KU LEUVEN

FACULTEIT
INGENIEURSWETENSCHAPPEN

Assesor aanpassen in preamble

Voorwoord

Vooraf wil ik graag de mensen bedanken die mij geholpen hebben bij het schrijven van deze thesis: mijn ouders, die het mogelijk maakten om deze studies te volgen en mij hielpen waar mogelijk. Mijn begeleider, Bart Vanbrabant, voor de hulp, het geduld en de motivatie die hij gegeven heeft het afgelopen jaar. Thomas Margot, Yoshi Delaey, Hilke Cottenie en vele anderen voor het overlezen en corrigeren van de tekst. Kjell Deturck voor de emotionele steun in de moeilijkere momenten. Karlijn Ongena voor het vermelden van mijn naam in haar thesis.

Harm De Weirdt

Inhoudsopgave

Samenvatting

Configuratiemanagementsoftware is een vereiste geworden bij het onderhoud van grote gedistribueerde systemen. Ze laat toe een declaratief model op te stellen van de gewenste configuratie, waarna de software zorgt voor het correct uitrollen. In dit model staan de verschillende resources die deel uitmaken van de opstelling. Tussen deze resources kunnen afhankelijkheden en vereisten opgesteld worden. Zo is er bijvoorbeeld de aanwezigheid van een bovenliggende map een vereiste voor het creeëren van een bestand, of de installatie van een pakket een vereiste voor het opstarten van de bijhorende service. Een webserver kan afhankelijk zijn van een databaseserver voor het aanbieden van bepaalde functionaliteit. Als de vereisten en afhankelijkheden niet gespecificeerd (kunnen) worden in het model moet het uitrolproces vaak meerdere keren gestart worden voordat alle resources correct interageren.

Deze thesis maakt gebruikt van IMP, een nieuwe configuratietool die toelaat zowel afhankelijkheden als vereisten te modelleren. Er wordt onderzocht hoe, met behulp van extra informatie in het configuratiemodel, extra vereisten en afhankelijkheden kunnen toegevoegd worden om zo het aantal uitrolprocessen te reduceren.

Hoofdstuk 1

Inleiding

Configuratiebeheergereedschappen zijn ontwikkeld om het leven van systeembeheerders makkelijker te maken. De serverinfrastructuren die ze moeten onderhouden worden steeds uitgebreider en complexer. Manueel elke server configureren kost niet alleen te veel tijd, maar is ook erg foutgevoelig. Er bestaan reeds verschillende oplossingen voor dit probleem. Een eerste manier is scripts gebruiken. Dit is al een stap in de goede richting, maar is nog steeds niet voldoende: als de uitvoering van een script afgebroken wordt blijft het systeem in een onstabiele toestand.[**?**]

Een andere manier om een verzameling gelijkaardige machines van hun initiële configuratie te voorzien is het gebruik van images. Daarbij wordt eerst één machine manueel geconfigureerd en daarna wordt de volledige set-up gekloond naar de rest van de servers. Deze methode werkt niet meer voor het verdere onderhoud van de machines.

Dit onderhoudsprobleem komt nog prominenter voor als de infrastructuur niet lokaal, maar in de cloud gehost wordt. Een groot voordeel van werken in de cloud is de flexibiliteit waarmee servers kunnen toegevoegd en weggenomen worden. Dit proces gebeurt vaak automatisch waardoor manuele configuratie helemaal geen optie meer is. In een dergelijke omgeving is een tool die uit zichzelf de volledige infrastructuur kan beheren bijna een noodzaak.

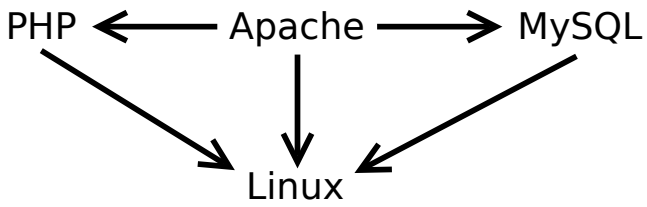
source?

Configuratiebeheergereedschappen (of CMS: Configuration Management Software, vanaf nu zal deze term gebruikt worden) zoals IMP[**?**], Puppet[**?**], CFEngine[**?**],... laten toe op een efficiënte manier IT-infrastructuren op te zetten en te onderhouden. De gebruiker van een dergelijke tool specifiëert eerst een model dat de gewenste toestand van de volledige infrastructuur beschrijft. Dit model bestaat uit een oplijsting van machines met de gewenste aanwezige resources (bestanden, mappen, services,...) die ze moeten aanbieden. Sommige tools laten ook toe samenhorende resources te groeperen in een concept, zoals een webserver of een databaseserver. Dit vermijdt duplicatie van code in het model.

Bij het uitrollen van een model (een "deployment run") inspecteert de CMS de huidige toestand van elke machine en vergelijkt ze met de gewenste toestand. Als er een verschil is maakt de CMS de nodige aanpassingen, indien niet onderneemt ze geen actie. De beheerder van de verzameling systemen moet dus na het opstellen van

de initiële configuratie zelf geen stappen meer ondernemen om te verzekeren dat de gewenste situatie bereikt wordt. Als er later nog aanpassingen moeten gebeuren moet enkel het model aangepast worden en een nieuwe deployment run gestart worden.

Een belangrijk aspect van elk gedistribueerd systeem is de afhankelijkheden die bestaan tussen de verschillende onderdelen. Stel het voorbeeld van een LAMP-stack: een Linuxdistributie met de Apache webserver, de MySQL database en PHP. Daar kan de webserver niet zijn volledige functionaliteit aanbieden voordat de database online is. De webserver is dus afhankelijk van de databaseserver. Zolang PHP niet geïnstalleerd is kan de webserver ook geen dynamische pagina's aanbieden. Ze is dus ook afhankelijk van de PHP-installatie. Een volledige mapping van de verschillende afhankelijkheden binnen de LAMP-stack is te zien op figuur 1.1.



