account, easier in comparison to prior development of Puppet manifests and manual checking these results from each node's syslog. Email reports are also invaluable for further development of Puppet infrastructure after this project for its ability to keep me informed about changes made to Puppet configuration by other Conformiq employees.

6.2 Package management

Enhancement of package management with Puppet had two objectives: to make sure that packages are in sync across Puppet nodes and to enable distributing custom Debian packages which are built and served inside company's internal network. Previously, packages were managed with module "packages" that defined which packages have to be installed on which servers; after some time versions of these packages started varying between each other even on machines with an identical purpose e.g. because someone decided to manually upgrade some packages on one machine for building Conformiq releases but not on all. And since identical purpose implies identical configuration, this clearly had to be fixed. Implementing modules "unattended-upgrades" and "postfix" ensured that packages which are expected to be upgraded are upgraded and kept in sync across various same-purpose machines and that all results of unattended-upgrades runs are emailed to system administrator.

The bottom line is that previously in order to keep packages in sync I had to follow Debian mailing lists (at least debian-security@lists.debian.org), when appropriate patches were available to manually login to each server and run an upgrade command, and to then go through the output of each machine's results to make sure everything went as expected. As a result sometimes it was taking time to make sure packages are in sync and during other times [when it is not taken care of] they were not [in sync]. Now the I simply have to check my email and go through summarized reports of each of unattended-upgrades runs.

Setting up a local repository for custom Debian packages was meant to ease installation of some specific software which has to be built in-house - it gives the possibility to build a specific package once, publish it in the repository and declare in module 'packages' that it has to be installed on some machine(s). During this project it did prove to be useful once when one of Conformiq developer's built several packages of specific JDK-6 version (6.27), which was not available in Debian Squeeze repositories. So, he published it through the local repository, which soon resulted in all Squeeze machines having the openjdk-6.27. Considering that there were only two Squeeze servers running at that time an immediate value of the local repository serving openjdk-6.27 might appear quite low, however, considering the potential scalability value of e.g. not having to go through the build process every time a Squeeze server needs to be reinstalled (or additional Squeeze server is added) or the other packages need to be built from source (as it currently is with qt4 and acetao) one can conclude that benefits from the local repository are yet to be realized and appreciated.

6.3 User management

Automating and centralizing user management involved simplifying two aspects of user management: users' self-management (e.g. changing password or default shell) and additions of new users to Puppet infrastructure. The main idea behind this part was to cut down any unnecessary actions from user's and from system administrator's sides

If previously a user for changing his/her password on all Puppet nodes needed to access the central server, change the password there, get the hash of this password, and update his/her entry in the Puppet "users" manifest with this hash, now the user needs to access the central server and update his/her password, the rest will be taken care of behind the scenes. The benefit of this is that no knowledge of Puppet is required to perform such action, also it ensures that passwords will be in sync and that there will be no such situation when user updates the password on the central server but does not update the Puppet manifest because of forgetting or not knowing about this procedure.

Addition of new users had previously required system administrator to edit two manifests by hand, which is requiring time and effort, and is error-prone. After implementing improvements to user management there is only one module and even this does not have to be worked with directly, instead it is enough to add a new user to the central server and the changes will be propagated to this Puppet manifest behind the scenes.

To sum up, this enables users to more easily manage their own accounts, and system administrator - to easier and quicker add new users.

6.4 Workflow for further development of infrastructure with Puppet

With SVN pre- and post-commit hooks the process of development of Puppet manifests and modules became more standardized, to be more exact here were introduced first steps of a standard procedure for development of Puppet manifests and modules (further steps mentioned later). This includes a single way to work with manifests only through the SVN repository, where manifests are checked for syntax before committing and deployed to Puppet master working directory after committing. The value added with this feature is that several developers can work on Puppet modules independently, changes are deployed from SVN to Puppet working directory automatically, and that syntax errors will not creep into production environment.

With Vagrant boxes the further development of infrastructure with Puppet is more safe. Checking for syntax errors in SVN pre-commit hook is great, but it does not prevent from semantic errors that can result in unexpected system behaviors included corrupting system and losing data. Vagrant boxes can serve as test-beds where results of configuration changes can be studied in detail before applying these changes in production. This allows more developers with little or no Puppet experience to start working with Puppet without any fear of breaking configuration in production.

A feature for visualization of Puppet dependencies was planned to be implemented to better support understanding of what configuration is served to which kind of nodes and how some configuration definitions are influencing other definitions, but due to lack of time implementation of this feature had to be postponed and later cancelled. However, this is the very next step to be done (though already outside of this project) to enable developers to better understand interrelationship of Puppet definitions, and to thus make the development process more confident.

7 Conclusion

Right tools can be a prerequisite of work done well and on time. Puppet as a framework for automated and centralized system configuration management is definitely the right tool for Conformiq Software, and to a great extent this tool is one of the reasons for the success of this project.

