Konfigurationsmanagement mit cfengine

Ralph Angenendt / Ulrich Habel

Ziel und Aufbau des Vortrages

Dieser Vortrag zeigt ein mögliches Szenario auf, wie Konfigurationsmanagement mit cfengine umgesetzt werden kann. Es zeigt den Verlauf eines Projektes des Bayerischen Rundfunks in der Hauptabteilung Multimedia im Jahr 2004.

Architektur und Aufbau cfengine

cfengine ist ein Administrationswerkzeug zur Verwaltung von verschiedenen Servern und ihren angebotenen Diensten. cfengine bedient sich dabei einer plattformunabhängigen Beschreibungssprache, die von einem Executable interpretiert und ausgeführt wird. Die Syntax von cfengine ist an keine bekannte Beschreibungssprache angelehnt. Der Autor von cfengine ist Mark Burgess, der cfengine als Teil eines kontinuierlichen Forschungsprojekts an der Universität Oslo entwickelte. Ziel des Projektes ist es Wege und Möglichkeiten der Verwaltung von heterogenen Netzwerken zu entdecken und auszuloten. Eine Einteilung von einzelnen Teilen des Netzwerkes wird durch ein Klassenmodell erreicht.

Das Werkzeug cfengine wird durch Anweisungen in Konfigurationsdateien gesteuert. Diese Dateien enthalten Variablen, Klassendefinitionen und Aufgabensequenzen. Die Besonderheit gegenüber anderen Tools ist hierbei die automatische Einteilung der Plattformen in unterschiedliche Klassen. Die Klassen können in den Aufgabensequenzen einzeln angesprochen werden. Der Ablauf des cfengine Aufrufs erfolgt dabei in zwei, voneinander abhängigen, Phasen. In der ersten Phase, die Updatephase genannt wird, besteht die Möglichkeit vorbereitende Aktionen vor dem eigentlichen cfengine-Lauf durchzuführen. Hier können beispielsweise Konfigurationsdateien von cfengine für den Hauptlauf aktualisiert, erforderliche Konfigurationsänderungen oder ähnliches durchgeführt werden. Sollte man sich durch das administrationstypische Missgeschick ausgesperrt haben, kann man cfengine z.B. zur Wiederherstellung der passwd Datei verwenden.

Beispiel:

```
# Definition einer Klasse
web_server = ( web-server-1 web-server-2 web-server-3)

# Beispiel einer Aufgabensequenz in cfengine
# Kopieren von Dateien
files::
      # Kopieren nur für die Klasse "web_server"
      web_server::
            $(files_directory)/web_server/ dest=/ server=$(config_server)
                  recurse=inf trustkey=true encrypt=true verify=true
      # Kopieren für alle anderen Klassen
      any::
            $(files_directory)/all_hosts/ dest=/ server=$(config_server)
                  recurse=inf trustkey=true encrypt=true verify=true
```

Innerhalb von cfengine sind bereits zahlreiche Sequenzen vorbelegt, z.B.:

tidy	-	Task zum Löschen von Dateien
resolv	-	Task zur Konfiguration von DNS-Clients
editfiles	-	Task zum automatischen Editieren von Dateien

cfengine verwendet für die Distribution der Konfigurationsdateien ein selbstimplementiertes Protokoll, welches aus Sicherheitsgründen via OpenSSL verschlüsselt wird. Bei der Verschlüsselung werden hostabhängige Keys verwandt, die durch ein eigenes Programm erzeugt werden können. Durch einen flexiblen Aufbau der Architektur sind vielfältige Realisierungen möglich. Der Betrieb als Client und als Server ist gleichzeitig möglich. Das Grundprinzip von cfengine im Client/Serverbetrieb baut sich aus mehreren Prozessen wie folgt auf:

– cfservd

  Der cfservd ist ein Daemon welcher üblicherweise mittels eines Initscriptes beim Booten des Rechners mitgestartet wird. Es ist dabei nicht entscheidend ob der Rechner als Client oder als Server für die cfengine eingesetzt wird. Dieser Daemon stellt die Schaltzentrale für alle einkommenden Requests von cfengine dar. Er ist gleichzeitig die ausliefernde Instanz bei Actionsequences, die das Kopieren von Dateien (Konfigurationsdateien von cfengine selber oder auch zur Distribution von Dateien) übernimmt.

cfagent

  Der cfagent ist das Programm welches auf dem Client gestartet wird. Es führt die Befehlssequenzen der Konfigurationsdateien aus. Das Programm kann entweder von Hand im Clientbetrieb oder vom cfengine Server über den cfservd aufgerufen werden. Wird innerhalb der Konfigurationsdateien ein Server als Konfigurationsbackend angegeben, kontaktiert cfagent den Daemon cfservd auf dem Konfigurationsserver.

cfrun

  Das Programm cfrun wird auf dem cfengine Server mit einem Hostnamen als Parameter aufgerufen. Es kontaktiert den parametrisierten Client über den cfservd Daemon und veranlasst ihn auf dem Client das Kommando cfagent aufzurufen.

cfkeys

  Das Programm cfkeys wird zur Erzeugung der hostabhängigen Keys verwandt.

