

Ontwerp en ontwikkeling van een testraamwerk installer

Pieter-Jan ROBRECHT

Promotor: dr. ir. Annemie Vorstermans 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 communication technologies

Co-promotor: Carl Eeckhout (Televic)

©Copyright KU Leuven

Zonder voorafgaande schriftelijke toestemming van zowel de promotor(en) als de auteur(s) is overnemen, kopiëren, gebruiken of realiseren van deze uitgave of gedeelten ervan verboden. Voor aanvragen i.v.m. het overnemen en/of gebruik en/of realisatie van gedeelten uit deze publicatie, kan u zich richten tot KU Leuven Technologiecampus Gent, Gebroeders De Smetstraat 1, B-9000 Gent, +32 92 65 86 10 of via e-mail iiw.gent@kuleuven.be.

Voorafgaande schriftelijke toestemming van de promotor(en) 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.

Dankwoord

Ik zou graag mijn promotoren Annemie Vorstermans en Carl Eeckhout willen bedanken voor hun begeleiding en ondersteuning tijdens deze masterthesis.

Ik zou ook graag mijn vriendin, vrienden en familie willen bedanken voor hun steun en motivatie.

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

| | |
|---|-----------|
| 1 Inleiding | 1 |
| 1.1 Situering | 1 |
| 1.2 Probleemstelling | 1 |
| 1.3 Overzicht | 3 |
| 2 Literatuurstudie | 4 |
| 2.1 Inleiding | 4 |
| 2.2 Software deployment | 4 |
| 2.3 Case studies | 6 |
| 2.3.1 Electric cloud | 6 |
| 2.3.2 Redhat package manager | 8 |
| 2.3.3 ATLAS | 8 |
| 2.3.4 ORYA | 10 |
| 2.3.5 Ansible | 11 |
| 2.4 Recovery na fouten | 12 |
| 2.4.1 Rollback strategieën | 12 |
| 2.4.2 Virtualisatie | 13 |
| 2.4.3 Docker | 14 |
| 2.5 Deployment strategieën | 15 |
| 2.5.1 Data focus | 15 |
| 2.5.2 User focus | 16 |
| 2.6 Architecturen | 17 |
| 2.6.1 Software dock architectuur | 17 |
| 2.7 Technologieën voor software packaging | 20 |
| 2.7.1 Conclusie | 22 |
| 3 Analyse en ontwerp | 23 |
| 3.1 Inleiding | 23 |
| 3.2 Analyse | 23 |
| 3.3 Architectuur | 25 |
| 3.3.1 Deployment server | 29 |

| | |
|--|-----------|
| 3.3.2 Packager | 32 |
| 3.3.3 Databank ontwerp | 35 |
| 3.3.4 Deployment environment | 38 |
| 4 Implementatie | 41 |
| 4.1 Inleiding | 41 |
| 4.2 Server-side | 41 |
| 4.2.1 Deployment_server | 41 |
| 4.2.2 Dock | 42 |
| 4.2.3 Release_dock | 43 |
| 4.2.4 Agents | 44 |
| 4.3 Client-side | 46 |
| 4.3.1 Deployment_client | 46 |
| 4.3.2 Field_dock | 46 |
| 5 Evaluatie | 47 |
| 5.1 Testscenario's | 47 |
| 5.1.1 Meerdere clients | 47 |
| 5.1.2 Slecht werkend pakket | 48 |
| 5.1.3 Netwerk monitoring | 48 |
| 5.1.4 Cross-platform containers | 49 |
| 5.1.5 Meerdere servers | 49 |
| 5.2 SWOT analyse | 52 |
| 5.2.1 Strengths | 52 |
| 5.2.2 Weaknesses | 52 |
| 5.2.3 Opportunities | 53 |
| 5.2.4 Threats | 53 |
| 6 Conclusie | 54 |
| A Activiteitendiagrammen | 60 |
| B Listings | 64 |
| C Testscenario's | 70 |
| C.1 Meerdere servers | 70 |
| C.2 Meerdere clients | 73 |
| C.3 Slecht werkend pakket | 87 |
| C.4 Netwerk monitoring | 89 |
| D Beschrijving van deze masterproef in de vorm van een wetenschappelijk artikel | 94 |

E Poster

101

Lijst van figuren

| | | |
|------|--|----|
| 1.1 | Voorstelling van software verspreiding | 2 |
| 2.1 | De levenscyclus van software [3] | 5 |
| 2.2 | ElectricFlow Architectuur [6] | 7 |
| 2.3 | LJSFi Architectuur [9] | 9 |
| 2.4 | Deployment-procesmodel [14] | 11 |
| 2.5 | Voorbeeld van een deployment [14] | 11 |
| 2.6 | Type 1 hypervisor in vergelijking met een type 2 hypervisor [22] | 13 |
| 2.7 | Architectuur van Virtuele Machine ten opzichte van Docker [24] | 14 |
| 2.8 | Software Dock Architectuur [1] | 17 |
| 2.9 | Broker netwerk [32] | 19 |
| 2.10 | Publish/Subscribe service [33] | 19 |
| 2.11 | Publish/Subscribe Event [33] | 19 |
| 2.12 | Folder structuur voor installer | 22 |
| 3.1 | Overzichtsdiagram van de algemene structuur | 24 |
| 3.2 | Architectuur van het prototype | 26 |
| 3.3 | Klassendiagram van de applicatie | 28 |
| 3.4 | Deployment diagram van de applicatie | 28 |
| 3.5 | Ontwerp van de managersinterface | 29 |
| 3.6 | Ontwerp van de release creatie interface | 30 |
| 3.7 | Sequentie diagram voor het opstarten van een server | 33 |
| 3.8 | Structuur van een installer bestaande uit drie pakketten | 34 |
| 3.9 | Sequentie diagram voor het releasen van een installer | 36 |
| 3.10 | Ontwerp van de databank | 37 |
| 3.11 | Sequentie diagram voor het opstarten van een client | 39 |
| 3.12 | Structuur van een field dock | 40 |
| 3.13 | Grafische user interface van de client | 40 |
| 3.14 | Grafische user interface bij eerste gebruik | 40 |
| 4.1 | Sequentie diagram voor het opstarten van een server | 42 |

| | | |
|------|---|----|
| 5.1 | Error tijdens het gebruiken van Windows Docker containers | 50 |
| 5.2 | Ontvangen van client informatie bij servers | 51 |
| A.1 | Activiteitendiagram voor een rollback uit te voeren | 60 |
| A.2 | Activiteitendiagram voor het creëren van een installer | 61 |
| A.3 | Activiteitendiagram van een InstallAgent | 62 |
| A.4 | Activiteitendiagram voor het installer van een software pakket | 63 |
| C.1 | Beschrijven van een client | 70 |
| C.2 | Client overview van meerdere servers | 71 |
| C.3 | Error tijdens het aanpassen van de field dock gegevens | 72 |
| C.4 | Beschrijven van client in Windows | 74 |
| C.5 | Beschrijven van client in Linux | 75 |
| C.6 | Overview van de clients tijdens het toevoegen | 76 |
| C.7 | Overview van clients na toevoegen van de clients | 76 |
| C.8 | Creatie van een nieuwe installer | 77 |
| C.9 | Overview van clients na creëren van een installer | 77 |
| C.10 | Selecteren van een folder voor de installer in te plaatsen | 78 |
| C.11 | Output tijdens het installeren van de software | 79 |
| C.12 | Error tijdens het installeren van een installer in Linux | 80 |
| C.13 | Installatie van een eerste installer | 81 |
| C.14 | Overview van de clients na het verspreiden van de eerste installer | 82 |
| C.15 | Rapport tabel na uitvoeren van een installatie | 82 |
| C.16 | Output na het opstarten van de geïnstalleerde software | 83 |
| C.17 | Installatie van een tweede installer | 84 |
| C.18 | Output na het opstarten van de geïnstalleerde software | 85 |
| C.19 | Overview van de clients na het verspreiden van een tweede installer | 86 |
| C.20 | Opsomming van aanwezige containers voor het installatieproces | 87 |
| C.21 | Opsomming van aanwezige containers na het installatieproces | 87 |
| C.22 | Output tijdens het installatieproces | 88 |
| C.23 | Doorsturen van een notificatie | 90 |
| C.24 | Doorsturen van een installer en agenten naar een field dock | 91 |
| C.25 | Doorsturen van een rapport naar de broker | 92 |
| C.26 | Algemene statistieken bij het verzenden van een installer | 93 |

Hoofdstuk 1

Inleiding

1.1 Situering

Met meer dan 30 jaar ervaring in het ontwerpen en onderhouden van on-board communicatiesystemen is Televic Rail een toonaangevende producent van Passenger Information Systems, Entertainment Systems and Infotainment Systems. LiveCom is Televic Rail nieuwste generatie van informatiemanagement systemen. Het integreert alle aspecten van de on- en off-board reizigersinformatie, de infotainment en de entertainment. Het stelt operatoren in staat om hun volledige verkeersschema's, de dienstregelingen, de routes, de 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 communicatiemanagement, biedt het openbaar vervoer en spoorwegondernemingen een centraal systeem voor het creëren, het beheren, het distribueren en het 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 controlesystemen ontworpen. Alle systemen en apparaten zijn ontworpen in overeenstemming met de betreffende spoorwegsectornormen en voldoen aan de eisen voor passagiersruimte, draaistel en asmontage. On-board controllers verwerken sensordata-informatie en sturen deze naar de betreffende actuators en treinbeheersingssystemen. Fysische parameters die momenteel worden ondersteund zijn onder andere de versnelling, de druk, de rotatie, de temperatuur, het geluid en de verplaatsing.

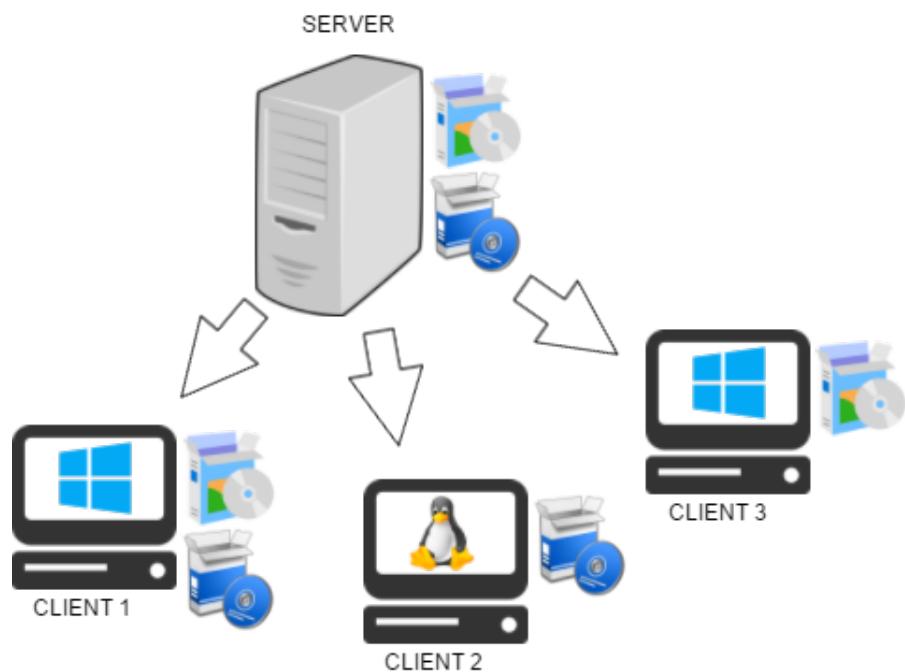
Om te voldoen aan de strenge veiligheidsnormen heeft Televic Rail een Python test framework ontworpen waarmee Televic in staat is om verschillende producten te onderwerpen aan verschillende testscenario's. Het framework werd ontworpen om gebruikt te worden op specifieke testtorens en werd later aangepast 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

Het Python testraamwerk moet correct functioneren met een grote hoeveelheid producten van Televic. Hierdoor gebruikt het Python testraamwerk verschillende drivers en bibliotheken. Een direct gevolg hiervan is dat het installatieproces tijdrovend en foutgevoelig is. Verder dient elk nieuw toe-

stel op het raamwerk ondersteunt te worden waardoor er jaarlijks vele releases van het raamwerk verspreid worden. Het installatieproces en het updateproces vragen om een vereenvoudiging zodat het testraamwerk gebruiksvriendelijker wordt.

Doordat het installatieproces en het updateproces foutgevoelig zijn, moet een “rampenplan” voorzien worden. Door de aanwezigheid van een “rampenplan”, worden mogelijke fouten vermeden en indien nodig opgevangen en verholpen. Zo wordt vermeden dat de productielijn moet worden stilgelegd door bijvoorbeeld een fout tijdens het updaten van een applicatie. Het is dus belangrijk dat applicaties op een eenvoudige manier geïnstalleerd en geüpdatet kunnen worden op verschillende hardware. Om dit te laten slagen moet het testraamwerk eerst bij de verschillende doelsystemen geraken. Er zullen dus producenten van software zijn die de verschillende testraamwerken produceren en willen verspreiden maar ook gebruikers die deze willen ontvangen. Het proces waarbij software van een producent naar alle toepasbare gebruikers wordt verspreid en vervolgens geïnstalleerd wordt, wordt ook wel het deployment proces genoemd. Een schets is zichtbaar in Figuur 1.1.



Figuur 1.1: Voorstelling van software verspreiding

Tijdens het zoeken naar een oplossing voor het installatieproces en updateproces van het Python testraamwerk, moet gezocht worden naar een oplossing die ook antwoorden biedt op toekomstige problemen. Tot op heden werd ervan uitgegaan dat de enige applicatie die verspreid, geïnstalleerd en geüpdatet moet worden het Python testraamwerk is. In de toekomst moet het evenwel mogelijk zijn om verschillende applicaties te ondersteunen. Het is dan ook belangrijk dat een oplossing gevonden wordt die een groeiend aantal gebruikers ondersteunt. Naargelang het aantal applicaties die ondersteund worden stijgt, zullen meer systemen gebruik maken van de applicatie. Naarmate het aantal gebruikers stijgt, stijgt ook de vraag naar een algemene administratieve interface. Hierdoor stijgt de vraag naar een algemene administratieve interface. Met een administratie interface wordt het mogelijk om bij te houden hoe het uitrollen van een nieuwe versie van een applicatie verloopt. Deze informatie kan gebruikt worden om het verspreidingsproces bij te sturen zodat een

volgende keer het proces vlotter verloopt. Naast het ondersteunen van verschillende applicatie moet tijdens het ontwerpen rekening gehouden worden met verscheidene besturingssystemen. De huidige programma's draaien op Windows maar het gebruik van Linux of andere systemen is in de toekomst niet uit te sluiten.

Het doel van deze thesis is dan ook een oplossing te vinden voor het complexe installatieproces en updateproces. Daarbij moet de software eerst van de producent bij de gebruikers geraken en moeten deze processen rekening houden met een "rampenplan" om fouten tijdens een installatie of update op te vangen. Verder wordt tijdens de ontwerp fase van de applicatie rekening gehouden met een groeiend aantal applicaties die verspreid en geïnstalleerd moeten worden, een groeiend aantal gebruikers die deze applicaties willen ontvangen en de diversiteit van besturingssystemen van deze gebruikers.

1.3 Overzicht

Het tweede hoofdstuk bevat een literatuurstudie over het deployment probleem. Er wordt nagegaan welke technologieën en architecturen beschikbaar zijn om de verschillende deelproblemen op te kunnen lossen.

Vervolgens vindt in het derde hoofdstuk een bespreking plaats van de ontworpen architectuur. In dit hoofdstuk wordt toegelicht waarom bepaalde architecturen en technologieën gekozen worden. Vervolgens wordt besproken hoe de verschillende architecturen en technologieën gecombineerd worden tot één geheel die de basis vormt voor de implementatie.

In het vierde hoofdstuk worden implementatiekeuzes toegelicht en soms uitgediept.

In hoofdstuk vijf om de ontworpen applicatie te onderwerpen aan verscheidene scenario's om te achterhalen of de applicatie voldoet aan de verwachtingen. Hierna volgt een SWOT analyse om na te gaan wat de sterktes en zwaktes zijn van de applicatie.

Hoofdstuk 2

Literatuurstudie

2.1 Inleiding

In dit hoofdstuk, wordt een bespreking gegeven over alle mogelijke technieken, technologieën, architecturen, ... die een oplossing kunnen bieden voor het probleem dat in het vorige hoofdstuk besproken werd. Eerst wordt het deploymentproces besproken en wordt nagegaan welke problemen hiermee geassocieerd worden. Vervolgens worden enkele case studies besproken om een beeld te krijgen van alle verschillende tools die aanwezig zijn en die gebruikt kunnen worden om het probleem omtrent het verspreiden van de software van de producent naar de gebruiker op te lossen. Na de cases studies worden ook enkele architecturen besproken die gebruikt kunnen worden om een applicatie te ontwerpen die om kan gaan met een groeiend aantal gebruikers. Hierna wordt nagegaan op welke manier een "rampenplan" geïmplementeerd kan worden zodat de problemen die ontstaan tijdens het deploymentproces opgelost kunnen worden. Uit de probleembespreking is ook gebleken dat er nood is aan een applicatie die overweg kan met verschillende programma's die Televic gemaakt heeft of nog zal maken. Om te achterhalen wat mogelijk is om dit te realiseren, worden verscheidene technologieën besproken die hiervoor een oplossing kunnen bieden.

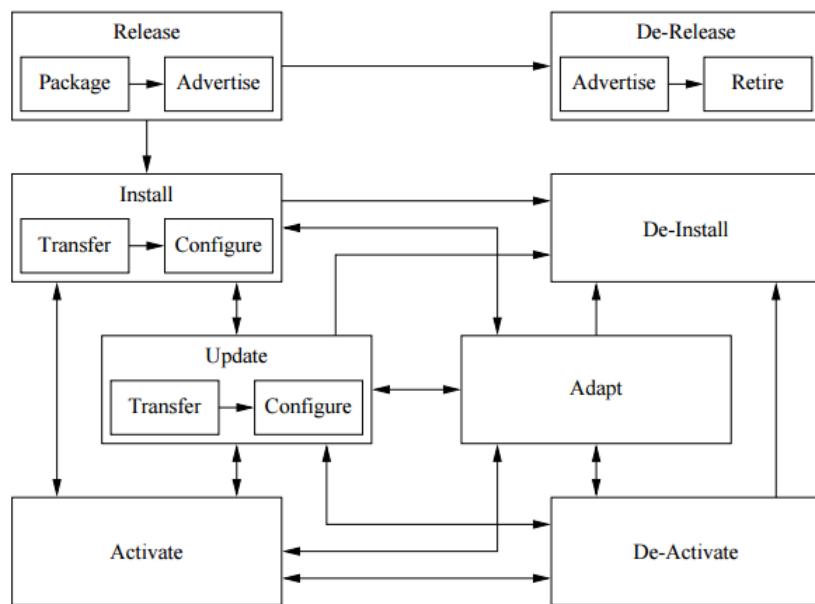
2.2 Software deployment

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 gebruiker 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 het tegengestelde van activeren. Deze stap is nodig zodat een aanpassing of herconfiguratie uitgevoerd kan 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 gebruikerssysteem gehaald worden.

De enige fase van de levenscyclus die zich uitsluitend op de server afspeelt is de release fase. De rest van de fases spelen zich af op de verschillende gebruikerssystemen.



Figuur 2.1: De levenscyclus van software [3]

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 deploymentproces. Deze oorzaken kunnen in twee grote categorieën ingedeeld worden, namelijk de omgevings- en de onderhoudsproblemen.

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

- De deployment van software kan een gedistribueerd probleem opleveren. Software kan afhankelijk zijn van componenten draaiende op externe systemen of van andere processen draaiende op het doelsysteem.
 - Software is vaak afhankelijk van verschillende andere softwarecomponenten. Deze afhankelijkheden (dependencies) moeten voor de deployment bepaald worden. Dit proces is moeilijk en een fout kan pas laat ontdekt worden.
 - De afhankelijkheden moeten compatibel zijn met wat de software verwacht. Bepaalde versies van een afhankelijkheid zijn compatibel met de software terwijl andere versies dit niet

zijn. Dit komt doordat sommige dependencies build-time variaties vertonen. At build-time worden bepaalde eigenschappen geselecteerd met als gevolg dat de dependency meer of minder functionaliteiten bevat ten opzichte van een vorige build.

- Sommige softwarecomponenten 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 moet aan deze eisen voldaan 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 worden. Enkele voorbeelden van zulke acties zijn:

- Tijdens het verwijderen van software moeten alle componenten verwijderd worden. Er mogen echter 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 vermeden.
- Na het upgraden/updaten van een component, is het soms aangewezen om een rollback uit te voeren. Zo'n actie kan overwogen worden als na de upgrade belangrijke functionaliteiten van de software niet meer functioneren.

2.3 Case studies

Uit de vorige sectie blijkt dat het software-deploymentproces een uitgebreid en ingewikkeld proces is. Er moet rekening gehouden worden met verscheidene stappen die elk een eigen doel en functie hebben, maar ook met verschillende problemen die kunnen optreden voor, tijdens en na deze stappen. 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 en wordt nagegaan op welke manier zij software-deployment aanpakken. Op deze manier wordt een beeld gecreëerd van de mogelijke technologieën die gebruikt kunnen worden om de software van Televic van de producenten naar de gebruikers te krijgen.

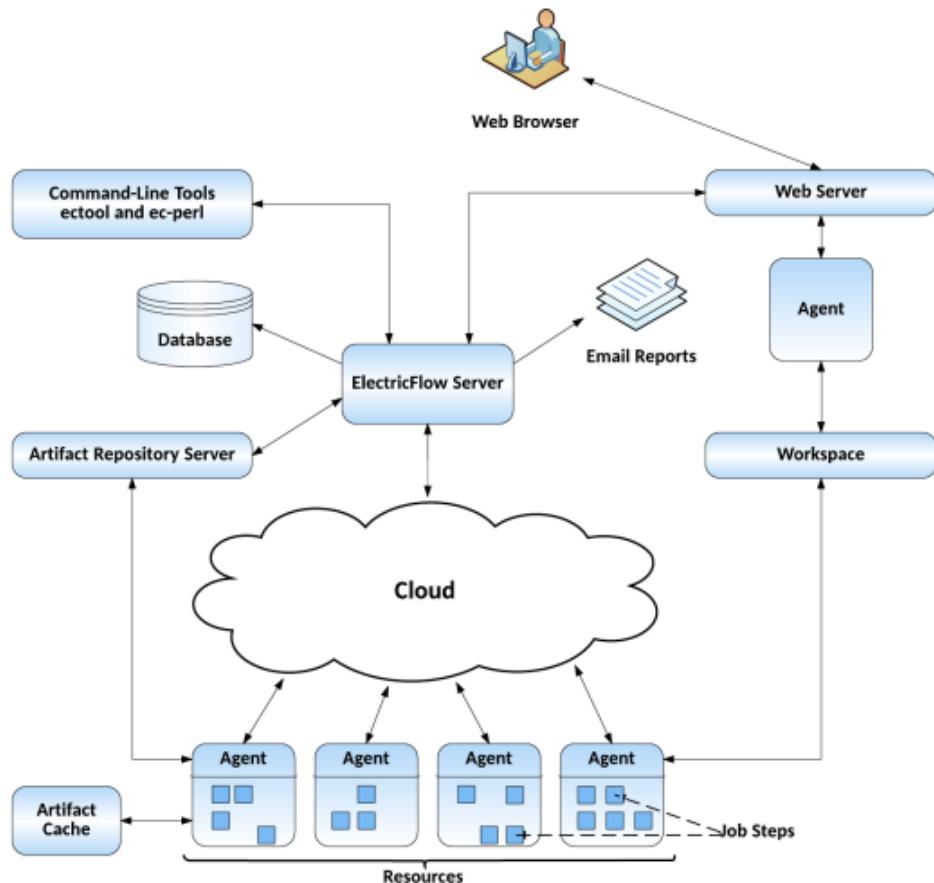
2.3.1 Electric cloud

Electric cloud is een bedrijf opgericht in 2002 met een focus op application release automation (ARA). Hun product, ElectricFlow, is ontworpen om het beschikbaar stellen, het bouwen en het verspreiden van multilayered applicaties te vereenvoudigen. Dit gebeurt aan de hand van de model-driven architectuur [5]. Cloud [6] beschrijft ElectricFlow als een enterprise-grade DevOps Release Automation platform. Met behulp van de model-driven aanpak is het mogelijk voor gebruikers om meerdere pipelines en releases over verschillende infrastructuren te coördineren op een eenvoudige manier.

De kern van ElectricFlow bestaat uit een web-based systeem dat gebruikt wordt voor het automatiseren en het onderhouden van het bouw-, test-, deployment en releaseproces. Het automatiseringsplatform bestaat uit een drie-lagen architectuur, een web interface en mogelijkheden om build en release analyses uit te voeren. De drie-lagen architectuur van ElectricFlow bestaat uit:

- **ElectricFlow Server:** een server die instaat voor het managen van resources, commando's en het generen van rapporten.
- **Databank:** een databank die instaat voor het opslaan van commando's, meta-data en logfiles.
- **Agents:** verschillende agenten die instaan voor het uitvoeren van commando's, het monitoren van statussen en het verzamelen van resultaten.

Figuur 2.2 geeft de architectuur weer van ElectricFlow. In de figuur zijn de verschillende lagen zichtbaar samen met de verschillende andere tools die vervaat zitten in het automatiseringsplatform [6].



Figuur 2.2: ElectricFlow Architectuur [6]

Om ElectricFlow te gebruiken voor bouw en test automatie, is het nodig om de volgende objecten te creëren, te configureren en bij te houden.

- **Project:** een project in ElectricFlow dient als container waarin procedures, stappen, workflows, ... zitten. Op deze manier is het mogelijk om een scheiding te creëren tussen projecten die bijvoorbeeld een ander doel hebben.
- **Resource:** een resource wordt gedefinieerd als een agent machine waar stappen in uitgevoerd kunnen worden.
- **Procedures:** procedure en stappen worden gebruikt om taken in ElectricFlow te definiëren. Een procedure bestaat uit één of meerdere stappen waarbij in ieder stap een commando of script wordt uitgevoerd.
- **Workflow:** een workflow stelt de gebruiker in staat om build-test-deploy levenscycli te definiëren. Zo wordt het managen van verschillende procedures en stappen in een project eenvoudiger.

Met ElectricFlow is het mogelijk om verscheiden applicaties op een autonome wijze te verspreiden naar de gebruikers. Hierbij kan door middel van een procedure een gepersonaliseerde afhandeling plaatsvinden. Door gebruik te maken van de verscheidene projecten is het mogelijk om de release van verschillende applicaties op een geordende manier aan te pakken. Verder wordt niets vermeld over hoe omgaan wordt met installaties en updates die slecht zijn afgelopen.

2.3.2 Redhat package manager

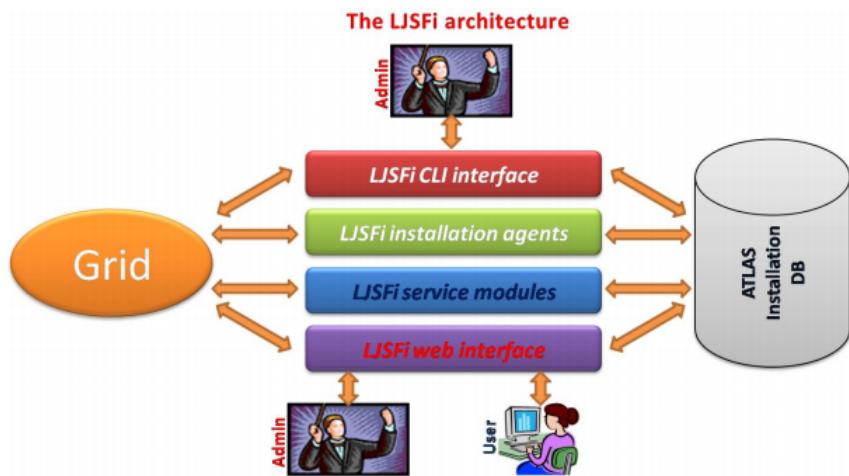
In Linux wordt de Redhat package manager (RPM) het vaakst gebruikt voor de deployment van software. Met hulp van RPM is het mogelijk om enkele operaties uit te voeren zoals onder andere installatie, updaten, De operaties worden ondersteund door een databank die alle informatie en details van de geïnstalleerde pakketten bevat. Een RPM pakket bestaat uit executables gecombineerd met configuratiebestanden en documentatie. Doordat een pakket executables bevat, zal een pakket gekoppeld zijn aan het besturingssysteem van de host [7]. 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 een grofkorrelige, taal onafhankelijke maar besturingssysteem afhankelijke oplossing. Het grootste probleem van RPM schuilt in de afhankelijkheden tussen de pakketten. Niet alle afhankelijkheden zijn expliciet gemodelleerd en de afhankelijkheden die wel gemodelleerd zijn, zijn gevormd met de nadruk op de inhoud en niet op de pakketten zelf [2].

2.3.3 ATLAS

ATLAS is één van de vier grote experimenten bij de Large Hadron Collider (LHC) in CERN (de Europees organisatie voor nucleair onderzoek).

Om in de ATLAS-samenwerking (specifiek voor het LCG/EGEE project, LHC Computing Grid/E-enabling Grids for E-sciencE [8]) om te gaan met de grote hoeveelheid bronnen, is er een volledig automatisch installatiesysteem ontworpen. Salvo, Barchiesi, Gnanno e.a. [9] beschrijft de architectuur van het ontworpen systeem. Het ontwerp van het installatiesysteem werd gebaseerd op het Light Job Submission Framework for installation, ook wel LJSFi.



Figuur 2.3: LJSFi Architectuur [9]

De architectuur van het framework is zichtbaar in Figuur 2.3. Het framework vormt een dunne laag over de middleware van Grid (Grid is een gedistribueerde computerinfrastructuur voor geavanceerde wetenschappen. De focus ligt bij het delen van resources en innovatieve applicaties.[10]). De kern van het systeem bestaat uit de installatiedatabase en de command line interface (CLI). De laatste zorgt voor de interacties met de Grid middleware. Met hulp van de installatiedatabase kan de CLI de verschillende taken en de job informatie opslaan. Aan de hand van deze informatie kunnen installaties uitgevoerd worden. De installatiedatabase 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 servicemodules en extensies waarmee installatie aanvragen afgehandeld worden. Het systeem bevat drie verschillende componenten die horen bij de LJSFi service modules:

- **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 de software release aangeduid staat als productie of verouderd en als de release aangeduid staat met de parameter auto-installatie. De module zal respectievelijk de software installeren of verwijderen op alle sites waar de software tag nog niet aanwezig 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 installatiedatabase gelezen en via de CLI interface worden de nodige installatieprocessen opgestart. Elk installatieproces wordt bijgestaan door een installatie agent. De agent zal instaan voor het updaten van de installatiedatabase met real-time informatie die online zichtbaar is.

Naast de verscheidene automatische services biedt LJSFi enkele gebruikerservices aan. Een gebruiker kan zich inschrijven voor bepaalde acties op een doelsysteem. 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 sitecontrole uitgevoerd door een piloot taak naar de site te sturen. Als de check succesvol uitgevoerd wordt, kan het installatieproces beginnen. De acties tijdens het installatieproces worden uitgevoerd door softwaremanagementscripts. Op het einde van het proces, haalt het systeem de joboutput en exit-code op. De laatste wordt opgeslagen in de installatiedatabase.

