KU LEUVEN

FACULTEIT INDUSTRIELE
INGENIEURSWETENSCHAPPEN

TECHNOLOGIECAMPUS GENT

Ontwerp en ontwikkeling van een testraamwerk installer

Pieter-Jan ROBRECHT

Promotor: Annemie Vorstermans

Co-promotor: Carl Eeckhout

Masterproef ingediend tot het behalen van de graad van master of Science in de industriële wetenschappen: master of Science in de industriële wetenschappen ICT Advanced Communicatie Technologies

Academiejaar 2016 - 2017

Dankwoord

Dank aan een ander

Abstract

Televic Rail ontwikkelde een Python testraamwerk voor het testen van verschillende producten. De ontwikkelde software, die op verschillende platformen moet draaien, gebruikt verschillende drivers en bibliotheken. Om producten correct te kunnen testen, wordt het raamwerk vaak geüpdatet, bijvoorbeeld bij het uitbrengen van een nieuwe driver, bibliotheek of om nieuwe producten te ondersteunen. Het installatieproces is tijdrovend, foutgevoelig en wordt best geautomatiseerd. Door dit proces te automatiseren wordt het mogelijk om informatie over het installatie- en updateproces te verzamelen. Het doel van de thesis is het uitvoeren van onderzoek naar een efficiënte oplossing en het ontwikkelen van een prototype. Dit prototype wordt onderverdeeld in drie componenten: een packager, een deployment server en een deployment omgeving. In een eerste fase wordt de packager ontworpen. Deze staat in voor het samenvoegen van de software componenten. Fase twee van de thesis bestaat uit het uitwerken van de deployment server. Met de server worden de verschillende installers verspreid en wordt er informatie verzameld over de deployment environments. Als laatste wordt dan de deployment environment behandeld. In deze geïsoleerde omgeving kan het installatie- en updateproces veilig gebeuren. Na een grondige evaluatie van een eerste basisprototype wordt het ontwerp eventueel aangepast. Het prototype wordt in een laatste fase uitgebreid zodat een rapportering over geïnstalleerde versies, deployment status, ... beschikbaar wordt voor het bedrijf.

Trefwoorden: Automatische – installer – testraamwerk - Python

Inhoudsopgave

Lijst van figuren

Hoofdstuk 1

Inleiding

1.1 Situering

Met meer dan 30 jaar ervaring in het ontwerpen en onderhouden van on-board communicatie-systemen, Televic Rail is een toonaangevende leverancier voor Passenger Information Systems, Entertainment Systems and Infotainment Systems. Dit internationale bedrijf, met vestingen in zowel Europa als de Verenigde State, combineren hun kennis en ervaring met een constante drang naar innovatie en zijn zo in staat om projectgerichte, cutting-edge oplossingen af te leveren die betrouwbare communicatie in treinen voorzien.

LiveCom is Televic Rail nieuwste generatie van informatie management systemen, de integratie van alle aspecten van de on- en off-board reizigersinformatie, infotainment en entertainment. Het stelt operatoren in staan om hun volledige verkeer schema's, dienstregelingen, routes, stations en alles met betrekking tot informatie en infotainment omtrent passagiers te beheren, met behulp van off-board software tools.

iCoM, de geïntegreerde oplossing van Televic Rail voor passagiersgegevens en communicatie management, biedt het openbaar vervoer en spoorwegondernemingen een centraal systeem voor het creëren, beheren, distribueren en uitvoeren van real-time on en off-board generieke en commerciële passagiersinformatie op de vloot, in stations en bij haltes.

Naast deze systemen heeft Televic verschillende mechatronica sensoren en veiligheid controle-systemen ontworpen. Alle systemen en apparaten zijn ontworpen in overeenstemming met de betreffende spoorwegsector normen en geschikt voor auto-body, draaistel en as montage eisen te voldoen. On-board controllers verwerken sensordata informatie, sturen deze naar de betreffende actuators en treinbeheersingssystemen. Fysische parameters die momenteel worden ondersteund zijn onder andere versnelling, druk, rotatie, temperatuur, akoestische muziek en de verplaatsing.

Om te voldoen aan de strenge veiligheidsnormen heeft Televic Rail een Python test framework ontworpen waarmee zij in staat zijn om verschillende producten te onderwerpen aan verschillende testscenario's. Het framework werd ontworpen om gebruikt te worden op verschillende testtorens en werd later aangepast om bruikbaar te zijn op gewone computers. Dit framework wordt intensief gebruikt tijdens het productieproces en is cruciaal voor het afleveren van producten die voldoen aan de strenge veiligheidsnormen.

1.2 Probleemstelling

Om een goede werking te verkrijgen steunt het Python testraamwerk op een verschillende drivers en bibliotheken. Hiernaast moet het raamwerk correct functioneren met de grote hoeveelheid aan producten die Televic fabriceert. Om ook deze te ondersteunen zijn er wederom verschillende drivers en bibliotheken nodig. Het gevolg hiervan is dat het installatieproces tijdrovend is en foutgevoelig. Bij het uitbrengen van een nieuwe versie van de applicatie, bijvoorbeeld bij het uitbrengen van een nieuwe driver, bibliotheek of om nieuwe producten te ondersteunen, moet de applicatie geüpdatet worden. Dit proces lijdt aan dezelfde gebreken als het installatieproces. Het installatie- en updateproces vraagt om een vereenvoudiging zodanig dat het testraamwerk gebruiksvriendelijker wordt.

Naast het installatie- en updateproces vormt ook de groeiende hoeveelheid gebruikers een probleem. Naarmate het aantal gebruikers stijgt, stijgt ook de vraag naar een algemene administratie interface. Hiermee wordt het mogelijk om verschillende gebruikers bij te staan maar ook om bij te houden hoe het uitrollen van een nieuwe versie van het testraamwerk verloopt. Deze informatie kan gebruikt worden om het verspreidingsproces bij te sturen zodanig dat een volgende keer het proces vlotter verloopt.

1.3 Overzicht

Het doel van deze thesis is dan ook een oplossing te vinden voor het bovengenoemde probleem.

Hoofdstuk 2

Literatuurstudie

In wat volgt, wordt een bespreking gegeven over het deployment proces maar ook van de verschillende problemen die horen bij dit proces. Hiernaast zullen enkele case studies besproken worden en gaat er onderzocht worden hoe de verscheidene cases zijn omgegaan met de problemen die horen bij het deployment proces.

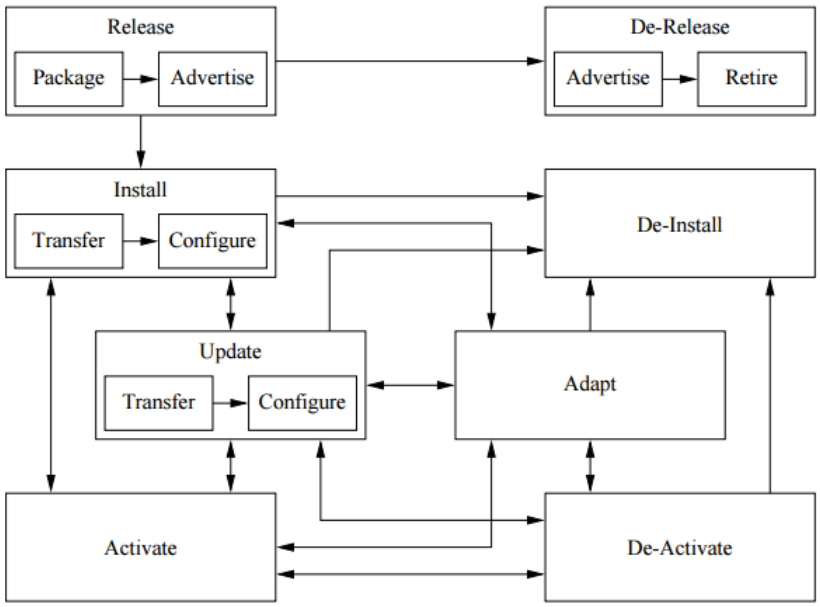
2.1 Software levenscyclus

De levenscyclus van software deployment kan volgens Hall, Heimbigner en Wolf [1] en Dearie [2] beschreven worden in verschillend stappen, namelijk:

- *Release*: de software is volledig samengesteld uit pakketten die voldoende metadata bevatten om de verschillende bronnen te beschrijven waarvan het pakket afhangt.
- *Installatie*: de software moet overgebracht worden naar de client en geconfigureerd worden in voorbereiding op de activatie.
- *Activeren*: tijdens de activatie wordt de software uitvoering opgestart of worden de nodige triggers geplaatst om de executie op het gepaste tijdstip op te starten.
- *Deactiveren*: dit is tegengesteld aan activeren. Deze stap is voor nodig voordat een aanpassing of herconfiguratie kan uitgevoerd worden.
- *Updaten*: dit is het proces waarin de software wordt aangepast. Deze stap wordt vaak geactiveerd door het uitbrengen van een nieuwe versie van de software.
- *Deïnstallatie*: tijdens deze stap zal de geïnstalleerd software van het client systeem gehaald worden.

De enige fase van de levenscyclus die zich uitsluitend op de server afgespeeld is de release fase. De rest van de fases spelen zich af op de verschillende client systemen.

In theorie zou het deployen van software een eenvoudige klus moeten zijn. Aangezien software bestaat uit een set van bestanden, zou het deployen van software naar een doelcomputer slechts bestaan uit het kopiëren van de nodige bestanden. Maar dit is vaak niet het geval. Volgens Dolstra [4] zijn er in de praktijk verschillende oorzaken die aan de basis liggen van een ingewikkeld deployment proces. Deze oorzaken kunnen in twee grote categorieën ingedeeld worden, namelijk de omgevings- en onderhoudsproblemen.



Figuur 2.1: Levenscyclus van software [3]

Omgevingsproblemen In de eerste categorie ligt de nadruk vooral op correctheid. Voordat de software geïnstalleerd wordt op een doelsysteem, wordt de doelomgeving ondervraagt naar alle eigenschappen: zijn de nodige programma's aanwezig, bestaan alle configuratie bestanden, ... Als deze eisen niet voldaan zijn, dan zal de software niet werken zoals gewenst. Dolstra [4] haalt enkele concrete voorbeelden aan van omgevingsproblemen:

- De deployment van software kan een gedistribueerd probleem zijn. Software kan afhankelijk zijn van componenten draaiende op verwijderde systemen of van andere processen draaiende op het doelsysteem.
- Software is vaak afhankelijk van verschillende andere software componenten. Deze afhankelijkheden, of ook wel dependencies genoemd, moeten voor de deployment bepaald worden. Dit proces is moeilijk en een fout kan pas laat ontdekt worden.
- De dependencies moeten compatibel zijn met wat er verwacht wordt van de software. Niet elke versie zal werken. Sommige depencies vertonen build-time variaties. De component kan dan gebouwd zijn met enkele optionele eigenschappen of eigenschappen die gekozen worden at build-time.
- Sommige software componenten zijn afhankelijk van specifieke hardware. Dit kan enkel verholpen worden door op voorhand te controleren welke hardware aanwezig is.

Uit deze concrete voorbeelden wordt al snel duidelijk dat er twee problemen zijn: de verschillende eisen van de software moeten geïdentificeerd worden en vervolgens moeten deze gerealiseerd worden in het doelsysteem.

Onderhoudsproblemen Naast de verschillende omgevingsproblemen, beschrijft Dolstra [4] ook enkele onderhoudsproblemen. Deze hebben te maken met het feit dat software moet kunnen

“evolueren”. Om dit te ondersteunen, moeten allerlei actie zoals upgraden en updaten uitgevoerd wordens. Enkele voorbeelden van zulke acties zijn:

- Tijdens het verwijderen van software, moeten alle componenten verwijderd worden. Ondertussen mogen geen componenten verwijderd worden die nog in gebruik zijn door andere software.
- Ook tijdens het updaten van software moet rekening gehouden worden met andere software. Het updaten van een component kan voor problemen en failure zorgen in een andere component. Een DLL-hell wordt best ten alle tijden vermeden.
- Na het upgraden/updaten van een component, is het soms aangewezen om een roll back uit te voeren. Zo’n actie kan overwogen worden als de upgrade belangrijke functionaliteiten van de software kapot maakt.

2.2 Deployment strategieën

De eerste stap in de softwarelevenscyclus bestaat uit het uitbrengen van de software. De software moet bij de gebruiker terecht komen voordat hij/zij deze kan gebruiken.

