BACHELOROPPGAVE

Automatisering av linuxtjenere i Nasjonalbiblioteket

-Bachelorprosjekt

Utarbeidet av:

Henning Fjellheim

Studium:

Bachelor i informasjonssystemer

Innlevert:

Vår 2015

www.hinesna.no

Forord

Prosjektet er en gjennomgang av fordeler og ulemper ved innføring av automatisering av linuxtjenere i Nasjonalbiblioteket. Det har blitt utført av en student ved studieretningen informasjonssystemer ved Høgskolen i Nesna og er utformet sammen med Nasjonalbiblioteket, som er oppdragsgiver. Utarbeidelsen av denne rapporten innebar variert arbeid og har vært svært interessant og utfordrende både faglig og tidsmessig.

Jeg ønsker å takke veiledere Rune Bostad og Øyvind Hanssen for oppfølging og oppmuntring. 0ppdragsgiver Sverre Bang og Marianne Drotninghaug for deres hjelp og tilbakemeldinger underveis.

____

Henning Fjellheim

Innholdsfortegnelse

FORORD I

INNHOLDSFORTEGNELSE II

LISTE OVER FIGURER IV

1 INNLEDNING 1
1.1 BAKGRUNNEN FOR PROSJEKTET. 1
1.2 PROBLEMSTILLING 1
1.3 BESKRIVELSE AV NASJONALBIBLIOTEKET. 2
1.4 PROSJEKTMÅL 2
1.5 OPPGAVEBESKRIVELSE 3
1.6 SAMMENLIGNINGSKRITERIER FOR AUTOMATISERINGSVERKTØY. 4
1.7 OMKRINGLIGGENDE INFRASTRUKTUR 4

2 OPPSETT AV KJØREMILJØ 5
2.1 KOMPONENTER I KJØREMILJØET. 5
2.2 HVORDAN KJØREMILJØET ER SATT OPP. 5

3 OPPSETT AV TESTER I NAGIOS 8

4 INSTALLASJON AV VIRTUELLE LINUX-TJENERE 10

5 GJENNOMGANG AV CFENGINE 12
5.1 INSTALLASJON AV CFENGINE TJENER 12
5.2 OPPSETT AV INSTALLASJON FOR KLIENTER 13
5.2.1 Antall linjer kode endret mellom Centos 6.6 og 7 13
5.3 FORDELER MED CFENGINE 14
5.4 ULEMPER MED CFENGINE 14
5.5 HVORDAN MØTER CFENGINE RISIKO NEVNT UNDER 9.3 15

6 GJENNOMGANG AV PUPPET 17
6.1 INSTALLASJON AV PUPPET TJENER 17
6.2 INSTALLASJON AV PUPPET PÅ KLIENTHOSTER 17
6.3 ANTALL LINJER KODE ENDRET MELLOM CENTOS 6.6 OG 7 19
6.4 FORDELER MED PUPPET 20
6.5 ULEMPER MED PUPPET 20
6.6 HVORDAN MØTER PUPPET RISIKO NEVNT UNDER 9.3 21

7 GJENNOMGANG AV PATCHING 23
7.1 LÅSING AV PAKKER TIL EN SPESIFIKK VERSJON. 23
7.2 HVORFOR MAN IKKE BRUKER AUTOMATISERINGSVERKTØYET TIL SYSTEMPATCHING 24

8 AUTOMATISERT UTRULLING AV SYSTEMER. 27
8.1 HVORFOR MAN ØNSKER Å AUTOMATISERE HELE UTRULLINGSPROSESSEN 27
8.2 FORSKJELLIGE MÅTER Å UTFØRE AUTOMATISERT UTRULLING. 28

9 GEVINSTER, KOSTNADER OG RISIKO VED Å INNFØRE AUTOMATISERING 29
9.1 GEVINSTER 29

9.2 KOSTNADER 30
9.3 RISIKO 30
10 MÅLOPPNÅELSE 32
11 ANBEFALING TIL NASJONALBIBLIOTEKET 34
11.1 VALG AV VERKTØY FOR AUTOMATISERING. 34
11.2 VEIEN VIDERE 35
12 KONKLUSJON PROSJEKT 37
VEDLEGG A: REFERANSELISTE 38
VEDLEGG B: TEKNISK ORDLISTE 40
VEDLEGG C FORPROSJEKT 42
VEDLEGG D INSTALLASJONSMANUAL 62

Liste over figurer

Figur 1 Nettverksoversikt 7

Figur 2 Nagios feilmeldinger 8

Figur 3 Nagios ok meldinger 9

Figur 4 Oppretting av ny virtuell linux-tjener 11

Figur 5 Låsing mot pakkebrønn 24

Figur 6 Oppsett av systempatching 26

1 Innledning

I dette prosjektet er det igjennomgtått automatisering av linuxtjenere i Nasjonalbiblioteket. Det blir gjennomgått hvilken gevinster automatisering av linuxtjenere kan gi for Nasjonalbiblioteket.
For å finne fordeler og ulemper, er automatiseringsverktøyene Puppet og CFEngine blitt brukt. Dette er i henhold til bestillingen som kom fra Nasjonalbiblioteket (bestillingen ligger under avsnitt 1.5)

I dette prosjektet, har det blitt forsøkt å determinere hvilke av disse 2 automatiseringsverktøyene som passer best for Nasjonalbiblioteket. Det har videre blitt utarbeidet en anbefaling på hvordan Nasjonalbiblioteket burde gå frem med automatisering.

1.1 Bakgrunnen for prosjektet.

Bakgrunnen for dette prosjektet er at jeg skal utføre en bacheloroppgave ved Høgskolen i Nesna.
I forbindelse med dette, tok jeg kontakt med min arbeidsgiver med forespørsel om det var mulig å få ett bachelorprosjekt som involverte automatisering. Siden dette er noe som interesserer meg.

1.2 Problemstilling

Nasjonalbiblioteket har en stor park av linuxtjenere som i dag har flere ulike kjøremiljøer. Der flere av miljøene som skal være uniforme ikke er det. Ikke uniforme systemer har den ulempen at de kan introdusere uventede feil. For eksempel hvis man har ulik installasjon mellom testmiljø og produksjonsmiljø.

1.3 Beskrivelse av Nasjonalbiblioteket.

Nasjonalbibliotekets sin organisasjon (Nasjonalbiblioteket mandat, 2015) er forankret i lov av 9. Juni 1989 om avleveringsplikt for allment tilgjengelige dokumenter. Formålet med loven er å sikre levering av dokument med allment tilgjengelig informasjon til nasjonalbiblioteket slik at vitnesbyrd om norsk kultur og samfunnsliv blir bevart og gjort tilgjengelig som kildemateriale for forskning og dokumentasjon.

Som man kan se av det mandatet som har blitt gitt til Nasjonalbiblioteket, får og behandler Nasjonalbiblioteket ganske store mengder med data. (Film, musikk, bøker, aviser etc.) De tar vare på det som er født digitalt og det digitaliseres en stor mengde materiale som ikke er født digitalt. Nasjonalbiblioteket har derfor en maskinpark som inneholder over 100 linuxtjenere, som gjør forskjellige jobber knyttet til det å digitalisere og å vise fram materiell som er digitalisert. Dette gjøres blant annet fra tjenester som bokhylla (Bokhylla, 2015), der Nasjonalbiblioteket viser fram bøker og tjenester som er digitalisert og offentlig tilgjengelig.

1.4 Prosjektmål

Målet med dette prosjektet er å finne ut hvilke gevinster og kostnader som vil komme med innføring av automatisering for linuxtjenerne ved Nasjonalbiblioteket.

Det skal utarbeides en rapport som beskriver hvilken gevinster som kan realiseres ved å innføre automatisering av nevnte linuxtjenere. I rapporten vil det også bli beskrevet hvilke kostnader som vil komme med en eventuell innføring av automatisering.

Det vil bli forsøkt å avdekke risikoer ved innføring av automatisering og se på hvordan disse automatiseringsverktøyene møter på disse risikoene.

Det skal i prosjektet utprøves 2 automatiseringsverktøy for å besvare dette. Fordeler og ulemper med det enkelte verktøy skal beskrives.

Delmål vil være å få på plass en anbefaling om innføring av verktøy for automatisering og i så fall hvilket verktøy Nasjonalbiblioteket burde velge. Hvis det ikke lar seg gjøre å konkludere, skal det skrives en anbefaling om veien videre.

1.5 Oppgavebeskrivelse

Her er bestillingen fra Nasjonalbiblioteket. Skrevet av prosjektgruppen ved Sverre Bang og Marianne Drotninghaug.

Nasjonalbiblioteket har et stort antall Linux-tjenere i sin maskinpark. Det er knyttet betydelige utfordringer til installasjon og vedlikehold av disse. Noen av utfordringene som oppleves er:

- Tidkrevende installasjon
- Tjenere som skal være uniforme opptrer i ulike varianter av OS, patchnivå og andre konfigurasjoner
- Vedlikehold og patching er i stor grad uavhengig utført på hver av tjenerne og med varierende frekvens
- Ulik praksis fra person til person ved både installasjon, oppsett, vedlikehold og patching

Nasjonalbiblioteket ønsker svar på hvilke gevinster som kan realiseres ved å innføre automatisert installasjon og patching av Linuxtjenere. Et av virkemidlene for å besvare dette vil være utprøving og evaluering av verktøyene Puppet (Puppet hjemmeside, 2015)og CFEngine (CFEngine hjemmeside, 2015). Fordeler og ulemper med det enkelte verktøy skal beskrives.

Nasjonalbiblioteket ønsker en anbefaling om innføring av verktøy for automatisering, og i så fall hvilket verktøy som bør velges. Om det i prosjektet ikke lar seg gjøre å konkludere ønskes det en anbefaling om veien videre.

1.6 Sammenligningskriterier for automatiseringsverktøy.

Prosjektleder hadde tidlig i prosjektet ett møte med prosjektstyringsgruppen der sammenligningskriteriene for Puppet og CFEngine ble diskutert. Kriteriene for sammenligning er:

- Hvor enkelt er det å få ting ”gratis”. (ferdigskrevet kode for Puppet/CFEngine)
- Subjektiv vurdering av hvor enkelt det er å finne kodeeksempler, samt en generell vurdering av produkter. (tas i etterkant)
- Hvor mange flere linjer koder må man skrive når man skal endre fra Centos6.x til Centos7.x.
- Teste ferdigskrevene moduler mot det som skal utføres. (Tas mot slutten av prosjektet).

1.7 Omkringliggende infrastruktur

Prosjektleder hadde tidlig i prosjektet ett møte med prosjektstyringsgruppen der det ble sagt hva som skulle installeres for å sjekke ut automatisering med Puppet og CFEngine, samt at det ble lagt en skisse for hva som skulle installeres av omkringliggende struktur

Omkringliggende infrastruktur:

- Virtualiseringsserver, Ovirt.
- IPAserver.
- DHCP server.
- Server for utdeling av image.
- OMD server for testing.