Figuur 1.1: Grafische voorstelling van de afhankelijkheden binnen een LAMP-stack

Als deze afhankelijkheden niet gespecificeerd worden in het model kan de CMS er ook geen rekening mee houden. Het kan dat de tool het model in een foute volgorde uitrolt: eerst de webserver, dan PHP en uiteindelijk de database. De webserver zal bij het opstarten proberen te verbinden met de database, maar deze is nog niet online. In de tijd tussen het opstarten van de webserver en de installatie van PHP zal ze ook geen dynamische webpagina's kunnen tonen. Een op het eerste zicht succesvolle deployment run kan dus leiden tot een configuratie die niet volledig werkt.

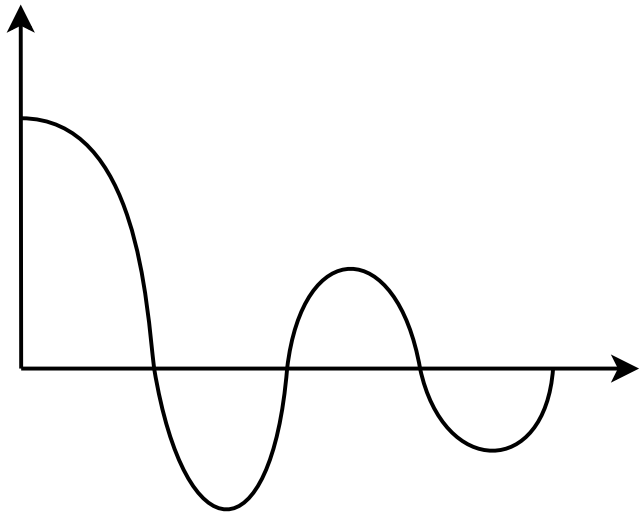
In vergelijking met de beginsituatie is de toestand van de configuratie na de ene run wel al minder afwijkend van de gewenste situatie: de verschillende pakketten, services en configuratiebestanden zijn al aanwezig. Tijdens de volgende deployment run zal de Apache service herstart worden en dan zal ze wel kunnen connecteren met de database.

CMS zal nooit aanpassingen maken die zorgen voor een configuratie die verder afwijkt van het model dan voorheen. Na een paar iteraties zal uiteindelijk altijd de gewenste configuratie bereikt worden. Grafisch wordt dit voorgesteld op figuur 1.2. Het aantal iteraties is afhankelijk van de hoeveelheid afhankelijkheden die bestaan, maar niet aanwezig zijn in het model.

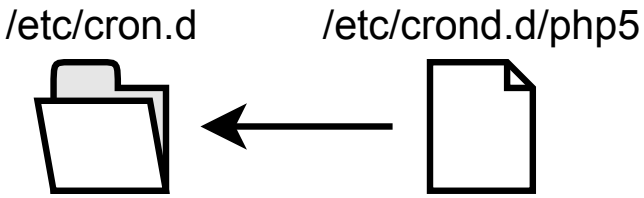
"sprong" in onderwerp

Beter woord voor abstracties

Databases en webservers zijn abstracties die bestaan uit een verzameling basisobjecten zoals bestanden, pakketten en services.Tussen deze objecten bestaan er natuurlijk ook afhankelijkheden, bijvoorbeeld tussen een bestand en de map waarin het staat: als de CMS eerst probeert het bestand te creeëren en dan pas de map zal de deployment run slechts gedeeltelijk slagen want een bestand kan niet bestaan zonder een bovenliggende map. Figuur 1.3 stelt deze afhankelijkheid nog eens grafisch voor. In tegenstelling met het voorbeeld van de LAMP-stack zal de gebruiker direct



FIGUUR 1.2: Grafische voorstelling van de convergentie na een reeks deployment runs.



FIGUUR 1.3: Grafische voorstelling van de afhankelijkheid tussen een bestand en zijn bovenliggende folder

na het uitrollen al zien dat er iets foutgaat. De CMS zal namelijk tonen dat sommige resources niet werden uitgerold. Bij de LAMP-stack werd alles succesvol uitgerold, maar de gebruiker kan pas zien dat er iets fout is als hij de logs van de Apache service bekijkt, of probeert een site te bezoeken.

We maken dus het onderscheid tussen twee gevallen: vereisten en afhankelijkheden. In het geval van het bestand en de map is er sprake van een *vereiste* die niet voldaan is: de map moet bestaan vóór het aanmaken van het bestand of deze wordt niet aangemaakt. In het geval van de LAMP-stack houdt de CMS geen rekening met de *afhankelijkheid* tussen de webserver en de databaseserver. Alle resources worden foutloos aangemaakt, maar toch werkt de uiteindelijke configuratie niet omdat een foute volgorde werd gehanteerd. In beide gevallen is een extra deployment run nodig voordat de configuratie correct werkt.

Als de CMS wel rekening houdt met de afhankelijkheden en vereisten moet het model maar één keer uitgerold worden. Het doel van deze thesis is dan ook om uit te zoeken hoe de software deze extra informatie automatisch uit het model kan afleiden.

Doel pas later vermelden?

De huidige CMS laten toe om op het niveau van bestanden, packages en services vereisten en afhankelijkheden te specificeren. De specificatie van een vereiste en een afhankelijkheid overlapt bij deze tools en wordt een "requirement" genoemd. Bij het uitrollen van een model gebruikt de software die informatie om een volgorde op te leggen waarmee de verschillende objecten aangemaakt worden.

Deze tools compileren tijdens de deployment voor elke machine hun deel van het model. Elke machine krijgt dus enkel informatie over wat hij zelf moet doen. Afhankelijkheden en vereisten binnen eenzelfde machine verwerken is geen probleem, maar tussen verschillende machines is dit niet mogelijk. Als de LAMP-stack op één machine geïnstalleerd wordt kan de gebruiker nog aangeven dat de Apache service afhankelijk is van de databaseservice en PHP. Als daarintegen de Apache service op een machine draait en de MySQL-service op een andere kan de gebruiker dit niet meer aangeven. De enige optie is dan meerdere keren het model uitrollen tot de configuratie correct werkt. [1]

IMP (Infrastructure Management Platform) is een nieuwe tool die momenteel nog in ontwikkeling is. Ze verschilt op twee vlakken van de andere CMS: gebruik van het model tijdens het uitrolproces en de specificatie van vereisten en afhankelijkheden.