Features implemented during this project have all been tested and at the moment of writing are in production.

The improved package management ensures that the quality of configuration meets the expectations on all the 20+ nodes in the Puppet farm. Improved user management enables system administrator and users themselves to better administer appropriate user resources. Integration of Puppet with Conformiq's version control empowers distributed development, basic Puppet syntax check and automated deployment of changes into production. Environment for testing changes in configuration minimizes the risk of breaking down systems that are controlled by Puppet. And last but not least, all configuration changes distributed within the Puppet farm are reported to the system administrator; staying informed is staying in control.

With these features in production Conformiq Software will enjoy higher quality of configuration of its server farm, and quicker, more scalable and safer further development of its infrastructure with Puppet.

8 Recommendations for further development

Development of an IT infrastructure is a constantly ongoing activity and there can be an endless amount of recommendations for further development. However, stemming right from this project I have selected several suggestions that in my opinion are worth considering.

Puppet has been used in Conformiq to also configure two user workstations, but this was more of an experiment. It has not yet been productized, partly because this is not a very frequent use case, partly because users like keeping maintenance of their machines to themselves. But since most of user workstations are running Linux it could be beneficial to find the golden middle of what kind of configuration could be served by Puppet to save users' time and effort. One aspect should however be well considered: user workstations can contain users' private data and since Puppet development is open for anyone in the company there must be a way to make sure that hacking a colleague's machine is not possible (for fun or not).

Vagrant virtual machines used as test-beds for new Puppet configuration can also be used for other purposes, for example as test-beds for testing of some bleeding-edge features of Conformiq's own software products. Also, more virtual machines can be packed into Vagrant boxes, which can then be used for e.g. showing customers demos of Conformiq Software products integrations with other software tools.

Conformiq's Windows machines have been controlled by Puppet in a very limited way. The last time I checked Puppet could only install .msi packages on Windows and Windows did not provide API for managing passwords in any other way than clear-text. However, consider taking more under the same hood, look for workarounds, you are probably not the first person who encountered this problem. Getting more under the same hood makes maintenance and further development easier, especially when the "hood" is a well-developed, well-documented, open-source framework like Puppet.

9 Summary

The basis of this thesis was a necessity to further develop part of the IT infrastructure of Conformiq Software Oy, where at the moment of writing I am working as the system administrator. This part of the infrastructure consisted of a group of servers used by Conformiq Software to develop, test, benchmark and release its software products. The quality of configuration of this group of servers as well as its availability and scalability was of great importance to the company. So, already before this project this part of the infrastructure was developed and maintained by a system configuration framework called Puppet.

This project's main objectives were:

- additional configuration of this group of servers using the Puppet framework
- improvement of a workflow and its underlying infrastructure for better supporting further development of this group of servers with Puppet

These objectives split down into features were implemented one after another. As a result of implementing these features the state of configuration of this group of servers satisfies the requirements. And further development of this group of servers with Puppet is supported by an advanced workflow, specifically preconfigured and integrated tools, and the necessary documentation.

Bibliography

Arundel, J. 2011. Puppet 2.7 Cookbook. Packt Publishing

git-reset(1) Manual Page. URL: https://www.kernel.org/pub/software/scm/git/docs/git-reset.html (accessed on 2013-08-30)

Jan Wolter, Unix Incompatibility Notes. URL: http://unixpapa.com/incnote/setuid.html (accessed on 2013-08-27)

John Altenmueller, setuid on shell scripts URL: http://www.tuxation.com/setuid-on-shell-scripts.html (accessed on 2013-08-27)

Mendel Cooper, Advanced Bash-Scripting Guide. URL: http://www.tldp.org/LDP/abs/html/io-redirection.html (accessed on 2013-08-06)

Mitchell Hashimoto, VagrantDocs: Packaging. URL: http://docs-v1.vagrantup.com/v1/docs/getting-started/packaging.html (accessed on 2013-12-13)

nixCraft, What is Umask. http://www.cyberciti.biz/tips/understanding-linux-unix-umask-value-usage.html (accessed on 2013-08-09)

Puppet Labs, Facter URL: http://puppetlabs.com/facter (accessed on 2013-12-13)

Puppet Labs, Type Reference. URL: http://docs.puppetlabs.com/references/2.7.stable/type.html (accessed on 2013-12-13)

Python Software Foundation, Subprocess management. URL: http://docs.python.org/2.6/library/subprocess.html (accessed on 2013-09-12)

SPI Inc, Debian – Details of package reprpepro in wheezy. URL: http://packages.debian.org/sid/reprepro (accessed on 2013-08-14) stack exchange inc, How to have git-svn take care of empty directories gracefully. URL: http://stackoverflow.com/questions/8051991/how-to-have-git-svn-take-care-of-empty-directories-gracefully accessed on (2013-12-09)

stack exchange inc, How to undo the last Git commit. URL: http://stackoverflow.com/questions/927358/how-to-undo-the-last-git-commit (accessed on 2013-08-06)