Hva skal settes opp med Puppet/CFEngine:

- Linux-tjener skal meldes inn i IPA.
- Det skal settes opp en enkel webside.
- Brannveggen på linux-tjeneren skal slås av.
- SELinux skal slås av.
- Nagiostester skal installeres

2 Oppsett av kjøremiljø

Beskrivelse av kjøremiljøet der linux-tjenerne blir installert for uttesting av automatisering.

2.1 Komponenter i kjøremiljøet.

Kjøremiljøet inneholder flere forskjellige komponenter som trengs for å utføre utprøvingen av automatiseringen, disse er:

DHCP Server: Det blir satt opp en router som sørger for å dele ut ip til linux-tjenerne via DHCP.

Virtualiseringsplattform: oVirt (oVirt, 2015) er valgt. Grunnen til dette er at Nasjonalbiblioteket bruker oVirt som virtualiseringsløsning, slik at erfaring som blir knytter opp mot oVirt forhåpentligvis kan brukes ved Nasjonalbiblioteket etter at bachelorprosjektet er ferdig.

Virtuell linux-tjener for utrulling av OS: Det blir satt opp en linux-tjener med Centos 7. Denne linux-tjeneren vil ha en enkel webserver som skal hoste installasjonspakker. Disse pakkene skal brukes til å automatisere installasjonen av nye linux-tjenere. Mer om dette under avsnitt 2.2

IPA tjener: Det blir satt opp en virtuell linux-tjener med Centos 7, som vil bli satt opp med freeIPA for å ha en sentral brukerstyring.

Testing/overvåkning: Det vil bli satt opp en virtuell linux-tjener med Centos 7 der Nagios blir installert, for å sjekke at CFEngine/Puppet sine automatiseringsprosedyrer utfører de målene som er satt.

2.2 Hvordan kjøremiljøet er satt opp.

DHCP server: DHCP serveren er satt opp på en standard linksys router. Denne er satt opp med ipadressen 10.0.1.1 og deler ut DHCP adresser fra 10.0.1.100 til 10.0.1.200. Det er valgt å bruke private ipadresser av ett par grunner. Det er ikke er tilgjengelig mer enn en offentlig ipadresse på det nettverket som dette prosjektet står på. En annen grunn er av sikkerhetshensyn. En del av oppgaven i prosjektet går ut på å slå av brannmuren og deaktivere en av sikkerhetsinnretningene. Det var ikke ønskelig å ha disse linux-tjenerne koblet rett mot internett.

Virtualiseringsplattform: Det virtuelle miljøet blir kjørt på en IBM X3100 m4 server med 32Gb med ram, de virtuelle linux-tjenerne blir lagret på en ekstern NAS som er koblet til virtualiseringsplattformen. Her er det lagt inn en standardinstallasjon med Centos 7 som oVirt er installert på. Den får ipadressene fra DHCP serveren. På denne serveren vil alle de virtuelle linux-tjenerne kjøres.
Denne serveren er satt opp som en såkalt self hosted engine. Siden oVirt egentlig består av 2 deler, en virtualiseringsmanager som holder styr på alt og noder, der de virtuelle linux-tjenerne kjøres. Ovirt hosted betyr at man kjører virtualiseringsmanageren på selve noden, slik at man slipper å ha 2 fysiske maskiner for å kjøre oVirt.

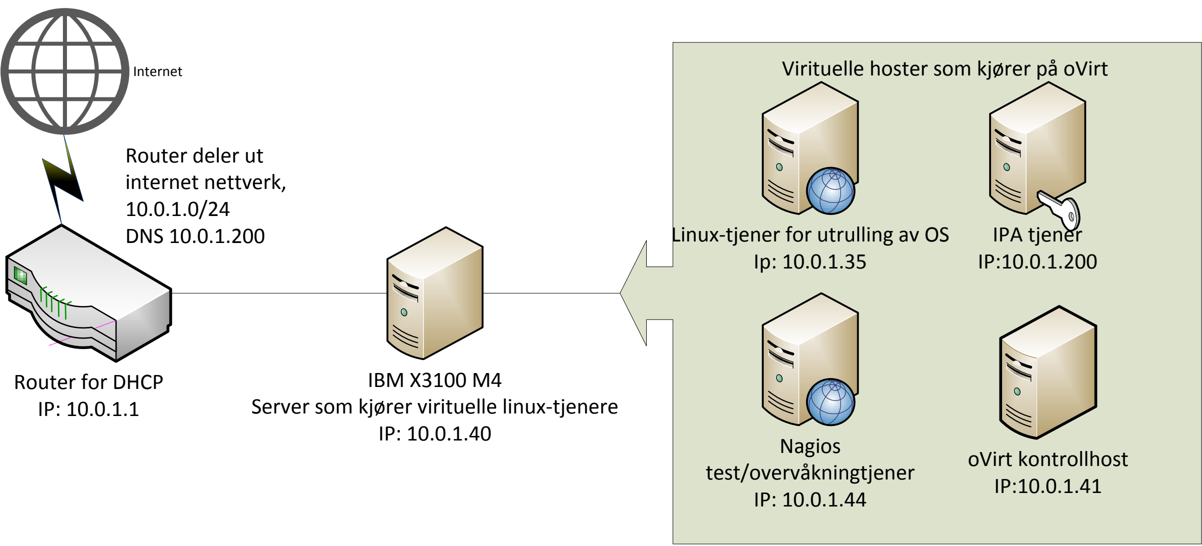
Det er ett par grunner til at det ble valgt å kjøre dette prosjektet på ett virtuelt miljø. Disse grunnene er : Man vet at linux-tjeneren blir lik hver gang man oppretter den, men den viktigste grunnen er at det er tidkrevende å lete opp 6 fysiske maskiner, samt at det plassmessig er vanskelig å finne plass til 6 maskiner til testoppsettet.

Virtuell linux-tjener for utrulling av os: Denne linux-tjeneren er installert på oVirt. Dette er en standard Centos 7 linux-tjener som det er installert apache på. Deretter er installasjonsmediet til Centos 6 og Centos 7 lastet ned til denne linux-tjeneren, før det har blitt pakket ut og gjort tilgjengelig via webtreet. Kickstartfilene som brukes til automatisk utrullingen av nye linux-tjenere, ligger på denne linux-tjeneren. Mer om dette under avsnitt 4

IPA Server: Dette er en virtuell linux-tjener som kjører med Centos 7 og en standard installasjon av freeIPA. Denne linux-tjeneren fungerer også som DNS server for dette nettverket, siden alle nye linux-tjenere som blir satt opp blir meldt inn i IPA. De registrerer seg i domenet slik at de andre linux-tjenerne kan gjøre DNS og reverse DNS oppslag mot disse linux-tjenerne. Man kan da bruke både shortname og FQDN for å finne linux-tjenerne som er meldt inn i IPA.

Testing/overvåkning: Her er det satt opp en linux-tjener med Centos 7 som det er installert Nagios på. Denne linux-tjeneren brukes for å kjøre tester mot linux-tjenerne som installeres i dette prosjektet, slik at det kan verifiseres at alt fungerer slik som det er tiltenkt. Mer om dette i oppsett av tester i Nagios.

Oversikt over nettverket slik det ser ut nå.



Figur 1 Nettverksoversikt

3 Oppsett av tester i Nagios

Her er en oversikt over testene som ble skrevet i Nagios. Disse ble skrevet før selve oppskriftene for å rulle ut linux-tjenere i CFEngine og Puppet ble skrevet.

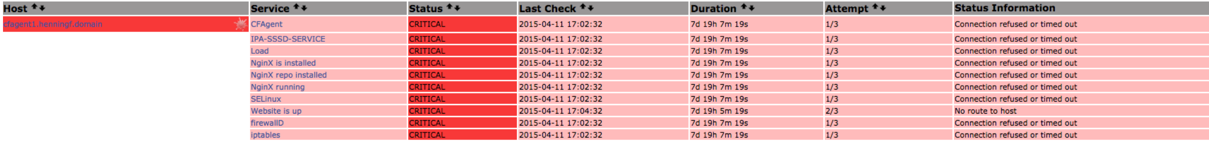
Selve koden til testene ligger som vedlegg.

Det er en del tester for de oppgavene som skulle gjennomføres. Disse testene er som følger:

- Sjekk om linux-tjeneren er online.
- Sjekke om Puppet eller CFEngine kjører.
- Sjekke at iptables eller firewalld er slått av.
- Sjekke at IPA er installert og kjører.
- Sjekke at SELinux er slått av.
- Sjekke at pakkebrønnen til NginX er installert.
- Sjekke at NginX er installert.
- Sjekke at NginX kjører.
- Sjekke at websiden som skal fremvises blir fremvist korrekt.

Etter dette, så ble testene startet , for å se at de feilet.

Her er ett skjermbilde av en linux-tjener før oppsettet ble startet



Host	Service	Status	Last Check	Duration	Attempt	Status Information
cfagent1.henningf.domain	CFAgent	CRITICAL	2015-04-11 17:02:32	7d 19h 7m 19s	1/3	Connection refused or timed out
	IPA-SSSD-SERVICE	CRITICAL	2015-04-11 17:02:32	7d 19h 7m 19s	1/3	Connection refused or timed out
	Load	CRITICAL	2015-04-11 17:02:32	7d 19h 7m 19s	1/3	Connection refused or timed out
	NginX is installed	CRITICAL	2015-04-11 17:02:32	7d 19h 7m 19s	1/3	Connection refused or timed out
	NginX repo installed	CRITICAL	2015-04-11 17:02:32	7d 19h 7m 19s	1/3	Connection refused or timed out
	NginX running	CRITICAL	2015-04-11 17:02:32	7d 19h 7m 19s	1/3	Connection refused or timed out
	SELinux	CRITICAL	2015-04-11 17:02:32	7d 19h 7m 19s	1/3	Connection refused or timed out
	Website is up	CRITICAL	2015-04-11 17:04:32	7d 19h 5m 19s	2/3	No route to host
	firewallD	CRITICAL	2015-04-11 17:02:32	7d 19h 7m 19s	1/3	Connection refused or timed out
	iptables	CRITICAL	2015-04-11 17:02:32	7d 19h 7m 19s	1/3	Connection refused or timed out

Figur 2 Nagios feilmeldinger

Som forventet så feilet alle testene. Det ble gått videre med å sette opp CFEngine og Puppet. Deretter sjekket jeg Nagios fortløpende for å se at automatiseringen ble utført korrekt og testene ble ok.

Etter at alle konfigurasjonsfilene var ferdigskrevet, ble det sjekket at alle testene ble korrekt utført. Her kan man se ett bilde av hvordan Nagios-sjekkene ser ut etter at en linux-tjener er ferdig oppsatt.