De andere aanpak tijdens het uitrollen van een model laat toe afhankelijkheden tussen hoog-niveau objecten te specificeren: in tegenstelling tot de vorige tools krijgt elke machine het volledige model ter beschikking en niet alleen zijn eigen deel. Machines kunnen zo niet alleen rekening houden met afhankelijkheden en vereisten tussen eigen concepten, maar ook tussen andere machines.

Vervolgens maakt IMP, in tegenstelling tot andere tools, wel het onderscheid tussen vereisten en afhankelijkheden: een vereiste is net zoals bij de andere tools een requirement, maar afhankelijkheden worden gemodelleerd door middel van relaties tussen concepten. Aangezien IMP tijdens het uitrollen elke machine het volledig model ter beschikking stelt kunnen intermachinale afhankelijkheden ook verwerkt worden.

Klopt dit volledig? Mijn heuristiek leidt afhankelijkheden af uit de relaties, maar dat is daarom niet het enige nut ervan.

IMP kan het volledig model in één keer uitrollen als het genoeg informatie krijgt in de vorm van vereisten en afhankelijkheden. De doelstelling van deze thesis is aan de hand van heuristieken extra informatie uit het model halen en zo het aantal nodige deployment runs te verminderen. De veronderstelling is wel dat degene die het model opstelt genoeg domeinspecifieke informatie aanlevert die door deze heuristieken gebruikt kan worden. *TODO* : hoe oplossing + resultaten

Belang van correct uitrollen: kan soms interessanter zijn dan snel uitrollen

Gebruik van "dus"

[1] Voor Puppet bestaat er wel een workaround, zie [**?**]. "Orchestration" CMS zoals Ansible kan dit wel. *TODO* : Verschil opzoeken en uitleggen hier of in related works?

Hoofdstuk 2

Naar het automatisch toevoegen van vereisten en afhankelijkheden

Configuratiebeheergereedschappen die gebruik maken van vereisten en afhankelijkheden reduceren daarmee het aantal deployment runs dat nodig is om een stabiele toestand te bereiken. Dit hoofdstuk introduceert enkele heuristieken die ,naast degene die de gebruiker specificeert, extra vereisten en afhankelijkheden probeert toe te voegen.

Secties 2.1, 2.2 en 2.3 introduceren heuristieken die extra vereisten toevoegen aan het model en zo het aantal fouten reduceren.

De heuristieken uit secties 2.4 en 2.5 werken samen om afhankelijkheden om te zetten in vereisten om zo een efficiëntere volgorde opleggen aan het uitrolproces.

2.1 Vereisten tussen bestanden en mappen

Deze heuristiek zorgt dat elk bestand zijn bovenliggende map vereist. Een bestand moet namelijk deel uitmaken van een map. Figuur 1.3 geeft hiervan een visuele voorstelling.

Het algoritme dat hiervoor gebruikt wordt ziet eruit als volgt (pseudocode):

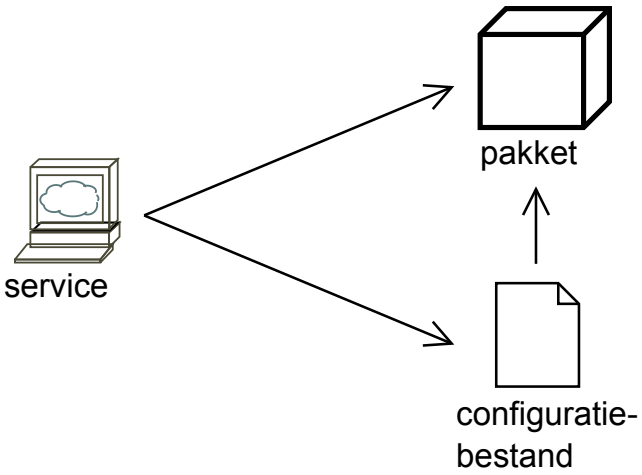
```
for file in host.resources:
    for dir in host.resources:
        if get_directory(file.path) == dir.path:
            file.requires.add(dir)
```

Als de map niet vermeld wordt in het model wordt deze ook niet toegevoegd omdat dit ongewenste gevolgen kan hebben.

Welke?

2.2 Vereisten tussen services, packages en configuratiebestanden

Net zoals bij bestanden en mappen zijn er voorwaarden voor het correct uitrollen van packages, services en hun eventuele configuratiebestanden: een service kan niet starten als zijn package niet geïnstalleerd is en zal niet correct werken zonder een aangepast configuratiebestand. De verzameling van een service, het pakket dat die service installeert en de configuratiebestanden wordt vanaf nu een stack genoemd. Een schematische voorstelling van de onderlinge vereisten bnnen een stack is te vinden op figuur 2.1.



Figuur 2.1: Grafische voorstelling van de onderlinge vereisten van packages, services en configuratiebestanden binnen een stack

Een eerste aanpak om de correcte afhankelijkheden te introduceren is alle pakketten en bestanden een vereiste maken van alle services en alle pakketten een vereiste van alle bestanden. De configuratiebestanden mogen pas geplaatst worden nadat de packages geïnstalleerd zijn omdat anders het aangepast configuratiebestand overschreven wordt. Dit introduceert uiteraard veel afhankelijkheden die niet overeenstemmen met de werkelijkheid maar ze maken daardoor het resultaat van het uitrolproces niet fout. Als het model maar één stack bevat voegt de heuristiek geen enkele overbodige vereiste toe, bij twee stacks zes overbodige vereisten, bij drie stacks 27,... Een algoritme in pseudocode ziet er uit als volgt:

```
for service in host.resources:
    for file in host.resources:
        service.requires.add(file)
    for package in host.resources:
        service.requires.add(package)
for file in host.resources:
    for package in host.resources:
        file.requires.add(package)
```

Deze heuristiek resulteert in een soort batchuitvoering waarbij de CMS eerst alle pakketten installeert, dan alle (configuratie)bestanden en uiteindelijk alle services.

