KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

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

Harm De Weirdt

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, Dr. Dimitri Van Landuyt

Academiejaar 2013 – 2014

© Copyright KU Leuven

Zonder voorafgaande schriftelijke toestemming van zowel de promotor als de auteur is overnemen, kopiëren, gebruiken of realiseren van deze uitgave of gedeelten ervan verboden. Voor aanvragen tot of informatie i.v.m. het overnemen en/of gebruik en/of realisatie van gedeelten uit deze publicatie, wend u tot het Departement Computerwetenschappen, Celestijnenlaan 200A bus 2402, B-3001 Heverlee, +32-16-327700 of via e-mail `info@cs.kuleuven.be`.

Voorafgaande schriftelijke toestemming van de promotor is eveneens vereist voor het aanwenden van de in deze masterproef beschreven (originele) methoden, producten, schakelingen en programma's voor industrieel of commercieel nut en voor de inzending van deze publicatie ter deelname aan wetenschappelijke prijzen of wedstrijden.

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

Herschrijven naar opmerkingen

Configuratiemanagementsoftware is een vereiste geworden bij het onderhoud van grote gedistribueerde systemen. Deze software gebruikt een declaratief model van de gewenste configuratie om de nodige resources uit te rollen. In dit model staan de verschillende resources die deel uitmaken van de opstelling. Tussen deze resources kunnen relaties in de vorm van 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. De huidige configuratiesoftware laat enkel toe vereisten tussen enkelvoudige resources te specificeren. 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 aan de hand van heuristieken extra vereisten en afhankelijkheden kunnen toegevoegd worden om zo het aantal uitrolprocessen te reduceren.

Uit de resultaten blijkt dat het gebruik van heuristieken succesvol het aantal uitrolprocessen reduceert. Toegepast op een use case van een complex document processing systeem is slechts één proces meer nodig.

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.[1] Er bestaan reeds verschillende oplossingen voor dit probleem: een eerste manier is het gebruik van scripts. Dit automatiseert al een deel van het werk, maar heeft ook een belangrijk nadeel: als de uitvoering van een script afgebroken wordt blijft het systeem in een onstabiele toestand.[2]

Een tweede 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. [3, 4]

Configuratiebeheergereedschappen (of CMS: Configuration Management Software, vanaf nu zal deze term gebruikt worden) zoals Ansible[5], Puppet[6], CFEngine[7],... 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 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 zijn 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.

Is PHP een vereiste of een afhankelijkheid? Vereiste definieren als altijd binnen 1 host?

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.

In tegenstelling tot het voorbeeld van de LAMP-stack zal de gebruiker direct na het uitrollen al zien dat er iets foutgaat. De CMS zal namelijk melden 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 en opgestart, maar toch werkt de uiteindelijke configuratie niet

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

omdat een foute volgorde werd gehanteerd. In beide gevallen is een extra deployment run nodig voordat de configuratie correct werkt.

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 en vereisten die bestaan, maar niet aanwezig zijn in het model.

Als de CMS weet heeft van de afhankelijkheden en vereisten kan het aantal uitrolmomenten drastisch verminderd worden.

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 heeft dus enkel weet van de eigen resources, en er is ook geen communicatie tussen machines onderling. 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 met de huidige tools niet meer aangeven. De enige optie is dan meerdere keren het model uitrollen tot de configuratie correct werkt. [1]

[1] Voor Puppet bestaat er wel een workaround, zie [8]. "Orchestration" CMS zoals Ansible[5] kan dit wel.

IMP (Infrastructure Management Platform)[9] is een nieuwe tool die momenteel nog in ontwikkeling is. Ze verschilt op twee vlakken van de andere CMS.

Ten eerste kan de gebruiker naast vereisten ook relaties specificeren. Deze relaties worden gedefiniëerd tussen twee entiteiten. Entiteiten in IMP zijn de klasses uit een objectgerichte programmeertaal, en resources zijn de instanties van dergelijke klasses. Relaties gelden dus voor alle instanties van een entiteit, in tegenstelling tot vereisten die tussen twee specifiece resources gespecificeerd worden. De gebruiker kan zelf aangeven of een relatie al dan niet afhankelijk is.

Ten tweede krijgt elke machine tijdens het uitrolproces het volledige model ter beschikking en niet alleen zijn eigen deel. De verschillende machines wisselen ook berichten met elkaar uit over de resources die ze al uitgerold hebben. Machines kunnen zo niet alleen rekening houden met afhankelijkheden en vereisten tussen eigen resources, maar ook met andere machines.

Deze twee eigenschappen samen zorgen ervoor dat in IMP afhankelijkheden tussen samengestelde entiteiten die op verschillende machines gedefiniëerd zijn verwerkt kunnen worden. Een voorbeeld hiervan is de afhankelijkheid tussen de webserver en databaseserver uit de gedistribueerde versie van de LAMP-stack. Deze kan niet gemodelleerd worden in de andere CMS.

Het is duidelijk dat het belangrijk is om alle relaties (zowel afhankelijkheden en vereisten) die aanwezig zijn tussen resources van een gedistribueerde infrastructuur expliciet te vermelden in het configuratiemodel. Deze thesis onderzoekt hoe aan de hand van heuristieken automatisch deze relaties kunnen toegevoegd worden aan het configuratiemodel. Dit vermindert de moeite die gebruiker van de software moet doen bij het opstellen van het model, en vermindert mogelijks het aantal fouten dat gebeurt tijdens het uitrollen. Daartegenover staat wel dat als de gebruiker alle functionaliteit van IMP wil benutten hij/zij ook de relaties tussen de gedefiniëerde entiteiten zal moeten specifiëren.

Als IMP een optimale volgorde gebruikt bij het uitrollen van een model kan na één uitrolproces een volledig functionerende infrastructuur opgestart worden. Elke uitrolvolgorde die resulteert in een volledig werkende configuratie na één uitrolmoment wordt als optimaal aanzien.

De veronderstelling is wel dat degene die het model opstelt genoeg domeinspecifieke informatie aanlevert die door deze heuristieken gebruikt kan worden.

Verschil opzoeken en uitleggen hier of in related works?

Hoofdstuk 2

IMP

Dit deel geeft een introductie in configuratiemanagementtools en IMP specifiek. Configuratietools laten toe op een efficiënte manier complexe IT-infrastructuren te onderhouden. Sectie 2.1 geeft een inleiding tot configuratiemanagement. Sectie 2.2 geeft meer uitleg over hoe IMP werkt en op welke vlakken dit verschilt met de werking van de huidige tools.

2.1 Configuratiemanagement

2.2 Werking

Deze sectie beschrijft hoe een IMP-opstelling er uitziet en hoe een model opgesteld en uitgerold wordt. Sectie 2.2.1 beschrijft de verschillende onderdelen van een gedistribueerd systeem dat gebruikt maakt van IMP, en hoe deze interageren. Sectie 2.2.2 geeft uitleg over het declaratief model dat de gebruiker opstelt. Er wordt uitleg gegeven over de structuur van zo'n model, welke elementen er in voorkomen en hoe die elementen gevormd worden. Daarna volgt uitleg over de verwerking van dat model tijdens het uitrolproces.

