

Beheer van aanpasbare softwarediensten in cloud- en netwerkomgevingen

Management of Customizable Software-as-a-Service
in Cloud and Network Environments

Hendrik Moens

Promotoren: prof. dr. ir. F. De Turck, prof. dr. ir. B. Dhoedt
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen

Vakgroep Informatietechnologie
Voorzitter: prof. dr. ir. D. De Zutter
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2014 - 2015



ISBN 978-90-8578-807-2
NUR 986, 988
Wettelijk depot: D/2015/10.500/51



Universiteit Gent
Faculteit Ingenieurswetenschappen en Architectuur
Vakgroep Informatietechnologie

Promotoren: prof. dr. ir. Filip De Turck
prof. dr. ir. Bart Dhoedt

Universiteit Gent
Faculteit Ingenieurswetenschappen en Architectuur
Vakgroep Informatietechnologie
Gaston Crommenlaan 8 bus 201, B-9050 Gent, België
Tel.: +32-9-331.49.00
Fax.: +32-9-331.48.99



Dit werk kwam tot stand in het kader van een
specialisatiebeurs van het IWT-Vlaanderen
(Instituut voor de aanmoediging van Innovatie door
Wetenschap en Technologie in Vlaanderen).



Proefschrift tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen:
Computerwetenschappen
Academiejaar 2014-2015

Dankwoord

Toen ik in 1990 voor het eerst met mijn kleine rugzakje naar de kleuterschool vertrok en huilend aan de schoolpoort achterbleef denk ik niet dat iemand gedacht had dat ik 25 jaar later nog steeds aan het studeren zou zijn om nog een allerlaatste diploma te halen. Nuja, studeren... De laatste vijf jaar als doctoraatsstudent waren natuurlijk een heel stuk anders dan de jaren daarvoor. Het leven als doctoraatsstudent verschilt heel wat van dat van de gewone student: geen lessen en huiswerk, maar wel korte cursussen, conferenties, publicaties, les geven en projectwerk. Toch blijf je nog de hele tijd vaardigheden en kennis vergaren. En, net als voor al die diploma's daarvoor, is er om af te sluiten toch nog één allerlaatste examen.

Een doctoraatsboek is een individueel werk, maar dat wil niet zeggen dat al het werk dat in dat boek terechtkomt ook volledig alleen uitgevoerd werd. Veel van de ideeën in dit boek zijn dan ook het resultaat van lange discussies en intensieve samenwerkingen met verscheidene collega's, die ik bij deze dus ook zeer graag wil bedanken. Vooraleerst wil ik mijn promotoren, Filip De Turck en Bart Dhoedt bedanken voor al hun hulp, begeleiding en het vertrouwen dat ze in mij gesteld hebben tijdens mijn doctoraat. Filip wil ik bedanken voor de vele interessante discussies en omdat hij altijd klaarstond om te helpen om problemen op te lossen, pakweg wanneer je een week voor je boek indient ontdekt dat je eigenlijk al een tijdje niet meer als doctoraatsstudent ingeschreven bent¹. Ook buiten het werk kon ik op hem rekenen: toen mijn vrienden mijn vrijgezellendag (of eerder verlengd vrijgezellenweekeind) organiseerden, hielp hij mijn maandag vrij te houden door mijn agenda te blokkeren (al heb ik er geen idee van hoe het kan dat ik het niet op zijn minst verdacht vond dat hij proactief en een paar weken op voorhand een meeting om een CNSM paper te bespreken in mijn agenda vastlegde) en nam hij zelfs de tijd om een kort filmpje op te nemen voor de gelegenheid. Ook op Bart kon ik steeds rekenen voor feedback, interessante discussies, of een korte babbel.

Een doctoraat doen kost tijd en geld. Ik ben dan ook het instituut voor Innovatie door Wetenschap en Technologie (IWT) dankbaar voor de financiële ondersteuning voor mijn werk. Dankzij mijn IWT specialisatiebeurs kon ik mij concentreren op het inhoudelijke werk, zonder mij zorgen te moeten maken over jaarlijkse contractverlengingen.

Binnen IBCN waren er naast mijn promotoren nog verscheidene mensen met wie ik vaak samenwerkte en die op hun manier bijgedragen hebben aan mijn

¹Ik raad iedereen die zijn boek wil indienen aan om op *voorhand* eens op Plato te checken of de link "mijn doctoraat" er tussen staat. Het kan je veel last-minute stress besparen.

doctoraat. Om te beginnen wil ik hier Femke De Backere en Kristof Steurbaut bedanken voor hun goede begeleiding van mijn masterthesis. De eerste keren dat ik feedback kreeg op stukken tekst, was er op de bladzijden die ik terugkreeg meer rood dan zwart te zien. Ik heb misschien wel een andere weg ingeslagen met het onderwerp van mijn doctoraat, maar zonder jullie zou ik mogelijk nooit aan een doctoraat begonnen zijn en van jullie heb ik veel bijgeleerd over het schrijven van goed gestructureerde teksten. Daarnaast wil ik Steven Latré en Jeroen Famaey bedanken, die mij in het begin op weg geholpen hebben, mij geholpen hebben om een IWT specialisatiebeurs te behalen en mij begeleid hebben bij mijn eerste publicaties. Verder wil ik ook Pieter-Jan Maenhaut en Maryam Barshan bedanken voor de samenwerking bij verscheidene projecten. Via verschillende industrieprojecten, CUSTOMSS, PUMA en MECaNO werkte ik ook samen met verscheidene mensen van andere universiteiten en bedrijven, die mij hielpen om vanuit een ander standpunt naar verschillende problemen te kijken. Graag wil ik dan ook Stefan Walraven, Eddy Truyen, Bert Lagaisse, Dimitri Van Landuyt en Wouter Joosen van KU Leuven, Marino Verheye van Televic Healthcare en Koen Handekeyn van up-nxt bedanken voor hun input tijdens deze projecten die bijgedragen hebben aan dit boek.

IBCN heeft een zeer sterke aanwezigheid binnen de network management community, waardoor we dus vaak met velen op conferentie gingen. Ik wil daarom dan ook onze (intussen in sommige gevallen ex-)UGent IM-NOMS-CNSM-club bedanken voor het aangename gezelschap op de vele edities van deze conferenties op de verschillende al dan niet exotische locaties. Filip De Turck, Steven Latré, Jeroen Famaey, Pieter-Jan Maenhaut, Niels Bouten, Maxim Claeys, Stefano Petrangeli, Thomas Vanhove, Matthias Strobbe, Jeroen van der Hooft en Tim De Pauw: bedankt voor het steeds aangename gezelschap, zowel tijdens de uren van de conferentie, als bij de mojitos in Maui, de caipirinhas in Rio de Janeiro, de Guinness in Dublin en de pintjes in Gent (en de vele andere minder exotische locaties) buiten de conferentie-uren.

Daarnaast waren er nog een hele hoop andere interessante en onverwachte conferentie-ervaringen: een speech van Obama in het midden van Dublin, waardoor alle straten potdicht zaten en waardoor niet iedereen plaats had in het gereserveerde hotel, plotse hagelbuien in Firenze in september, een voetbalmatch op het strand in Rio en herten die vanuit het niets voor de auto springen in Canada (en waar gelukkig op tijd voor gestopt kon worden)².

Gedurende mijn doctoraat heb ik ook mogen proeven van het onderwijs, wat ik een heel toffe ervaring vond en wat het werk een stuk gevarieerder maakte. Het begeleiden van practica heeft mij geholpen om vlotter en zelfverzekerder voor grote groepen te staan. Ik wil dan ook de teams van de verschillende vakken bedanken voor de aangename samenwerking. Voor het vak informatica is dit na 5 jaar onder leiding van Bart Dhoedt toch wel een hele lijst geworden (waarbij ik hoop dat ik niemand vergeten ben): Vincent Sercu, Tom Van Haute, Bram Gadeyne, Thijs

²In tegenstelling tot hoe dit misschien overkomt kan tevens toch bevestigen dat er ook nog (regelmatig) serieus gewerkt wordt op conferenties.

Walcarius, Steven Bohez, Cedric De Boom, Ine Melckenbeeck, Piet Smet, Dieter De Witte, Samuel Dauwe, Tom Van Haute, Nick Vercammen, Sofie Demeyer, Pieter Becue, An De Moor, Sofie Demeyer en Olivier Van Laere. Van het vak softwareontwikkeling, ook onder leiding van Bart, herinner ik mij vooral de fijne samenwerking bij de jaarlijkse begeleiding van de vele projectgroepen met Stijn Verstichel. Daarnaast waren er van dit vak ook tal van practica, met ook een hele reeks mede-begeleiders: Stijn Verstichel, Steven Van Canneyt, Steven Bohez, Cedric De Boom, Tim Verbelen, Cedric De Boom, Dieter Plaetinck en Olivier Van Laere. Verder waren er ook nog twee vakken onder leiding van Filip De Turck: ODS, waar ik samen met Tim Verbelen en Jeroen Famaey enkele kleine projectjes begeleidde en PDS waarvoor ik het cloud practicum maakte en dat samen met Femke De Backere begeleidde.

Wie mij goed kent weet dat ik een hekel heb aan administratie en procedures. Ik zou daarom Martine Buysse en Davinia Stevens willen bedanken voor alle hulp bij het navigeren van het administratieve doolhof van UGent – iMinds en het oplossen van de hieruit voorkomende problemen³. Ook wil ik hierbij de mensen van financiën, Bernadette Becue, Karien Hemelsoen, Nathalie Vanhijfte, Joke Stalens en Dalila Lauwers, bedanken voor de ondersteuning met de financiële kant van de administratie. Een heel ander soort administratie wordt verzorgd door onze (ex-)admins, die ik eveneens dankbaar ben voor hun snelle respons wanneer er ergens problemen waren: Bert De Vuyst, Joeri Casteels, Simon Roberts, Bert De Knijf, Serge van Ginderachter, Jonathan Moreel, Wouter Adem, Pascal Vandeputte en Johan De Knijf. Om van de kantoren van IBCN een aangename werkplek te maken, is het natuurlijk ook belangrijk dat ze proper gehouden worden. Daarvoor wil ik dan ook graag Sandra en Sabine bedanken.

Een goede werksfeer en toffe collega's maken het werk een stuk aangener. Ik wil daarom dan ook graag mijn bureagenoten en ex-bureaugenoten bedanken voor de aangename sfeer en de verscheidene gezellige bureauactiviteiten: Wannes Kerckhove, Thomas Dupont, Femke De Backere, Femke Ongenae, Gregory Van Seghbroek, Bert Vankeirsbilck, Jonas Anseeuw, Ali Farhan Azmat, Maryam Barshan, Rafael Xavier dos Santos, Jeroen Famaey, Samuel Dauwe, Lien Deboosere en Pol Dockx. Hierbij wil ik ook Thomas nog eens specifiek en expliciet bedanken voor het afdrukken van de eerste editie van dit boek op de dag van de deadline toen mijn PC weer eens besliste dat hij een andere taal spreekt dan de printer.

De boog kan natuurlijk niet altijd gespannen staan. Af en toe eens ontspannen met een goede barbecue, een whisky tasting of een trappist tour, waarbij we op veel te korte tijd alle trappistenabdijen bezochten (toen dat nog op twee dagen mogelijk was) is goed voor de motivatie en inspiratie. Ik wil dan ook Tom De Nies, Christophe Billiet, Nicholas Overloop en David Verhasselt bedanken voor al deze toffe momenten. Daarnaast wens ik Tom en Christophe ook veel succes met hun eigen doctoraat! Verder wil ik ook mijn ouders en familie bedanken. Zonder jullie

³ Zoals bijvoorbeeld het niet formeel ingeschreven meer zijn voor een doctoraat de week voor het indienen ervan.

steun, zorgen en vertrouwen, zou ik het waarschijnlijk nooit aangedurfd hebben om informatica te beginnen studeren, waardoor ik hier nooit geraakt zou zijn.

Om af te sluiten wil ik ook de belangrijkste persoon bedanken die mij doorheen mijn hele doctoraat gesteund heeft. Ze begon als mijn vriendin, werd in de loop van mijn doctoraat mijn verloofde en intussen is ze mijn vrouw. Isabelle, je was er altijd om mij te steunen als ik het even niet zag zitten. Je hebt mij altijd geholpen en goed verzorgd als er weer eens één of andere deadline aan zat te komen door op die momenten een veel groter stuk van het huishouden op jou te nemen en hebt af en toe actief geholpen door je door de bergen tekst heen te worstelen en ze grondig na te lezen... Bedankt voor al je hulp en ondersteuning. Nu mijn doctoraat ten einde is, is het mijn beurt om jou in de komende jaren te ondersteunen bij je eigen doctoraatsonderzoek. Ik hoop dat ik dat voor jou even goed zal kunnen doen als jij dat voor mij deed!