De volgende aanpak gebruikt meer info uit het model en resulteert in minder overbodige afhankelijkheden. In de modelcode worden bestanden, packages en services die bij elkaar horen meestal binnen eenzelfde implementatie gespecifiëerd. Een voorbeeld is de implementatie van een MySQL server:

```
implementation mysql:
    pkg = std::Package(host= host, name= "mysql-server", state= "installed")
    svc = std::Service(host= host, name= "mysqld", state= "running", onboot=
        true)

    config= std::ConfigFile(host= host, path= "/etc/my.cnf", content= template
        ("mysql/my.cnf.tmpl"), requires= pkg, reload= true)
    conf_dir= std::Directory(host= host, path= "/etc/mysql.conf.d", owner= "
        root", group= "root", mode= 755)

    dblist= std::ConfigFile(host= host, path= "/etc/sysconfig/mysql", reload=
        true, content= template("mysql/databases.tmpl"))
end
```

Bij het verwerken van het model tijdens het uitrolproces zijn de resources van deze stack binnen éénzelfde scope gedefiniëerd. De heuristiek zoekt verzamelingen van packages en services die binnen éénzelfde scope gedefiniëerd zijn en stelt de correcte afhankelijkheden op tussen enkel die groep resources. De aanwezigheid van files is optioneel: sommige services hebben geen aangepast configuratiebestand nodig.

```
srv_stacks = []
for resource in resources:
    same_scope = [res in resources where res.scope == resources.scope]
    if same_scope.contains(services) and same_scope.contains(packages):
        srv_stacks.add(same_scope)

for stack in srv_stacks:
    for service in stack:
        for file in stack:
            service.requires.add(file)
        for package in stack:
            service.requires.add(package)
    for file in stack:
        for package in stack:
            file.requires.add(package)
```

Deze heuristiek voegt geen overbodige vereisten toe, op voorwaarde dat degene die het model opstelt de verschillende stacks opsplitst in verschillende implementaties. Resultaten van het gebruik van deze heuristiek staan in sectie 3.3.

2.3 Vereisten tussen resources met gelijkaardige naam

De laatste heuristiek heeft een gelijkaardige doel als de vorige: vereisten opstellen tussen de onderdelen van een stack. In plaats van te werken binnen een scope zoekt deze heuristiek naar resources met een gelijkaardige naam.

In het geval van de vereisten tussen bestanden en pakketten negeert de heuristiek de cijfers op het einde van de naam van een pakket en als er een splitsingsteken staat in de naam houdt het enkel rekening met het eerste deel. Zo stelt de heu-

ristiek ook afhankelijkheden op tussen bijvoorbeeld het pakket “cassandra12” en het configuratiebestand “/etc/cassandra.conf” of het pakket “openssh-server” en “/etc/openssh.conf”.

Uitleggen waarom

Namen gebruiken levert meer vereisten op dan het gebruik van scopes.

```
for service in host.resources:
    similar_resources = []
    for file in host.items:
        if file.name.contains(service.name):
            service.requires.add(file)
    for package in host.items:
        if package.name.contains(service.name):
            service.requires.add(package)

for package in host.resources:
    name = remove_digits(package.name)
    name = name.split("-")[0]
    for file in host.items:
        if file.name.contains(service.name):
            package.requires.add(file)
```

2.4 Afhankelijkheden door relaties

IMP laat toe relaties tussen concepten te modelleren. Een voorbeeldrelatie is de volgende:

```
BaseClient clients [0:] -- [0:] BaseServer servers
```

Deze betekent dat een BaseClient nul of meerdere BaseServers nodig heeft, en omgekeerd. Een ander voorbeeld is

```
Host host [1] -- [0:] File files
```

Dit betekent dat op een Host nul of meerdere files kunnen staan, maar dat elke File één Host moet hebben. Deze heuristiek leidt hieruit af dat een File niet kan bestaan zonder een host en het dus nodig is dat eerst de Host uitgerold wordt voordat geprobeerd wordt de File te creeëren.

Algemeen kan men dus besluiten dat elke relatie waar de ene kant een multipliciteit van [0] of [0:] heeft en de andere kant multipliciteit [n] of [n:] de eerste entiteit afhankelijk is van de tweede. De code voor deze heuristiek ziet er uit als volgt:

```
for lib in model.get_scopes():
    for concept in lib.variables():
        if concept.hasattr(relation)
            if concept.relation.low == 0 and concept.relation.end == 1:
                concept.relation.depends = True
```

Als enkel deze heuristiek gebruikt wordt zal IMP niets veranderen aan het uitrolproces: IMP houdt momenteel nog geen rekening met afhankelijkheden. De extra informatie kan wel gebruikt worden door andere heuristieken. Een voorbeeld hiervan is deze hieronder (sectie 2.5) die toelaat afhankelijkheden zoals deze tussen een webserver en databaseserver te gebruiken voor een optimaler uitrolproces.

2.5 Vereisten vanuit afhankelijkheden

IMP laat niet alleen toe om afhankelijkheden tussen enkelvoudige concepten zoals bestanden, services,... te specifiëren maar ook tussen samengestelde concepten zoals webservers, databaseservers,... Deze betekenen dat de ene kant niet zijn volledige functionaliteit kan aanbieden zonder de aanwezigheid van de andere kant. Aangezien een server zijn functionaliteit aanbiedt aan de hand van een service en niet zijn configuratiebestanden en pakketten stelt deze heuristiek enkel afhankelijkheden op tussen de services.

IMP laat toe om verschillende libraries te gebruiken (en zelf te definiëren) met daarin voorgedefinieerde concepten. Het algoritme begint met het doorzoeken van deze libraries naar concepten die services bevatten. Dan kijkt het of dat concept afhankelijk is van een ander concept. Als dit het geval is wordt gekeken of dat ander concept ook services bevat. Zo ja stelt de heuristiek vereisten op tussen de services.

In pseudocode ziet dit er uit als volgt:

```
for lib in model:
    for concept in lib.variables():
        concept_services = get_services(concept)
        if concept_services is not None:
            for relation in concept.get_attributes():
                if relation.depends: #afhankelijke relatie
                    for instance in concept.values: #Voor elke instantie v/h
                        concept
                        req_concepts = relation.end
                        req_resources = []
                        for req_concept in req_concepts:
                                for srv in get_services(req_concept):
                                    req_resources.add(srv)
                        if req_resources is not None:
                            for service in concept_services:
                                for req_res in req_resources:
                                    service.requires.add(req_res)
```

Het is belangrijk dat deze heuristiek wordt opgeroepen na deze van sectie 2.4, daar worden namelijk extra afhankelijke relaties opgesteld die hier kunnen gebruikt worden.