2.2.1 Operationele structuur

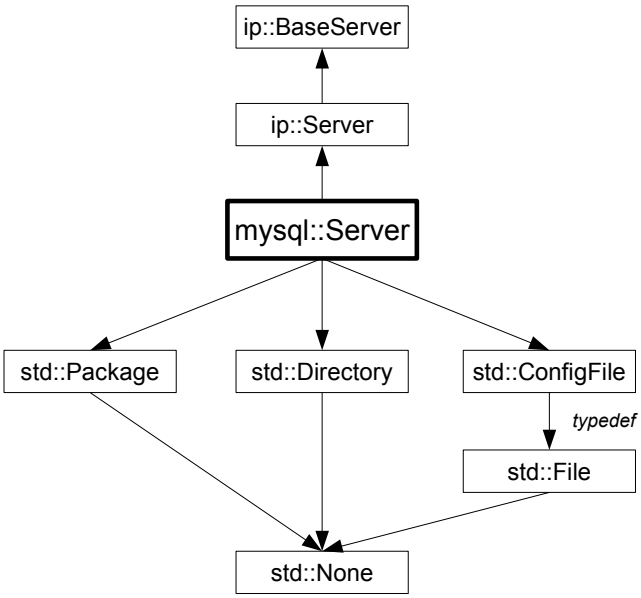
Structureel bestaat IMP uit één management server en verschillende agents. De management server houdt het configuratiemodel bij. Als de gebruiker het model wil exporteren zal de management server eerst dat model compileren en dan naar de verschillende agents versturen.

2.2.2 Opstellen van het model

Het configuratiemodel van IMP kan beschreven worden in twee delen: het configuratiemodel zelf en de bibliotheken. In het configuratiemodel staat de infrastructuur die de gebruiker wenst te bekomen. Deze bestaat uit een oplijsting van machines en de resources die moeten aanwezig zijn op die machines. Deze resources kunnen in complexiteit variëren van een simpele file tot een volledige webserver.

De verschillende types resources worden gedefiniëerd als entiteiten in bibliotheken. Zo is er de standaardbibliotheek “std” met onder andere de definitie van een host, een bestand, een service,... Het is belangrijk een duidelijk onderscheid te maken tussen entiteiten en implementaties uit bibliotheken en resources gedefiniëerd in een configuratiemodel. Entiteiten definiëren de naam en attributen van een domeinconcept. De implementatie van een entiteit specifiëert uit welke andere entiteiten ze is opgebouwd.

De resources vermeld in het model zijn instanties van de entiteiten uit de bibliotheken. Een entiteit kan één of meerdere reeds bestaande entiteiten overerven. Zo kan de gebruiker makkelijk verschillende entiteiten combineren, zonder limiet op het aantal niveaus van overerving. Een visuele voorstelling van de overerving en de resources binnen een entiteit en zijn implementatie is te vinden op figuur 2.1. Hier is duidelijk te zien dat er sprake is van twee hiërarchieën: de gedefiniëerde entiteiten kunnen van elkaar overerven en vormen zo een opwaartste hiërarchie. Een implementatie is een lijst van geïnstantieerde enteteiten (resources dus) die elk weer kunnen bestaan uit verzamelingen van resources. Dit vormt een neerwaartse hiërarchie.



Figuur 2.1: Voorstelling van de hiërarchie die aanwezig is tussen de verschillende entiteiten. De mysql-server erft over van de Server -en BaseServerentiteit uit de ip-bibliotheek. De resources gedefiniëerd binnen de implementatie van een mysqlserver erven ook over van andere entiteiten.

Een voorbeeld van een eenvoudige bibliotheek is deze van de MySQL-database:

```
1 \begin{minipage}
2 \begin{lstlisting}[label=listing:mysql_model]
3 entity MysqlServer extends ip::services::Server:
4   """
```

```
5       Mysql server configuration
6     """
7   end
8
9   entity Database:
10    string name
11    string user
12    string password
13  end
14
15  MysqlServer server [1] -- [0:] Database databases
16
17  typedef Server as MysqlServer(services = mysql::s_server)
18
19  mysql_server_range = ip::Port(low = 3306)
20  s_server = ip::DstService(proto = "tcp", dst_range = mysql_server_range)
21
22  implement Database using std::none
23  implement MysqlServer using mysqlRedhat when host.os is "redhat"
24
25  implementation mysqlRedhat for MysqlServer:
26    # install mysql server
27    pkg = std::Package(host = host, name = "mysql-server", state = "installed")
28    svc = std::Service(host = host, name = "mysqld", state = "running", onboot =
        true)
29    svc.requires = [pkg, config, conf_dir]
30
31    config = std::ConfigFile(host = host, path = "/etc/my.cnf", content =
        template("mysql/my.cnf.tmpl"), requires = pkg, reload = true)
32    conf_dir = std::Directory(host = host, path = "/etc/mysql.conf.d", owner = "
        root", group = "root", mode = 755)
33
34    dblist = std::ConfigFile(host = host, path = "/etc/sysconfig/mysql", reload
        = true,
35      content = template("mysql/databases.tmpl"))
36  end
37  \end{lstlisting}
38  \end{minipage}
```

Net zoals de meeste bibliotheken begint deze met een oplijsting van de verschillende entiteiten die gedefiniëerd worden. Zoals daarnet gezegd kunnen entiteiten opgebouwd worden aan de hand van andere entiteiten. Hier wordt een MysqlServer entiteit gedefiniëerd als een uitbreiding op de Server entiteit uit de ip bibliotheek. Entiteiten kunnen attributen hebben, in dit geval heeft een Database een naam, een gebruiker en een wachtwoord. Daarbovenop komen nog de attributen die gedefiniëerd zijn voor een Server.

Regel 13 toont een belangrijke feature van IMP: relaties. Relaties zijn een vorm van domeinspecifieke informatie over hoe twee entiteiten in relatie staan met elkaar. In dit geval weet de gebruiker dat voor elke MySQL-server nul of meerdere databases moeten gespecifiëerd worden, en dat voor elke database exact één MySQL-server moet gespecifiëerd worden. Aan de hand van deze informatie kan IMP onder andere bij het compileren nakijken of aan deze relatie voldaan is en zoniet dit melden aan de gebruiker. Dit reduceert de kans op configuratiefouten. Concreet zal in het configuratiemodel naast de gewone attributen van een mysqlserver of een database nog een extra attribuut moeten opgegeven worden: respectievelijk de database of server waarmee ze in relatie staat. Andere CMS laat niet toe dit soort informatie op

te geven en kunnen dit soort checks dus niet doen. Deze relaties zullen in sectie 3.4 ook gebruikt worden om het uitrolproces te optimaliseren.

Het laatste deel van een bibliotheek bestaan uit de implementaties van de ervoor gedefiniëerde entiteiten. In deze implementaties worden de entiteiten opgebouwd aan de hand van instanties van andere entiteiten. In dit geval bestaat een MysqlServer-entiteit uit een instantie van het mysql-server pakket, de mysqld-service en een paar configuratiebestanden-en mappen. Deze komen allemaal uit de std-bibliotheek. In de declaratie van deze resources is ook te zien hoe de attributen hun waarden krijgen. Op regel 30 staat de declaratie van de mysqld-service. De attributen “name”, “state” en “onboot” worden in de bibliotheek zelf opgegeven. Het attribuut “host” wordt opgegeven door de gebruiker bij het invullen van de attributen van een MysqlServer in het configuratiemodel.

Een voorbeeld van een configuratiemodel staat in listing 2.2.2. Dit model stelt een installatie van Drupal voor op één host. Voor Drupal zijn een werkende web -en databaseserver nodig.

```
1  \begin{minipage}
2  \begin{lstlisting}[label:listing:drupal_main]
3  # define the machine we want to deploy Drupal on
4  vm1 = ip::Host(name = "server", os = "fedora-18", ip = "172.16.34.14")
5
6  # add a mysql and apache http server
7  web_server = httpd::Server(host = vm1)
8  mysql_server = mysql::Server(host = vm1)
9
10 # define a new virtual host to deploy drupal in
11 vhost_name = httpd::VhostName(name = "localhost")
12 vhost = httpd::Vhost(webserver = web_server, name = vhost_name, document_root
       = "/var/www/html/drupal_test")
13
14 # deploy drupal in that virtual host
15 drupal::Common(host = vm1)
16 db = mysql::Database(server = mysql_server, name = "drupal_test", user = "
       drupal_test", password = "Str0ng-P433w0rd")
17 drupal::Site(vhost = vhost, database = db)
18 \end{lstlisting}
19 \end{minipage}
```

Regel 1 toont dat dit model een host bevat met een bepaalde hostname (“server”) en een bepaald IP-adres. Alle elementen van de drupal installatie zullen geplaatst worden op deze host.