Turnbull, J. & McCune, J. 2011, Pro Puppet. Apress

Turnbull, J. 2007. Pulling Strings with Puppet: Configuration Management Made Easy. Apress

unixh4cks, Using SVN commit hooks to validate puppet syntax: URL: http://wiki.unixh4cks.com/index.php/Using_SVN_commit_hooks_to_validate_puppet_syntax (accessed 2013-07-17)

Appendices

For ease of read appendices contain only code relevant to this project, i.e. code that existed before this project and is of no relevance is not included. For example, Appendix 1's "puppet.conf" is not complete, but it contains code/options written in this project and referred to in this report.

Commented out (i.e. not active) configuration options in some configuration files served by modules are not included, again - purely for the ease of read. For example, default configuration that comes with unattended-upgrades (Appendix 5's "50unattended-upgrades.tmpl") is much longer, but at the same time it contains mostly commented out options. Such commented out options which come with default configuration files seemed to not add any value here, hence they were left out.

Appendix 1 - Puppet master's "puppet.conf" and "tagmail.conf"

Extract from Puppet master's puppet.conf

```
# Send reports of Puppet agents' service runs
reports = store,tagmail
reportfrom = XXXX@conformiq.com
tagmap = /etc/puppet/tagmail.conf

# Definitions of module-paths for environments
[production]
modulepath = $confdir/modules

[testing]
modulepath = $confdir/$environment/modules:$confdir/modules
```

Puppet master's tagmail.conf

```
# Send all reports to admin
# possible options are: all, err, or a custom tags defined e.g. in a class
all: XXXX@conformiq.com
```

Appendix 2 - Module "puppet_conf"

puppet_conf module consists of a manifest and two templates, it serves Puppet nodes' configuration files.

Manifest

```
# This class serves puppet configuration files to Puppet nodes.
# Note that /etc/puppet/puppet.conf and /etc/default/puppet serve
templates,
# i.e. their contents will depend on facts received from agents!
# To see the facts run facter on a Puppet node!

class puppet_conf {
    file { "puppet.conf":
       path => "/etc/puppet/puppet.conf",
       owner => "root",
       group => "root",
       mode => 0644,
       content => template("puppet_conf/puppet.conf.tmpl"),
       require => Package["puppet"],
    }
    file { "puppet":
       path => "/etc/default/puppet",
       owner => "root",
       group => "root",
       mode => 0644,
       content => template("puppet_conf/puppet.tmpl"),
       require => Package["puppet"],
    }
}
```

Template puppet.conf.tmpl serving Puppet nodes' /etc/puppet/puppet.conf

```
# report results of Puppet service runs to master
report = true

<%# hostnames ending on "-v" are vagrant boxes and belong to testing
environment %>
<% if @hostname =~ /.*-v$/ then %>
# this node is operating in testing environment
environment = testing
<% end %>
```

Appendix 3 - Module "reprepro"

Manifest

```
# This class installs and configures reprepro (a local Debian repository)
# For adding more distributions edit modules/reprepro/files/distributions

class reprepro {

   package { 'reprepro':
       ensure => installed,
   }

   file { [ '/var/www/repos', '/var/www/repos/apt',
'/var/www/repos/apt/debian', '/var/www/repos/apt/debian/conf']:
       ensure => directory,
       owner => 'root',
       group => 'root',
       mode => 0644,
   }

   file { '/var/www/repos/apt/debian/conf/distributions':
       ensure => present,
       owner => 'root',
       group => 'root',
       mode => 0644,
       source => "puppet:///modules/reprepro/distributions",
   }

   file { '/var/www/repos/apt/debian/conf/options':
       ensure => present,
       owner => 'root',
       group => 'root',
       mode => 0644,
       source => "puppet:///modules/reprepro/options",
   }

   file { '/etc/apache2/conf.d/repos':
       ensure => present,
       owner => 'root',
       group => 'root',
       mode => 0644,
       source => "puppet:///modules/reprepro/repos",
   }

   exec { '/etc/init.d/apache2 reload':
       subscribe => File["/etc/apache2/conf.d/repos"],
       refreshonly => true,
   }
}
```

distributions file

```
Origin: Conformiq
Label: Conformiq
Codename: squeeze
Architectures: i386 amd64
Components: main
Description: Conformiq repository for custom Debian Squeeze packages
```

options file

```
verbose
basedir /var/www/repos/apt/debian
```

repos file

```
# /etc/apache2/conf.d/repos

<Directory /var/www/repos/ >
       # We want the user to be able to browse the directory manually
       Options Indexes FollowSymLinks Multiviews
       Order allow,deny
       Allow from all
</Directory>

# This syntax supports several repositories, e.g. one for Debian, one for
Ubuntu.
# Replace * with debian, if you intend to support one distribution only.
<Directory "/var/www/repos/apt/*/db/">
       Order allow,deny
       Deny from all
</Directory>

<Directory "/var/www/repos/apt/*/conf/">
       Order allow,deny
       Deny from all
</Directory>

<Directory "/var/www/repos/apt/*/incoming/">
       Order allow,deny
       Deny from all
</Directory>
```