Obreshkov, Albrand, Collot e.a. [11] bespreken hoe het ATLAS project te werk gaat bij het inpakken van alle nodige software. Het ATLAS project gebruikt CMT [12] als configuratie manager. Met behulp van een configuratie bestand weten verscheidene tools hoe ze een pakket moeten afhandelen. Rybkin [13] 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 behandeld moet worden.

De LJSFi architectuur is een architectuur die om kan gaan met een grote hoeveelheid pakketten. Door gebruik te maken van de modules is het mogelijk om de verschillende pakketten te installeren en te verwijderen op een grote schaal zonder dat menselijke interactie nodig is. Hierbij zorgt CMT voor de nodige meta-data waardoor installatiertools niet afhankelijk zijn van de mens. Een nadeel aan deze architectuur is dat het gebruik van Grid nodig is om de LJSFi architectuur in te bouwen. Verder wordt in de lectuur niet ingegaan op welke wijze er wordt omgegaan met een fout tijdens het installatieproces of het verwijderproces. Het enige dat hierover vermeld wordt is dat de exit-code van de job niet rechtstreeks gebruikt worden.

2.3.4 ORYA

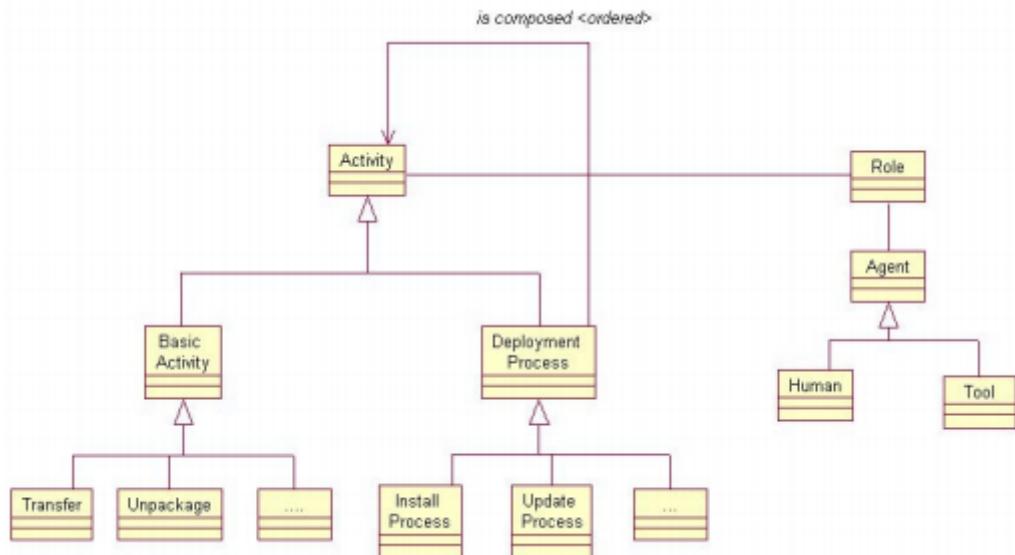
Lestideau en Belkhatir [14] leggen uit hoe ORYA (Open enviRonment to deploY Applications) verschillende deployment functionaliteiten aanbiedt aan gedistribueerde, autonome entiteiten zoals workstations en servers. Aan de hand van een deployment PSEE [15] wordt het mogelijk om het installatieproces te automatiseren.

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

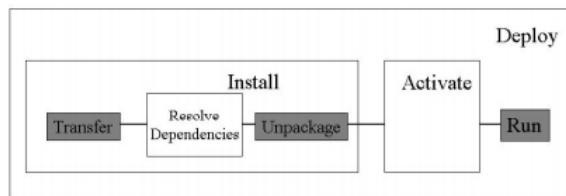
- **Applicatie Server** De applicatie server bevat de informatie nodig voor de installatie. Hieronder bevindt zich onder andere een pakket met de nodige resources en een manifest waarin de afhankelijkheden, beperkingen en kenmerken staan.
- **Target** De target is het doel waarop de deployment uitgevoerd 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 andere programma's correct blijven functioneren.

Lestideau en Belkhatir [14] beschrijven verder de verschillende modellen die gehanteerd worden om een succesvolle deployment uit te voeren. In Figuur 2.4 is het deployment-procesmodel terug te vinden. Een deploymentproces zal bestaan uit verschillende basisactiviteiten en deploymentprocessen. Iedere activiteit wordt uitgevoerd door een agent.

Een toegepast voorbeeld van een deployment aan de hand van dit model is terug te vinden in Figuur 2.5. Een basisactiviteit wordt voorgesteld aan de hand van een grijze rechthoek en een

**Figuur 2.4:** Deployment-procesmodel [14]

deploymentproces aan de hand van een witte rechthoek. In het voorbeeld zijn dus vier deploymentprocessen aanwezig en 3 basisactiviteiten.

**Figuur 2.5:** Voorbeeld van een deployment [14]

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

2.3.5 Ansible

Als laatste case studie wordt Ansible besproken. Ansible is een open source IT configuratiemanagement-, deployment- en organisatietool. De architectuur van Ansible bestaat uit een agentless push model. Dit wil zeggen dat er geen additionele software nodig is op de clienttoestellen. Dit wordt gerealiseerd door gebruik te maken van de remote management 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 [16].

[16] beschrijft verder dat Ansible gebruik maakt van *Playbooks* voor de organisatie van de IT omgevingen. Playbooks zijn YAML¹ 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

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

op een subset van de inventory uitgevoerd kan worden. Een taak bestaat uit Ansible modules (Een klein stuk code met een specifieke taak).

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 en output gebruikt moet worden. De inventory van een Playbook kan dynamisch ontdekt worden at runtime. In tegenstelling tot ATLAS en Electric Cloud bevat Ansible geen agenten die gebruikt worden om het deploymentproces in goede banen te leiden. Wederom is het niet duidelijk wat er gebeurt mochten er fouten optreden tijdens het installatieproces of updateproces.

2.4 Recovery na fouten

Tijdens de besprekking van het probleem in Sectie 1.2 werd al aangehaald dat een soort van “rampenplan” voorzien moet worden. Zowel omgevingsproblemen als onderhoudsproblemen (zie Sectie 2.2) kunnen tijdens één van de toestanden in de deployment levenscyclus optreden. Voor Televic is het niet wenselijk dat een probleem tijdens één van deze toestanden ervoor zorgt dat hun software niet meer correct functioneert aangezien dit kan leiden tot het stilleggen van de productie. Daarom wordt in de volgende secties verschillende mechanismen besproken waarmee de verscheidene problemen aangepakt kunnen worden. In wat volgt, wordt vooral rekening gehouden met mechanismen die gebruikt kunnen worden als het probleem zich al voor heeft gedaan. Deze strategieën zullen typisch proberen terug te keren naar een vorige correct werkende toestand van de software. De actie waarbij wordt teruggekeerd naar een vorige toestand wordt ook wel rollback genoemd.

2.4.1 Rollback strategieën

Srinivasan, Kandula, Andrews e.a. [17] spreken over drie manieren waarop rollback strategieën geïmplementeerd worden in hedendaagse systemen. Checkpointing, main-memory transactions en software rejuvenation zijn strategieën die vaak gebruikt worden.

Checkpointing Checkpointing is een eenvoudige recovery-strategie. De toestand 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 [18]. Bij een checkpointing systeem wordt de volledige toestand van het programma opgeslagen. Dit zorgt voor een verhoogde overhead waardoor het niet mogelijk is om frequent een checkpoint uit te voeren [17].

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 [18], [19].

Main-memory transactions Een andere rollback strategie is de main-memory transactions. Systemen die deze transacties ondersteunen bezitten vaak de mogelijkheid om terug te keren naar een vorig uitvoeringspunt. Om deze strategie te kunnen implementeren, moeten applicaties gebruik

maken van het transactieprogrammeermodel, waardoor de keuzevrijheden van de programmeur beperkt worden [17].

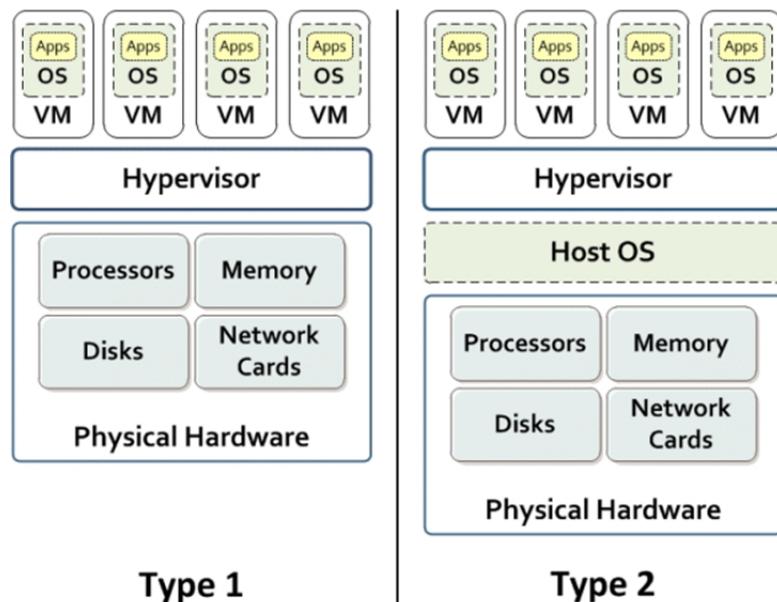
Software rejuvenation Huang, Kintala, Kolettis e.a. [20] definieert software rejuvenation als volgt:

“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. De meeste studies over software rejuvenation focussen vooral op het herstarten van de volledige applicatie en werken dus niet op een fijnkorrelige schaal [17].

2.4.2 Virtualisatie

Door middel van virtualisatie wordt de complexiteit die ontstaat door de interactie tussen het programma, de installatieomgeving en de uitvoeringsbeperkingen gelimiteerd. Er zijn verschillende voordelen gekoppeld aan het gebruiken van een virtuele machine (VM). Bijvoorbeeld, besturingssystemen op verschillende hardware platformen eisen verschillende drivers en deze drivers hebben misschien afhankelijkheden op een bepaalde firmware en BIOS. Een gastbesturingssysteem draaiende op een virtuele machine heeft deze eisen niet [21].

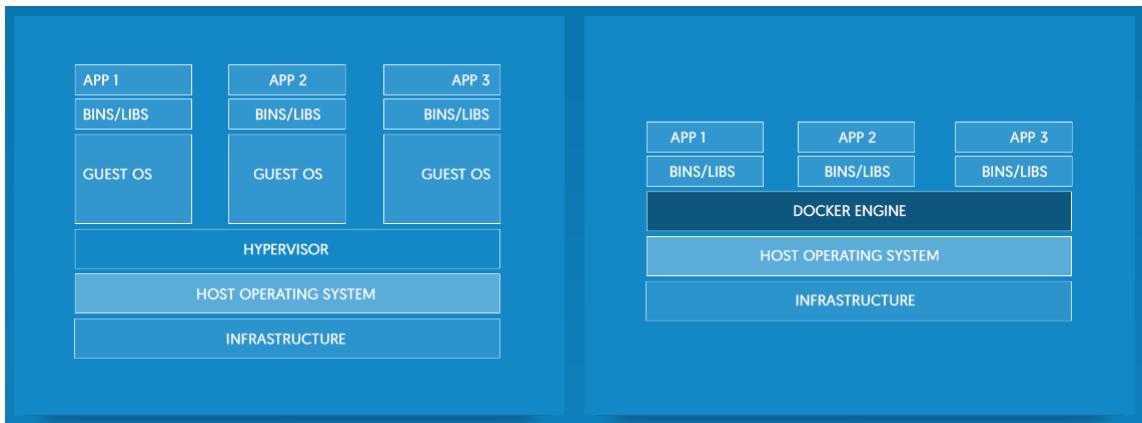


Figuur 2.6: Type 1 hypervisor in vergelijking met een type 2 hypervisor [22]

Met virtualisatie is het mogelijk om van één computer meerder computers te maken. Dit kan bereikt worden door gebruik te maken van een speciaal programma (een hypervisor). Volgens Fenn, Murphy, Martin e.a. [23] bestaan er twee types hypervisors (zie Figuur 2.6):

- **Type 1** hypervisors zorgen voor een directe interface naar de host hardware. Er is typisch één virtuele machine met speciale privileges die de andere virtuele machines onderhoudt.
- **Type 2** hypervisors draaien als een normaal programma in het host besturingssysteem. Iedere virtuele machine draait dan als proces in het host besturingssysteem.

Een virtuele machine bevat een volledig besturingssysteem. Een gevolg hiervan is dat het uitvoeren van handelingen (zoals opstarten, afsluiten, kopiëren, ...) lang duren. Hiernaast moet tijdens het instellen van een virtuele machine een volledig besturingssysteem geïnstalleerd worden. De tijd die nodig is om een VM in te stellen neemt hierdoor drastisch toe. Aangezien een virtuele machine een volledig besturingssysteem omvat, heeft deze ook de resources nodig om correct te kunnen functioneren. Om verschillende virtuele machines naast elkaar te kunnen laten draaien, moet de host voldoende RAM, CPU en harde schijf ruimte hebben. Door gebruik te maken van een virtuele machine wordt een veilige omgeving gecreëerd maar dit neemt kostbare tijd en resources in beslag.



Figuur 2.7: Architectuur van Virtuele Machine ten opzichte van Docker [24]

2.4.3 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 filesystem dat alle benodigdheden bevat om correct te functioneren. Hierdoor zal de software overal op eenzelfde manier draaien, ongeacht de omgeving [24]. Docker containers zijn niet hetzelfde als virtuele machines. In [25] 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 kan van grootte verschillen. Er bestaan kleine appartementen maar ook grote met meerder slaapkamers. Men huurt enkel hetgeen nodig is.

In Figuur 2.7 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 gedraagt zich als een geïsoleerd proces in de user space van het host besturingssysteem.

Chamberlain en Schommer [26] bespreken 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 virtualisatietechniek waarmee virtuele omgevingen in Linux opgebouwd kunnen worden. De containers zullen processen van elkaar gescheiden houden zodat een proces een ander niet kan beïnvloeden [27]. 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 filesystem voor de containers. Doordat het filesystem gelaagd is, is het mogelijk om verschillende filesystemen over elkaar te leggen.

Merkel [27] vergelijkt de twee virtualisatie technieken. 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.

2.5 Deployment strategieën

Voordat software bij een gebruiker geïnstalleerd of geüpdatet kan worden, moet de software eerst bij de gebruiker geraken. Er kan op twee manieren naar dit proces gekeken worden:

- Wat wordt er precies verzonden naar de gebruiker?
- Wie krijgt de software als eerste?

In de volgende secties worden deze twee verschillende focussen verder besproken.

2.5.1 Data focus

Tian, Zhao, Gao e.a. [28] haalt drie methodes aan om software te deployen:

- disk image-based deployment
- behavior-based deployment
- package-based deployment

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. Zo zal de image-server van software A een andere image-server zijn dan de image-server die software B gebruikt. 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. [28] al aangeeft, is deze methode niet flexibel. Enkel software die op voorhand werd geconfigureerd op de image-server kan worden gedeployed. Gebruikers met speciale noden kunnen moeilijk worden geholpen. Voor Televic is flexibiliteit een hoofddoelstelling. Iedere node bevat verschillende hardware en is verschillend geconfigureerd. Een disk image-based deployment zal hierdoor niet gebruikt kunnen worden. Het basisidee achter behavior-based deployment is het opnemen van de schijfoperaties tijdens het deployen. Als geweten is welke bestanden aangepast, gecreëerd zijn, ... dan kan het proces nagebootst worden op andere nodes [28]. Zo een proces nabootsen is moeilijk. Kerneloperaties 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. [28] 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.

2.5.2 User focus

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 [29] 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 [29] 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. [30] spreken 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.

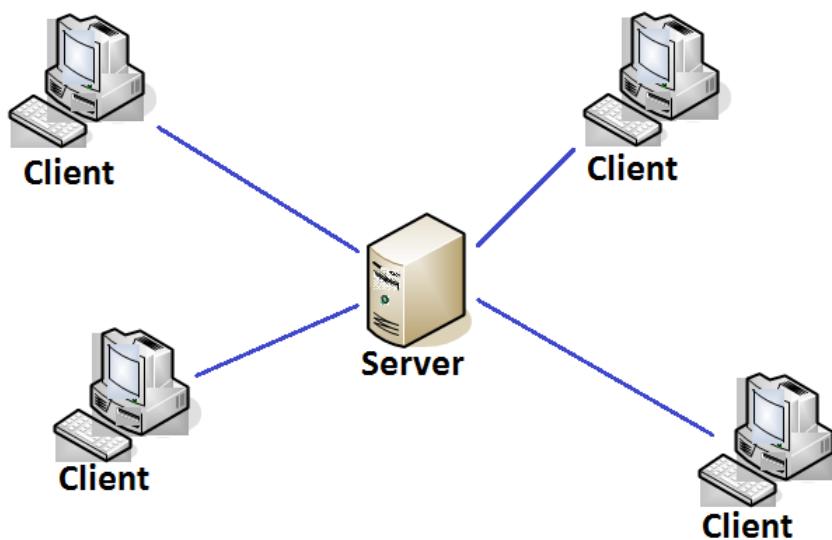
Aan beide strategieën zijn zowel voor- en nadelen gekoppeld. Zo zal de big-bang strategie niet voordelig zijn om uit te voeren als de gebruikers verspreid zitten over de wereld. Door het tijdsverschil zal de deployment bij sommige gebruikers plaatsvinden tijdens de werkuren en bij anderen midden in de 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.

2.6 Architecturen

Tot op heden werd vooral gekeken naar mogelijke technologieën en tools die gebruikt kunnen worden voor het verspreiden van software. In de komende secties worden enkele architecturen aangehaald die ook een oplossing bieden voor het verspreiden van software. Met een architectuur is het mogelijk om zelf verscheidene ontwerp- en implementatiebeslissingen te nemen. Zo ligt de programmeertaal niet vast en kan een taal gekozen worden die besturingssysteem onafhankelijk is. In de volgende secties wordt vooral gekeken welke architecturen een schaalbare oplossing bieden voor het deploymentproces.

2.6.1 Client-server architectuur

[55] beschrijft de client-server architectuur als een architectuur van een computernetwerk waarin verscheidene clients services van een gecentraliseerde server opvragen en ontvangen. De client computers dienen als interface voor de gebruikers om services op te vragen en het resultaat van de server weer te geven. De architectuur bevordert het ontwerpen van systemen die modulair, flexibel en uitbreidbaar zijn door eventuele data, resources of computeroperaties te integreren, te organiseren en aan te bieden als een service [46]. Figuur ?? geeft de client-server architectuur weer.



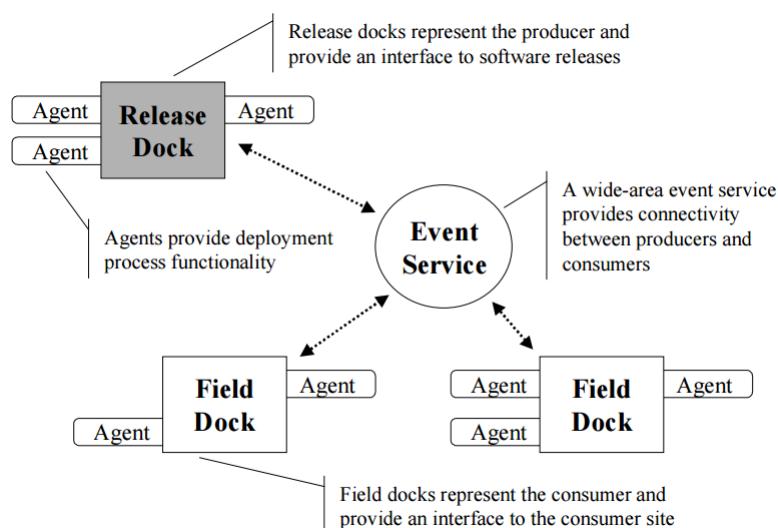
Figuur 2.8: Client-server architectuur

Een voordeel aan de client-server architectuur is dat clients zich niet moeten bezig houden met het verwerken van data. Clients kunnen zich focussen op het opvragen van informatie van de gebruikers en de server en clients kunnen zich focussen op het weergeven van deze data [56]. Doordat de architectuur bestaat uit een server die verscheidene requests van clients afhandelt, is deze server ook de zwakke schakel in de architectuur. Bij het uitvallen van deze component, is het niet mogelijk om een netwerk waar slechts één server in aanwezig is, te bevorderen dan ook niet de schaalbaarheid. Hiernaast wordt de communicatie tussen de client en de server geïnitieerd door de client die een service van de server wilt gebruiken. Aangezien het doel het verspreiden van software vanuit de server is, moet het mogelijk zijn om vanuit de server updates te verspreiden.

Mesbah en Van Deursen [57] legt uit dat het client-geïnitieerde afhaalmodel inefficiënt is voor real-time event notificaties. Een oplossing voor dit probleem is het hanteren van een push-gebaseerd model waarbij de server aanpassingen uitzendt naar alle clients.

2.6.2 Software dock architectuur

Hall, Heimbigner en Wolf [1] bespreken een interessante architectuur die gebruikt kan worden voor het verspreiden van software. Het Software Dock research project creëerde een raamwerk om de samenwerking tussen software producenten en gebruikers te verbeteren. In Figuur 2.8 wordt de ontworpen architectuur voorgesteld. Er worden twee verschillende componenten gedefinieerd waarmee de producenten en gebruikers voorgesteld worden. In de architectuur worden de verschillende producenten voorgesteld aan de hand van een release dock en worden de gebruikers voorgesteld als een field dock. Aan deze docks worden verschillende agenten gekoppeld. Elke agent hoort typisch bij één stap uit de software levenscyclus die besproken werd in Sectie 2.2. Naast de verschillende docks wordt ook een wide-area eventsysteem gedefinieerd. Met dit systeem wordt de communicatie tussen de docks geregeld. Hall, Heimbigner, Van Der Hoek e.a. [31] bespreken in detail hoe de Software Dock architectuur gebruikt kan worden voor het verspreiden van software.



Figuur 2.9: Software Dock Architectuur [1]

De release dock is een softwarecomponent die zich bevindt bij de softwareproducent. De release dock biedt een release-repository aan waaruit de gebruikers de applicaties selecteren voor deployment. In de release dock wordt ieder release semantisch beschreven aan de hand van een Deployable Software Description file. Dit bestand bevat onder andere:

- **Assertion:** deze worden gebruikt om beperkingen die aan de gebruikerskant waar moeten te beschrijven. Als niet aan deze beperkingen voldaan wordt, zal het deploymentproces falen.
- **Afhankelijkheden:** deze worden gebruikt om beperkingen van de release te beschrijven. Als deze niet voldaan zijn aan de gebruikerskant dan kunnen nog oplossingen gevonden

worden. Bijvoorbeeld door additionele software te installeren kan aan de beperking voldaan worden.

- **Configuratie:** deze zorgt voor een beschrijving van de software die wordt uitgegeven. Hierbij horen bijvoorbeeld varianten en aangepaste versies.

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. Agenten kunnen zich inschrijven voor deze events en weten zo wanneer bepaalde handelingen uitgevoerd moeten worden [1].

De field dock dient als een interface naar de gebruikerskant toe. Deze interface biedt informatie over de resources en configuratie van het gebruikerssysteem. Op basis van deze informatie wordt een context opgebouwd waarin de releases van de resource dock worden gedeployed. De agenten die horen bij een release, docken zichzelf in de field dock en kunnen aan de hand van deze interface het gebruikerssysteem 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].

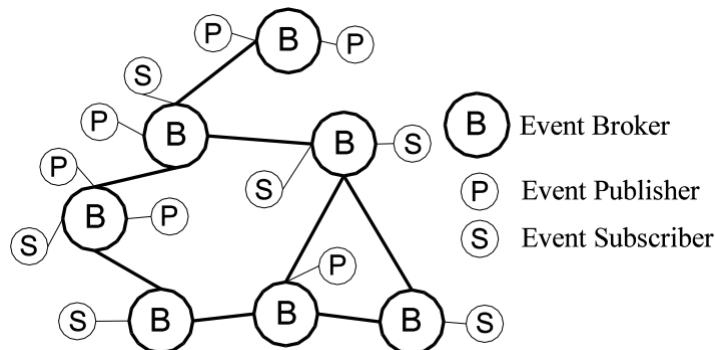
Communicatie

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 [32] en Carzaniga, Rosenblum en Wolf [33] is het mogelijk om een besprekking te geven over hoe de event service geïmplementeerd kan worden.

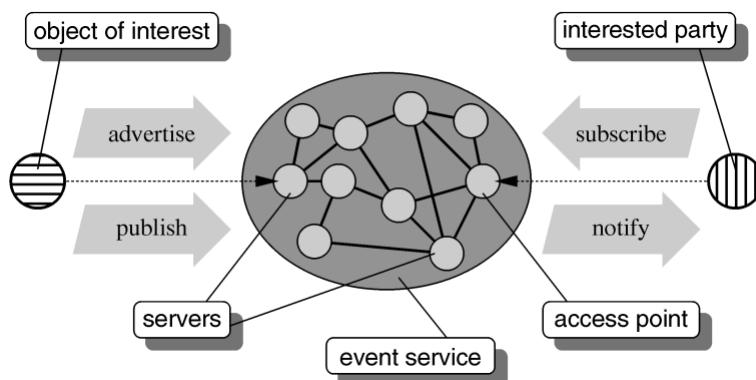
De meeste middleware systemen zijn invocation-based systemen waarbij het request/response paradigma gebruikt wordt om de communicatie tussen client en server te onderhouden. Een client verzoekt een service van de server, waar de server vervolgens op antwoordt. Het toepassen van deze strategie is doenbaar 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 2.9. 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 [32].

Een naïeve aanpak, volgens Carzaniga, Rosenblum en Wolf [33], 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 2.9 waar verschillende "Brokers" aanwezig zijn. Het is belangrijk om stil te staan bij enkele ontwerp beslissingen zodanig dat de service die in Figuur 2.10 zichtbaar is, implementeerbaar is.

Naast de architectuur is het belangrijk om te weten wat wordt verzonden en op welke manier. Carzaniga, Rosenblum en Wolf [33] halen een structuur aan waarin een event beschreven wordt



Figuur 2.10: Broker netwerk [32]



Figuur 2.11: Publish/Subscribe service [33]

als een set van attributen. Ieder attribuut bestaat uit een type, naam en waarde. De naam van een 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 2.11

| | |
|--------|---------------------------------|
| 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 2.12: Publish/Subscribe Event [33]

2.7 Technologieën voor software packaging

In de vorige secties werden oplossingen gezocht voor het verspreiden van software naar verschillende gebruikers. Hierbij werd vooral rekening gehouden met de wijze waarop de software verzonden wordt. Er werd nagegaan of een oplossing zorgde voor een schaalbare architectuur die mogelijkheden biedt naar de toekomst toe. In wat volgt wordt niet meer gekeken naar hoe de

software bij de gebruiker raakt maar wat er bij de gebruiker raakt.

Implementaties in verschillende programmeertalen, verschillende datarepresentatieformaten of incompatibele runtime-environments kunnen aan de basis liggen van een lastige integratie van computerprogramma's. Door gebruik te maken van additionele software, wordt het mogelijk om het gat tussen de verschillen te overbruggen [34]. Het Python testraamwerk is een verzameling van verschillende drivers en bibliotheken elk met een eigen implementatie. Door additionele software toe te voegen is het mogelijk dat verschillende software "pakketten" op een gelijkaardige manier behandeld worden. In wat volgt zullen verschillende technologieën en tools besproken worden die ervoor zorgen dat verschillende software "pakketten" een geheel vormen. Dit geheel kan vervolgens verzonden worden naar de gebruikers waar de software geïnstalleerd moet worden.

Om de verschillende technologieën te vergelijken werd een algemeen testscenario uitgedacht. Er moet een geheel 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

Windows installer XML Toolset is een set van build tools waarmee Windows Installer packages gemaakt worden met behulp van XML broncode. De toolset is geschreven in C# en heeft het .Net framework nodig om te kunnen functioneren. Broncode 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 [35].

Aan deze technologie zijn verschillende voor- en nadelen verbonden. De WiX toolset is diep geïntegreerd met Windows waardoor er tijdens de installatie van software gebruik gemaakt kan worden van verscheidene functionaliteiten die bij Windows horen. De WiX toolset maakt installers die uitsluitend bedoeld zijn voor de Windows installation engine. Hierdoor worden verscheidene functionaliteiten eenvoudig te gebruiken, zoals het maken van uitzonderingen in de Windows Firewall. Hiernaast kan tijdens het installatieproces rekening gehouden worden met externe installers. Op deze manier is het mogelijk om verscheidene software pakketten te combineren tot één geheel. Een fragment van de WiX toolset code is terug te vinden in Listing B.1. Door gebruik te maken van de Windows installation engine is het niet mogelijk om de executable te gebruiken in Linux omgevingen. Dit kan eventueel omzeild worden door het gebruik te maken van software zoals Wine [36]. Als geen alternatieven aanwezig zijn, dan is deze strategie eventueel het overwegen waard. WiX maakt gebruik van XML broncode om verschillende elementen te definiëren. Liefke en Suciu [37] geeft al aan dat XML niet een van de meest efficiënte dataformaten is, maar het verhoogt 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 het nodige om verschillende installatietaken 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-

dige manier aan een installer toegevoegd worden voor een verhoogde functionaliteit [38].

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). De scripting taal is intuïtiever te gebruiken in vergelijking met de XML bestanden van WiX Toolset en is dan ook een voordeel bij deze technologie. Dankzij de 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 executable en de opmerking gegeven bij de WiX Toolset is hier ook van toepassing en is dan ook een nadeel van deze technologie. Het feit dat NSIS bedoeld 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. Een grote hoeveelheid aan pakketten leidt tot wanorde. NSIS biedt 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 & apt-get

Volgens [39] is Chocolatey een package manager 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.

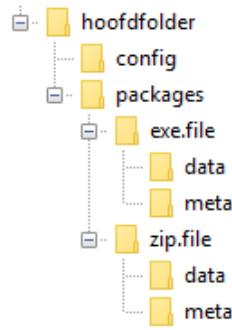
Het grootste voordeel dat bekomen wordt bij het gebruiken van een package manager is het al bestaan van een deployment infrastructuur. Na het installeren van Chocolatey op de client kunnen alle nodige packages voor het framework geïnstalleerd worden. Hiernaast kunnen scripts gekoppeld worden aan iedere 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. Dit is echter eenvoudig op te lossen door het toevoegen van een grafische user interface. 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 geleid. Hierbij kunnen scripts gebruikt worden om het proces te vereenvoudigen [40]. Door een folder structuur op te bouwen zoals Figuur 2.12 kan het Qt installer framework een installer creëren die vervolgens op het doelsysteem uitgevoerd kan worden.