De volgende 2 regels specificeren dat op de daarnet gedefiniëerde host vm1 een httpd-server en een mysql-database moeten aanwezig zijn. Zoals daarnet besproken moet enkel nog het “host”-attribuut van een mysql-server ingevuld worden door de gebruiker, de andere attributen werden in de bibliotheek al ingevuld. De declaratie van de drupalsite zelf (regel 15) toont ook duidelijk dat relaties ook attributen introduceren: zowel “vhost” als “database” zijn attributen die door een relatie gespecificeerd worden.

belang van opgeven van alle informatie

2.2.3 Uitrollen van het model

Als het model volledig opgesteld is kan de gebruiker het uitrollen. Dit proces bestaat uit meerdere stappen. Eerst komt een kort overzicht van het proces, waarna dieper wordt ingegaan op elke stap.

Zoals gezegd bestaat het uitrolproces uit meerdere stappen:

1. Compilatie van het model
   a) IMP vereenvoudigt de resources stap per stap tot de basis (bestanden, mappen, pakketten en services)
   b) De verschillende dependency managers worden opgeroepen en doen verdere aanpassingen aan het model
2. verdeling van het model
3. Toepassen van configuratieaanpassingen
   a) De resources die geen vereisten hebben worden uitgerold
   b) De uitgerolde resources worden verwijderd uit de vereisten van andere resources
   c) Stap 1 en 2 worden herhaald tot alle resources uitgerold zijn.

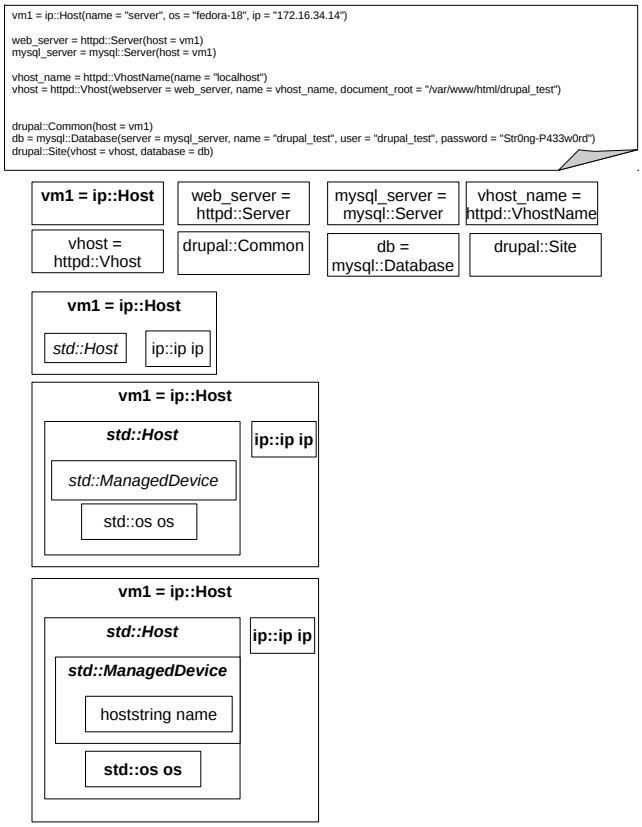
vereisten/relaties vermelden

Compilatie van het model In de compilatiestap zal IMP de samengestelde resources stapsgewijs omzetten tot basisresources zoals bestanden, pakketten en services. Zoals in sectie 2.2.2 vermeld laat IMP namelijk toe om entiteiten te combineren om zo herbruikbare concepten te definiëren. (Zie ook figuur 2.1.) In de compilatiestap wordt dus de omgekeerde weg terug gevolgd. Dit proces wordt "concept refinement" genoemd en wordt grafisch voorgesteld op figuur 2.2.

Als het laagste niveau bereikt is en enkel nog basisresources overblijven is de eerste stap van het compilatieprocess af. Nu krijgen de dependency managers de kans om nog aanpassingen te doen aan het model. Dependency managers zijn Python functies die de gebruiker kan schrijven als deel van een bibliotheek. Deze krijgen als argumenten het hoogste niveau van het model, het configuratiemodel, en de lijst met uiteindelijke resources, het resourcemodel.Daarmee kan de gebruiker extra aanpassingen uitvoeren op de modellen voordat het verder uitgerold wordt. In deze thesis worden de dependency managers gebruikt om de gevonden heuristieken toe te passen.

termen vroeger vermelden

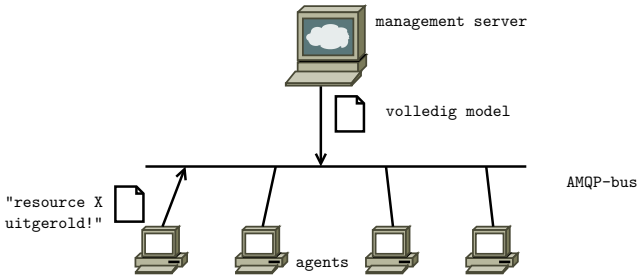
verdeling van het model Eens het model opgesteld is moet elke host zijn deel van het model met de relevante resources ontvangen. De communicatie tussen de management server en de verschillende agents die actief zijn op de hosts gebeurt via een AMQP-bus (zie figuur 2.3). De agents schrijven zich in voor alle resources die gedefiniëerd zijn voor hun host en ontvangen zo enkel het voor hen relevante deel van het model. Als een agent een resource uitgerold heeft zet hij ook een bericht op de AMQP-bus zodat andere agents weten als een vereiste voldaan is. In tegenstelling



Figuur 2.2: Voorstelling van hoe de hoger-niveau resources worden gerefined tot lager-niveau resources en hun attributen.

tot andere CMS is er dus communicatie tussen de verschillende agents, wat toelaat op een gedistribueerde manier de vereisten correct af te handelen.

Vergelijking maken met andere CMS? Hier of in related work?



Figuur 2.3: Deployment model van IMP met de management server en de agents, allemaal verbonden via een AMQP-bus

Hoofdstuk 3

Naar het automatisch toevoegen van vereisten en afhankelijkheden

Configuratiebeheergereedschappen die gebruik maken van vereisten en afhankelijkheden reduceren het aantal fouten tijdens het uitrolproces en zo het aantal deployment runs dat nodig is om een stabiele toestand te bereiken. Dit hoofdstuk introduceert enkele heuristieken die proberen extra vereisten en afhankelijkheden toe te voegen aan het configuratiemodel.

De heuristieken zijn deels geïnspireerd door situaties die in de inleiding al voorgesteld werden. Secties 3.1 beschrijft een heuristiek die garandeert dat bestanden en mappen altijd in juiste volgorde gecreëerd worden. Sectie 3.2 en 3.3 introduceren heuristieken die vereisten toevoegen tussen logisch samenhorende resources. De heuristieken uit secties 3.4 en 3.5 werken samen om relaties om te zetten in vereisten om zo een efficiëntere volgorde opleggen aan het uitrolproces.

3.1 Vereisten tussen bestanden en mappen

Deze heuristiek garandeert dat elk bestand pas aangemaakt wordt na zijn bovenliggende map. Als namelijk een omgekeerde volgorde gehanteerd wordt mislukt het uitrollen van het bestand. Figuur 3.1 geeft hiervan een visuele voorstelling.