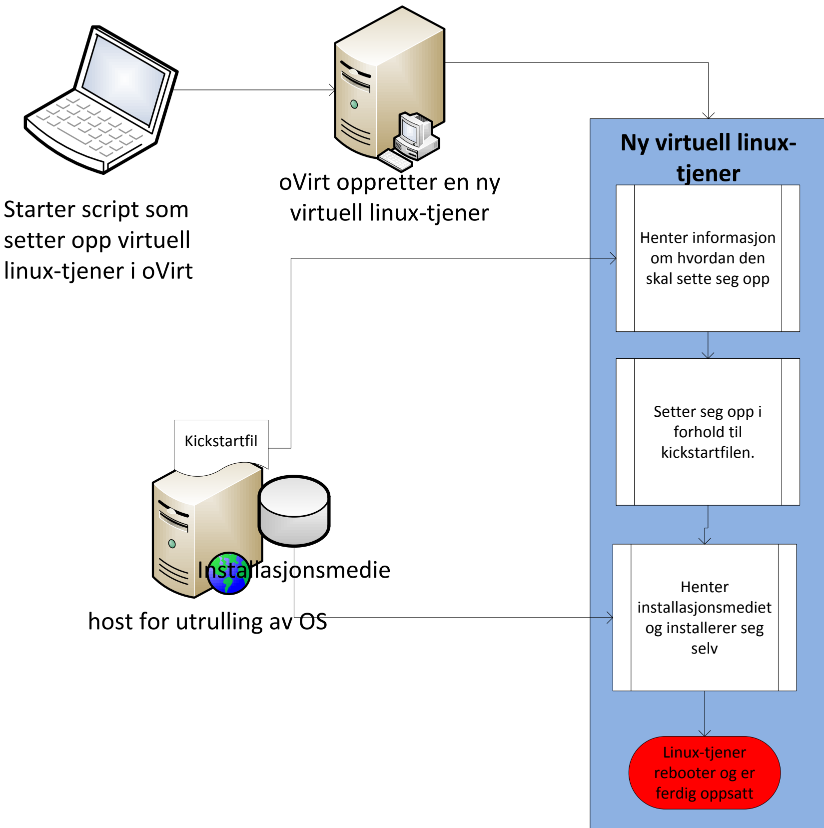
puppetclientc6.henningf.domain	IPA-SSSD-SERVICE	OK	2015-04-15 18:37:54	11d 20h 46m 35s	1/3	PROCS OK: 1 process with command name 'sssd'
	Load	OK	2015-04-15 18:42:00	13d 3h 32m 29s	1/3	OK - load average: 0.00, 0.00, 0.00
	NginX is installed	OK	2015-04-15 18:37:10	11d 19h 7m 19s	1/3	OK - /etc/nginx/nginx.conf : EXISTS ::
	NginX repo installed	OK	2015-04-15 18:37:47	11d 19h 6m 42s	1/3	OK - /etc/yum.repos.d/nginx.repo : EXISTS :: [nginx]
	NginX running	OK	2015-04-15 18:37:08	11d 19h 7m 21s	1/3	PROCS OK: 2 processes with command name 'nginx'
	Puppetagent	OK	2015-04-15 18:42:00	13d 3h 33m 21s	1/3	PROCS OK: 1 process with command name 'puppet'
	SELinux	OK	2015-04-15 18:36:46	13d 3h 32m 29s	1/3	OK - SELinux status is: Disabled
	Website is up	OK	2015-04-15 18:37:54	11d 19h 6m 35s	1/3	HTTP OK: HTTP/1.1 200 OK - 82323 bytes in 0.001 second response time
	firewallD	OK	2015-04-15 18:37:08	11d 20h 47m 21s	1/3	PROCS OK: 0 processes with command name 'firewalld'
	iptables	OK	2015-04-15 18:37:17	11d 20h 47m 12s	1/3	PROCS OK: 0 processes with command name 'iptables'

Figur 3 Nagios ok meldinger

4 Installasjon av virtuelle linux-tjenere

Tidlig i dette prosjektet ble det avdekket at det var flere linux-tjenere som skulle settes opp. Flere ganger hvis det ble avdekket problemer. Dette førte til at muligheten for å automatisere utrullingen av virtuelle linux-tjenere via script ble sjekket ut, og det ble laget ett script for å automatisk rulle ut nye linux-tjenere(scriptet ligger som vedlegg).
Scriptet kobler seg til oVirt og oppretter en ny virtuell linux-tjener. Man trenger bare å sette inn informasjon om den virtuelle linux-tjeneren, slik som: Navn på linux-tjeneren, ipadresse, hvor stor harddisk den skal ha og menge ram. Standarden som ble brukt for å rulle ut linux-tjenere i dette prosjektet var ett nettverkskort, en prosessor og en gb med ram.

Virtuelle linux-tjenere har blitt installert slik: scriptet kontakter oVirt og oppretter en ny virtuell linux-tjener, det blir så lastet en oppstartsfil fra oVirt som har blitt lastet opp til oVirt sitt filområdet. Deretter har det blitt lagt inn informasjon om at det skal kontakte linux-tjeneren som står for utrulling av operativsystem, der blir en såkalt kickstartfil (Kickstart, 2015)avhentet. Denne filen forteller hvordan linux-tjeneren skal installeres (kickstartfilen ligger som vedlegg). Hvordan harddisker skal settes opp, hvilke type pakker som skal installeres og hvilken lokasjon disse pakkene skal hentes fra. Etter at filen har blitt avhentet, starter selve installasjonen av den virtuelle linux-tjeneren etter oppskriften i kickstartfilen. Installasjonsmediet blir hentet fra utrullings-tjeneren og den nye linux-tjeneren blir installert. Etter at installasjonen er ferdig, slås den av. Dette oppdager scriptet og deretter fjerner scriptet en del innstillinger som ble satt for å utføre den automatiske installasjonen. Scriptet starter så den virtuelle linux-tjeneren og avsluttes. Den nye linux-tjeneren er nå ferdig installert.



Figur 4 Oppretting av ny virtuell linux-tjener

5 Gjennomgang av CFEngine

Her er en oversikt over hvordan det var å jobbe med CFEngine og forskjellige fordeler og ulemper som ble funnet med CFEngine.

5.1 Installasjon av CFEngine tjener

Å installere CFEngine tjeneren, var en ganske enkel oppgave. Kjør ett enkelt script fra linux-tjeneren som skulle være CFEngine tjeneren.

```
wget -O- https://s3.amazonaws.com/cfengine.packages/quick-install-cfengine-community.sh |
sudo bash
```

Når scriptet er kjørt, er det første som må gjøres å "bootstrappe" tjeneren til seg selv. Når dette blir utført på CFEngine tjeneren, forstår CFEngine automatisk at den skal sette seg opp som en CFEngine tjener og ikke en klient. Det blir gjort med denne linjen:

```
cf-agent --bootstrap <IP>
```

Det som da gjensto på tjeneren var å gjøre noen små endringer i CFEngine slik at den leser promisefilene (Informasjon om promisefiler, 2015) fra den plasseringen der det er ønskelig at den skal lese dem. I dette prosjektet er alle promisefilene lagt under ett filområde. Hvis man senere skal utvide dette til ett større miljø, er dette noe som må planlegges godt. Filområdene kan for eksempel gjenspeile kjøremiljøene eksempelvis test, development og production. Andre alternativ ville vært å splitte de forskjellige miljøene mellom forskjellige tjenere.
Ett slikt alternativ vil føre til en unødvendig høy økning av kompleksiteten av miljøet, dette grunnet at man da har flere tjenermiljø å justere inn. Vedlikehold av disse miljøene vil kreve mer av driftsavdelingen.

5.2 Oppsett av installasjon for klienter

CFEngine klientene settes opp nesten på samme måte som CFEngine tjeneren. Forskjellen er at istedenfor å bootstrappe klientene til sin egen ip, så bootstrappes den til tjeneren sin ip adresse. Dette utføres i installasjonsscriptet, slik at nå når klient(e) er ferdig oppsatt så kjører CFEngine på disse klientene.

CFEngine på klientene vil da, som nevnt i forprosjektet sjekke tjeneren om det er kommet nye promisefiler som gjelder for dem. Hvis det er kommet nye promisefiler, vil disse lastes ned og kjøres. Hvis ikke, vil klientene kjøre de promisefilene som allerede ligger på klienten.

Når man ønsker å koble en klient til en tjener, utføres dette med samme kommando som den som er nevnt under avsnitt 5.1, men <IP> må byttes ut med ipadressen til tjeneren og ikke klienten sin egen ipadresse

```
cf-agent --bootstrap <IP>
```

5.2.1 Antall linjer kode endret mellom Centos 6.6 og 7

Det varierer litt, men. ca. 3 linjer endret for hver service som skifter navn. En linje for å fortelle om det er centos6 eller centos7. Den andre for å fortelle hvilke andre tjenester som skal kjøres. Det som ble funnet ut i prosjektet var at de fleste linjene som måtte endres, var grunnet forskjeller i operativsystemene. Disse forskjellen var gjerne forskjellige navn på installasjonspakker. Hvor mange linjer man må endre kommer mer an på hvordan promisefilene blir satt opp.

På 5 forskjellige promisefiler. Så var 20 linjer endret. Altså 5 linjer pr. promisefil var endret for å få CFEngine til å fungere mellom Centos 6 og Centos 7.

5.3 Fordeler med CFEngine

Fordelene med CFEngine er at det er skalerbart og det er meget konfigurerbart. Dette forklares ved at CFEngine mener den som setter det opp, skal bestemme hvordan det skal settes opp. Dette gir en frihet når man skal designe miljøet, siden man veldig enkelt kan justere inn hvordan ting skal henge sammen.
CFEngine sine promisefiler skrives på en slik måte at man selv kan bestemme rekkefølgen på hvordan de skal kjøres, slik at man hele tiden vet rekkefølgen de utføres i.

Det er en fordel at dersom en klient mister kontakten med tjeneren, har klienten fremdeles promisefilene lagret lokalt. På denne måten kan den fortsatt kjøre promisefilene sine, selv om klienten har mistet kontakten med tjeneren.

5.4 Ulemper med CFEngine

Lærekurven på CFEngine var relativt bratt, det tok en stund før alt satt i fingrene og det begynte å bli fart når man skulle skrive konfigurasjonen for å automatisere installasjoner med CFEngine.

CFEngine lar deg konfigurere mye, så mye at det er på grensen til å være litt for mye av det gode. Etter at Puppet var blitt testet ut, føltes det som om hver minste lille ting som skulle utføres i CFEngine måtte gjøres i 3 steg: Først. Fortelle CFEngine i hvilken rekkefølge den skulle utføre promisefilen, deretter hva den skulle gjøre og til slutt hvordan den skulle gjøre dette.