Die am häufigsten verwandten Realisierungmethoden werden nachstehend beschrieben.

a.) Verwendung eines zentralen Servers

Bei der Verwendung eines zentralen Servers werden durch eine cfengine Installation die zu bedienenden Clients gesteuert. Prinzipiell kann der Start der jeweiligen cfengine auf den Clients durch einen Cronjob gesteuert werden, jedoch bietet sich ggf. ein manuelles Aufrufen per Hand an. Diese Methodik wird von BR-Online verwandt um gezielt einzelne Rechnergruppen und deren Konfiguration bzw. Dateien zu administrieren. Der Aufruf zum Starten von cfengine auf dem jeweiligen Client entsteht durch das Kommando cfrun <clientname>. Das Kommando cfrun kontaktiert den Daemon cfservd auf dem jeweiligen Client, der seinerseits das Kommando cfagent aufruft.

b.) Verwendung des zentralen Servers vom Client aus

Genau wie der cfengine-Distributionsserver den Client aktivieren kann, ist es möglich vom Client aus einen Lauf von cfengine zu initiieren. Durch den Aufruf des Kommandos cfagent wird der Daemon cfservd auf dem Konfigurationsserver kontaktiert.

c) Client Betrieb

Für den eigentlichen Betrieb von cfengine ist ein zentraler Konfigurationsserver nicht erforderlich. Denkbar ist es auch die Konfigurationsdateien von cfengine in ein

distributionsspezifisches Paket zu verpacken und über das lokale Paketmanagement zu pflegen bzw. auszuspielen. In diesem Fall bleibt der cfengine-Connect auf dem lokalen Host gebunden. In Zonen höchster Sicherheit bietet sich dieses Vorgehensmodell an. Das zyklische Aufrufen von cfengine bietet sich durch einen Cronjob oder anacron Eintrag an.

Anbindung von cfengine an Versionssysteme

In cfengine ist die Anbindung an ein Versionsverwaltungssystem seitens der Implementierung nicht realisiert. Ein Verwalten der Konfigurationsdateien über ein Versionierungssystem ist jedoch problemlos möglich. Sollte ein zentraler Server, der auch Dateien distributiert, ist auf eine Rechteverwaltung zu achten. cfengine spielt die Dateien mit den gleichen Rechten aus, wie sie auch auf dem Installationsserver vorgefunden werden, daher ist es bei einigen Paketen unvermeidlich, dass man innerhalb von file- oder copy-sequences die Rechte pro Datei oder Verzeichnis anpassen muss.

Standortbestimmung Bayerischer Rundfunk

Die Hauptabteilung Multimedia des Bayerischen Rundfunks betrieb zu Beginn des Jahres 2004 rund 70 Server unter SuSE Linux 7.3 auf homogener IBM Hardware. Die Server wurden mittels des Tools Alice von SuSE initial installiert. Durch eigene Shellscripte konnte Alice auch als Tool für Konfigurationsmanagement verwandt werden.

Bei Einführung des Releases Suse 8.0 wurde das Tool Alice seitens SuSE durch Autoyast ersetzt. Autoyast ermöglicht zwar ebenfalls wie Alice die initiale Installation, jedoch waren alle erstellen Scripts und Tools rund um Alice nicht portierbar. Bedingt durch diese Tatsache startete ein Projekt mit dem Ziel der Evaluation verschiedener Linuxdistributionen und deren Systeme zum Konfigurationsmanagement. Es wurde die Nähe zu einer Enterprise Platform angestrebt. Als wesentliches Ziel des Projektes galt es ein System zu finden welches die schnelle Austauschbarkeit von Rechnern ermöglichte.

Vor allem im Bereich der Webserver wurde die Skalierbarkeit der Gesamtplattform durch schnelles einfügen weiterer Webserver geplant. Die Server sollten daher vollständig automatisch installier- und konfigurierbar sein.

Als Projektergebnis stand folgende Struktur fest:

- Linux Distribution CentOS 3
- Ablösung des Tools Alice für die Installation durch Kickstart
- Ablösung des Konfigurationsmanagement von Alice durch cfengine Version 2
- Etablierung eines zentralen Updatemanagements via cfengine und OS Bordmittel

Teil II - Die Praxis:

Durchführung der Migration auf cfengine

Jetzt ist es also so weit: Alice fehlt in SuSE ab der 8er Serie, autoyast kann nur installieren, aber nicht das Konfigurationsmanagement im laufenden Betrieb übernehmen, eine Auftrennung von Installation und Konfiguration auf verschiedene Tools ist also unumgänglich. Dass wir zusätzlich auf CentOS umsteigen, macht die Migration auch nicht einfacher, der Aufwand in die Einarbeitung in das neue Konfigurationsmanagement ist aber sehr viel größer als der Umstieg auf die neue Distribution. Vor allem wird die Entwicklung bei uns nicht gestresst, da die vorhandenen RPMs sehr einfach an die neue Distribution angepasst werden können. Und wenn das Konfigurationsmanagement einmal läuft, können wir die bisher mit den RPMs ausgespielten Konfigurationen sehr einfach via cfengine auf die Rechner

bringen, ohne gleich das Paketmanagement bemühen zu müssen - die Entwickler verlieren sogar Arbeit, die sie vorher hatten. Durchaus ein Vorteil der neuen Methode.

Arbeit, die jetzt erst einmal vor uns, den Administratoren, liegt. Da wir allerdings den unschätzbaren Vorteil haben, dass wir unter Alice eine sehr gut funktionierende Verwaltung der wichtigsten Konfigurationsdateien haben, entscheiden wir uns, das Modell welches unter Alice verwendet wird, hart auf cfengine zu portieren. Davon versprechen wir uns, dass auch die Administratoren, die nicht primär an der Migration mitarbeiten, schnell auf cfengine umsteigen können. Da alle zumindest soviel Grundwissen haben müssen, um "mal eben" einen Rechner via Alice neu aufzusetzen und zu konfigurieren, hoffen wir, auf existierendes Know-How aufsetzen zu können.

Um den Aufwand dieser Portierung verstehen zu können - und auch die Probleme, in die wir sehenden Auges gelaufen sind - muss man das durch Alice etablierte Modell für die Rechnerkonfiguration zumindest gesehen haben.