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

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

Als de map niet vermeld wordt in het model wordt deze ook niet toegevoegd omdat dit ongewenste gevolgen kan hebben. Dit zou namelijk fouten verbergen die de gebruiker kan maken bij het opstellen van het model. Als hij bijvoorbeeld een bestand plaatst in de configuratiemap van een pakket (bvb “/etc/ldap”) maar het

Figuur 3.1: Grafische voorstelling van de afhankelijkheid tussen een bestand en zijn bovenliggende folder

pakket zelf (“ldap”) vergeet toe te voegen aan het model krijgt hij geen foutmelding. De map wordt namelijk automatisch aangemaakt.

3.2 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 3.2.

Figuur 3.2: 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:

```
1 for service in host.resources:
2   for file in host.resources:
3     service.requires.add(file)
4   for package in host.resources:
5     service.requires.add(package)
6 for file in host.resources:
7   for package in host.resources:
8     file.requires.add(package)
```

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

De volgende aanpak gebruikt meer info uit het model en resulteert in minder overbodige afhankelijkheden. In het model worden bestanden, pakketten en services die deel uitmaken van een geheel gegroepeerd in een "implementatie". Een voorbeeld is de MySQL server:

```
1 implementation mysql:
2   pkg = std::Package(host= host, name= "mysql-server", state= "installed")
3   svc = std::Service(host= host, name= "mysqld", state= "running", onboot=
      true)
4
5   config= std::ConfigFile(host= host, path= "/etc/my.cnf", content= template("
      mysql/my.cnf.tmpl"), requires= pkg, reload= true)
6   conf_dir= std::Directory(host= host, path= "/etc/mysql.conf.d", owner= "root
      ", group= "root", mode= 755)
7
8   dblist= std::ConfigFile(host= host, path= "/etc/sysconfig/mysql", reload=
      true, content= template("mysql/databases.tmpl"))
9 end
```

Alles dat staat tussen "implementation mysql" en "end" is gedefiniëerd binnen éénzelfde scope. Deze 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. Resultaten van het gebruik van deze heuristiek staan in sectie 4.3.

3.3 Vereisten tussen resources met gelijkaardige naam

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

Om de vereisten tussen services en de bijhorende pakketten en bestanden op te stellen wordt de naam van de service gebruikt.

Om de vereisten tussen pakketten en de configuratiebestanden op te stellen worden twee aanpassingen gedaan aan de naam van het pakket. Allereerst worden alle cijfers uit de naam van het pakket verwijderd. Daarna wordt alles na een splitsingsteken verwijderd. Zo stelt de heuristiek ook afhankelijkheden op tussen bijvoorbeeld het pakket “cassandra12” en het configuratiebestand “/etc/cassandra.conf” of het pakket “openssh-server” en “/etc/openssh.conf”.

waarom niet gewoon naam van service gebruiken? Truukjes zijn dan niet nodig.

```
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)
```

3.4 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

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

Dit betekent dat een Host nul of meerdere files kan bevatten, maar dat elke File op maximaal één Host kan uitgerold worden. De heuristiek leidt hieruit af dat een File niet kan bestaan zonder een host en het dus nodig is dat eerst de Host wordt uitgerold voordat de File wordt aaangemaakt.

ander woord zoeken

Algemeen kan men 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:

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

Als enkel deze heuristiek gebruikt wordt zal IMP niets veranderen aan het uitrolproces. IMP doet zelf niets met de gedefiniëerde relaties. De extra informatie kan wel gebruikt worden door andere heuristieken. Een voorbeeld hiervan is deze hieronder (sectie 3.5) die afhankelijke relaties omzet in vereisten.

3.5 Vereisten door afhankelijkheden

Deze heuristiek heeft als doel afhankelijke relaties om te zetten in een concrete vereiste tussen resources. Samengestelde entiteiten zoals een webserver bestaan uit verschillende resources: configuratiebestanden, pakketten en services. Als een entiteit afhankelijk is van een andere betekent dat dat hij zijn volledige functionaliteit niet kan aanbieden als de andere entiteit nog niet volledig uitgerold is. Het aanbieden van functionaliteit gebeurt door services, niet door de aanwezigheid van bestanden of pakketten. Een afhankelijkheid tussen twee entiteiten kan dus gereduceert worden tot een vereiste tussen de services van de twee entiteiten.

Deze heuristiek zoekt naar afhankelijke relaties en voegt de gepaste vereisten toe tussen de services die in relatie staan met elkaar. 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 3.4, daar worden namelijk extra afhankelijke relaties opgesteld die hier kunnen gebruikt worden.

3.6 Besluit van dit hoofdstuk

Hoofdstuk 4

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 4.1 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

4.1 Simulator

Als deel van deze thesis werd een simulator voor IMP ontwikkeld. Deze heeft een belangrijk aandeel in het evalueren van de ontwikkelde heuristieken. Ze laat namelijk toe op een snelle en eenvoudige manier een uitgebreid model uit te rollen en geeft volledige controle over welke informatie tijdens dat proces beschikbaar wordt gesteld aan de gebruiker. De snelheid en eenvoud van de simulator in vergelijking met op fysieke hosts uitrollen is te danken aan twee redenen:

ten eerste is de uitroltijd evenredig met het aantal resources in het model. Elke resource moet verwerkt worden, en vooral pakketten duren lang om uit te rollen. Deze moeten niet alleen gedownload worden maar ook geïnstalleerd, beide taken kunnen aardig wat tijd innemen. Bij de simulator komt het uitrollen van een pakket overeen met het wegschrijven van een reeks waarden in een database, wat significant sneller is. Hierbij moet wel een kanttekening gemaakt worden: bij fysiek uitrollen wordt het uitrollen van het model verdeeld over de verschillende hosts, elk zijn eigen deel van het model. De simulator moet alle resources van alle hosts alleen verwerken. Zelfs dan nog is de snelheidswinst bij de simulator significant.

Ten tweede laat een simulator toe modellen met honderden machines uit te rollen op één enkele pc. Naarmate de modellen groter worden wordt het steeds moeilijker het model op fysieke hosts te testen. En het zijn juist de grotere modellen, met meer hosts, die interessant zijn voor de evaluatie van de heuristieken.

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

4.1.1 Uitwerking

De simulator is een Pythonscript dat als invoer een JSON-bestand met daarin het gecompileerde model. IMP zelf stelt dit bestand op als de optie "-j [naam JSON]" wordt meegegeven. Tijdens het uitvoeren van dit script wordt een sqlite-database opgesteld. De tabellen van deze database zijn te zien in de tabellen 4.1.

Agent	
name	Text

Attribute	
name	Text
value	Text
ResourceId	Text

Resource	
Id	Text

Tabel 4.1: De verschillende tabellen die aanwezig zijn in de deploymentdatabase.

In de Attribute tabel staan de attributen opgeslagen als volgt:

```
name     value       ResourceId
-------- ---------- ------------------------------------------------
group    root        std::File[server,path=/tmp/test],v=1389435342
owner    root        std::File[server,path=/tmp/test],v=1389435342
path     /tmp/test   std::Directory[server,path=/tmp/test],v=1389435342
```

Om de simulatie zo waarheidsgetrouw mogelijk te maken houdt de simulator rekening met volgende aspecten van het fysieke 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.

Nakijken of alle bijhorende pakketten en bestanden van een service aanwezig zijn wordt gebruikt gemaakt van de repositorydata van de yum pakketmanager. De repositorydata bevat twee handige databases: de primary en de filelists. De relevante attributen de gebruikte tabellen zijn te vinden in tabel 4.2.