Gent, juni 2015
Hendrik Moens

Table of Contents

Dankwoord	i
Samenvatting	xxvii
Summary	xxxii
1 Introduction	1
1.1 Everything as a service	2
1.2 Problem statement	7
1.3 Definitions & terminology	8
1.4 Research contributions	9
1.5 Outline of this dissertation	11
1.6 Publications	14
1.6.1 Publications in international journals (listed in the Science Citation Index)	14
1.6.2 Publications in international conferences (listed in the Science Citation Index)	15
1.6.3 Publications in other international conferences	15
References	18
2 Feature-Based Application Development and Management of Multi-Tenant Applications in Clouds	19
2.1 Introduction	20
2.2 Related work	21
2.3 Software Product Line Engineering	22
2.4 SaaS multi-tenancy approaches	24
2.4.1 Application development, deployment and management	27
2.4.2 Application-Based Binary (ABB) applications	27
2.4.3 Feature-Based Binary (FBB) application development	28
2.4.4 FBB application deployment	30
2.4.5 FBB application management	31
2.4.6 Hybrid multi-tenancy	32
2.5 Analysis	33
2.5.1 ABB instance count	34
2.5.2 FBB instance count	35

2.5.3	Hybrid instance count	35
2.5.4	Worst-case comparison	36
2.6	Evaluation results	38
2.7	Discussion	40
2.8	Conclusions	41
	References	42
3	Cost-Effective Feature Placement of Customizable Multi-Tenant Applications in the Cloud	45
3.1	Introduction	46
3.2	Related work	49
3.2.1	Software Product Line Engineering	49
3.2.2	Application Placement	49
3.3	Feature placement concepts	51
3.4	Formal problem description	56
3.4.1	Optimization objective	56
3.4.2	Input variables	58
3.4.3	Decision variables	61
3.4.4	Auxiliary variables	61
3.4.5	Constraint details	62
3.4.5.1	Feature-based constraints	62
3.4.5.2	Application resource requirement constraints	63
3.4.5.3	Resource constraints	63
3.4.5.4	Application provisioning constraints	64
3.4.5.5	Cascading failure of features	64
3.4.5.6	Server usage constraints	65
3.5	Solution techniques	66
3.5.1	Integer Linear Programming (ILP)	66
3.5.2	Heuristic algorithms	66
3.5.2.1	Feature ordering	70
3.5.2.2	Grouping strategies	71
3.5.2.3	Server ordering	71
3.5.2.4	Feature model conversion	72
3.5.2.5	Heuristic algorithms	72
3.6	Evaluation setup details	72
3.6.1	CUSTOMSS problem model	73
3.6.2	Generated problem models	75
3.6.3	Evaluation methodology	76
3.6.3.1	Load-based evaluation	76
3.6.3.2	Execution time evaluations	77
3.6.3.3	Quality evaluation metrics	77
3.7	Evaluation results	77
3.7.1	Degree of multi-tenancy	77
3.7.2	Impact of the model constraints	79
3.7.3	Placement quality	82

3.7.3.1	Generated problem models	82
3.7.3.2	CUSTOMSS problem model	85
3.7.4	Execution speed evaluation	85
3.8	Conclusions	87
	References	89
4	Allocating Resources for Customizable Multi-Tenant Applications in Clouds Using Dynamic Feature Placement	95
4.1	Introduction	96
4.2	Related Work	98
4.3	Feature Placement	101
4.4	Feature Placement Model	103
4.4.1	Dynamic feature placement: resource migrations	106
4.4.2	Iterative migration minimizing model	109
4.5	The Dynamic Feature Placement Algorithm	109
4.5.1	Placing applications	111
4.5.1.1	The place function	112
4.5.1.2	Choosing the best feature selection	114
4.5.1.3	Allocating resources	116
4.5.1.4	Server selection	117
4.5.2	Refining placements	119
4.6	Evaluation Setup	120
4.6.1	Evaluated algorithms	120
4.6.2	Simulation parameters	120
4.7	Evaluation Results	122
4.7.1	Underloaded datacenter	122
4.7.2	Overloaded datacenter	124
4.8	Conclusions	127
	References	128
5	Developing and Managing Customizable Software as a Service Using Feature Model Conversion	133
5.1	Introduction	134
5.2	Related work	135
5.3	Variable Software as a Service development	136
5.4	System architecture overview	139
5.4.1	Development platform	139
5.4.2	Execution platform	140
5.4.3	Execution platform components	140
5.4.4	Role of feature models in the application architecture	141
5.5	Feature model conversion	142
5.5.1	Removal of configuration changes	144
5.5.2	Feature elimination	144
5.5.3	Feature expansion	145
5.5.4	Feature model conversion algorithm	147

5.6	Evaluation Results	147
5.7	Conclusions	150
	References	155
6	Shared Resource Network-Aware Impact Determination Algorithms for Service Workflow Deployment	157
6.1	Introduction	158
6.2	Related work	161
6.3	Network-Aware Impact Determination	162
6.3.1	Base model: determining service workflow network impact	165
6.3.2	Non-workflow service graphs	168
6.4	Shared Resource NAID	169
6.4.1	Adding resource sharing: network edge sharing	169
6.4.2	Adding resource sharing: server capacity sharing	172
6.5	Runtime conflict management	173
6.5.1	Adding quality levels: class-aware NAID	173
6.5.2	Workflow addition algorithm	174
6.5.2.1	Addition algorithm termination	177
6.6	Evaluation approach	177
6.6.1	Medical communications use case	178
6.6.2	Generated use case	179
6.6.3	Simulation approach	180
6.7	Evaluation	181
6.7.1	Quality	181
6.7.1.1	Medical communication use case	181
6.7.1.2	Generated use case	185
6.7.2	Execution speed	187
6.7.3	Algorithm scalability	188
6.8	Conclusions	190
	References	192
7	Customizable Function Chains: Managing Service Chain Variability in Hybrid NFV Networks	195
7.1	Introduction	196
7.2	Related Work	198
7.3	Network Functions Virtualization (NFV) resource management architecture	199
7.4	Modeling Customizable Function Chains (CFCs) and Network Functions (NFs)	201
7.5	Function chain resource allocation	204
7.5.1	General model parameters	205
7.5.2	Service Function Chain Placement (SFC-P)	206
7.5.2.1	Resource management constraints	207
7.5.2.2	Network constraints	208
7.5.2.3	Optimization objective	209

7.5.3	Customizable Function Chain Placement (CFC-P)	210
7.5.3.1	Network constraints	212
7.5.3.2	Optimization objective	212
7.6	Evaluation Setup	213
7.7	Evaluation Results	218
7.7.1	Total cost comparison	218
7.7.2	Server use costs	221
7.7.3	Time-limited heuristics	222
7.8	Conclusions	222
	References	225
8	Conclusion	229
8.1	Contributions	229
8.1.1	An approach for modeling and managing customizability of multi-tenant Software-as-a-Service (SaaS) applications	229
8.1.2	A resource allocation approach for managing customizable SaaS applications within datacenters	230
8.1.3	Network-aware modeling and management algorithms for inter-cloud network environments	231
8.2	Future perspectives	231
8.2.1	Advanced variability modeling	232
8.2.2	Federated management of customizable SaaS	232
8.2.3	Resilient management of customizable SaaS	233
8.2.4	Streamlining the configuration process	233
8.2.5	Containerization	234
8.2.6	Managing the Internet of Things	234
A	Design and Evaluation of a Hierarchical Application Placement Algorithm in Large Scale Clouds	235
A.1	Introduction	236
A.2	Related work	237
A.3	System Architecture	238
A.4	Formal Problem Description	240
A.5	Hierarchical management	241
A.5.1	Hierarchical Management Structure	241
A.5.2	Algorithm Details	242
A.5.2.1	Resource availability aggregation	243
A.5.2.2	Demand decoupling	245
A.5.2.3	Overview	246
A.6	Evaluation Results	246
A.7	Conclusion	252
	References	254

B	Migrating Legacy Software to the Cloud: Approach and Verification by means of Two Medical Software Use Cases	257
B.1	Introduction	258
B.2	Related Work	259
B.3	Migration Strategy	261
B.3.1	Cloud Migration	262
B.3.1.1	Selecting Components	262
B.3.1.2	Determining Provider Compatibility	263
B.3.1.3	Determining Impact on Client Network	263
B.3.1.4	Scaling the Application	264
B.3.2	Multi-Tenancy	264
B.3.2.1	Decoupling Databases	264
B.3.2.2	Adding Tenant Configuration Database	265
B.3.2.3	Providing Tenant Configuration Interface	266
B.3.2.4	Dynamic Feature Selection	267
B.3.2.5	Managing Tenant Data, Users and Roles	267
B.3.2.6	Mitigating Security Risks	267
B.4	Case Study 1: Medical Communications System	268
B.4.1	Introduction	268
B.4.2	Cloud migration	269
B.4.2.1	Selecting Components	269
B.4.2.2	Determining Provider Compatibility	270
B.4.2.3	Determining Impact on Client Network	272
B.4.2.4	Scaling the Application	272
B.4.3	Multi-Tenancy	272
B.4.3.1	Decoupling Databases	272
B.4.3.2	Adding Tenant Configuration Database	273
B.4.3.3	Providing Tenant Configuration Interface	274
B.4.3.4	Dynamic Feature Selection	274
B.4.3.5	Managing Tenant Data, Users and Roles	275
B.4.3.6	Mitigating Security Risks	275
B.5	Case Study 2: Medical Appointments Schedule Planner	276
B.5.1	Introduction	276
B.5.2	Cloud Migration	277
B.5.2.1	Selecting Components	277
B.5.2.2	Determining Provider Compatibility	277
B.5.2.3	Determining Impact on Client Network	279
B.5.2.4	Scaling the Application	280
B.5.3	Multi-Tenancy	280
B.5.3.1	Decoupling Databases	281
B.5.3.2	Adding Tenant Configuration Database	281
B.5.3.3	Providing Tenant Configuration Interface	281
B.5.3.4	Dynamic Feature Selection	281
B.5.3.5	Managing Tenant Data, Users and Roles	282
B.5.3.6	Mitigating Security Risks	282

B.6	Discussion and Evaluation	282
B.6.1	Increased flexibility and elasticity	283
B.6.2	Decreased deployment time	283
B.6.3	Ease of Maintenance	284
B.6.4	Migration Cost	285
B.6.5	Remaining Risks	285
B.6.6	Change in Cost Model	286
B.6.7	Performance Comparison	287
B.7	Conclusions	290
	References	291

List of Figures

1.1	The three cloud service models defined by the National Institute of Standards and Technology (NIST): Infrastructure-as-a-Service (IaaS), Software-as-a-Service (SaaS) and Platform-as-a-Service (PaaS).	3
1.2	Estimated evolution of datacenter workloads.	4
1.3	Multi-tenancy can be achieved at multiple levels, resulting in differing granularity.	5
1.4	Schematic position of the different chapters in this dissertation. . .	13
2.1	An illustration of a hierarchically structured feature model.	23
2.2	An illustrative example comparing the ABB and a FBB multi-tenancy approaches.	26
2.3	The processes for developing and deploying multi-tenant feature-based SaaS applications using the ABB and FBB approaches. . .	27
2.4	An illustrative example of a development and runtime feature model.	29
2.5	The FBB approach management system. The feature placement algorithm is responsible for allocating resources for the various application instances.	31
2.6	By adding specific application customizations as separate instances to the feature model, a hybrid approach using both FBB and ABB approaches can be used.	33
2.7	The worst case feature model for the ABB approach resulting in the maximum possible instance type count.	36
2.8	The Maximum number of Instance Types (MIT) for the ABB and FBB approaches for three commercial applications, <i>MC</i> , <i>DP</i> and <i>MDM</i> , and a composed model containing all three models, <i>Full</i> . . .	39
2.9	The median of MIT for the ABB and FBB approaches for varying numbers of features.	39
2.10	The distribution of MIT for the ABB approach for varying numbers of features.	40
3.1	A schematic representation of the feature placement problem. . . .	48
3.2	A feature model fragment for a medical data processing application.	52

3.3	Features are associated with code modules. Applications containing the features are created by instantiating these features and linking them together.	53
3.4	Different feature model selections for two applications.	54
3.5	A detailed overview of the feature placement inputs and outputs. . .	54
3.6	The combined feature model containing the CUSTOMSS applications. Each entry in this model corresponds to a feature in one of three real-life applications.	74
3.7	The maximum number of tenants sharing instances for an application-based approach, where variants are created as monolithic instances, and a feature-based approach, where applications are composed from feature instances.	78
3.8	Performance of ILPs, ILPrp and ILPrpl using different quality evaluation metrics using the Varying Costs (VC) scenario.	81
3.9	Performance of the heuristics and the ILP algorithm for the different scenarios.	82
3.10	Percentiles for the performance of the heuristic and ILP algorithms for the VC scenario.	83
3.11	The quality of feature placement as a function of the number of servers and applications.	84
3.12	The execution speed of the feature placement algorithm as a function of varying application counts, server counts and feature counts. .	86
4.1	An illustration of a scenario where the application is offered to end users by a hierarchy of the three types of tenants: resellers, clients, and client departments.	97
4.2	The dynamic feature placement, its inputs and its function within a management system.	98
4.3	An illustrative example of a small feature model using the Pure::Variants notation.	102
4.4	An overview of the relevant components within the cloud management system and the communication flow between the components. .	103
4.5	The input and output of the feature placement.	104
4.6	An illustration of the difference between resource shift migrations and instance count change migrations.	107
4.7	The different optimization steps of the migration minimizing model. .	109
4.8	A high level overview of the Dynamic Feature Placement Algorithm (DFPA) steps.	110
4.9	A high level overview of the placeAll function which is responsible for allocating all unplaced applications.	111
4.10	A high level overview of the place function which is responsible for allocating a single application.	112
4.11	An illustration of how TR values can be determined for an application a using CPU and memory resources.	115
4.12	The piecewise linear function used to determine the quality of a fit. .	118

4.13	Total placement cost in an underloaded datacenter.	122
4.14	The number of instance migrations for every iteration of the dynamic placement algorithms in an underloaded datacenter.	123
4.15	The amount of CPU resources that are moved between instances in subsequent algorithm invocations in an underloaded datacenter. . .	123
4.16	Total placement cost in an overloaded datacenter.	125
4.17	The number of instance migrations for every iteration of the dynamic placement algorithms in an overloaded datacenter.	126
4.18	The amount of CPU resources that are moved between instances in subsequent algorithm invocations in an overloaded datacenter. . .	126
5.1	Runtime resource allocation for applications based on a logical feature model.	135
5.2	Linking code modules and configuration to features.	139
5.3	The system architecture.	139
5.4	A detailed overview of the components in the execution platform. .	141
5.5	Elimination of empty mandatory features.	144
5.6	Elimination of empty or features.	145
5.7	Feature expansion when it is applied to a subtree.	146
5.8	The feature model of a Medical Communication application in the development stage, after expansion and in the runtime stage. . . .	148
5.9	The features present in the models of the Document Processing (DP), Medical Information Management (MIM) and Medical Communications (MC) applications in the different feature expansion phases.	149
6.1	An illustration of the impact on a client network of service instantiations, migrations and changes.	159
6.2	The Network-Aware Impact Determination (NAID) problem as a flow network.	163
6.3	An illustration of the effect of adding workflows on the guaranteed network share of workflows.	164
6.4	An illustration of the Shared Resource Network-Aware Impact Determination (SRNAID) problem input and output.	165
6.5	Hierarchical capacity sharing between edges ensures more capacity is available for individual workflows than physically present. . . .	170
6.6	The network set-up used in the evaluation environment. The environment models two buildings with multiple floors and a collection of end-user devices on every floor.	178
6.7	<i>EqualOF</i> : Fail rates using SRNAID in groups with varying overprovisioning factors.	182
6.8	<i>FixedRoot</i> : Fail rates using SRNAID in groups with varying overprovisioning factors and fixed root overprovisioning.	183
6.9	<i>Flat</i> : Fail rates using SRNAID without hierarchies.	183