Alice und ihre Klassen

Alice verteilt sich auf mindestens zwei Rechner (die theoretisch auch auf einer Maschine liegen können): Den Installationsserver und den Arbeitsplatzrechner des Administrators, der Alice bedient. Der Installationsserver hat neben der Konfiguration auch die komplette Distribution auf der Festplatte liegen, bei der Installation zieht sich der zu installierende Rechner die Daten der Distribution entweder per NFS vom Installationsserver oder der Administrator erstellt am Installationsserver eine Boot-CD, mit der er den Rechner installiert. Für die weiteren Erklärungen ist der Installationsserver allerdings nicht mehr wichtig, man muss nur wissen, dass er vorhanden ist.

Alice ist eine Shellskriptsammlung mit zwei darunter liegenden Perlmodulen, die nichts anderes macht, als einen Stapel Konfigurationsdateien zu lesen, darin enthaltene Variablendefinitionen zu parsen und sie in die entsprechenden Konfigurationsdateien zu schreiben. Wo die Bordmittel von Alice nicht genügen, um komplexe Konfigurationen selbsttätig zu parsen und zu ändern, wird einfach die komplette Konfiguration in eine Variable gesteckt um später in die entsprechend definierte Konfigurationsdatei geschrieben zu werden. Die Definitionen sind in einem Pseudo-XML-Format gehalten, somit können einzelne Konfigurationssegmente oder -dateien durch Tags getrennt werden. Dadurch lassen sich alle zu einem Rechner gehörende Konfigurationen in einer einzigen Datei ablegen.

Auf der anderen Seite "weiß" Alice genügend über den Rechner, der gerade installiert wird, um ihn mit einer angepassten Linuxinstallation zu versehen. Mit anderen Worten: Alice bekommt eine Paketliste vorgegeben, ihr wird erklärt welche Hardware sich in dem Rechner befindet, wie sie die Festplatten partitionieren und formatieren soll usw.

Die Beschreibung für einen Rechner findet sich allerdings nicht in einer Datei wieder, sondern ist über mehrere Dateien verteilt, die einem Klassifizierungssystem gehorchen. Dabei unterscheidet Alice zwischen Konfiguration für Programme, Netzwerkinformationen, systemrelevanten Konfigurationen durch den Suffix der Datei. Konfigurationen finden sich in `.conf.tcf`-Dateien, die Netzwerkinformationen in `network.tcf`, Systeminformationen in `.sys.tcf` usw.

Das ist Alice, wie sie am Arbeitsplatz des Administrators aussieht:

```
[angenenr@localhorst alice]$ls -1
classes
CVS
info
templates
[angenenr@localhorst alice]$
```

Es sind drei Verzeichnisse vorhanden, in denen sich die Konfigurationsdateien befinden.

- templates

  Diese Konfigurationen betreffen alle Rechner die installiert werden. Die Daten in diesem Verzeichnis eignen sich z.B. dazu, eine Standardliste aller Softwarepakete zu definieren, die auf allen Rechnern ausgespielt werden soll, oder sitewide vorhandene Nutzer bzw. Gruppen einzutragen. Auch eine überall vorhandene hosts-Datei sollte hier definiert werden, ebenfalls können die ssh-keys der Administratoren definiert werden.

- classes

  Alice kennt Rechnerklassen, mit denen Rechner, die sich Konfigurationen teilen, an nur einer Stelle definiert werden müssen. Wir haben hier unter anderem Hardwareklassen definiert, in denen zum Beispiel Partitionierungsdaten oder die benötigten Kernel für bestimmte Hardwarekonfigurationen festgelegt werden. Die Konfiguration der Webserver sind ebenfalls hier zusammengefasst.

- info

  Hier befinden sich die Konfigurationen für den einzelnen Rechner, so z.B. Informationen über das Netz in dem er sich befindet, der Hostname des Rechners und Konfigurationen, die sich nicht in einer Klasse zusammenfassen lassen. Die Information in welchen Klassen sich ein Rechner befindet, werden ebenfalls hier festgelegt.

Diese Informationen wertet Alice in der Reihenfolge Templates->Classes->Info aus, wobei Informationen, die in einer niedrigen Kategorie definiert wurden, von einer höheren Kategorie wieder überschrieben werden können. Wird also unterhalb von classes eine Klasse "webserver" definiert, die unter anderem die Information über die httpd.conf bereithält, so kann in info ein Rechner der Klasse "webserver" zugehörig erklärt werden - mit einer anderen httpd.conf.

Wie sieht das nun bei uns aus? Templates haben wir sehr wenige definiert, eine Userliste, eine Gruppenliste und der SSH-Key von zwei Usern sind neben der Paketliste die einzigen vorhandenen Einträge.

Bei den Klassen sind wir etwas kreativer vorgegangen. Da sehr homogene Hardware vorhanden ist, haben wir eine Klasse pro Hardware definiert, die u.a. die Partitionierung und die Informationen über Hardwaretreiber bereithält. Um die Segmentierung unseres Netzwerks abzubilden, haben wir pro Netzwerksegment eine weitere Klasse definiert, in der sich Routen und weitere Netzwerkkonfiguration widerspiegelt - sozusagen "statisches DHCP".

In der info-Sektion sind dann die Konfigurationen für die einzelnen Rechner abgelegt, zum Beispiel zusätzliche Softwarepakete, Hostnamen, rechnerspezifische Konfigurationen usw. Daraus ergibt sich folgende quantitative Verteilung: In templates liegen 11 Dateien, in classes 49 und in info 157. Die 49 Dateien in classes verteilen sich auf 19 von uns definierte Klassen, die meisten davon Hardwareklassen.