De uiteindelijke database die gebruikt wordt door te simulator bestaat uit de samenvoeging van de attributen uit de twee databases. Aangezien sommige pakketten een eigen yum repository hebben is voor die gevallen manueel de repositorydata toegevoegd. Een voorbeeld van de data die zo bekomen wordt:

```
name          dirname                    filenames
------------  -------------------------  ------------------------------
```

packages	
pkgKey	Integer
name	Text

filelist	
pkgKey	Integer
name	Text
dirname	Text
filenames	Text

Tabel 4.2: De relevante attributen van de repositorydatabases die gebruikt worden door de simulator

```
oniguruma     /usr/lib                  libonig.so.2.0.0/libonig.so.2
oniguruma     /usr/share/doc            oniguruma-5.9.2
openCOLLADA   /usr/share/doc            openCOLLADA-0
```

Bij het installeren van een pakket in de simulator wordt een lijst van alle bestanden en mappen die bij dat pakket horen opgesteld. Elk element van deze lijst wordt als entry toegevoegd aan de Attribute tabel, met als value het pad naar dat bestand. Als de simulator probeert een bepaalde service uit te rollen zoekt hij alle entries met de servicenaam en stelt een lijst met alle nodige bestanden en mappen op. Daarna wordt in de Attribute tabel gecheckt of deze allemaal aanwezig zijn.

Naast de repositorydata wordt ook nog een lijst gebruikt met daarin de bestanden en mappen die standaard aanwezig zijn op een nieuwe Fedora 18 installatie. Zonder deze lijst zou elke poging tot het uitrollen van een bestand of map falen, omdat voor geen enkel bestand de bovenliggende map bestaat.

In pseudecode ziet het uitrollen van elke type resource er uit als volgt:

```
1  def deploy_file(file):
2   #Check if parent folder exists already
3   if parent(file) in std_filesystem or parent(file) in deployment_database:
4    deployment_database.write(file)
5   else:
6    error('Parent folder doesn't exist!')
7
8  def deploy_service(srv):
9   #Check if required files have been deployed
10  requirements = repodata_database.execute('select * from pkgdata where name
       like srv.name')
11  if all([req in deployment_database for req in requirements]):
12   deployment_database.write(srv)
13  else:
14   error('Not all required resources were deployed!')
15
16 def deploy_package(pkg):
17  pkg_files = repodata_database.execute('select * from pkgdata where name like
       pkg.name')
18  foreach file in pkg_files:
19   deployment_database.write(file)
```

Bij het uitrollen van een pakket worden geen controles gedaan. De simulator gaat er van uit dat het correct uitrollen van een pakket de verantwoordelijkheid is van de pakketmanager, niet de CMS.

Om zo goed mogelijk een fysiek uitrolproces te simuleren gebruikt de simulator een gelijkaardig algoritme voor het uitrollen van het model. Daarbij worden eerst

alle resources weggeschreven die geen vereisten hebben. Daarna wordt gekeken of de resources die net zijn uitgerold een vereiste waren van de overblijvende resources. Als dit het geval is worden de geschreven resources verwijderd uit de lijst met vereisten. Zo komen (hopelijk) nieuwe resources vrij die geen vereisten meer hebben en die kunnen in de volgende herhaling uitgerold worden. Dit proces herhaalt zich tot er geen resources meer moeten uitgerold worden.

Verschil in uitroltijd simulator/IMP?

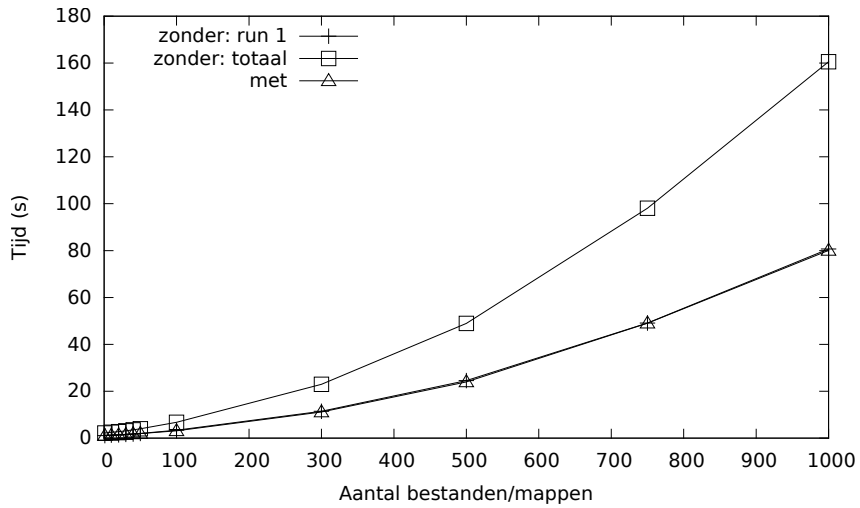
Besluit sectie

4.2 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.[1] 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 4.1 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 ter beschikking. Het model bestaat iedere keer uit een vast aantal bestanden en mappen, één bestand per map.



Figuur 4.1: Testresultaten bij het uitrollen van een stijgend aantal bestanden en mappen, met en zonder gebruik van de heuristiek

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

[1] Twee runs als maar één bovenliggende map moet aangemaakt worden. Per bijkomend niveau is mogelijks een extra run nodig.

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

Tabel 4.3: Meetresultaten

4.3 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 4.3.

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

4.4 Relaties en afhankelijkheden tussen hoog-niveau concepten

De heuristieken uit sectie 3.4 en 3.5 werken best samen. De eerste vormt relaties met een bepaalde multipliciteit om naar afhankelijke relaties. De tweede zet afhankelijke relaties tussen entiteiten die services bevatten om naar vereisten tussen die services. Zo wordt vermeden dat een extra deployment run nodig is de services correct op te starten.

Deze heuristieken werden niet apart getest, enkel in de use cases. In individuele testen zouden de resultaten gelijkaardig zijn aan die van de bestanden en mappen heuristiek.

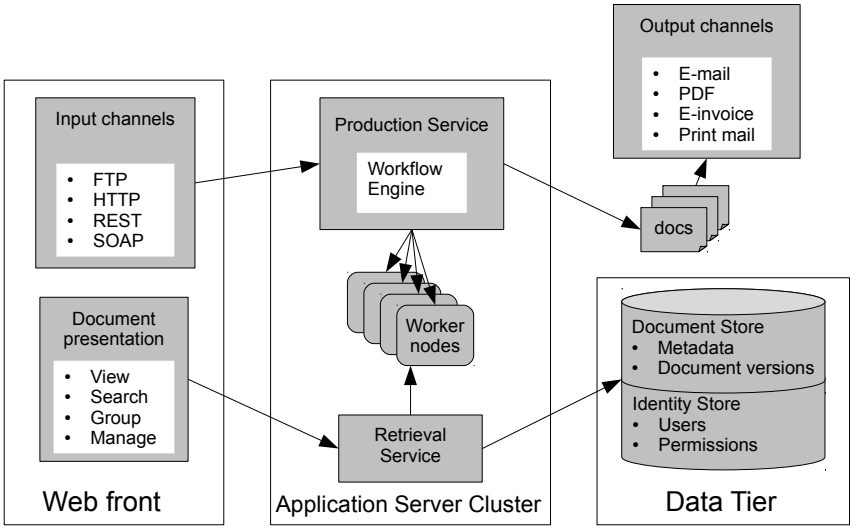
4.5 Use cases

4.5.1 Document processing

Deze eerste use case studie betreft een uitgebreide en complexe infrastructuur gebruikt voor de geautomatiseerde verwerking van documenten. De infrastructuur ondersteunt de creatie, het opstellen van de layout, de processing en het opslaan van bedrijfsdocumenten (bvb facturen, orders, aandelen,...). Ze wordt meestal uitgerold als een cloudservice. Figuur 4.2 toont de verschillende onderdelen van deze

cite

opstelling: de gebruiker kan via verschillende inputkanalen data doorspelen aan de productieservice. Die verwerkt de gegevens in verschillende stappen. De worker nodes zijn verantwoordelijk voor de verwerking van die acties. De resultaten worden geplaatst in een database en kunnen via verschillende outputkanalen opgevraagd worden.