6.10	Required bandwidth for the three discussed SRNAID approaches with varying overprovisioning factors.	184
6.11	The required total network bandwidth for varying OF parameters in the generated scenario.	185
6.12	Failure rates of Data Intensive (DI) workflows in the DI, Server Intensive (SI) and balanced scenarios using a heavily underprovisioned network.	186
6.13	The execution speed of the SRNAID algorithm for varying network edge capacities.	187
6.14	The execution speed of the SRNAID algorithm compared to the execution speed of the SR SPLIT algorithm.	189
7.1	An NFV burst scenario: physical hardware is fully utilized by a base load, while spillover is handled by utilizing on-demand Virtual Network Functions (VNFs).	197
7.2	A high level architectural overview of how a hybrid NFV resource management system can be constructed.	200
7.3	Network asset allocation approaches.	202
7.4	An illustration of a feature model containing six features.	203
7.5	An illustration of how hybrid NFV scenarios can be modeled using CFCs.	204
7.6	Connectivity service feature model and associated NFs.	213
7.7	An illustration of the various possible connectivity service CFC configurations.	214
7.8	The evaluation network contains a backbone network and multiple edge nodes that are interconnected using switches.	217
7.9	Total failure cost comparison for increasing network and service demand.	218
7.10	Server use cost comparison in network-constrained evaluation scenarios.	219
7.11	A breakdown of the failures occurring during the evaluation scenario by CFC and failure type.	220
7.12	Server use cost comparison of in an underloaded scenario containing only SampledDPI and StrictFullDPI CFCs in the pure NFV network.	221
7.13	Total cost comparison of time-limited CFC-P in the hybrid and pure NFV networks.	223
A.1	The system components on management and execution servers . . .	239
A.2	Solving overutilization by splitting a node and promoting a child node (grey) to peer status.	243
A.3	Solving underutilization by removing a node and distributing its children (grey) amongst the node's peers.	243

A.4	The origin and destination of the different <i>applace</i> in- and outputs. A single management server, containing the <i>applace</i> -function, aggregation and decoupling mechanisms is shown.	246
A.5	The quality of the allocation after subsequent placement calculations.	248
A.6	Execution time of the hierarchical and centralized allocation strategies with varying server and application counts.	249
A.7	Comparison of the average satisfied demand of the different allocation strategies.	249
A.8	Illustration of the management overhead and performance penalties induced by the hierarchy with a very low branching factor.	250
A.9	Illustration of the management overhead and performance penalties with a higher branching factor.	251
A.10	Execution time of the hierarchical and centralized allocation strategies with fixed application counts.	251
A.11	The maximum number of applications known per node at different management levels.	252
B.1	An overview of the different cloud service models used in cloud computing.	260
B.2	A summary of the different steps required to migrate an existing application to the cloud, and to add multi-tenancy to the application.	262
B.3	An illustrative example of the possible communication between components after migration to a hybrid cloud.	263
B.4	Possible architecture of the application after decoupling the databases.	265
B.5	Possible architecture of the application after adding the tenant configuration interface.	266
B.6	An example architecture of a nurse call system, with the communication between the different elements when a patient makes a call.	269
B.7	Architecture of the application before and after migration to the cloud and adding support for multi-tenancy.	271
B.8	An overview of the possible communication between actors and the different components of the medical communications system. .	273
B.9	Pre-migration single-tenant architecture of the medical appointments schedule planner.	276
B.10	Revised architecture of the schedule planner after adding multi-tenancy to the application and migration to the public cloud. . . .	280
B.11	A comparison of the average page generation times for 3 test pages over 50 iterations.	288
B.12	A comparison of the average end-to-end transaction times for 3 test pages over 50 iterations.	289

List of Tables

1.1	Comparison of the different types of multi-tenancy for 1500 tenants with each a single end user.	6
1.2	An overview of the contributions (Section 1.4) per chapter in this dissertation.	12
3.1	Graphical representation of feature models, description of relations, and formal representation. The nodes on the left are parent features, those on the right are child features.	51
3.2	The different symbols used in Section 3.4.	57
3.3	Conversion of relations in the feature model \mathcal{F} to constraints. . . .	63
3.4	The different functions used in Section 3.5.2.	66
3.5	The costs for the different evaluation scenarios.	76
4.1	An overview of the relation between the DFPA introduced in this chapter and previous work.	101
4.2	An analysis of the distribution of the total cost of the algorithms. .	125
5.1	Graphical representation of feature models, description of relations, and formal representation.	137
5.1	Conversion of feature model relations to logical statements.	138
5.2	Properties of the different feature model types.	143
5.3	Execution speed of the different operations.	150
5.4	The execution time of the feature placement algorithm for different models. Times are measured in seconds.	150
6.1	An illustration of how various service graphs can be represented within the flow-based model.	169
6.2	Medical Communications (MC) evaluation scenario workflows. .	179
6.3	Parameters used in the generated evaluation scenario workflows. .	180
7.1	The relation types used to structure feature models.	202
7.2	Model parameters: network and service specifications.	205
7.3	Model parameters: SFC-P.	206
7.4	Model parameters: CFC-P.	210
7.5	The constraints associated with the various feature model relations.	211
7.6	Resource and network impacts in the connectivity service scenario.	215

7.7	Evaluation CFCs.	215
7.8	Evaluation network hardware parameters.	217
7.9	Evaluation VNF specification.	217
A.1	Symbols	240
B.1	Overview of standard instances on Windows Azure.	271
B.2	Overview of the available T2 instance types on Amazon EC2.	278
B.3	Overview of the available M3 instance types on Amazon EC2.	278
B.4	Overview of the most important changes for migration to Google AppEngine.	279
B.5	Initial configuration of new tenant before migration: time estimation for the MC software case study.	283
B.6	Initial configuration of new tenant after migration: time estimation for the MC software case study.	284
B.7	Initial deployment after migration: time estimation for the MC software case study.	284
B.8	Tasks to be done for a production ready application in the cloud: time estimation for the MC software case study.. . . .	286
B.9	An overview of the different deployment environments. The mentioned cost values are valid at the time this appendix was submitted as an article.	287
B.10	Average page generation times (in seconds) and standard deviations for 3 test pages over 50 iterations.	288
B.11	Average end-to-end transaction times (in seconds) and standard deviations for 3 test pages over 50 iterations.	288

List of Acronyms

A

ABB	Application-Based Binary
AC	Application Costs
AOP	Aspect-Oriented Programming
API	Application Programming Interface

B

BPaaS	Business-Process-as-a-Service
-------	-------------------------------

C

CAPP	Cloud Application Placement Problem
CA	Cheapest Application
CFC-P	Customizable Function Chain Placement
CFC	Customizable Function Chain
CFO	Cost Feature Order
CS100	Cheapest Shortened 100
CS10	Cheapest Shortened 10
CS	Cheapest Shortened

D

DCaaS	Datacenter-as-a-Service
DFPA	Dynamic Feature Placement Algorithm
DI	Data Intensive
DPI	Deep Packet Inspection
DP	Document Processing

F

FBB	Feature-Based Binary
FC	Feature Costs

G

GA	Genetic Algorithm
----	-------------------

I

ICB	Instance Count Based
IC	Identical Costs
IDE	Integrated Development Environment
ILP	Integer Linear Programming
IaaS	Infrastructure-as-a-Service
IoT	Internet of Things

L

LLDP	Link Layer Discovery Protocol
LP	Linear Programming

M

MC	Medical Communications
MIM	Medical Information Management
MIT	Maximum number of Instance Types

N

NAID	Network-Aware Impact Determination
NC	Nurse Call
NDP	Neighbor Discovery Protocol

NF-P	Network Function Placement
NFV	Network Functions Virtualization
NF	Network Function
NIST	National Institute of Standards and Technology
NRDs	Cost of Non-Realized Demand Simple
NRD	Cost of Non-Realized Demand
NaaS	Network-as-a-Service

P

PNFD	Physical Network Function Descriptor
PNF	Physical Network Function
PSO	Particle Swarm Optimization
PaaS	Platform-as-a-Service
PoC	Proof of Concept
pCA	prepared Cheapest Application
pCS	prepared Cheapest Server

Q

QoS	Quality of Service
-----	--------------------

R

rps	requests per second
-----	---------------------

S

SDN	Software-Defined Networking
SFC-P	Service Function Chain Placement
SFC	Service Function Chain
SI	Server Intensive
SLA	Service Level Agreement
SOA	Service-Oriented Architecture
SPLE	Software Product Line Engineering
SRNAID	Shared Resource Network-Aware Impact Determination
SaaS	Software-as-a-Service

U

UC Utilization Cost

V

VC Varying Costs
VM Virtual Machine
VNF-P Virtual Network Function Placement
VNFD Virtualised Network Function Descriptor
VNFFGD VNF Forwarding Graph Descriptor
VNF Virtual Network Function
VoIP Voice over IP

W

WAN Wide Area Network

X

XaaS Everything-as-a-Service

Samenvatting

– Summary in Dutch –

Het gebruik van *cloud computing* is de laatste jaren sterk toegenomen. Cloud computing is een technologie die het mogelijk maakt om toepassingen en diensten op aanvraag aan te bieden aan meerdere eindgebruikers. Hierdoor kunnen virtuele computersystemen, softwareontwikkelingsplatformen en toepassingen verhuurd worden, waarbij typisch enkel de systeembronnen die gebruikt worden betaald moeten worden. Dit komt de flexibiliteit van toepassingen ten goede en maakt het mogelijk om snel systeembronnen toe te voegen en te verwijderen naarmate de belasting op de toepassing verandert. Deze cloud toepassingen en diensten worden gehuisvest in een rekencentrum, waarvan de infrastructuur en toepassingen gedeeld worden tussen meerdere gebruikers. De schaalvoordelen die hierdoor bekomen worden zorgen ervoor dat deze diensten aan een lagere kost geleverd kunnen worden. Om de voordelen van cloud computing maximaal te benutten, moeten softwarediensten ontwikkeld worden zodat systeembronnen tussen verschillende gebruikers gedeeld kunnen worden. In de praktijk betekent dit dat één enkele instantie van de toepassing gedeeld wordt tussen verschillende gebruikers, wat ervoor zorgt dat toepassingen meestal slechts beperkt aanpasbaar en configureerbaar zijn: veel aanpassingen zorgen er immers voor dat wijzigingen aan de code van de toepassing nodig zijn, waardoor meerdere instanties van de toepassing nodig zijn, wat er dan weer toe leidt dat systeembronnen niet meer gedeeld kunnen worden over verschillende gebruikers heen.

Voor de meeste toepassingen die een zeer grote hoeveelheid gebruikers aanspreken is dit een zeer goed werkend distributiemodel. Door een beperkte set goed gekozen configuratieopties aan te bieden kan aan de noden van de meerderheid van de klanten voldaan worden. Individuele klanten kunnen in deze situatie slechts weinig eisen stellen, gezien ze elk slechts een zeer klein aandeel van de inkomsten representeren. Het gebrek aan aanpasbaarheid is echter voor sommige toepassingsdomeinen een struikelblok dat een migratie naar de cloud blokkeert. Dit is bijvoorbeeld het geval wanneer een toepassing niet aan een grote hoeveelheid eindgebruikers aangeboden wordt, maar eerder verhuurd wordt aan een veel kleinere hoeveelheid organisaties, die elk een groep gebruikers kunnen hebben. Dit zorgt ervoor dat iedere individuele huurder belangrijker wordt en dus meer eisen kan stellen, wat er toe leidt dat de toepassingen vaak aangepast worden aan hun specifieke noden. Voor deze sterk aanpasbare toepassingen heeft een migratie naar de cloud slechts beperkt nut, aangezien de nodige aanpassingen voor iedere huurder

er toe leiden dat aparte instanties van de toepassing nodig zijn, waardoor geen systeembronnen gedeeld kunnen worden tussen verschillende huurders.

Dit proefschrift heeft tot doel het ontwerpen en bestuderen van technologieën en beheertechnieken die gebruikt kunnen worden om aanpasbare cloudtoepassingen te ontwerpen en beheren. Met dergelijke technieken wordt het mogelijk om systeembronnen te delen over meerdere huurders heen en zo kosten uit te sparen. Hierdoor kunnen toepassingen naar de cloud gemigreerd worden en kunnen nieuwe aanpasbare toepassingen ontwikkeld worden. Om dit te bereiken werden drie uitdagingen beschouwd:

1. Een manier om aanpasbare softwarediensten te ontwerpen en modelleren moet ontwikkeld worden. Deze aanpak moet het delen van systeembronnen tussen verschillende huurders ondersteunen en ervoor zorgen dat deze cloudtoepassingen toch zeer aanpasbaar zijn.
2. Eens er manieren zijn om aanpasbare softwarediensten te ontwikkelen, wordt het nuttig om te onderzoeken hoe deze toepassingen binnen een rekencentrum beheerd kunnen worden.
3. Gezien de softwarediensten over een netwerk aangeboden worden, is het nodig om de invloed van het netwerk op deze toepassingen te beschouwen, evenals hoe de ontwikkelde beheertechnieken binnen netwerken gebruikt kunnen worden.