Tian, Zhao, Gao e.a. [5] haalt drie methodes aan om software te deployen: disk image-based deployment, behavior-based deployment en package-based deployments. Bij disk image-based deployment worden de software en het besturingssysteem op eenzelfde moment naar de target node verzonden. Er zullen verschillende image servers aanwezig zijn die elk een service aanbieden. Het voordeel van deze strategie is dat, zolang de hardware en software vereisten voldaan zijn, de deployment bestaat uit een simpele read-write operatie. Maar, zoals Tian, Zhao, Gao e.a. [5] al aangeeft, deze methode is niet flexibel. Voor de applicatie is flexibiliteit een must. Iedere node bevat verschillende hardware en is verschillende geconfigureerd. Een disk image-based deployment zal hierdoor niet gebruikt worden. Het basis idee achter behavior-based deployment is het opnemen van de schijf operaties tijdens het deployen. Als geweten is welke bestanden aangepast zijn, gecreëerd zijn, ... dan kan het proces nagebootst worden op andere nodes [5]. Zo een proces nabootsen is moeilijk. Kernel operaties moeten getraceerd worden. Deze methode biedt een verhoogde flexibiliteit aan ten opzichte van de disk image-based deployment maar dit is nog niet voldoende om deployments uit te voeren die uniek zijn per node. De laatste techniek die Tian, Zhao, Gao e.a. [5] aan haalt is package-based deployment. Met behulp van een batch file, waarin alle nodige commando’s aanwezig zijn, kan een installatie pakket gedeployed worden naar een target node. Door het gebruik van een batch file wordt de flexibiliteit van de deployment verhoogd.

De geproduceerde software moet bij verschillende gebruikers terecht komen. Een eenvoudige oplossing zou zijn dat slechts één gebruiker per keer geholpen wordt en dat iedere gebruiker zijn beurt afwacht. Zo’n oplossing is misschien doenbaar mochten slechts een handvol gebruikers de applicatie nodig hebben maar dit is vaak niet het geval. Patterson [6] haalt verschillende argumenten aan voor het distribueren van deze service en haalt enkele punten aan (betrouwbaarheid, bandbreedte en lage wachttijden) waar rekening mee moet gehouden worden alvorens een ontwerpbeslissing genomen wordt. Zoals Patterson [6] aanhaalt, is het belangrijk dat alle gebruikers ten alle tijden de service kunnen gebruiken. Hiernaast moet er rekening gehouden worden met de deployment strategie. Münch, Armbrust, Kowalczyk e.a. [7] spreekt over verschillende strategieën om software uit te brengen:

- *Big-bang*: iedere gebruiker van de applicatie zal op eenzelfde moment overschakelen van de oude naar de nieuwe software. Hierdoor wordt vermeden dat verschillende afdelingen met een andere versie van de software werken. Een nadeel is dat voldoende support aanwezig moet zijn om mogelijke problemen op te lossen.
- *Gefaseerd*: de nieuwe software zal bij een gefaseerde deployment enkel toegepast worden in specifiek geselecteerde projecten. Als deze strategie voor een verlengde periode wordt toegepast, zullen verschillende versies van de software continu aanwezig zijn onder de gebruikers.

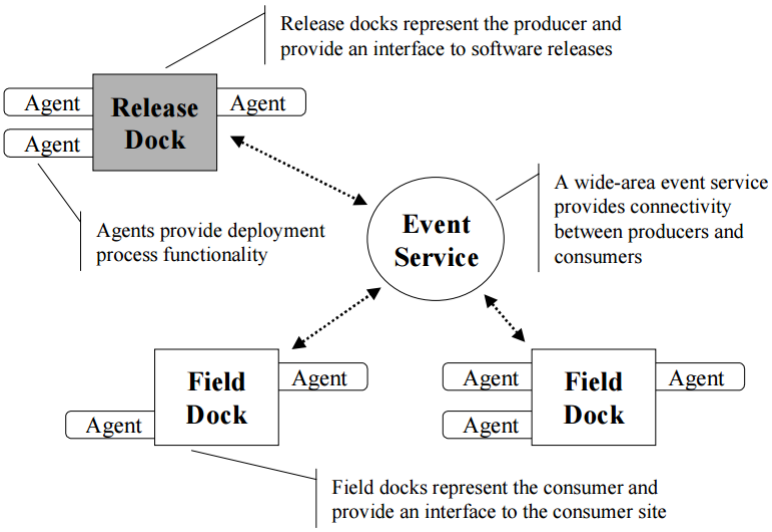
Aan beide strategieën zijn zowel voor- en nadelen gekoppeld. Zo zal de big-bang strategie niet voordelig zijn om uit te voeren als de gebruikers verspreid zitten over de wereld. Door het tijdsverschil zal de deployment bij sommige gebruikers plaatsvinden tijdens de werkuren en bij anderen midden in de nacht. Het voordeel van de big-bang strategie is dat alle doelsystemen op een korte periode omgeschakeld worden naar de nieuwe versie van de applicatie. Het gebruik van de gefaseerde strategie kan het probleem met de tijdszones omzeilen maar deze strategie is ook niet ideaal aangezien het omschakelen van de software naar de nieuwe versie lang kan duren. Een hybride oplossing is hier dus aangewezen.

2.3 Software Dock Architectuur

Hall, Heimbigner en Wolf [1] bespreekt een interessante architectuur. Het Software Dock research project creëerde een framework om de samenwerking tussen software producenten en verbruikers te verbeteren. In Figuur 2.2 op de volgende pagina wordt de ontworpen architectuur voorgesteld. Er worden twee verschillende componenten gedefinieerd waarmee de producenten en verbruikers voorgesteld worden. In de architectuur worden de verschillende producenten voorgesteld aan de hand van een release dock en worden de verbruikers voorgesteld als een field dock. Aan deze docks worden verschillende agents gekoppeld. Elke agent hoort typisch bij één stap uit de software levenscyclus die besproken wordt in Sectie 2.1 op pagina 3. Naast de verschillende docks wordt ook een wide-area event systeem gedefinieerd. Met dit systeem wordt de communicatie tussen de docks aangeboden. Hall, Heimbigner, Van Der Hoek e.a. [8] legt alle stappen van de Software Dock architectuur uit.

De release dock is een server die zich bevindt bij de software producent. Dit dock biedt een release repository aan waar de verbruikers de nodige software selecteren voor deployment. In de release dock wordt ieder release semantisch beschreven aan de hand van een Deployable Software Description file. Elke release wordt vergezeld door enkele agents die de semantische betekenis lezen en zo de deployment kunnen uitvoeren. Aan de hand van interfaces kunnen de agents aan de services en inhoud van de release dock. Bij het wijzigen van een software release zal de release dock verschillende events afvuren. Agents kunnen zich subscriben bij deze events en weten zo wanneer bepaalde handelingen uitgevoerd moeten worden [1].

De field dock dient als een interface naar de verbruiker kant toe. Deze interface biedt informatie over de resources en configuratie van het verbruiker systeem. Op basis van deze informatie wordt een context opgebouwd waarin de releases van de resource dock worden gedeployed. De agents die horen bij een release, docken zichzelf in de field dock en kunnen aan de hand van deze interface het verbruikers systeem ondervragen. Aangezien kritische client-side informatie op een gestandaardiseerde wijze aangeboden wordt, met behulp van een geneste collectie van pair-values die een hiërarchie vormen, kan de installatie van de software gepersonaliseerd worden [1].



Figuur 2.2: Software Dock Architectuur [1]

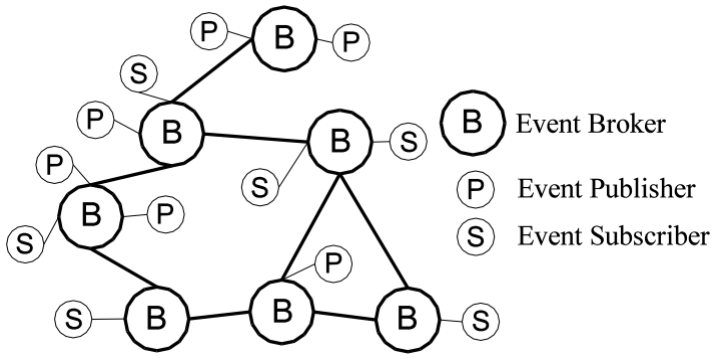
2.4 Event based services

Een belangrijk onderdeel van de software dock architectuur is de event service. Deze handelt de communicatie af tussen de verschillende docks en is een spilfiguur in de architectuur. Aan de hand van Pietzuch en Bacon [9] en Carzaniga, Rosenblum en Wolf [10] is het mogelijk om een bespreking te geven over hoe de event service geïmplementeerd kan worden.

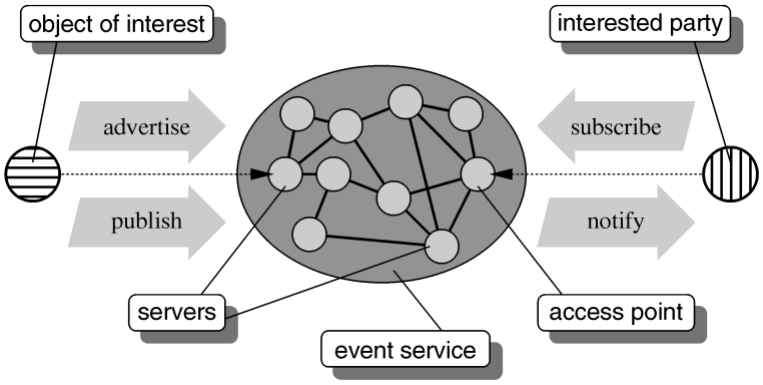
Meeste middleware systemen zijn invocation-based systemen waarbij het request/response paradigma gebruikt wordt om de communicatie tussen client en server te onderhouden. Een client verzoekt een service van de server, waar de server vervolgens op antwoord. Het toepassen van deze strategie is doenbaar in een omgeving met een beperkt aantal clients en servers. Om een grootschalig netwerk op te bouwen en om te kunnen gaan met een dynamische omgeving moet een andere manier van communiceren gebruikt worden. Publish/subscribe systemen bieden een oplossing voor deze problemen. Clients, event subscribers, tonen hun interesse voor een bepaald onderwerp en servers, event publishers, produceren een event die naar alle geïnteresseerden wordt doorgestuurd. Een algemeen voorbeeld is terug te vinden in Figuur 2.3 op de volgende pagina. Door deze manier van communiceren te hanteren, ontstaat een natuurlijke ondersteuning voor many-to-many relaties tussen de clients en servers. De twee worden ook van elkaar ontkoppeld. Voor de client maakt het niet uit welke publisher de producent is van een event. Hiernaast moet de server niet weten wie allemaal de events ontvangt die hij produceert [9].

Een naïeve aanpak, volgens Carzaniga, Rosenblum en Wolf [10], is het gebruiken van één centrale server waar alle subscripties worden bijgehouden, waar alle events toekomen, waar de bestemming van het event beslist wordt en waar het event wordt doorgestuurd naar de gepaste subscribers. Deze strategie is eenvoudig te implementeren maar deze werkt de schaalbaarheid tegen. Dit was ook al duidelijk in Figuur 2.3 op de pagina hierna waar verschillende “Brokers” aanwezig zijn. Het is belangrijk om stil te staan bij enkele ontwerp beslissingen zodanig dat de service die in Figuur 2.4 op de volgende pagina zichtbaar is, implementeerbaar is.

Naast de architectuur is het belangrijk om te weten wat wordt verzonden en op welke manier. Carzaniga, Rosenblum en Wolf [10] haalt een structuur aan waarin een event beschreven wordt als een set van attributen. Ieder attribuut bestaat uit een type, naam en waarde. De naam van een



Figuur 2.3: Publish/Subscribe voorbeeld [9]



Figuur 2.4: Publish/Subscribe service [10]

attribuut is een string en het type komt uit een set van primitieven die terug gevonden worden bij de meeste hedendaagse programmeertalen. Een voorbeeld van zo'n event is terug te vinden in Figuur 2.5

string	*class = finance/exchanges/stock*
time	*date = Mar 4 11:43:37 MST 1998*
string	*exchange = NYSE*
string	*symbol = DIS*
float	*prior = 105.25*
float	*change = -4*
float	*earn = 2.04*

Figuur 2.5: Publish/Subscribe Event [10]

2.5 Case studies

Door de jaren heen zijn er verschillende technologieën ontwikkeld die het probleem van software deployment aanpakken. In wat volgt, worden enkele van deze technologieën besproken gebaseerd op enkele case studies die Dearie [2] aanreikt.

2.5.1 Java Beans

Enterprise JavaBeans (EJB) zijn een standaard voor het bouwen van server-side componenten. De EJB's zijn speciaal ontworpen voor het vereenvoudigen van de deployment. Dearie [2] beschrijft JavaBeans als eenheden van business logic in een component die uitgevoerd wordt in een container. De verschillende containers zorgen voor een abstractie van de hosting omgeving en bieden verscheidene services aan. Een JavaBean moet ingepakt worden volgens de specificaties die Sun Microsystems oplegt. Met deze standaard is het mogelijk om verschillende management en deployment tools te schrijven die de EJB's kunnen beïnvloeden. Enterprise JavaBeans worden ingepakt in de standaard Java JAR file, samen met een XML deployement descriptor. De descriptor beschrijft de verschillende eigenschappen van de bijhorende EJB.

De Enterprise JavaBeans zijn volgens Dearie [2] fijnkorrelige en taalafhankelijke oplossing voor het deployment probleem. Door het isoleren van de Beans door middel van een gestandaardiseerde container interface zullen Enterprise JavaBeans zo een oplossing vinden voor het deployment probleem. Hierdoor moeten de EJB's zodanig ontworpen worden dat ze voldoen aan de eisen van de interface. Enterprise JavaBeans hebben geen idee van het op afstand installeren van componenten. Een groot probleem bij EJB's is dat referenties naar afhankelijke beans gebeurd via niet unieke namen. Twee verschillende beans met eenzelfde naam moeten hierdoor manueel herladen worden zodanig dat de bindings up-to-date zijn [11].