Figuur 4.2: Overzicht van de document processing infrastructuur

bibliotheken vermelden, mogelijks een hoofdstuk introductie tot IMP?

Het volledige model in IMP van deze use case bestaat uit 35 bibliotheken waaronder Apache, HBase, Cassandra, ... In totaal zijn er reeds 82 relaties gespecifiëerd, maar het model is nog niet volledig.

In afbeelding 4.3 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.

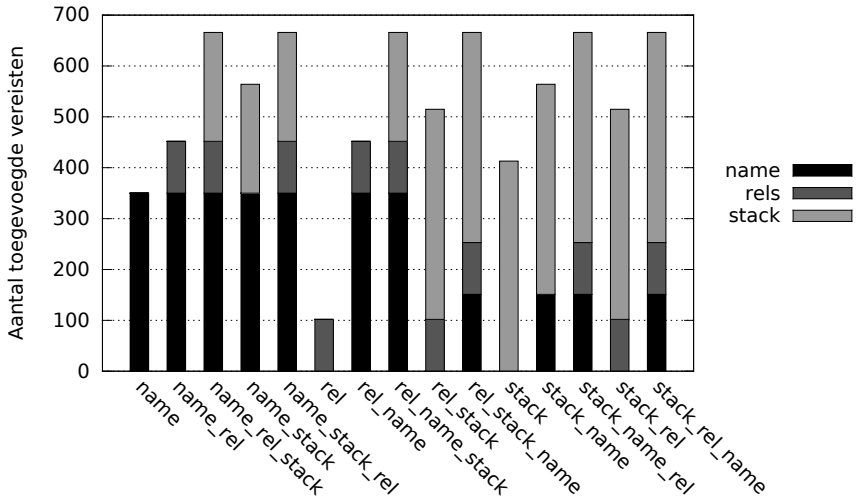
Dit is eigenlijk geen uitleg van de toepassing van de heuristieken op de use case, meer een uitleg/test van de heuristieken

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

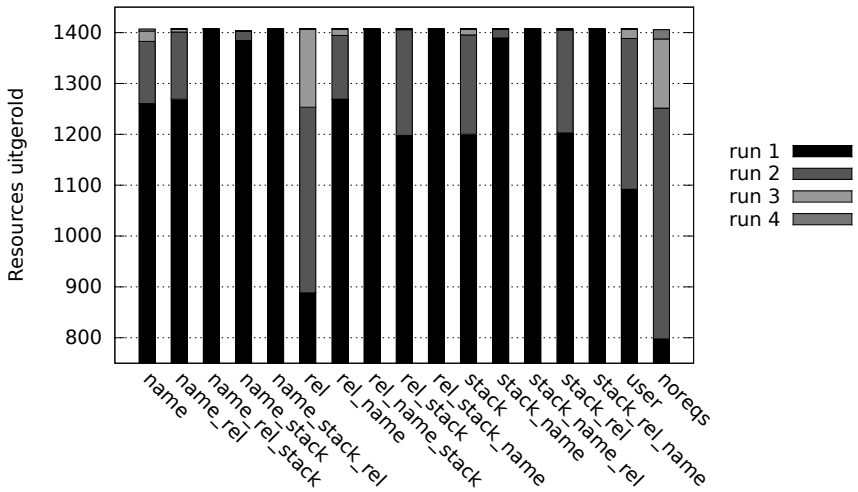
Als een bepaalde heuristiek eerst toegepast wordt kan hij al zijn vereisten toevoegen. Daaropvolgende heuristieken zullen enkel het deel van hun vereisten toevoegen die nog niet deel uitmaken van het model. Ongeacht de volgorde zal een combinatie heuristieken altijd dezelfde set vereisten toevoegen.

In afbeelding 4.4 is te zien welke impact de toegevoegde vereisten hebben op het uitrolproces.

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.



Figuur 4.3: Aantal toegevoegde vereisten voor elke combinatie van heuristieken in de document processing use case



Figuur 4.4: Gemiddelde aantal uitgerolde resources per deployment run voor elke combinate van heuristieken in de document processing use case

Het volledige model bevat 82 relaties.

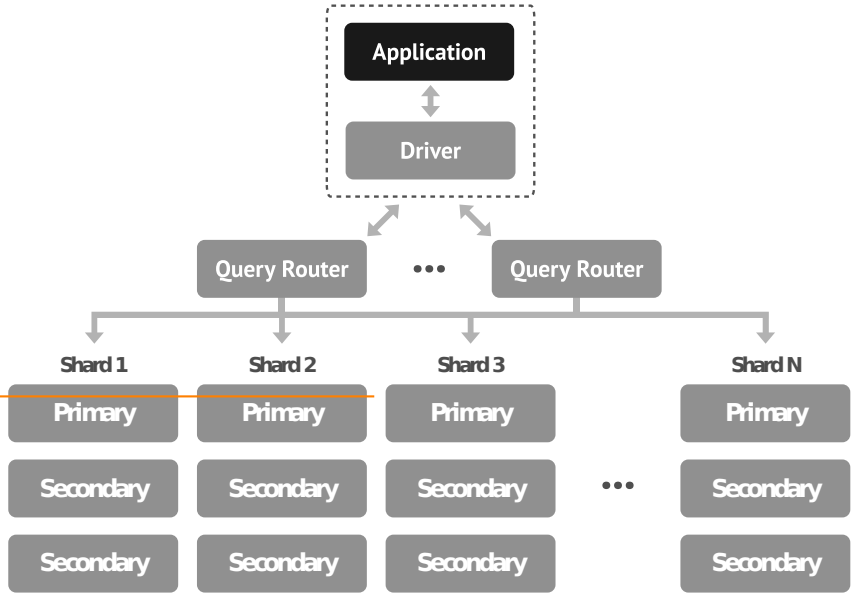
4.5.2 MongoDB

MongoDB is een van de meer bekende NoSQL databases. Een volledige configuratie bestaat uit verschillende services die elkaar ondersteunen. Figuur 4.5 geeft een schematische voorstelling van een dergelijke set-up.

In het model dat Thomas Uyttendaele opgesteld heeft krijgen de onderdelen de

cite

cite



Figuur 4.5: Architectuur van de MongoDB database

volgende namen:

Query Router	AccessServer
Primary	ReplicaSetController
Secondary	Node

De onderdelen zullen pas correct kunnen samenwerken als ze in de juiste volgorde worden opgestart. Figuur 4.6 toont de implementatie van MongoDB in IMP, samen met de volgorde waarin de services gestart moeten worden.