Appendix 4 - Module "apt"

Manifest

```
# vim:et sw=4 ts=4:
class apt {
   file {"/etc/apt/sources.list":
       ensure => present,
       owner => 'root',
       group => 'root',
       mode => 0644,
       content => template("apt/sources.list"),
   }
   file {"/etc/apt/apt.conf":
       ensure => present,
       owner => 'root',
       group => 'root',
       mode => 0644,
       source => "puppet:///modules/apt/etc/apt/apt.conf",
   }

   # Run apt-get update when anything beneath /etc/apt/ changes
   exec {"apt-get update":
       command => "/usr/bin/apt-get update",
       onlyif => "/bin/sh -c '[ ! -f /var/cache/apt/pkgcache.bin ] ||
/usr/bin/find /etc/apt/* -cnewer /var/cache/apt/pkgcache.bin | /bin/grep .
> /dev/null'",
   }
}
```

sources.list

```
<%# This is a template for serving list of repository sources for various
distributions %>
deb http://ftp.fi.debian.org/debian <%= @lsbdistcodename %> main
deb-src http://ftp.fi.debian.org/debian <%= @lsbdistcodename %> main

deb http://security.debian.org/ <%= @lsbdistcodename %>/updates main
deb-src http://security.debian.org/ <%= @lsbdistcodename %>/updates main

<%# sources for Debian Squeeze (6) only %>
<% if @lsbdistcodename == "squeeze" then %>
# squeeze-updates, previously known as 'volatile'
deb http://ftp.fi.debian.org/debian/ squeeze-updates main
deb-src http://ftp.fi.debian.org/debian/ squeeze-updates main

# conformiq internal packages repository
deb http://XXXX/repos/apt/debian <%= @lsbdistcodename %> main
<% end %>
```

Appendix 5 - Module “unattended-upgrades”

Manifest

```
# This class installs and configures unattended-upgrades.
# Configuration options are in the two files (one is a template) referenced
below.

class unattended-upgrades  {

   package { "unattended-upgrades":
       ensure => present,
   }
   file { "/etc/apt/apt.conf.d/50unattended-upgrades":
               ensure => present,
               owner => 'root',
               group => 'root',
               mode => 0644,
               content  => template("unattended-upgrades/50unattended-
upgrades.tmpl"),
               require => Package["unattended-upgrades"],
   }
   file { "/etc/apt/apt.conf.d/10periodic":
               ensure => present,
               owner => 'root',
               group => 'root',
               mode => 0644,
               source => "puppet:///modules/unattended-
upgrades/10periodic",
               require => Package["unattended-upgrades"],
   }
}
```

10periodic file

```
// Enable the update/upgrade script (0=disable)
APT::Periodic::Enable "1";

// Do "apt-get update" automatically every n-days (0=disable)
APT::Periodic::Update-Package-Lists "1";

// Do "apt-get upgrade --download-only" every n-days (0=disable)
APT::Periodic::Download-Upgradeable-Packages "1";

// Run the "unattended-upgrade" security upgrade script
// every n-days (0=disabled)
// Requires the package "unattended-upgrades" and will write
// a log in /var/log/unattended-upgrades
APT::Periodic::Unattended-Upgrade "1";

// Do "apt-get autoclean" every n-days (0=disable)
APT::Periodic::AutocleanInterval "1";

APT::Periodic::RandomSleep "10";
APT::Periodic::Verbose "1";
```

50unattended-upgrades.tmpl template

```
// Automatically upgrade packages from these origin patterns
       // migration to the specified archive (e.g. testing becomes the
       // new stable).
// allowed origins for <%= @lsbdistcodename %>
<% if @lsbdistcodename == "wheezy" then %>
Unattended-Upgrade::Origins-Pattern {
    "o=Debian,a=stable";
    "o=Debian,a=stable-updates";
    "origin=Debian,archive=stable,label=Debian-Security";
<% end %>
<% if @lsbdistcodename == "squeeze" then %>
Unattended-Upgrade::Allowed-Origins {
       "${distro_id} oldstable";
       "${distro_id} ${distro_codename}-security";
<% end %>
};
```

Appendix 6 - Module “postfix”