Die Migration

Dieses Modell haben wir aus den oben erwähnten Gründen eins zu eins in cfengine übernommen. Mit einigen gravierenden Unterschieden. So arbeitet cfengine dateibasiert, eine `httpd.conf`, die auf dem Zielsystem in `/etc/httpd/conf/httpd.conf` liegen soll, muss auch auf dem Installationsserver genau dort liegen. Bestand also in Alice die Möglichkeit, alle zu einer Klasse gehörenden Konfigurationsdateien in einer Datei auf dem Installationsserver vorzuhalten und innerhalb dieser Datei zu definieren, wo die entsprechende Datei auf dem Zielsystem liegt, muss unter cfengine für die entsprechende Klasse das Dateisystem nachgebildet werden.

cfengine sieht also auf der Arbeitsstation des Administrators wie folgt aus:

```
[ralph@logout cfengine]$ find . -maxdepth 2 -type d
/files
/files/any
/files/hw
/files/sw
/inputs
[ralph@logout cfengine]$
```

Hier zeigt sich ein weiterer Unterschied zwischen cfengine und Alice: Sind in Alice die Steuerungsinformationen innerhalb der Dateien in wohldefinierten Tags gespeichert (also z.B. wo gehört welche Datei auf dem Zielsystem hin), erwartet cfengine diese Informationen in Dateien unterhalb des inputs-Verzeichnis. Dies natürlich in der cfengine eigenen Syntax.

Geht man tiefer in die Verzeichnisstruktur, dann zeigt sich, dass mit der harten Migration der Daten sich nicht nur die Dateianzahl massiv erhöht hat (269 vs. 217, es sind aber noch bei weitem nicht alle Rechner in cfengine abgebildet), auch die Anzahl der Verzeichnisse insgesamt ist von 3 auf 378 angewachsen. Die relativ flache Struktur von Alice expandiert also zu recht tiefen Teilbäumen im Dateisystem.

Nimmt die Übersichtlichkeit im Dateisystem mit cfengine gegenüber Alice ab, so nimmt die Übersichtlichkeit der einzelnen Konfigurationen zu. So bilden wir in Alice z.B. alle Zonen, die unser Nameserver kennt, in einer Datei ab:

```
[angenenr@localhorst classes]$ls -l infoserver.zones.tcf
-rw-r--r--  1 angenenr angenenr 276096 Dec 17 11:42 infoserver.zones.tcf
[angenenr@localhorst classes]$grep "<MISC" infoserver.zones.tcf |wc -l
253
[angenenr@localhorst classes]$
```

Mit cfengine gibt es dann ein Verzeichnis `files/sw/nameserver/var/named/zones/` in dem sich pro Zone eine Datei befindet - wie auf dem Zielserver auch. Dadurch wird die Änderung einer einzelnen Zone stark vereinfacht, da man nicht in einer über 250kB großen Datei nach der jeweiligen Zone suchen muss.

Deshalb findet sich der Administrator innerhalb der von cfengine verwendeten Verzeichnisse besser zurecht, da die Dateien dort liegen, wo er sie auch auf dem Zielsystem vermutet. Wenn es da nicht einen ganz großen Drawback gäbe: Die Beschreibungssprache. Aber dazu später mehr.

Die Probleme

Wie bei jeder Umstellung gilt auch bei dieser: Der erste Wurf war nicht wirklich gelungen. So wird die von Alice benötigte Information über Hardware und Netzstruktur in cfengine an einer ganz anderen Stelle erwartet. Im Falle der Hardware muss cfengine sogar gar nicht wissen, was in den Rechnern vorhanden ist, da die Konfiguration (cfengine) hart von der Installation (kickstart) getrennt ist.

Weiterhin kopieren wir die unter `files/sw/` liegenden Dateien pro Rechner/Rechnerklasse einfach auf das Zielsystem. Haben wir Alice für die

Übertragung von Konfigurationen mit anschließendem Neustart des Services in eigene Shellskripte eingebunden, so ist das bei cfengine nicht möglich. Durch den simplen Kopiervorgang nehmen wir uns vor allem das Feature von cfengine, dass es Services selber neu starten kann, wenn sich die Konfiguration geändert hat. Dadurch müssen wir einige Dateien zweimal kopieren, um diesen Effekt zu erreichen. Und wir müssen darauf achten, dass wir diese zwei Kopiervorgänge in der richtigen Reihenfolge machen, damit cfengine mitbekommt, dass sich die Datei gegenüber dem Zielsystem geändert hat.

Im Großen und Ganzen ist dieser Teil der Migration aber gelungen, wir haben im täglichen Geschäft dazugewonnen, weil Konfigurationsdateien einfacher aufzufinden sind - und wir nur ein Tool benötigen, wo wir vorher eigene Shellskripte schreiben mussten.

Auf der anderen Seite haben wir uns cfengine eingekauft. Mit allen Vor- und Nachteilen. Dazu jetzt mehr.

Der Einsatz von cfengine

Für den Übergangsbetrieb bei der Migration haben wir uns für die Push-Variante entschieden, da wir damit von einem zentralen Ort aus entscheiden können, wann welcher Rechner seine Konfiguration ändert. Sind wir allerdings sicher, dass unsere Idee eines Konfigurationsmanagements funktioniert, werden wir auf per crontab automatisierte Pull-Variante umsteigen, so dass z.B. eine geänderte `virtual.conf` für die Webserver auf den zentralen Installationsserver gelegt wird, die Clients holen sich diese Datei dann irgendwann selber ab und sorgen dafür, dass der HTTP-Server seine Konfiguration neu einliest.