De datafolder van een pakket bevat alle nodige databestanden die moeten uitgevoerd worden (zoals een executable of een zip archief). De metafolder bevat een beschrijving van het pakket (zie voorbeeld in Listing B.5). Hiernaast is ook een installatiescript aanwezig dat uitgevoerd wordt tijdens de installatie van het pakket zelf (zie Listing B.3). Met de installer informatie uit de config folder (zie Listing B.4) weet het Qt installer framework voldoende om een executable te produceren die alle pakketten installeert aan de hand van de opgegeven installatiescripts. Op deze wijze is het mogelijk om verschillende pakketten te combineren tot één geheel en ondertussen toch ieder pakket apart te behandelen.

Aan deze technologie zijn verschillende plus- en minpunten verbonden. Het grootste voordeel van het Qt Installer framework is het cross-platform zijn. Hierdoor is het mogelijk om installers te maken

**Figuur 2.13:** Folder structuur voor installer

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

Tabel 2.1: Voor- en nadelen van de verschillende technologieën

voor zowel Windows als Linux. Een nadeel dat hieraan verbonden is, is dat een Linux installer enkel kan gemaakt worden 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.

2.7.1 Conclusie

Tabel 2.1 geeft een overzicht van de besproken technologieën en de voor- en nadelen die aan iedere technologie gekoppeld zijn. Hieruit wordt snel duidelijk dat geen enkele technologie enkele pluspunten bevat. Naar het ontwerp van de applicatie toe, moet er rekening gehouden worden met alle voor- en nadelen die aan de technologieën gekoppeld zijn. De beste technologie wordt gekozen of er wordt zelf software voor het combineren van pakketten ontworpen die gebaseerd is op deze technologieën.

Hoofdstuk 3

Analyse en ontwerp

3.1 Inleiding

Uit de probleemstelling werd al snel duidelijk dat het probleem omtrent het installeren en updaten van programma's complexer is dan op het eerste zicht lijkt. Bestaande systemen bieden geen volledige oplossing op de verscheidene problemen van Televic. In dit hoofdstuk zal het probleem verder geanalyseerd worden. Aan de hand van deze bevindingen gaat een architectuur ontworpen of geselecteerd worden voor een prototype. Het doel van het prototype is het illustreren van het volledige installatieproces startende van het toevoegen van een testtoren aan het systeem tot en met het uitbrengen en installeren van een nieuwe versie van een applicatie.

3.2 Analyse

Televic heeft 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 veel tijd in beslag neemt en foutgevoelig is. Het is belangrijk om rekening te houden met deze fouten en een "rampenplan" te voorzien. Hiernaast groeit het aantal gebruikers van het testraamwerk continu evenals het aantal drivers, bibliotheken en programma's die verspreid moeten worden. Verder dient elk nieuw toestel op het raamwerk ondersteund te worden waardoor er jaarlijks vele releases van het raamwerk verspreid moeten worden. Het doel is om een systeem te ontwikkelen dat Televic kan bijstaan bij het installatieproces en het verspreidingsproces maar ook om een systeem te ontwikkelen dat kan blijven gebruikt worden in de toekomst. Schaalbaarheid en flexibiliteit zijn hierbij zeer belangrijk.

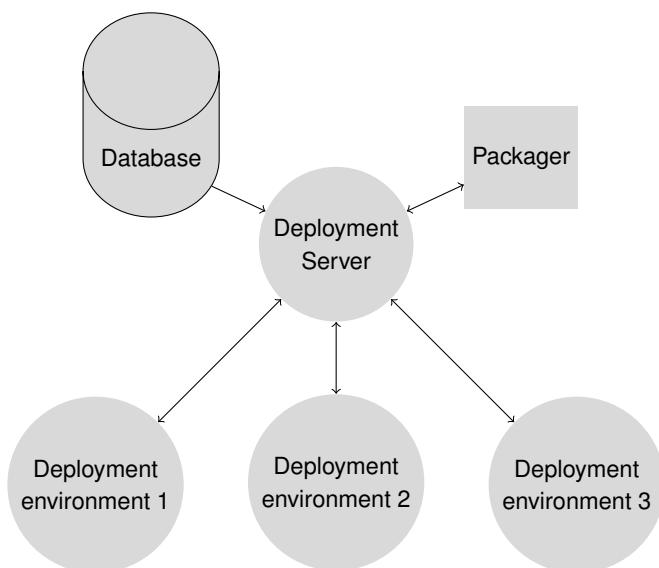
De probleemanalyse onthulde dus dat het probleem onder te verdelen is in verschillende deelproblemen: wat wordt verzonden en hoe wordt dit verzonden. Het testraamwerk bestaat uit verschillende componenten, hieronder vallen de drivers en bibliotheken. Elke component heeft een aparte installatiewijze en sommige componenten moeten 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 is afhankelijk van testtoren waarop het testraamwerk op geïnstalleerd word. Door gebruik te maken van additionele software worden verschillen in implementaties, door bijvoorbeeld verschillende programmeertalen, opgevangen. Een deel van de

applicatie zal dus bestaan uit deze additionele software die instaat voor het inpakken van de componenten: de *packager*. Hiervoor kan beroep gedaan worden op verscheidene technologieën, structuren en architecturen die besproken werden in Sectie 2.7. Door gebruik te maken van één van deze technologieën is geweten wat er wordt verzonden.

Naast de wat moet er ook geweten zijn hoe de software bij de gebruikers moet geraken. Door dit proces te automatiseren, is het mogelijk om waardevolle informatie te verzamelen. Met deze informatie kunnen rapporten gegenereerd worden over het deploymentproces. In Secties 2.5 - 2.3 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 2.2 werd besproken welke problemen kunnen optreden tijdens het installatieproces. Deze problemen moeten opgevangen worden om een schaalbare oplossing te bedenken. Om dit op te vangen, kan er gebruik gemaakt worden van één (of meerdere) strategieën die besproken werden in Sectie 2.4. Dit onderdeel van de applicatie zal vooral aanwezig aan de client-side aangezien dat de plaats is waar het testraamwerk aanwezig zal zijn. Er 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 bestaat de deployment server die instaat is om de installers die de packager aflevert, te verspreiden naar de verschillende gebruikers. Aan de client-side zal de deployment environment aanwezig zijn waardoor installatie-complicaties verminderd 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.



Figuur 3.1: Overzichtsdiagram van de algemene structuur

3.3 Architectuur

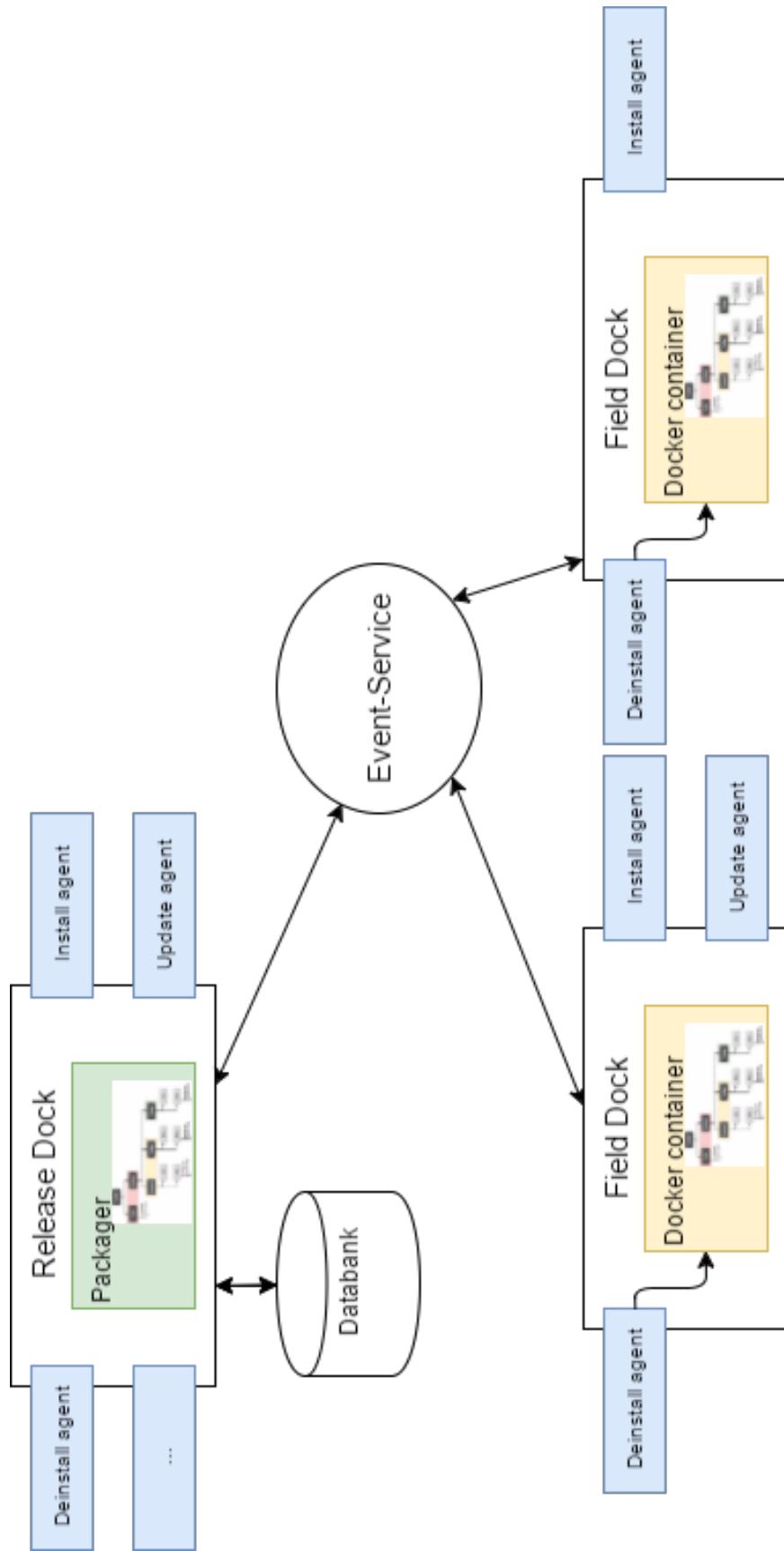
De huidige architecturen en technologieën voldoen niet aan de verwachtingen en bieden geen sluitende oplossing voor alle problemen van Televic. De meeste technologieën bieden een deeloplossing aan. Zo biedt de LJSFi architectuur van ATLAS een mooie oplossing aan voor het verspreiden van verschillende softwarepakketten naar een groot aantal computers maar bij deze architectuur moet gebruik gemaakt worden van de Grid middleware. De Redhat package manager biedt een volledig beschikbare infrastructuur aan maar afhankelijkheden tussen pakketten zijn dan weer minder goed gedefinieerd. De client-server architectuur is een goede architectuur voor het verspreiden van de software maar biedt geen mogelijkheden aan voor de installatie van softwarepakketten. Om toch een oplossing te bieden voor de problemen wordt zelf een architectuur ontworpen die bestaat uit een combinatie van besproken technologieën en architecturen.

Zoals in de analyse al werd aangehaald, zal de architectuur bestaan uit de volgende componenten:

- De deployment server
- De packager
- De database
- De deployment environment

Figuur 3.2 geeft de volledig ontworpen architectuur weer. De basis van de architectuur is overgenomen van de software dock architectuur uit Sectie 2.6.1. Door gebruik te maken deze architectuur is het mogelijk om de deploymentserver samen met de packager te combineren tot een release dock en de deployment client te implementeren als een field dock. Het gebruik van software dock zorgt voor een schaalbare architectuur waarbij het aantal gebruikers en software producenten mag toenemen. Zowel het release dock als het field dock moeten niet bijhouden welke docks aanwezig zijn in het netwerk aangezien dit de taak is van de broker. Verder zorgen de agenten voor een verhoogde flexibiliteit in alle stappen van de deployment levenscyclus. Technologieën zoals ElectricFlow en ATLAS gebruiken ook agenten voor het installeren van software. Bij deze technologieën zijn de agenten afgestemd op de software die moet geïnstalleerd worden en niet op een stap uit de deployment levenscyclus.

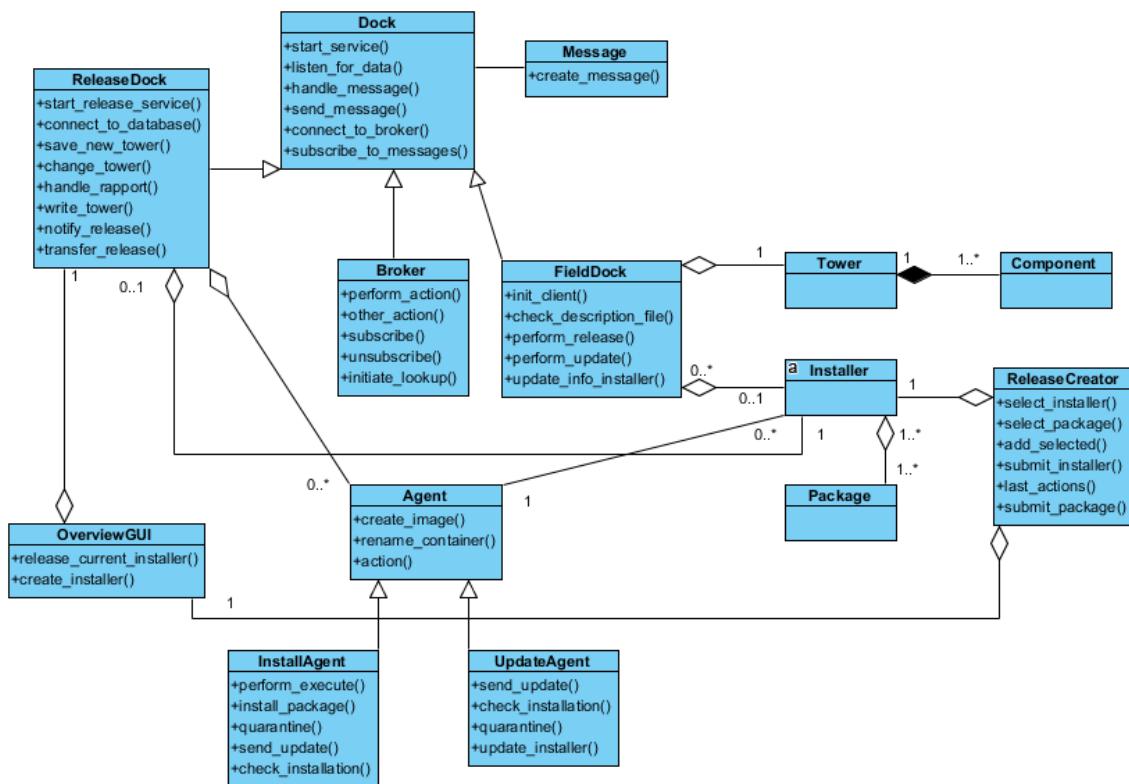
Om de functionaliteiten van de packager te implementeren wordt een systeem ontworpen dat gebaseerd is op het Qt installer framework. Andere software packaging oplossingen (zoals NSIS en WiX Toolset) bieden ook een oplossing voor het combineren van softwarepakketten. Er wordt toch voor het Qt installer framework als basis gekozen omdat softwarepakketten beter van elkaar gescheiden worden. Dit geeft een beter overzicht van wat aanwezig is in een installer en welke installatiescripts uit de meta folder invloed heeft op de data. Door gebruik te maken van een gelijkaardige structuur wordt een packager ontworpen die kan omgaan met verschillende types van softwarepakketten. Het Qt installer framework zelf volstaat niet aangezien het niet mogelijk is om op één besturingssysteem een installer te creëren die werkt op zowel Linux als Windows. Aangezien de packager zelf ontworpen wordt, is het mogelijk om eerder welke programmeertaal te kiezen. Er wordt dus best een taal gekozen die volledig besturingssysteem-onafhankelijk is. Door deze packager te combineren met de software dock architectuur is het mogelijk om naast het personaliseren van iedere stap in het deploymentproces ook de behandeling van ieder softwarepakket apart te personaliseren. De packager wordt opgenomen in de code van het release dock en wordt door gebruikt om installers te produceren.



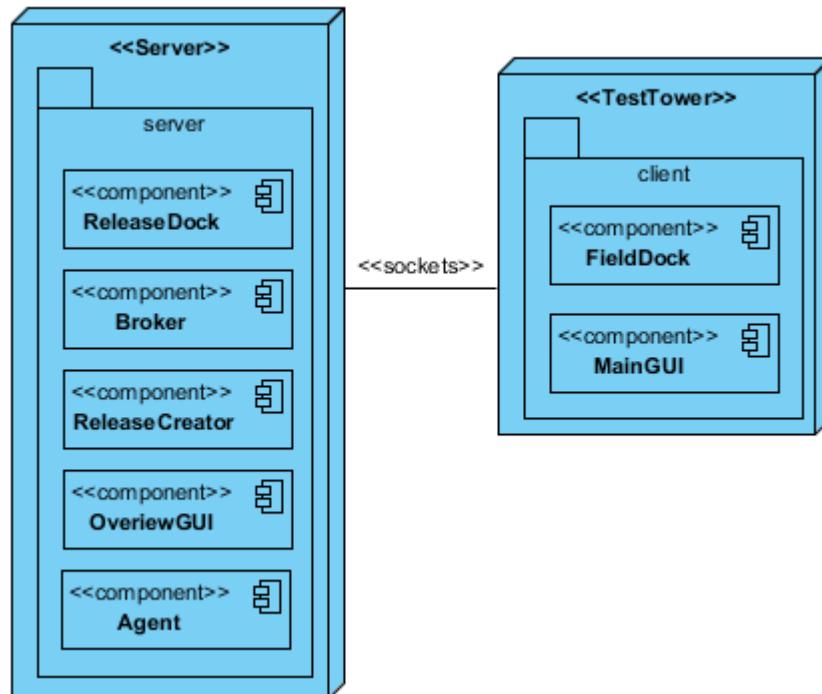
Figuur 3.2: Architectuur van het prototype

Een laatste element uit de architectuur is de deployment omgeving die zal bestaan uit het field dock van de software dock architectuur. Het doel van de deployment omgeving is het creëren van een veilige omgeving waarin de installer van het release dock in geïnstalleerd en geüpdatet kan worden. Zoals reeds werd aangegeven is het installatieproces en updateproces foutgevoelig en moet hiervoor een oplossing gevonden worden. Dit wordt dan ook opgelost door Docker containers te gebruiken. Er wordt niet gekozen voor één van de rollback strategieën aangezien deze ofwel een beperking opleggen (zoals het gebruik moeten maken van het transactieprogrammeermodel) ofwel veroorzaken deze extra overhead (zoals bij het gebruiken van checkpoints). Virtuele machines kunnen eventueel ook gebruikt worden als oplossing. Bij het optreden van een fout kan de VM verwijderd worden zodanig dat er geen software achterblijft die later voor problemen kan zorgen. Het nadeel van virtuele machines is dat acties die uitgevoerd worden op VMs lang duren. Docker containers hebben dit probleem niet. Er wordt een veilige omgeving gecreëerd die snel opgestart en afgebroken kan worden.

Na het ontwerpen van de algemene architectuur wordt een klassendiagram (Figuur 3.3) en deployment diagram (Figuur 3.4) ontworpen die gebruikt worden voor de implementatie. Uit het deployment diagram wordt het snel duidelijk dat de server de meeste componenten bevat. In wat volgt, wordt dieper ingegaan op de vier componenten waaruit de architectuur bestaat aan de hand van het klassendiagram en het deployment diagram.



Figuur 3.3: Klassendiagram van de applicatie



Figuur 3.4: Deployment diagram van de applicatie

3.3.1 Deployment server

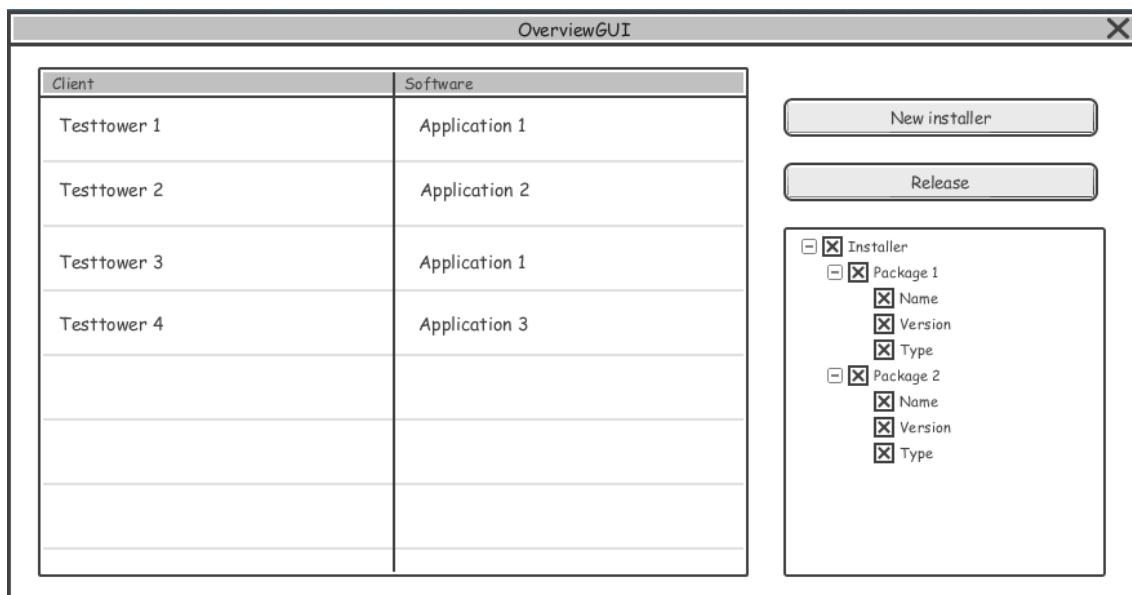
Het centrale systeem in de architectuur is de deployment server. Zoals reeds uitgelegd zal dit onderdeel instaan voor het verspreiden van de verschillende installers en functioneren als een verzamelcentrum voor alle informatie. De software dock architectuur bestaat uit 4 grote componenten, namelijk het release dock, field dock, event service en de agenten (zie figuur 3.2). Alle componenten behalve het field dock zijn aanwezig aan de serverzijde.

Release Dock

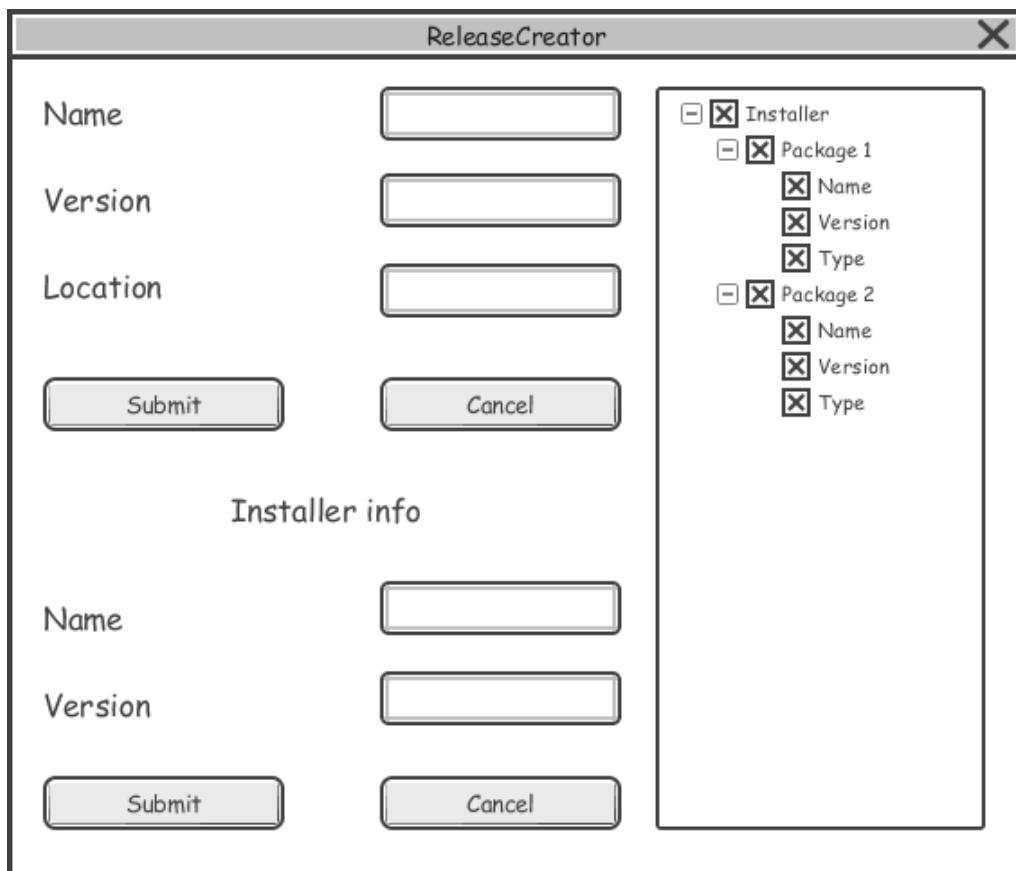
Uit het klassendiagram in Figuur 3.3 wordt al snel duidelijk dat de ReleaseDock klasse geassocieerd kan worden met de OverviewGUI klasse en ReleaseCreator klasse. Dit is ook duidelijk zichtbaar in het deployment diagram waarin weergegeven wordt dat deze drie aanwezig zijn op één server. De belangrijkste functionaliteiten van het release dock bestaat uit het wegschrijven van informatie naar de databank en het afhandelen van de verschillende release-aanvragen. De weg te schrijven informatie bestaat vooral uit gegenereerde rapporten van verscheidene testen maar bijvoorbeeld ook nieuwe gebruikers die moeten toegevoegd worden aan de databank.

Om de verscheidene grafische user interfaces te ontwerpen en implementeren, wordt er gebruik gemaakt van het model-view-controller ontwerppatroon. Een view wordt eerst geschetst met behulp van een wireframe en vervolgens geïmplementeerd met behulp van wxPython [41]. Vervolgens wordt een controller gedefinieerd en geïmplementeerd die data uit het model haalt om deze weer te geven in de view.

De OverviewGUI klasse wordt ontworpen als managersinterfacecontroller. De view die hiervoor ontworpen werd, is terug te vinden in Figuur 3.5. Het doel is om de informatie die aanwezig is in de database weer te geven. Op deze wijze is het mogelijk om een overzicht te geven van alle clients die verbonden zijn met het systeem maar ook welke software zij gebruiken. Door dit overzicht krijgt Televic een algemeen beeld van hoe het verspreiden van hun software verloopt.



Figuur 3.5: Ontwerp van de managersinterface



Figuur 3.6: Ontwerp van de release creatie interface

Het release dock is indirect verbonden met de ReleaseCreator. Deze is de controller van de view die instaat voor het creëren van een nieuwe release indien dit nodig is. Figuur 3.6 geeft het ontwerp van de view weer. Zoals reeds werd toegelicht, zal een installer bestaan uit verschillende softwarepakketten. Hoe een installer exact wordt opgebouwd en welke bestanden worden aangemaakt, wordt verderop besproken. Na het creëren van de installer wordt deze doorgegeven aan de ReleaseDock. Vanaf dat moment is het mogelijk om vanuit de OverviewGUI de installer op te geven voor release.

Event Service

Naast de verschillende docks bevat 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 de info uit Sectie 2.6.1. Er worden evenwel enkele kleine aanpassingen gemaakt aan het formaat van de publish/subscribe eventen. Er wordt een eigen formaat ontworpen zodat alle nodige elementen zeker in een bericht vervat zit. Hiernaast wordt de Broker klasse zo ontworpen dat er slechts één broker aanwezig kan zijn. Dit zorgt voor een vereenvoudiging van de implementatie aangezien geen protocollen voorzien moeten worden om tussen verschillende brokers te kunnen communiceren. Het is wel belangrijk om hierbij te vermelden dat dit een impact heeft op de schaalbaarheid van de applicatie. Van zodra een te groot aantal docks aanwezig is, zal de broker een bottleneck vormen voor de applicatie. Dit kan echter opgevangen worden door

meerdere brokers samen te laten werken zodat dat er aan load balancing gedaan kan worden. De brokers zullen een netwerk vormen zoals in Figuur 2.9.

Bij 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 zender van het bericht toevoegen of verwijderen uit de de verschillende lijsten. 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 2.11. In Listing 3.1 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 bedoeld 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 zodat 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: Formaat voor een bericht

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

Agenten

Een laatste element die aangemaakt wordt aan de serverzijde zijn de agenten. Deze staan in voor het uitvoeren van allerlei deploymentgerelateerde handelingen. Iedere agent is gekoppeld aan één

stap uit de software levenscyclus die besproken werd in Sectie 2.2. Hiernaast zal aan iedere installer afkomstig van het release dock een subset van alle agenten toegevoegd en naar het field dock verscheept worden. 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 deploymentproces die besproken werd in de ORYA case studie in Sectie 2.3.4. Net zoals bij ORYA wordt ieder deploymentproces beschreven aan de hand van andere deploymentprocessen en basisactiviteiten. Zo zal tijdens de creatie van een installer een agent voorzien worden 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 gezien als zo'n basisactiviteit. In Bijlage A zijn verschillende activiteitendiagrammen 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 2.3.3, 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.

Initialisatie

Het deployment diagram in Figuur 3.4 geeft weer dat aan de serverzijde verschillende componenten aanwezig zijn die samenwerken. Hierbij is het belangrijk om een goede opstartsequentie te voorzien. Figuur 3.7 geeft weer hoe de nodige componenten opgestart worden.

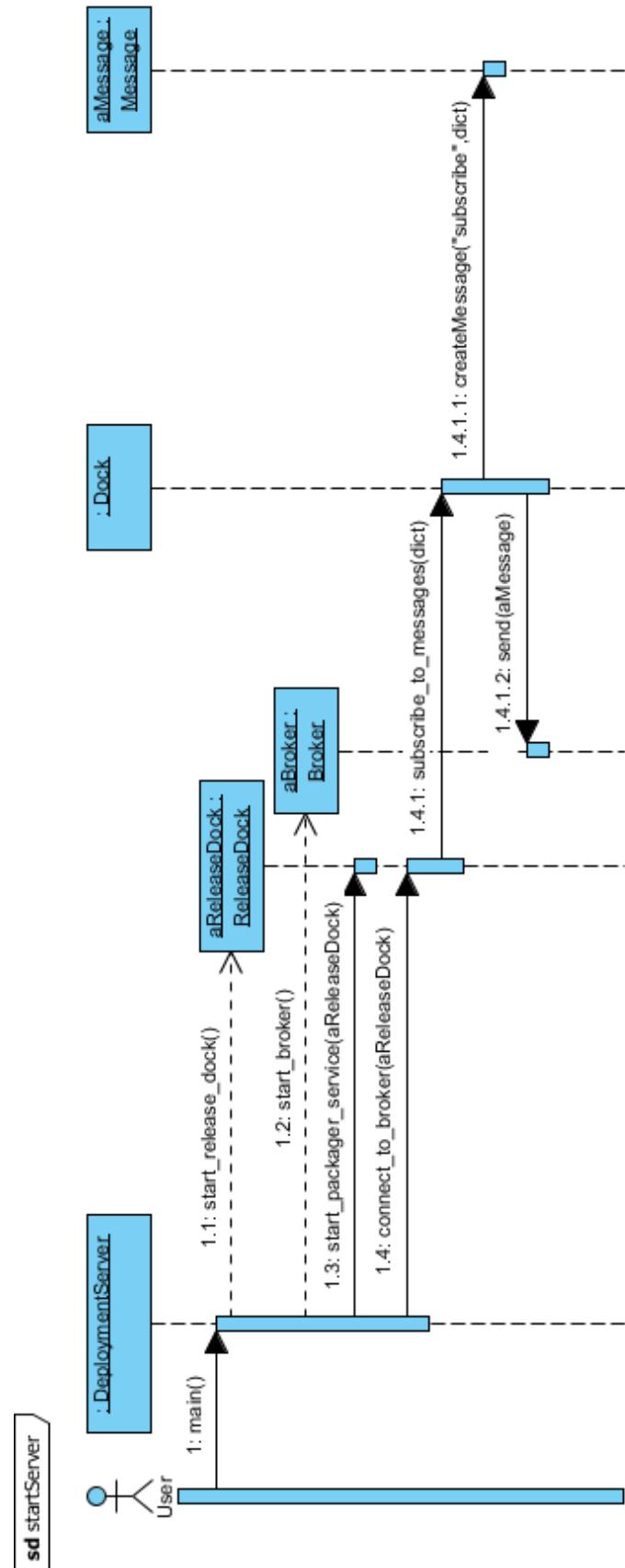
Als eerste wordt een ReleaseDock object aangemaakt, gevolgd door een Broker object. Uit het klassendiagram in Figuur 3.3 blijkt dat beide objecten van hetzelfde object overerven. Dit werd zo ontworpen aangezien beide objecten dezelfde functionaliteiten moeten implementeren (namelijk het verzenden van berichten, het luisteren voor nieuwe data en het afhandelen van binnenkomende berichten). Vervolgens zal het release dock zich gaan inschrijven voor berichten met als type new, change, en rapport. Nadat de nodige services zijn opgestart is het mogelijk om de overviewGUI op te starten.