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

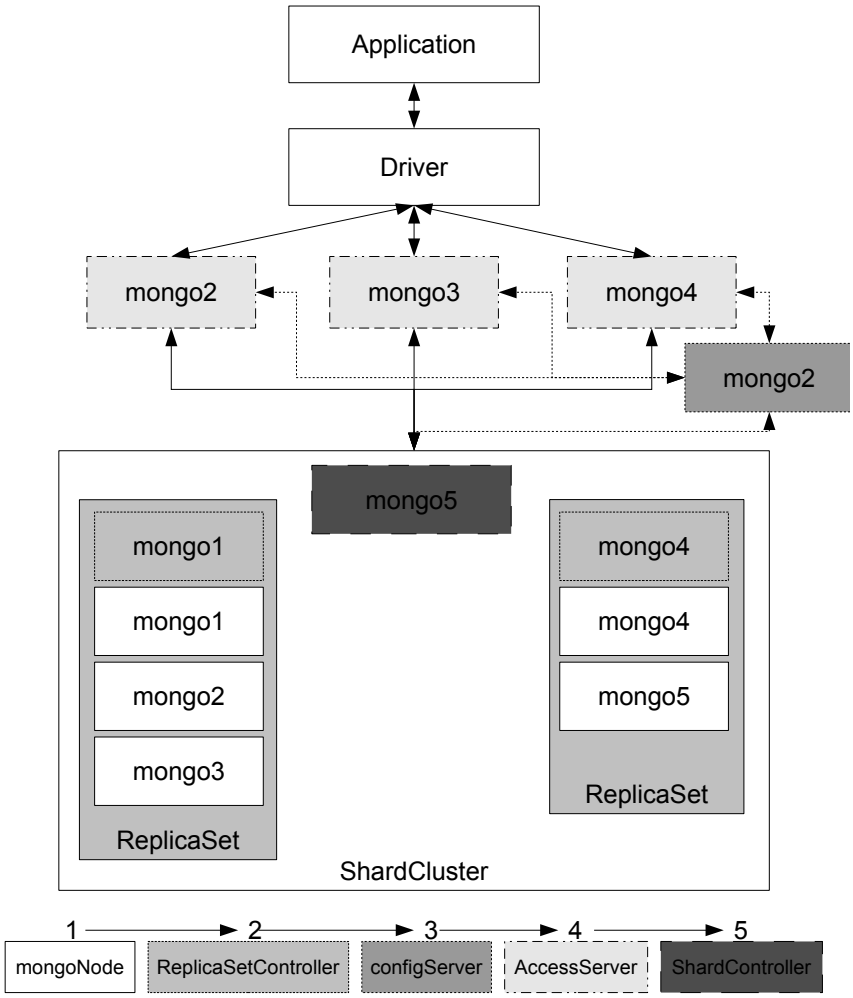
Het is dus weerom van groot belang dat alle afhankelijkheden in het model vermeld worden.

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 4.7 toont de vereisten die elke combinatie van heuristieken toevoegt.

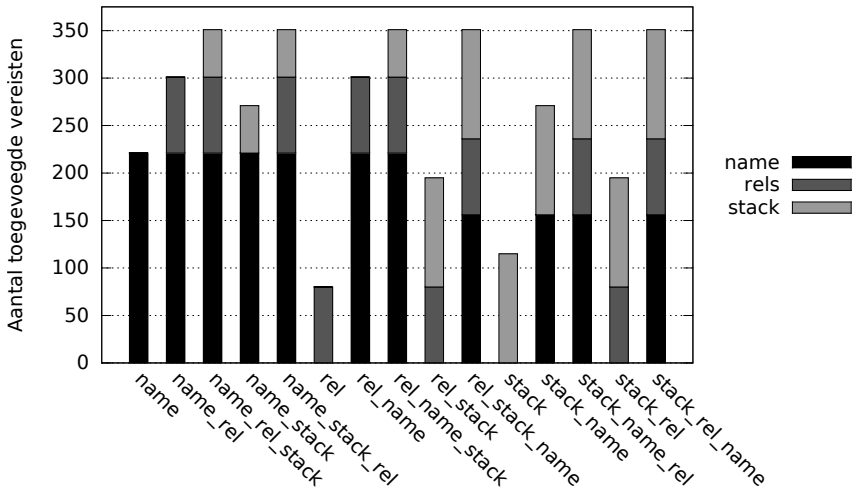
Figuur 4.8 toont hoeveel deployment runs nodig zijn om een volledig werkende MongoDB database te bekomen, en hoeveel resources er per run gedeployed worden.

De library voor MongoDB bestaat uit 185 regels code[2], waarvan er 24 relaties beschrijven. Alle libraries samen bevatten 50 relaties.

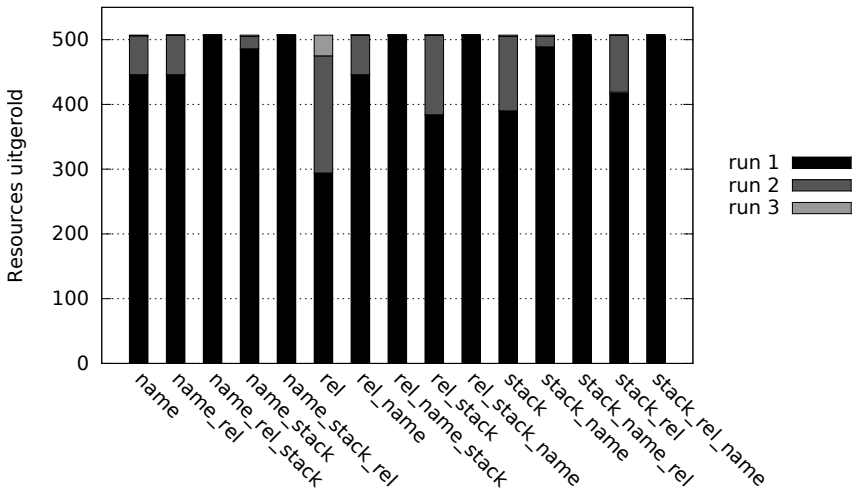
[2]Blanco lijnen en commentaar niet meegerekend



Figuur 4.6: Implementatie van MongoDB in IMP, met de correcte opstartvolgorde aangeduid



Figuur 4.7: Aantal toegevoegde vereisten voor elke combinatie van heuristieken bij MongoDB



Figuur 4.8: Gemiddelde aantal uitgerolde resources per deployment run voor elke combinate van heuristieken bij MongoDB

Hoofdstuk 5

Besluit

5.1 Gerelateerd werk

Cloud Resource Orchestration: A Data-Centric Approach: andere aanpak om te vermijden dat inconsistente configuraties bereikt worden. Datacentrische aanpak (declaratief) itt tot andere tools die heel imperatief werken (bvb ansible)

verwijzen naar overzichtspaper

IBM

5.2 Verder werk

Een eerste onderzoeksrichting betreft de heuristieken. Niet alleen worden deze beter nog in andere use cases gebruikt om hun validiteit verder te bevestigen, nieuwe use cases kunnen ook leiden tot nieuwe heuristieken.

De heuristieken kunnen mogelijks ook gebruikt worden als onderdeel van een intelligente autocompletetool. Een dergelijke tool stelt de gebruiker voor om vereisten tussen resources binnen een stack toe te voegen, of suggereert om bepaalde relaties als afhankelijk te specificeren.

Verder onderzoek kan ook pogen de minimale set vereisten te definiëren die nodig is om een model correct uit te rollen. De heuristieken die voortkomen uit dit onderzoek voegen liever een extra vereiste toe dan geen. Deze aanpak is goed zolang er geen incorrecte of overbodige vereisten worden toegevoegd, maar elke vereiste moet verwerkt worden door IMP. Dit zorgt voor een kleine verhoging in de uitvoertijd. Een minimale set vereisten kan dus een positief effect hebben op de totale uitroltijd.

Een derde richting die verder onderzocht kan worden is hoe IMP (of CMS in het algemeen) kan omgaan met verschillende tussen opeenvolgende versies van een configuratiemodel. 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. Mogelijks leidt dit zelfs tot andere resources die overbodig worden, zoals een database die enkel gebruikt door een webserver. Als de webserver uitgeschakeld wordt kan het wenselijk zijn om automatisch ook de databaseserver stop te zetten. Dit soort acties

zijn mogelijk in IMP omdat in het model de relaties staan die aanwezig zijn in de opstelling.