Die Konfiguration

cfengine ist/beinhaltet neben der Ausspielmöglichkeit von Konfigurationen eine eigene Beschreibungssprache, in der die Steuerdateien von cfengine geschrieben werden. Die drei wichtigsten Konfigurationsdateien sind

- cfservd.conf

  In dieser Datei wird die Konfiguration des Installationsservers festgelegt.

- update.conf

  Diese Datei wird vom Client ausgelesen, hier wird festgelegt, welche Daten er vom Konfigurationsserver auslesen muss, damit seine lokale Konfiguration von cfengine vollständig ist.

- cfagent.conf

  Diese Datei wird beim Aufruf von cfagent auf dem Client aufgerufen. Diese Datei legt fest, in welcher Reihenfolge welche Aktionen auf dem Client durchgeführt werden.

`cfservd.conf` und `update.conf` werden sehr einfach und simpel gehalten, damit interne Updates von cfengine funktionieren - die Daten auf dem Installationsserver und auf den Clients also stets gleich sind. Auf eine simplifizierte Version unserer `cfagent.conf` möchte ich hier allerdings kurz eingehen:

```
control:
  actionsequence = (
    resolve
    tidy
    copy
    packages
    editfiles
    files
    shellcommands
    processes
    links
  )
```

Das hier ist das Kernstück unserer cfengine-Installation - die actionsequence in der `cfagent.conf` legt fest, in welcher Reihenfolge welche Bereiche der Konfiguration abgearbeitet werden - die cfservd.conf kann z.B. komplett anders aussehen, weil man z.B. via push nur Teilbereiche auf dem Zielhost konsistent halten möchte.

Was in den einzelnen Bereichen genau passiert, steht ein paar Zeilen weiter unten, hier also nur ein grober Überblick über das, was hier ausgelöst wird:

resolve: Die `/etc/resolv.conf` wird geprüft - ist der Rechner mittlerweile in ein anderes Netz gewandert, so wird sie automatisch angepasst.

tidy: In diesem Prozess wird Müll weggeräumt, so z.B. Backupfiles, oder alte . cfsaved-Dateien von cfengine selber.

copy: Kopiert einzelne Dateien auf die Zielsysteme.

packages: Hier kann für einzelne Rechner definiert werden, welche Pakete auf jeden Fall vorhanden sein müssen - diese können dann via cfengine installiert werden.

editfiles: Auf dem System vorhandene Dateien werden auf Vollständigkeit geprüft - sind sie es nicht, werden Zeilen eingefügt.

files: Kopiert die oben aufgezeigten Dateistrukturen unter `files/sw/` auf die jeweiligen Rechner.

shellcommands: Definiert auf dem Host auszuführende "Shellkommandos" für einzelne Klassen. Beispiele hierfür folgen.

processes: Hiermit lässt sich überprüfen, ob Prozesse auf dem Zielsystem laufen – und diese können im Ernstfall neu gestartet oder gestoppt werden.

links: Legt auf dem Zielsystem symbolische oder harte links an.

```
  config_server = ( forge )
  cfengine_root = ( /local/var/cfengine )
  files_directory = ( "$(cfengine_root)/files" )
  config_directory = ( "$(cfengine_root)/inputs" )
  DefaultPkgMgr = ( rpm )
  yum_install = ( "/usr/bin/yum -y -d0 -e0 install" )
  yum_update = ( "/usr/bin/yum -y -d0 -e0 update" )
```

Dieser Bereich dient nur für die interne Konfiguration. Der Rechner forge ist Installations- bzw. Konfigurationsserver, wobei cfengine die passende Domain selbst hinzufügt. Der `DefaultPkgMgr` wird unter anderem benötigt, um auf dem Zielsystem auf benötigte Pakete zu prüfen, wohingegen wir lieber mit `yum` als mit `rpm` installieren, damit Abhängigkeiten automatisiert nachinstalliert werden.

```
 ipv4_195::
    domain = ( br-online.de )

 INTERNAL_HOSTS::
    domain = ( br.de )
```

Wer sich vorher fragte, woher cfengine die passende Domain kennt (resolve und beim config_server) - hierher. Alles, was mit `195.` anfängt, steht außerhalb des internen Netzes und bekommt daher einen offiziellen Domainnamen. Alles, was wir woanders als `INTERNAL_HOSTS::` deklarieren (später mehr), bekommt die Pseudodomain `br.de`.

```
ignore:
  any::
    CVS
```

Und Daten, die in irgendwelchen Verzeichnissen mit dem Namen `CVS` liegen, möchten wir auf den Zielhosts lieber nicht sehen. Genauso gut ließe sich hier verhindern, dass aus Versehen Dateien kopiert werden, die Passwörter enthalten oder den Konfigurationsserver aus anderen Gründen nicht verlassen sollten.

```
import:
  any::
    cf.groups
  linux::
    cf.linux
  solaris::
    cf.solaris
```

Hier werden noch ein paar Definitionen nachgeladen - so z.B. unterschiedliche Konfigurationen für Linux und für Solaris. `cf.linux` selber inkludiert wiederum andere Dateien, in denen definiert wird, was in `copy`, `shellcommands`, `files` usw. ausgeführt wird. Was in `cf.groups` steht, gilt für alle Rechner. Hier werden unsere eigenen Rechnerklassen (`webserver`, `ntpserver` usw.) definiert.

`linux::` und `solaris::` sind interne Klassen von cfengine, diese müssen vorher nicht definiert werden, da cfengine selbst in der Lage ist zu erkennen, auf welchem Betriebssystem es läuft. Selbst innerhalb von `linux::` oder `solaris::` sind noch automatisierte Unterscheidungen möglich, so erkennt cfengine SuSE, RedHat und Mandrake (und andere), innerhalb der Distributionen sogar einzelne Versionsnummern.