Manifest

```
# This class installs postfix, serves appropriate config files (one is a
template), and makes sure services are restarted when needed

class postfix {

   package { "postfix":
       ensure => installed,
   }

   service { "postfix":
       ensure    => running,
       enable    => true,
       hasstatus => true,
       restart   => '/etc/init.d/postfix reload',
       require   => Package['postfix'],
   }

   file { "/etc/postfix/main.cf":
       ensure => present,
       owner => "root",
       group => "root",
       mode => 0644,
       content => template("postfix/main.cf.tmpl"),
       require => Package["postfix"],
       notify  => Service['postfix'],
   }

   file { "/etc/aliases":
       ensure => present,
       owner => "root",
       group => "root",
       mode => 0644,
       source => "puppet:///modules/postfix/aliases",
       require => Package["postfix"],
   }

   exec { "/usr/sbin/postalias /etc/aliases":
       subscribe => File["/etc/aliases"],
       refreshonly => true,
   }
}
```

mail.cf.tmpl template

```
# Postfix main.cf configuration, fed by puppet (upon refreshing all locally
made changes will be reset)
# This template serves dynamic values, like fqdn, and domain. These values
are provided by facter

myhostname = <%= @fqdn %>
myorigin = <%= @domain %>
mydestination = <%= @fqdn %> localhost
relayhost = [XXX.XXX.XXX.XXX]
mynetworks = XXX.XXX.XXX.XXX/XX XXX.XXX.XXX.XXX/XX

alias_maps = hash:/etc/aliases
alias_database = hash:/etc/aliases
mailbox_size_limit = 0
recipient_delimiter = +
inet_interfaces = loopback-only
smtpd_banner = $myhostname ESMTP $mail_name (Debian/GNU)
biff = no

# appending .domain is the MUA's job.
append_dot_mydomain = no

# Uncomment the next line to generate "delayed mail" warnings
#delay_warning_time = 4h

readme_directory = no

# TLS parameters
smtpd_tls_cert_file=/etc/ssl/certs/ssl-cert-snakeoil.pem
smtpd_tls_key_file=/etc/ssl/private/ssl-cert-snakeoil.key
smtpd_use_tls=yes
smtpd_tls_session_cache_database = btree:${data_directory}/smtpd_scache
smtp_tls_session_cache_database = btree:${data_directory}/smtp_scache

# See /usr/share/doc/postfix/TLS_README.gz in the postfix-doc package for
# information on enabling SSL in the smtp client.
```

Appendix 7 - SVN pre-commit hook

```
#!/bin/sh

REPOS="$1"
TXN="$2"

# Define variables
SVNLOOK=/usr/bin/svnlook
PUPPET="/usr/bin/puppet"

# Make sure that the log message contains some text.
$SVNLOOK log -t "$TXN" "$REPOS" | \
  grep "[a-zA-Z0-9]" > /dev/null || exit 1

tmpfile=$(mktemp)
export
PATH="/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin"

# go through every file in commit and look for Puppet manifests (ending on
.pp)
for file in $($SVNLOOK changed -t "$TXN" "$REPOS" | awk '/^[^D].*\.pp$/
{print $2}')
do
       #
       $SVNLOOK cat -t $TXN $REPOS $file > $tmpfile
       $PUPPET parser validate  $tmpfile 1>&2
       # if there are errors echo the error message and exit with code 1
       if [ $? -ne 0 ]
       then
               echo "Puppet syntax error in $file" 1>&2
               exit 1
       fi
done

# Clean up
rm -rf "$tmpfile"

# All checks passed, so allow the commit.
exit 0
```

** part of this pre-commit hook was readily available at unixh4cks, Using SVN commit hooks to validate puppet syntax

Appendix 8 - SVN post-commit hook

```
#!/bin/bash

REPOS="$1"
REV="$2"
UUID=$(svnlook uuid $REPOS)

mychanges="$(mktemp)"
svnlook changed --revision "$REV" "$REPOS" > "$mychanges"

# If Puppet-related commit run the binary to deploy changes production
if grep -q 'XXXXX/puppet' "$mychanges";
then
   echo "will restart services and move configuration"
   /usr/local/svn2/hooks/post-commit.puppet
else
   echo "uninteresting commit"
fi
rm -rf "$mychanges"
```

Appendix 9 - C wrapper for SVN post-commit hook

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
   // set puppet uid
   setreuid( 116,116 );

   // get latest puppet manifests to a temporary directory
   int exitcode = system( "/usr/local/bin/svn export --force
file:///usr/local/svn2/it/trunk/puppet /tmp/puppet" );

   // if export was successful: stop puppet master, copy new configuration
and clean up, start puppet master
   if ( exitcode == 0) {
       system( "/etc/init.d/puppetmaster stop" );
       system( "/bin/rm -rf /etc/puppet/* ; /bin/cp -r /tmp/puppet/*
/etc/puppet/ ; /bin/rm -rf /tmp/puppet" );
       system( "chown -R puppet:puppet /etc/puppet" );
       system( "chmod -R 755 /etc/puppet" );
       system( "/etc/init.d/puppetmaster start" );
   }
   return 0;
}
```

Appendix 10 - Module “users”

Manifest