Vooraleerst werd bestudeerd op welke manier gedeelde aanpasbare softwarediensten ontwikkeld kunnen worden. Hierbij werden verschillende manieren bestudeerd om de tegenstrijdigheid tussen de aanpasbaarheid van toepassingen en het delen van systeembronnen tussen huurders op te lossen. Als eerste stap werd bekeken hoe de variabiliteit van de toepassingen gemodelleerd kan worden. Daarna werd een naïeve aanpak, waarbij verschillende instanties van de toepassing gegenereerd worden die elk andere aanpassingen bevatten, vergeleken met een oplossing waarbij toepassingen opgebouwd worden uit gedeelde componenten. Bij de componentgebaseerde oplossing verzorgt iedere component een specifieke functionaliteit van de toepassing en worden verschillende variaties van de toepassing bekomen door dynamisch verschillende componenten aaneen te rijgen. Elk van deze componenten kan door verschillende gebruikers gedeeld worden, wat het delen van systeembronnen tussen huurders mogelijk maakt. Deze componentgebaseerde oplossing heeft als voordeel dat systeembronnen beter gedeeld kunnen worden over de verschillende huurders heen: indien twee toepassingen op sommige vlakken vergelijkbare functionaliteit aanbieden, dan kunnen ze de voor die functionaliteit gebruikte componenten delen, ook al zijn de toepassingen op andere vlakken niet identiek.

Omdat toepassingen in deze aanpak bestaan uit meerdere gedeelde componenten die individuele functies van de toepassing realiseren, wordt het nuttig om te bestuderen hoe systeembronnen op een optimale manier aan deze componenten toegewezen kunnen worden binnen een cloudomgeving. Daarom werden in een

volgende stap verschillende functieplaatsingsalgoritmen ontwikkeld, die bepalen waar de verschillende componenten waaruit de toepassingen samengesteld zijn, geplaatst kunnen worden binnen een rekencentrum. Deze algoritmen houden rekening met de variabiliteit van toepassingen en kunnen daardoor intelligentere beslissingen nemen dan standaard cloudbeheeralgoritmen: enerzijds kan de hoeveelheid systeembronnen die nodig is om de toepassing aan te bieden verminderd worden door rekening te houden met alternatieve manieren om toepassingen aan te bieden en anderzijds kan variabiliteitsinformatie in rekening gebracht worden indien er een tijdelijk tekort is aan systeembronnen, om de kwaliteit van toepassingen maximaal te behouden en een minimale dienstverlening te garanderen.

Als laatste werden softwarediensten binnen volledige netwerken beschouwd, wat het mogelijk maakt om rekening te houden met de toestand van het netwerk dat klanten met het rekencentrum verbindt en om diensten doorheen het hele netwerk te verspreiden. Om te beginnen werd hierbij bestudeerd hoe de netwerkimpact van softwarediensten bepaald kan worden. Dit is nuttig wanneer de toepassing gebruik maakt van bij de klant geïnstalleerde toestellen en maakt het mogelijk na te gaan of er voldoende netwerk en service capaciteit is voordat diensten uitgerold of aangepast worden. Daarnaast werd ook bekeken hoe aanpasbare softwarediensten aangeboden kunnen worden in netwerkomgevingen. Hierbij werden de concepten die gebruikt werden bij de ontwikkeling van de functieplaatsingsalgoritmen toegepast binnen netwerken. Omdat deze netwerken vaker een grotere heterogeniteit hebben dan de netwerken binnen een rekencentrum, werd de componentgebaseerde aanpak hiervoor uitgebreid zodat niet alleen gedeelde virtuele diensten, maar ook fysieke toestellen en door de klant aangeleverde virtuele machines gebruikt kunnen worden als component.

Met de in dit proefschrift voorgestelde oplossingen, wordt het dus mogelijk om zeer aanpasbare softwarediensten aan te bieden gebruik makende van cloud-technologieën waarbij systeembronnen gedeeld worden tussen verschillende huurders. Dit maakt het mogelijk om bestaande op maat aangepaste toepassingen naar de cloud te migreren en om nieuwe soorten aanpasbare softwarediensten te ontwikkelen en aan te bieden. Hierdoor kunnen aanbieders van deze diensten de kostenvoordelen en toegenomen flexibiliteit die voortkomen uit het gebruik van cloudinfrastructuur benutten.

Summary

In recent years, the use of cloud computing has increased significantly. Cloud computing technologies can be used to offer on-demand applications and services to multiple end users. This allows service providers to offer virtualized computer systems, software platforms and applications to their users, who typically only have to pay for the resources they use. This increases application flexibility, and allows users to quickly add and remove resources as the application demand fluctuates. Cloud resources are hosted in datacenters, where they are shared between a large number of users. This allows service providers to exploit economies of scale, making it possible to offer these resources at a relatively low cost. To fully benefit from the advantages of cloud computing, software must be developed in a way that ensures that resources can be shared between multiple end users. In practice, a single application instance is therefore usually shared between a large number of end users. This however has the adverse effect of limiting application configurability and customizability: many changes would require modifications to the application code, which means these changes could result in the creation of separate application binaries for the applications. This would lead to the creation of multiple separate application instances for different users, preventing the effective sharing of resources.

For various applications which are developed for a large number of end users, this distribution model can work very well. By providing a limited set of well-chosen configuration options, the needs of the majority of customers can be met. Meeting the demands of individual customers with specific requirements is in this case not needed, and would not be profitable as each customer only represents a small share of the total revenue. However, this approach does not work for every type of application. In some domains, the limitation of application customization is a roadblock preventing a migration to the cloud. This is, for example, the case when applications are not sold to a large number of end users, but instead to a smaller set of tenants that each have a large number of end users. In such a scenario, an individual tenant becomes much more important, allowing some tenants to demand a highly customized application which is tailored to their needs. For these highly customizable applications, a cloud migration only results in limited advantages, as customizations result in separate application instances that can not be shared between tenants.

This dissertation aims to investigate technologies and management techniques that can be used to design and manage customizable multi-tenant Software-as-a-Service (SaaS) applications. These techniques allow the sharing of resources among

tenants, the migration of existing customizable applications to multi-tenant cloud environments, and the development of novel customizable SaaS applications and services. To achieve this, three challenges were considered:

1. An approach for designing and modeling multi-tenant customizable SaaS is needed. This approach should support the sharing of resources between tenants, while still providing high customizability.
2. Once customizable SaaS can be modeled, it becomes useful to study how customizable applications can be managed within a datacenter.
3. SaaS is offered to end users over a network. Therefore, it is important to investigate the impact of the various applications on the network, and how the management techniques developed in this dissertation can be used within network environments.

To tackle the above challenges, approaches for designing and modeling customizable SaaS were first developed. To achieve this, a methodology for modeling the customizability of applications was studied. Then, three alternative approaches for resolving the conflict between software customization and multi-tenancy were studied: a naive approach, which generates multiple application binaries based on the customizations, was compared to an approach where applications are composed out of multi-tenant components and a hybrid approach combining properties of both. In the component-based approach, every component realizes specific features that the application can provide, and customized applications are built by dynamically chaining together different components. Every individual component can be shared between multiple applications, enabling multi-tenancy. This component-based approach has the advantage of better sharing system resources over different tenants, as applications can share resources when they share individual components, even if they are not identical when it comes to other functionalities.

In this approach, applications are built out of shared components, each providing part of the application features. Thus, it is useful to study how resources can be assigned optimally to these shared components within a cloud datacenter. Therefore, feature placement algorithms, which determine where the components the applications are composed of are allocated within a datacenter, were developed. These algorithms take the application customizability into account, enabling them to make more intelligent decisions than standard cloud resource management algorithms: when sufficient resources are available, the amount of system resources needed to provision applications can be reduced by taking into account alternative ways to provide applications, ensuring resources are shared as much as possible, while when there are insufficient resources customization information can be utilized to maintain quality as well as possible, providing a minimal service.

Finally, customizable SaaS services were considered within networks, making it possible to take the network connecting the cloud datacenter to clients into account when services are deployed, and to spread application components throughout the network. First, an approach for determining the network impact of SaaS services

was detailed. This is for example useful when applications make use of client-side hardware devices, and makes checking whether there is sufficient network and service capacity before services are deployed possible. In addition, an approach for offering customizable SaaS in network environments is studied. Here, the concepts previously used to develop feature placement algorithms for resource allocation in datacenters are applied within network environments. As these networks are more heterogeneous than the networks within a single datacenter, the component-based approach was extended to support different types of components. In addition to using virtualized shared services, physical devices and virtual machines provided by clients can be used as application components.

The solutions presented in this dissertation enable the development of highly customizable SaaS applications, making use of cloud technologies, where system resources are shared between multiple tenants. This allows the migration of custom tailored applications to the cloud, and the development of novel customizable SaaS offerings built using cloud technologies, making it possible for service providers to take advantage of the cost benefits and increased flexibility that result from the use of cloud infrastructure and technologies.

1

Introduction

“The interesting thing about cloud computing is that we’ve redefined cloud computing to include everything that we already do”

–Larry Ellison (Oracle CEO, 2008)

In common use, “cloud computing” has grown to become synonymous with offering applications and services over the Internet. This has stretched the concept to encompass many types of applications, making cloud computing omnipresent in IT news in recent years. While this has diluted the meaning of the word cloud, a clear definition of it does in fact exist: the National Institute of Standards and Technology (NIST) defines the cloud computing as follows [1]:

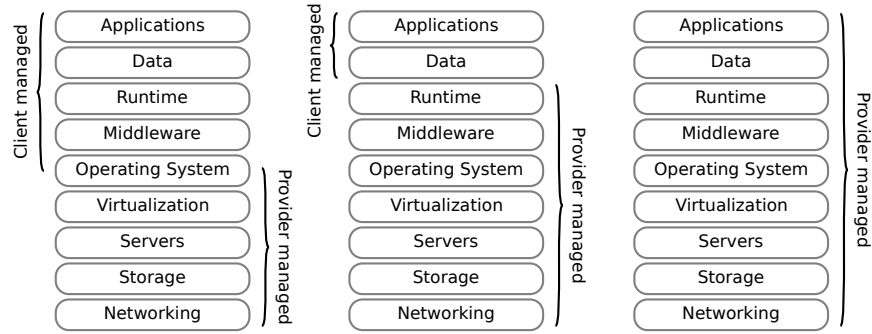
“Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.”

The NIST definition of cloud computing states that a cloud must have five essential characteristics: (1) the resources must be provided *on-demand* and without requiring human interaction by a service provider when a client requests them; (2) the resources must be accessible over the network using *standard technologies*, making them accessible by many different clients; (3) the resources are *pooled*, meaning that physical and virtual resources can be shared between multiple consumers using a multi-tenant model; (4) it should be possible to dynamically increase or decrease the amount of resources allocated to a tenant when needed, a concept referred to as *rapid elasticity*; and (5) the service should be *metered*, making it possible to report and bill based on the resources used.

1.1 Everything as a service

Cloud computing is the latest evolution of the utility computing concept, which was already envisioned in the early 1960s. In those days, computers were very rare and expensive, making it very wasteful not to fully utilize them. Because of this, computers were shared between multiple users. A user would offer programs, which were sent to a time-shared mainframe, where they would be executed. Since the Internet as we know it did not exist at the time, this was done by directly accessing the mainframe or by using dumb terminals that were attached to the computer. At most private networks were used, possibly using leased lines to connect multiple sites. As computational capacity was shared between users, this led some to believe that computing would eventually become a new utility, where *computation* would be a service like water or electricity [2].

With the rise of microcomputers, computers became less expensive and more widespread. The personal computers offered their users sufficient computing capacity at a low enough cost, thus removing the need for time-sharing. Computation shifted to the edge of the networks, or to terminals that were not connected to a network at all. This changed when the Internet gained popularity during the 1990s, which caused more and more functionality to be shifted back to remote datacenters. The mobile revolution further reinforced this trend, as it made energy-efficiency more important, and as the computing capacity of these devices is more limited than that of traditional computers, making remote computation more useful. Furthermore, the use of multiple computing devices further increased the benefits of using remote servers, as this made the synchronization and management of data easier. In 2014, 81% of mobile data traffic was already generated by cloud applications, and it is expected that this will further increase to 90% by 2019 [3]. Following this trend, various applications, which were at some point deployed on end user terminals or on-premise such as mail, calendar, office suites, enterprise resource planning, and many others are moving to a service-based distribution model. This



(a) Infrastructure-as-a-Service (b) Platform-as-a-Service (c) Software-as-a-Service

Figure 1.1: The three cloud service models defined by the NIST: Infrastructure-as-a-Service (IaaS), Software-as-a-Service (SaaS) and Platform-as-a-Service (PaaS).

approach of offering software as an on-demand service over a network is referred to as Software-as-a-Service (SaaS).

This trend is supported by an increase in flexibility within the datacenters, now often built using cheap commercial off-the-shelf hardware, where multiple Virtual Machines (VMs) can be deployed using virtualization. A VM behaves like a physical computer, but multiple VMs can be instantiated on a single host. This makes the deployment of additional application instances, which is done by instantiating new VMs, more flexible, and in turn makes it possible to elastically scale applications based on changing demand. New service models make use of this flexibility, making it possible to rent VMs by the hour. This resulted in the Infrastructure-as-a-Service (IaaS) model, where VMs are offered on-demand using a pay-per-use model, and the Platform-as-a-Service (PaaS) model, where a software platform on top of which applications can be built is offered.

IaaS, PaaS and SaaS are considered the three service models of cloud computing [1]. As illustrated in Figure 1.1, these models differ in how much of the traditional IT stack is provided as a service by the service provider, and how much of it is managed by the clients. Many other, less common models have also been proposed, such as e.g. Network-as-a-Service (NaaS), where a virtualized network is offered, Datacenter-as-a-Service (DCaaS) where both virtualized hosts and networks are offered, and Business-Process-as-a-Service (BPaaS) where entire business processes are outsourced and provided on-demand. This has resulted in the popularity of the all-encompassing Everything-as-a-Service (XaaS) concept, which embodies the notion of offering any type of application or resource as an on-demand metered service. XaaS and cloud computing are closely related, and nearly

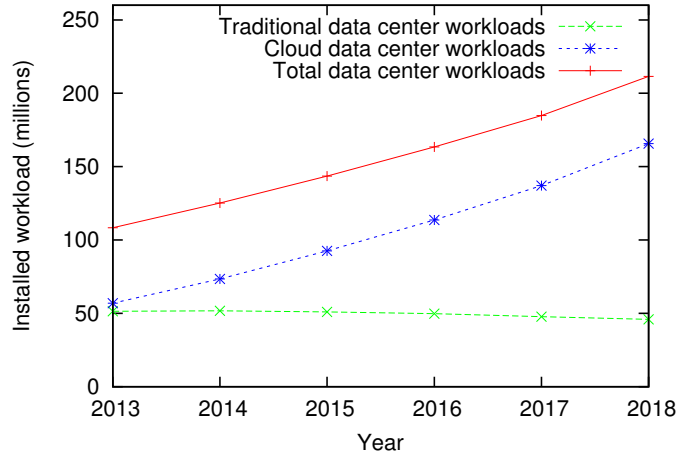


Figure 1.2: Estimated evolution of datacenter workloads. Datacenter workloads are expected to increase with a significant increase in the number of workloads handled by clouds. By 2018, 78% of all workloads will be processed in cloud datacenters. (Data obtained from [4])

every type of XaaS can be reduced to one of the three main cloud service models. NaaS and DCaaS can for example be seen as specific (though less traditional) IaaS variants. BPaaS can generally be considered either as SaaS or PaaS depending on whether the client can plug into the process with custom components¹. The popularity of these new paradigms is rising, and workloads are increasingly migrated from traditional datacenters to datacenters managed using cloud technologies [4]. Figure 1.2 demonstrates this evolution in recent years and shows a forecast for the near future.