3.3.2 Packager

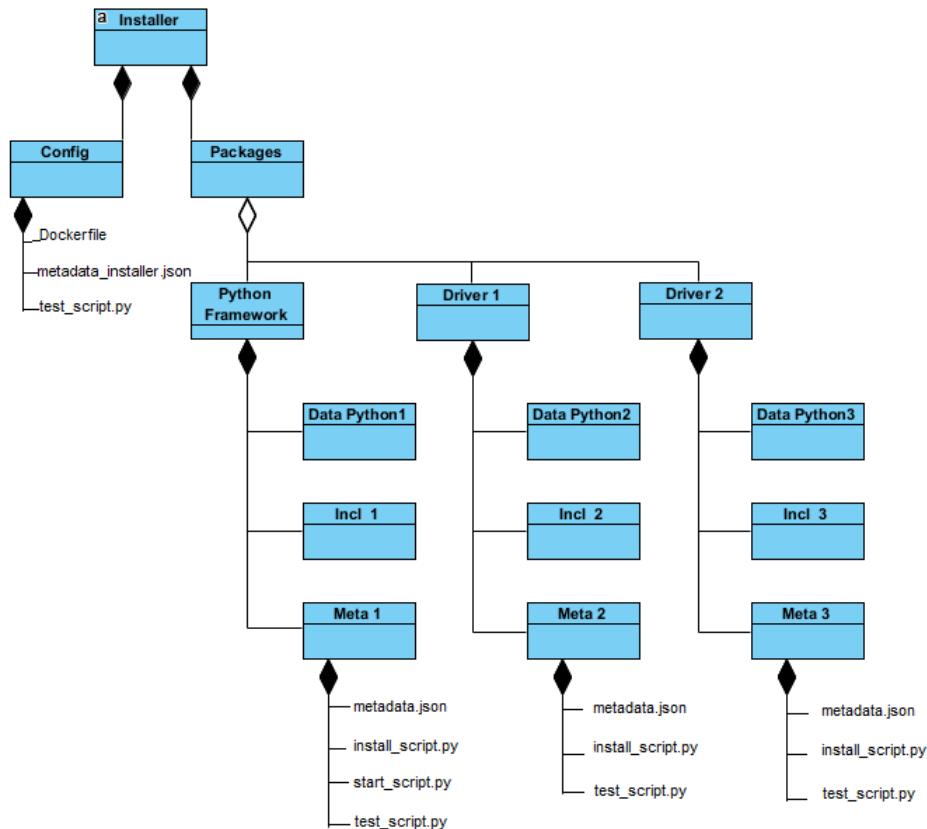
De architectuur van de packager wordt gebaseerd op de architectuur van het Qt installer framework. Er wordt een installer geproduceerd die bestaat uit verschillende pakketten die elk instaan voor het installeren van een software component. Voor iedere installer wordt een aparte folder structuur aangemaakt die zichtbaar is in Figuur 3.8.

Door zelf een packager te produceren, is het mogelijk om iedere stap in het deploymentproces 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. Hiernaast kan ervoor gezorgd worden dat de softwarepakketten beter overweg kunnen met Docker. Zo is er een vlottere integratie van Docker.

In de config folder van de installer worden alle globale scripts en beschrijvingsbestanden bijge-



Figuur 3.7: Sequentie diagram voor het opstarten van een server



Figuur 3.8: Structuur van een installer bestaande uit drie pakketten

houden. Verder bevat de installer subtrees voor ieder pakket. Een pakket bestaat vervolgens uit een data-, include- en metafolder. De datafolder wordt gebruikt om de effectieve driver/bibliotheek in op te slaan. Hiernaast is een includefolder aanwezig waarin verschillende afzonderlijke scripts toegevoegd kunnen worden. Op deze manier kunnen willekeurige scripts (bijvoorbeeld een script die de firewall instellingen aanpast) gemakkelijk toegevoegd worden aan een pakket. Als laatste bevat de metafolder alle meta-data horende het pakket.

Tijdens het creëren van de installer moet één van de pakketten aangeduid worden als een applicatie pakket. Bij het toevoegen van de verschillende bestanden zal dit pakket een start_script meekrijgen. Dit is het script dat wordt uitgevoerd iedere keer als het framework wordt opgestart. Hierna worden alle pakketten overlopen. Als het pakket nieuw is, worden de nodige folders en bestanden aangemaakt anders worden de oude bestanden uit de meta folder gekopieerd. Nadat de folder structuur aangemaakt is, is het mogelijk om de verschillende scripts aan te passen. Net als bij het QT Installer Framework is het de bedoeling om te scripts te personaliseren naargelang het pakket. Op deze wijze heeft ieder pakket een uniek installatie- en testproces. De installer is hierna klaar om released te worden.

Vanuit de Overview is het mogelijk om een installer te releaseen. Figuur 3.9 geeft het sequentie diagram weer dat gevuld wordt om de installer van het release dock naar het field dock te brengen. Het release dock maakt een Message object aan, geeft de installer informatie mee en zet het type van het bericht op release. Vervolgens wordt deze doorgestuurd naar de broker. Deze zal op zijn beurt het bericht inpakken als een notificatie bericht en het nieuwe bericht doorsturen naar

de docks die voor release gesubscribed zijn. Met de informatie uit het bericht is het field dock in staat om een Installer object te maken. Het field dock achterhaalt wie de zender van het bericht en contacteert dan direct de zender. Dit is het enige moment dat directe communicatie tussen de field en release docks mogelijk is. Nadat een connectie tussen de twee werd opgestart, wordt de gezippe installer samen met de nodige agenten verzonden naar het field. De lijst van agenten wordt overlopen en het install-agentobject zal zijn functie uitvoeren.

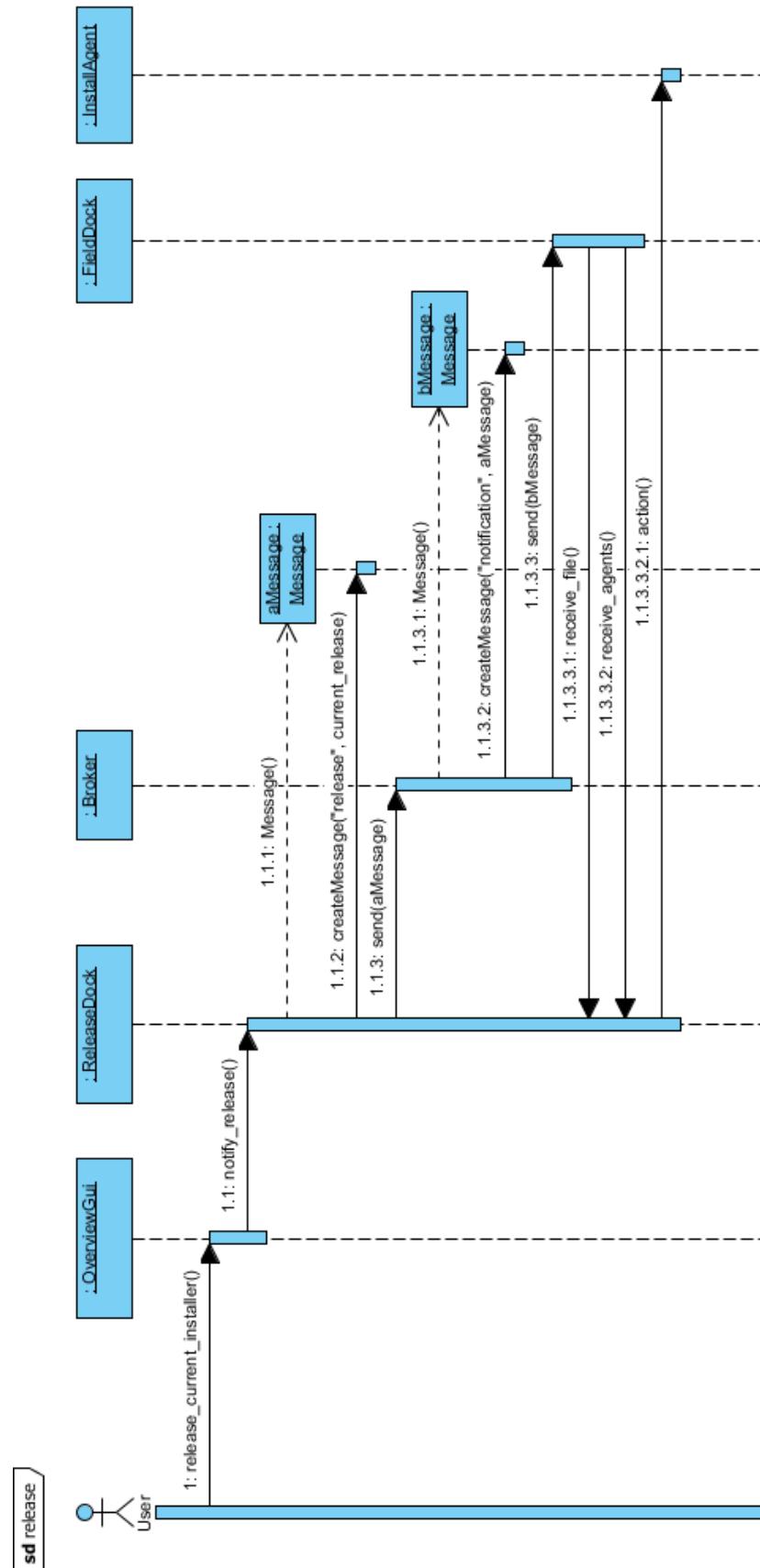
3.3.3 Databank ontwerp

Eén van de problemen is de continue toename aan pakketten waar het framework gebruik 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 als de gebruikers. In overleg met Televic werd ervoor gekozen om MySQL te gebruiken als databasesysteem. Het ontwerp van de databank is terug te vinden in Figuur 3.10.

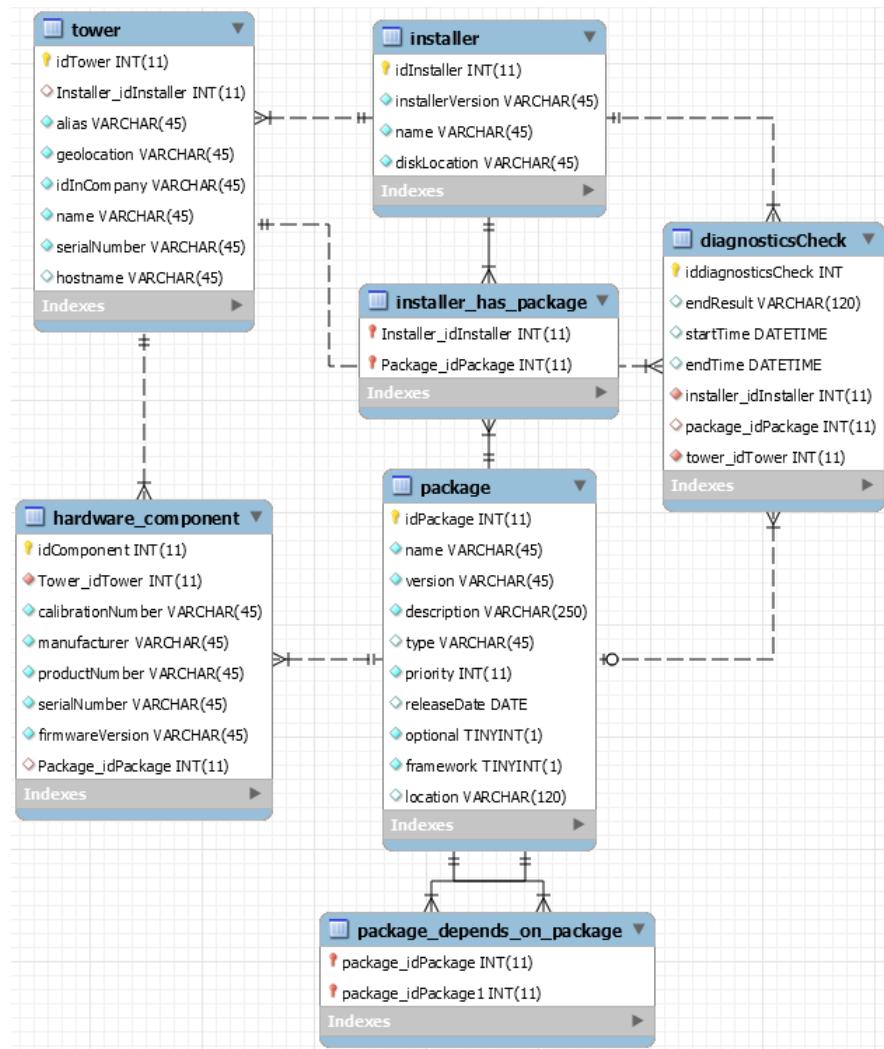
De tabellen tower en hardware_component dienen om iedere gebruiker, typisch een testtoren of een laptop, te beschrijven. Iedere toren heeft een ID, naam en serienummer. De combinatie van deze drie waarden is uniek binnen het bedrijf en de combinatie kan gebruikt worden als identificatie binnen het systeem 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 hardwarecomponenten, 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 hardwarecomponent 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. Hardware componenten te koppelen aan torens
3. Firmware versies te koppelen aan hardware componenten

Naast informatie over de gebruikers wordt ook informatie over de verschillende installers en pakketten bijgehouden. Een installer bestaat uit een combinatie van verschillende softwarepakketten. Bij deze pakketten moet één pakket aanwezig zijn dat de applicatie bevat. Hiernaast zijn verschillende andere pakketten aanwezig voor drivers. In Figuur 3.8 is een voorbeeld zichtbaar van een installer die bestaat uit drie verschillende pakketten. Eén pakket wordt gebruikt door de applicatie en de twee anderen voor drivers die nodig zijn om de applicatie correct te laten functioneren. Iedere toren wordt gekoppeld aan één installer en zo aan één applicatie. 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 testraamwerkpakket en een pakket waarmee Python geïnstalleerd wordt. Het testraamwerk is afhankelijk van Python om



Figuur 3.9: Sequentie diagram voor het releasen van een installer



Figuur 3.10: Ontwerp van de databank

correct te functioneren. Zo worden de verschillende software-software afhankelijkheden bijgehouden. Voordat een installer gemaakt wordt, die een testraamwerkpakket bevat, kan gecontroleerd worden dat ook het Python-installatiepakket aanwezig is.

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 functioneert en op het einde van het installatieproces kan 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 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 testoren te vermijden.

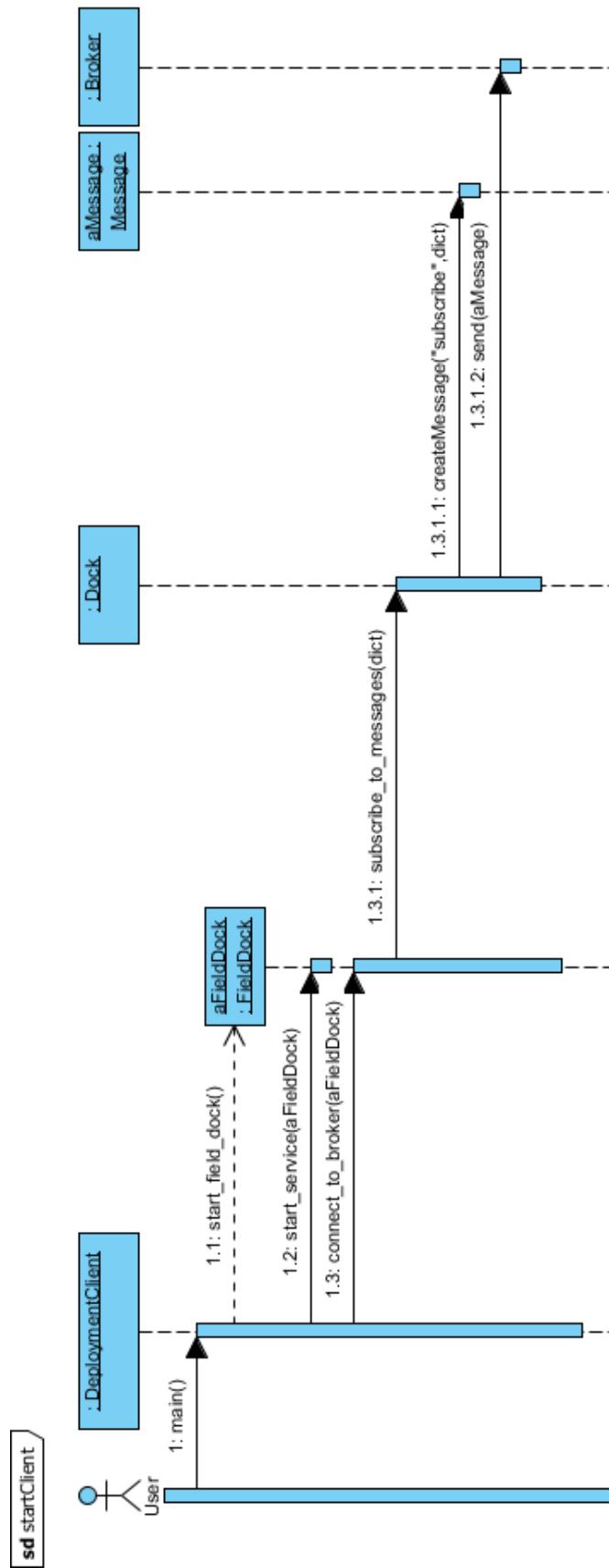
3.3.4 Deployment environment

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 zodat de software geïnstalleerd wordt. Aan dit proces zijn de volgende problemen verbonden die grondig besproken werden in Sectie 2.2.

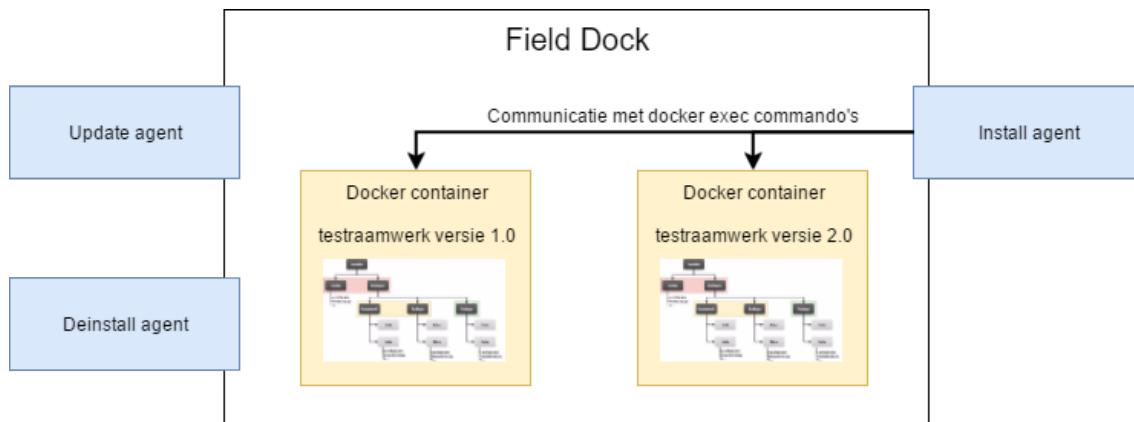
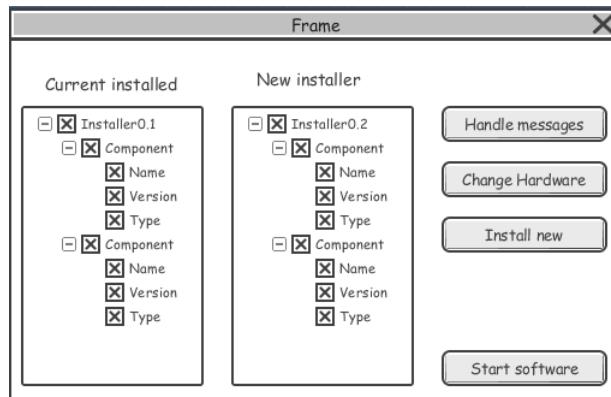
Het opstarten van een client gebeurt op een gelijkaardige manier als het opstarten van een server. Figuur 3.11 bevat het sequentie diagram voor het opstarten van een client. Hiervoor wordt eerst een FieldDock aangemaakt en deze zal zich vervolgens bij de broker inschrijven voor berichten van het type release en update. Bij het eerst gebruik moet het systeem eerst beschreven worden zodanig dat deze beschrijving toegevoegd kan worden aan de database. Na het registeren van de client kan de gebruiker gebruik maken van de view in Figuur 3.13. Het ontwerp van de view is terug te vinden in Figuur 3.14. Aan deze views is de MainGUI controller gekoppeld. Van hieruit is het mogelijk om nieuwe software die van het release dock te beoordelen en eventueel te installeren.

Om de verschillende deploymentproblemen te vermijden en om ervoor te zorgen dat er geen uitgebreide rollback strategieën nodig zijn, wordt een geïsoleerde omgeving voorzien waarin de software geïnstalleerd wordt. Dit wordt gerealiseerd aan de hand van virtualisatietechnieken, 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 virtualisatietechniek wordt gebruikt, wordt het zeer eenvoudig om problemen tijdens de verschillende processen op te vangen. In Figuur A.1 is het duidelijk dat, door het gebruik van Docker, het rollback proces zeer eenvoudig is. Hierbij is het belangrijk om te weten dat de container met de defecte code niet wordt verwijderd. De defecte code wordt in "quarantaine" geplaatst. Mochten logische fouten aanwezig zijn in een test kan dit leiden tot het afkeuren van de software terwijl de installatie wel correct verlopen is. Door de container in quarantaine te plaatsen is het mogelijk om na het installatieproces de container te ondervragen en te achterhalen wat fout gelopen is. Figuur 3.12 geeft weer op welke manier de agenten met de containers gaan communiceren en geeft ook weer hoe Docker wordt geïntegreerd in het geheel.

Alle componenten samen vormen het geheel die zichtbaar was in Figuur 3.2 op pagina 26. Deze architectuur vormt een goede basis van de implementatie die in het volgende hoofdstuk besproken wordt.



Figuur 3.11: Sequentiële diagram voor het opstarten van een client

**Figuur 3.12:** Structuur van een field dock**Figuur 3.13:** Grafische user interface van de client

Grafische user interface for first use:

- The window title is **Frame**.
- The interface is divided into two main sections:
 - Hardware component** (left):
 - Fields: **Name**, **Firmware Version**, **Serial Number**.
 - Buttons: **Submit**, **Cancel**.
 - Tower** (right):
 - Fields: **Alias**, **Id in Company**, **Serial Number**.
 - Buttons: **Submit**, **Cancel**.
 - A nested tree view under **Tower** shows:
 - Tower** (expanded):
 - Component** (expanded):
 - Name**, **Firmware**, **Serial Number**
 - Component** (expanded):
 - Name**, **Firmware**, **Serial Number**

Figuur 3.14: Grafische user interface bij eerste gebruik

Hoofdstuk 4

Implementatie

4.1 Inleiding

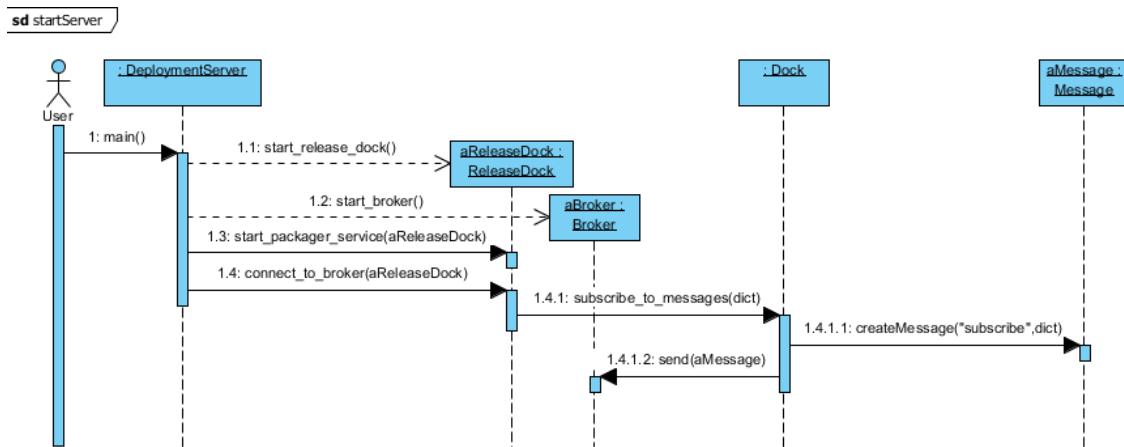
Na de literatuurstudie en de ontwerpfase, moet het ontwerp dat besproken werd in het vorige hoofdstuk geïmplementeerd worden. Als programmeertaal werd Python gekozen. In het bedrijf is een zeker kennis van Python aanwezig waardoor de overname vlotter zal verlopen. Het doel is om een programma te ontwikkelen dat de belangrijkste functies omvat en dat gemakkelijk uitgebreid kan worden. In de komende secties worden de verschillende implementatie beslissingen toegelicht en wordt de geschreven code toegelicht. In wat volgt, zullen de server-side en client-side afzonderlijk besproken worden. De verschillende Python modules worden overlopen en alle methoden met speciale kenmerken worden toegelicht. De volledige code is terug te vinden in de bijgevoegde CD-Rom.

4.2 Server-side

4.2.1 Deployment server

Zoals werd aangegeven in Hoofdstuk 3.3.1 bestaat het opstarten van een server uit het creëren van een ReleaseDock object gevolgd door het creëren van een Broker object. Vervolgens gaat het release dock zich inschrijven voor de nodige berichten (zie Figuur 4.1).

Bij het opstarten van de docks moeten verschillende parameters meegegeven worden. Listing 4.1 geeft weer welke parameters ingevuld moeten worden voor een correcte functionaliteit. De broker interface en port worden gebruikt door de Broker om een socket te openen maar ook door ReleaseDock om te connecteren met de broker. De release_dock interface en port zijn bedoeld om in de ReleaseDock een socket te openen waar de Broker berichten naar kan verzenden. Als laatste moeten de release interface en port gespecificeerd worden. Met deze variabelen wordt een socket geopend die gebruikt wordt door het FieldDock om een installer op te halen. Het ophalen van een installer is het enige moment waarop een FieldDock direct connecteert met een ReleaseDock.



Figuur 4.1: Sequentie diagram voor het opstarten van een server

```

broker_interface = sys.argv[1]
broker_port = 12347
release_dock_interface = sys.argv[2]
release_dock_port = 12345
release_interface = sys.argv[3]
release_port = 12346
  
```

Listing 4.1: Parameters voor server en broker

4.2.2 Dock

Starten van de services Na het creëren van een dock moeten verscheidene acties uitgevoerd worden. Zo zal een ReleaseDock een connectie met een databank opstarten en een FieldDock zal een connectie maken met de Docker client. Uit het klassendiagram in Figuur 3.3 bleek al dat beide klassen methoden overerven van de Dock klasse. Ieder Dock object moet dan ook in staat zijn om binnenkomende berichten correct af te kunnen handelen. Om deze taak uit te voeren, worden twee threads opgestart (zie Listing 4.2).

```

self.open_socket()
thread = Thread(target=self.listen_for_data, args=())
thread.daemon = True
thread.start()

message_thread = Thread(target=self.handle_message, args=())
message_thread.daemon = True
message_thread.start()
  
```

Listing 4.2: Starten van dock services

De listen_for_data methode wacht continue op het toekomen van data op de socket. Van zodra data toekomt, wordt deze in een Queue object gestoken. Zowel het accepteren van een inkomende connectie als het verwijderen van een object uit de queue zijn blocking methodes. Dit wil zeggen dat zolang er geen binnenkomende connecties zijn en zolang er geen objecten in de queue zitten,

zullen deze threads ook niks doen. De threads worden geblockt totdat de juiste acties uitgevoerd worden.

```
def listen_for_data(self):
    while True:
        conn, address = self.message_socket.accept()
        print("DOCK — Connected by \n\t\t" + str(address))
        :
        data = conn.recv(1024)
        print("DOCK — received data: \n\t\t" + data)
        self.message_queue.put(data)

def handle_message(self):
    while True:
        data = self.message_queue.get()
        :
```

Listing 4.3: Ontvangen en afhandelen van data

4.2.3 Release_dock

Kopiëren van bestanden Tijdens het creëren van een installer wordt gecontroleerd of een geselecteerd pakket al bestaat. Mocht dit het geval zijn dan moeten de nodige bestanden uit de originele meta-data folder gekopieerd worden naar de nieuwe locatie. Om dit uit te voeren, wordt er gebruik gemaakt van de `copy_tree` methode uit de `distutils` module. De doellocatie wordt door deze methode overschreven met bestanden uit de oorspronkelijke locatie.