2.6 Besluit van dit hoofdstuk

Hoofdstuk 3

Evaluatie

Om de impact van de verschillende heuristieken te meten volgt nu een overzicht van de testresultaten. Voor het merendeel van de testen is een simulator gebruikt. Sectie 3.1 legt uit waarom een simulator gebruikt wordt in plaats van IMP zelf, en hoe deze werkt.

De daarop volgende secties bespreken de resultaten van elke heuristiek. De algemene verwachting is dat het gebruik van heuristieken het aantal deployment runs reduceert en zo ook de totale uitroltijd. Deze optimalisatie kost wel extra verwerkingstijd (de uitvoering van de verschillende heuristieken) maar deze is normaal gezien te verwaarlozen ten opzichte van het volledige uitrolproces.

meten

3.1 Simulator

Er zijn twee belangrijke voordelen aan het gebruiken van een simulator. Ten eerste is de uitroltijd evenredig met de grootte van het model. Zelfs met het gebruik van heuristieken om het aantal deployment runs te minimaliseren blijft een significant deel van het proces gespendeerd aan bijvoorbeeld het downloaden van pakketten. Ten tweede laat een simulator toe modellen met honderden machines uit te rollen op één enkele pc.

De simulator bootst een systeem na dat Fedora 18 draait en gebruik maakt van de yum pakket manager.

3.1.1 Uitwerking

Het proces begint nog altijd bij IMP zelf. Als de gebruiker de optie “-j” meegeeft compileert IMP het model volledig maar in plaats van het dan uit te rollen, schrijft hij het weg in een JSON bestand. De simulator gebruikt dat bestand als invoer om het uitrolproces te simuleren. Het resultaat van een simulatierun is een sqlite-database waarin alle hosts en hun resources vermeld staan (de deployment database genoemd). Om de simulatie zo waarheidsgetrouw mogelijk te maken houdt de simulator rekening met volgende aspecten van het uitrolproces:

- Bestanden en mappen kunnen niet aangemaakt worden voordat de bovenliggende map bestaat
- Services kunnen niet gestart worden voordat het bijhorende pakket en bestanden aanwezig zijn
- Resources worden pas aangemaakt als hun afhankelijkheden voldaan zijn

De simulator maakt gebruik van twee externe informatiebronnen: een lijst van standaard mappen die aanwezig zijn na een nieuwe Fedora installatie en de repository data van verschillende yum repositories. Beide helpen bij het correct uitrollen van de resources.

Het uitrollen van een file gaat als volgt:

```
def deploy_file(file):
  #Check if parent folder exists already
  if file in std_filesystem or file in deployment_database:
    deployment_database.write(file)
  else:
    error('Parent folder doesn't exist!')
```

Het uitrollen van een service gaat als volgt:

```
def deploy_service(srv):
  #Check if required files have been deployed
  requirements = repodata_database.execute('select * from pkgdata where name
      like srv.name')
  if all([req in deployment_database for req in requirements]):
    deployment_database.write(srv)
  else:
    error('Not all required resources were deployed!')
```

Het uitrollen van een pakket gaat als volgt:

```
def deploy_package(pkg):
  pkg_files = repodata_database.execute('select * from pkgdata where name like
      pkg.name')
  foreach file in pkg_files:
    deployment_database.write(file)
```

We gaan er van uit dat fouten bij het uitrollen van een pakket de verantwoordelijkheid zijn van de package manager (hier yum), niet de CMS.

Verschil in uitroltijd simulator/IMP?

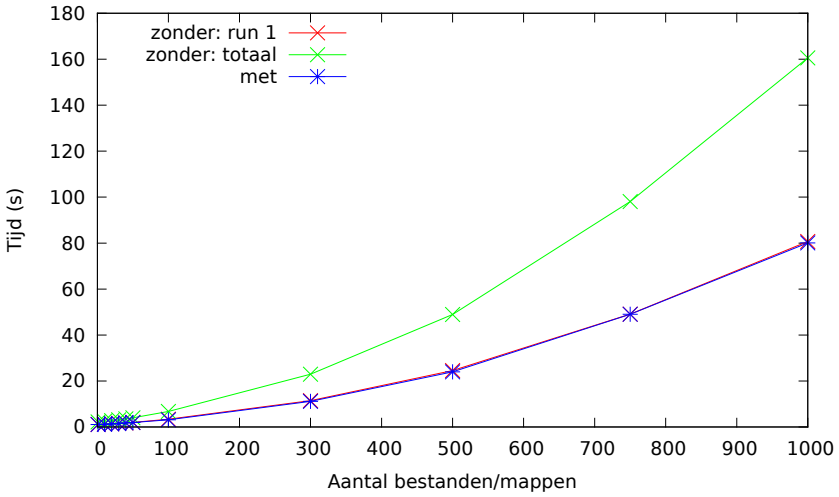
Besluit sectie

3.2 Afhankelijkheden tussen bestanden en mappen

Aangezien een bestand niet kan gecreëerd worden zonder zijn bovenliggende map voegt deze heuristiek automatisch de afhankelijkheid tussen beide toe aan het model.

Aangezien mappen altijd kunnen gecreëerd worden zijn er maximaal twee deployment runs nodig als er geen heuristiek gebruikt wordt. Bij gebruik van de heuristiek moet er exact één uitgerold worden. De verwachting is dat het uitrollen met gebruik van de heuristiek dan ook ongeveer half zo lang duurt.

Figuur 3.1 toont de resultaten van deze test. Elk datapunt is het gemiddelde van 30 runs, uitgevoerd op een virtuele machine met twee cores van 2Ghz en 2GB RAM



FIGUUR 3.1: Testresultaten bij het uitrollen van een stijgend aantal bestanden en mappen, met en zonder gebruik van de heuristiek

ter beschikking. Het model bestaat iedere keer uit vast aantal bestanden en mappen, één bestand per map.