Dokumentasjon: Man finner som regel informasjonen man søker, men man må gjerne lete for å finne denne. Det ble ikke funnet en samling av ferdigskrevne moduler som man kan ta i bruk når det ble søkt etter det under prosjektet. De modulene som ble funnet var av veldig varierende kvalitet. Dette fører til at man må skrive nesten alt man ønsker å gjøre på nytt og en del av forslagene som lå online under dokumentasjonen måtte justeres en god del for å få de til å fungere.

Kjøremiljø: Det er fullt mulig å sette opp flere kjøremiljø, men dette må planlegges og det ble i løpet av bachelorprosjektet ikke funnet en god måte å utføre dette på. Dette skal være bygd inn i CFEngine Enterprise, men denne ble ikke testet ut i dette prosjektet. Hovedpoenget er at det ble ikke funnet en god enkel måte å sette dette opp i gratisversjoen av CFEngine som ble testet ut i prosjektet.

Rapporter: Det er mulig å få tak i rapporter av hva som har feilet på klienter. Dette er noe som man enten må ha enterpriseversjonen, for å få eller belage seg på å sette opp selv.

5.5 Hvordan møter CFEngine risiko nevnt under 9.3

Konfigurasjonsfeil: Det er mulig med å sette opp flere miljøer i CFEngine, som nevnt så ble det i prosjektet ikke funnet en god måte å sette opp test, development og lignende miljøer i CFEngine. Oppsett av flere miljøer vil kreve planlegging.
Ett annet moment under dette er god versjonshåndtering, dette har ikke noe med selve CFEngine å gjøre direkte. Versjonskontroll (Versjonskontroll, 2015) av promisefiler gjør slik at man enkelt kan rulle tilbake til en tidligere versjon av promisefilen hvis noe skulle gå galt.

Single point of failure: Fordelen med CFEngine er at du er ikke avhengig av å ha kontakt med policytjeneren for å kjøre promisefilene. Klientene kan kjøre de policyfilene de har lokalt til policytjeneren kommer ”online” igjen. CFEngine selv sier at det ikke nødvendigvis er en fordel å ha klientene til å snakke med mer enn en tjener.

Multiple reporting hubs.
Although it is technically possible to have multiple reporting hosts, this is recommended against. Multiple hubs will only increase the overhead on all parts of the system for little return.
Our general recommendation is to introduce as few network relationships as possible (make every system as autonomous as possible). Shared storage is more of a liability than an asset in general networks, as it increases the number of possible failure modes. Backup of the hub workspace is desirable, but not a show-stopper.
(CFEngine scaling, 2015)

Automatisk oppstart av stoppede tjenester: Det er ikke nødvendigvis ett problem at automatiseringsverktøyet starter tjenester som er stoppet. Problemer oppstår først hvis dette ikke blir oppdaget. Man burde ha en plass der dette blir fanget opp. I CFEngine, vil dette være å sette opp promisefilene til å gi en beskjed til ”noe”. Dette kan være å sende en rapport til en webside, eller å sende en epost hver gang den omstarter en tjeneste. Hvis man skal stoppe tjenester som automatiseringsverktøyet starter automatisk, blir det dessverre opp til brukeren å stoppe CFEngine for å unngå dette.

Programmerings loop: CFEngine kjører sine filer fra topp til bunn (ikke helt sant, men man definerer rekkefølgen på hvordan filene skal kjøres) ergo har man ikke behov for å utføre koblinger mellom tjenester på samme måte som i Puppet. Dette problemet har ikke kommet opp under utprøving av CFEngine.

Tjeneste loop: Siden CFEngine pr. design ikke varsler hvis dette ikke er manuelt konfigurert inn, må man sette opp denne varslingen selv. Her kan man velge å bruke ett eksternt overvåkningsverktøy og la CFEngine rapportere til dette, eller man gå for Enterpriseversjonen som har denne tjenesten innebygd (CFEngine enterprice reporting, 2015). Det viktige er at man har en lokasjon som man kan sjekke, hvis det er tjenester som blir omstartet av CFEngine oftere enn man forventer.

6 Gjennomgang av Puppet

Her er en oversikt over hvordan det var å jobbe med Puppet og forskjellige fordeler og ulemper som ble funnet med Puppet.

6.1 Installasjon av Puppet tjener

Installasjonen av puppet-tjeneren var en ganske enkel oppgave. Puppet tjeneren ble installert på en Centos 7 installasjon. Det første som må utføres er å installere puppet pakkebrønnen:

```
sudo rpm -ivh http://yum.puppetlabs.com/puppetlabs-release-el-7.noarch.rpm
```

Hvis man ønsker å installere på ett Centos 6 system, vil pakkebrønnen man skal installere være:

```
sudo rpm -ivh http://yum.puppetlabs.com/puppetlabs-release-el-6.noarch.rpm
```

Deretter må man installere tjeneren via pakkebehandleren:

```
sudo yum install puppet-server
```

Den siste som må utføres er å navnet på tjeneren inn i configfilen til Puppet (configfilen ligger som vedlegg) og deretter kjøre kommandoen:

```
sudo puppet master --verbose --no-daemonize
```

Dette blir gjort for å lage sertifikater som man senere skal bruke for å ha puppet-tjeneren til å snakke med klientene. Dette vil bli forklart litt mer om dette under installasjonen av puppet på klienttjenerne.

6.2 Installasjon av Puppet på klienthoster

For å installere Puppet på klientene, installerer man først samme pakkebrønnen som man bruker for å installere Puppet tjeneren, som beskrevet under avsnitt 6.1

Deretter må man installere selve puppet klientpakken:

```
sudo yum install puppet -y
```

Puppet på klientsiden fungerer på den måten at man setter inn informasjon om Puppet tjeneren i Puppet konfigurasjonsfilen.
Etter at man har lagt til informasjonen til tjeneren i konfigurasjonsfilen, kan man koble til tjeneren ved å kjøre denne kommandoen:

```
puppet agent -t
```

Klienten vil nå forsøke å koble seg til tjeneren, med mindre man aktivt har gått inn på tjeneren og sagt at den automatisk skal godta alle nye klienter. Man må inn på tjeneren og godkjenne sertifikatforespørselen.

Inne på Puppet tjeneren så skriver man først:

```
puppet cert list
```

Man får opp de sertifikatene som ligger inne, som ønsker godkjenning. Man kan nå velge om man skal godkjenne alle:

```
puppet cert sign all
```

Hvis ikke, kan man godkjenne en forespørsel:

```
puppet cert sign <HOSTNAVN>
```

Man kan nå gå tilbake til klienten og kjøre

```
puppet agent -t
```

Nå vil klienten koble seg til tjeneren og sende informasjon om seg selv til tjeneren. Tjeneren ser igjennom den informasjonen som er sendt fra klienten og sender tilbake informasjon til klienten om hvordan den skal sette seg opp. Klienten vil sette seg opp etter den ”oppskriften” som ble sendt fra tjeneren.

6.3 Antall linjer kode endret mellom Centos 6.6 og 7

På dette punktet er Puppet og CFEngine ganske like. Man setter opp variabler for distoversjonen man skal installere og setter den opp. Det som ble oppdaget i prosjektet er, at Puppet bygger mer rundt klasser. Dette fører til at det ofte skrives mer i Puppet enn i CFEngine. I Puppet ender man oftere opp med å spre konfigurasjonen over flere filer, kontra CFEngine der man ofte skriver all konfigurasjon i en fil. Dette fører igjen til at man ofte skriver mer i Puppet, siden man tar høyde for flere eventualiteter i konfigurasjonsfilene.

Når det kommer til endringer som gjøres mellom Centos 6 og Centos 7, ble det oppdaget at måten å gjøre dette på er ganske lik.

CFEngine:

```
centos_6::
        "pakke1" string
        => "centos6pakke" ;

centos_7::
        "pakke1" string
                => "centos7pakke" ;
```

Puppet:

```
case $::operatingsystemrelease {
 /^6/: {
   $pakke1 = "centos6pakke"
 }
 /^7/: {
   $pakke1 = "centos7pakke"
 }
}
```

Som man ser, er den største forskjellen at man bruker klammeparantes når man velger operativsystem i Puppet, samt at man må skrive det inn i en case, istedenfor å skrive direkte at det skal ligge under Centos 6 eller Centos 7. Konklusjonen som ble nådd på dette punktet var at Puppet og CFEngine stiller likt når det kommer til antall endringer mellom Centos 6 og Centos 7.

6.4 Fordeler med Puppet

Puppet har en sentral styring, som betyr at alle klienter kontakter tjeneren, tjeneren lager så klar oppskriften på hvordan klienten skal være satt opp og sender den tilbake til klienten. Når klienten har utført denne operasjonen, sender klienten informasjon om seg selv til tjeneren. Slik at man her har en sentral plass, der man kan finne informasjon om klienten.

Puppet har en mengde med ferdigskrevne moduler som man kan ta i bruk, slik at man slipper å bruke tiden på å skrive alt man trenger fra bunnen av. Dette gjør at man sparer en god del tid på at man kan bruke ferdigskrevne moduler. En annen fordel med disse modulene er de har brukerscore på modulene og de har også score etter hvor godt skrevet de er. Noen av de beste modulene har Puppet selv gått god for. Man kan både sjekke ut hvordan andre som bruker modulen har rangert den, hvordan Puppet har rangert den og man kan sjekke ut hvor mange som har lastet den ned. Man har en god oversikt over hvilke moduler som er godt skrevne og mye i bruk.

GUI verktøy, Puppet har flere GUI verktøy til open source versjonen som er gratis.
For å nevne noen: Dasboard (Puppet dashboard, 2015), Foreman (TheForman hjemmeside, 2015) pluss at det er verktøy som binder sammen flere av disse enkeltstående verktøyene til en mer komplett verktøykasse for å styre miljøer. Slik som for eksempel katello (Katello hjemmeside, 2015)

Innebygd, god støtte for å sette opp flere miljøer, som eksempelvis test, development og drift. Man slipper å bruke tid på å tenke dette ut selv, man kan velge hvilket miljø klienten skal være på. Enten ved å sette dette på klienten (Dette er en innstilling man kan sette i konfigurasjonsfilen til puppetklienten). Eller man kan bruke External Node Classifier (ENC, 2015)

6.5 Ulemper med Puppet

Puppet har en sentral styring, samtidig som dette er en fordel, så er det også en ulempe. Hvis noe skulle gjøre slik at en klient ikke får kontakt med tjeneren, får ikke den klienten utført noe av det som Puppet mener den skal gjøre. I motsetning til CFEngine der tjeneren har all informasjon om hva den skal utføre lagret lokalt på klienten, spør klienten tjeneren etter denne informasjonen hver gang den kjøres. Tjeneren kompilerer en konfigurasjonsfil og sender dette tilbake til klienten.