Agenten verzenden Een onderdeel van het deploymentproces bestaat uit het kopiëren van de nodige agenten naar het field dock. Om dit uit te voeren wordt `Dill` [42] gebruikt om de nodige objecten te serialiseren en deserialiseren. Listing 4.4 geeft de code weer die hiervoor gebruikt wordt. De `dill.dumps` methode zorgt ervoor dat het object omgezet wordt naar een string die kan worden verzonden over een socket. Om dit correct uit te voeren wordt eerst de lengte van de string verzonden zodat aan de field dock zijde geweten is hoeveel bytes ontvangen moeten worden. Vervolgens kan het field dock de string terug omzetten naar een object door gebruik te maken van `dill.loads`.

```

def send_agents(self, conn, address):
    ready = conn.recv(1024)
    if str(ready) == "Ready":
        list_agents = dill.dumps(self.agents)
        file_size = len(list_agents)
        conn.send(str(file_size))
        okay = conn.recv(1024)
        if str(okay) == "Received length":
            conn.send(list_agents)
        # send_file(conn, pkl)
        print("RELEASE DOCK — " +
              "Done sending release to " + str(address))
    conn.close()

def receive_agents(s):
    s.send("Ready")
    length = s.recv(1024)
    s.send("Received length")
    file_size = int(str(length))
    data = s.recv(file_size)
    list_agents = dill.loads(data)

```

Listing 4.4: Verzenden en ontvangen van agenten

4.2.4 Agents

Maken van een Docker image De basis van een container bestaat uit een image die opgebouwd wordt aan de hand van een Dockerfile. Om een image te maken aan de hand van een Dockerfile wordt gebruik gemaakt van de docker-py module [43]. Eerst wordt naar de locatie van de Dockerfile gezocht en vervolgens wordt deze meegegeven als parameter. De tag parameter zorgt ervoor dat de image eenvoudiger kan teruggevonden worden en de rm parameter zorgt ervoor dat tussenliggende containers verwijderd worden. Tijdens de implementatie werd altijd gebruik gemaakt van Docker images met Linux als basis. Het is mogelijk om Docker images te creëren met Windows als basis. Dit zorgt weliswaar voor errors. Dit wordt in het volgende hoofdstuk verder besproken.

```

dockerfile = str(self.find("Dockerfile", self.release_zip_location))
self.docker_image = self.client.images.build(path=dockerfile,
                                              tag="fieldimage", rm=True)

```

Listing 4.5: Creatie van een Docker image

Container creatie Na het creëren van de Docker image en het hernoemen van de oude container, moet een nieuwe container aangemaakt worden. Listing 4.6 bevat de code die een container aanmaakt met als naam “fieldcontainer”. Bij het creëren worden de volgende parameters meegegeven:

- **entrypoint:** dit is de executable die wordt uitgevoerd bij het opstarten van de container.

- **tty**: deze parameter zorgt voor de allocatie van een pseudo-tty. Hierdoor is het mogelijk om de stdin en stdout van een container te koppelen aan de terminal waarin het programma uitgevoerd wordt.
- **environment**: deze zorgt voor het instellen van enkele omgevingsvariabelen in de container.

In Docker containers is het niet mogelijk om grafische user interfaces op te starten tenzij een vnc gebruikt wordt. Een andere manier is het doorgeven van een X11 socket en die gebruiken om grafische user interfaces te gebruiken. Om dit toe te kunnen passen, moet een X11 server opgestart worden op de computer waar het field dock op draait. Hierna moet de "DISPLAY" variabele in de container ingesteld worden op het lokale IP adres van de X11 server.

```
ip = socket.gethostname()
socket.gethostbyname(socket.gethostname())
self.client.containers.create(self.docker_image, entrypoint="/bin/bash",
                               tty=True, name="fieldcontainer",
                               environment=[("DISPLAY=" + ip + ":0.0")])
for container in self.client.containers.list(all=True):
    if container.name == "fieldcontainer":
        self.container = container
```

Listing 4.6: Creatie van fieldcontainer

Installatie pakket Het activiteitendiagram in Figuur A.4 geeft weer welke stappen doorlopen worden voor het installeren van een software pakket in een container. Listing 4.7 geeft de code weer die gebruikt wordt om een pakket naar een container te kopiëren. Dit is enkel mogelijk als de bron een tar archief is. Verder bevatten Listing 4.8de code om een pakket te installeren.

```
package_name = package.name + package.version
package_location = self.find(package_name, self.release_zip_location)
has_include_folder = has_include(package_name, self.release_zip_location)
tar = create_zip(str(os.path.join(package_location, package_name)))
# This path has been added by the Dockerfile
self.container.put_archive("/usr/test/", tar)
```

Listing 4.7: Kopieer pakket naar container

```
script_location = "/usr/test/" + package_name +
                  "/meta/install_script.py"
command = "python " + script_location
self.container.start()
exe_start = self.container.exec_run(command, stream=True)
```

Listing 4.8: Installatie van een pakket

Controle van installatie Na het installeren van een pakket moet nagegaan worden of de installatie correct verlopen is. Dit wordt gedaan door gebruik te maken van de code in Listing 4.9. Er wordt een commando opgebouwd dat het test_script zal uitvoeren. Vervolgens wordt de output die de container genereert opgevangen. Start de output met een 0 dan wordt aangenomen dat de test geslaagd is. Zo niet dan is de test gefaald en wordt de container gemarkeerd voor quarantaine.

```

package_name = package.name + package.version
script_location = "/usr/test/" + package_name + "/meta/test_script.py"
command = "python " + script_location
self.container.start()
exe_start = self.container.exec_run(command, stream=True)
for val in exe_start:
    if val.startswith("0"):
        print("INSTALL AGENT — Package " + package_name +
              " was correctly installed")
        answer = True
    else:
        print("INSTALL AGENT — Package " + package_name +
              " ended with error \n\t ERROR " + val)
        answer = False

```

Listing 4.9: Controle van een pakket

4.3 Client-side

4.3.1 Deployment_client

Net als bij het opstarten van de server, moet bij het opstarten van een client verschillende parameters meegegeven worden. Uiteraard moet de broker interface en port overeenkomen met de parameters die zijn meegegeven tijdens het opstarten van de server.

4.3.2 Field_dock

Een belangrijk element uit de FieldDock klasse is de code die weergegeven wordt in Listing 4.10. Tijdens het opstarten van de services van het field dock, wordt een client aangemaakt om te kunnen interageren met Docker-machine. Deze stap is enkel nodig als er gewerkt wordt met de Docker Toolbox [44]. De Docker Toolbox is een alternatieve Docker omgeving voor Windows computers die niet voldoen aan de voorwaarden (Microsoft Windows 10 Professional of Enterprise 64-bit) om Docker te installeren. Computers met Windows 10 Professional of met Linux moeten environment variabele meegeven aan de methode.

```

def init_client(self):
    env = {"DOCKER_TLS_VERIFY": "1",
           "DOCKER_HOST": "tcp://192.168.99.100:2376",
           "DOCKER_CERT_PATH": "C:/Users/Pieter-Jan/" +
               ".docker/machine/machines/default",
           "DOCKER_MACHINE_NAME": "default",
           "COMPOSE_CONVERT_WINDOWS_PATHS": "true"}
    self.client = docker.from_env(environment=env)

```

Listing 4.10: Parameters voor de Docker client

Hoofdstuk 5

Evaluatie

5.1 Testscenario's

De ontworpen applicatie heeft verscheidene afhankelijkheden met Python modules en externe programma's. Om de applicatie uit te kunnen testen, moet de testomgeving correct geconfigureerd zijn. Om de verscheidene omgevingen correct in te stellen, wordt gebruik gemaakt van Listing B.6.

5.1.1 Meerdere clients

DoeI

Eén van de hoofddoelstellingen was het ontwerpen van een applicatie die software kan installeren bij een groeiend aantal gebruikers. Hiernaast moest de ontworpen applicatie bruikbaar zijn op verschillende besturingssystemen. Om dit uit te testen werd de volgende test ontworpen: Er wordt een testomgeving opgezet die bestaat uit één release dock, één broker en negen field docks. Twee van de negen field docks gebruiken Linux als besturingssysteem. De rest gebruikt Windows als besturingssysteem. Vervolgens wordt een installer gecreëerd en gereleased. Het doel is dat de software op alle doelsystemen correct wordt geïnstalleerd en dat de aanpassingen in de databank van het release dock zichtbaar zijn. Hierna worden enkele field docks uitgeschakeld en wordt een tweede installer verspreid.

Uitvoering

Figuren C.4, C.6 en C.5 geven weer hoe enkele testtoren beschreven worden en hoe deze zijn toegevoegd aan de databank. Hieruit blijkt al snel de verschillende field docks correct kunnen communiceren met het release dock. Uit de figuren blijkt ook dat de communicatie ondersteund wordt op alle toestellen (zowel Windows als Linux). Eventuele fouten die zouden kunnen optreden, zijn dan ook afkomstig van de agenten of omdat nog niet alle functionaliteiten aanwezig zijn.

Tijdens het installatieproces traden echter enkele fouten op. Op één van de Windows machines kwam het systeem vast te zitten tijdens het creëren van de Docker image. De oorzaak ligt hoogstwaarschijnlijk bij het afhalen van de Docker basisimages. Er is dus geen logische fout aanwezig in de code van de InstallAgent. Hiernaast trad er een fout op bij beide Linux machines tijdens het installeren van de software (zie Figuur C.12). Het enige dat geweten is, is dat er een segmentatiefout optrad en dat deze optrad ter hoogte van het uitpakken van de gezippte installer. De oorzaak van

deze fout is niet geweten en er zal extra onderzoek uitgevoerd moeten worden om te achterhalen hoe deze fout opgelost kan worden. Hierbij moet wel meegegeven worden dat de software, die aanwezig was tijdens het installeren van de nieuwe software, geen invloed van de fout ondervond. De software zat afgezonderd in zijn eigen container en blijft beschikbaar.

Op alle andere systemen verliep het installatieproces zonder veel moeite. De software werd correct geïnstalleerd op alle systemen zoals zichtbaar is op Figuur C.14. Ook de tweede installer wordt correct geïnstalleerd op de systemen die nog aanstonden en die nog geen fout hadden gegeven.

5.1.2 Slecht werkend pakket

Doeleind

De geschreven applicatie bevat enkele methodes om te controleren of het installatieproces foutloos is verlopen. Mocht dit niet het geval zijn, dan wordt de foutieve container in quarantaine geplaatst. Het doel van deze test is het uittesten of dit wel degelijk gebeurt.

Om deze eigenschap van de applicatie uit te testen wordt een simpele omgeving gerealiseerd waarin getest kan worden. Eén release dock en field dock worden opgestart in het begin. Vervolgens wordt een installer gemaakt die bestaat uit twee pakketten. Het eerste pakket wordt gebruikt om een bestand aan te maken dat nodig is voor het framework pakket. Het eerstgenoemde pakket zal weliswaar geen bestand aanmaken.

Om de test te laten slagen, moet de test van het niet-framework pakket registeren dat het bestand niet is aangemaakt. Op het einde van het installatieproces moet de container in quarantaine geplaatst worden. Zo is het mogelijk om na het installatieproces te controleren wat het probleem is in de container.

Uitvoering

De figuren in Sectie C.3 tonen welke stappen doorlopen worden tijdens het installatieproces. Het test script eindigt met een status code die niet nul is. Hierdoor wordt de container gemarkeerd om in quarantaine geplaatst te worden (zie Figuur C.22). Na het installatieproces wordt de container hernoemd waardoor het mogelijk is om achteraf een manuele controle van de container uit te voeren.

5.1.3 Netwerk monitoring

Doeleind

Als laatste test wordt nagegaan hoe het netwerkverkeer eruit ziet tijdens het releasen van een nieuwe installer. Deze test wordt gebruikt om te achterhalen hoeveel en wat voor netwerkverkeer gecreëerd wordt door de applicatie. Het monitoren van het netwerk wordt uitgevoerd tijdens de meerdere client test. De test produceert voldoende netwerk verkeer om een idee te krijgen van de hoeveelheid verkeer die zal geproduceerd worden. De volledige communicatie tussen de verschillende docks wordt opgenomen met Wireshark om er later een analyse op uit te voeren.

Uitvoering

De figuren in Sectie C.4 geven een notie van het netwerk verkeer dat gegenereerd wordt door de applicatie. Het genereerde verkeer blijft beperkt tot enkele kilobytes voor iedere handeling die wordt uitgevoerd. Hierbij moet wel vermeld worden dat meerdere van deze berichten verzonden worden per release. Wat de impact gaat zijn als een groot aantal gebruikers aanwezig is, is moeilijk in te schatten. Figuur C.26 geeft wel een algemeen beeld hiervoor. Wat ook vermeld moet worden, is het feit dat de installer grotendeels bestond uit scripts en tekstbestanden. Het gebruik van executables gaat echter voor meer netwerkverkeer zorgen.

5.1.4 Cross-platform containers

Doe

Zoals reeds werd aangegeven (Sectie 4.2.4), werd tijdens de implementatie van de applicatie Linux gebruikt als basis voor de containers. Het is echter wel belangrijk om te achterhalen in hoeverre het mogelijk is om de ontworpen applicatie te gebruiken in combinatie met Windows images. Om dit uit te testen, werd een test ontworpen.

Het eerste deel van de test bestaat uit het creëren van een container die losstaat van de applicatie. Door dit uit te voeren, is het eenvoudig om te achterhalen of het mogelijk is om:

- een image te maken van een Dockerfile
- een container te maken met de gecreëerde image
- commando's uit te voeren in de container

Als dit lukt, wordt in deel twee een testomgeving opgezet die bestaat uit één release dock, één broker en één field dock. Er wordt een installer gecreëerd die vervolgens gereleased wordt. Op deze manier is het mogelijk om de mogelijkheden van de ontworpen applicatie in te schatten.

Om dit uit te kunnen testen, wordt er gebruik gemaakt van een laptop. Deze is voorzien van Windows 10 Professional 64-bit versie aangezien dit een vereiste is om gebruik te kunnen maken van Docker for Windows [45].

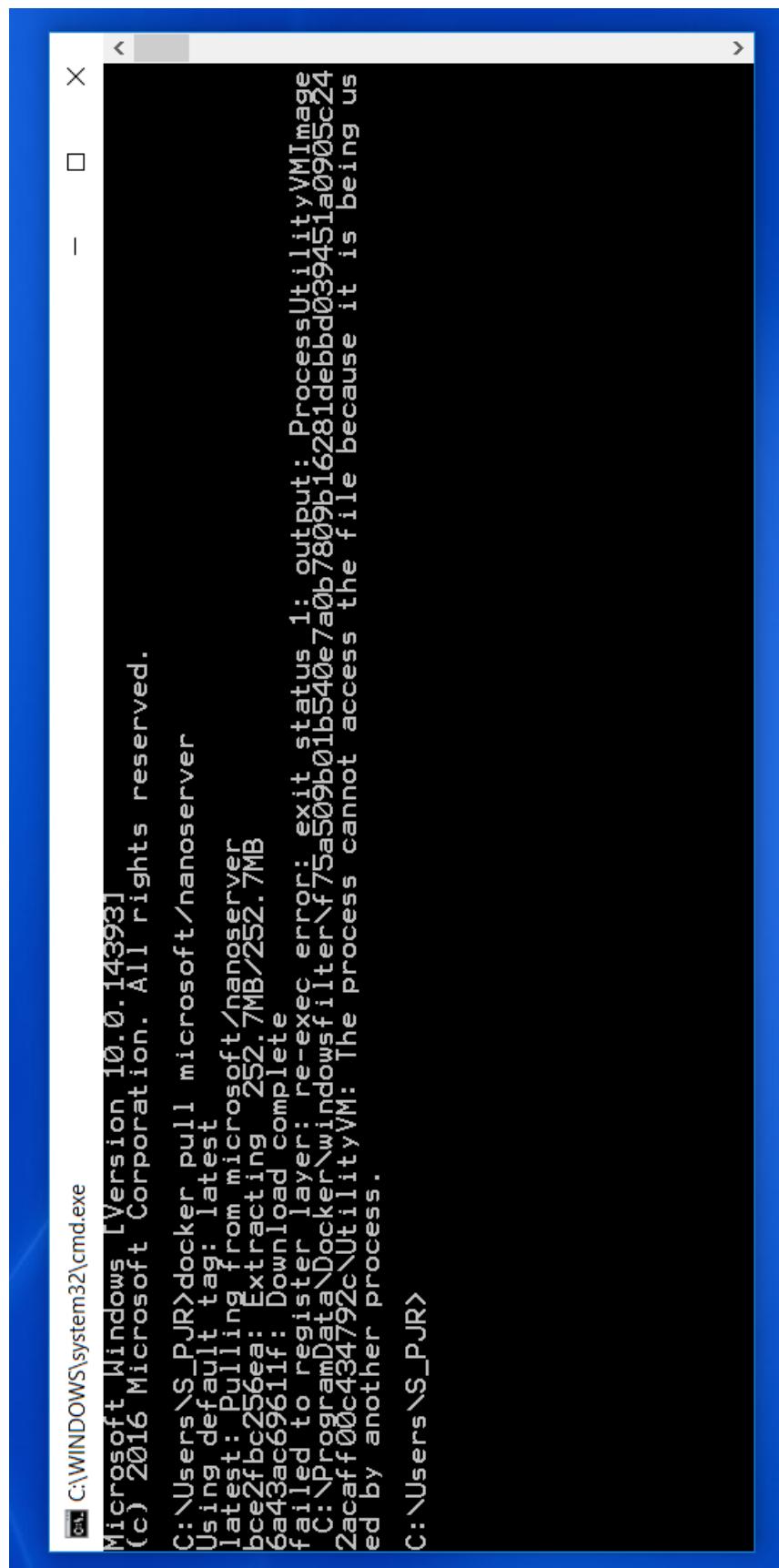
Resultaten

Tijdens het eerste deel van de test werd gebruik gemaakt van de microsoft/nanoserver als basis-image. Om gebruik te kunnen maken van deze image werd het commando “docker pull microsoft/nanoserver” gebruikt. Dit zorgde direct voor een error die zichtbaar is in Figuur 5.1. De oorzaak van het probleem ligt hoogstwaarschijnlijk bij de geïnstalleerde anti-virus software (<https://github.com/moby/moby/issues/30296>). Dit is een speculatie aangezien geen tijd meer aanwezig was om hierop dieper in te gaan.

5.1.5 Meerdere servers

Doe

Een volgende test bestaat uit het testen van de functionaliteiten als meerdere servers aanwezig zijn. Dit gebeurt aan de hand van een virtuele machine met Windows 10. Iedere virtuele machine



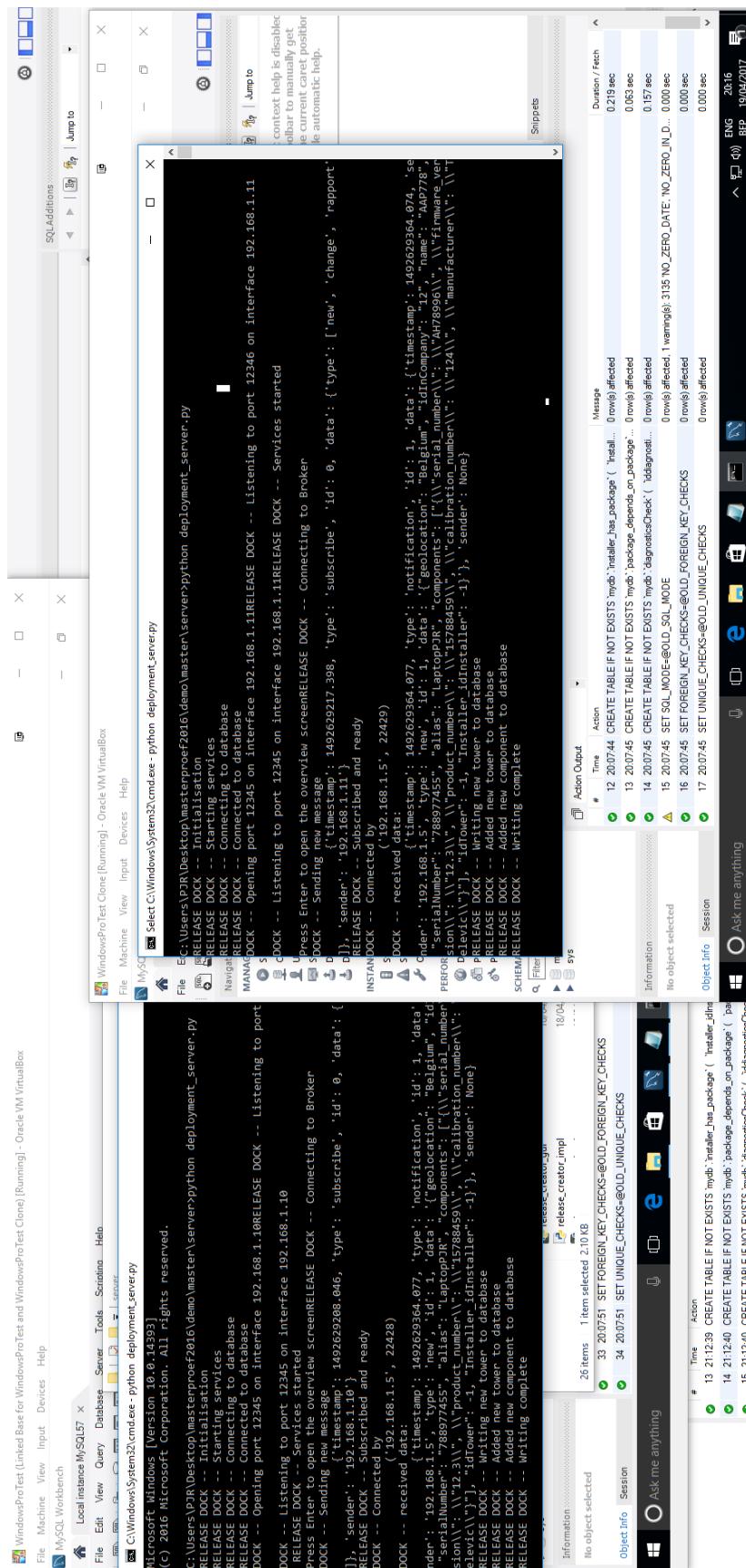
A screenshot of a Windows Command Prompt window titled 'cmd C:\WINDOWS\system32\cmd.exe'. The window shows the following command and its output:

```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\S_PJR>docker pull microsoft/nanoserver
Using default tag: latest
latest: Pulling from microsoft/nanoserver
bce2fbc256ea: Extracting 252.7MB/252.7MB
6a43ac69611f: Download complete
failed to register layer: re-exec error: exit status 1: output: ProcessUtilityVMImage
C:\ProgramData\Docker\windowsfilter\f5a509b01b540e7a0b7809b16281debb039451a0905c24
2acaff00c434792c\UtilityVM: The process cannot access the file because it is being us
ed by another process.

C:\Users\S_PJR>
```

Figuur 5.1: Error tijdens het gebruiken van Windows Docker containers



Figuur 5.2: Ontvangen van client informatie bij servers

bevat de server code samen met een databank om alle gegevens in op te slaan. De testomgeving bestaat uit drie release docks, één broker en één field dock.

De test bestaat uit het toevoegen van de beschrijving van het field dock aan alle databanken. Hierna wordt een installer gecreëerd op één van de release docks en wordt deze gereleased. Het is de bedoeling om in ieder release dock de aanpassing door te voeren.

Resultaten

Het eerste deel van de test bestaat uit het toevoegen van de client aan de nodige databanken. Figuur C.1 toont de beschrijving van het field dock weer. Figuur 5.2 geeft weer hoe de beschrijving van het field dock toekomt bij de release docks. Figuur C.2 toont vervolgens de managersinterface van beide release docks. Hieruit kan afgeleid worden dat ieder release dock de beschrijving correct heeft toegevoegd aan de databank.

Bij het releasen van de installer loopt het echter verkeerd af. De gereleasede installer komt correct bij de client toe (zie Figuur C.3) en wordt correct geïnstalleerd maar het verzenden van een update naar de release docks zorgt voor een fout. De fout die opgegooid wordt, ontstaat doordat het release dock de ID van de installer gaat opvragen in de databank. Het release dock die de installer heeft aangemaakt, vindt de correcte installer maar de andere release docks vinden deze niet terug in de databank. Dit komt doordat de informatie van de installer niet doorgegeven wordt aan de verschillende release docks. De installer komt dus niet in de databank terecht. In het ontwerp werd hier echter wel rekening mee gehouden. Deze functionaliteit werd nog niet geïmplementeerd.

De functionaliteit kan op verscheidene manier geïmplementeerd worden. Zo kan bijvoorbeeld gewerkt worden met maar één databank. Een andere manier bestaat uit het inschrijven van de release docks bij de broker voor berichten van het type “release”. Mocht een notificatie van de broker toekomen met hierin de informatie over een nieuwe installer, dan kan deze gebruikt worden om de installer toe te voegen aan de databank.

5.2 SWOT analyse

5.2.1 Strengths

Het gebruik van de software dock architectuur zorgt voor een schaalbare en flexibele applicatie die de continue toename van gebruikers en software pakketten ondersteunt. De ontworpen applicatie vormt een goede basis voor Televic om uit te breiden.

Hiernaast zorgt Docker voor een veilige installatieomgeving. Dit zorgt ervoor dat fouten tijdens het installatieproces geen invloed hebben op het systeem en er altijd een versie van de software van Televic beschikbaar gaat zijn.

5.2.2 Weaknesses

Een zwakte van de architectuur ligt bij de broker. Aangezien de broker alle berichten moet afhandelen, vormt deze een bottleneck in de architectuur. Er zijn echter oplossingen mogelijk waarbij de broker een handle terug geeft aan de docks [46]. Een andere strategie bestaat uit het gebruiken van meerdere brokers.

Veiligheid vormt ook een zwakte voor de applicatie. Het verzenden en ontvangen van berichten gebeurt momenteel niet veilig aangezien alle berichten ongeëncrypteerd verzoeden worden. Dit is echter op te lossen door cryptografische primitieven toe te voegen aan de applicatie.

De voornaamste zwakte bestaat uit het verder vervolledigen van de applicatie. Naast het installatieproces moet het ook mogelijk zijn om een updateproces uit te voeren. Hiervoor zijn al enkele methodes aanwezig die gebruikt kunnen worden.

5.2.3 Opportunities

Met de ontworpen architectuur is het mogelijk om verschillende applicaties te verspreiden naar een groep gebruikers. Dit biedt de mogelijkheid voor Televic om samen te werken met verschillende softwareverkopers om software tussen softwaregebruikers te verspreiden.

5.2.4 Threats

Een bedreiging van de applicatie ligt bij het gebruik van Docker. Mochten problemen bij de Docker Python module of Docker voor Windows, dan is Televic afhankelijk van een aparte groep van ontwikkelaars die deze problemen zo snel mogelijk moeten oplossen.

Hoofdstuk 6

Conclusie

Het doel van deze thesis is het ontwerpen van een oplossing voor het complexe installatieproces en updateproces. Er moet rekening gehouden worden met een toenemend aantal gebruikers en softwarepakketten die geïnstalleerd moeten worden en bij de gebruiker moeten geraken. Verder dient elk nieuw toestel op het raamwerk ondersteunt te worden waardoor er jaarlijks ettelijke releases van het raamwerk verspreid worden. Hierbij was het belangrijk om een oplossing te voorzien die zowel schaalbaar als flexibel is.

Om dit probleem op te lossen wordt gebruik gemaakt van de software dock architectuur die de basis vormt voor de applicatie. Door deze architectuur te combineren met een software packager waarvan het architectuur gebaseerd is op dat van het Qt installer framework. Met Docker was het mogelijk om een flexibele en schaalbare oplossing te voorzien die kan omgaan met fouten tijdens het installatieproces. De implementatie van het ontwerp vormt een goede basis die Televic kan gebruiken om uit te breiden en aan te passen.

Uit de verschillende testen bleek dat het doel van de thesis gehaald werd. Het is mogelijk om een softwarepakket te creëren bij een installer en deze te verspreiden naar een aantal gebruikers. Hierbij kan iedere stap in het deployment proces gepersonaliseerd worden door gebruik te maken van agenten. De opdeling van de software in pakketten zorgt ervoor dat ieder pakket op een unieke manier behandelt kan worden. De doelstelling om een flexibele oplossing te vinden is dus behaald. Verder ondersteunt de architectuur een groeiend aantal producenten en gebruikers. Uit de testen bleek dat het mogelijk is om verscheidene release docks en field docks toe te voegen aan een netwerk en deze correct met elkaar te laten communiceren. De doelstelling om een schaalbare oplossing te vinden is dus behaald. Additionele proeven tonen aan dat het gebruik van Docker voor een afscheiding zorgt tussen de verscheidene releases van de software. Fouten die optreden tijdens het installatieproces hebben geen invloed op eerdere releases aangezien deze in een aparte container zitten.

Verder onderzoek is echter wel nodig. Nog niet alle functionaliteiten zijn aanwezig. Het updateproces wordt momenteel nog niet ondersteund en uit testen bleek dat er nog geen volledige ondersteuning aanwezig is om meerdere servers te gebruiken. Deze functies moeten nog geïmplementeerd worden maar de ontworpen architectuur is hiervoor de geschikte basis.

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] C. Fletcher, D. P. Williams en L. F. Wurster, *Magic Quadrant for Application Release Automation*, <https://www.gartner.com/doc/reprints?id=1-3DSWYP2&ct=160801&st=sb>, [Online; geraadpleegd 22-04-2017], 2016.
- [6] E. Cloud, *ElectricFlow 7.2 API Guide*, http://docs.electric-cloud.com/eflow_doc/7_2/API/PDF/APIflow_7_2.pdf, [Online; geraadpleegd 22-04-2017], 2016.
- [7] E. C. Bailey, *Maximum rpm*, 1997.
- [8] 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.
- [9] 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.
- [10] I. Foster, C. Kesselman en S. Tuecke, „The anatomy of the grid: Enabling scalable virtual organizations”, *International journal of high performance computing applications*, deel 15, nr. 3, pp. 200–222, 2001.
- [11] 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>.
- [12] *CMT configuration management tool*, <http://www.cmtsite.org/>, [Online; geraadpleegd 20-04-2017], 2016.

- [13] G. Rybkin, „ATLAS software packaging”, *Journal of Physics: Conference Series*, deel 396, nr. 5, p. 4, 2012.
- [14] 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.
- [15] N. Belkhatir, J. Estublier en W. Melo, „THE ADELE-TEMPO experience: an environment to support process modeling and enactment.”, *Software Process Modelling and Technology Research Studies Press*, pp. 1–37, 2007.
- [16] *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.
- [17] 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.
- [18] J. S. Plank, M. Beck, G. Kingsley en K. Li, *Libckpt: Transparent checkpointing under unix*. Computer Science Department, 1994.
- [19] 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.
- [20] 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.
- [21] S. Shumate, *Implications of Virtualization for Image Deployment*, 2004. adres: <http://www.dell.com/downloads/global/power/ps4q04-20040152-Shumate.pdf>.
- [22] J. De Jesús, *How, where, and why IBM PureApplication fits in your cloud*, https://www.ibm.com/developerworks/websphere/techjournal/1506_dejesus/1506_dejesus-trs.html, [Online; geraadpleegd 22-04-2017], 2015.
- [23] M. Fenn, M. A. Murphy, J. Martin en S. Goasguen, „An evaluation of KVM for use in cloud computing”, in *Proc. 2nd International Conference on the Virtual Computing Initiative, RTP, NC, USA*, 2008.
- [24] *Docker Main Page*, <https://www.docker.com/>, [Online; geraadpleegd 16-08-2016], 2016.
- [25] Docker, „Docker for the Virtualization Admin”, 2016.
- [26] R. Chamberlain en J. Schommer, „Using Docker to support reproducible research”, DOI: <http://dx.doi.org/10.6084/m9.figshare>, deel 1101910, 2014.
- [27] D. Merkel, „Docker: lightweight linux containers for consistent development and deployment”, *Linux Journal*, deel 2014, nr. 239, p. 2, 2014.
- [28] 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.
- [29] D. A. Patterson, „The data center is the computer”, *Communications of the ACM*, deel 51, nr. 1, pp. 105–105, 2008.