De verwachtingen zijn volledig ingelost: zonder heuristiek zijn er twee deployment runs nodig, met heuristiek slechts één. Dit weerspiegelt zich in de gehalveerde uitroltijd.

3.3 Afhankelijkheden tussen services, pakketten en configuratiebestanden

De specificatie van een service in het configuratiemodel gaat vaak gepaard met de pakket en de configuratiebestanden die die service nodig heeft. Deze combinatie van resources wordt een stack genoemd. Aangezien de service niet correct werkt zonder de aanwezigheid van het pakket en de configuratiebestanden voegt deze heuristiek de gepaste vereisten toe aan het model.

Voor deze test moest IMP 30 keer de NTP service uitrollen op een testmachine. NTP bestaat uit één pakket, één configuratiebestand en één service. In het slechtste geval zijn er normaal gezien dus drie deployment runs nodig om de service correct werkende te krijgen. Als de correcte afhankelijkheden worden opgesteld is er slechts één run nodig. De testresultaten zijn te vinden in tabel 3.3.

De resultaten van de test zijn zoals verwacht: dankzij de heuristiek is maar één deployment run nodig.

	tijd(s)	gemiddeld aantal runs nodig
zonder	3.21	2.0
met	2.21	1

Tabel 3.1: Meetresultaten

3.4 Relaties en afhankelijkheden tussen hoog-niveau concepten

Door het omzetten van relaties naar afhankelijkheden kan het nodige aantal deployment runs gereduceerd worden.

3.5 Use cases

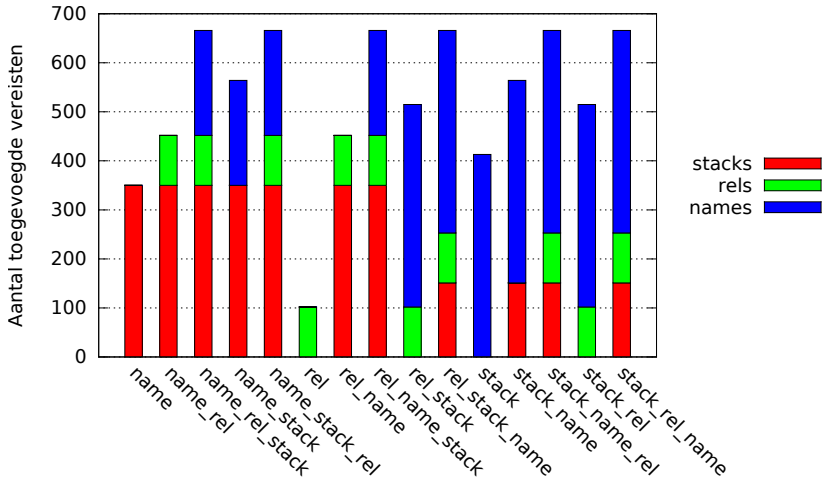
3.5.1 Document processing

Uitleg geven over use case

Dit is eigenlijk geen uitleg van de toepassing van de heuristieken op de use case, meer een uitleg/test van de heuristieken

In afbeelding 3.2 is te zien hoeveel vereisten elke heuristiek toevoegen. De heuristiek die de bovenliggende map toevoegt aan de vereisten van een map of bestand wordt altijd uitgevoerd. Deze heuristiek beschouwen we namelijk als fundamenteel: de vereisten die ze toevoegd zijn sowieso juist, terwijl sommige vereisten die door andere heuristieken worden toegevoegd mogelijks overbodig zijn.

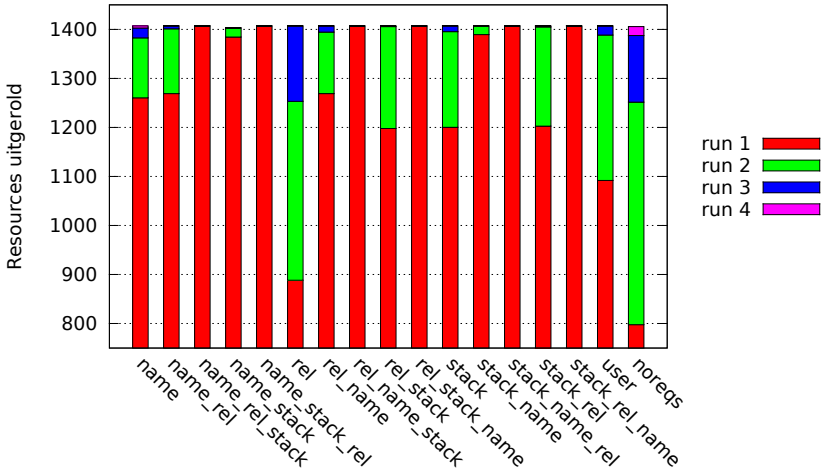
De verwachting is dat sommige heuristieken zullen overlappen, bijvoorbeeld “stack” en “name”. Resources binnen dezelfde stack hebben namelijk vaak een gelijkaardige naam.



Figuur 3.2: Aantal toegevoegde vereisten voor elke combinatie van heuristieken in de document processing use case

Als een bepaalde heuristiek eerst toegepast wordt kan hij al zijn vereisten toevoegen. Daaropvolgende heuristieken zullen enkel het deel van hun vereisten toevoegen die nog niet deel uitmaken van het model. Ongeacht de volgorde zal een combinatie heuristieken altijd dezelfde set vereisten toevoegen.

In afbeelding 3.3 is te zien welke impact de toegevoegde vereisten hebben op het uitrolproces.



FIGUUR 3.3: Gemiddelde aantal uitgerolde resources per deployment run voor elke combinate van heuristieken in de document processing use case

De resultaten bevestigen onze aanname dat het toevoegen van vereisten kan leiden tot een daling in het aantal deployment runs. Elke combinatie van de drie heuristieken leidt zelfs tot een "one-shot" uitrolproces waarin in één keer het volledige model correct uitgerold wordt.