2.5.2 Redhat package manager

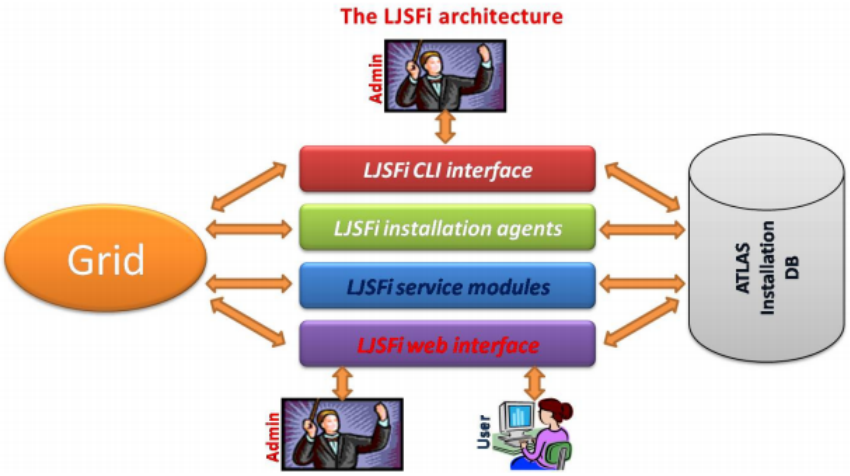
In Linux wordt de Redhat package manager (RPM) het vaakst gebruikt voor de deployment van software. Met hulp van de RPM is het mogelijk om enkele operaties uit te voeren zoals onder andere installatie, updaten, ... De operaties worden ondersteund door een databank die alle informatie en details van de geïnstalleerde pakketten bevat. Een RPM pakket bestaat uit executables gecombineerd met configuratie bestanden en documentatie. Doordat een pakket executables bevat, zal een pakket gekoppeld zijn aan het besturingssysteem van de host [12]. Naast de RPM files bevat een pakket verschillende scripts geschreven in de standaard Unix scripting taal. De verscheidene scripts zijn ingedeeld in sets horende bij een specifieke taak. Bij een error moet een roll back uitgevoerd worden. Dit is de taak van de script schrijver [2].

De Redhat Package Manager is in tegenstelling met de EJB's een grofkorrelige, taal onafhankelijke maar besturingssysteem afhankelijke oplossing. Het grootste probleem van RPM schuilt in de afhankelijkheden tussen de pakketten. Niet alle afhankelijkheden zijn explicite gemodelleerd en de afhankelijkheden die wel gemodelleerd zijn, zijn gevormd met de nadruk op de inhoud en niet op de pakketten zelf [2].

2.5.3 ATLAS

Om in de ATLAS samenwerking om te gaan met de grote hoeveelheid bronnen, is er een volledig automatisch installatie systeem ontworpen voor het LCG/EGEE project, LHC Computing Grid/Enabling Grids for E-sciencE [13]. Salvo, Barchiesi, Gnanvo e.a. [14] beschrijft de architectuur van het ontworpen systeem. Het ontwerp van het installatie systeem werd gebaseerd op het Light Job Submission Framework for installation, ook wel LJSFi.

De architectuur van het framework is zichtbaar in Figuur 2.6 op de volgende pagina. Het framework vormt een dunne laag over de middleware van Grid. De kern van het systeem bestaat uit de



Figuur 2.6: LJSFi Architectuur [14]

installatie database en de command line interface (CLI). De laatste zorgt voor de interacties met de Grid middleware. Met hulp van de installatie database kan de CLI verschillende taken en job informatie opslaan. Aan de hand van deze informatie kunnen installaties uitgevoerd worden. De installatie databank staat in contact met alle componenten van het framework. Zo kan de status van verscheidene acties en configuraties van verschillende taken opgeslagen worden.

Naast deze twee grote componenten bevat LJSFi modules en extensies waarmee installatie aanvragen afgehandeld worden. Het systeem bevat drie verschillende componenten:

- **RAI module** De Request An Installation module dient als web interface voor het ontvangen van user-driven installatie aanvragen.
- **AIR module** De Automatic Installation Requester schiet in actie als software release aangeduid staan als productie of auto-installatie. De module verwijdert of installeert de software op alle sites waar de software nog niet gepubliceerd is. Door de AIR module periodiek te gebruiken, zullen de nodige aanvragen snel afgehandeld worden.
- **InAgent module** Met de InAgent module wordt het mogelijk om volledig geautomatiseerde installatieprocessen te voorzien. Iedere 10 minuten wordt de Installation database gelezen en via de CLI interface worden de nodige installatieprocessen opgestart. Elk installatieproces wordt bijgestaan door een installation agent. De agent zal instaan voor het updaten van de Installatie database met real-time informatie die zichtbaar is online.

Naast de verscheidene automatische services biedt LJSFi enkele gebruiker services aan. Een gebruiker kan zich subscriben voor bepaalde acties op een doel systeem. Als deze actie wordt uitgevoerd dan krijgt de gebruiker een mail. Hiernaast kan een gebruiker een software release vastpinnen zodat deze niet verwijderd kan worden door het systeem.

Het installatieproces wordt uitgevoerd in drie verschillende stappen. In een eerste stap wordt een site check uitgevoerd door een pilot job naar de site te sturen. Als de check succesvol uitgevoerd wordt, kan het installatieproces beginnen. De acties tijdens het installatieproces worden uitgevoerd door softwaremanagement scripts. Op het einde van het proces, haalt het systeem de job output en exit code op. De laatste wordt opgeslagen in de Installation database.

Obreshkov, Albrand, Collot e.a. [15] bespreekt hoe het ATLAS project te werk gaat bij het inpakken van alle nodige software. Het ATLAS project gebruikt CMT als configuratie manager. Met behulp van een configuratie bestand weten verscheidene tools hoe ze een pakket moeten afhandelen. Rybkin [16] spreekt ook over CMT als informatiebron voor het ophalen van meta-data. Aan de hand van deze data kan een Pacman pakket geproduceerd worden. Met behulp van een "Pacman file" is geweten hoe de ingepakte software behandelt moet worden.

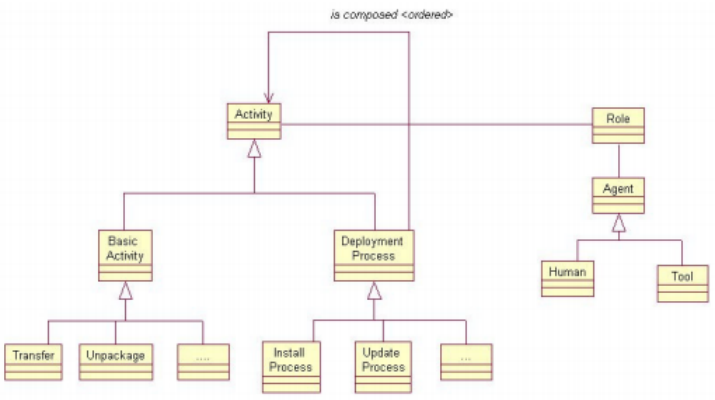
2.5.4 ORYA

Lestideau en Belkhatir [17] legt uit hoe met ORYA (Open enviRonment to deploY Applications) verschillende deployment functionaliteiten aanbiedt aan gedistribueerde, autonome entiteiten zoals workstations en servers. Aan de hand van een deployment PSEE [18] wordt het mogelijk om het installatieproces te automatiseren.

In het ontwerp van ORYA worden er drie verschillende entiteiten besproken die nodig zijn om het automatische installatieproces mogelijk te maken.

- **Applicatie Server** De applicatie server bevat nodige informatie nodig voor de installatie. Hieronder bevindt zich onder andere een pakket met de nodige resources en een manifest waarin de afhankelijkheden, beperkingen en features staan.
- **Target** De target is het doel waarop de deployment uitgevoerd wordt. Iedere target wordt beschreven door de verschillende applicaties die al aanwezig zijn en de fysische beschrijving.
- **Deployment Server** De deployment server vormt de kern van de deployment omgeving en staat in voor het uitvoeren van de deployments. De deployment server zoekt de nodige pakketten, voert een transfer van de pakketten uit en installeert de applicatie. Op het einde moet de deployment server garanderen dat vorige programma's correct blijven functioneren.

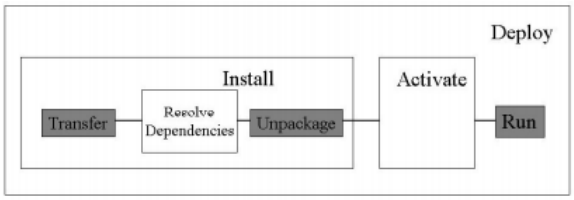
Lestideau en Belkhatir [17] beschrijft verder de verschillende modellen die gehanteerd worden om een succesvolle deployment uit te voeren. In Figuur 2.7 is het deployment proces model terug te vinden. Een deployment proces zal bestaan uit verschillende basis activiteiten en deployment processen. Iedere activiteit wordt uitgevoerd door een agent.



Figuur 2.7: Deployment proces model [17]

Een toegepast voorbeeld van een deployment aan de hand van dit model is terug te vinden in Figuur 2.8 op de volgende pagina. Een basis activiteit wordt voorgesteld aan de hand van een

grijze rechthoek en een deployment proces aan de hand van een witte rechthoek. In het voorbeeld zijn dus vier deployment processen aanwezig en 3 basis activiteiten.



Figuur 2.8: Voorbeeld van een deployment [17]

Aan de hand van deze structuur wordt het mogelijk om het volledige deployment proces voor te stellen.

2.5.5 Nix

Dolstra [4] bespreekt hoe Nix systemen omgaan met het deployment probleem. Nix houdt verschillende componenten bij in de component store waarbij iedere component een set van bestanden is. De componenten worden van elkaar gescheiden door een unieke naam. Dit wordt bekomen door een cryptografische hash op te nemen in de naam. Nix biedt geen software deployment aan. Het biedt verscheidene mechanismen aan waarmee verschillende deployment beleid beschikbaar worden. Met de volgende deployment models is het mogelijk om in Nix de Nix expressies (de bouwstenen van de Nix componenten) te verspreiden:

- **Handmatige download** Een gebruiker kan zelf pakketten downloaden in de vorm van tar archieven, deze zelf uitpakken en vervolgens installeren. Deze strategie is arbeidsintensief en maakt het moeilijk om alles up-to-date te houden.
- **Updaten aan de hand van een versie management systeem** Een andere strategie is het gebruik van een versie management systeem. Hiermee is het up-to-date houden van de pakketten zeer eenvoudig.
- **Kanalen** Een verdere uitbreiding zijn de kanalen. Een kanaal is een URL naar een tar archief die de nodige Nix expressies bevat. Met deze strategie is het even eenvoudig om pakketten te installeren en up-to-date te houden.
- **One-click installatie** Als er enkel één pakket geïnstalleerd moet worden, dan is de one-click installatie de eenvoudigste optie. Via de website van de verdeler kan een link gebruikt worden om het nodige pakket te installeren.

2.5.6 Ansible

Als laatste case studie wordt Ansible besproken. Ansible is een open source IT configuratie management, deployment en organisatie tool. de architectuur van Ansible bestaat uit een agentless push model. Dit wil zeggen dat er geen additionele software nodig is op de client toestellen. Dit wordt behaald door gebruik te maken van the remote manegement frameworks aanwezig op de toestellen, SSH voor Linux en UNIX en WinRM voor Windows. Door geen agents te gebruiken, zal Ansible geen resources gebruiken zolang de Ansible het systeem niet aan het gebruiken is [19].

Whitepaper: Ansible in depth [19] beschrijft verder dat Ansible gebruik maakt van *Playbooks* voor de organisatie van de IT omgevingen. Playbooks zijn YAML[1] definities van taken die beschrijven hoe een taak moet geautomatiseerd worden. Een Playbook bestaat uit een aantal “plays” die uitgevoerd kunnen worden op een set van hosts, ook wel een “inventory” genoemd. Iedere play bestaat uit een set van taken die op een subset van de inventory uitgevoerd kan worden. Een task zelf voert een Ansible module uit[2].

Hiernaast is het mogelijk om de mogelijkheden van Ansible uit te breiden. Modules kunnen zelf geschreven worden in eender welke taal met als enige beperking dat een JSON bestand als input moet gegeven worden en dat een JSON bestand gegenereerd moet worden. De inventory van een Playbook kan dynamisch ontdekt worden at runtime.

2.6 Rollback

Het deployen van software op een doelsysteem brengt verscheidene problemen met zich mee. In de levenscyclus van software zijn er verschillende staten waarin de software moet aangepast worden. Zoals reeds besproken in Sectie 2.1 op pagina 3, zijn er verschillende problemen die kunnen optreden tijdens één van deze staten. In de volgende secties gaan verschillende mechanismen besproken worden waarmee de verscheidene problemen aangepakt kunnen worden.