- [31] 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.
- [32] 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.
- [33] 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.
- [34] J. R. Callahan, „Software packaging”, tech. rap., 1998.
- [35] *WiX Toolset*, <http://wixtoolset.org/>, [Online; geraadpleegd 3-08-2016], 2016.
- [36] B. Amstadt en M. K. Johnson, „Wine”, *Linux Journal*, deel 1994, nr. 4es, p. 3, 1994.
- [37] 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>.
- [38] *NSIS Main page*, http://nsis.sourceforge.net/Main_Page, [Online; geraadpleegd 3-08-2016], 2016.
- [39] *Chocolatey About page*, <https://chocolatey.org/about>, [Online; geraadpleegd 3-08-2016], 2016.
- [40] *QT Installer Framework Documentation*, <http://doc.qt.io/qtinstallerframework/>, [Online; geraadpleegd 4-08-2016], 2016.
- [41] wxPython, *wxPython*, <https://wxpython.org/>, [Online; geraadpleegd 08-05-2017], 2017.
- [42] Python, *dill 0.2.6*, <https://pypi.python.org/pypi/dill>, [Online; geraadpleegd 08-05-2017], 2017.
- [43] D. Inc., *Docker SDK for Python*, <https://docker-py.readthedocs.io/en/stable/>, [Online; geraadpleegd 08-05-2017], 2017.
- [44] ——, *Docker Toolbox*, <https://www.docker.com/products/docker-toolbox>, [Online; geraadpleegd 08-05-2017], 2017.
- [45] ——, *Docker For Windows*, <https://www.docker.com/docker-windows>, [Online; geraadpleegd 08-05-2017], 2017.
- [46] R. M. Adler, „Distributed coordination models for client/server computing”, *Computer*, deel 28, nr. 4, pp. 14–22, 1995.
- [49] *WiX Toolset Tutorial*, <https://www.firegiant.com/wix/tutorial/>, [Online; geraadpleegd 3-08-2016], 2016.
- [50] *Chocolatey Main page*, <https://chocolatey.org/>, [Online; geraadpleegd 3-08-2016], 2016.
- [51] M. D. Hanson, „The Client/Server Architecture”, *Server Management*, p. 3, 2000.
- [52] Microsoft, *Chapter 5: Layered Application Guidelines*, <https://msdn.microsoft.com/en-us/library/ee658109.aspx>, [Online; geraadpleegd 25-04-2017], 2017.

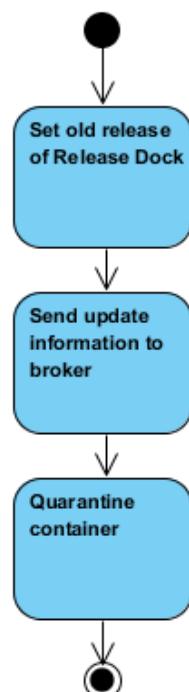
- [53] A. Ortiz Ramirez, *Three-Tier Architecture*, <http://www.linuxjournal.com/article/3508>, [Online; geraadpleegd 25-04-2017], 2000.
- [55] T. E. of Encyclopaedia Britannica, *Client-server architecture*, <https://www.britannica.com/technology/client-server-architecture>, [Online; geraadpleegd 09-05-2017], 2017.
- [56] Oracle, *The Oracle Client/Server Architecture*, https://docs.oracle.com/cd/A57673_01/DOC/server/doc/SCN73/ch20.htm, [Online; geraadpleegd 09-05-2017], 1996.
- [57] A. Mesbah en A. Van Deursen, „A component-and push-based architectural style for ajax applications”, *Journal of Systems and Software*, deel 81, nr. 12, pp. 2194–2209, 2008.
- [58] CERN, *About the ATLAS Experiment*, <https://atlas.cern/discover/about>, [Online; geraadpleegd 22-04-2017], 2017.
- [59] ——, *The Standard Model*, <http://home.cern/about/physics/standard-model>, [Online; geraadpleegd 22-04-2017], 2017.
- [60] Wikipedia, *Client (computing)*, [https://en.wikipedia.org/wiki/Client_\(computing\)](https://en.wikipedia.org/wiki/Client_(computing)), [Online; geraadpleegd 25-04-2017], 2017.
- [61] ——, *Server (computing)*, [https://en.wikipedia.org/wiki/Server_\(computing\)](https://en.wikipedia.org/wiki/Server_(computing)), [Online; geraadpleegd 25-04-2017], 2017.
- [62] 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.
- [63] M. Friis, *Build and run your first docker windows server container*, <https://blog.docker.com/2016/09/build-your-first-docker-windows-server-container/>, [Online; geraadpleegd 10-04-2017], 2016.

Bibliografie

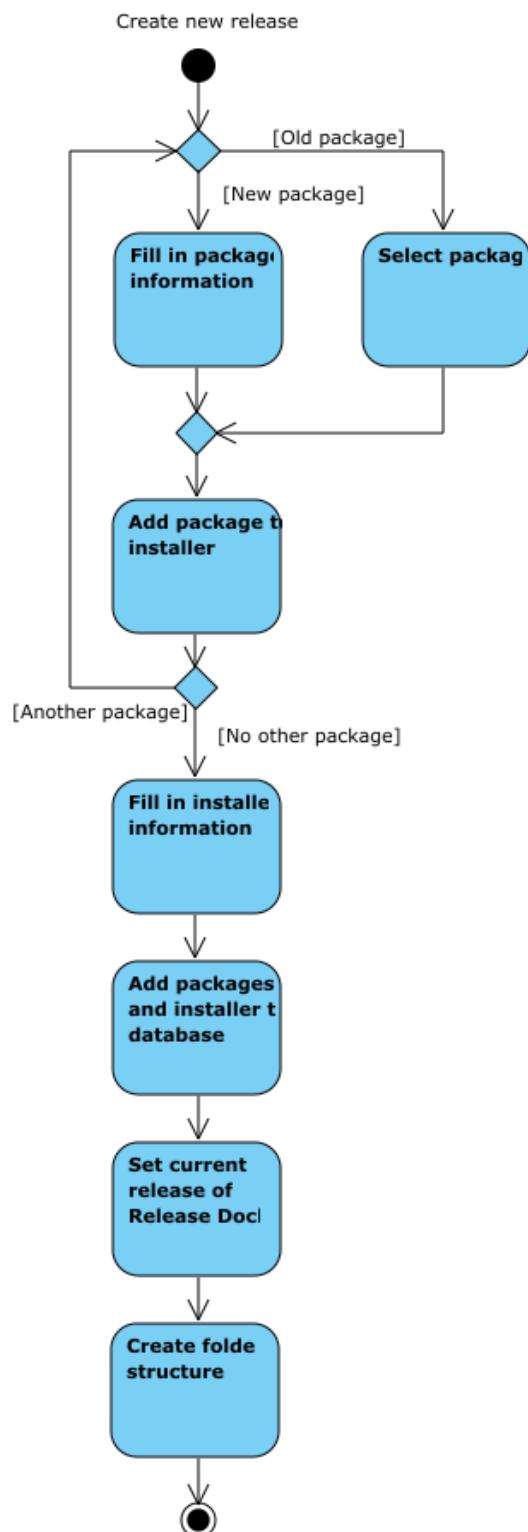
- [30] J. Münch, O. Armbrust, M. Kowalczyk en M. Sotó, *Software process definition and management*. Springer Science & Business Media, 2012.
- [47] R. S. Pressman, *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.
- [48] D. Pitts, B. Ball e.a., *Red Hat Linux*. Sams, 1998.
- [54] G. S. Machado, F. F. Daitx, W. L. da Costa Cordeiro, C. B. Both, L. P. Gaspari, 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

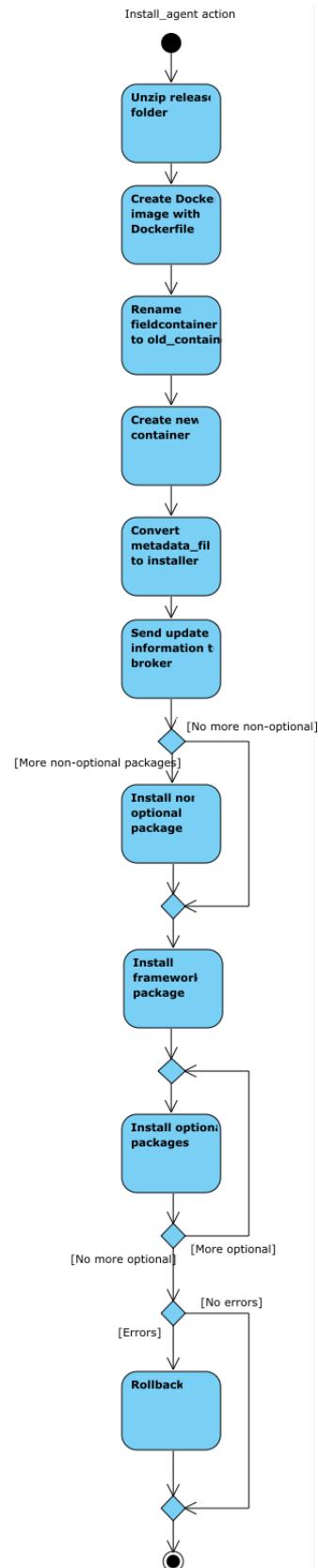
Activiteitendiagrammen



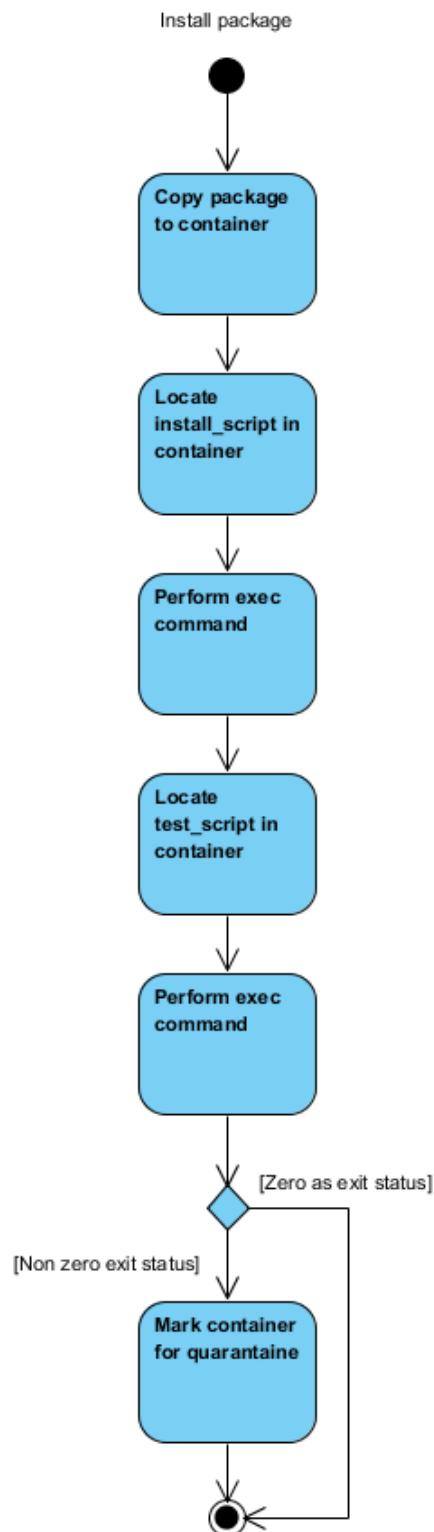
Figuur A.1: Activiteitendiagram voor een rollback uit te voeren



Figuur A.2: Activiteitendiagram voor het creëren van een installer



Figuur A.3: Activiteitendiagram van een InstallAgent



Figuur A.4: Activiteitendiagram voor het installeren van een software pakket

Bijlage B

Listings

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

```

Listing B.3: Qt installer framework installatiescript

```

function Component()
{
    // Default component
    // Script van exe
}

Component.prototype.createOperations = function()
{
    // call default implementation to actually install README.txt !
    component.createOperations();

    if (systemInfo.productType === "windows") {
        component.addOperation("Execute"
            , "msiexec"
            , "/i"
            , "@TargetDir@\python-2.7.3.msi"
            , "/quiet"
            , "UNDOEXECUTE"
            , "msiexec"
            , "/qb"
            , "/x"
            , "@TargetDir@\python-2.7.3.msi")
    } else {
        component.addOperation("Execute"
            , "tar"
            , "-xvf"
            , "@TargetDir@/Python-2.7.11.tgz"
    }
}

```

```

        , “-C”
        , “@TargetDir@ /”
    )
    component.addOperation(“Execute”
        , “@TargetDir@ / Python –2.7.11/configure”
    )
    component.addOperation(“Execute”
        , “make”
    )
    component.addOperation(“Execute”
        , “make”
        , “install”
    )
}
}

```

Listing B.4: Qt installer framework installer beschrijving

```

<?xml version=“1.0” encoding=“UTF–8”?>
<Installer>
    <Name>Eerste Test</Name>
    <Version>1.0.0</Version>
    <Title>Playground</Title>
    <Publisher>PJ Industries </Publisher>
    <!-- Directory name is used in component.xml -->
    <StartMenuDir>PythonFrameWork</StartMenuDir>
    <TargetDir>@HomeDir@/ IfwExamples/eigenTest</TargetDir>
    <RemoteRepositories>
        <Repository>
            <Url>http :// localhost / repository </Url>
        </Repository>
    </RemoteRepositories>
</Installer>

```

Listing B.5: Qt installer framework pakket beschrijving

```

<?xml version=“1.0” encoding=“UTF–8”?>
<Package>
    <DisplayName>Executable </DisplayName>
    <Description>Python die moet worden geïnstalleerd </Description>
    <Version>1.0.2–1</Version>
        <Script>installscript.qs</Script>
        <SortingPriority>100</SortingPriority>
    <ReleaseDate>2015–01–01</ReleaseDate>
    <Default>true </Default>
</Package>

```

Listing B.6: Installatie volgorde voor de ontworpen applicatie

```
#####
#INSTALLATIE#
#####

WINDOWS 10
*****  
  
CLIENT
_____  
  
Python 2.7 installeren  
wxPython installeren  
docker of docker toolbox installeren  
pip install dill  
pip install docker  
De nodige interfaces aanpassen  
Bij het gebruik van de docker toolbox , env variabele in het field dock  
    ↳ aanpassen  
INSTELLEN X11 SERVER IN CYGWIN  
* export DISPLAY=:0.0  
* startxwin --listen tcp &  
* xhost +  
client en server package in site-packages van Python installatie  
    ↳ plaatsen  
  
SERVER
_____  
  
Python 2.7 installeren  
wxPython installeren  
pip install dill  
pip install mysql.connector  
server package in site-packages van Python installatie plaatsen  
Mysql database installeren  
database aanmaken met sql bestand  
  
LINUX
*****  
  
CLIENT
_____  
  
installeer Python  
sudo apt-get install python-wxgtk3.0  
sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 \
    --recv-keys 58118E89F3A912897C070ADBF76221572C52609D  
sudo apt-add-repository 'deb https://apt.dockerproject.org/repo ubuntu-
    ↳ xenial main'
```

```
sudo apt update
sudo apt install linux-image-generic linux-image-extra-virtual
sudo reboot
pip install dill
pip install docker
```

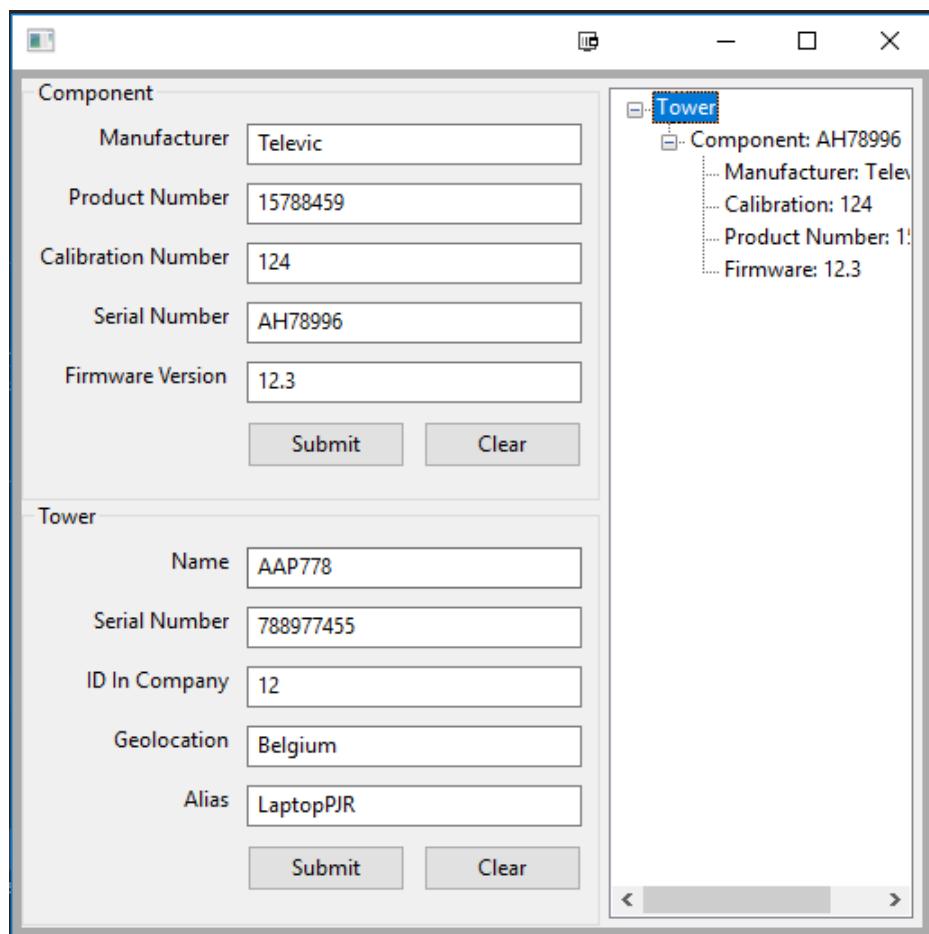
SERVER

```
Python 2.7 installeren
sudo apt-get install python-wxgtk3.0
pip install dill
pip install mysql.connector
server package in site-packages van Python installatie plaatsen
Mysql database installeren
database aanmaken met sql bestand
```

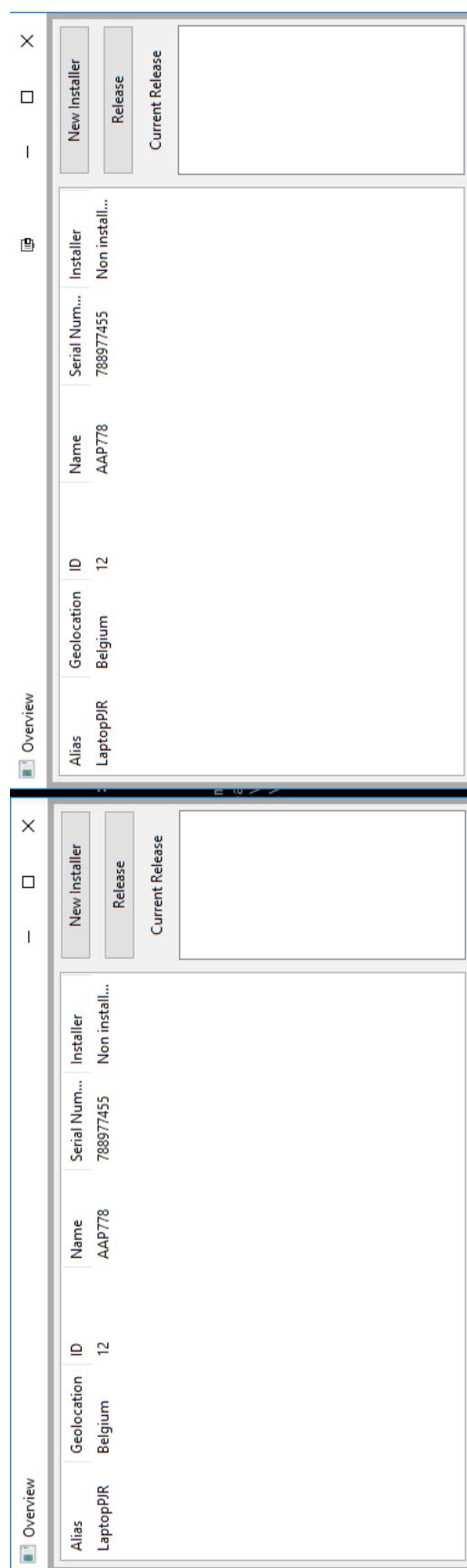
Bijlage C

Testscenario's

C.1 Meerdere servers



Figuur C.1: Beschrijven van een client



Figuur C.2: Client overview van meerdere servers

The screenshot shows a Windows desktop environment with several open windows. In the foreground, a terminal window titled 'Recycle B' displays the output of a Python script named 'deployment_server.py'. The script is running on port 12345 and is connected to a broker. It receives a message from a Docker container and sends a response back to it. The terminal window has a dark theme.

```

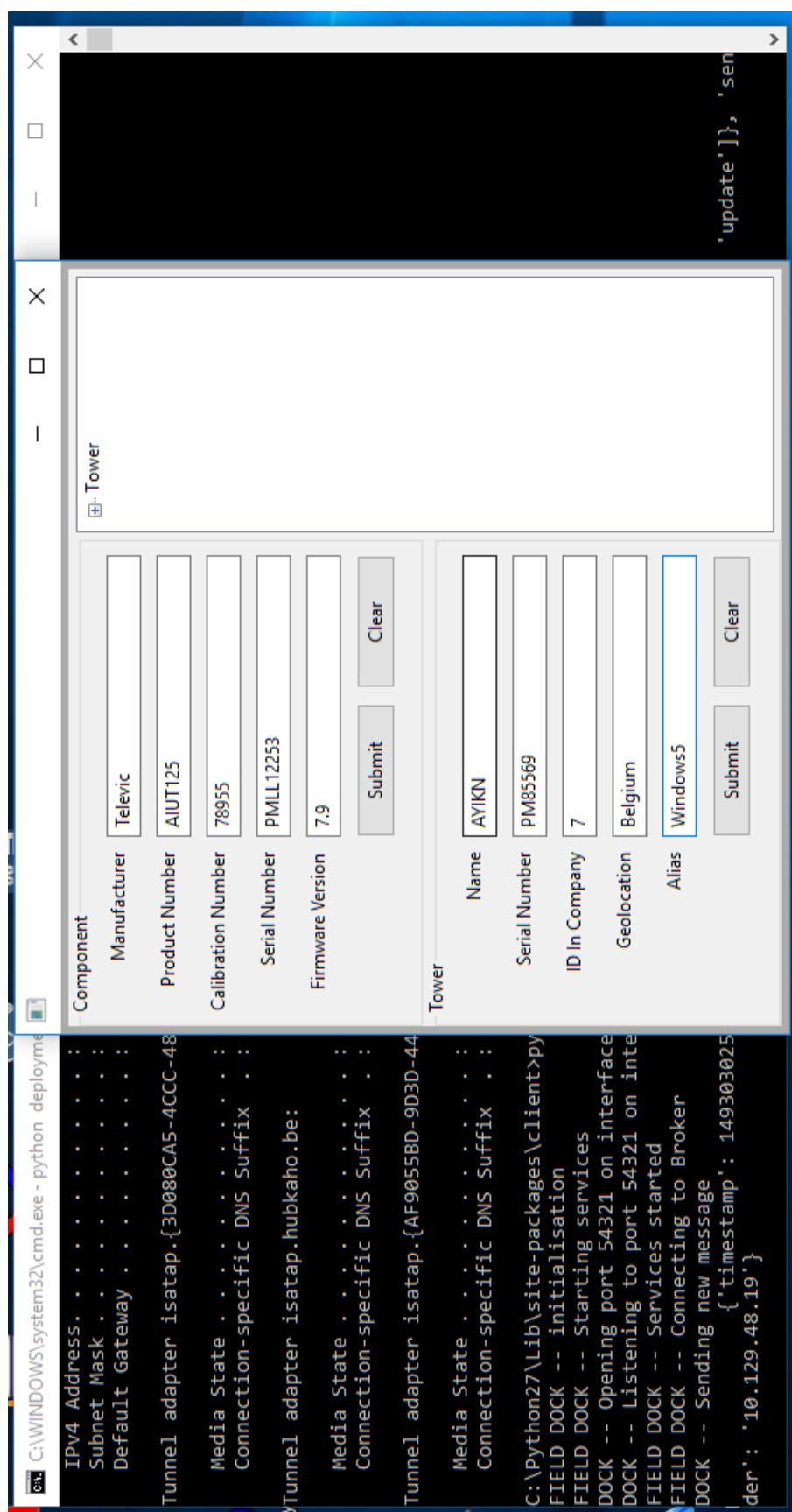
Recycle B
File Machine View Input Devices Help
C:\Windows\System32\cmd.exe - python deployment_server.py
> Dock -- Listening to port 12345 on interface 192.168.1.10RELEASE DOCK -- Services started
Dock -- Open the overview screenRELEASE DOCK -- Connecting to Broker
Docker -- Sending new message
{'timestamp': 1493554496.341, 'type': 'subscribe', 'id': 0, 'data': {'type': ['new', 'change', 'rapport'], 'sender': '192.168.1.10'}}
Quickstart..RELEASE DOCK -- Subscribed and ready
Dock -- Connected by
('192.168.1.9', 32370)
Dock -- received data:
{'timestamp': 1493554541.853, 'type': 'notification', 'id': 3, 'data': {'timestamp': '3.0', 'name': 'Deployment', 'idInstaller': 0}, 'sender': None}
Kitematica: '192.168.1.9', 'type': 'change', 'id': 1, 'data': {'version': '3.0', 'name': 'Deployment', 'idInstaller': 0}}
(Alpha) sender: None
Exception in thread Thread-3:
Traceback (most recent call last):
  File "C:\Python27\lib\threading.py", line 801, in _bootstrap_inner
    self.run()
  File "C:\Python27\lib\threading.py", line 754, in run
    self._target(*self._args, **self._kwargs)
  File "C:\Users\PR\Desktop\masterprotoF2016\demo\master\server\release_dock.py", line 257, in handle_message
    self.actions[replied.message.type](replied_message)
  File "C:\Users\PR\Desktop\masterprotoF2016\demo\master\server\release_dock.py", line 272, in change_tower
    self.update_installer_info(d, sender)
  File "C:\Users\PR\Desktop\masterprotoF2016\demo\master\server\release_dock.py", line 405, in update_installer_info
    id_tower = get_tower_of_sender(self.cnx, sender)
  File "C:\Users\PR\Desktop\masterprotoF2016\demo\master\server\release_dock.py", line 137, in get_tower_of_sender
    return id_tower
UnboundLocalError: local variable 'id_tower' referenced before assignment
masterproto...

```

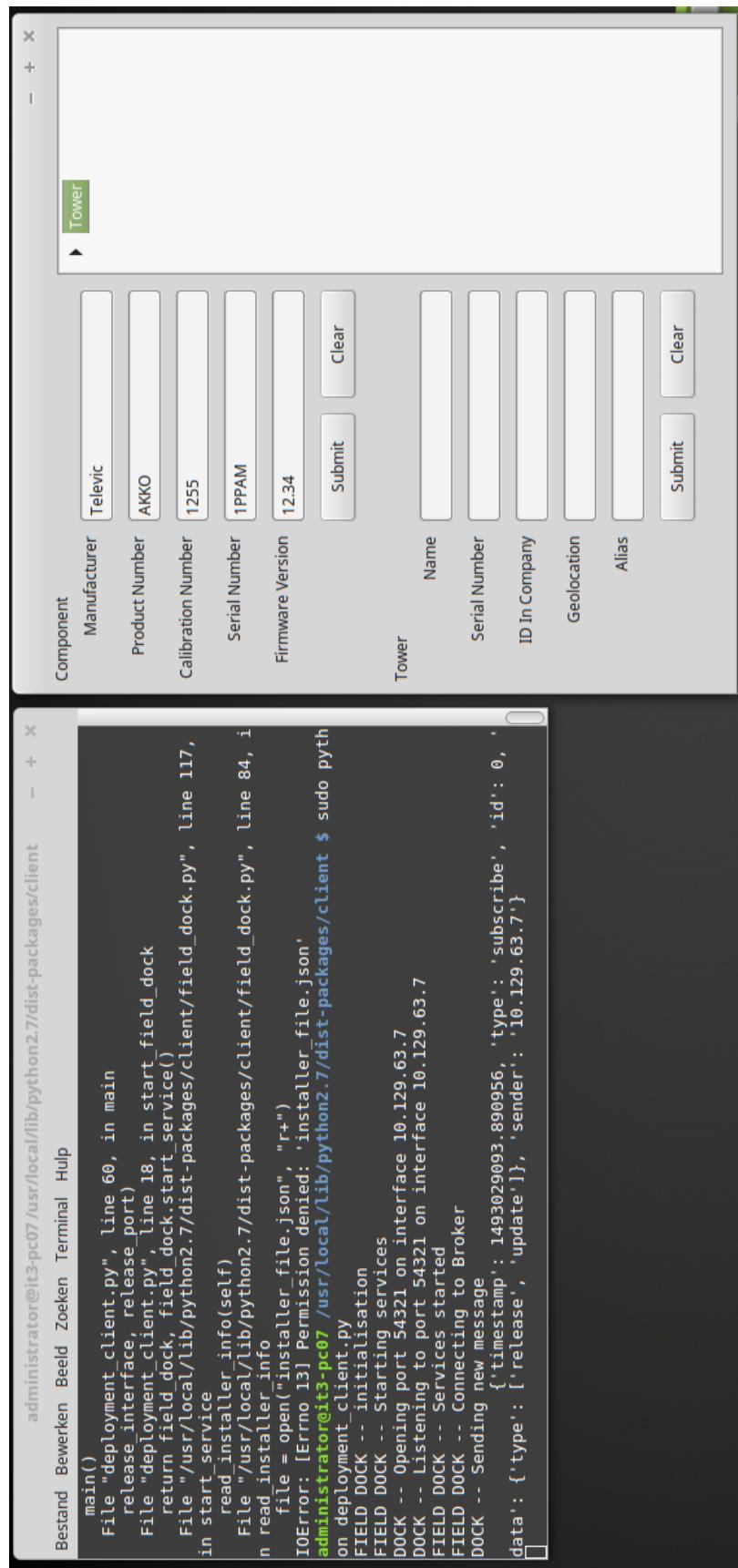
In the background, a file explorer window is open, showing a folder named 'masterproto...' containing three files: 'release_creator_gui', 'release_creator_gui', and 'release_creator_impl'. The file 'release_creator_gui' was modified on 18/04/2017 at 15:45 and has a size of 15 KB. The other two files were modified on 18/04/2017 at 16:38 and have sizes of 13 KB each. The taskbar at the bottom of the screen shows icons for various applications, including a browser, file explorer, and system icons.

Figuur C.3: Error tijdens het aanpassen van de field dock gegevens

C.2 Meerdere clients



Figuur C.4: Beschrijven van client in Windows



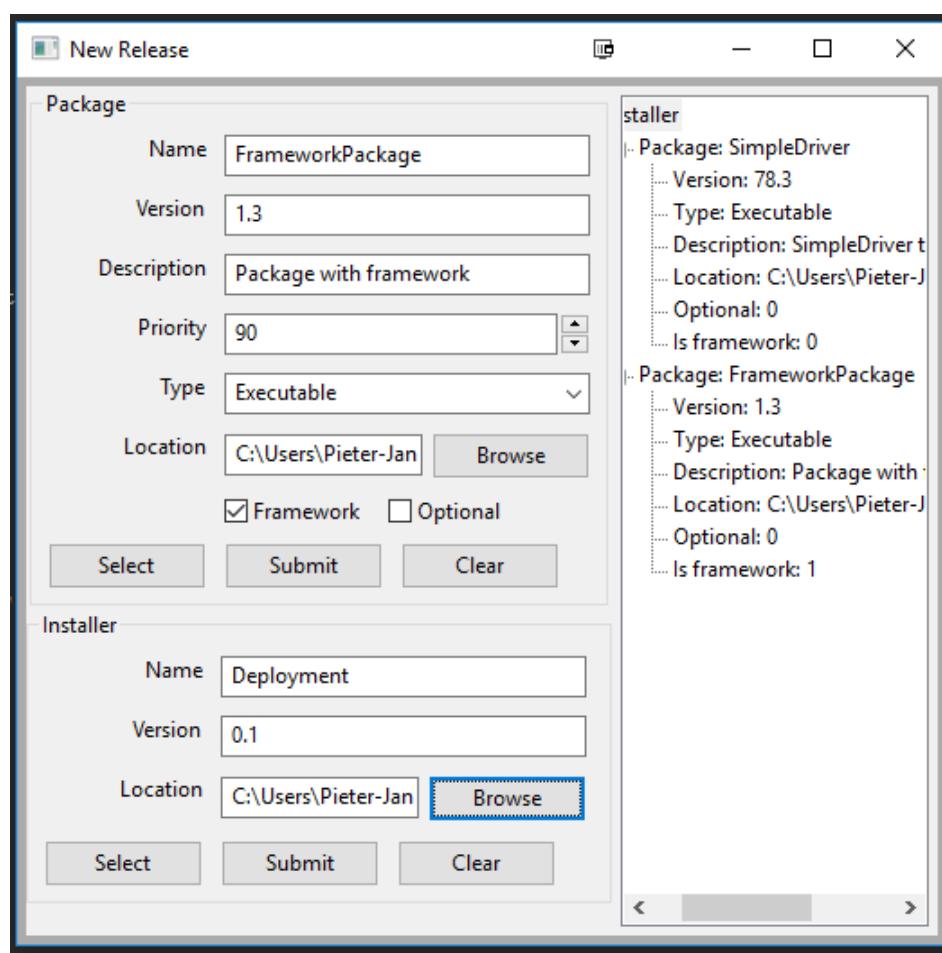
Figuur C.5: Beschrijven van client in Linux

| Alias | Geolocation | ID | Name | Serial Num... | Installer |
|---------------|-------------|----|----------|---------------|----------------|
| LinuxMachi... | Belgium | 1 | JJAK55 | APPROE | Non install... |
| LinuxMachi... | America | 2 | HAU885 | 123PL | Non install... |
| Windows1 | Belgium | 3 | PPOE | 788955SDD5 | Non install... |
| Windows2 | Belgium | 4 | POEI99 | SN778 | Non install... |
| Windows3 | America | 5 | PMLLK77 | RYT85 | Non install... |
| Windows4 | Belgium | 6 | IUYT7458 | 4568LK | Non install... |

Figuur C.6: Overview van de clients tijdens het toevoegen

| Alias | Geolocation | ID | Name | Serial Num... | Installer |
|---------------|-------------|----|-----------|---------------|----------------|
| LinuxMachi... | Belgium | 1 | JJAK55 | APPROE | Non install... |
| LinuxMachi... | America | 2 | HAU885 | 123PL | Non install... |
| Windows1 | Belgium | 3 | PPOE | 788955SDD5 | Non install... |
| Windows2 | Belgium | 4 | POEI99 | SN778 | Non install... |
| Windows3 | America | 5 | PMLLK77 | RYT85 | Non install... |
| Windows4 | Belgium | 6 | IUYT7458 | 4568LK | Non install... |
| Windows5 | Belgium | 7 | AYIKN | PM85569 | Non install... |
| Windows6 | Belgium | 8 | POHIGH785 | 7855JK | Non install... |
| Windows7 | Belgium | 9 | CNVB785 | 78554QD | Non install... |

Figuur C.7: Overview van clients na toevoegen van de clients



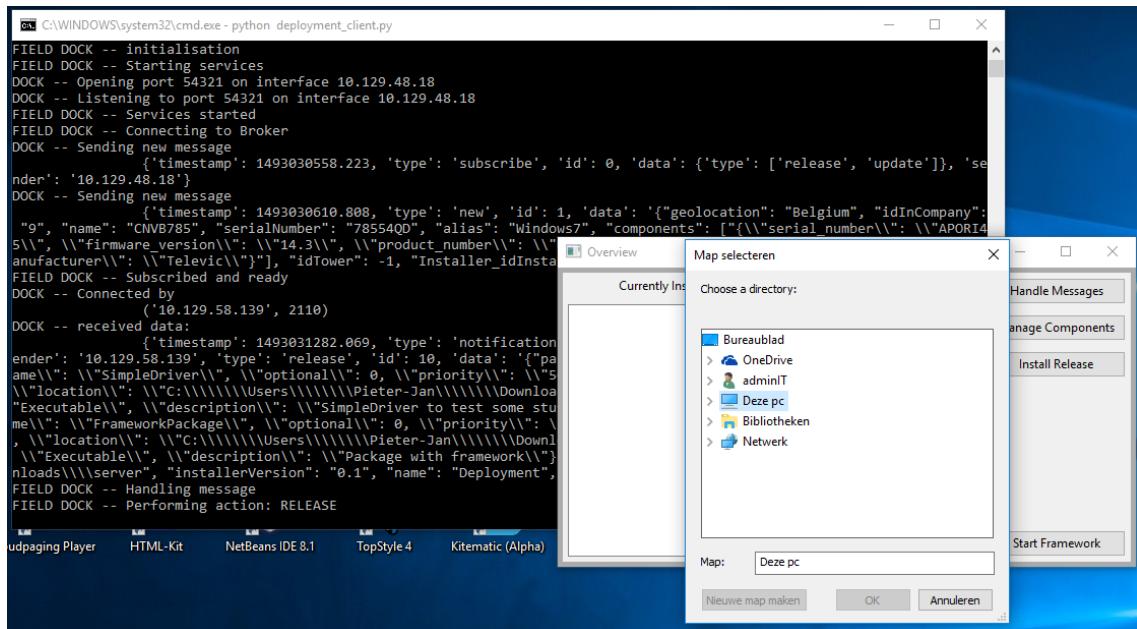
Figuur C.8: Creatie van een nieuwe installer

| Alias | Geolocation | ID | Name | Serial Num... | Installer | |
|---------------|-------------|----|----------|---------------|----------------|--|
| LinuxMachi... | Belgium | 1 | JIAK55 | APPROE | Non install... | |
| LinuxMachi... | America | 2 | HAU885 | 123PL | Non install... | |
| Windows1 | Belgium | 3 | PPOE | 788955SDD5 | Non install... | |
| Windows2 | Belgium | 4 | POEI99 | SN778 | Non install... | |
| Windows3 | America | 5 | PMLLK77 | RYT85 | Non install... | |
| Windows4 | Belgium | 6 | IUYT7458 | 4568LK | Non install... | |
| Windows5 | Belgium | 7 | AYIKN | PM85569 | Non install... | |
| Windows6 | Belgium | 8 | POGHHT85 | 7855JK | Non install... | |
| Windows7 | Belgium | 9 | CNVB785 | 78554QD | Non install... | |

On the right, there is an 'Overview' window showing the clients and their details. It includes sections for 'New Installer', 'Release', and 'Current Release'. The 'Current Release' section shows the newly created 'Deployment' installer with its components:

- .. Installer: Deployment0.1
 - .. Package: SimpleDriver
 - .. Version: 78.3
 - .. Type: Executable
 - .. Description: SimpleDriver to test some stuff
 - .. Location: C:\Users\Pieter-Jan\Downloads\opt
 - .. Optional: 0
 - .. Is framework: 0
 - .. Package: FrameworkPackage
 - .. Version: 1.3
 - .. Type: Executable
 - .. Description: Package with framework
 - .. Location: C:\Users\Pieter-Jan\Downloads\opt2

Figuur C.9: Overview van clients na creëren van een installer



Figuur C.10: Selecteren van een folder voor de installer in te plaatsen