3.5.2 MongoDB

MongoDB is een van de meer bekende NoSQL databases. Een volledige configuratie bestaat uit verschillende services die elkaar ondersteunen. Figuur 3.4 geeft een schematische voorstelling van een dergelijke set-up.

cite

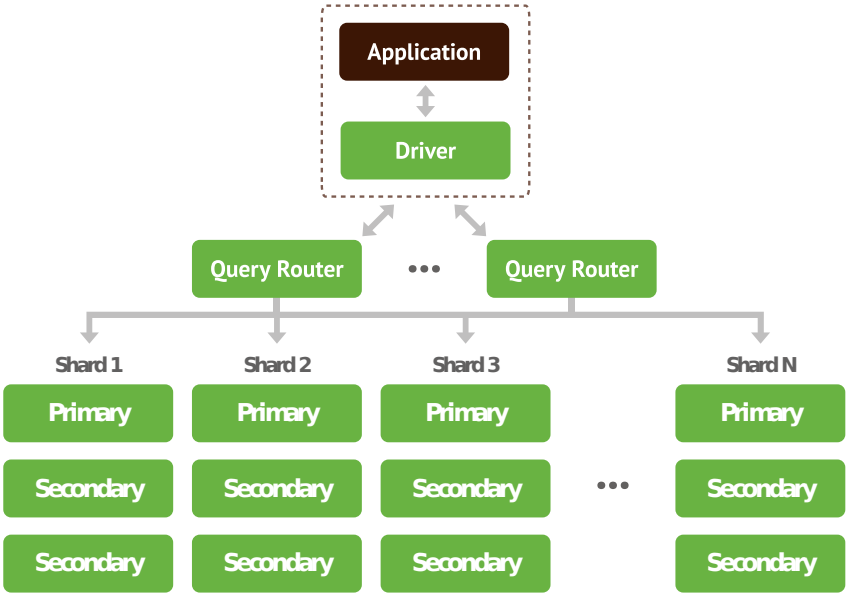
In het model dat Thomas Uyttendaele opgesteld heeft krijgen de onderdelen de volgende namen:

cite

Query Router	AccessServer
Primary	ReplicaSetController
Secondary	Node

De onderdelen zullen pas correct kunnen samenwerken als ze in de juiste volgorde worden opgestart. Het is dus weerom van groot belang dat alle afhankelijkheden in

Vroeger vermelden dat het van groot belang is dat het model compleet is

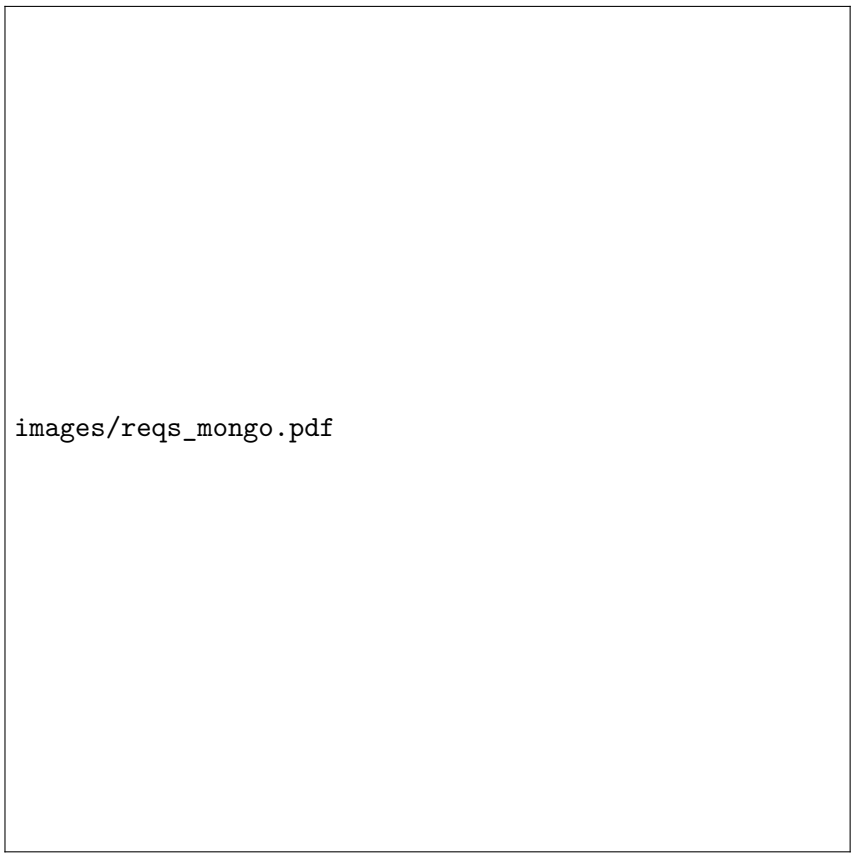


Figuur 3.4: Architectuur van de MongoDB database

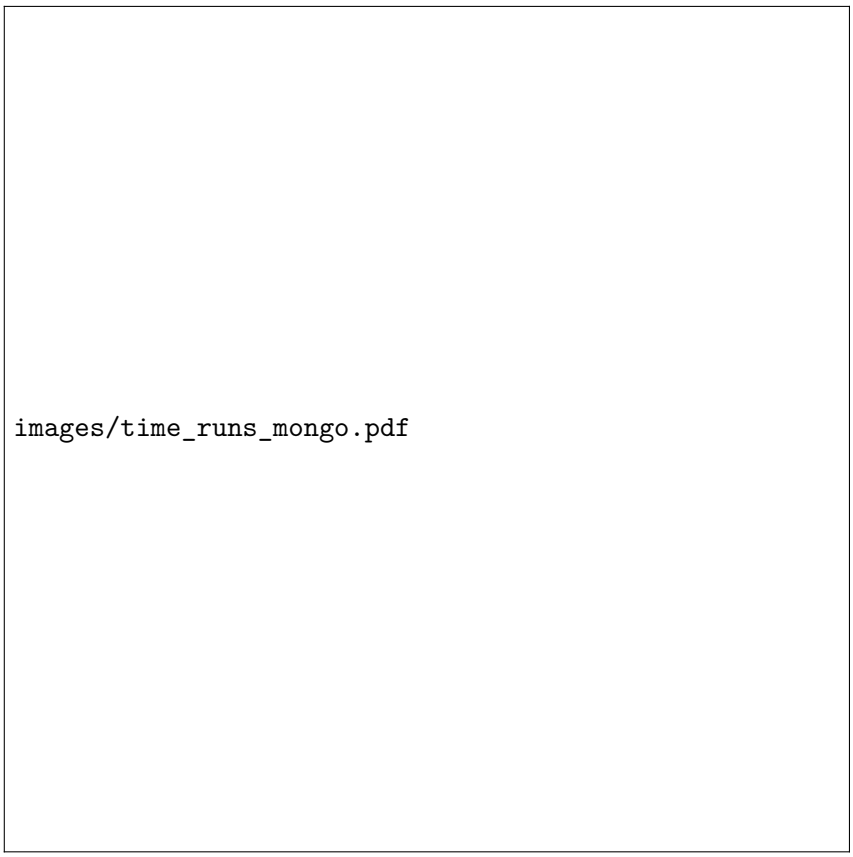
het model vermeld worden.