The characteristics of clouds offer their operators and users many benefits, amongst others the following:

1. Applications are centralized within a datacenter. This makes it easier to manage and upgrade the application. It also makes it easier to synchronize application data as it can be managed in a centralized way.
2. The datacenter contains a large pool of resources which is shared by multiple groups of clients and users, referred to as tenants. While the resource capacity is finite, it generally greatly exceeds the resource requirements of a single tenant. From the point of view of individual tenants, the cloud therefore generally seems to offer limitless resource capacity.

¹ In some cases, a BPaaS workflow can also contain tasks which are outsourced to humans. These human actors do not fit within the traditional cloud computing layers, and show that XaaS, while supported by cloud technologies and concepts, is a broader concept than cloud computing.

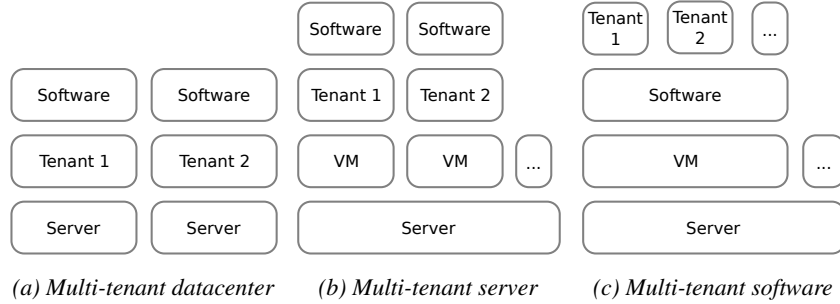


Figure 1.3: Multi-tenancy can be achieved at multiple levels, resulting in differing granularity.

3. Economies of scale are leveraged by the datacenter operator, enabling him to offer resources at a lower cost. In addition, spikes in the resource utilization of one tenant can be smoothed by the varying resource utilization of other tenants, and are smaller relative to the total number of available resources. This reduces the need for overprovisioning of resources for the service provider, further reducing costs.
4. Resources can be acquired and released quickly and on-demand without human interaction, allowing for quick reactions to changing demands. This makes it possible to elastically scale applications up and down, removing the need to overprovision for peak resource use by the tenant.

Clouds generally offer resources to different clients, organizations and groups of end users, each referred to as tenants. When a single user subscribes for a service such as for example Dropbox or Microsoft Office 365, he is both the tenant and the end user. It is however also possible for an organization to subscribe for these services, offering the service to its members. In this case, the organization buying the service is a tenant, while the members of the organization are the end users. A single tenant can therefore correspond to one or more end users.

Clouds are generally multi-tenant: multiple tenants can make use of the same shared resource pool. This multi-tenancy can however occur at multiple granularity levels. At the datacenter level, a single server may be dedicated to a single tenant. A single server may also be split into virtual partitions, each emulating an entire computer system. In this case, a single VM can be dedicated to a single tenant. Finally, multi-tenancy can also be handled at the software level. In this case, software is designed in such a way that is able to handle requests of the end users of multiple tenants.

These alternative approaches to multi-tenancy are illustrated in Figure 1.3, and impact the number of servers that are needed to provision an application: when

	Multi-tenant datacenter	Multi-tenant servers	Multi-tenant software
Maximum # tenants	1	180 ²	1500 ³
Servers needed	1500	9	1
Server utilization	0.1%	11%	100%

Table 1.1: Comparison of the different types of multi-tenancy for 1500 tenants with each a single end user.

datacenter multi-tenancy is used, at least one server must be provided for every tenant, even if these servers are underutilized, while software multi-tenancy is only limited by the number of requests that a single server can handle. Table 1.1 illustrates this difference by showing the number of servers needed to provide resources for an application with 1500 tenants, each having a single end user. While every server may be able to handle requests for a similar number of end user requests (in this case assumed to be 1500 end users, based on salesforce.com numbers), datacenter-level multi-tenancy and server-level multi-tenancy are instead limited by the number of tenants that a single server can provide resources for. Multi-tenant software, by contrast, is only constrained by the number of end users a single server can support, irrespective of the number of tenants. As shown in the table, this can significantly impact the number of servers needed to provide a software service, the server utilization, and therefore the cost of providing a service. In this dissertation, the focus is specifically on multi-tenancy at the software level.

More recently, there has been an evolution to move away from single datacenter deployments, once again bringing applications closer to the end user, as this can improve both service reliability and quality. Multi-cloud deployments can improve service availability and quality, as this prevents a service from breaking down when a datacenter fails, and can reduce latency when the cloud closest to an end user is used to service his requests. Edge clouds can also be used to lower network latency and network loads e.g. by using caching. Fog computing [6] goes even further, by not only leveraging edge clouds, but also other hardware such as set-top-boxes and access points. In addition, there has been an increasing focus on the networks within clouds and those interconnecting them. This has resulted in technologies

² Assuming a high-end server with 60 available cores is used with 3 VMs per core. In practice, deploying this many VMs on a single host may lead to significant performance penalties due to the overhead incurred by context switches and the large number of running operating systems. A lower number of VMs may be preferred to ensure applications perform consistently.

³ This number is dependent only on the executed application and the server capacity, and not on the number of tenants. The value of 1500 is based on the server and user numbers of salesforce.com in 2009. At a salesforce.com event in New York City, founder and CEO Marc Benioff stated that their multi-tenant software was hosted on 1000 servers that together were able to handle the requests of their 1.5 million end users [5], implying that a single of their servers is, on average, responsible for handling the requests of 1500 end users.

such as Software-Defined Networking (SDN) [7], which are used to make the network programmable, and Network Functions Virtualization (NFV) [8] which allows service providers to replace expensive dedicated networking hardware by dynamically instantiated cloud services.

1.2 Problem statement

As migrating applications to cloud environments can lead to significant cost savings, more and more applications are offered using cloud infrastructures, migrating them from on-premise deployments to remote datacenters. Usually these applications are offered to a large number of customers, who can then request the SaaS on-demand, and without human interaction. These mass-market applications are usually provided with only limited customizability: clients order or register for a take-it-or-leave-it version of the software with limited configurability. As shown by the increasing popularity of these types of services, this approach works well for cloud-scale applications with huge numbers customers.

Some applications are however typically sold to a small number of large clients, who each pay a considerable amount of money for the provided service. Since individual clients are much more important in this instance, they are able to demand applications that are custom tailored to their specific needs. This traditionally results in heavily customized applications that are built from a customized code base, which may contain client-specific changes. When these applications are hosted at the client site, the service becomes more expensive as sufficient hardware must be provided to handle peak workloads. Moving these applications to the cloud reduces the need to provision resources for peak workloads, but as every client requires its own active application instances, and no resources can be shared between clients, the benefits of using cloud infrastructure are diminished. Ideally, customizable SaaS should be developed in such a way that it supports multi-tenancy at a software level, ensuring every tenant can make use of a single, shared application instance that can scale on-demand. As customizations often result in changes to the application binary, an approach for developing and managing customizable SaaS is needed.

Many applications that are currently deployed on the client-side could benefit from leveraging cloud technologies. To achieve this, the applications must be migrated to a cloud environment, turning them into a SaaS offering. Depending on the application, it is however not always possible to completely migrate applications to a remote datacenter, as there are various reasons why application components may not be migrated: local physical terminals may provide part of the service's functionality, local hardware may be needed to provide fallback functionality when high reliability is needed, or the application may require local data that may not be moved for compliance reasons. In each of these cases, only a partial migration

can be achieved. This implies that the network connecting the local components to the remote environment and its capacity must be taken into account as well during service deployment.

A significant growth is expected in cloud workloads in the next few years. The greatest growth is expected in SaaS, which is expected to have a 59% share of all cloud workloads by 2018 [4]. To achieve this, a cloud migration path for existing complex applications, and an approach for managing SaaS variability are needed. The central research question in this dissertation is “How can application and service customizability be handled in multi-tenant cloud and network environments?”. In particular, the following issues are targeted:

1. While variability management approaches exist that can be used to maintain multiple customized versions of an application [9], these approaches generally result in multiple application binaries, making it impossible to share resources between tenants. Therefore, an approach is needed to design and model customizable multi-tenant SaaS applications. This approach should fit within a generic migration approach that can be used to migrate existing customizable applications to the multi-tenant cloud.
2. SaaS customizability can be taken into account during the runtime management of these applications. This makes it possible for the management system to minimize resource usage, and to take application variability into account when service failures occur due to insufficient resource availability. To achieve this, novel datacenter resource management algorithms, which can be used to manage customizable SaaS applications, are needed.
3. When applications are migrated to a remote cloud environment, it is not always possible or desirable to host the entire application in a single datacenter. In some cases, application components must remain client-side, for compliance and quality reasons, or because the service depends on the availability of physical devices. In other cases, multi-cloud deployments and technologies such as fog computing make it possible to spread application components throughout the network. This puts an increasing strain on the underlying network. Therefore, strategies are needed to take the underlying network into account during the deployment and management of customizable applications.

1.3 Definitions & terminology

In this section, a definition is provided of the most important concepts used throughout this Ph.D. dissertation:

- **Virtualization:** Virtualization is used to make an abstraction of physical hardware, turning physical resources into logical resources. Virtualized resources provide identical functionality compared to their physical counterparts, emulating their behavior. They can however be deployed with more flexibility, e.g. by executing multiple virtualized resources on a single physical machine.
- **Virtual machine:** A Virtual Machine (VM) is, as the name implies, a virtualized computer system. VMs can be deployed on physical hardware, and each VM behaves like an isolated computer system. VMs are generally used as the primary resources offered and hosted by IaaS clouds.
- **Multi-tenancy:** A tenant is a group of users, typically belonging to a single organization that have a similar view on a software application. In multi-tenant applications, a single application instance is used to provide resources for multiple tenants, making it possible to share resources between the different tenants. This contrasts with applications where a separate instance is created for every tenant.
- **Configuration and customization:** When applications must be tailored for a tenant, a distinction can be made between configuration changes and customization changes. Configuration changes are smaller changes that can be implemented by changing the application configuration. For traditional applications, these changes are generally carried out by changing configuration files. These configuration changes are usually easy to implement, and can therefore be supported with relative ease in SaaS applications. Customization changes are more invasive changes that significantly impact the functionality of an application or its quality characteristics. In traditional applications, these changes are generally carried out by changing the application at compile-time, resulting in a separate binary. These changes can therefore not be provided using multi-tenant SaaS.
- **Feature:** A feature is a distinct application functionality that may or may not be included in an application. By including and excluding features multiple application variants can be created with differing functionalities. Features can therefore be used to model application customizability. Feature models, which define relations between features, are used to express which feature combinations result in valid, feasible application instances.

1.4 Research contributions

The main contributions of this dissertation address the challenges involved with the modeling and management of customizable SaaS applications and services, and

how customizable applications can be migrated from on-premise deployments to remote datacenters. The ultimate goal is to develop technologies that can be used to improve the customizability of SaaS applications, making it possible to provide tailored applications at a reduced cost. The following contributions are detailed in this dissertation:

1. *An approach for modeling and managing customizability of multi-tenant SaaS applications.*

Offering customizable SaaS applications is simple when multi-tenancy isn't needed, and multi-tenant SaaS can be delivered when customizability is limited. Designing applications that support both properties is however more complicated. To achieve this, the following contributions are presented:

- (a) An approach for modeling SaaS applications by splitting them up into individual components each realizing application features has been developed. Using this approach, applications are composed out of multiple multi-tenant components using a Service-Oriented Architecture (SOA).
- (b) An application development platform and a runtime platform are presented, demonstrating how multi-component applications can be designed, and how they can be managed.

2. *A resource allocation approach for managing customizable SaaS applications within datacenters.*

Using a component-based approach for modeling SaaS applications makes it necessary to develop management algorithms that can be used to allocate these multi-component applications within the datacenter. These algorithms can be made aware of application customizability, making it possible to take variability into account during the application allocation. Concretely, the following contributions are presented:

- (a) The feature placement problem is introduced. This algorithmic problem deals with the resource-allocation of component-based customizable SaaS applications, and takes application variability into account during the resource allocation process to minimize associated costs.
- (b) Static and dynamic optimal and heuristic algorithms that solve the feature placement problem are presented and evaluated.
- (c) A scalable hierarchical resource management system that can be used to improve the scalability of centralized cloud management algorithms such as the heuristic feature placement algorithms is presented.

3. *Network-aware modeling and management algorithms for inter-cloud network environments.*

SaaS applications and services are not always contained within a single cloud

environment, but may also be distributed over multiple clouds, devices and networks. To support this scenario, a management and modeling approach for interconnected services in network environments is needed. This is achieved by the following contributions:

- (a) An approach for modeling service and network resource utilization within networks is presented. This model can be used to determine the impact of service migrations and deployments on client networks, and can be used as an access filter before service changes are enacted.
- (b) A generalized network-aware modeling and management approach for customizable component-based multi-tenant SaaS services in wide area networks is introduced.