```

C:\WINDOWS\system32\cmd.exe - python deployment_client.py

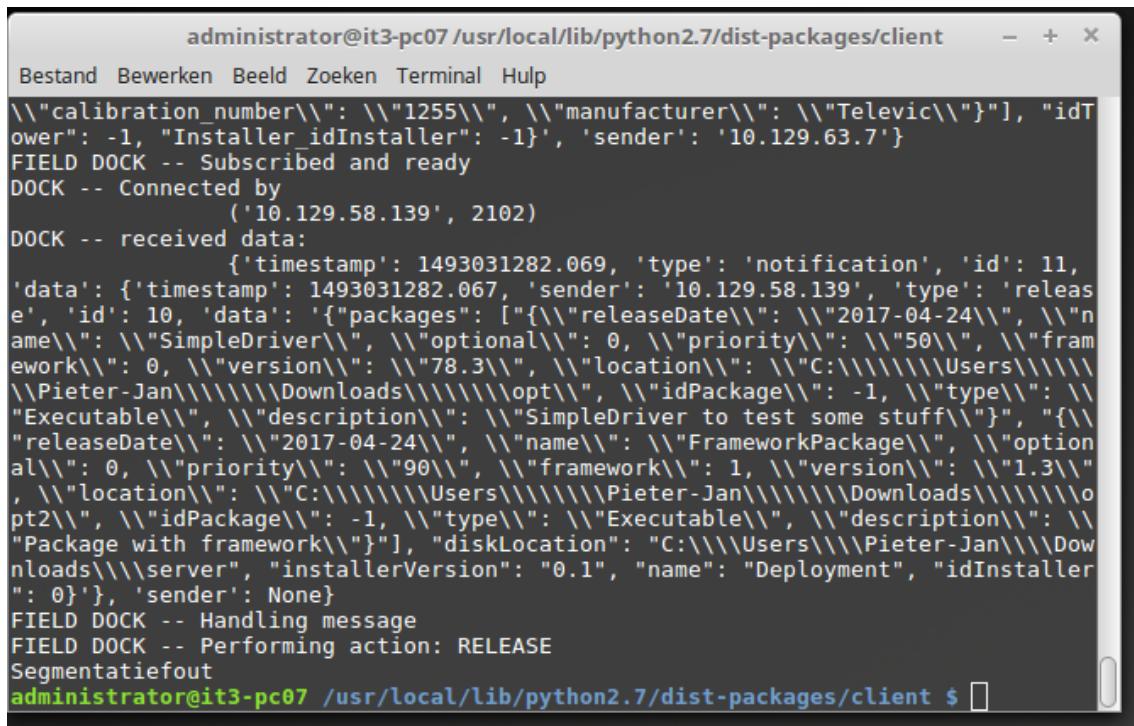
FIELD DOCK -- Connecting to Broker
DOCK - Sending new message
DOCK - Sending new message
DOCK - Sending new message
DOCK - Connected by ('10.129.58.139', 2105)
DOCK - received data:
ender': '10.129.58.139', 'type': 'notification'
ender': '10.129.58.139', 'type': 'release', 'id': 10, 'data': {
  "name": "POE199", "serialNumber": "SN778", "alias": "Windows2", "components": [
    {"serialNumber": "\\\\AUTI\\\\", "productNumber": "\\\\URYYH789\\\\", "calibrationNumber": "\\\\1699\\\\", "manufacturer": "\\\\Telenic\\\\"}, {"productNumber": "\\\\18.6\\\\", "calibrationNumber": "\\\\1699\\\\", "sender": "10.129.58.140"}]
}
FIELD DOCK -- Subscribed and ready
DOCK - Connected by ('10.129.58.139', 2105)

FIELD DOCK -- Handling message
FIELD DOCK -- Performing action: RELEASE
FIELD DOCK -- Release zip made
FIELD DOCK -- Received release, ready to install
FIELD DOCK -- Downloading agents
FIELD DOCK -- Downloaded all the agents
INSTALL AGENT -- Starting

```

The screenshot shows a software application window with a dark blue header bar. The main area contains a terminal-like log window displaying the output of a Python script named 'deployment_client.py'. The log shows the client connecting to a broker, sending and receiving messages, and performing various actions like handling messages and installing components. Below the log is a large central panel with several tabs: 'Overview' (which is selected), 'Currently Installed', 'New Available', 'Handle Messages', 'Manage Components', and 'Install Release'. The 'Overview' tab displays a grid of items, some of which are highlighted in red. At the bottom of the screen, there is a taskbar with icons for 'Upgrading Player' and other system applications.

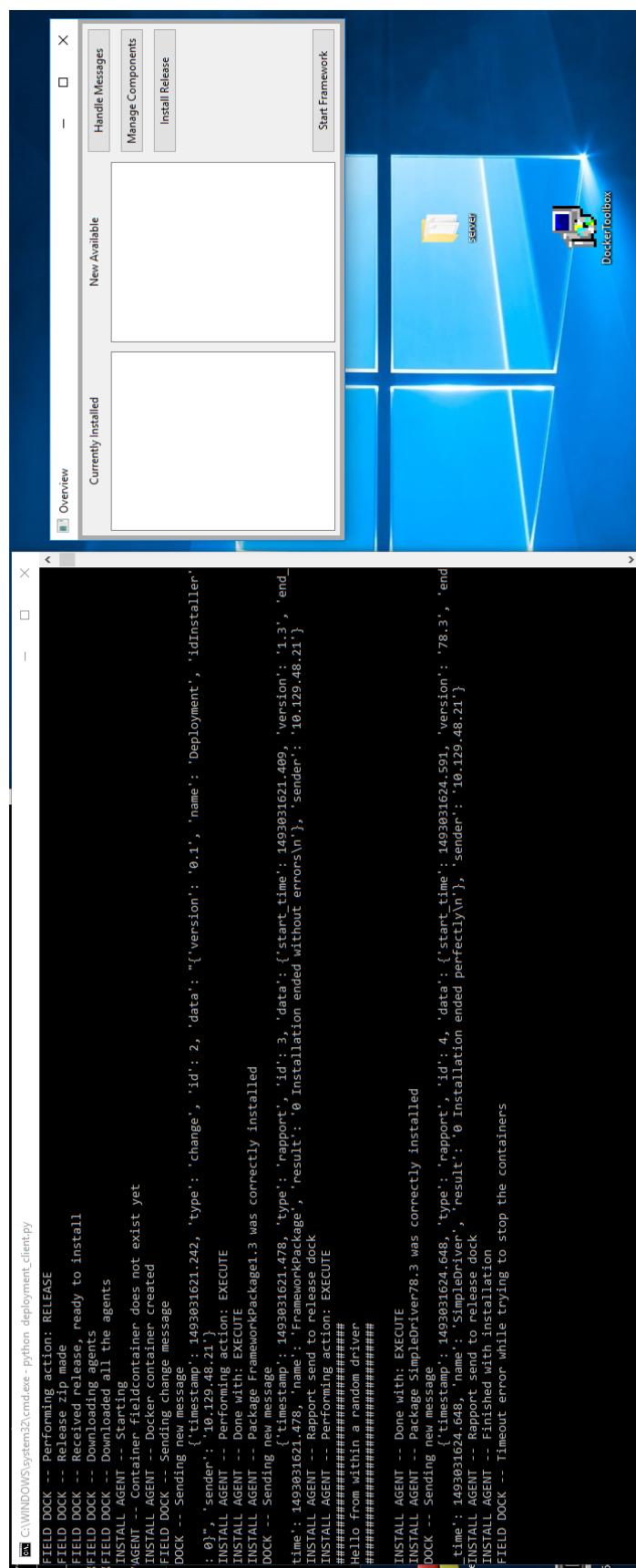
Figuur C.11: Output tijdens het installeren van de software



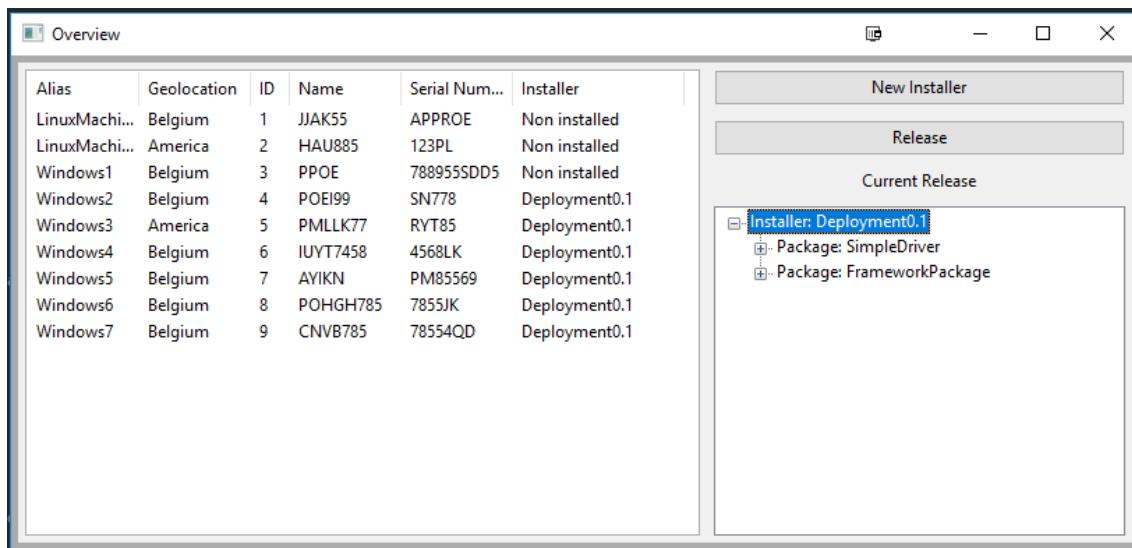
The screenshot shows a terminal window titled 'administrator@it3-pc07 /usr/local/lib/python2.7/dist-packages/client'. The window contains the following text:

```
Bestand Bewerken Beeld Zoeken Terminal Hulp
\"calibration_number\": \"1255\", \"manufacturer\": \"Televic\"}], \"idT
owner\": -1, \"Installer_idInstaller\": -1}', 'sender': '10.129.63.7'}
FIELD DOCK -- Subscribed and ready
DOCK -- Connected by
    ('10.129.58.139', 2102)
DOCK -- received data:
    {'timestamp': 1493031282.069, 'type': 'notification', 'id': 11,
'data': {'timestamp': 1493031282.067, 'sender': '10.129.58.139', 'type': 'releas
e', 'id': 10, 'data': {'packages': [{"releaseDate": "\\"2017-04-24\\\"", "n
ame": "\\\"SimpleDriver\\\"", "optional": 0, "priority": "\\\"50\\\"", "fram
ework": 0, "version": "\\\"78.3\\\"", "location": "\\\"C:\\\\\\\\Users\\\\\\\\\\
Pieter-Jan\\\\\\\\Downloads\\\\\\\\opt\\\"", "idPackage": -1, "type": "\\\"Exe
cutable\\\"", "description": "\\\"SimpleDriver to test some stuff\\\""}, {"\\\"r
eleaseDate\\\": \\"2017-04-24\\\"", "name": "\\\"FrameworkPackage\\\"", "option
al": 0, "priority": "\\\"90\\\"", "framework": 1, "version": "\\\"1.3\\\"", "lo
cation": "\\\"C:\\\\\\\\Users\\\\\\\\Pieter-Jan\\\\\\\\Downloads\\\\\\\\o
pt2\\\"", "idPackage": -1, "type": "\\\"Executable\\\"", "description": "\\\"Pac
kage with framework\\\""}], "diskLocation": "C:\\\\\\Users\\\\\\\\Pieter-Jan\\\\\\Dow
nloads\\\\server", "installerVersion": "0.1", "name": "Deployment", "idInstalle
r": 0}}, 'sender': None}
FIELD DOCK -- Handling message
FIELD DOCK -- Performing action: RELEASE
Segmentatiefout
administrator@it3-pc07 /usr/local/lib/python2.7/dist-packages/client $
```

Figuur C.12: Error tijdens het installeren van een installer in Linux



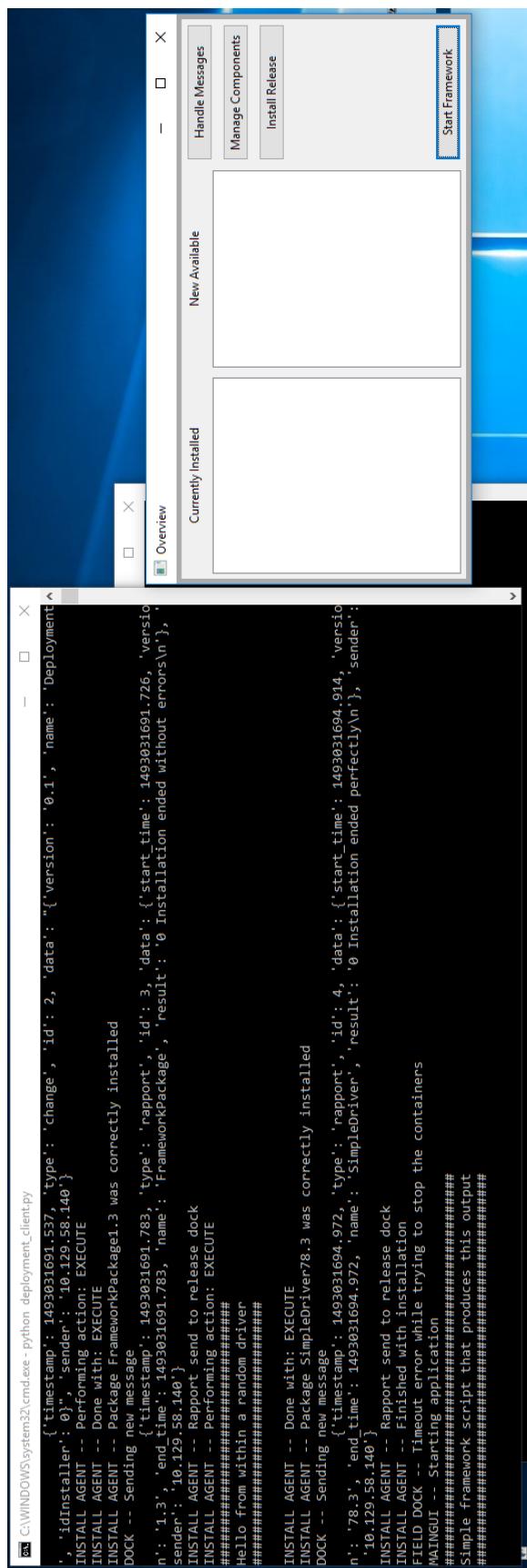
Figuur C.13: Installatie van een eerste installer



Figuur C.14: Overview van de clients na het verspreiden van de eerste installer

| | iddiagnosticsCheck | endResult | startTime | endTime | installer_idInstaller | package_idPackage | tower_idTower |
|---|--------------------|-------------------------------------|---------------------|---------------------|-----------------------|-------------------|---------------|
| ▶ | 1 | 0 Installation ended without errors | 2017-04-24 13:00:21 | 2017-04-24 13:00:21 | 1 | 2 | 5 |
| | 2 | 0 Installation ended perfectly | 2017-04-24 13:00:24 | 2017-04-24 13:00:24 | 1 | 1 | 5 |
| | 3 | 0 Installation ended without errors | 2017-04-24 13:01:31 | 2017-04-24 13:01:31 | 1 | 2 | 4 |
| | 4 | 0 Installation ended perfectly | 2017-04-24 13:01:34 | 2017-04-24 13:01:34 | 1 | 1 | 4 |
| | 5 | 0 Installation ended without errors | 2017-04-24 13:02:07 | 2017-04-24 13:02:07 | 1 | 2 | 6 |
| | 6 | 0 Installation ended perfectly | 2017-04-24 13:02:10 | 2017-04-24 13:02:10 | 1 | 1 | 6 |
| | 7 | 0 Installation ended without errors | 2017-04-24 13:02:18 | 2017-04-24 13:02:18 | 1 | 2 | 9 |
| | 8 | 0 Installation ended perfectly | 2017-04-24 13:02:21 | 2017-04-24 13:02:21 | 1 | 1 | 9 |
| | 9 | 0 Installation ended without errors | 2017-04-24 13:02:47 | 2017-04-24 13:02:47 | 1 | 2 | 7 |
| | 10 | 0 Installation ended perfectly | 2017-04-24 13:02:50 | 2017-04-24 13:02:50 | 1 | 1 | 7 |
| | 11 | 0 Installation ended without errors | 2017-04-24 13:02:56 | 2017-04-24 13:02:56 | 1 | 2 | 8 |
| | 12 | 0 Installation ended perfectly | 2017-04-24 13:02:59 | 2017-04-24 13:02:59 | 1 | 1 | 8 |
| | 13 | 0 Installation ended without errors | 2017-04-24 13:17:59 | 2017-04-24 13:17:59 | 2 | 2 | 9 |
| | 14 | 0 Installation ended perfectly | 2017-04-24 13:18:02 | 2017-04-24 13:18:02 | 2 | 1 | 9 |
| | 15 | 0 Installation ended without errors | 2017-04-24 13:18:17 | 2017-04-24 13:18:17 | 2 | 2 | 8 |
| | 16 | 0 Installation ended perfectly | 2017-04-24 13:18:20 | 2017-04-24 13:18:20 | 2 | 1 | 8 |
| | 17 | 0 Installation ended without errors | 2017-04-24 13:18:35 | 2017-04-24 13:18:35 | 2 | 2 | 5 |
| | 18 | 0 Installation ended perfectly | 2017-04-24 13:18:38 | 2017-04-24 13:18:38 | 2 | 1 | 5 |
| | 19 | 0 Installation ended without errors | 2017-04-24 13:18:51 | 2017-04-24 13:18:51 | 2 | 2 | 4 |
| | 20 | 0 Installation ended perfectly | 2017-04-24 13:18:54 | 2017-04-24 13:18:54 | 2 | 1 | 4 |
| | 21 | 0 Installation ended without errors | 2017-04-24 13:26:24 | 2017-04-24 13:26:24 | 2 | 2 | 9 |
| | 22 | 0 Installation ended perfectly | 2017-04-24 13:26:27 | 2017-04-24 13:26:27 | 2 | 1 | 9 |
| | 23 | 0 Installation ended without errors | 2017-04-24 13:26:34 | 2017-04-24 13:26:34 | 2 | 2 | 8 |
| | 24 | 0 Installation ended perfectly | 2017-04-24 13:26:37 | 2017-04-24 13:26:37 | 2 | 1 | 8 |
| | 25 | 0 Installation ended without errors | 2017-04-24 13:26:43 | 2017-04-24 13:26:43 | 2 | 2 | 9 |
| | 26 | 0 Installation ended perfectly | 2017-04-24 13:26:46 | 2017-04-24 13:26:46 | 2 | 1 | 9 |
| | 27 | 0 Installation ended without errors | 2017-04-24 13:26:52 | 2017-04-24 13:26:52 | 2 | 2 | 5 |
| | 28 | 0 Installation ended perfectly | 2017-04-24 13:26:54 | 2017-04-24 13:26:55 | 2 | 1 | 5 |
| | 29 | 0 Installation ended without errors | 2017-04-24 13:27:01 | 2017-04-24 13:27:01 | 2 | 2 | 8 |

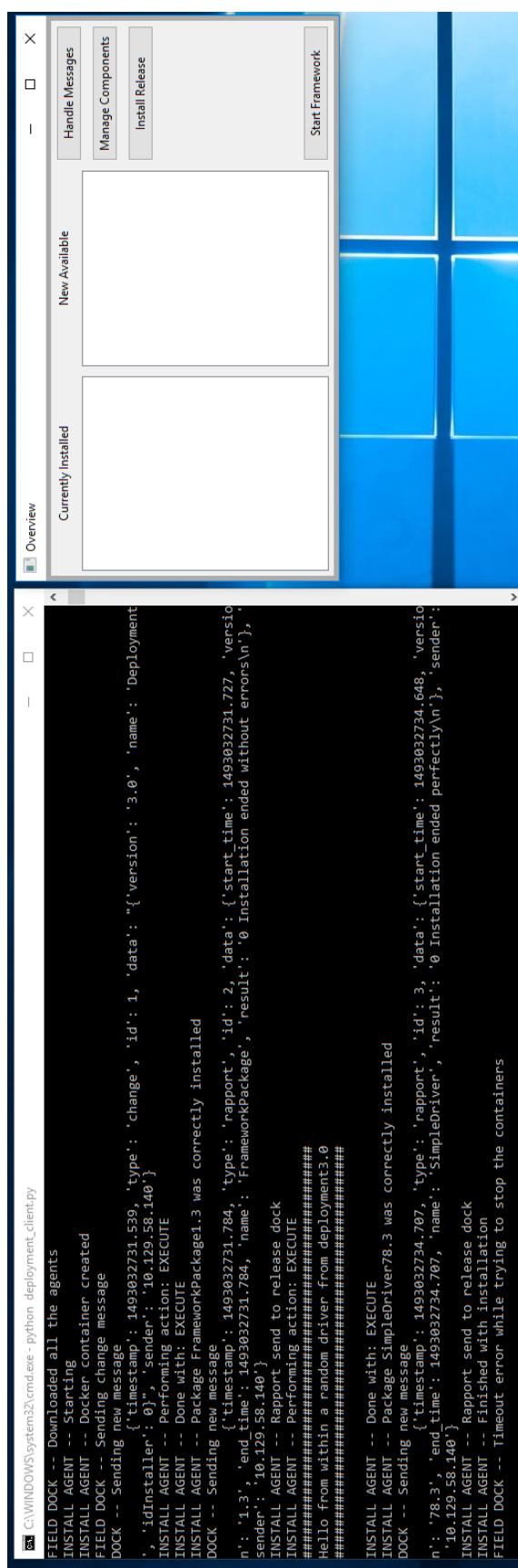
Figuur C.15: Rapport tabel na uitvoeren van een installatie



The screenshot shows a Windows command prompt window titled 'C:\WINDOWS\system32\cmd.exe - python deployment_client.py'. The window displays the output of a Python script named 'deployment_client.py'. The output is a series of log messages from a software framework, likely Rappor, detailing the installation and execution of various components. Key messages include:

- 'idinstaller': 0, 'timestamp': 1493031691.537, 'type': 'change', 'id': 2, 'data': {'version': '0.1', 'name': 'Deployment'}
- INSTALL AGENT -- Performing action: EXECUTE
- INSTALL AGENT -- Done with: EXECUTE
- INSTALL AGENT -- Package FrameworkPackage1.3 was correctly installed
- DOCK -- Sending new message
- { 'timestamp': 1493031691.783, 'type': 'rapport', 'id': 3, 'data': { 'start_time': 1493031691.726, 'version': '1.3', 'end_time': 1493031691.783, 'name': 'FrameworkPackage', 'result': 0 } Installation ended without errors.\n}, 'sender': '10.129.58.140' }
- INSTALL AGENT -- Report send to release dock
- INSTALL AGENT -- Performing action: EXECUTE
- ##### Performing action: EXECUTE
- Hello from within a random driver #####
- #####
- INSTALL AGENT -- Done with: EXECUTE
- INSTALL AGENT -- Package SimpleDriver78.3 was correctly installed
- DOCK -- Sending new message
- { 'timestamp': 1493031694.972, 'type': 'rapport', 'id': 4, 'data': { 'start_time': 1493031694.914, 'version': '1.3', 'end_time': 1493031694.972, 'name': 'SimpleDriver', 'result': 0 } Installation ended perfectly.\n}, 'sender': '10.129.58.140' }
- INSTALL AGENT -- Rapport send to release dock
- INSTALL AGENT -- Finished with installation
- FIELD DOCK -- Timeout error while trying to stop the containers
- MAINUI -- Starting application
- ##### Simple framework script that produces this output #####

Figuur C.16: Output na het opstarten van de geïnstalleerde software



Eigenvector C-17: Installatie van een tweede installeer