```
# USER DEFINITIONS 1 - HEAD
# vim:et sw=4 ts=4:
class users {
   include users::virtual
   include data_users_dir
}

class data_users_dir {
   file{ "/data/users":
       ensure => "directory",
       owner => "root",
       group => "users",
       mode => "755",
   }
}

class users::virtual {
   define localuser ($uid,$gid,$password="",$shell,$home) {
       user { $title:
           ensure    =>      "present",
           uid        =>      $uid,
           groups    =>    $gid,
           shell    =>    $shell,
           home    =>    "$home",
           comment    =>    $realname,
           password    =>    $password,
           managehome    =>      true,
       }

       file { "/data/users/${title}":
           ensure => "directory",
        owner => "${title}",
        group => "users",
        mode => "755"
       }
   }
}

class mysite::users {
   include users::virtual

   case $operatingsystem {
       # name the 'sudo' group for Debian and Suse
       Debian: { $sudo = "sudo" }
       SuSE: { $sudo = "wheel" }
   }

# USER DEFINITIONS - USERS
   @users::virtual::localuser { ‘XXXX':
       gid              => ['users',$sudo],
       home             => '/home/XXXX',
       password         => '$X$XXXXXXXXXXXXXXXXXXXXXXXXXXXX',
       shell            => '/bin/zsh',
       uid              => 'XXXX’,
   }
   @users::virtual::localuser { ‘XXXX’':
       gid              => ['users',$sudo],
       home             => '/home/XXXX’,
       password         => '$X$XXXXXXXXXXXXXXXXXXXXXXXXXXXX',
       shell            => '/bin/bash',
       uid              => ‘XXXX’,
   }
# USER DEFINITIONS 3 - END
}
```

Appendix 11 – Script for comparing definitions from Unix users and Puppet users definitions from production

```
#! /bin/sh

## Script for making sure that users have same definitions on local system
and in Puppet 'users' manifests
## To be run in cron every 30 minutes

TMPDIR="/root/puppet"

# generate users' definitions from current UNIX users of XXXXX
python $TMPDIR/puppet-users-generate.py

# get users' definitions from current Puppet 'users' module
awk '/# USER DEFINITIONS/{n++}{print > "/root/puppet/"n".txt" }'  \
/etc/puppet/modules/users/manifests/init.pp

# compare the resulted files, if they differ - ACT!
DIFFOUTPUT=$(diff "$TMPDIR"/2.txt "$TMPDIR"/puppet-users.tmp)
if [ "$DIFFOUTPUT" != "" ]
then
    echo "Users' definitions have changed! The changes are:"
    echo "$DIFFOUTPUT"
    echo "Putting up new 'users' manifest.."
    cd $TMPDIR && cat 1.txt puppet-users.tmp 3.txt > new.init.pp

    echo "Checking out the latest 'users' manifest from SVN, replacing it
with the new manifest, committing it to SVN.."
    svn checkout file:///XXXXXXX/puppet/modules/users/manifests
"$TMPDIR"/manifests
    cat "$TMPDIR"/new.init.pp > "$TMPDIR"/manifests/init.pp
    svn commit "$TMPDIR"/manifests -m "puppet: automated update of 'users'
manifest"
fi

# clean up
cd $TMPDIR && rm -rf 1.txt 2.txt 3.txt manifests new.init.pp puppet-
users.tmp
```

Appendix 12 – Script for generating Puppet user definitions from Unix users

```
/root/puppet/puppet-users-generate.py
## This script generates users' definitions appropriate for Puppet
configuration.
## Invoked by puppet-users-check.sh from the same directory.

import subprocess

# get a list of users belonging to group XXX (puppetusers)
getent_out = subprocess.Popen(['/usr/bin/getent', 'group', 'XXX'],
stdout=subprocess.PIPE).communicate()[0]
getent_out_split = getent_out.split(":")
usersline = getent_out_split[-1].rstrip()
users = usersline.split(",")

# open the temporary file for writing and insert beginning of the block
f = open("/root/puppet/puppet-users.tmp", "w")
f.write("# USER DEFINITIONS 2")

# iterate through users, generate Puppet definitions
for u in users:
   newblock = ""

   # get block with user definition and split them into lines
   userblock = subprocess.Popen(['puppet', 'resource', 'user', u],
stdout=subprocess.PIPE).communicate()[0]
   blocklines = userblock.splitlines()