Hands on!

Na gut, noch nicht ganz - vorher sollten wir noch ein paar Bemerkungen zum Klassensystem von cfengine machen. Eine Aktion in cfengine sieht wie folgt aus:

```
action-type:              # z.B. copy, resolve, shellcommands
      class::             # Die Klassendefinition
            definition    # was passiert hier?
```

Klassen können verschiedene Dinge sein, so z.B. der Name einer Betriebssystemarchitektur wie `sun4` oder `linux`, der unqualifizierte Name eines hosts (wenn der Resolver qualifizierte Hostnamen auswirft, so werden sie von cfengine in die unqualifizierte Form überführt), der Name einer selbstdefinierten Gruppe von Rechnern, aber auch der Name des Wochentages, die Stunde, Viertelstunden innerhalb einer Stunde, Monatsnamen oder -Tage und natürlich auch IP-Adressen der Form `ipv4_192_0_0`. Klassen können miteinander kombiniert werden - entweder mit OR ('|') oder AND ('.', bzw '&' in neueren Versionen).

So werden Klassendefinitionen der Form `linux.ipv4_195::` nur dann beachtet, wenn der ausführende Rechner sowohl Linuxrechner ist als auch im IP-Bereich `195.0.0.0/8` steht, eine Klasse der Form `linux|solaris` dann, wenn der Rechner entweder ein Linuxrechner oder ein Solarisrechner ist.
Kleines Beispiel: Würden wir jeden Freitag zwischen 10 Uhr und 12 Uhr Wartungsarbeiten durchführen und das den Betrachtern unserer Webseite durch eine spezielle index.html bekannt machen wollen, sähe das im Pseudocode so aus:

```
copy:
```

```
webserver.Friday.(Hr10|Hr11)::
       copy index.html.Wartung $(webroot)/index.html
webserver.!Friday|(!Hr10|!Hr11)::
       copy index.html $(webroot)/index.html
```

Der copy-Befehl sieht natürlich im endgültigen Code etwas anders aus. webserver ist hier eine selbstdefinierte Klasse, in der alle Webserver enthalten sind, die nach außen sichtbar sind, der Rest kommt aus cfengine. Von der Reihenfolge her: Der NOT-Operator bindet am stärksten, dann kommt AND und OR. Alles in () wird präferiert ausgeführt.

Die selbstdefinierte Klasse webserver sieht wie folgt aus:

```
webserver = ( web-1 web-2 web-3 www1 www2 www3 )
```

Es wären also sechs Rechner von dieser Kopieraktion betroffen.

Ein weiteres Beispiel für das Arbeiten mit Klassen: Wir haben jeweils im internen und im externen Netz einen NTP-Server und möchten für alle Rechner die gleiche ntp.conf ausspielen - außer natürlich für die Server, welche eine andere benötigen. Zuerst definieren wir also die Klasse ntpserver:

```
ntpserver = ( www1 www2 auskunft-1 auskunft-2 )
```

Auch hier stehen in der Klasse nur unqualifizierte Namen, obwohl `www1` und `www2` in einer anderen Domain stehen als `auskunft-1` und `auskunft-2`.

Die ntp.conf wird bei uns auf *alle* Rechner ausgespielt:

```
copy:               # action type
       any::        # class - hier *alle* Rechner
              $(files_directory)/any/ dest=/ server=$(config_server)
               recurse=inf trustkey=true encrypt=true verify=true
```

Die letzten beiden Zeilen sind der Kopierbefehl - `$(files_directory)/any` (wird in der `cfagent.conf` definiert) ist das Herkunftsverzeichnis, es wird nach / auf dem Zielserver ausgespielt - und die Datei kommt vom `$(config_server)`. In der letzten Zeile gebe ich an, dass er rekursiv alles unterhalb von `any/` kopieren soll - und zwar nur, wenn für den Zielrechner der trustkey stimmt. Das alles verschlüsselt und nach der Kopieraktion noch einmal verifiziert.

Liegt jetzt unter `any/etc/` eine Datei `ntp.conf`, dann hätten wir natürlich ein Problem - die Unterscheidung zwischen Client und Server schlägt Fehl.

Aber:

```
[ralph@logout files]$ ls -1 any/etc/ntp.*
any/etc/ntp.conf.client
any/etc/ntp.conf.server
```

Es werden also zwei Dateien kopiert. Jetzt kommt die in der cfagent in der actionsequence definierte link-Sektion ins Spiel:

```
links:
       any.!ntpserver::
              /etc/ntp.conf -> /etc/ntp.conf.client
       ntpserver::
              /etc/ntp.conf -> /etc/ntp.conf.server
```

In Worten: Gehört der Rechner zur Klasse any - das ist jeder Rechner - aber nicht zur Klasse ntpserver - wird die ntp.conf.client nach ntp.conf gelinkt.

Gehört der Rechner zur Klasse ntpserver, dann wird der Link auf die Serverkonfiguration gesetzt.

Ein paar Szenarien

Was zeichnet einen Webserver aus? Unter anderem die Tatsache, dass auf ihm eine Software läuft, die Webseiten ausliefert, wenn ein Nutzer von außen eine Anfrage mit einem Webbrowser stellt. Also stellen wir doch sicher, dass sich auf jedem Webserver eine solche Software befindet.

Dafür kennt cfengine den action type `packages`.

```
packages:
       webserver::
              httpd  elsedefine=need_httpd
              httpd cmp=gt version=2.0.46-43 elsedefine=update_httpd
```

Da wir in der `cfagent.conf` `rpm` als `DefaultPkgMgr` vorgeben passiert folgendes: Beim Aufruf von cfagent wird per rpm nachgeguckt, ob das Paket `httpd` installiert ist. Ist es das, wird noch verglichen, ob die Version größer als 2.0.46-43 ist (cmp=gt bedeutet greater than).