1.5 Outline of this dissertation

This dissertation is composed of a number of publications that were realized during this Ph.D. research (as typically the case for dissertations at the Faculty of Engineering and Architecture of Ghent University). The selected publications provide an integral and consistent overview of the work performed. The different research contributions are detailed in Section 1.4 and the complete list of publications that resulted from this work is presented in Section 1.6.

This section provides an overview of the remainder of this dissertation, showing how the different chapters are linked together. A schematic overview of how the chapters (Ch.) and appendices (App.) are related to each other and to the research contributions is depicted in Figure 1.4. Table 1.2 shows how the chapters in this dissertation relate to the contributions listed in Section 1.4.

In Chapter 2 multiple alternative approaches for modeling and managing customizable multi-tenant SaaS application are presented and evaluated. An approach where customizability is handled by creating and managing multiple application binaries based on specific application configurations is compared with a SOA approach where customization is handled by composing applications out of different services. Both approaches are evaluated, showing that the SOA-based approach reduces the number of required instances when highly customizable applications are considered, which translates to a greater potential for multi-tenancy. In addition, a hybrid approach extending the SOA-based approach, and combining properties of both basic approaches is presented.

Chapter 3 introduces the feature placement problem, which can be used to allocate resources for customizable SaaS applications that are built using a SOA. A formalized model for this problem is defined and multiple algorithms that can be used to solve the static feature placement problem are defined. In Chapter 4, dynamic algorithms solving the feature placement problem are designed and evaluated

in detail. These algorithms take changes in deployed applications and application workloads into account. By limiting the number of migrations, these algorithms are more suitable for cloud deployments. To improve the scalability of these centralized datacenter management algorithms, a hierarchical management framework, such as the one presented in Appendix A can be used.

The various customizable SaaS management concepts are framed within a wider development and runtime platform in Chapter 5. The main focus of this chapter is on the development process, showing how the SOA-based principles can be utilized during development and deployment of the application. To achieve this, an automated approach to convert between a development application model and a runtime application model is presented.

When applications and services are migrated from localized deployments to remote cloud environments, this has the risk of impacting the underlying network infrastructure, as workloads may stress a larger part of the network. When software is migrated, deployed or upgraded, it is therefore important to take network capabilities into account. Chapter 6 addresses how the impact of service deployments on networks can be determined, focusing on client networks. Appendix B shows how this impact analysis approach fits within the cloud migration framework.

Chapter 7 combines the presented approach for management of customizable SaaS applications with network-awareness, and studies how service variability can be handled in service provider networks. To support heterogeneous networks, the SOA-based approach is extended to support hardware devices and both virtual and physical services.

Finally, in Chapter 8, conclusions and research perspectives are presented.

	Ch.2	Ch.3	Ch.4	Ch.5	Ch.6	Ch.7
Modeling and management of customizable SaaS	•	•	•	•		•
Datacenter management algorithms		•	•			•
Network-aware service management					•	•

Table 1.2: An overview of the contributions (Section 1.4) per chapter in this dissertation.

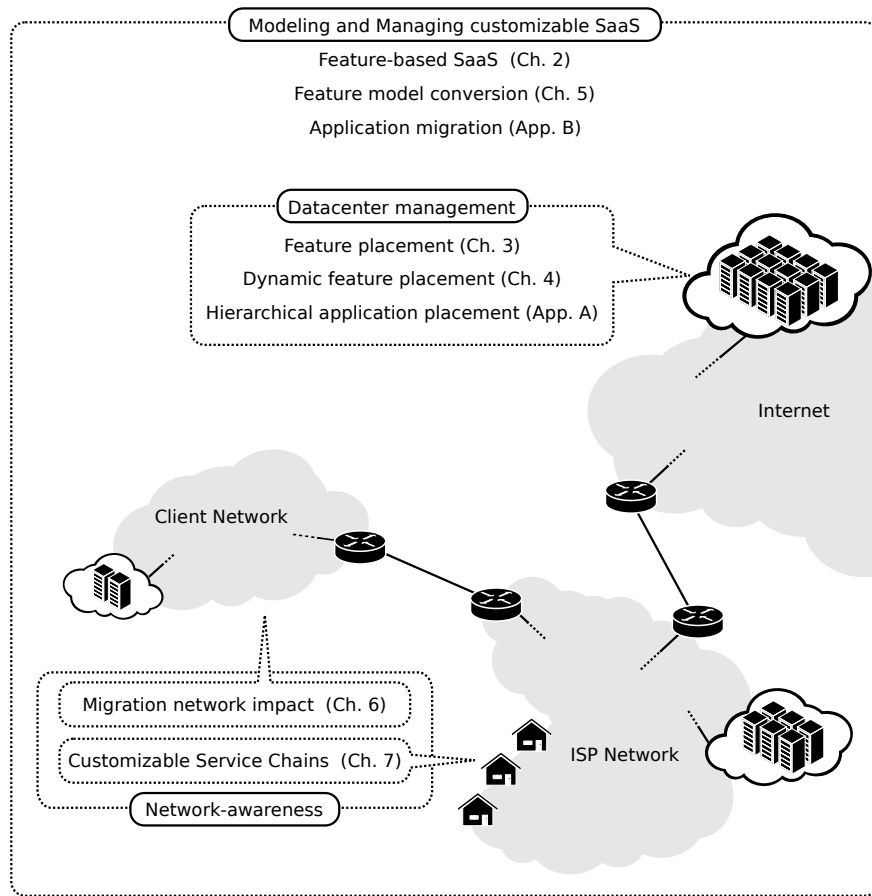


Figure 1.4: Schematic position of the different chapters in this dissertation.

1.6 Publications

The research results obtained during this Ph.D. research have been published in scientific journals and presented at a series of international conferences. The following list provides an overview of the publications during this Ph.D. research.

1.6.1 Publications in international journals (listed in the Science Citation Index⁴)

1. F. De Backere, **H. Moens**, K. Steurbaut, K. Colpaert, J. Decruyenaere, and F. De Turck. *Towards automated generation and execution of clinical guidelines: Engine design and implementation through the ICU Modified Schofield use case*. Computers in Biology and Medicine, 42(8):793–805, aug 2012. doi:10.1016/j.combiomed.2012.06.003.
2. **H. Moens**, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. *Cost-Effective Feature Placement of Customizable Multi-Tenant Applications in the Cloud*. Journal of Network and Systems Management, 22(4):517–558, oct 2014. doi:10.1007/s10922-013-9265-5.
3. **H. Moens** and F. De Turck. *Shared resource network-aware impact determination algorithms for service workflow deployment with partial cloud offloading*. Journal of Network and Computer Applications, 49(0):99–111, mar 2015. doi:10.1016/j.jnca.2014.11.004.
4. **H. Moens**, B. Dhoedt, and F. De Turck. *Allocating Resources for Customizable Multi-Tenant Applications in Clouds Using Dynamic Feature Placement*. Future Generation Computer Systems, 2015. Accepted for publication.
5. P.-J. Maenhaut, **H. Moens**, V. Ongenae, and F. De Turck. *Migrating Legacy Software to the Cloud: Approach and Verification by means of Two Medical Software Use Cases*. Software: Practice and Experience, 2015. Accepted for publication. doi:10.1002/spe.2320.
6. **H. Moens** and F. De Turck. *Customizable Function Chains: Managing Service Chain Variability in Hybrid NFV Networks*. Transactions on Networking, 2015. Submitted.

⁴The publications listed are recognized as ‘A1 publications’, according to the following definition used by Ghent University: A1 publications are articles listed in the Science Citation Index, the Social Science Citation Index or the Arts and Humanities Citation Index of the ISI Web of Science, restricted to contributions listed as article, review, letter, note or proceedings paper.

1.6.2 Publications in international conferences (listed in the Science Citation Index⁵)

1. **H. Moens**, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. D. Turck. *Feature Placement Algorithms for High-Variability Applications in Cloud Environments*. In Proceedings of the 13th Network Operations and Management Symposium (NOMS 2012), pages 17–24. IEEE, apr 2012. doi:10.1109/NOMS.2012.6211878.
2. **H. Moens**, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. *Developing and Managing Customizable Software as a Service Using Feature Model Conversion*. In Proceedings of the 3rd IEEE/IFIP Workshop on Cloud Management (CloudMan 2012), pages 1295–1302. IEEE, apr 2012. doi:10.1109/NOMS.2012.6212066.
3. **H. Moens**, K. Handekyn, and F. De Turck. *Cost-Aware Scheduling of Deadline-Constrained Task Workflows in Public Cloud Environments*. In Proceedings of the 13th IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), pages 68–75. IEEE, may 2013.
4. P.-J. Maenhaut, **H. Moens**, M. Verheyne, P. Verhoeve, S. Walraven, W. Joosen, and V. Ongenae. *Migrating Medical Communications Software to a Multi-Tenant Cloud Environment*. In Proceedings of the 13th IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), pages 900–903. IEEE, may 2013.

1.6.3 Publications in other international conferences

1. F. De Backere, **H. Moens**, K. Steurbaut, F. De Turck, K. Colpaert, C. Dancneels, and J. Decruyenaere. *Automated generation and deployment of clinical guidelines in the ICU*. In Proceedings of the 23rd IEEE International symposium on Computer-Based Medical Systems, pages 197–202. IEEE, oct 2010. doi:10.1109/CBMS.2010.6042640.
2. **H. Moens**, J. Famaey, S. Latré, B. Dhoedt, and F. De Turck. *Design and Evaluation of a Hierarchical Application Placement Algorithm in Large Scale Clouds*. In Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011), pages 137–144. IEEE, may 2011. doi:10.1109/INM.2011.5990684.

⁵The publications listed are recognized as ‘P1 publications’, according to the following definition used by Ghent University: P1 publications are proceedings listed in the Conference Proceedings Citation Index - Science or Conference Proceedings Citation Index - Social Science and Humanities of the ISI Web of Science, restricted to contributions listed as article, review, letter, note or proceedings paper, except for publications that are classified as A1.

3. K. Geebelen, S. Walraven, E. Truyen, S. Michiels, **H. Moens**, F. De Turck, B. Dhoedt, and W. Joosen. *An Open Middleware for Proactive QoS-Aware Service Composition in a Multi-Tenant SaaS Environment*. In Proceedings of the 2012 International Conference on Internet Computing (ICOMP 2012). CSREA Press, jul 2012.
4. **H. Moens**, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. *Network-Aware Impact Determination Algorithms for Service Workflow Deployment in Hybrid Clouds*. In Proceedings of the 8th International Conference on Network and Service Management (CNSM 2012), pages 28–36. IEEE, oct 2012.
5. **H. Moens** and F. De Turck. *A Scalable Approach for Structuring Large-Scale Hierarchical Cloud Management Systems*. In Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013), pages 1–8. IEEE, oct 2013. doi:10.1109/CNSM.2013.6727803.
6. **H. Moens**, B. Hanssens, B. Dhoedt, and F. De Turck. *Hierarchical Network-Aware Placement of Service Oriented Applications in Clouds*. In Proceedings of the 14th Network Operations and Management Symposium (NOMS 2014), pages 1–8. IEEE, may 2014. doi:10.1109/NOMS.2014.6838230.
7. M. Barshan, **H. Moens**, S. Latré, and F. De Turck. *Algorithms for Efficient Data Management of Component-Based Applications in Cloud Environments*. In Proceedings of the 14th Network Operations and Management Symposium (NOMS 2014), pages 1–8. IEEE, may 2014. doi:10.1109/NOMS.2014.6838257.
8. P.-J. Maenhaut, **H. Moens**, M. Decat, J. Bogaerts, B. Lagaisse, W. Joosen, V. Ongenae, and F. De Turck. *Characterizing the Performance of Tenant Data Management in Multi-Tenant Cloud Authorization Systems*. In Proceedings of the 14th Network Operations and Management Symposium (NOMS 2014), pages 1–8, Krakow, Poland, may 2014. IEEE. doi:10.1109/NOMS.2014.6838232.
9. **H. Moens** and F. De Turck. *Feature-Based Application Development and Management of Multi-Tenant Applications in Clouds*. In Proceedings of the 18th International Software Product Line Conference (SPLC 2014), pages 72–81. ACM, sep 2014. doi:10.1145/2648511.2648519.
10. P.-J. Maenhaut, **H. Moens**, V. Ongenae, and F. De Turck. *Scalable User Data Management in Multi-Tenant Cloud Environments*. In Proceedings of the 10th International Conference on Network and Service Management (CNSM 2014), pages 268–271, Rio de Janeiro, Brazil, nov 2014. IEEE. doi:10.1109/CNSM.2014.7014171.

11. M. Barshan, **H. Moens**, and F. De Turck. *Design and Evaluation of a Scalable Hierarchical Application Component Placement Algorithm for Cloud Resource Allocation*. In Proceedings of the 10th International Conference on Network and Service Management (CNSM 2014), pages 175–180. IEEE, nov 2014. doi:10.1109/CNSM.2014.7014155.
12. **H. Moens** and F. De Turck. *VNF-P : A Model for Efficient Placement of Virtualized Network Functions*. In Proceedings of the 10th International Conference on Network and Service Management (CNSM 2014), pages 418–423. IEEE, nov 2014. doi:10.1109/CNSM.2014.7014205.
13. M. Barshan, **H. Moens**, J. Famaey, and F. De Turck. *Algorithms for Advance Bandwidth Reservation in Media Production Networks*. In Proceedings of the 14th IFIP/IEEE International Symposium on Integrated Network Management (IM 2015), pages 183–190. IEEE, may 2015.
14. P.-J. Maenhaut, **H. Moens**, V. Ongenae, and F. De Turck. *Design and Evaluation of a Hierarchical Multi-Tenant Data Management Framework for Cloud Applications*. In 7th International Workshop on Management of the Future Internet (ManFI), pages 1208–1213. IEEE, may 2015.