Å skrive enkle ting i Puppet som for eksempel å installere ett program på en linux-tjener går fort og er enkelt. Men for å bruke Puppet til litt større ting, så burde man skrive moduler.

Det ligger heldigvis gode "oppskrifter" på hvordan det er anbefalt å skrive moduler (Puppetmoduler referanse, 2015) men man bruker litt tid før man får dette inn i fingrene. Selve syntaksen som man bruker i Puppet føles enklere enn det som brukes i CFEngine, men å skrive en god modul er gjerne like vanskelig som å skrive en god promisefil.

Puppet er for det meste deklarativt, noe som i begynnelsen kan være litt vanskelig å forstå. Dette fordi Puppet kjører igjennom klassene i den rekkefølgen den ønsker. Puppet sin tankegang er at istedenfor en serie med steg som skal følges, definerer man en ønsket sluttstatus. Puppet antar at ressurser ikke henger sammen. De vil ikke bli lagt til systemet i den rekkefølgen de er skrevet. Hvis ressurser henger sammen og en ressurs er avhengig av en annen, må dette defineres eksplisitt.
Dette er gjerne det første "problemet" personer som er vant med imperative språk møter. Er man vant med å skrive i Python, Perl, Java etc, så er man vant med at et program blir lest fra toppen til bunnen og ikke i tilfeldig rekkefølge. Som beskrevet står det at Puppet for det meste er deklarativt og med det menes at noen ting må kjøres i rekkefølge, man må for eksempel sette variabler før man kan referere til den.

6.6 Hvordan møter Puppet risiko nevnt under 9.3

Konfigurasjonsfeil kan få store konsekvenser: Puppet har ett godt oppsett for å bruke flere miljø. Måten å unngå å få inn konfigurasjonsfeil på vil være å sette opp ett testmiljø der man tester ut nye puppetklasser, før man setter disse inn i produksjon. Som nevnt i CFEngine, burde man ha versjonskontroll av konfigurasjonen. Dette for at man enkelt kan rulle tilbake til forrige fungerende versjon hvis en feil skulle oppstå.
Uttesting av klasser i ett testmiljø før det rulles ut i produksjon er en god praksis for å sjekke etter feil som man kan ha oversett når automatiseringsscriptene ble oppdatert.

Single point of failure: Puppet har en del måter å løse dette problemet på. Man har mulighet til å sette opp flere puppetmaster tjenere. Problemet hvis man har en puppetmaster er at man verken får kjørt oppdateringer av klientkonfigurasjoner, eller klientkonfigurasjoner som allerede er implementert. Dette har med Puppet sin design å gjøre. Puppet klienten kontakter Puppet tjeneren, som igjen vil kompilere hvordan klienten skal sette seg opp. Hvis klienten

ikke får kontakt med tjeneren, så vil den aldri få beskjed om hvordan den skal sette seg opp. Det er mulig å sette opp en tjeneste foran Puppet tjeneren som sørger for at failover, dette kan være eksempelvis NginX eller Apache.

Så det er mulig å konfigurere seg bort fra dette problemet (en annen løsning som blir foreslått av Puppet kan være å ha 2 tjenere og endre i DNS'en hvis en av disse 2 tjenerne skulle feile)

Automatisk oppstart av stoppede tjenester: Som nevnt under CFEngine, er ikke dette ett problem hvis det blir oppdaget. Løsningen på dette under Puppet (siden klientene sender informasjon om både seg selv og hvordan det gikk når konfigurasjonen ble kjørt tilbake til tjeneren) er å la overvåkningen av dette bli satt til tjeneren. Man kan blant annet bruke Katello til å sjekke dette i ett GUI, eller sette opp sin egen tjeneste (for eksempel en webside) som sender ut informasjon om hvilke tjenester som har blitt omstartet og hvor ofte. Når det kommer til tjenester som man ønsker å stoppe midlertidig, er nok det enkleste her bare stoppe automatiseringsverktøyet mens man utfører vedlikehold eller feilsøking.

Programmerings loop: Puppet bruker å kjøre sine filer i mer eller mindre tilfeldig rekkefølge som forklart. Altså er det fullt ut mulig å lage en loop imellom tjenester i Puppet. Det ble gjort ved ett par anledninger med ett uhell. Fordelen her er at Puppet oppdager dette automatisk og varsler deg. Så dette var ikke ett så stort problem i Puppet

Tjeneste loop: Puppet rapporterer hvor mange endringer den har utført for hver gang den kjører. Hvis en tjeneste blir omstartet sendes det informasjon om dette til Puppet tjeneneren. Bruker man ett GUI til Puppet, er det en enkel sak å hente ut informasjon om alle linux-tjenerne og de endringene som har blitt utført.
Man kan da enkelt se om det er en tjeneste som blir omstartet oftere enn normalt. Hvis en tjener rapporterer at den omstarter en service ofte (eller hver gang Puppet kjører), burde man sjekke ut konfigurasjonen for å avdekke om det er noen tjenester som kolliderer med hverandre.

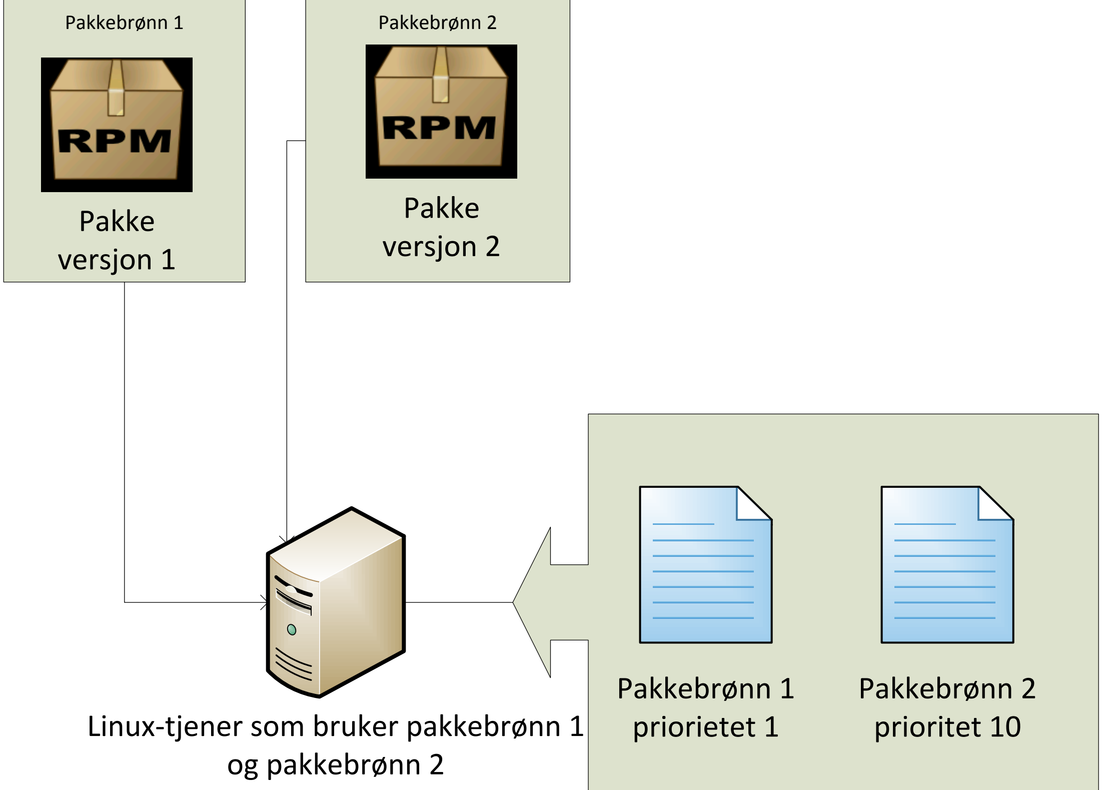
7 Gjennomgang av Patching

Det blir her forklart en del rundt patching og forskjellige aspekter ved patching.

7.1 Låsing av pakker til en spesifikk versjon.

Hvis det er en god grunn til å holde seg til en spesifikk versjon av en pakke, kan man bruke automatiseringsverktøyet til å gjøre dette. Om dette er en måte som ville vært anbefalt er mer usikkert. Måten man kan gjøre dette på er å ekskludere pakken fra å bli oppdatert (yum exclude, 2015). En annen måte kan være å sette prioritet på pakkebrønnene (yum priorities, 2015), der kan man ved hjelp av en sentralisert pakkebehandler (eller for enkelhetens skyld, sette opp sin egen pakkebrønn via f.eks en webside) sette den lokale pakkebrønnen til høyeste prioritet. Man kan da legge inn de pakkene som man ikke ønsker å oppgradere i denne pakkebrønnen. Selv om det da kommer nyere versjoner av disse pakken, vil de ikke bli oppgradert til nyere versjoner. Dette er noe som det dessverre ikke ble nok tid til å teste ut under dette prosjektet.

Her er en tenkt oversikt over hvordan dette kan brukes. Man har en klient som det er ønskelig at man ikke skal oppdatere en pakke på. Man legger denne i en lokal pakkebrønn med pakker i. Deretter setter man prioriteten på denne pakkebrønnen til å være høyere enn en annen pakkebrønn som man bruker (dette kan være fra en lokal pakkebrønn, eller det kan være en ekstern pakkebrønn man henter pakker fra)



Figur 5 Låsing mot pakkebrønn

7.2 Hvorfor man ikke bruker automatiseringsverktøyet til systempatching

Automatiseringsverktøy er gode til å automatisere installasjon og konfigurasjon av linux-tjenere.

Når det kommer til å patche hele system, blir det mye konfigurasjon som må til for å utføre dette med ett automatiseringsverktøy. Man ønsker jo å patche alle pakker som kommer inn, både med tanke på sikkerhet og med tanke på at man ønsker å ta i bruk nye funksjoner som kommer i de oppdaterte versjonene av programvaren(e).

Automatiseringsverktøy slik som for eksempel Puppet, har av det som har blitt testet i dette prosjektet ikke noen god måte å utføre dette på. Man kunne, i hver enkelt modul sagt hvilken versjon av pakker som er ønskelig. Man kan også si at det er ønskelig at disse pakkene skal være oppdatert til siste versjon. Men, hvis man skal ha moduler til over 100 linux-tjenere og alle disse modulene skulle inneholde informasjon om hvilke pakker som skulle til hvilket miljø. Det blir fort mye informasjon å ha oversikt over, samt at man får veldig mye manuelt arbeid med å oppdatere disse modulene, som igjen øker risikoen for å gjøre en feil i en av disse. Det andre problemet med dette er hvordan skal man oppdatere resten av systemet? Man kan jo ikke kjøre en kommando som f.eks: ”yum update –y” siden dette ville effektivt

oppdatert alle pakker på systemet, selv de pakkene som man ikke ønsker å oppdatere. (Med mindre man hadde gått inn og ekskludert alle pakkene man ikke ønsket å oppgradere)

Noe annet man kan gjøre, er å si i automatiseringsverktøyet at det automatisk skal oppdatere alle pakkene ved gitte intervaller. At man har satt opp en linux-tjener i ett test eller utviklingsmiljø, med samme oppsett som en linux-tjener som går i ett driftsmiljø. Men, man ville da måtte ha en god timeplan for når disse automatiske scriptene går. Siden man hadde måttet sjekke alle linux-tjenerne etter oppgraderingen. Hvordan skal man for eksempel kunne rulle ut en ny linux-tjener hvis en av pakkene krasjet hele systemet på en linux-tjener og dette ikke ble oppdaget før dette ble rullet ut i produksjon?