Ist also kein `httpd` installiert, wird die Klasse `need_httpd` definiert, ansonsten die Klasse `update_httpd` (`httpd` ist vorhanden, aber in einer falschen Version).

Um die Variable jetzt auswerten zu können, definieren wir uns noch ein passendes Shellcommand hinzu:

```
shellcommand:
       webserver.need_httpd::
              $(yum_install) httpd      # ist in der cfagent.conf deklariert.
              /etc/init.d/httpd restart # Natürlich müssen wir den httpd dann
                                        #auch neu starten
       webserver.update_httpd::
              $(yum_update) httpd
              /etc/init.d/httpd restart
```

Wissen wir jetzt, dass eine neue Version des `httpd` herausgekommen ist, wird einfach die Versionsnummer in der `packages`-Sektion erhöht - und beim nächsten Lauf von cfengine haben wir eine neue Version des `httpd` auf unseren Webservern.

Sicherlich ist ein Update der Software nicht der einzige Zeitpunkt, zu dem man den httpd-Prozess neu starten möchte, bei Ressourcenkonflikten könnte das auch nötig sein. Laufen also zu wenige httpds (sprich: keiner) oder zu viele (nehmen wir an, der Rechner verträgt nicht mehr als 256 Prozesse dieser Art), dann wollen wir den httpd ebenfalls neu starten.

```
processes:
       webserver::
              "httpd" restart "/etc/init.d/httpd restart" matches>1
              signal=sigterm
              action=signal
              "httpd" restart "/etc/init.d/httpd restart" matches<256
              signal=sigterm action=signal
```

Wer sich jetzt über die Zahlenwerte wundert - ja, das ist etwas missverständlich: Hier werden Ober- und Untergrenzen definiert. Das >1 bedeutet, dass mindestens ein Prozess laufen muss, das <256 bedeutet, dass das die absolute Obergrenze ist. Ist also weniger als ein Prozess oder sind mehr als 256 Prozesse vorhanden, dann wird den Prozessen mit Namen "httpd" ein SIGTERM geschickt - und `/etc/init.d/httpd restart` ausgeführt.

Durch geschickte Klassendefinitionen könnte man durch diesen Mechanismus auch sicherstellen, dass auf bestimmten Rechnern nur im Wartungsfenster ein ssh-Daemon läuft - oder dass das P2P-Tool, welches der Nutzer fool dauernd startet direkt wieder beendet wird.

Als letztes Beispiel noch etwas aus dem täglichen Betrieb, diesmal unser Nameserver. Ändert sich auf unserem primären Nameserver die Konfiguration, weil z.B. eine Zone hinzugekommen ist oder eine Zone wegfällt, muss `named` via `rndc` darüber informiert werden.

Natürlich besteht die Möglichkeit, bei jedem Lauf von cfagent einfach ein `rndc reload` ausführen, bei der Menge an Daten, die unser `named` kennt, wäre das auch kein großes Problem. Aber elegant ist das nicht. Also wird eine Möglichkeit gesucht, diesen Befehlt nur auszuführen wenn sich die Datei `named.conf` ändert

In der Klasse `nameserver` stehen unsere Nameserver:

```
nameserver = ( dns1 dns2 )

copy:
       nameserver::
              $(files_directory)/sw/nameserver/var/named/chroot/etc/named.conf
              dest=/var/named/chroot/etc/named.conf server=$(config_server)
                     mode=664 owner=named group=named type=sum action=fix
                     define=reload_named

shellcommands:
       reload_named::
              "/usr/sbin/rndc reload"
```

In der `copy`-Sektion findet sich ein `type=sum` - cfengine vergleicht hier die auf dem Konfigurationsserver liegende `named.conf` via md5sum mit der auf dem Produktivserver liegenden. Gibt es einen Unterschied, dann wird diese Datei kopiert. Und auch erst dann wird die Klasse `reload_named` definiert.

Trifft obiges zu, dann wird also zuerst die `named.conf` auf die Produktivserver ausgespielt und in dem Moment, wo in der `cfagent.conf` die `shellcommand`-Sektion angesprochen wird, stellt cfengine fest, dass `reload_named` als Klasse existiert. Dadurch wird dann auf dem Produktivsystem der Befehl `/usr/sbin/rndc reload` ausgeführt - named liest seine Konfiguration neu ein.

Genau so verfahren wir bei uns auch mit Konfigurationsdateien vom Apache-Webserver, ändert sich also eine Datei unterhalb von `/etc/httpd/conf` bzw. `conf.d/`, wird `/etc/init.d/httpd reload` ausgeführt.

Ausblick

Bis jetzt sind wir mit der Migration von Alice auf cfengine recht zufrieden, vor allem nachdem wir die ersten Klippen umschifft haben. Es ist zwar recht komplex, Aktionen zu definieren, aber das normale Ausspielen von Konfigurationsdateien ist wirklich einfacher geworden.

Man muss allerdings dazu sagen, dass wir bei weitem noch nicht alle Rechner migriert haben, die Verzeichnisstruktur also noch komplexer wird - und eventuell finden wir dabei auch Systeme, die sich nicht einfach in cfengine eingliedern lassen.

Weiterhin steht noch keiner unserer SUN-Server in cfengine - auch hier wird sich noch ein weites Betätigungsfeld finden lassen.

Und die Frage, die sich vor allem stellt: Wo wollen wir mit cfengine hin? Was fehlt noch vollständig?