References

- [1] P. Mell and T. Grance. *The NIST Definition of Cloud Computing* [online]. 2011. Available from: <http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf>.
- [2] S. Garfinkel. *Architects of the Information Society: 35 Years of the Laboratory for Computer Science at MIT*. MIT Press, 1999.
- [3] Cisco Systems. *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2014-2019* [online]. Last accessed: February 2015. Available from: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper_c11-520862.html.
- [4] Cisco Systems. *Cisco Global Cloud Index: Forecast and Methodology, 2013-2018* [online]. Last accessed: February 2015. Available from: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/Cloud_Index_White_Paper.html.
- [5] E. Schonveld. *The Efficient Cloud: All Of Salesforce Runs On Only 1,000 Servers* [online]. 2009. Last accessed: May 2015. Available from: <http://techcrunch.com/2009/03/23/the-efficient-cloud-all-of-salesforce-runs-on-only-1000-servers/>.
- [6] Cisco Systems. *Fog Computing, Ecosystem, Architecture and Applications* [online]. Last accessed: February 2015. Available from: http://www.cisco.com/web/about/ac50/ac207/crc_new/university/RFP/rfp13078.html.
- [7] Open Networking Foundation. *Software-Defined Networking: The New Norm for Networks* [online]. 2012. Last accessed: February 2015. Available from: <https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf>.
- [8] ETSI. *Network Functions Virtualisation – Introductory White Paper* [online]. 2012. Last accessed: February 2015. Available from: https://portal.etsi.org/NFV/NFV_White_Paper.pdf.
- [9] K. Pohl, G. Böckle, and F. Van Der Linden. *Software product line engineering: foundations, principles, and techniques*. Springer-Verlag New York, Inc., 2005.

2

Feature-Based Application Development and Management of Multi-Tenant Applications in Clouds

This chapter introduces an approach for designing multi-tenant Software-as-a-Service (SaaS), and is the basis for the next chapters. In recent years, there has been a rising interest in cloud computing, which is often used to offer SaaS over the Internet. SaaS applications can be offered to clients at a lower cost as they are usually multi-tenant: many end users make use of a single application instance, even when they are from different organisations. It is difficult to offer highly customizable SaaS applications that are still multi-tenant, which is why these SaaS applications are often offered in a one size fits all approach. In this chapter, multiple approaches for developing and managing customizable multi-tenant SaaS offerings are explored. We compare two approaches: an Application-Based Binary (ABB) approach which focuses on deploying multiple multi-tenant application variants, and a Feature-Based Binary (FBB) approach where applications are composed out of multi-tenant services using a service oriented architecture. In addition, a third, hybrid approach, combining properties of both is presented. When many application variants are possible, the FBB results in the fewest application instances at runtime, increasing the resource sharing that can be achieved. In instances where ABB achieves more resource sharing, the hybrid approach, which is based on FBB, can be used instead. Because of these conclusions we focus on the FBB approach in the next chapters.

2.1 Introduction¹

There has been a growing interest in cloud computing, where applications are no longer executed on-premise but in remote datacenters. For application providers, offering Software-as-a-Service (SaaS) via the Internet makes it possible to offer applications to large numbers of end users, resulting in cost savings through economies of scale. These cost-savings are partially achieved by using multi-tenancy: multiple end users make use of the same application instances, reducing the total resource need and reducing the impact of varying numbers of end users. This however results in limited customizability, often causing SaaS applications to offer a one size fits all approach. For some applications, such as e.g. document processing, medical communications and medical information management a very high customizability is needed. In these situations, multiple very large client organizations, referred to as tenants, require extensive customization options for their specific use cases. For these application cases, customizations were traditionally provided on an ad-hoc basis resulting in limited code reuse, increasing management complexity and high development costs. The development, deployment and management approaches discussed in this chapter are based on our experiences with these three applications.

Offering multi-tenant SaaS applications with high customizability using a single application instance is hard as all of the customization must then be handled at runtime using a single application binary. Even in cases where this is possible, e.g. if all customizations can be implemented using Aspect-Oriented Programming (AOP) [1], this may still be a problem as executing different code paths for different end users may impact the application performance, and by consequence the quality characteristics of the service. This problem may be further exacerbated by additional quality customizations making it even more difficult to provision the application for all users using a single instance type.

These functional and quality considerations make it infeasible to use a single application instance for all application variants. In this situation, two approaches can be used to offer customizable SaaS applications. 1) Multiple application instances can be built that each include a different set of customizations. In this scenario, multiple multi-tenant application variants are built statically and managed independently. Multi-tenancy can then only be achieved when two tenants make use of the same application customization. We refer to this method for achieving multi-tenancy as the Application-Based Binary (ABB) approach. 2) Alternatively, an approach where every application is built using a Service-Oriented Architec-

¹ This chapter is based on the paper *Feature-Based Application Development and Management of Multi-Tenant Applications in Clouds* by H. Moens and F. De Turck which was published in proceedings of the 18th international Software Product Line Conference (SPLC 2014). For this dissertation, Section 2.3 (Software Product Line Engineering) was added to more clearly introduce the software product line engineering concepts.

ture (SOA) can be used. We refer to this as the Feature-Based Binary (FBB) approach because each of the application services is used to offer one or more of the application features. In this chapter, we analyze the approaches for offering customizable multi-tenancy, comparing development, deployment and runtime management; and the amount of multi-tenancy that can be attained using both approaches. Additionally, we propose a hybrid approach that adds properties of the ABB approach to the FBB approach.

In the next section we discuss related work. In Section 2.3 an approach for modeling application variability is discussed. Afterwards, in Section 2.4 we discuss the ABB and FBB approaches for managing variability of SaaS applications in clouds and describe a hybrid approach incorporating ABB applications in the FBB approach. Then we analyze the approaches theoretically in Section 2.5 and experimentally in Section 2.6. This is followed by a brief discussion in Section 2.7. Finally, in Section 2.8 we discuss our conclusions.

2.2 Related work

Dynamic software product lines [2, 3] can be used to develop applications that can be customized at runtime. Many different approaches have been proposed in recent years [4]. An aspect-oriented approach for dynamically managing variability is presented in [1]. This approach can be used to create highly customizable applications that all function using a common code base, and thus using common application instances. It is however not always possible to capture all possible variation using a single code base, limiting the potential customizability of applications. More importantly, using different code paths for different tenants may impact application quality, causing tenants to influence each others performance. Therefore other approaches for offering multi-tenant customizability in SaaS applications, such as the approaches presented in this chapter, are still needed. The approaches presented in this chapter benefit from such approaches, as they can be used to increase the runtime customizability of individual application features.

Another approach for managing application customizability is by using SOA architectures. An approach using runtime application customization was discussed in [5], where runtime adaptation of applications based on changing application context is achieved for applications built using service-oriented architectures. The authors however do not focus on the running multiple customizations at the same time, which is necessary for multi-tenant applications where tenants have custom customizations.

In [6] Mietzner et al. present an approach for constructing customizable multi-tenant applications using a SOA. This approach is similar to the FBB approach which we discuss in this chapter. We however extend the approach by also considering the runtime management and resource allocation of applications. Furthermore,

we compare the FBB approach with an alternative ABB approach to determine when each approach is preferable using a theoretical and experimental analysis, and we present a hybrid approach combining properties of both FBB and ABB. In [7], the authors make a distinction between external and internal variation. Only the former variations visible to end users while the other variants may be left undecided when applications are specified resulting in open variation points. We exploit this concept of open variation points to reduce resource costs in the FBB management algorithms.

Some approaches, such as [8], focus on customizing applications by changing the workflow in SOA applications. The FBB approach described in this chapter is similar in that we use a SOA, but we focus on replacing components based on tenant customizations for performance isolation rather than on customizing the interactions between the components. These workflow customization approaches are complementary to the FBB approach as they can be used to coordinate between the resulting application components and offer additional application customizability.

The concepts presented in this chapter are based on service lines [9], which are used to construct customizable workflows of customizable services. The authors however focus on AOP and dependency injection to offer customizations. The FBB approach presented in this chapter extends the approach by offering greater customizability of services by permitting the use of separate application instances when doing so is needed for performance isolation and higher customizability. Furthermore, the management approach discussed in this chapter can be used to reduce management costs by exploiting open variation points at runtime.

This chapter builds on our previous work in runtime management of customizable cloud applications [10, 11], which will be discussed in-depth in Chapters 3 and 4, where we discuss how SOA applications can be managed by a cloud management system, and how open variation points can be exploited at runtime to reduce management costs. This chapter describes the broader approach, discussing how these component-based applications can be developed, deployed and managed. We also compare the FBB approach with an alternative ABB approach, and describe a hybrid approach containing properties of both approaches.

2.3 Software Product Line Engineering

Software Product Line Engineering (SPLE) [12] is an engineering domain that focuses on the development of methods, techniques and tools that can be used to manage software variability. To represent application variability, an application is modeled as a collection of *features*. A feature is a concept which is used to model a distinct set of functionality that can be part of an applications. Features can be implemented in multiple ways: they may link to application configuration changes, they may refer to application code assets, or they may refer to AOP aspects that

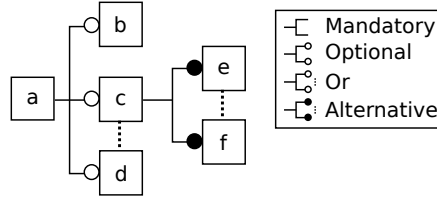


Figure 2.1: An illustration of a hierarchically structured feature model. Every box represents a feature. Feature *a* is the root of the feature model, while the other features are related using various relations, which are represented using the Pure::Variants notation.

can be woven into an application to modify specific application functionality. By selecting a subset of features, and including them in an application, an application variant is created. Application customization can therefore be achieved by selecting and deselecting different sets of features for different application variants, and subsequently building the applications. Some features that do not contain any code or configuration changes, can also be included. These empty features can be used to add structure to the feature model.

To better structure the application customization process, the application features are related using a feature model. This makes it easier to specify the sets of features that may occur at the same time in an application, as the inclusion of some features may result in the inclusion or exclusion of other features. Feature models are generally structured hierarchically, and are constructed making use of a limited set of relations that associate parent features with one or more child features. This hierarchical approach makes it easier to reason using these models, and also makes it possible to graphically represent feature models. Figure 2.1 shows an illustrative feature model using the Pure::Variants [13] feature model notation which we use throughout this dissertation. The following four relation types are generally discerned:

- **Mandatory(a, b):** If a feature *a* is included, the feature *b* must be included as well.
- **Optional(a, b):** If a feature *a* is included, the feature *b* may be included. Conversely, the feature *b* must not be included if *a* is not included.
- **Alternative(a, S):** If a feature *a* is included exactly one of the features contained in the set *S* must be included. If *a* is not included, none of the features in *S* may be included.
- **Or(a, S):** If a feature *a* is included, at least one of the features contained in the set *S* must be included. If *a* is not included, none of the features in *S* may be included.

2.4 SaaS multi-tenancy approaches

Multi-tenancy is an important concept for reducing costs in cloud environments. When an application is multi-tenant, multiple end users and tenants make use of a single shared application instance. An *instance* in this context is a compiled artifact that is executed on physical or virtualized hardware. The different ways in which variability is handled can have an impact on development complexity, performance consistency and flexibility. We focus on an approach using feature modeling, where the variability of an application is represented using *features*. A feature is a specific application functionality that can be included in an application. These functionalities can be represented in a feature model, a formal representation of relations between features that can be used to determine which feature combinations are possible. We consider two types of variability:

1. **Compile time variation:** If variations are compiled into the application, this results in the maximum flexibility when developing applications as entirely different code may be used for each different application variant. Furthermore, it is easier to ensure there is more consistent application performance for different users as all users make use of the same code. The cost of this approach is however higher, as it is impossible to share resources between tenants with different configurations. This results in an increase of instance types and management complexity. These variations can be either defined by developing separate code modules, or alternatively by defining AOP aspects that are compiled statically into the application binary.
2. **Runtime variation:** Two runtime variation types can be distinguished: configuration changes and customization changes [14]. Runtime configuration can only be used for smaller changes that can be done by e.g. changing configuration files. These changes are easy to implement and are therefore already supported by many SaaS applications. They are however also the least flexible. Runtime customization changes are harder to implement as they result in the execution of different code paths. Using AOP techniques, the code of running applications can be changed at runtime by dynamically weaving changes into the runtime binary of an application. While dynamic AOP is well-established, it is more complicated to develop and test applications using this approach compared to an approach where static compilation is used.

In practice applications have multiple customization and configuration options. Therefore, it is possible to combine the various variation generation approaches, handling some of the changes at runtime while managing others at compile time. As in pure multi-tenant applications all end users make use of the same application instance, customization can only be offered through runtime variation. This limits

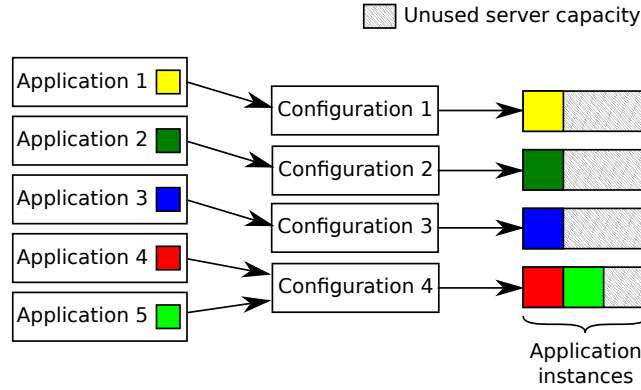
the customizability of the application. Alternatively, an approach with multiple application instances using different compile-time customizations must be used. When using multiple instance types, there are two possible approaches for managing customizability:

- *Application-Based Binary (ABB) approach:* Using the ABB approach, custom application binaries are generated for every application variant that is used by tenants. Resources can be shared between application instances when the users have identical customizations. Instances of every used application variant must always be active in the cloud environment, even if there are no users, to ensure new users requests can be handled with acceptable quality of service². Provided the number of different customizations is limited, this approach can be acceptable, but as the number of application variants increases the cost of using this approach increases as well.
- *Feature-Based Binary (FBB) approach:* In the feature-based approach, an application is split into multiple interacting components. These components are implemented as services that provide a distinct part of the application functionality. Service instances are associated with individual features, and the application is composed out of these resulting feature instances by using a SOA. Individual feature instances provide a specific, well-defined part of the application functionality, and as all applications making use of a feature make use of the same customizations, resources can be shared within these instances. Using this approach the customization is achieved by changing the services that are active and by the way in which the services are composed.