Voor de volgende resultaten werd een instantie van MongoDB uitgerold met vijf nodes, verdeeld in twee replicasets van twee en drie nodes elk. Drie nodes nemen ook de rol van Query Router op zich. Figuur 3.5 toont de vereisten die elke combinatie van heuristieken toevoegt.

Figuur 3.6 toont hoeveel deployment runs nodig zijn om een volledig werkende MongoDB database te bekomen, en hoeveel resources er per run gedeployed worden.



FIGUUR 3.5: Aantal toegevoegde vereisten voor elke combinatie van heuristieken bij MongoDB



Figuur 3.6: Gemiddelde aantal uitgerolde resources per deployment run voor elke combinate van heuristieken bij MongoDB

Hoofdstuk 4

Besluit

4.1 Gerelateerd werk

verwijzen naar overzichtspaper
IBM

4.2 Verder werk

Een eerste taak is het testen van de heuristieken op enkele andere use cases om zo hun validiteit verder te bevestigen. Verder kan het bekijken van andere cases ook leiden tot het vinden van nieuwe heuristieken. Een deel van de heuristieken werden namelijk geïntroduceerd na onderzoek van de document processing use case.

Een tweede onderzoeksvraag die verder kan onderzocht worden is wat de minimale set vereisten is om een model correct uit te rollen. De huidige heuristieken voegen liever een extra vereiste toe dan één te weinig. Deze aanpak is goed zolang er geen incorrecte of overbodige vereisten worden toegevoegd. Elke vereiste moet verwerkt worden door IMP en zorgt dus voor een kleine verhoging in de uitvoertijd. Een minimale set vereisten heeft dus een positief effect op de totale uitroltijd.

In tegenstelling tot andere CMS laat IMP toe afhankelijkheden en relaties te specifiëren. Deze thesis gebruikt die extra informatie om het uitrolproces te optimaliseren. Het is aan de creativiteit van de gebruiker om nog andere doeleinden te vinden. Een mogelijkheid is uitzoeken hoe beter omgegaan kan worden met resources die in één versie van het model staan maar niet meer in een volgende versie. Momenteel stopt alle CMS dan met het onderhoud van die resource. In sommige gevallen kan dit ongewenste gevolgen hebben: een database die online blijft maar een niet meer geupdated wordt is een veiligheidsrisico. De service stoppen als ze niet meer in het model vermeld staat zou hier een betere optie zijn. Hierbij kan bijvoorbeeld de database resource weten of hij nog in de configuratie van zijn client staat door te kijken naar de relatie die beide hebben. Dit is maar één voorbeeld van wat de extra informatie in het model toelaat.

zoeken van cycles in dependencies

Bijlagen

Bijlage A

De eerste bijlage

In de bijlagen vindt men de data terug die nuttig kunnen zijn voor de lezer, maar die niet essentieel zijn om het betoog in de normale tekst te kunnen volgen. Voorbeelden hiervan zijn bronbestanden, configuratie-informatie, langdradige wiskundige afleidingen, enz.

Fiche masterproef

Student: Harm De Weirdt

Titel: Configuratieafhankelijkheden gebruiken om gedistribueerde applicaties efficiënt te beheren in een hybride cloud

Engelse titel: Configuratieafhankelijkheden gebruiken om gedistribueerde applicaties efficiënt te beheren in een hybride cloud

UDC: T134

Korte inhoud:

Context

Om IT infrastructuren efficient te beheren wordt er gebruik gemaakt van configuratiebeheergereedschappen. Deze gereedschappen zijn model gebaseerd, waarbij het model de gewenste toestand van de configuratie beschrijft. Om de configuratie door te voeren wordt de gewenste toestand vergeleken met de huidige toestand en worden de nodige acties afgeleid die nodig zijn om de infrastructuur in die gewenste toestand te brengen. Huidge systemen zijn reeds in staat om eenvoudige afhankelijkheden af te leiden. Bijvoorbeeld dat een service eerst genstalleerd moet worden voordat die service gestart kan worden. Wat ontbreekt is afhankelijkheden tussen services op verschillende systemen in rekening brengen.

Doel

Het doel van deze thesis is onderzoeken hoe afhankelijkheden in een configuratiemodel gebruikt kunnen worden om de initiele configuratie en mogelijke herconfiguraties van een hybrid cloud zo efficient mogelijk uit te voeren.

Onderzoeksvragen

1. Hoe kunnen afhankelijkheden in een configuratiemodel gebruikt worden om veranderingen zo snel mogelijk uit te rollen?
2. Kan de gevraagde tijd gesimuleerd worden in functie van het configuratie model?

Uitwerking

Fase 1 Vertrouwd geraken met het configuratiebeheergereedschap

Fase 2 Onderzoeken van bestaande configuratiemodellen

Fase 3 Implementeren van een oplossing

Fase 4 Valideren van de oplossing door middel van de configuratiemodellen op een private en publieke cloud en een simulator

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdspecialisatie Gedistribueerde systemen

Promotor: Prof. dr. ir. Wouter Joosen

Assessoren: Ir. W. Eetveel
W. Eetrest

Begeleider: Ir. Bart Vanbrabant