Bijlagen

Bijlage A

Code van de heuristieken

A.1 Bestanden en mappen

```
def dir_before_file(model, resources):
  for _id, resource in resources.items():
    model = resource.model
    res_class = model.__class__
    if model.__module__ == "std" and (res_class.__name__ == "File" or
    res_class.__name__ == "Directory"):
      host = model.host

      for dir in host.directories:
        dir_res = Resource.get_resource(dir)
        if dir_res is not None and os.path.dirname(resource.path) == dir_res.
    path:
          if dir_res.id not in resource.requires:
            resource.requires.add(dir_res.id)
```

A.2 Services, packages en configuratiebestanden

A.2.1 Methode 1

```
TODO: herschrijven
```

A.2.2 Methode 2

```
1  def scope_dependencies(model, resources):
2    srv_stacks = find_srv_stacks(resources)
3    deps_per_stack = []
4    for stack in srv_stacks:
5      deps_per_stack.append(setup_stack_deps(stack))
6
7  def find_srv_stacks(resources):
8    stacks = []
9    for res in resources.values():
10     scope_pkg = []
11     scope_cfg = []
12     scope_srv = []
13     model_instance = res.model
14     instance_scope = model_instance.__scope__
15     scope_vars = instance_scope.variables()
16     scope_cfg = [x.value for x in scope_vars if x.value.__class__.__name__ ==
       "File"]
17     scope_srv = [x.value for x in scope_vars if x.value.__class__.__name__ ==
       "Service"]
18     scope_pkg = [x.value for x in scope_vars if x.value.__class__.__name__ ==
       "Package"]
19
20     if scope_srv and (scope_pkg or scope_cfg):
21       stacks.append({'pkg': scope_pkg, 'srv': scope_srv, 'cfg': scope_cfg})
22
23   return stacks
24
25 def setup_stack_deps(stack):
26   deps = 0
27   for srv in stack['srv']:
28     srv = Resource.get_resource(srv)
29     for pkg in stack['pkg']:
30       pkg = Resource.get_resource(pkg)
31       if pkg.id not in srv.requires:
32         srv.requires.add(pkg.id)
33         deps = deps + 1
34     for cfg in stack['cfg']:
35       cfg = Resource.get_resource(cfg)
36       if cfg.id not in srv.requires:
37         srv.requires.add(cfg.id)
38         deps = deps + 1
39   for cfg in stack['cfg']:
40     cfg = Resource.get_resource(cfg)
41     for pkg in stack['pkg']:
42       pkg = Resource.get_resource(pkg)
43       if pkg.id not in cfg.requires:
44         cfg.requires.add(pkg.id)
45         deps = deps + 1
46   return deps
```

A.3 Resources met gelijkaardige namen

```
1  def name_dependencies(model, resources):
2    deps = 0
3    for _id, res in resources.items():
4      similar_resources = []
5      res_model = res.model
6      if res_model.__module__ == "std" and res_model.__class__.__name__ == "
       Service":
7        for other_id, other_res in resources.items():
8          similar_reg = re.compile("std::File.*%s.*%s.*" % (res.model.host.name,
        res.name))
9          if other_res.model.__class__.__name__ == "File" and similar_reg.match(
       str(other_id)):
10           similar_resources.append(other_res)
11
12       for similar_res in similar_resources:
13         if similar_res.id not in res.requires:
14           res.requires.add(similar_res.id)
15
16     if res_model.__module__ == "std" and res_model.__class__.__name__ == "
       Package":
17       pkg_name = ''.join(i for i in res.name if not i.isdigit())
18       pkg_name = pkg_name.split("-")[0]
19       for other_id, other_res in resources.items():
20         similar_reg = re.compile("std::File.*%s.*%s.*" % (res.model.host.name,
        pkg_name))
21         if other_res.model.__class__.__name__ == "File" and similar_reg.match(
       str(other_id)):
22           similar_resources.append(other_res)
23
24       for similar_res in similar_resources:
25         if res.id not in similar_res.requires and similar_res not in res.
       requires:
26           similar_res.requires.add(res.id)
```

A.4 Afhankelijkheden door relaties

```
1  def use_rel_multiplicity(model, resources):
2    added = 0
3    for lib in model.get_scopes():
4      for concept in lib.variables():
5        if concept.get_name().startswith("Entity"):
6          for attr_name in concept.value.get_all_attribute_names(): #anders niet
        de attr vd parents
7            attr_value = concept.value.get_attribute(attr_name)
8            if hasattr(attr_value, "end"): #Kijk of het een relatie is
9              attr_end = attr_value.end
10             if attr_end.low >= 1 and attr_end.end.low == 0 and not attr_value.
       depends:
11               added = added + 1
12               attr_value.depends = True
```

A.5 Vereisten door afhankelijkheden

```
1  def use_relations(model, resources):
2    depends = 0
3    added = 0
4    for lib in model.get_scopes():
5      for concept in lib.variables():
6        if concept.get_name().startswith("Entity"):
7          concept_services = get_services(concept)
8          if concept_services is not None: #Als er geen services zijn moeten er
     sowieso geen deps opgesteld worden
9            for attr_name in concept.value.get_all_attribute_names(): #anders
     niet de attr vd parents
10             attr_value = concept.value.get_attribute(attr_name)
11             if hasattr(attr_value, "end"): #Kijk of het een relatie is
12               if attr_value.end.depends: #Kijk of het concept afhankelijk is
     vd andere kant
13                 depends = depends + 1
14                 for instance in concept.value: #Voor elke instantie v/h
     concept de deps opstellen
15                   req_concepts = getattr(instance, attr_name)
16                   req_resources = []
17                   if hasattr(req_concepts, 'append'): #Check if QList
18                     for req_concept in req_concepts:
19                       for srv in get_child_services(req_concept):
20                         req_resources.append(srv)
21                   else:
22                     for srv in get_services(req_concepts):
23                         req_resources.append(srv)
24                   for req_res in req_resources:
25                     for service in concept_services:
26                       if req_res not in service.requires and req_res.state !=
     "stopped":
27                         service.requires.add(req_res)
28
29 def get_child_services(object):
30   services = []
31   for child_res in object._childeren:
32       if child_res.__module__ == "std" and child_res.__class__.__name__ == "
     Service":
33         services.append(Resource.get_resource(child_res))
34   return services
35
36 def get_services(concept):
37   found_services = []
38
39   if hasattr(concept, "value"):
40     for instance in concept.value:
41       if concept.__module__ == "std" and concept.__class__.__name__ == "
     Service":
42         found_services.append(Resource.get_resource(concept))
43
44       if hasattr(instance, "_childeren"):
45         found_services.extend(get_child_services(instance))
46
47   if hasattr(concept, "_childeren"):
48         found_services.extend(get_child_services(concept))
49
50   return set(found_services)
```

Bibliografie

[1] Tudor Dumitras and Priya Narasimhan. Why do upgrades fail and what can we do about it?: toward dependable, online upgrades in enterprise system. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, page 18. Springer-Verlag New York, Inc.

[2] Jordan Sissel. Sysadvent: Day 19 - why use configuration management? http://sysadvent.blogspot.be/2011/12/day-19-why-use-configuration-management.html.

[3] Damien Cerbelaud, Shishir Garg, and Jeremy Huylebroeck. Opening the clouds: qualitative overview of the state-of-the-art open source (vm-based) cloud management platforms. In *Proceedings of the 10th (ACM/IFIP/USENIX) International Conference on Middleware*, page 22. Springer-Verlag New York, Inc.

[4] Johannes Kirschnick, Jose M. Alcaraz Calero, Patrick Goldsack, Andrew Farrell, Julio Guijarro, Steve Loughran, Nigel Edwards, and Lawrence Wilcock. Towards an architecture for deploying elastic services in the cloud. 42(4):395–408.

[5] Ansible is simple IT automation. http://www.ansible.com/home.

[6] Puppet labs: IT automation software for system administrators. http://puppetlabs.com/.

[7] Configuration management software | open source configuration management - CFEngine - distributed configuration management. http://cfengine.com/.

[8] Feature #16187: Relationships should work between hosts - puppet - puppet labs. http://projects.puppetlabs.com/issues/16187.

[9] Bart Vanbrabant and Wouter Joosen. A framework for integrated configuration management tools. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, page 534540. IEEE.

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, Dr. Dimitri Van Landuyt