2.6.1 Rollback strategieën

Srinivasan, Kandula, Andrews e.a. [20] spreekt over drie manieren waarop rollback strategieën geïmplementeerd worden in hedendaagse systemen. Checkpointing, main-memory transactions en software rejuvination zijn strategieën die vaak gebruikt zijn.

Checkpointing Checkpointing is een eenvoudige recovery strategie. De staat van een programma wordt periodiek opgeslagen in een bestand op een extern opslagmedium. Deze kan, na het falen van het programma, gebruikt worden om te herstellen van fouten [21]. Bij een checkpointing systeem wordt de volledige staat van het programma opgeslagen. Dit zorgt voor een verhoogde overhead waardoor het niet mogelijk is om frequent een checkpoint uit te voeren [20].

Door gebruik te maken van incrementele checkpoints is het evenwel mogelijk om dit probleem aan te pakken. Enkel de veranderingen van de laatste checkpoint worden opgeslagen in een nieuwe checkpoint. Het deel dat niet is aangepast, kan hersteld worden aan de hand van de vorige checkpoint. Dankzij deze strategie is het mogelijk om de hoeveelheid data die moet worden opgeslagen te verkleinen. Hierdoor zullen wel verschillende recovery bestanden nodig zijn. Het is aangewezen om op regelmatige tijdstippen de verschillende recovery bestanden samen te voegen tot één bestand [21], [22].

Main-memory transactions Een andere rollback strategie zijn de main-memory transactions. Systemen die deze transacties ondersteunen bezitten vaak de mogelijk om terug te keren naar een vorig executie punt. Om deze strategie te kunnen implementeren, moeten applicaties gebruik maken van het transactie programmeermodel. Waardoor de keuzevrijheden van de programmeur beperkt worden [20].

[1] Een human-readable data serialisatie taal (Een superset van JSON)

[2] Een klein stuk code met een specifieke taak

Software rejuvenation Huang, Kintala, Kolettis e.a. [23] definieert software rejuvination als volgt:

> "Software rejuvenation is the concept of gracefully terminating an application and immediately restarting it at a clean internal state."

Tijdens het langdurig uitvoeren van een programma treedt *process aging* op. Door geheugenlekken, niet vrijgegeven bestandlocks, data corruptie, ... zal de performantie van het uitvoerende programma aangetast worden waardoor het programma uiteindelijk faalt. Door het heropstarten van de applicatie worden eventuele fouten uit het systeem gehaald en wordt de software verjongd. Meeste studies over software rejuvenation focussen vooral op het herstarten van de volledige applicatie en werken dus niet op een fijnkorrelige schaal [20].

2.6.2 Virtualisatie

Dearie [2] brengt virtualisatie ter sprake. Door middel van virtualisatie wordt de complexiteit die ontstaat door de interactie tussen het programma, de installatieomgeving en de uitvoeringsbeperkingen gelimiteerd. Door het creëren van een perfecte omgeving komen deze problemen niet voor. Er zijn verschillende voordelen gekoppeld aan het gebruiken van een virtuele machine. Bijvoorbeeld, besturingssystemen op verschillende hardware platformen eisen verschillende drivers en deze drivers hebben misschien afhankelijkheden op een bepaalde firmware en BIOS. Een guest OS draaiende op een virtuele machine heeft deze eisen niet [24].

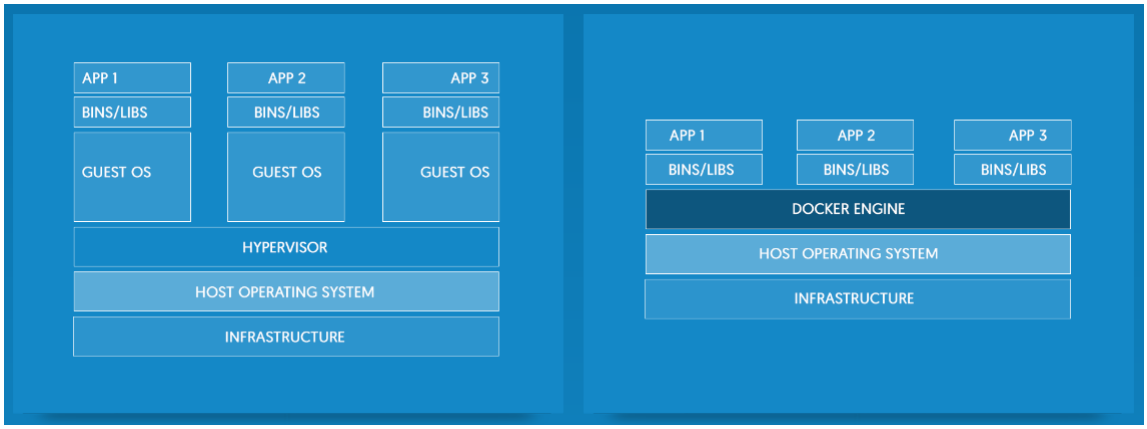
2.6.3 Docker

Virtualisatie is niet de enige techniek die gebruikt kan worden om rollbacks te vermijden. Docker containers is een technologie waarmee een stuk software wordt ingepakt in een volledig filesysteem dat alle benodigdheden bevat om correct te functioneren. Hierdoor zal de software overal op eenzelfde manier draaien, ongeacht de omgeving [25]. Docker containers is niet hetzelfde als virtuele machines. Docker [26] bespreekt de twee aan de hand van een vergelijking. In de vergelijking worden virtuele machines voorgesteld als huizen terwijl Docker containers worden voorgesteld als appartementen.

De huizen staan volledig op zichzelf en bieden bescherming tegen ongewenste gasten. Ze hebben een eigen infrastructuur met hun eigen water, verwarming, ... Hiernaast zal ieder huis op zijn minste een badkamer, living, slaapkamer en keuken hebben. Het vinden van een klein huis is een ganse klus en vaak zal een huis meer bevatten dan nodig is want dat komt door de manier waarop huizen gebouwd worden.

Appartementen bieden ook bescherming tegen ongewenste gasten, maar zij zijn gebouwd rond een gemeenschappelijke infrastructuur. Het appartementsgebouw biedt gemeenschappelijk water, verwarming, ... aan, aan elk appartement. Elk appartement verschilt ook nog van grootte. Er bestaan kleine appartementen maar ook grote met meerder slaapkamers. Men huurt enkel hetgeen dat nodig is.

In Figuur 2.9 op de volgende pagina zijn de architecturen van virtuele machines en Docker terug te vinden. Het verschil tussen beiden wordt al snel duidelijk. Een virtuele machine zal typisch de applicatie, de nodige binaries en bibliotheken en een volledig besturingssysteem bevatten. Een container bevat de applicatie en de verschillende dependencies maar de kernel wordt gedeeld met alle andere containers en gedragen zich als een geïsoleerd proces in de user space van het host besturingssysteem.



Figuur 2.9: Architectuur van Virtuele Machine ten opzichte van Docker [25]

Chamberlain en Schommer [27] bespreekt kort hoe Docker werkt. Docker is een platform dat gebruik maakt van de Linux Containers (LXC de user-space control package voor Linux Containers) om software te encapsuleren. LXC is een virtualisatie techniek waarmee virtuele omgevingen in Linux opgebouwd kunnen worden. De containers zullen processen van elkaar sandboxen zodanig dat een proces een ander niet kan beïnvloeden [28]. Docker zal de LXC software uitbreiden waardoor deployment, distributie en versioning mogelijk wordt. Naast LXC gebruikt Docker AuFS (Advanced Multi-Layered Unification Filesystem) als het filesysteem voor de containers. Doordat het filesysteem gelaagd is, is het mogelijk om verschillende filesystemen over elkaar te leggen.

Merkel [28] vergelijkt de twee virtualisatie technieken en bespreekt de verschillende tussen de twee. Bij virtuele machines moet voor iedere virtuele machine een besturingssysteem geïnstalleerd worden. Al deze besturingssystemen verbruiken RAM, CPU en bandbreedte. Containers zullen piggybacken op het bestaande host besturingssysteem. Hierdoor zal het resource gebruik efficiënter zijn. Een container is goedkoop waardoor het creëren en verwijderen van containers een snelle operatie. Dit komt omdat er enkel een proces moet afgesloten worden in tegenstelling tot het afsluiten van een volledig besturingssysteem. Een voordeel van de VMs ten opzichte van Docker is hun maturiteit. VMs bestaan langer en hebben zichzelf kunnen bewijzen in verschillende situaties.

2.6.4 Technologieën

Implementaties in verschillende programmeertalen, verschillende data representatie formats of incompatibele runtime environments kunnen aan de basis liggen voor een lastige integratie van computerprogramma's. Door gebruik te maken van additionele software, wordt het mogelijk om het gat tussen de verschillen te overbruggen [29]. Het Python testraamwerk is een verzameling van verschillende drivers en bibliotheken elk met een eigen implementatie. In de volgende Sectie wordt er gekeken naar verschillende software oplossingen waarmee dit probleem opgelost kan worden.

Om de verschillende technologieën te vergelijken werd een algemeen testscenario uitgedacht. Het scenario ziet er als volgt uit: er moet een installer gemaakt worden waarmee twee verschillende pakketten (die drivers en bibliotheken moeten voorstellen) geïnstalleerd moeten worden. Hierna werd er onderzocht hoe één van de twee pakketten geüpdatet kon worden. Door de technologieën te onderwerpen aan een test, wordt het mogelijk om de voor- en nadelen van iedere technologie te achterhalen. Hiernaast wordt het ook mogelijk om de technologieën te vergelijken aangezien zij eenzelfde functionaliteit moeten voorzien.

WiX Toolset	
Pro	Cons
Diepe integratie met Windows	XML structuren zorgt voor veel overhead
Mogelijkheid om externe executables te includeren	Niet cross-platform
NSIS	
Pro	Cons
Scripting taal	Niet cross-platform
Verschillende plug-ins beschikbaar	Geen structuur voor packages
Chocolatey	
Pro	Cons
Volledige deployment infrastructuur al aanwezig	Niet cross-platform
	Command-line tool
Qt Installer Framework	
Pro	Cons
Cross-platform	XML structuren zorgt voor veel schrijfwerk
Mogelijkheid om externe executables te includeren	Enkel Linux installer maken in Linux

Tabel 2.1: Voor- en nadelen van de verschillende technologieën

WiX Toolset

Windows installer XML Toolset is een set van build tools waarmee Windows Installer packages gemaakt worden van XML broncode. De toolset is geschreven in C# en heeft het .Net framework nodig om te kunnen functioneren. Bron code wordt gecompileerd en vervolgens gelinkt om een executable te maken. Met de toolset kunnen .msi installatie pakketten, .msm merge modules en .msp patches gecombineerd worden tot een Windows executabel.[30].

Aan deze technologie zijn verschillende voor- en nadelen verbonden (zie Tabel 2.1). Een fragment van de WiX toolset code is terug te vinden in Listing B.1 op pagina 41. De WiX toolset maakt installer uitsluitend bedoelt voor de Windows installation engine. Hierdoor worden verscheidene functionaliteiten eenvoudig te gebruiken, zoals het maken van uitzonderingen in de Windows Firewall. Door gebruik te maken van de Windows installation engine is het niet mogelijk om de executabel te gebruiken in Linux omgevingen[3]. WiX maakt gebruik van XML broncode om verschillende elementen te definiëren. Liefke en Suciu [32] geeft al aan dat XML niet een van de meest efficiënte dataformaten is, maar het verhoogd de flexibiliteit wel. Het creëren van een XML bestand met de hand is een langdradig en moeilijk werk.

NSIS

Nullsoft Scriptable Install System is een open source systeem waarmee Windows installers gemaakt kunnen worden. Zoals de naam aangeeft, is NSIS script-based. Hierdoor bevatten installers de nodige om verschillende installatie taken uit te voeren. Door de grote gebruikersbasis is een grote hoeveelheid plug-ins en scripts beschikbaar. Alle plug-ins en scripts kunnen op een eenvou-

[3] Dit kan eventueel omzeilt worden door het gebruik te maken van software zoals Wine [31]. Als geen alternatieven aanwezig zijn, dan is deze strategie eventueel het overwegen waard.

dige manier aan een installer toegevoegd worden voor een verhoogde functionaliteit [33].

De voor- en nadelen verbonden aan NSIS zijn terug te vinden in Tabel 2.1. Met de scripting taal van NSIS is het mogelijk om eenvoudige installer te definiëren (een voorbeeld hiervan is terug te vinden in Listing B.2 op pagina 42). De scripting taal is intuïtiever te gebruiken in vergelijking met de XML bestanden van Wix Toolset. Dankzij grote hoeveelheid aan plug-ins die aanwezig zijn, is het eenvoudig om een installer te creëren met verschillende functionaliteiten. De gecreëerde installer is een Windows executabel en de opmerking gegeven bij de WiX Toolset is hier ook van toepassing. Het feit dat NSIS bedoelt is om eenvoudige installers te maken zorgt ervoor dat het niet mogelijk is om aparte pakketten te definiëren. Ieder pakket kan wel een eigen configuratie hebben maar dit wordt allemaal toegevoegd aan één script. Bij een grote hoeveelheid aan pakketten leidt tot wanorde en is er geen globaal overzicht. NSIS beidt ook geen mogelijkheden aan om geïnstalleerde software up te daten. Om die eigenschappen toe te voegen moet er beroep gedaan worden op andere software.