   # iterate through lines in blocklines and construct own blocks with only
necessary options:
   # i.e. user, gid, home, password hash, uid, shell
   for line in blocklines:
       # catch the beginning of the block
       # add our custom beginning and the rest
       if "user {" in line:
           line = "\n    @" + line[:4] + "s::virtual::localuser" + line[4:]
           line += "\n        gid              => ['users',$sudo],"
           newblock += line
           newblock += "\n        home             => '/home/%s'," % u
       if "password " in line or "uid" in line or "shell" in line:
           line = "\n      " + line
           newblock += line
       # catch the ending of the block
       if "}" in line:
           line = "\n    " + line
           newblock += line
   # after iterating over all users insert the whole block with user
definitions into temp file
   f.write(newblock)
f.write("\n")
f.close()
```

Appendix 13 – Wiki: Puppet

- **Development of Puppet with Vagrant boxes**

There now is a safe and simple way to try out new things in Puppet before moving them into production.

- **The way in a nut shell**
    - get a virtual machine which is pre-configured to work with modules in testing environment
    - work with modules in testing environment
    - verify that new functionality does what it is supposed to with the new modules
    - check that it does not break something one might expect or suspect
    - after validating the module, move it to production environment

- **Example of a workflow**

Bellow is a workflow example for editing some existing module (say packages) for linrels.

```
# Get started with the linrel vagrant box
mkdir linrel-v && cd linrel-v
vagrant box add linrel-v /storage/vagrant/linrel-v.box
vagrant init linrel-v
vagrant up
vagrant ssh
```

NB! Successful execution of the last command assumes your private key is in ~/.ssh/id_rsa logs you into the Vagrant box. More details on Vagrant boxes are in https://XXXXXXX.conformiq.com/XXXXXXX/wiki/Vagrant

```
# in linrel-v apply latest changes from Puppet production
sudo puppet agent --server=XXXXXXXXX.conformiq.com --no-daemonize
--verbose -onetime

# checkout latest Puppet code
git svn clone svn+ssh://$(whoami)@XXXXX/puppet puppet
cd puppet

# Copy "packages" module from production into testing environment
cp -r modules/packages testing/modules/

# edit, add and commit
vim testing/modules/packages/manifests/init.pp
...
git add testing/modules/packages
git commit
git svn dcommit
```

NB! Now, there are two modules "packages": one in production and one in testing environment. Puppet nodes in testing environment first look in their specified modulepath, then in production, i.e. at this point our linrel-v will be served "packages" module from testing environment and the rest will come from production.

```
# Apply your latest changes commited moments ago

sudo puppet agent --server=XXXXXXXXX.conformiq.com --no-daemonize
--verbose --onetime

# Observe the output, check the system, try something out
# make changes if necessary,
# and when happy and satisfied move the module from testing into
# production
cp -r testing/modules/packages modules/
git rm -r testing/modules/packages
git commit
git svn dcommit --rmdir
```

NB Git-users! In the last command make sure you use --rmdir switch to remove empty directories behind you, because empty directories with module names left in testing/modules will confuse and stall environment Puppet nodes. They will complain that "Cannot compile catalog.." and give an error in manifests/nodes.pp

Vagrant boxes are located in /storage/vagrant, they are populated over time, to request more boxes email XXXXXXX@conformiq.com

- **User management**

Puppet is used to manage users on all of its nodes. Users definitions are stored in a Puppet module "users" at svn+ssh://@XXXX/puppet/modules/users/manifests/init.pp. These definitions are generated automatically by a cronned script, which converts puppet.conformiq.com unix-users belonging to local group "users" into Puppet user definitions. The script runs every 30 minutes and compares latest generated definitions with current ones, and commits latest changes to SVN whenever there are any. Changes can e.g. be addition of a new user, or change of user's password or default shell.

To change a password or a default shell a user should not edit his/her user definition in the above mentioned manifest, instead the user should log in to XXXXXX.conformiq.com and make the necessary operation(s) there; the changes will be auto-generated and auto-committed to the appropriate manifest in SVN, and from there - applied to the Puppet nodes.

For example, to change password and default shell to zsh on all the Puppet nodes:

```
ssh XXXXX.conformiq.com
passwd
chsh -s /bin/zsh
```

- **User folders**

Each user has two user-specific folders on every Puppet node:

- **/home/[user]** - is user's home folder
  - mounted over NFS from XXXXXXX
  - can be used to store stuff which is needed often at least several machines
  - over-the-network performance < locally performed operations
  - backed up
  - disk space is limited - might get a request from admin to clean up

- **/data/users/[user]** is a local folder
    - better performance than over NFS
    - not backed-up
    - disk space not that scarce

- **Editing Puppet manifests**

  The new way of editing [also creating new] Puppet manifests is through our SVN repository. Please note that this is also the only way supported - all local modifications in the Puppet working directory will be overwritten by SVN export, thus all changes made locally will be lost [forever]. The the new workflow is:

    - checkout latest revision of Puppet working directory
    - make necessary modification to manifest[s]
    - commit changes to SVN
    - you changes will be **automatically** validated for Puppet syntax errors by a pre-commit hook
        - commits with no syntax errors will be allowed through
        - commits with syntax errors will be aborted; a message with error details will be displayed in the ouput, after correcting the error commit again
    - your changes will be deployed **automatically** from SVN to Puppet working directory on the Puppet master server by a post-commit hook

- **svn**