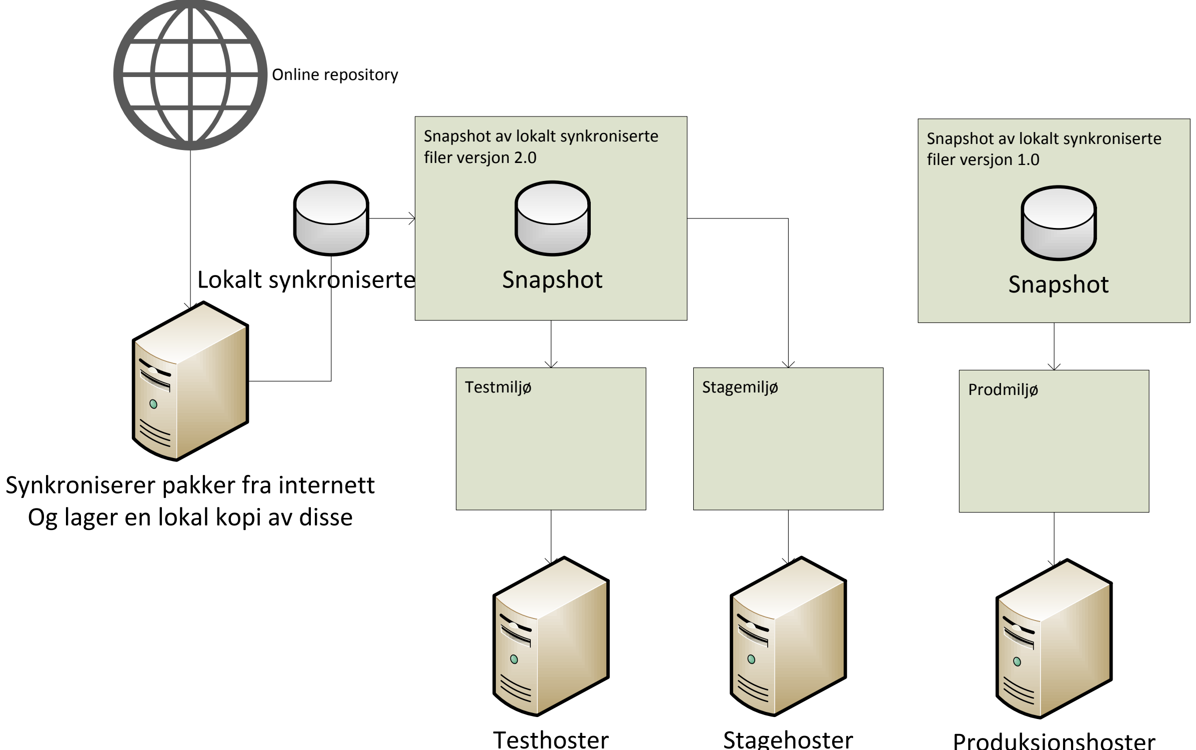
Å løse denne problemstillingen uten å bruke ett automatisert verktøy som gir deg flere muligheter til å styre pakkene som kommer inn, hvilke pakker som skal få presedens og som kan hjelpe til med å rulle tilbake til ett tidligere fungerende sett av systempakker. Uten ett slikt verktøy, vil ting bli tungvint. Man kan sette opp sine egne lokale pakkebrønner, for så å hente pakker til disse og bruke disse til oppdatering.
Dette ble ikke testet ut i dette prosjektet, da det ble funnet ett verktøy som kan brukes til dette, som holder oversikt over pakker. Dette verktøyet har ett GUI som hjelper brukeren med dette.

Heldigvis finnes slike verktøy og i dette prosjektet ble ett av disse testet ut. Det som ble testet ut følger med Katello og heter pulp (Pulp, 2015). Dette verktøyet gir muligheten til å sette opp ett miljø for en linux-tjener, eller en type linux-tjenere, der det enkelt kan synkroniseres data fra en ekstern pakkebrønn. Her kan en selv bestemme når denne dataen skal gå ut og til hvilket miljø den skal gå ut til.

Man gjør dette ved å lage noe som kalles for ett view, ett view er en samling av pakker, oppdateringer og bugfixer. Viewet kan bestå av en eller flere pakkebrønner som enten er synkronisert lokalt eller fra ett eksternt bibliotek. Når man har laget ett slikt view, kan man dytte dette til ett miljø, for eksempel stage. Finner man ut at alt går greit i stage, kan man dytte dette viewet videre til produksjon. Fordelen her er, man kan oppdatere alle pakker automatisk og hvis noen av pakkene skulle gjøre slik at funksjoner i test (eller produksjon) slutter å virke, kan man trekke det tilbake og sette tilbake det forrige viewet som fungerte. Har man linux-tjenere som er automatisk utrullet, er det enkelt å bøte på problemer med utrulling i

produksjon. Alt man trenger å gjøre er å gå tilbake til det forrige versjon av viewet som fungerte, for deretter å provisjonere ut de linux-tjenerne som fikk feil når de fikk pakkene som kom med det nye viewet. Siden de fungerte på den gamle versjonen av viewet, vil de fungere når de blir rullet ut på nytt mot de gamle pakkene. Man kan deretter gå tilbake til testmiljøet, rulle ut den nye versjonen av viewet der. Rulle ut den linux-tjeneren som feilet og fikse de problemene som oppsto med oppgraderingen. Når man har fikset de problemene, kan man rulle ut feilfiksen til produksjon før man igjen ruller ut de pakkene som forårsaket problemet i utgangspunktet.

Her er ett tenkt scenario, der man har en tidligere kopi av en pakkebrønn som kjører mot produksjonsmiljøet, mens man sjekker at det ikke er noen problemer med det siste snapshottet av dette i stagingmiljøet, her kan man se at versjon 1 kjører mot produksjonsmiljøet og versjon 2 med de nyeste pakkene kjører mot test og stagingmiljøet



Figur 6 Oppsett av systempatching

8 Automatisert utrulling av systemer.

Her vil det bli gjennomgått automatisert utrulling av systemer.

8.1 Hvorfor man ønsker å automatisere hele utrullingsprosessen.

Man ønsker å automatisere hele utrullingsprosessen av ett par grunner.

Tidsbesparelse: Når man har hele prosessen automatisert, vil man spare tid. Grunnen til dette er følgende: For å sette opp ett system, har man en plass å sette inn innstillingene. Man trykker på en knapp, etter en stund er systemet ferdig satt opp.

Minimere sannsynlighet for feil: Man har færre plasser man må inn og manuelt justere innstillinger, slik at det blir mindre sannsynlighet for at man skriver inn feil parameter, eller glemmer å sette inn parameter. Siden man med å automatisere prosessen får en plass å justere innstillinger. Så vet man at alle nye systemer får samme oppsett når de er installert.

Enklere for ikke tekniske personer å sette opp systemer: Hvis man automatiserer utrullingen av ett system og trenger flere noder på dette systemet, trenger man ikke å ha en person med dyp teknisk innsikt om dette systemet til å sette opp en ny node. Alt man trenger er basisinformasjon, man kan lage en kortere og mer lettfattelig bruksanvisning hvis det er behov for det. Man slipper å huske på alle plassene der innstillinger skal justeres. Som regel er det en del basisinnstillinger som er likt for alle linux-tjenerne med ett gitt system, da kan man finne hvilke parameter som må endres og lage en oppskrift for å endre kun disse innstillingene. Ta scenarioet som er testet her, de eneste parameterne som trengtes å endres for å rulle ut linux-tjenere: navn og ipadresse.

8.2 Forskjellige måter å utføre automatisert utrulling.

Det er flere måter å utføre automatisk utrulling av systemer og her er en oversikt over noen av dem.

Script: Kobler seg til de forskjellige komponentene via forskjellige API, for eksempel så gjorde det scriptet som ble brukt i dette prosjektet akkurat det, det koblet seg til API'et til oVirt, laget en ny linux-tjener. Startet denne, før den automatiske utrullingsprosessen startet.

Nettverk: Man setter inn parameter i DHCP som forteller linux-tjeneren hvor på nettverket de finner installasjonsfilene og hvor de finner de automatiske oppstartfilene, som sørger for oppsettet av linux-tjeneren. Man setter deretter linux-tjeneren til såkalt nettverksoppstart, der får linux-tjeneren som blir startet som ikke har ett operativsystem får informasjon fra DHCP, slik at linux-tjeneren selv henter installasjonsfilen via nettverket og starter å installere seg selv. Den henter da de resterende filene som den trenger for installasjonen fra nettverket.

Image based: Denne metoden blir gjerne brukt for utrulling av virtuelle linux-tjenere. Man har her ett image av en virtuell linux-tjener, satt opp for at man skal kunne lage mange kopier av dette. Man starter her dette virtuelle bildet, som etter at installasjonen er ferdig gjerne kobler seg opp mot for eksempel Puppet som sørger for at resten av det som trengs for å fullføre installasjonen.

9 Gevinster, kostnader og risiko ved å innføre automatisering.

Det vil her diskuteres en del rundt gevinster, kostnader og risiko man vil få ved automatisering av linuxparken i Nasjonalbiblioteket.

9.1 Gevinster

Gevinster man får med automatisering som ble funnet ut i dette prosjektet var blant annet:

Tid: Når alt var på plass for å automatisk rulle ut en linux-tjener, sparer man tid på å rulle ut en ny versjon av denne linux-tjeneren. Man slipper å bruke tid på å lete igjennom dokumentasjon for å finne ut om det var noe man hadde glemt/oversett.

Oversikt: Når man automatiserer, har man en plass der man kan se alle modulene som er brukt for å sette opp en linux-tjener.

Enklere å oppdatere til en nyere versjon av software: Når man har på plass en oppskrift som er brukt i produksjon, slipper man å bruke lang tid på å lese dokumentasjon på hvordan denne linux-tjeneren var satt opp, for å forsøke å gjenskape en eksakt lik linux-tjener i ett testmiljø. Alt man trenger å gjøre er å bruke samme oppskrift for å rulle ut en ny linux-tjener i test, deretter er det bare å oppdatere versjonen på linux-tjeneren i test og se at alt fungerer slik som det skal. Gitt at alt fungerer som tiltenkt, så er det bare å rulle ut den nye versjonen av oppskriften, for å oppdatere linux-tjeneren.