Chocolatey

Volgens *Chocolatey About page* [34] is Chocolatey een package mangager voor Windows net zoals apt-get voor Linux is. Het is ontworpen als een gedecentraliseerd framework met als doel het snel installeren van applicaties en tools. Chocolatey werd gebouwd boven op de NuGet infrastructuur gecombineerd voor het verspreiden van de packages en gebruikt PowerShell voor een gepersonaliseerde installatie.

In Tabel 2.1 zijn de verschillende voor- en nadelen gekoppeld aan Chocolatey terug te vinden. Het grootste voordeel dat bekomen wordt bij het gebruiken van een package manager is het al bestaan van een deployment infrastructuur. Na het installeren van Chocolatey op de client kunnen alle nodige packages voor het framework geïnstalleerd worden. Hiernaast kunnen scripts gekoppeld worden aan iedere package zodanig dat een aangepaste installatie mogelijk is. Net zoals apt-get voor Linux, is Chocolatey te gebruiken in de command-line. Dit is vooral een nadeel naar gebruiksvriendelijkheid toe aangezien er vanuit wordt gegaan dat de gebruikers amper tot geen ervaring hebben met de command-line in Windows/Linux. Het grootste nadeel aan deze technologie is, net zoals de vorige opties, het niet cross-platform zijn.

Qt Installer Framework

Het Qt Installer Framework biedt een set van tools aan voor het creëren van installers op verschillende platformen. Aan de hand van een set van pagina's wordt de gebruiker door het installatie-, update- en verwijderproces. Hierbij kunnen scripts gebruikt worden om het proces te vereenvoudigen [35].

Aan deze technologie zijn verschillende plus- en minpunten verbonden. Een vergelijking van de verschillende voor- en nadelen is terug te vinden in Tabel 2.1. Het grootste voordeel van het Qt Installer framework is het cross-platform zijn. Hierdoor is het mogelijk om installers te maken voor zowel Windows als Linux. Een nadeel dat hieraan verbonden is, is dat een Linux installer enkel kan gemaakt worden op in een Linux omgeving. Het is niet mogelijk om een Linux installer te maken op een Windows systeem. Hiernaast is het wel mogelijk om voor ieder pakket een aparte installatieprocedure te implementeren.

Hoofdstuk 3

Analyse en ontwerp

Uit de probleemstelling werd het snel duidelijk dat het probleem omtrent het Python testraamwerk complexer is dan op het eerste zicht lijkt. In dit hoofdstuk zal het probleem verder geanalyseerd worden. Aan de hand van deze bevindingen gaat een architectuur en structuur ontworpen worden. Deze vormen de basis voor de implementatie van de demo.

3.1 Analyse

Het probleem van Televic was het volgende: Televic fabriceert producten die moeten voldoen aan strenge veiligheidsnormen. Om hun producten hierop te kunnen testen heeft Televic een Python testraamwerk ontworpen waarmee het mogelijk wordt om de producten aan verschillende testscenario's te onderwerpen. Dit testraamwerk maakt gebruik van een grote set aan drivers en bibliotheken om een correcte werking te garanderen. Een direct gevolg hiervan is dat het installatieproces op een nieuwe testtoren een uitgebreide klus is. Hiernaast groeit het aantal gebruikers van het testraamwerk continu samen met het aantal drivers en bibliotheken. Het doel is om een systeem te ontwikkelen dat Televic kan bijstaan bij het installatie en verspreidingsproces. Een verdere analyse is wel nodig. Zo is het mogelijk om een applicatie te ontwikkelen die voldoet aan de initiële probleemstelling maar ook uitbreidbaar is naar de toekomst toe.

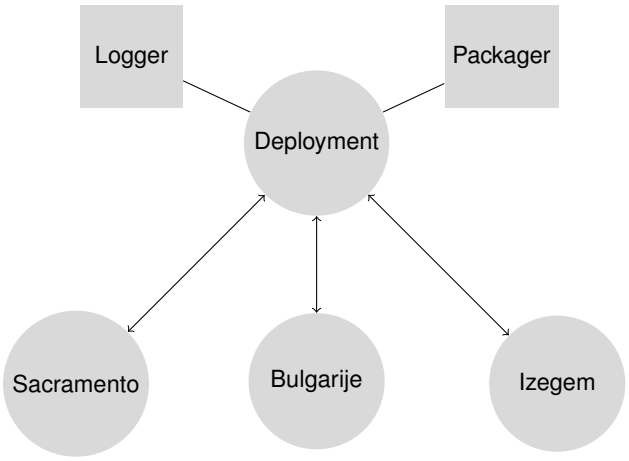
De probleemanalyse onthulde al snel dat dit probleem onder te verdelen is in verschillende deelproblemen. Het testraamwerk bestaat uit verschillende componenten, hieronder vallen de drivers en bibliotheken. Elke component heeft een aparte installatiewijze en moeten sommige componenten voor andere geïnstalleerd worden. Zo zal Python één van de eerste componenten zijn die geïnstalleerd moet worden. Hiernaast moeten verscheidene componenten geconfigureerd tijdens het installatieproces aan de hand van een configuratiebestand. Dit configuratiebestand hoort samen met de testtoren waarop het testraamwerk op geïnstalleerd word. Door gebruik te maken van additionele software worden verschillen in implementaties, door bijvoorbeeld verschillende programmeertalen, opgevangen. Een stuk van de applicatie zal dus bestaan uit deze additionele software die instaat voor het inpakken van de componenten. In de rest van de thesis zal naar dit onderdeel de *packager* heten. Hiervoor kan beroep gedaan worden op verscheidene technologieën, structuren en architecturen die besproken werden in Sectie 2.6.4 op pagina 15.

De probleemanalyse onthulde ook dat de verschillende executables verspreidt moeten worden. Door dit proces te automatiseren, is het mogelijk om waardevolle informatie te verzamelen. Met deze informatie kunnen rapporten gegeneerd worden over het deployment proces. In Secties 2.2 -

2.5 werden verschillende problemen maar ook oplossingen besproken die aan de basis liggen voor het ontwerp van dit onderdeel van de applicatie. In het vervolg van de thesis zal dit onderdeel (dat zal instaan voor het verspreiden van het testraamwerk maar ook voor de communicatie tussen de producten van het testraamwerk en de gebruikers) vermeld worden als de *deployment server*.

In Sectie 2.1 op pagina 3 werd besproken welke problemen kunnen optreden tijdens het installatieproces. Deze problemen moeten opgevangen worden om een schaalbare oplossing te bedenken voor Televic. Om dit op te vangen, kan er gebruik gemaakt worden van één (of meerdere) strategieën die besproken werd in Sectie 2.6 op pagina 13. Dit onderdeel van de applicatie vooral aanwezig aan de client-side aangezien dat de plaats is waar het testraamwerk aanwezig zal zijn. In de loop van de thesis zal naar dit onderdeel verwezen worden als de *deployment environment*.

Na de probleemanalyse is het nu duidelijk dat het werk op te delen valt in drie grote componenten. Deze drie onderdelen zullen de basis vormen voor de architectuur en zullen gebruikt worden als leidraad. Het eerste onderdeel zal bestaan uit de packager met als doel het inpakken van de nodige drivers, bibliotheken, ... Naast de packager is er de deployment server instaat voor het verspreiden van de installers die de packager aflevert. Aan de client-side zal de deployment environment aanwezig zijn waardoor installatie-complicaties vermindert worden door de installatie te isoleren. Mocht een rollback nodig zijn, dan kan deze op een eenvoudige manier gebeuren. In Figuur 3.1 wordt de algemene structuur van de applicatie weergegeven. Met behulp van deze basis is het mogelijk om een demo te produceren voor de finale verdediging.

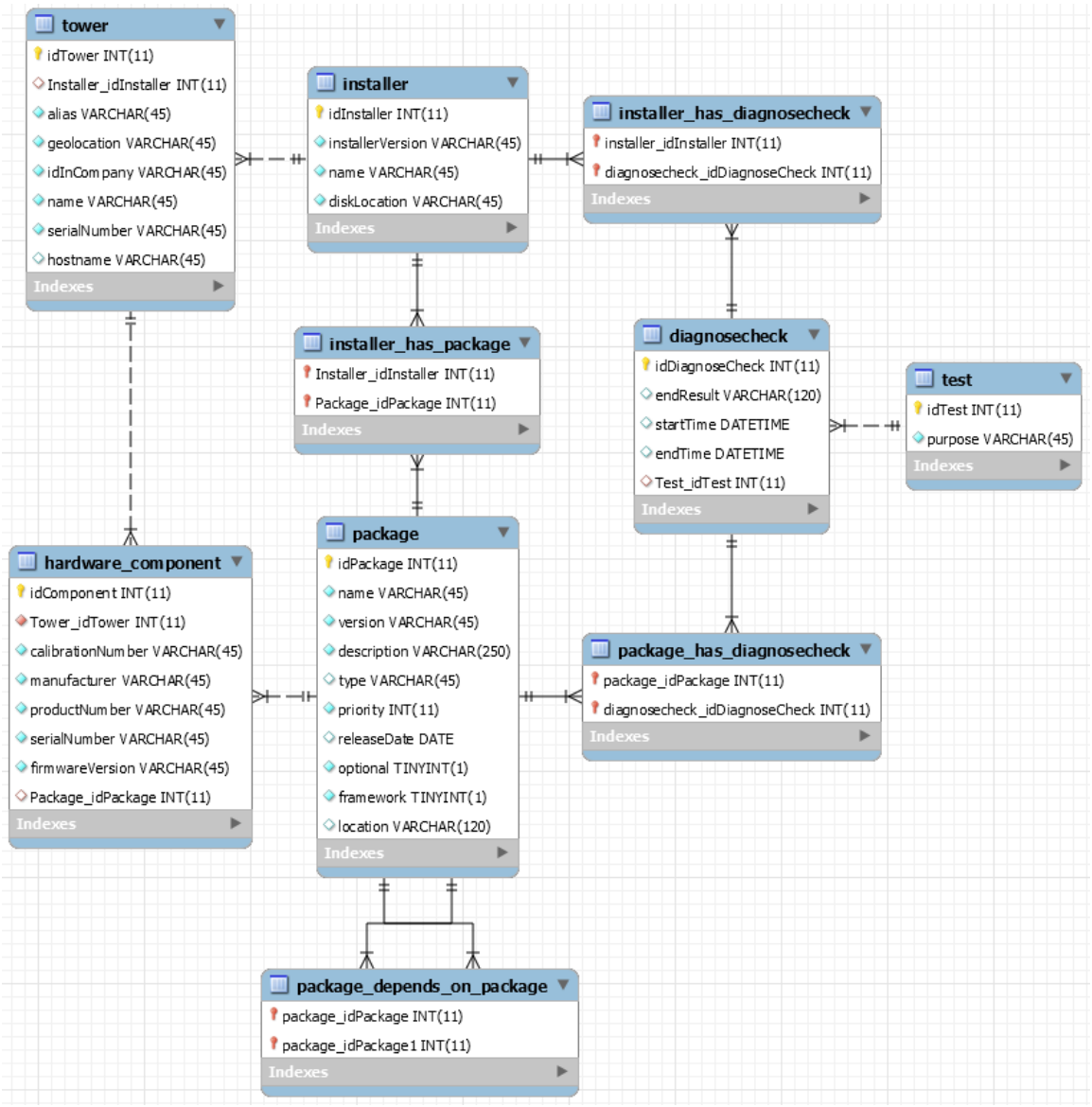


Figuur 3.1: Overzichtsdiagram van de algemene structuur

3.2 Databank ontwerp

In de voorgaande Sectie werd aangehaald met welke problemen Televic kampt. Eén van de problemen is de continue groei van pakketten waar het framework gebruikt van maakt en het aantal gebruikers die het framework gebruiken. Om dit probleem aan te pakken wordt een databank ontworpen voor het opslaan van alle cruciale data over zowel het installatieproces en alle gebruikers. In overleg met Televic werd ervoor gekozen om MySQL te gebruiken als managementsysteem. Het ontwerp van de databank is terug te vinden in Figuur 3.2 op de pagina hierna.