Wo wir hinwollen ist klar: Wir möchten auf jeden Fall auf der Seite der Linuxrechner die komplette Konfiguration in cfengine vorhalten (mit begleitenden Kickstart-Konfigurationen), um im Fall der Fälle innerhalb weniger Minuten einen Ersatzrechner automatisiert hochziehen zu können. Die Komplettkonfiguration via cfengine macht auch das Backup der Konfiguration einzelner Rechner einfacher, da nur das "cfengine-Verzeichnis" des Konfigurationsservers ins Backup aufgenommen

werden muss - und zusätzlich jeder, der mit cfengine arbeitet zumindest einen halbwegs aktuellen CVS-Stand auf seiner Workstation liegen hat.

Was noch vollständig fehlt: *Patchmanagement*.

cfengine nimmt uns nicht die Aufgabe ab, die Software auf unseren Rechnern up to date zu halten. Wie wir gesehen haben, ist das mit cfengine möglich, aber es wird schwierig für jeden Rechner alle installierten Pakete in cfengine mitzuziehen und dann vor allem Versionsänderungen nachzuführen.

Auf der anderen Seite wollen wir natürlich nicht, dass sich Rechner via cronjob selbst updaten, obwohl yum natürlich die Möglichkeit dazu bietet. Zwei Ideen sind vorhanden - beide sind ohne Eigenarbeit aber nicht einfach in cfengine zu integrieren. Die erste wäre es, dass jeder Rechner selber schaut, ob Updates für ihn vorhanden sind, cfengine diese Dateien dann zentral auf den Konfigurationsserver zieht und der Admin diese Dateien passend editiert. Aus diesen Dateien werden dann Listen für die einzelnen Rechner generiert - und vom cfagent wird ein yum update gestartet.

Die zweite Idee ist es, das packages-Feature von cfengine auszunutzen. natürlich kann niemand die Paketdateien aller Hosts von Hand pflegen, hier müsste eine Möglichkeit der automatischen Erstellung dieser Dateien gefunden werden - was die händische Pflege von cfengine wieder erschwert.

Fazit

cfengine ist ein sehr mächtiges Werkzeug zur Verwaltung einer großen Anzahl von Rechnern. Dort besteht leider auch das größte Problem. Es gibt kaum Einstiegsliteratur zu cfengine, genau wie ein grober Überblick über die Arbeitsweise von cfengine fehlt.

Wer also cfengine in seinem Bereich nutzen will, sollte sich auf eine längere Einarbeitungsphase einrichten, da vieles unter cfengine anders funktioniert, als man es von seiner eigenen Erfahrung aus erwarten würde – hier zeigt sich der akademische Einschlag.

Hat man die Einstiegshürden gemeistert, dann gibt einem cfengine viele flexible Verwaltungstools an die Hand, die eine zentrale Rechnerverwaltung wirklich vereinfachen können. Seine großen Vorteile spielt cfengine dort aus, wo viele sehr ähnliche Rechner benötigt werden, PC-Pools einer Hochschule oder Arbeitsplatzrechner in einer Firma oder Behörde seien als Beispiel genannt. Aber auch die Verwaltung eines nicht homogenen Rechenzentrums, wie es bei uns steht, wird mit cfengine vereinfacht, sobald jeweils ein paar Rechner gleiche oder ähnliche Konfigurationen haben. Ein weiterer Vorteil der zentralen Verwaltung von Konfigurationsdateien ist das einfach Ausspielen z.B. eines weiteren Webservers oder das Neuaufsetzen eines gestorbenen Rechners.

Ab einer gewissen Anzahl von verwalteten Rechnern (oder Daten) sollte man sich aber überlegen, ob man cfengine nur noch zur Ausspielung nutzt und die Daten nicht eventuell automatisiert erzeugt – dann allerdings ist es für den Administrator schwierig, die Dateien noch von Hand zu pflegen bzw. zu verstehen.

Auch mit dem mittlerweile bei uns vorhandenen Wissen würden wir erneut cfengine nutzen, wenn die Wahl bestünde, alleine Aufgrund der Anzahl der von uns verwalteten Rechner. Sind allerdings nur ein paar Rechner vorhanden, sollte man sich wirklich überlegen, ob man dieses Tool einsetzen will – oder ob man die Zeit der Einarbeitungsphase nicht dazu nutzen will, sich eine selbstgebaute Lösung via Perl oder Shell zu stricken.

Link/References

http://www.cfengine.org/
cfengine Homepage
http://www.onlamp.com/pub/a/onlamp/2004/04/15/
cfengine.html Einführung in cfengine Teil 1
http://www.onlamp.com/pub/a/onlamp/2004/05/13/distributed_cfengine.html
Einführung in cfengine Teil 2
http://www.cfwiki.org/
User Wiki von cfengine
http://lists.gnu.org/mailman/listinfo/help-cfengine
cfengine Mailinglist
http://www.tu-chemnitz.de/urz/kurse/unterlagen/cfengine/
cfengine Einführung (deutsch)

Über die Autoren

Ralph Angenendt: Geboren 1969, beschäftigt sich seit 1995 mit Linux und anderen Unixderivaten. Arbeitet seit 2002 als Systemadministrator für den Bayerischen Rundfunk in München. Momentaner Arbeitsschwerpunkt ist die Migration der Infrastruktur von SuSE auf CentOS und der Einführung von cfengine als Werkzeug für das Konfigurations- und Patchmanagement.

Ulrich Habel: Geboren 1971, beschäftigt sich seit 1995 mit BSD und anderen Unixderivaten. Er ist seit 2002 in Projekten des Bayerischen Rundfunks beschäftigt und seit 2004 als Systemadministrator für den Bayerischen Rundfunk tätig. Sein aktueller Arbeitsschwerpunkt ist die Weiterentwicklung und Migration des Content Management Systems.