Enklere å få opp linux-tjenere som har krasjet: Her er det ett lite men. Hvis disse linux-tjenerne som krasjet, har mye på lokal disk som må synkroniseres, er det ikke dette nødvendigvis tilfelle. For eksempel en webserver, som har nesten hele sin konfigurasjon i automatiseringsverktøyet vil man kunne rulle ut en ny versjon av, helt automatisk uten å måtte tenke over hvilken versjoner av de forskjellige pakkene som trengtes til akkurat denne linux-tjeneren. Dette fordi den har all konfigurasjonen i automatiseringsverktøyet.

Homogenitet: Miljøer som blir satt opp av ett automatiseringsverktøy vil bli mer homogene enn miljøer som blir satt opp manuelt. Dette er fordi det kan være små ting som personer vil utføre samme handling på ulike måter, slik som valg av tjenester som utfører det samme. Eller det kan være konfigurasjonsfiler som 2 personer ville behandlet forskjellig. Bruker man ett automatiseringsverktøy til å utføre dette, så vet man at verktøyet vil utføre den samme jobben likt om du kjører den 2 eller 10 ganger.

9.2 Kostnader

Kostnader knyttet til å automatisere er blant annet:

Tid: Før alt er på plass, så tar det tid å automatisere alle aspektene ved en tjener, noe som er enkelt. Man kan kanskje finne en del ferdigskrevne moduler på nett. Men, en del vil kreve at man justerer inn automatiseringsverktøyet.

Support: Her har det blitt sjekket priser mellom Puppet og CFEngine.
Ved forespørsel om pris, kom CFEngine med en pris på 90€ pr. klient (prisforespørsel på 100 klienter)
Puppet tar 105$ pr. klient mellom 100 og 249 klienter.

Omregnet til NOK, blir dette 765 kroner pr. klient for CFEngine og 794 kroner pr. klient for Puppet.
Altså: 76500 og 79400 for 100 klienter, for 100 klienter, så vil Puppet være ~2900 kroner dyrere for Puppet, Puppet er marginalt dyrere.

9.3 Risiko

Det er en del risikomomenter som kommer ved innføring av automatisering.

Som nevnt under, konfigurasjonsfeil kan få mye større konsekvenser.

Single point of failure: Det ble avdekket i prosjektet at dersom man kjører med en automatiseringstjener, så vil man få ett punkt som kan feile. Hvis denne feiler vil man ikke få dyttet ut konfigurasjonsendringer, siden man bruker automatiseringsprosedyrer fra denne tjeneren.

Automatisk oppstart av stoppede tjenester: Grunnen til at dette punktet ble satt opp, er at det er mulig å sette automatiseringsverktøyet til å automatisk omstarte tjenester hvis de skulle feile. Det skumle med å gjøre dette er at det kan maskere feil i konfigurasjonen, hvis ikke overvåkningsverktøyet plukker opp de feilene som oppstår og automatiseringsverktøyet bare starter tjenesten på nytt for hver gang den feiler. Det andre som kan skje, er at før man har fullautomatisert ett system. Det kan være tjenester man ønsker å stoppe og glemmer man seg bort, så kan automatiseringsverktøyet starte tjenestene automatisk. Dette kan være uheldig.

Programmerings loop: Som de fleste programmeringsspråk, er det mulig å lage en loop. Man har ressurs 1, som kaller ressurs 2 som igjen kaller ressurs 1 etc.

Tjeneste loop: Den første ressursen slår av en tjeneste (f.eks brannveggen) Den neste ressursen slår den deretter på igjen.

Kompleksitet: Kompleksiteten vil øke når man innfører ett automatiseringsverktøy. Det fine er at man har en plass å justere konfigurasjonen sin på. Dette betyr at man må lære seg å bruke verktøyet før man skjønner hvordan dette henger sammen. Det er ikke sikkert man ser en løsning med en gang.

Konfigurasjonsfeil kan få større konsekvenser: Når man justerer ting på den enkelte klient, vil en feil gjerne være knyttet til den klienten. Justerer man på konfigurasjonsfiler som gjelder flere klienter, kanskje flere forskjellige typer klienter (for eksempel hvordan klienter skal koble seg til domenekontrolleren, som vil gjøre endringer på en ganske stor del av miljøet) og det da får sneket seg inn en feil i denne konfigurasjonen, kan man oppleve at brukere mister tilgangen. Hadde man gjort dette manuelt, så hadde denne feilen vært begrenset til en, eller gjerne ett fåtall linux-tjenere.

10 Måloppnåelse

I Dette kapittelet så blir det snakket en del om måloppnåelse for prosjektet.

Målene for dette prosjektet var:

- Å beskrive hvilke gevinster som kan realiseres ved å innføre automatisering av linuxtjenere
- Hvilke kostnader som vil komme med en eventuell innføring av automatisering.
- Forsøke å avdekke risikoer ved innføring av automatisering.
- Utprøving av 2 automatiseringsverktøy for å se hvordan de vil møte disse risikoene.
- Beskrive fordeler og ulemper ved verktøyene.
- Anbefaling om innføring av verktøy for automatisering og anbefaling om veien videre.

Beskrive gevinster:
Dette målet ble nådd, det har blitt beskrevet hvilke gevinster som kan oppnås ved å innføre mer automatisering av linuxtjenere. Dette er beskrevet mer i detalj under avsnitt 9.1

Hovedpunktene:

- Tidsbesparelse.
- Bedre oversikt over linux-tjenere.
- Enklere å rulle ut oppdateringer til alle linux-tjenere.
- Enklere systemgjenoppretting ved systemfeil.
- Større grad av homogenitet på linux-tjenere som skal være homogene.

Finne kostnader:
Dette målet ble nådd, det har blitt beskrevet hvilke kostnader som kan oppstå ved innføring av automatisering av linuxtjenere. Dette er beskrevet mer i detalj under avsnitt 9.2

Hovedpunktene:

- Tid for å sette opp alle aspekter ved automatisering.
- Vedlikeholdskostnader på verktøy.

Forsøke å avdekke risikoer:

Dette målet ble nådd, det har blitt beskrevet hvilke risikoer som man kan møte på ved innføring av automatisering av linuxtjenere. Dette er beskrevet mer i detalj under avsnitt 9.3

Hovedpunkter:

- Single point of failure. Hvis en kritisk tjeneste feiler, så kan dette gjøre slik at andre tjenester som er avhengig av denne feiler. Manglende redundans.
- Automatisk oppstart av stoppede tjenester.
- Programmerings loop.
- Tjeneste loop.
- Konfigurasjonsfeil kan få større konsekvenser.
- Økt kompleksitet.

Utprøving av 2 automatiseringsverktøy:

Dette målet ble nådd, fikk testet ut både Puppet og CFEngine. Det er beskrevet hvordan disse har møtt risikoen som er beskrevet. Dette er beskrevet mer i detalj under avsnitt 5 og 6.

Beskrive fordeler og ulemper med verktøyene:

Dette målet ble nådd, fordeler og ulemper med Puppet og CFEngine ble beskrevet. Det står mer om dette under avsnitt 5.3, 5.4, 6.4 og 6.5

Anbefaling av verktøy for automatisering og anbefaling om veien videre:

Dette målet ble nådd ved at det har blitt skrevet en anbefaling om veien videre. Denne anbefalingen ligger under avsnitt 11.

11 Anbefaling til Nasjonalbiblioteket

Dette er anbefalingen til Nasjonalbiblioteket.

11.1 Valg av verktøy for automatisering.

Etter en gjennomgang av CFEngine og Puppet med de målsetningene som var satt for prosjektet, så vil anbefaling av innføring av automatiseringsverktøy være Puppet.
Hvis vi ser på de kriteriene som ble satt, det ble ikke funnet ferdigskrevne moduler for noen av oppgavene som skulle løses i dette prosjektet med CFEngine. Gode kodeeksempler var mangelvare i CFEngine og antall linjer som var endret mellom Centos 6 og 7 var likt mellom Puppet og CFEngine.

Prismessig, er det ~29 kroner dyrere pr. klient å kjøre Puppet, på 100 klienter, vil det koste 79400 pr. år å ha support på Puppet, 76500 for CFEngine. De er ganske like i pris for support.

	Puppet	CFEngine
Muligheten for ferdigskrevne moduler	✔	✖
Gode kodeeksempler funnet online.	✔	⊖
Antall linjer endret mellom Centos 6 og Centos 7	✔	✔
Pris.	✔	✔

✔ Ok, fungerer bra

✖ Mangler.

⊖ Fungerer mindre bra.

Det ble også funnet noen flere grunner under utførelsen av dette prosjektet der Puppet gjorde det bedre enn CFEngine.

- Oppsett av forskjellige kjøremiljø.
- Mulighet til å bruke ferdigskrevne moduler.
- Rapportering tilbake fra tjenere.

Konklusjonen på dette på prosjektet er at en innføring av automatiseringsverktøyet Puppet er å anbefale for Nasjonalbiblioteket.

Det er enklere å sette opp forskjellige kjøremiljø i Puppet, siden dette er noe som Puppet har støtte for. Det vil være greit å ha en mulighet for å sette opp flere miljøer, slik at man kan teste ut en konfigurasjon før man setter den i produksjon. Dette kan være at moduler har fått oppgraderinger som man ønsker å teste ut, at det kommer ny programvare som man vil teste ut før det skal i produksjon. Eller at man vil teste at en oppgradering av ett system går slik man hadde planlagt.

Ferdigskrevne moduler/promisefiler: CFEngine har få ferdigskrevne promisefiler. Puppet har puppetforge (Puppetforge, 2015)der man finner ferdigskrevne moduler. Flere av disse modulene blir brukt av mange brukere, som har gitt tilbakemelding på og har testet ut modulene. Dette fører til at Nasjonalbiblioteket slipper å bruke tid på å skrive en del moduler selv, da disse finnes ferdigskrevet og kan tas i bruk.

Rapportering tilbake til tjeneren: I Puppet er dette funksjonalitet som er innebygget, i CFEngine så er dette funksjonalitet som man må lage selv.

11.2 Veien videre

I dette prosjektet, har det blitt testet 2 forskjellige automatiseringsverktøy. Det har også blitt testet en del rundt automatisering og patching.

For å teste patching ble verktøyet Katello testet ut, Katello inneholder Foreman. Foreman blir da ett GUI som man bruker for å kontrollere Puppet med, i tillegg til det, gir Katello en enkel måte å utføre patching og vedlikehold av Linuxtjenere som nevn tidligere i dette prosjektet.