De tabellen tower en hardware_component dienen om iedere gebruiker, typisch een testtoren of een laptop, te beschrijven. Iedere toren heeft een ID, naam en serienummer. De combinatie van



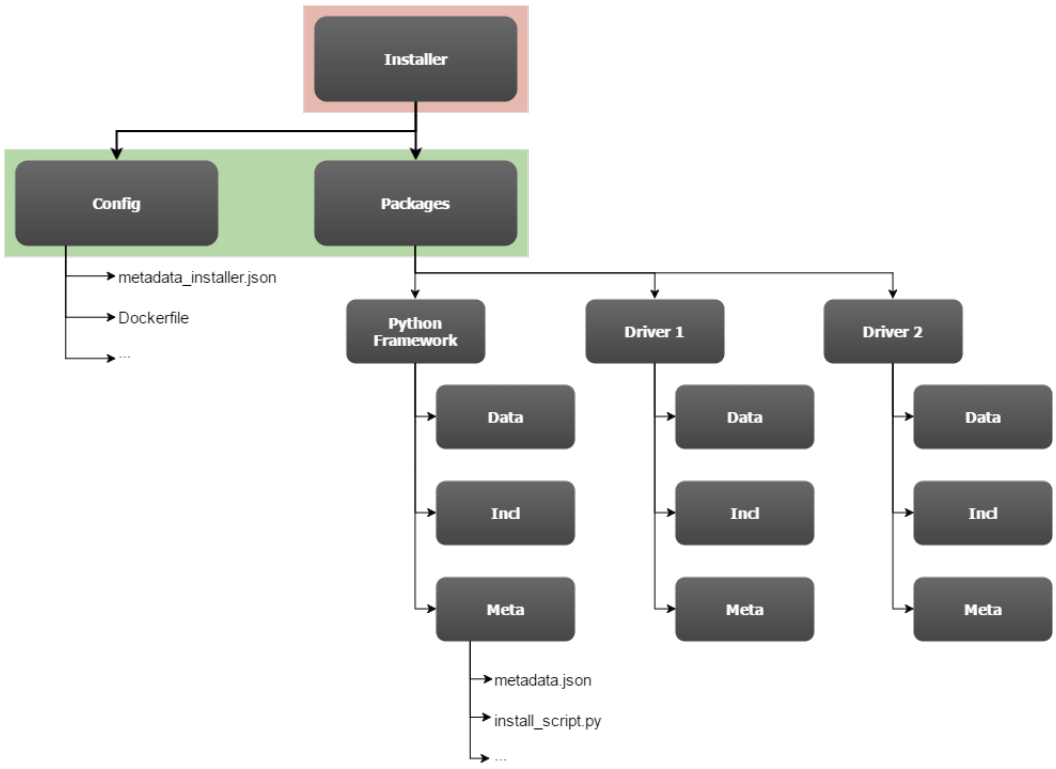
Figuur 3.2: Ontwerp van de databank

deze drie waarden is uniek binnen het bedrijf en de combinatie kan gebruikt worden als identificatie binnen het systeem maar deze combinatie draagt geen betekenis voor een gebruiker. Om dit op te vangen wordt aan iedere toren een alias gekoppeld waardoor de identificatie voor mensen vlotter kan verlopen. Elke toren bestaat typisch uit verschillende hardware componenten, zoals voedingen of netwerkkaarten, die nodig zijn om testen uit te voeren. Iedere component is gemaakt door een bepaalde fabrikant en krijgt van de fabrikant een serienummer. Vanuit het bedrijf wordt losstaand hiervan een nummer toegekend aan iedere component die gebruikt wordt om de calibratie instellingen te achterhalen. Iedere hardware component gebruikt firmware om correct te functioneren. Naast alle bovengenoemde informatie wordt ook de versie van de firmware opgeslagen. Door het opslaan van al deze informatie wordt het mogelijk om:

1. Torens van elkaar te onderscheiden

2. Achterhalen welke hardware componenten aanwezig zijn op welke toren
3. Welke firmware versie draait op welke hardware component

Naast informatie over de gebruikers wordt ook informatie over de verschillende installers en pakketten bijgehouden. Een installer bestaat uit een combinatie van verschillende software pakketten. Bij deze pakketten moet één pakket aanwezig zijn dat het Python testraamwerk bevat. Hiernaast zijn verschillende andere pakketten aanwezig voor drivers. In Figuur 3.3 is een voorbeeld zichtbaar van een installer die bestaat uit drie verschillende pakketten. Eén pakket wordt gebruikt door het testraamwerk en de twee anderen voor drivers die nodig zijn om het testraamwerk correct te laten functioneren. Iedere toren wordt gekoppeld aan één installer en zo aan één testraamwerk. Hiernaast is het mogelijk om pakketten te koppelen aan hardware componenten. Zo kan een driver voor een voeding gekoppeld worden aan de entry van de voeding die aanwezig is in de hardware_component tabel. Op deze wijze worden de hardware-software afhankelijkheden bijgehouden. Van ieder pakket wordt bijgehouden welk type pakket het is (een executabel, zip bestand, . . .), de prioriteit voor de installatievolgorde, een korte beschrijving en de release datum. Naast al deze informatie wordt er ook bijgehouden welke pakketten afhankelijk zijn van elkaar. Een voorbeeld is hiervan is een testraamwerk pakket en een pakket waarmee Python geïnstalleerd wordt. Het testraamwerk is afhankelijk van Python om correct te functioneren. Zo worden de verschillende software-software afhankelijkheden bijgehouden. Voordat een installer gemaakt wordt, die een testraamwerk pakket bevat, kan gecontroleerd worden dat ook het Python installatie pakket aanwezig is.



Figuur 3.3: Structuur van een installer bestaande uit drie pakketten

Verder zijn er enkele tabellen aanwezig voor het ondersteunen van testen. Tijdens en na het installatieproces moet het mogelijk zijn om testen uit te voeren. Dankzij deze testen is het duidelijk

of een bepaald pakket correct werkt en op het einde van het proces gecontroleerd worden of het volledige testraamwerk in zijn geheel functioneert. Doordat er een link wordt bijgehouden tussen een hardware component en een pakket, is het mogelijk om hieruit waardevolle informatie uit te halen. Zo kan bijvoorbeeld een verband gelegd worden tussen een bepaalde versie van een driver en de firmware die aanwezig is in een hardware component. Deze informatie kan gebruikt worden om problemen in testtorens te vermijden.

3.3 Architectuur

3.3.1 Packager

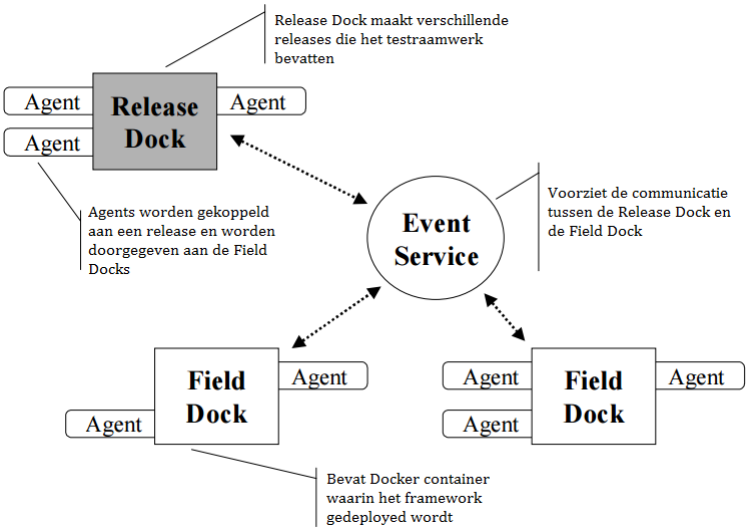
De architectuur van de packager wordt gebaseerd op de architectuur en structuur van het Qt installer framework. Er wordt een installer geproduceerd die bestaat uit verschillende pakketten die elk instaan voor het installeren van een software component. Voor iedere installer wordt een aparte folder structuur aangemaakt die zichtbaar is in Figuur 3.3 op de pagina hiervoor. In de config folder van de installer worden alle globale scripts en beschrijvingsbestanden bijgehouden. Verder bevat de installer subtrees voor ieder pakket. Een pakket bestaat vervolgens uit een data, include en meta folder. De data folder wordt gebruikt om de effectieve driver/bibliotheek in op te slaan. Hiernaast is een include folder aanwezig waarin verschillende afzonderlijke scripts toegevoegd kunnen worden. Op deze manier kunnen willekeurige scripts (bijvoorbeeld een script die de firewall instellingen aanpast) rap toegevoegd worden aan een pakket. Als laatste bevat de meta folder alle meta-data horende het pakket. Dit omvat onder andere een script die gebruikt wordt om te testen of het pakket wel correct functioneert maar ook een beschrijving van het pakket zelf.

Door zelf een packager te produceren, is het mogelijk om iedere stap in het deployment proces te personaliseren. Op deze manier kan na het installeren van een pakket een zo optimaal mogelijke afhandeling plaats vinden. Zo kunnen testen op ieder moment in het installatieproces toegevoegd worden, een handeling die met het Qt installer framework ook mogelijk is maar dit is moeilijker te realiseren. Doordat een gepersonaliseerde packager wordt ontworpen, worden problemen met Docker vermeden. Verder wordt besproken hoe Docker gebruikt wordt in de deployment environment om verscheidene problemen die gerelateerd zijn aan het installatieproces op te vangen. De Docker omgeving, zoals reeds vermeld in Sectie 2.6.3 op pagina 14, maakt gebruikt van LXC. Het besturingssysteem van de containers is hierdoor Linux. Windows gebruiken als besturingssysteem voor de containers is mogelijk in Docker is mogelijk maar deze optie staat nog altijd in beta schoenen. Mocht het Qt installer framework gebruikt worden als packager, dan moet het besturingssysteem van de server ook Linux zijn. Door ervoor te kiezen om zelf een packager te produceren, kan voor een cross-platform oplossing gezocht worden en kan de integratie van Docker vlotter verlopen. Op deze manier wordt er een abstractie gemaakt van het besturingssysteem van zowel de server als client.

3.3.2 Deployment server

Het centrale systeem in de architectuur is de deployment server. Zoals reeds uitgelegd zal dit onderdeel instaan voor het verspreiden van de verschillende installers en functioneren als een verzamelcenter voor alle informatie. De architectuur van de deployment server wordt gebaseerd op de software dock architectuur die besproken werd in Sectie 2.3 op pagina 6 en is terug te vinden in Figuur 3.4 op de volgende pagina. De software dock architectuur bestaat uit 4 grote

componenten, namelijk het release dock, field dock, event service en de agenten.



Figuur 3.4: Software Dock Architectuur [1]

Release Dock Het release dock bevindt zich aan de serverzijde en bevat de code voor zowel de packager als de monitor van de clients. Met hulp van de packager worden de verschillende installers geproduceerd. Naast het produceren van installers zal het release dock instaan voor het monitoren van de field docks. Hier wordt vooral gebruik gemaakt van de verschillende resultaten van de test die uitgevoerd worden tijdens het installatieproces. Op de manier wordt achterhaalt welke installer geïnstalleerd werd op een field dock en welke fouten eventueel optraden tijdens het installatieproces.

Event Service Nevens de verschillende docks beschrijft de software dock architectuur een Event Service. Deze staat in voor het afhandelen van de communicatie tussen de verschillende docks. Om dit te implementeren, wordt er beroep gedaan op Sectie 2.4 op pagina 7. Er worden evenwel enkele kleine aanpassingen gemaakt. De broker worden verschillende lijsten bijgehouden voor ieder type van bericht dat mogelijk is. Bij het toekomen van een subscribe/unsubscribe bericht, zal de broker de nodige lijsten aanpassen zodat de zender van het bericht wordt toegevoegd of verwijderd. Berichten in het netwerk zullen bestaan uit een set van attributen en waarden in een JSON formaat. Het uiterlijk van zo'n bericht is gelijkaardig als het uiterlijk van een bericht in Figuur 2.5 op pagina 8. In Listing 3.1 op de pagina hierna wordt een voorbeeld van een bericht weergegeven. Ieder bericht heeft dezelfde set aan attributen die elk een eigen functie hebben:

- *Timestamp*: ieder bericht zal op het moment van de creatie een timestamp krijgen. De timestamp wordt uitgedrukt in seconden sinds epoch. Op deze wijze wordt eenzelfde referentie punt gebruikt en is het mogelijk om de creatie van berichten in de tijd te ordenen.
- *Type*: ieder bericht hoort toe aan een bepaald type. De verschillende types van berichten zijn: subscribe, unsubscribe, new, change, rapport, update, release. Afhankelijk van het type bericht dat toekomt, bij zowel de broker als een dock, zal een andere actie ondernomen worden. De eerste twee types zijn uitsluitend bedoelt voor de broker. De volgende drie voor een release dock en de laatste twee voor de field docks.

- *ID*: naast een timestamp wordt aan ieder bericht een id meegegeven. Dit attribuut wordt meegegeven zodanig dat de field docks kunnen achterhalen of zij achterlopen op berichten.
- *Data*: Dit is het meest flexibele attribuut. Hier wordt de data meegegeven die hoort bij het bericht. Als het type van een bericht “subscribe” is, dan zit in het data veld een lijst met alle types waarvoor de gebruiker zich voor wilt inschrijven.
- *Sender*: Het laatste attribuut dat wordt meegegeven is de zender van het bericht. De broker moet bij een subscribe bericht kunnen achterhalen wie de zender is. Op die manier weet de broker wie moet worden toegevoegd aan de nodige lijsten.