```
svn checkout svn+ssh://$(whoami)@XXXXXXX/puppet ; cd puppet
# vim / emacs ...
svn commit
```

- **git svn**

```
git svn clone svn+ssh://$(whoami)@ XXXXXXX/puppet puppet ; cd
puppet
# vim / emacs ...
git add -p
git commit
git svn dcommit
```

NB Git svn users! If during `git svn dcommit` there appeared to be a syntax error in one of your manifests, simply correcting your error and pushing the manifest back to the repository with a newer error-free commit will not succeed, this is due to how git works [differently from svn] - you will be pushing both versions of the manifest: the one with the error and the one without. Most likely you want to keep the changes you just made and to only correct the syntax error, if so then do:

```
# reset to where you just were (and keep all your modifications)
git reset --soft HEAD^
# vim / emacs ...
git add -p
git commit
git svn dcommit
```

- **Puppet reports**

  Puppet is configured to generate and send report on each of Puppet service runs for each node. Reports are generated and sent via email to system administrator whenever there are any changes between the catalog generated by Puppet master and the slave's state.

  Implementation of this configuration has been documented in #3872 and r39735

- **Puppet handling Debian packages' automatic upgrades**

  Two Puppet modules are now taking care of automatic upgrades on lintests, linbenches, and linrels:

  - unattended-upgrades makes sure packages' versions are in sync and up-to-date
  - postfix makes sure that mail agent is installed and configured on each node, this is done so that Puppet nodes are able to email reports from unattended upgrades

  NB! on linrels only security upgrades are applied.

  Implementation of this feature has been documented in #3180 and #3859

- **Adding new node (slave) to Puppet master**

  - On master create a definition of the new node in /etc/puppet/manifests/nodes.pp by adding e.g.

    ```
    • node 'XXXXXX.conformiq.com' { include lintester }
    ```

  - On slave create an ssl certificate and send it to master for approval

    ```
    • sudo puppet agent --server=XXXXXX.conformiq.com --no-daemonize
        --verbose
    ```

  - On master view the request and sign the certificate

    ```
    sudo puppet cert -all
    sudo puppet cert --sign XXXXX.conformiq.com
    ```

  - On slave pull the configuration from master

    ```
    sudo apt-get update
    sudo puppet agent --server=XXXXXX.conformiq.com --no-
    daemonize --verbose --onetime --pluginsync
    ```

- **Vim Puppet editing support**

  ```
  sudo aptitude install vim-puppet
  mkdir -p ~/.vim/{ftdetect,syntax}
  ln -s /usr/share/vim/addons/ftdetect/puppet.vim ~/.vim/ftdetect
  ln -s /usr/share/vim/addons/syntax/puppet.vim ~/.vim/syntax
  ```

- **Puppet Slave**

  Update configuration on a slave manually:

  ```
  ssh someslave
  sudo puppet agent --server=XXXXXXXX.conformiq.com --no-daemonize -
  -verbose --onetime
  ```

  NB! : Puppet slaves are automatically pulling configuration from the master every hour.

Appendix 14 – Wiki: Vagrant

Vagrant Boxes

Vagrant boxes are easy-to-get throwaway virtualbox vms. Developers can get them from our storage whenever they need some certain environment, use them, and then throw away.

All boxes:

- are replicas of their counterparts (with same Puppet configuration that their counterparts have)
- have Puppet certificates signed by master
- are configured for networking behind your host's NAT
- assume that your private key (e.g. for accessing mylly) is in ~/.ssh/id_rsa (to override edit .....)
- have host name ending on "-v" for Puppet to recognize them and serve them as Puppet testing environment machines. Puppet is not running on them at boot. To invoke it run:

```
sudo puppet agent --server=XXXXX.conformiq.com --no-
daemonize --verbose --onetime
```

Install vagrant

```
sudo apt-get -y install vagrant virtualbox-ose
```

Get a Vagrant box from storage and start working with it

```
# example for getting a linrel vm
export machinename="linrel-v"
mkdir $machinename && cd $machinename
vagrant box add $machinename /storage/vagrant/$machinename.box
vagrant box list
vagrant init $machinename
vagrant up
vagrant ssh

# after the machine is not anymore needed you can destroy all its
# traces with
vagrant destroy
rm -rf ~/.vagrant.d/boxes/$machinename
```

Package some virtual machine into a Vagrant box

```
# vagrantfile additions to be packaged with the box
cat >> vagrantfile << EOF
Vagrant::Config.run do |config|
  # ssh to box as current user with existing own key-pair
  config.ssh.username = ENV['USER']
  config.ssh.private_key_path = '~/.ssh/id_rsa'
end
EOF

# pack the box
vagrant package --base linrel-v --vagrantfile vagrantfile
```