```

C:\Windows\system32\cmd.exe - python deployment-client.py
type \"\",\"ExecutableA\";\"Description\": \"Package with framework\\\"\",\"Deployment\": \"C:\\\\Users\\\\Pieter-Jan\\\\Downloads\\\\^
\\Server\",\"InstallerVersion\": \"3.0\",\"name\": \"Deployment\",\"idInstaller\": \"2\"},\"Sender\": \"None\"}
FIELD DOCK -- Handling message
FIELD DOCK -- Performing action: RELEASE
FIELD DOCK -- Release zip made
FIELD DOCK -- Received release, ready to install
FIELD DOCK -- Downloading agents
FIELD DOCK -- Download all the agents
INSTALL AGENT -- Starting
INSTALL AGENT -- Docker container created
FIELD DOCK -- Sending change message
DOCK -- Sending new message
DOCK -- Done with: EXECUTE
INSTALL AGENT -- Done with: EXECUTE
INSTALL AGENT -- Package FrameworkPackage1.3 was correctly installed
DOCK -- Sending new message
DOCK -- Done with: EXECUTE
INSTALL AGENT -- Done with: EXECUTE
INSTALL AGENT -- Package FrameworkPackage1.3 was correctly installed
time: 1493032715.658, name: 'FrameworkPackage', result: 0 Installation ended without errors\n}, 'sender': '10.129.48.21'}
INSTALL AGENT -- Report send to release dock
INSTALL AGENT -- Performing action: EXECUTE
#####
Hello from within a random driver from deployment3. 0
#####
INSTALL AGENT -- Done with: EXECUTE
INSTALL AGENT -- Package SimpleDriverv78.3 was correctly installed
DOCK -- Sending new message
DOCK -- Done with: EXECUTE
time: 1493032718.565, name: 'SimpleDriver', 'result': 0 Installation ended perfectly\n}, 'sender': '10.129.48.21'}
INSTALL AGENT -- Report send to release dock
INSTALL AGENT -- Finished with installation
FIELD DOCK -- Timeout error while trying to stop the containers
MATNGUI -- Starting application
#####
HELLO WORLD -- Deployment 3. 0
#####

```

Figuur C.18: Output na het opstarten van de geïnstalleerde software

| Alias | Geolocation | ID | Name | Serial Num... | Installer |
|---------------|-------------|----|----------|---------------|---------------|
| LinuxMachi... | Belgium | 1 | JJAK55 | APPROE | Non installed |
| LinuxMachi... | America | 2 | HAU885 | 123PL | Non installed |
| Windows1 | Belgium | 3 | PPOE | 788955SDD5 | Non installed |
| Windows2 | Belgium | 4 | POE99 | SN778 | Deployment3.0 |
| Windows3 | America | 5 | PMLLK77 | RVT85 | Deployment3.0 |
| Windows4 | Belgium | 6 | IUYT7458 | 4568LK | Deployment0.1 |
| Windows5 | Belgium | 7 | AYIKN | PM85569 | Deployment0.1 |
| Windows6 | Belgium | 8 | POHGH785 | 7855JK | Deployment3.0 |
| Windows7 | Belgium | 9 | CNVB785 | 78554QD | Deployment3.0 |

New Installer

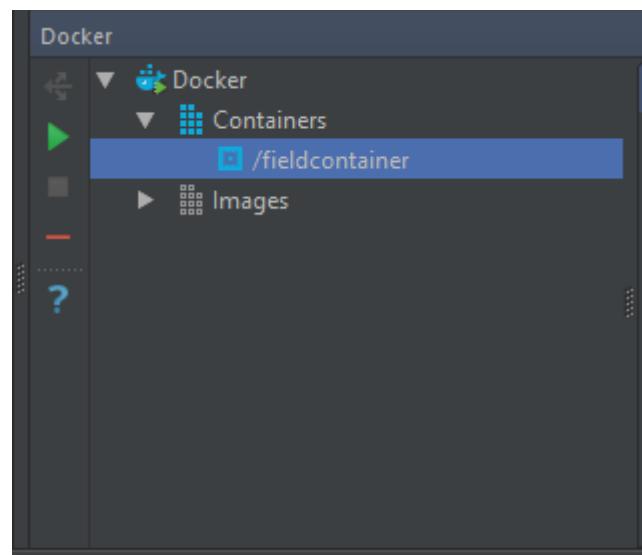
Release

Current Release

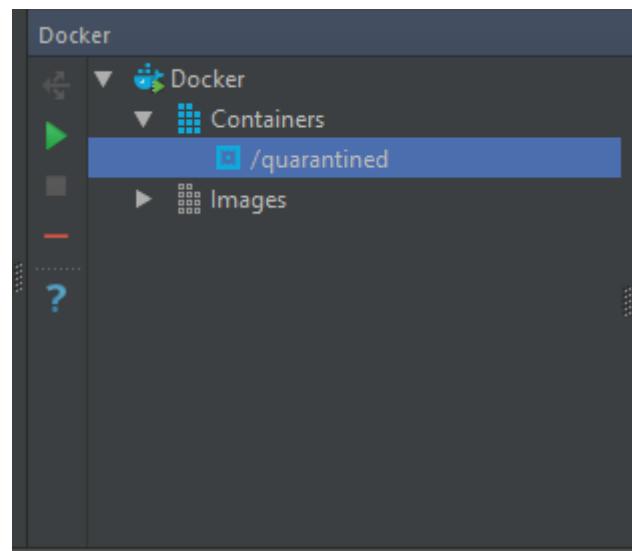
Installer: Deployment3.0

Figuur C.19: Overview van de clients na het verspreiden van een tweede installer

C.3 Slecht werkend pakket



Figuur C.20: Opsomming van aanwezige containers voor het installatieproces



Figuur C.21: Opsomming van aanwezige containers na het installatieproces

```

INSTALL AGENT -- Starting
AGENT -- API error: removing container with name old_container
INSTALL AGENT -- Docker container created
FIELD DOCK -- Sending change message
DOCK -- Sending new message
{'timestamp': 1493560280.572, 'type': 'change', 'id': 1, 'data': {'version': '0.1', 'name': 'FrameworkPackage1.3'}
INSTALL AGENT -- Performing action: EXECUTE
INSTALL AGENT -- Done With: EXECUTE
INSTALL AGENT -- Package FrameworkPackage1.3 was correctly installed
DOCK -- Sending new message
{'timestamp': 1493560280.95, 'type': 'rapport', 'id': 2, 'data': {'start_time': 1493560280.87,
INSTALL AGENT -- Rapport send to release dock
INSTALL AGENT -- Performing action: EXECUTE
#####
Hello from within a random driver
#####
INSTALL AGENT -- Done With: EXECUTE
INSTALL AGENT -- Package SimpleDriver78.3 ended with error
ERROR 1 No file created
DOCK -- Sending new message
{'timestamp': 1493560283.808, 'type': 'rapport', 'id': 3, 'data': {'start_time': 1493560283.73
INSTALL AGENT -- Rapport send to release dock
INSTALL AGENT -- Test failed, container will be quarantined in the end
INSTALL AGENT -- Performing action: EXECUTE
INSTALL AGENT -- Done With: EXECUTE
INSTALL AGENT -- Package FrameworkPackage1.3 was correctly installed
DOCK -- Sending new message
{'timestamp': 1493560286.799, 'type': 'rapport', 'id': 4, 'data': {'start_time': 1493560286.72
INSTALL AGENT -- Rapport send to release dock
INSTALL AGENT -- Not all tests finished correctly, rolling back to previous state
FIELD DOCK -- Sending change message
DOCK -- Sending new message
{'timestamp': 1493560289.584, 'type': 'change', 'id': 5, 'data': {'version': 'None', 'name': 'FrameworkPackage1.3'}

```

The diagram illustrates the flow of messages between three components: INSTALL AGENT, FIELD DOCK, and DOCK. The process starts with the INSTALL AGENT sending a 'change' message to the FIELD DOCK. The FIELD DOCK then sends a 'rapport' message to the DOCK. The DOCK sends a 'change' message back to the FIELD DOCK. A red arrow points from the 'rapport' message to the 'change' message, indicating a feedback loop or a specific sequence of events. The messages are timestamped and include details about package versions and names.

Figuur C.22: Output tijdens het installatieproces

C.4 Netwerk monitoring

| | | | | | |
|----|----------|---------------|---------------|-----|---|
| 12 | 9.293558 | 10.129.58.139 | 10.129.48.18 | TCP | 54 2282 → 54321 [ACK] Seq=1 Ack=1 Win=525568 Len=0 |
| 13 | 9.296550 | 10.129.48.18 | 10.129.58.139 | TCP | 60 54321 → 2282 [PSH, ACK] Seq=1 Ack=1 Win=525568 Len=5 |
| 14 | 9.296611 | 10.129.58.139 | 10.129.48.18 | TCP | 54 2282 → 54321 [ACK] Seq=1 Ack=6 Win=525568 Len=0 |
| 15 | 9.297112 | 10.129.58.139 | 10.129.48.18 | TCP | 58 2282 → 54321 [PSH, ACK] Seq=1 Ack=6 Win=525568 Len=4 |
| 16 | 9.297805 | 10.129.48.18 | 10.129.58.139 | TCP | 69 54321 → 2282 [PSH, ACK] Seq=6 Ack=5 Win=525568 Len=15 |
| 17 | 9.297843 | 10.129.58.139 | 10.129.48.18 | TCP | 54 2282 → 54321 [ACK] Seq=5 Ack=21 Win=525568 Len=0 |
| 18 | 9.297923 | 10.129.58.139 | 10.129.48.18 | TCP | 1088 2282 → 54321 [PSH, ACK] Seq=5 Ack=21 Win=525568 Len=1034 |

Figuur C.23: Doorsturen van een notificatie

| | | | | | | | |
|----|-----------|---------------|---------------|-----|------|--------------------------|------------------------------------|
| 21 | 11.891889 | 10.129.48.18 | 10.129.58.139 | TCP | 60 | 54186 → 12346 [ACK] | Seq=1 Ack=1 Win=52568 Len=0 |
| 22 | 11.894573 | 10.129.58.139 | 10.129.48.18 | TCP | 58 | 12346 → 54186 [PSH, ACK] | Seq=1 Ack=1 Win=65536 Len=4 |
| 23 | 11.894674 | 10.129.58.139 | 10.129.48.18 | TCP | 1514 | 12346 → 54186 [PSH, ACK] | Seq=5 Ack=1 Win=65536 Len=1460 |
| 24 | 11.894722 | 10.129.58.139 | 10.129.48.18 | TCP | 642 | 12346 → 54186 [PSH, ACK] | Seq=1465 Ack=1 Win=65536 Len=588 |
| 25 | 11.894800 | 10.129.58.139 | 10.129.48.18 | TCP | 1514 | 12346 → 54186 [PSH, ACK] | Seq=2053 Ack=1 Win=65536 Len=1460 |
| 26 | 11.894841 | 10.129.58.139 | 10.129.48.18 | TCP | 642 | 12346 → 54186 [PSH, ACK] | Seq=3513 Ack=1 Win=65536 Len=588 |
| 27 | 11.895701 | 10.129.48.18 | 10.129.58.139 | TCP | 60 | 54186 → 12346 [ACK] | Seq=1 Ack=3513 Win=525568 Len=0 |
| 28 | 11.895834 | 10.129.58.139 | 10.129.48.18 | TCP | 126 | 12346 → 54186 [PSH, ACK] | Seq=4101 Ack=1 Win=65536 Len=72 |
| 29 | 11.898418 | 10.129.48.18 | 10.129.58.139 | TCP | 60 | 54186 → 12346 [ACK] | Seq=1 Ack=4173 Win=524800 Len=0 |
| 30 | 11.898420 | 10.129.48.18 | 10.129.58.139 | TCP | 60 | 54186 → 12346 [PSH, ACK] | Seq=1 Ack=4173 Win=524800 Len=5 |
| 31 | 11.898588 | 10.129.58.139 | 10.129.48.18 | TCP | 54 | 12346 → 54186 [ACK] | Seq=4173 Ack=6 Win=65536 Len=0 |
| 32 | 11.902282 | 10.129.58.139 | 10.129.48.18 | TCP | 58 | 12346 → 54186 [PSH, ACK] | Seq=4173 Ack=6 Win=65536 Len=4 |
| 33 | 11.902813 | 10.129.48.18 | 10.129.58.139 | TCP | 69 | 54186 → 12346 [PSH, ACK] | Seq=6 Ack=4177 Win=524800 Len=15 |
| 34 | 11.902954 | 10.129.58.139 | 10.129.48.18 | TCP | 54 | 12346 → 54186 [ACK] | Seq=4177 Ack=21 Win=65536 Len=0 |
| 35 | 11.903091 | 10.129.58.139 | 10.129.48.18 | TCP | 1514 | 12346 → 54186 [ACK] | Seq=4177 Ack=21 Win=65536 Len=1460 |
| 36 | 11.903128 | 10.129.58.139 | 10.129.48.18 | TCP | 58 | 12346 → 54186 [PSH, ACK] | Seq=5637 Ack=21 Win=65536 Len=4 |

Figuur C.24: Doorsturen van een installer en agenten naar een field dock

| | | | | | | | | |
|----|-----------|---------------|---------------|-----|-----|-------|--------------------|---------------------------------|
| 59 | 15.047975 | 10.129.48.18 | 10.129.58.139 | TCP | 60 | 54188 | → 12347 [ACK] | Seq=157 Ack=22 Win=525568 Len=0 |
| 60 | 15.048028 | 10.129.58.139 | 10.129.48.18 | TCP | 59 | 12347 | → 54190 [PSH, ACK] | Seq=1 Ack=1 Win=65536 Len=5 |
| 61 | 15.048942 | 10.129.48.18 | 10.129.58.139 | TCP | 60 | 54190 | → 12347 [PSH, ACK] | Seq=1 Ack=6 Win=525568 Len=3 |
| 62 | 15.049091 | 10.129.58.139 | 10.129.48.18 | TCP | 54 | 12347 | → 54190 [ACK] | Seq=6 Ack=4 Win=65536 Len=0 |
| 63 | 15.049239 | 10.129.58.139 | 10.129.48.18 | TCP | 69 | 12347 | → 54190 [PSH, ACK] | Seq=6 Ack=4 Win=65536 Len=15 |
| 64 | 15.050059 | 10.129.48.18 | 10.129.58.139 | TCP | 302 | 54190 | → 12347 [PSH, ACK] | Seq=4 Ack=21 Win=525568 Len=248 |

Figuur C.25: Doorsturen van een rapport naar de broker

Statistics

| <u>Measurement</u> | <u>Captured</u> |
|------------------------|-----------------|
| Packets | 686 |
| Time span, s | 116.803 |
| Average pps | 5.9 |
| Average packet size, B | 135.5 |
| Bytes | 92618 |
| Average bytes/s | 792 |
| Average bits/s | 6343 |

Figuur C.26: Algemene statistieken bij het verzenden van een installer

Bijlage D

Beschrijving van deze masterproef in de vorm van een wetenschappelijk artikel

Design and development of a Python testing framework installer

Pieter-Jan Robrecht

May 7, 2017

Abstract

Televic Rail has developed a Python test framework for testing various products. The software, that needs to run on different platforms, uses multiple drivers and libraries. In order to test products correctly, the framework is often updated because of the release of a new driver library or to support new products. The installation process is time consuming, error prone and should therefore be automated. By automating this process, it becomes possible to collect information about the installation and update processes. The purpose of this article is to find an efficient solution and to develop a prototype. The prototype is divided in three components: a packager, a deployment server and a deployment environment. In the first part the packager is designed. This is responsible for the assembly of the software components. Phase two consists of the development of the deployment server. The server distributes all the installers and gathers information on the deployment environment. Lastly, the deployment environment is treated. In this isolated environment, installations and updates can be done safely. After a thorough evaluation, the first prototype design may be adjusted. The prototype will be expanded in the final stage to accommodate a reporting mechanism about installed versions, deployment status, etc. .

1 Introduction

Televic Rail, a Belgian company situated in Izegem, has more than 30 years of experience in the design and maintenance of on-board communication systems. This international company combines their knowledge and experience with a constant drive for innovation to deliver cutting-edge solutions for reliable communication in trains. At the same time, Televic rail designs several mechatronic sensors and safety control systems. Each device or system is designed to comply with the railway industry standards.

To meet the stringent safety standards, Televic has designed a Python testing framework to submit their products to several test scenarios. The framework was designed to be used on a powerful testing tower but has since been adapted to be usable on smaller computers as well.

Several hardware drivers and Python libraries are used to ensure a correct functionality of the Python testing framework. The amount of drivers and libraries is growing as the number of products Televic fabricates increases. This leads to a complex installation and update process which is time-consuming and error prone. In addition to the increasing number of drivers and libraries, there is also a growth in number of users. Because of these problems, there is an increasing demand for an application which facilitates the installation and update processes and provides a administration interface for client monitoring. The purpose of this work is to find a lasting solution for these problems and develop an application.

2 Architecture

[5, 8] describe the different stages in the deployment life cycle. Typical stages such as an installation, an update and an uninstallation are all present in modern day deployments. Each stage is equally important but each stage has different needs. Environment and manageability issues are the main issues that causes a difficult deployment process [6]. Architectures and technologies such as software dock architecture, ORYA, Ansible, Electric Cloud,... were researched in order to select the most versatile. Out of these architectures and technologies, the software dock architecture was selected because it offers the most scalability and flexibility. In the following section, the various components of the architecture are discussed.

2.1 Software Dock

As previously discussed, it is necessary to support the different needs of all stages of the software life cycle together with the growing amount of users. To support these needs, the software dock architecture was selected. [7, 8] discuss the software dock architecture: an agent based architecture with publish/subscribe communication between the different entities. Figure 1 shows a representation of the architecture. The architecture consists of three major components: the release dock, the field dock and the event service. [7] describes how a publish/subscribe architecture is used in the event service to enable scalable communication between the different docks.

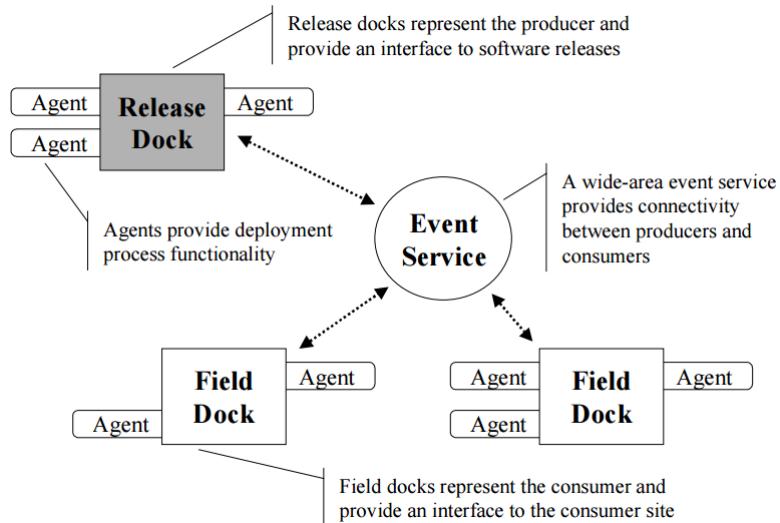


Figure 1: Software Dock Architecture [8]

The release dock is a software component running at the software producer. It contains a software repository that users can use to select the software to deploy. A Deployable Software Description (DSD) file is used to create a semantic description of the software. The DSD file together with several agents will be linked to a release [8].

When software is marked for release, the software will be transferred from the release dock to the field dock together with the DSD file and several agents. These agents use the DSD file to perform a part in the deployment process. Agents are capable of subscribing to different events. Every change to the release causes the triggering of an event which in turn will trigger the appropriate agents [8].

The field dock provides information about the resources and configuration of the user system. With this information, it is possible to build a context in which the release are deployed. When software is being released, the proper agents will dock themselves in the field dock and use the interface provided to install the release. This agent based approach and the use of an interface to describe the users ensures that each release can be personalized [8].

An important part of the software dock architecture is the event service. This deals with communication between the different docks and is a pivotal component in the architecture. By using a publish/subscribe architecture, it is possible for brokers to guide incoming messages to the right recipient. [10, 4] describe how the event-service might be implemented.

2.2 Software packaging

The Python testing framework consists of several software libraries and hardware drivers. Each library and driver could be seen as a separate package. Libraries and drivers will typically be written in different programming languages and will be handled differently. [3] states that the use of extra software enables the programmer to bridge the gap between different technologies. Various technologies such as Qt Installer framework, Chocolatey, NSIS, ... can be used to combine packages into one installer which can then be used to install every package. To implement the packer functionality, a system based on the Qt installer is designed. Other software packaging solutions (such as NSIS and WiX Toolset) also provide a solution for combining software packages. The Qt installer framework is chosen because of the better separation of software packages. This gives a better overview of what is present in an installer and which installation scripts from the meta folder affect the data.

2.3 Docker

As mentioned previously, environment and manageability issues are the main causes for an error prone deployment process. Different strategies can be used to counter these issues. Rollback strategies might be implemented to negate the negative effects of a bad installation. [11] discusses several strategies. Some strategies are more laborious to realize than others. [5] suggests virtualization as an alternative.

With Docker, an alternative virtualization technology, it is possible to create small containers in which the installation take place. A comparison between the normal virtualization architecture and the Docker architecture is visible in Figure 2. [9] states that the resource use of Docker is lower in comparison with virtual machines.

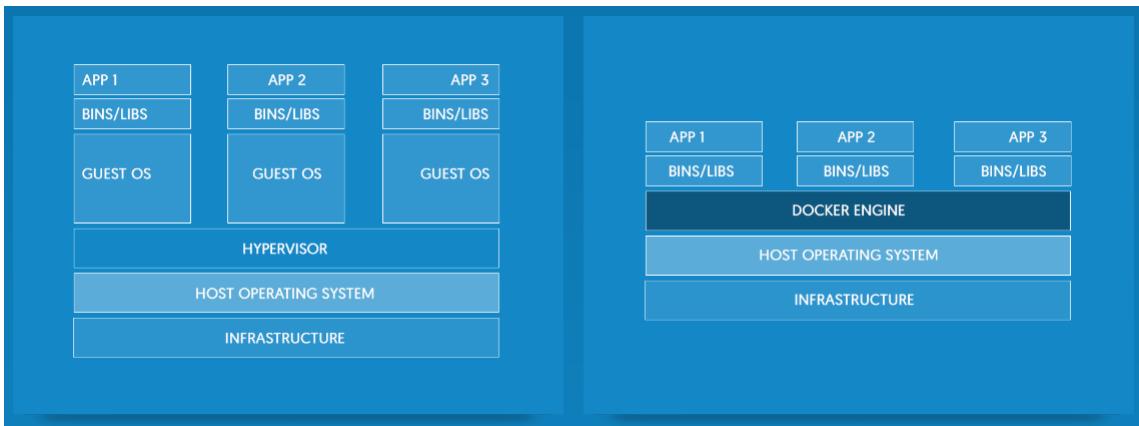


Figure 2: Comparison between VM's and Docker [1]

3 Implementation

Current architectures and technologies do not meet expectations and provide no conclusive solution to all Televic issues. Most technologies offer a partial solution. In order to solve the problems, an

architecture is designed that consists of a combination of the discussed architectures and technologies.

The final design of the application consists out of the four parts: a deployment server, a packager, a database and a deployment client. The software dock architecture is used as architecture. By using this architecture, it is possible to combine the deployment server with the packager and the database into a release dock and deploy the deployment client as a field dock. The use of the software dock architecture provides a scalable architecture which allows an increase of users and software producers. Both the release dock and the field dock should not track which docks are present in the network as that is the broker's task.

Using a similar structure as the Qt installer framework, a packager is designed that can handle different types of software packages. The Qt installer framework itself is not enough as it is not possible to create an installer that works on both Linux and Windows on one operating system. Since the packager itself is designed, it is possible to choose any programming language. Therefore, a language that is fully operating system-independent is chosen. By combining the packager with the software dock architecture, it is possible to personalize the installation of each software package separately together with the personalization of each step in the deployment process. The packager is included in the release dock code and is used to produce installers.

A final element of the architecture is the deployment environment that will consist of the field dock. The purpose of the deployment environment is to create a secure environment in which the installer of the release dock can be installed and updated. As already indicated, the installation process and update process are error prone and a solution must be found for this. This will be solved by using Docker containers. Information about the success or failure will be made available to the release dock by using a database. This database will store all important information about the client system and installation processes.

4 Evaluation

4.1 Strength

The use of the software dock architecture provides a scalable and flexible application that supports the continuous increase of user and software packages. The designed application provides a good basis for Televic to expand.

In addition, Docker provides a safe installation environment. This causes errors during the installation process to have no effect on the system and a working version of the software will always be available.

4.2 Weaknesses

A weakness of the architecture lies with the broker. As the broker has to handle all messages, it forms a bottleneck in the architecture. However, there are solutions where the broker returns a handle to the docks [2]. Another strategy is to use multiple brokers.

Security is also creates a weakness. The sending and receiving of messages is currently not safe because all messages are sent unencrypted. This can be easily solved by adding cryptographic capabilities to the application.

The main weakness consist of further completing the application. In addition to the installation process, it must also be possible to perform an update process. There are already some methods available that can be used.

4.3 Opportunities

With the designed architecture, it is possible to deploy different software to a group of software users. This opens up the opportunity for Televic to collaborate with different software vendors to distribute software between software users.

4.4 Threats

A threat to the application lies with the use of Docker. In case of problems with the Docker Python module or Docker for Windows, Televic is dependent on a separate group of developers who should resolve these issues as quickly as possible.

4.5 Tests

Several tests were conducted to test the capabilities of the prototype. Different scenarios were conceived to check whether the designed methods for software distribution and installation were viable. Test were run to check whether the prototype could be used on multiple operating systems. These test have shown that the designed methods are viable but also that several functionalities are not yet supported.

5 Conclusion

The purpose of this thesis was to design a solution for the complex installation and update processes. In addition, it must support an increasing number of user and software packages. It was important to provide a solution that is both scalable and flexible.

To solve this problem, the software dock architecture is used. By combining this architecture with a software packager whose architecture is based on the Qt installer framework and docker, it was possible to provide a flexible and scalable solution. The implementation of the design provides a good starting application that Televic can easily expand and adapt.

Several tests were run to test the capabilities of the prototype. These tests show that the designed methods for software distribution and installation are a viable solution for the problem.

References

- [1] Docker Main Page. <https://www.docker.com/>, 2016. [Online; consulted 16-08-2016].
- [2] Richard M Adler. Distributed coordination models for client/server computing. *Computer*, 28(4):14–22, 1995.
- [3] John R Callahan. Software packaging. Technical report, 1998.
- [4] Antonio Carzaniga, David S Rosenblum, and Alexander L Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS)*, 19(3):332–383, 2001.
- [5] A. Dearie. Software deployment, past, present and future. In *Future of Software Engineering, 2007. FOSE '07*, pages 269–284, May 2007.
- [6] Eelco Dolstra. *The purely functional software deployment model*. Utrecht University, 2006.
- [7] Richard S Hall, Dennis Heimbigner, Andre Van Der Hoek, and Alexander 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*, pages 269–278. IEEE, 1997.
- [8] Richard S Hall, Dennis Heimbigner, and Alexander L Wolf. A cooperative approach to support software deployment using the software dock. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 174–183. IEEE, 1999.
- [9] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.
- [10] Peter R Pietzuch and Jean M Bacon. Hermes: A distributed event-based middleware architecture. In *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*, pages 611–618. IEEE, 2002.
- [11] Sudarshan M Srinivasan, Srikanth Kandula, Christopher R Andrews, Yuanyuan Zhou, et al. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference, General Track*, pages 29–44. Boston, MA, USA, 2004.

Bijlage E

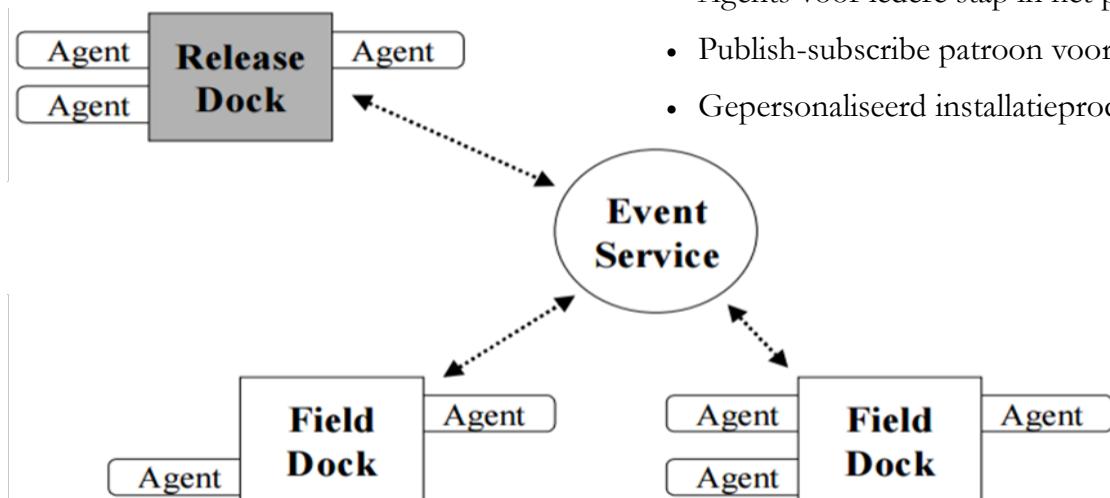
Poster

Ontwerp en ontwikkeling van een testraamwerk installer

Doel

- Installatieproces vereenvoudigen
- Schaalbare oplossing zoeken
- Client monitoring
- Prototype ontwerpen

Architectuur



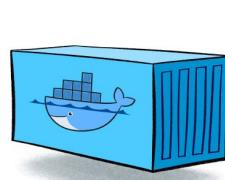
Release Dock



Server-side

- Software packaging
- Installer creatie
- Remote field dock monitoring

Field Dock



Client-side

- Docker containers
- Veilige deployment omgeving

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



LID VAN
**ASSOCIATIE
KU LEUVEN**