Listing 3.1: Format voor een bericht

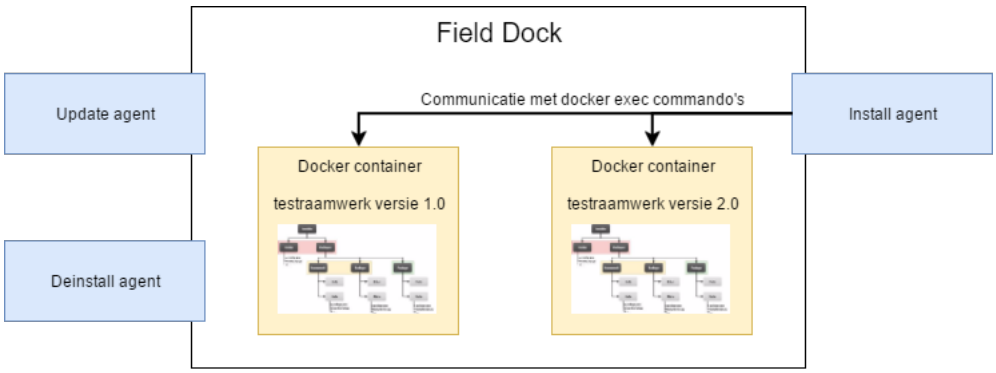
```
{
 'timestamp': 1491982212.555,
 'type': 'subscribe',
 'id': 0,
 'data': {
   'type': [
     'new',
     'change',
     'rapport'
   ]
 },
 'sender': 'localhost'
}
```

Agenten Naast de docks beschrijft de software dock architectuur ook agenten. Deze staan in voor het uitvoeren van allerlei deployment gerelateerde handelingen. Iedere agent is gekoppeld aan één stap uit de software levenscyclus die besproken werd in Sectie 2.1 op pagina 3. Hiernaast zal aan iedere installer afkomstig van het release dock een subset van alle agenten toegevoegd en verscheept worden naar het field dock. Op deze manier is het mogelijk om iedere stap in de levenscyclus van de installer te personaliseren. Ieder agent zal een bepaalde set van handelingen uitvoeren die overeenkomt met een deployment proces die besproken werd in de ORYA case studie in Sectie 2.5.4 op pagina 11. Net zoals bij ORYA wordt ieder deployment proces beschreven aan de hand van andere deployment processen en basis activiteiten. Een voorbeeld: Tijdens de creatie van een installer wordt een agent voorzien die instaat voor het installatieproces. De agent wordt samen met de installer verscheept naar het field dock waarna de agent op het gepaste moment in actie schiet. De agent begint met het hernoemen van de oude Docker container met daarin de vorige versie van een installer. Hierna wordt een nieuwe container aangemaakt waarin de nieuwe installer wordt losgelaten. Vervolgens zal de agent de installatie aanvangen en zullen de scripts horende bij de pakketten uitgevoerd worden in de container. Zoals reeds werd aangegeven, zijn de handelingen van een agent gebaseerd op het model van ORYA. Zo wordt het creëren van een nieuwe container in de installatie agent wordt gezien als zo’n basis activiteit. In Bijlage A op pagina 34 zijn verschillende flowcharts terug te vinden die horen bij enkele types van agenten. Door agenten te gebruiken, een strategie die ook gezien werd in de Atlas case studie in Sectie 2.5.3 op pagina 9, wordt het mogelijk om alle stappen in de software levenscyclus uniek te behandelen. Hiernaast kan bij iedere installer een andere set van agenten geassocieerd worden waardoor

iedere installer verder gepersonaliseerd kan worden.

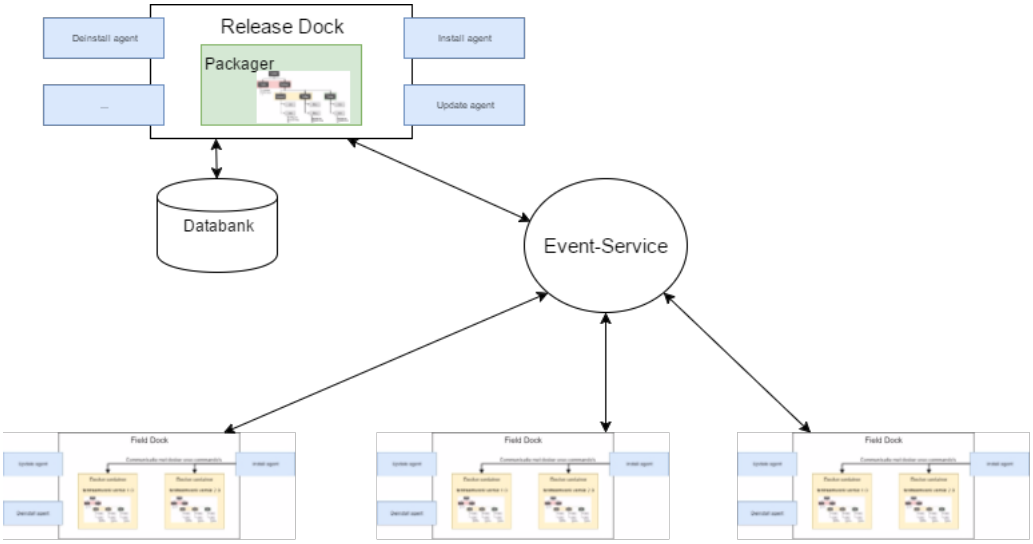
3.3.3 Deployment environment

De deployment environment komt overeen met de field dock in de software dock architectuur. In de omgeving gaat de installer, afkomstig van de packager, uitgevoerd worden zodanig dat het test framework geïnstalleerd wordt. Aan dit proces zijn de verschillende problemen verbonden die besproken werden in Sectie 2.1 op pagina 3. Om de verschillende deployment problemen te vermijden en om ervoor te zorgen dat geen uitgebreide rollback strategieën nodig zijn, wordt een geïsoleerde omgeving voorzien waarin de software geïnstalleerd wordt. Dit wordt gerealiseerd aan de hand van virtualisatie technieken, meer bepaald aan de hand van Docker. Docker wordt verkozen boven een gewone virtuele machine omdat het uitvoeren van handelingen (zoals opstarten, stoppen, . . .) op een container minder resources en tijd vraagt in vergelijking met een virtuele machine. Doordat een virtualisatie techniek wordt gebruikt, wordt het zeer eenvoudig om problemen tijdens de verschillende processen op te vangen. In Figuur A.6 op pagina 39 en Figuur A.7 op pagina 40 is het duidelijk dat, door het gebruik van Docker, het rollback proces zeer eenvoudig is. Figuur 3.5 geeft weer op welke manier de agenten met de containers gaan communiceren en geeft ook weer hoe Docker wordt geïntegreerd in het geheel.



Figuur 3.5: Structuur van een field dock

De volledige architectuur wordt weergegeven in Figuur 3.6 op de volgende pagina. In de figuur worden de verschillende docks afgebeeld samen met zowel Docker als de packager. Met deze architectuur is het mogelijk om aan de slag te gaan en een goede oplossing te vinden voor het probleem van Televic.



Figuur 3.6: Architectuur van het prototype

Hoofdstuk 4

Implementatie

Hoofdstuk 5

Evaluatie

Hoofdstuk 6

Conclusie

Referenties

[1] R. S. Hall, D. Heimbigner en A. L. Wolf, „A cooperative approach to support software deployment using the software dock”, in *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, IEEE, 1999, pp. 174–183.

[2] A. Dearie, „Software Deployment, Past, Present and Future”, in *Future of Software Engineering, 2007. FOSE '07*, mei 2007, pp. 269–284. DOI: 10.1109/FOSE.2007.20.

[3] A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A. Van Der Hoek en A. L. Wolf, „A characterization framework for software deployment technologies”, DTIC Document, tech. rap., 1998.

[4] E. Dolstra, *The purely functional software deployment model*. Utrecht University, 2006.

[5] H. Tian, X. Zhao, Z. Gao, T. Lv en X. Dong, „A Novel Software Deployment Method Based on Installation Packages”, in *2010 Fifth Annual ChinaGrid Conference*, jul 2010, pp. 228–233. DOI: 10.1109/ChinaGrid.2010.32.

[6] D. A. Patterson, „The data center is the computer”, *Communications of the ACM*, deel 51, nr. 1, pp. 105–105, 2008.

[8] R. S. Hall, D. Heimbigner, A. Van Der Hoek en A. L. Wolf, „An architecture for post-development configuration management in a wide-area network”, in *Distributed Computing Systems, 1997., Proceedings of the 17th International Conference on*, IEEE, 1997, pp. 269–278.

[9] P. R. Pietzuch en J. M. Bacon, „Hermes: A distributed event-based middleware architecture”, in *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*, IEEE, 2002, pp. 611–618.

[10] A. Carzaniga, D. S. Rosenblum en A. L. Wolf, „Design and evaluation of a wide-area event notification service”, *ACM Transactions on Computer Systems (TOCS)*, deel 19, nr. 3, pp. 332–383, 2001.

[11] M. J. Rutherford, K. Anderson, A. Carzaniga, D. Heimbigner en A. L. Wolf, „Reconfiguration in the Enterprise JavaBean component model”, in *International Working Conference on Component Deployment*, Springer, 2002, pp. 67–81.

[12] E. C. Bailey, *Maximum rpm*, 1997.

[13] I. Bird, K. Bos, N. Brook, D. Duellmann, C. Eck, I. Fisk, D. Foster, B. Gibbard, M. Girone, C. Grandi e.a., „LHC computing Grid”, *Technical design report*, p. 8, 2005.

[14] A. Salvo, A. Barchiesi, K. Gnanvo, C. Gwilliam, J. Kennedy, G. Krobath, A. Olszewski en G. Rybkine, „The ATLAS software installation system for LCG/EGEE”, in *Journal of Physics: Conference Series*, IOP Publishing, deel 119, 2008, p. 052 013.

[15] E. Obreshkov, S. Albrand, J. Collot, J. Fulachier, F. Lambert, C. Adam-Bourdarios, C. Arnault, V. Garonne, D. Rousseau, A. Schaffer, H. von der Schmitt, A. D. Salvo, V. Kabachenko, Z. Ren, D. Qing, E. Nzuobontane, P. Sherwood, B. Simmons, S. George, G. Rybkine, S. Lloyd, A. Undrus, S. Youssef, D. Quarrie, T. Hansl-Kozanecka, F. Luehring, E. Moyse en S. Goldfarb, „Organization and management of {ATLAS} offline software releases", *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, deel 584, nr. 1, pp. 244–251, 2008, ISSN: 0168-9002. DOI: http://dx.doi.org/10.1016/j.nima.2007.10.002. adres: http://www.sciencedirect.com/science/article/pii/S0168900207020967.

[16] G. Rybkin, „ATLAS software packaging", *Journal of Physics: Conference Series*, deel 396, nr. 5, p. 4, 2012.

[17] V. Lestideau en N. Belkhatir, „Providing highly automated and generic means for software deployment process", in *European Workshop on Software Process Technology*, Springer, 2003, pp. 128–142.

[18] N. Belkhatir, J. Estublier en W. Melo, „THE ADELE-TEMPO experience: an environment to support process modeling and enaction.", *Software Process Modelling and Technology Research Studies Press*, pp. 1–37, 2007.

[19] *Whitepaper: Ansible in depth*, https://cdn2.hubspot.net/hub/330046/file-480366556-pdf/pdf_content/Ansible_in_Depth.pdf?t=1487567092458, [Online; geraadpleegd 10-04-2017], 2016.

[20] S. M. Srinivasan, S. Kandula, C. R. Andrews, Y. Zhou e.a., „Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging.", in *USENIX Annual Technical Conference, General Track*, Boston, MA, USA, 2004, pp. 29–44.

[21] J. S. Plank, M. Beck, G. Kingsley en K. Li, *Libckpt: Transparent checkpointing under unix*. Computer Science Department, 1994.

[22] E. N. Elnozahy, L. Alvisi, Y.-M. Wang en D. B. Johnson, „A survey of rollback-recovery protocols in message-passing systems", *ACM Computing Surveys (CSUR)*, deel 34, nr. 3, pp. 375–408, 2002.

[23] Y. Huang, C. Kintala, N. Kolettis en N. D. Fulton, „Software rejuvenation: Analysis, module and applications", in *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, IEEE, 1995, pp. 381–390.

[24] S. Shumate, *Implications of Virtualization for Image Deployment*, 2004. adres: http://www.dell.com/downloads/global/power/ps4q04-20040152-Shumate.pdf.

[25] *Docker Main Page*, https://www.docker.com/, [Online; geraadpleegd 16-08-2016], 2016.

[26] Docker, „Docker for the Virtualization Admin", 2016.

[27] R. Chamberlain en J. Schommer, „Using Docker to support reproducible research", *DOI: http://dx. doi. org/10.6084/m9. figshare*, deel 1101910, 2014.

[28] D. Merkel, „Docker: lightweight linux containers for consistent development and deployment", *Linux Journal*, deel 2014, nr. 239, p. 2, 2014.

[29] J. R. Callahan, „Software packaging", tech. rap., 1998.

[30] *WiX Toolset*, http://wixtoolset.org/, [Online; geraadpleegd 3-08-2016], 2016.