Det som ble funnet ut, var at Katello gir det som etterspørres i oppgavebeskrivelsen.
Så anbefalingen om veien videre, vil være å starte med å først automatisere deler blir brukt av så mange systemer som mulig. Dette kan f.eks være å automatisere innmeldinger av linux-tjenere som har Puppet installert til en felles NTP server. Små oppgaver som det ikke tar så lang tid å automatisere vil gi en større gevinst. Siden man, etter å ha automatisert alle de små oppgavene, har mer tid til å jobbe med de litt større og vanskeligere tingene å automatisere.

Ved innføringen av Puppet, kan man bruke Katello som ett overbygg, det har blitt testet i dette prosjektet. Men ikke fullt ut, siden hovedfokus var å se om Puppet og CFEngine klarte å løse utfordringene med patching og vedlikehold alene. Vedlagt ligger en installasjonsmanual og dokumentasjon rundt det som har blitt gjort med Katello i dette prosjektet.

Det neste steget er å automatisere ett helt system og i første omgang ved at man kan installere en linux-tjener manuelt, melde den inn i Puppet. For deretter å se at den blir ferdig oppsatt ved hjelp av konfigurasjonen som man har skrevet i Puppet.

Når så dette er gjort, kan man lage klar kickstartfilen og velge å automatisere enten via script, eller ved hjelp av ett verktøy slik som f.eks Katello.

Når man har klart automatisk utrulling av linux-tjenere, kan man gå videre på å innføre automatisering av pakkebrønner. Testene som ble utført i dette prosjektet viser at Katello fungerer godt til dette.

En kort oppsummering for anbefalingen for veien videre vil være: Start med å automatisere pakker på systemer. Små enkle moduler som gjelder for flest mulig linux-tjenere, når man ser at man har automatisert oppsettet for standardinstallasjonen. Gå videre ved å lage en grunnpakke for installasjon av en basis-tjener og automatiser utrullingen av denne. Når man er klar med automatisk utrulling av basiskonfigurasjon, kan man gå videre med å teste ut automatiseringen av pakkebrønner.

Alt dette er ting som ble funnet i verktøyet Katello, en innføring av Katello. Eller ett lignende verktøy, f.eks standard Foreman, eller RedHat Sattelite 6 hvis man ønsker support.

Det kan lønne seg å starte med ett av disse verktøyene og legge på kompleksitet etter hvert, siden verktøyet støtter det og man blir vant med å bruke verktøyet istedenfor å bruke bare Puppet. En annen fordel er den grafiske oversikten man får i verktøyene.

12 Konklusjon prosjekt

Etter å ha testet ut automatisering med Puppet og CFEngine. Blir konklusjonen i dette prosjektet at det anbefalte verktøyet for automatising i Nasjonalbiblioteket Puppet. CFEngine har mange gode sider. Men, tar man i betraktning de sammenligningskriteriene som ble lagt i prosjektet for å sjekke hvilket automatiseringsverktøy som man burde innføre i Nasjonalbiblioteket, kommer Puppet bedre ut enn CFEngine.

Det har også blitt testet ut verktøy for å styre pakkebrønner (pulp). Dette ble testet ved hjelp av konfigurasjonsverktøyet Katello og en innføring av Katello anbefales. Katello kan brukes til å styre Puppet for konfigurasjonen av linux-tjenere. Dette for at bestillingen sier at man ønsker å se på hvilke gevinster som kan realiseres ved å innføre automatisering og patching.

Gevinstene med å innføre automatisering av installasjon og Patching som ble avdekket er at ved å innføre mer automatisering, vil Nasjonalbiblioteket få mer autonome systemer.
Det vil gå raskere å sette opp systemer. Det blir enklere å rulle ut nye systemer og det blir enklere å utføre systemendringer på alle systemene samtidig. Ønsker man å rulle ut en ny konfigurasjon til alle systemene, så kan dette utføres kjapt og enkelt fra en lokasjon.

Det har ikke blitt fokusert på dette i prosjektet. Siden Puppet klientene sender informasjon om seg selv til tjeneren, vil man ha en sentral plass der man har oversikt over alle klientene som Puppet er installert på. Man kan der hente ut informasjon om disse klientene (F.eks hvor mye ram, hvilken prosessorarkitektur som klienten har etc.).

For å oppnå ett system der man har automatisk patching og automatisk installasjon, så ble det i dette prosjektet avdekket at man trenger mer enn bare Puppet og CFEngine. Puppet og CFEngine, gir muligheten til å automatisere installasjon. De gir derimot ikke en fullgod løsning på patching av hele systemer. Man kan bruke de til automatisk patchin, men for å ha kontroll over hele system ønsker man gjerne noe mer. For å kunne få alt som ble etterspurt i bestillingen fra Nasjonalbiblioteket, ble konklusjonen at man trenger ett verktøy som for eksempel Katello.

Hvis man ønsker en supportert versjon av Katello, så finnes det. Den heter Sattelite version 6 (Sattelite, 2015)som blir supportert av RedHat (Redhat hjemmeside, 2015)

Vedlegg A: Referanseliste

Andersen, E. S. (2004). *Målrettet prosjektstyring.* NKI-forl.

Bokhylla. (2015, Februar 15). (Nasjonalbiblioteket) Hentet Februar 15, 2015 fra Bokhylla tjeneste levert av Nasjonalbiblioteket: http://www.nb.no/nbsok

CFEngine enterprice reporting. (2015, Mai 14). Hentet Mai 14, 2015 fra https://docs.cfengine.com/docs/3.5/manuals-enterprise-reporting.html

CFEngine hjemmeside. (2015, Mars 07). Hentet Mars 07, 2015 fra http://cfengine.com/

CFEngine scaling. (2015, Mai 03). (CFEngine, Produsent) Hentet Mai 03, 2015 fra https://auth.cfengine.com/archive/manuals/st-scale

ENC. (2015, Mai 03). Hentet Mai 03, 2015 fra Informasjon om puppet ENC: https://docs.puppetlabs.com/guides/external_nodes.html

Informasjon om promisefiler. (2015, Mai 03). Hentet Mai 03, 2015 fra http://www.centos.org/docs/5/html/Installation_Guide-en-US/ch-kickstart2.html#s1-kickstart2-whatis

Katello hjemmeside. (2015, Mai 03). Hentet Mai 03, 2015 fra hjemmesiden til katello: http://katello.org

Kickstart. (2015, Mai 03). Hentet Mai 03, 205 fra Centos webområde for kickstart: http://www.centos.org/docs/5/html/Installation_Guide-en-US/ch-kickstart2.html#s1-kickstart2-whatis

Nasjonalbiblioteket mandat. (2015, Februar 15). Hentet Februar 15, 2015 fra http://www.nb.no/Om-NB/Fakta/Mandat

oVirt. (2015, April 15). Hentet April 15, 2015 fra Webområde for Ovirt: http://www.ovirt.org/Home

Pulp. (2015, Mai 03). Hentet Mai 03, 2015 fra Hjemmesiden til pulp: http://www.pulpproject.org

Puppet dashboard. (2015, Mai 03). Hentet Mai 03, 2015 fra github: https://github.com/sodabrew/puppet-dashboard

Puppet hjemmeside. (2015, Mars 15). Hentet Mars 15, 2015 fra https://puppetlabs.com/

Puppetforge. (2015, Mai 03). Hentet Mai 03, 2015 fra webområde for puppetmoduler: https://forge.puppetlabs.com/

Puppetmoduler referanse. (2015, Mai 14). Hentet Mai 14, 2015 fra https://docs.puppetlabs.com/puppet/latest/reference/modules_fundamentals.html

Redhat hjemmeside. (2015, Mai 03). (RedHat, Produsent) Hentet Mai 03, 2015 fra http://www.redhat.com/en

Sattelite. (2015, Mai 03). Hentet Mai 03, 2015 fra redhat sattelite side: https://access.redhat.com/products/red-hat-sattelite

TheForman hjemmeside. (2015, Mai 03). Hentet Mai 03, 2015 fra http://theforeman.org

Versjonskontroll. (2015, Mai 15). Hentet Mai 15, 2015 fra GIT Versjonskontroll: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

yum exclude. (2015, Mai 14). Hentet Mai 14, 2015 fra redhat: https://access.redhat.com/solutions/10185

yum priorities. (2015, Mai 03). Hentet Mai 03, 2015 fra Centos informasjon om yum priorities: http://wiki.centos.org/PackageManagement/Yum/Priorities

Vedlegg B: teknisk ordliste

Bokhylla: Nettside der brukere kan lese elektroniske publikasjoner på nett. (Dette kan være aviser, bøker, tegneserier med mer.)

Centos: "gratisversjon" av RedHat. Bortimot det samme som RedHat, men man har ikke support.

CFEngine: Automatiseringsverktøy for å sørge for automatisk utrulling av tjenere.

DHCP: Dynamic Host Configruation Protocol, brukes for å allokere ipadresser til linux-tjenere automatisk

DNS: Domain Name System, navneoppslag. Brukes for å gjøre ipadresser om til navn på maskiner

Failover: Metode for å feile over til ett system som står klart hvis det systemet som er i bruk skulle feile

FirewallD: Brannvegg til linux

freeIPA/IPA: Dette er ett identitets, policy og revisjonsverktøy. Her legger man inn brukere, grupper og hvilke rettigheter disse har for å enklere kunne styre hvilke rettigheter brukere har ute på klientmaskiner som er tilkoblet denne.

GUI: Grafisk brukergrensesnitt.

IPtables: Brannvegg som brukes i Linux

Linux-tjener: Her rerert til en datamaskin som har installert linux, enten fysisk eller virtuell.

Nagios: Verktøy som brukes til overvåkning av servere, klienter.

NginX: Er en webserver, brukes for å vise fram websider.

oVirt: Virtualiseringsplattformen som ble brukt i dette prosjektet til å kjøre virtuelle maskiner på.

Patching: Feilretting av programmer.

Promisefiler: Konfigurasjonsfiler til CFEngine som brukes av CFEngine når den skal utføre en oppgave.

Puppet: Automatiseringsverktøy for å sørge for automatisk utrulling av tjenere.

Linuxtjener: Her referert til en datamaskin, enten fysisk eller virtuell som det er installert Linux operativsystem på.

Moduler: Brukt her om puppet, moduler er selvstendinger samlinger av kode og data som brukes for å kjøre automatiserte installasjoner med puppet

Python: Objekt orientert programmeringsspråk

RedHat: En linuxversjon der man betaler for support.

SELinux: Dette er en sikkerhetsmodul for Linux, skal sørge for bedre sikkerhet på linuxmaskiner.

Sudo: Er en kommando man bruker for å kjøre superbruker kommandoer på en datamaskin som det er installert ett *nix operativsystem på.

Virtuell: Linux-tjener som ikke er installert på en fysisk datamaskin, men kjøres inne i en annen linux-tjener.

Vedlegg C Forprosjekt

Vedlegg D Installasjonsmanual