[31] B. Amstadt en M. K. Johnson, „Wine", *Linux Journal*, deel 1994, nr. 4es, p. 3, 1994.

[32] H. Liefke en D. Suciu, „XMill: An Efficient Compressor for XML Data”, in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, reeks SIGMOD ’00, Dallas, Texas, USA: ACM, 2000, pp. 153–164, ISBN: 1-58113-217-4. DOI: 10.1145/342009.335405. adres: http://doi.acm.org/10.1145/342009.335405.

[33] *NSIS Main page*, http://nsis.sourceforge.net/Main_Page, [Online; geraadpleegd 3-08-2016], 2016.

[34] *Chocolatey About page*, https://chocolatey.org/about, [Online; geraadpleegd 3-08-2016], 2016.

[35] *QT Installer Framework Documentation*, http://doc.qt.io/qtinstallerframework/, [Online; geraadpleegd 4-08-2016], 2016.

[38] *WiX Toolset Tutorial*, https://www.firegiant.com/wix/tutorial/, [Online; geraadpleegd 3-08-2016], 2016.

[39] *Chocolatey Main page*, https://chocolatey.org/, [Online; geraadpleegd 3-08-2016], 2016.

Bibliografie

[7] J. Münch, O. Armbrust, M. Kowalczyk en M. Sotó, *Software process definition and management*. Springer Science & Business Media, 2012.

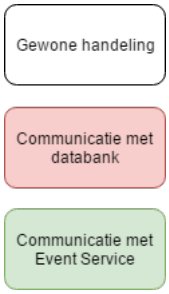
[36] R. S. Pressman, *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.

[37] D. Pitts, B. Ball e.a., *Red Hat Linux*. Sams, 1998.

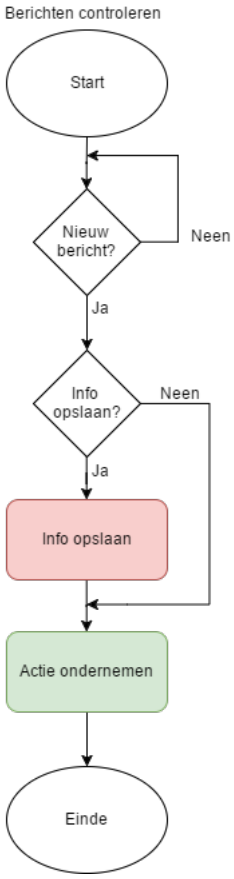
[40] G. S. Machado, F. F. Daitx, W. L. da Costa Cordeiro, C. B. Both, L. P. Gaspary, L. Z. Granville, C. Bartolini, A. Sahai, D. Trastour en K. Saikoski, „Enabling rollback support in IT change management systems", in *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*, IEEE, 2008, pp. 347–354.

Bijlage A

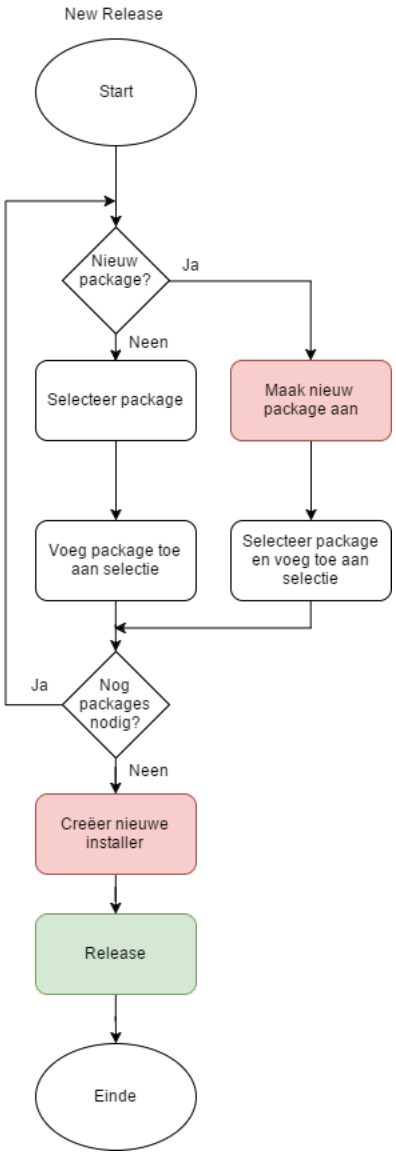
Flowcharts



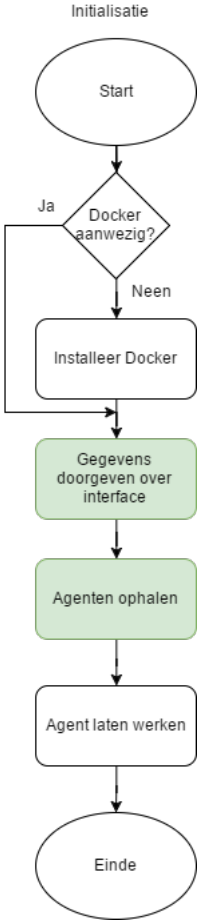
Figuur A.1: Flowchart kleurenlegende



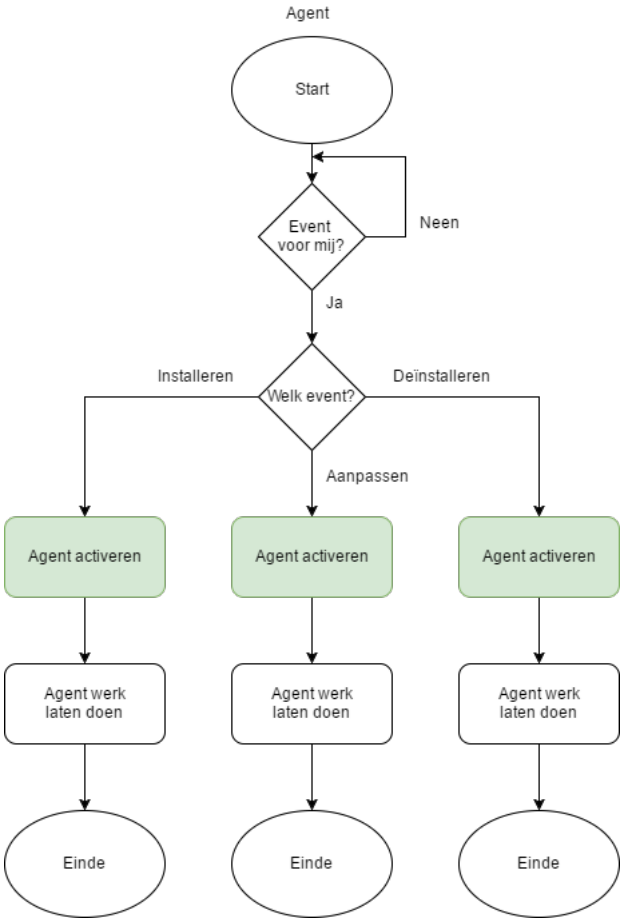
Figuur A.2: Flowchart voor het ophalen/controleren van berichten



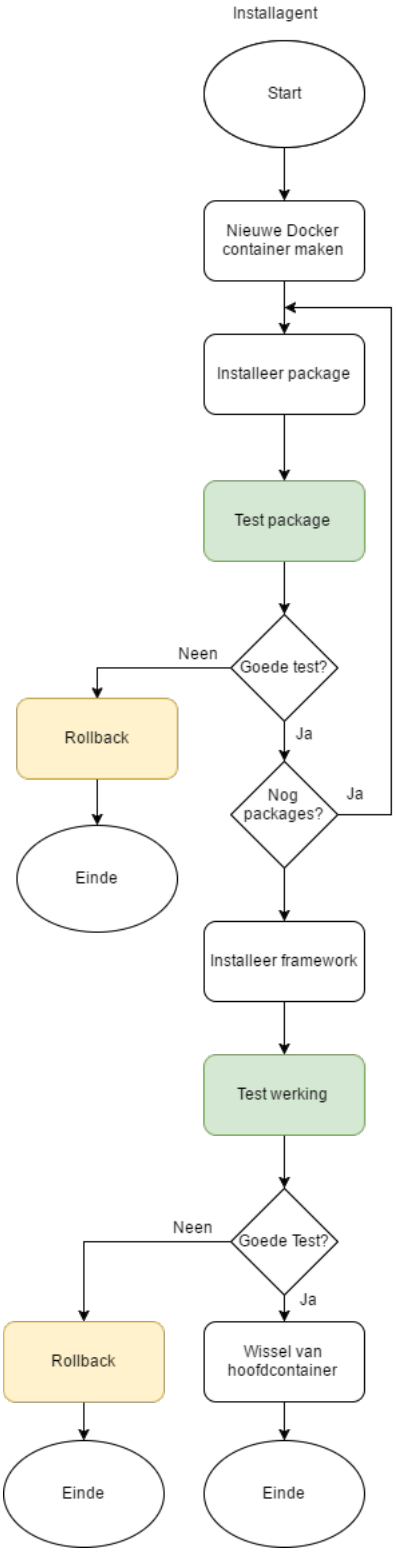
Figuur A.3: Flowchart van het creëren van een nieuwe release



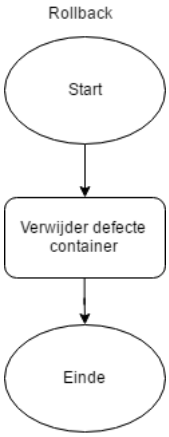
Figuur A.4: Flowchart van de initialisatie van een field dock



Figuur A.5: Flowchart van een agent



Figuur A.6: Acties van de installagent



Figuur A.7: Rollback

Bijlage B

Technologieën

Listing B.1: WiX Toolset installer

```
<?xml version="1.0"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
   <Product Id="*" UpgradeCode="12345678-1234-1234-1234-111111111111"
            Name="Python Framework Installer" Version="0.0.1"
               ↪ Manufacturer="PJ Industries" Language="1033">
      <Package InstallerVersion="200" Compressed="yes" Comments="Windows
          ↪  Installer Package"/>
      <Media Id="1" Cabinet="product.cab" EmbedCab="yes"/>

      <Directory Id="TARGETDIR" Name="SourceDir">
          <Directory Id="PersonalFolder">
          <Directory Id="INSTALLDIR" Name="Example">
           <Component Id="ApplicationFiles" Guid
               ↪ ="12345678-1234-1234-1234-222222222222">
              <RegistryKey Root="HKCU"
                                        Key="Software\My Application\
                                            ↪ Uninstall">
              <RegistryValue Value="testFile."
                                          Type="string"
                                          KeyPath="yes" />
              </RegistryKey>
                <File Id="ApplicationFile1" Source="python-2.7.3.msi
                    ↪ "/>
                <File Id="ApplicationFile2" Source="pyusb-1.0.0a2.zip
                    ↪ "/>
                <RemoveFolder Id='INSTALLDIR' On='uninstall' />
           </Component>
          </Directory>
          </Directory>
       </Directory>

      <Feature Id="DefaultFeature" Level="1">
```

```
            <ComponentRef Id="ApplicationFiles"/>
        </Feature>

        <InstallExecuteSequence>
            <Custom Action='FooAction' After='InstallFiles'/>
        </InstallExecuteSequence>
    </Product>
</Wix>
```

Listing B.2: NSIS installer

```
# define installer name
OutFile "installer0.2.0.exe"

# set desktop as install directory
InstallDir $PROFILE\NsisExample

# default section start
Section

# define output path
SetOutPath $INSTDIR

# specify file to go in output path
File python-2.7.3.msi
File pyusb-1.0.0a2.zip

# define uninstaller name
WriteUninstaller $INSTDIR\uninstaller.exe


#--------
# default section end
SectionEnd

Section

ExecWait '"msiexec" /i "$INSTDIR\python-2.7.3.msi" /quiet'

SectionEnd

Section

ZipDLL::extractall "$INSTDIR\pyusb-1.0.0a2.zip" "$INSTDIR"

ExecWait 'cmd /K "cd "$INSTDIR\pyusb-1.0.0a2" && "C:\Python27\python.exe
    ↪ " "setup.py" install && exit"'
```

```
SectionEnd

# create a section to define what the uninstaller does.
# the section will always be named "Uninstall"
Section "Uninstall"

ExecWait '"msiexec" /i "$INSTDIR\python-2.7.3.msi"'

# Always delete uninstaller first
Delete $INSTDIR\uninstaller.exe

# now delete installed file
Delete $INSTDIR\python-2.7.3.msi
Delete $INSTDIR\pyusb-1.0.0a2.zip

RMDir /r $INSTDIR\pyusb-1.0.0a2

SectionEnd
```

Bijlage C

Beschrijving van deze masterproef in de vorm van een wetenschappelijk artikel

Bijlage D

Poster

FACULTEIT INDUSTRIELE INGENIEURSWETENSCHAPPEN
TECHNOLOGIECAMPUS GENT
Gebroeders De Smetstraat 1
9000 GENT, België
tel. + 32 92 65 86 10
fax + 32 92 25 62 69
iiw.gent@kuleuven.be
www.iiw.kuleuven.be