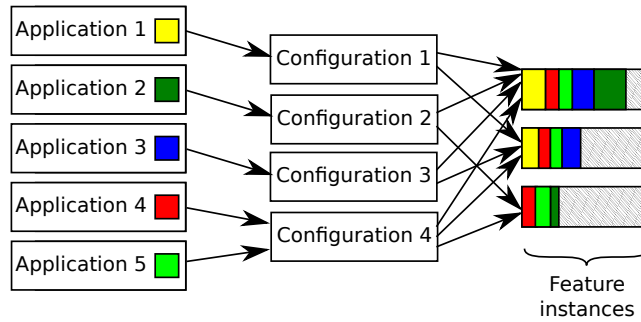
Both approaches still support some multi-tenancy: in the ABB approach this is by sharing resources between identically customized application, while in the FBB approach this is done by ensuring the feature instances themselves are multi-tenant. Figure 2.2 shows an illustrative example of both approaches. In this example, there are five applications, of which two make use of an identical feature configuration. In the ABB approach this results in four different instance types, one for every configuration. When the FBB approach is used, the number of instances is not dependent on the number of configurations, but rather on the number of different feature instances; in the example, we assume the application is composed out of three different feature instances. In the sample scenario, there are more application variants than there are features, which is why the FBB approach results in more multi-tenancy.

In practice, it is often the case that there are more potential application variants than there are application features, in this case making it preferable to a feature-

² If no instance would be active, a new instance would have to be created when a user request is received. As creating new application instances may require considerable time, this would result in unacceptable performance.



(a) Load on application instances created using the ABB multi-tenancy approach. Multiple applications may only use resources of a single application instance when their customizations are identical.



(b) Load on application instances created using the FBB multi-tenancy approach. Multiple applications can share the resources offered by a single feature instance if the feature is part of both applications, even when their configuration differs for other features.

Figure 2.2: An illustrative example comparing the ABB and a FBB multi-tenancy approaches, showing how applications can be assigned to shared multi-tenant application and feature instances. Every application and feature instance is contained in a separate VM. When only a few applications can be assigned to a single instance, server capacity assigned to the VM may be wasted, as then the VM may not fully utilize its available CPU or memory.

based approach. The FBB approach can also be modified to incorporate some properties of the ABB approach. This results in a hybrid approach that can support both feature-based and application-based instances. We will discuss this hybrid approach later in this section.

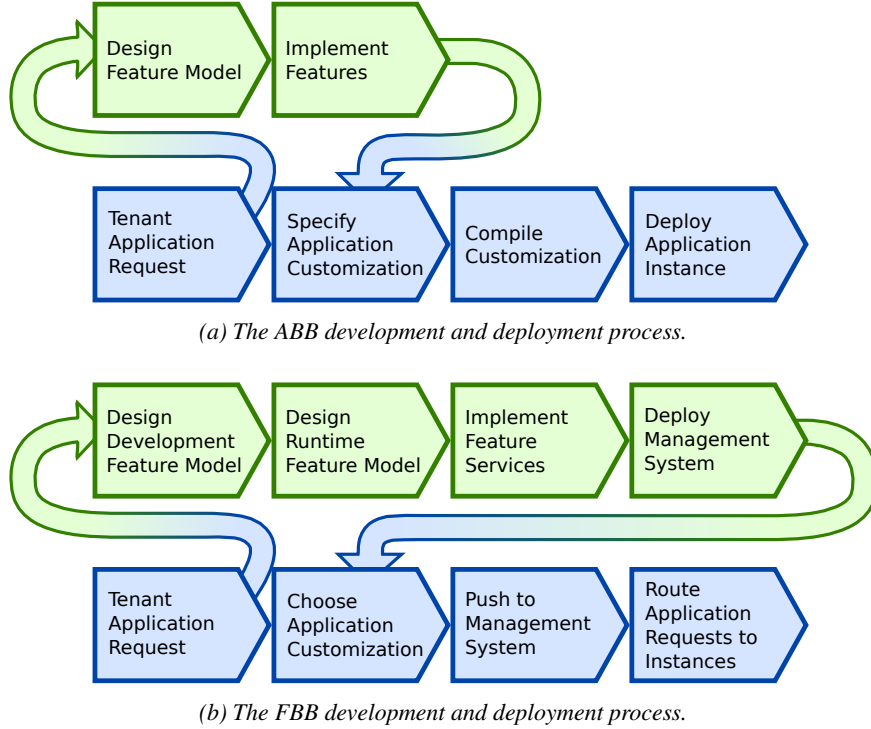


Figure 2.3: The processes for developing and deploying multi-tenant feature-based SaaS applications using the ABB and FBB approaches.

2.4.1 Application development, deployment and management

The processes for developing, deploying and managing applications differ between both approaches. Figures 2.3a and 2.3b show the processes for respectively the ABB and FBB approaches. When developing and deploying new applications, the topmost processes are executed. When new applications are instantiated for a tenant, the second processes are used. In some cases, new features may still have to be implemented for specific very large tenant organizations when they request new application instances, making it possible for the deployment workflow to be interrupted by an additional development phase.

2.4.2 ABB applications

The process for developing ABB applications is straightforward, as shown in Figure 2.3a: first a feature model is defined, then the features are implemented. For this process traditional SPLE approaches can be used.

Deploying applications is done in multiple steps:

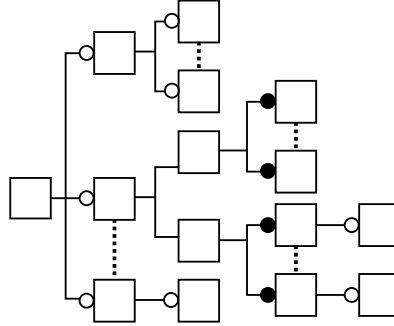
1. The process starts when a tenant requests a new application or requests changes to an existing application. For new tenants, this request is typically made by a seller employed by the platform provider. It may also be possible for existing clients to make these requests themselves using a configuration interface. When new organizations are added, it is possible for them to have new requirements that are not yet supported by the current application. If this is the case, and depending on the potential profit, a management decision may be made to implement these requirements, in which case the development process is started to first update the application by adding the additional features.
2. Next, a tenant configuration interface is used to specify the application configuration. This interface can be used to specify the features that are included in the tenant application and their configuration.
3. Once the customizations are selected, it is possible to compile the application instance. This is only needed if no instance with an identical configuration exist.
4. The application instance is deployed in the cloud environment. Managing applications can then be done using standard management techniques for multi-tenant applications such as [15]. If an identical application³ already exists, the application instance is not allocated; instead, the management system reconfigures the existing instance to offer it to the new tenant using multi-tenancy.

2.4.3 FBB application development

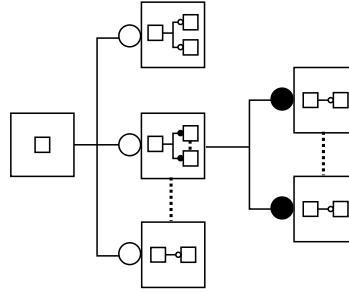
Developing applications using the FBB is a process that requires four steps as shown in Figure 2.3b:

1. First, an application feature model is defined. This model contains all the features that are defined by the application, and thus encompasses all the customization and configurations that a tenant may request.
2. The development model can then be analyzed to determine how each of the features can be implemented. This can be done by classifying them into compile time and runtime features. The former features are features that impact performance or require distinct code modules, and therefore require distinct application instances at runtime, while the latter features can be implemented by configuration or AOP changes to an instance at runtime. Based on this, a model where all runtime features are removed can

³ In this case an identical instance is one which contains the same static customizations. Two identical instances may still differ in functionality if they have different runtime weaving or configuration changes.



(a) A development feature model containing compile time and runtime variability.



(b) A collection of services that each deliver part of the functionality of an application. Features may or may not be customizable themselves (illustrated here by showing smaller feature models within the features), but this customization must be provided using runtime changes only.

Figure 2.4: An illustrative example of a development and runtime feature model. Both models offer the same customizations, but in the development model all features are contained within the model while the runtime model contains only compile-time customizations requiring separate feature instances. The runtime changes are handled by ensuring the feature instances themselves are customizable as well.

be determined. All of the features in this model are thus associated with specific code modules.

3. The services that are defined in the runtime feature model can then be implemented.
4. Finally, the service binaries and runtime feature model are pushed to the management system running within the cloud environment. Once this is done, it is possible to deploy the application services, but there are no applications making use of the newly developed features yet.

An important and complex step in the application development is the removal of runtime changes from the development feature model. To achieve this, all of the

features that can be provided at runtime are stripped from the model, resulting in a smaller runtime feature model containing only the features for which separate instances are needed. Each of these features can then be implemented as a separate instance that realizes specific application functionality. These feature instances may themselves be customizable, but these customizations may not require compile-time changes. The model transformation is illustrated in Figure 2.4, where we show how an example development feature model (Figure 2.4a) may be transformed into a runtime feature model (Figure 2.4b) where some features are realized by providing them using different service instances while others are realized by runtime variation of instances at runtime.

This model transformation can be a manual step during application development, but for changes that can be represented using AOP aspects that need to be applied to components, this approach can be automated. We described an approach for automated feature conversion in [16]. In this approach it is possible to automate the conversion of a feature model containing code modules and aspects applying to specific components to a runtime feature model where all application features refer to a code module.

In the presented approach, every feature in the runtime feature model is associated with a code module that is used to instantiate the feature. Some features may however be defined purely to add structure to the feature model. To make this possible, developers may define *empty* features. These features are not associated with code modules and including them does not create new instances.

2.4.4 FBB application deployment

Once the application is developed, new instances can be deployed for clients. Typically, this process is done in the four steps shown in the second process in Figure 2.3b:

1. First, a tenant may request a new application or modifications to an existing application. Like for ABB applications this request may be processed at once if all of the requested features already exist. Alternatively, this may also lead to a new development cycle where additional features are defined and implemented.
2. A tenant configuration interface is used to specify the application configuration. This interface is based on the development feature model (containing all of the changes) and may be generated automatically based on the feature model.
3. The new application configuration is then pushed to the management system. This management system then allocates resources on existing feature

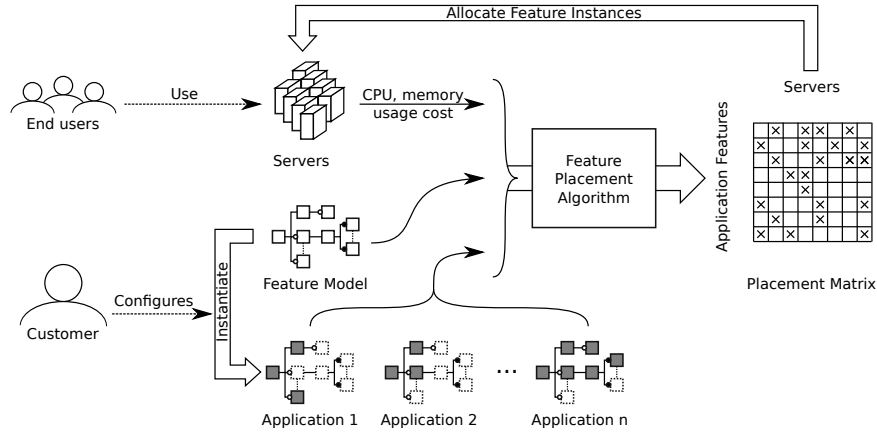


Figure 2.5: The FBB approach management system. The feature placement algorithm is responsible for allocating resources for the various application instances.

instances or instantiates additional feature instances to accommodate the application.

4. The application is then deployed and available.

An advantage of using a runtime feature model to represent application components is that it is possible to define open variation points [7]: features not just be either selected or excluded, but they may also remain undecided. This makes it possible to defer some decisions until runtime, reducing resource costs.

2.4.5 FBB application management

The cloud management system is responsible for allocating resources for the various feature instances that must be deployed in the FBB approach. The runtime feature model is known by the cloud management system, and is used by the management system to determine the features that are included in application. The final feature configuration of an application is dependent on the feature model, selected features, excluded features, and how open variation points are filled in by the management system.

Due to the presence of open variation points, there may be multiple possible feature configurations for an application, which results in interesting opportunities for reconfiguration of applications at runtime. This in turn results in multiple benefits. 1) It is possible to reduce the number of active instance types by preferring configurations that are already deployed within the cloud system. This increases multi-tenancy and reduces resource costs. 2) Similarly, it is possible to choose a feature configuration that results in the quickest deployment, which can also be

done by reusing feature instances that already exist and minimizing the number of new features that must be deployed.

To achieve these benefits, it is important to make use of management algorithms that are feature-aware. These management algorithms combine cloud application placement algorithms [15, 17, 18], which are used to allocate resources in clouds, with feature-awareness. The resulting feature placement algorithms can both determine an optimal application feature configuration and the placement of feature instances on servers in the cloud environment.

Figure 2.5 shows the how the feature placement algorithm functions. The feature placement is aware of the servers that are active within the cloud datacenter, the feature model, and the applications that make use of this feature model. Based on this information, a feature configuration can be determined for the applications, and an allocation can be determined indicating which feature instance is allocated on which server for which application. This application placement is then executed on the servers. These feature placement algorithms are discussed in-depth in Chapters 3 and 4.

2.4.6 Hybrid multi-tenancy

As mentioned previously, the FBB approach results in fewer application instance types provided that the number of features is lower than the possible number of different application variations. In some cases it may however be preferable to use ABB applications rather than FBB applications:

1. If the number of custom variants is lower than the number of features, using the FBB approach would result in more different instance types compared to the application-based approach.
2. If a specific variant has a very large number of users, or has many tenants using it. In this scenario it may be preferable to create a single instance implementing the variant as this results in lower network demand and communications, and thus conversely may result in lower costs and higher performance.
3. If a specific variant may only be used by multiple end users of the same tenant due to security reasons. In this scenario deploying separate instances for every feature results in more active application than in a scenario where a single instance is generated.

It is possible to extend the feature-based approach to also support ABB application variants. This can be achieved by adding a root node r' as a parent of the original root node r of the feature model. The set of specific application instances I is then made. Every instance $i \in I$ is a feature that links to a code module containing the entire application with specific set of customizations as a single

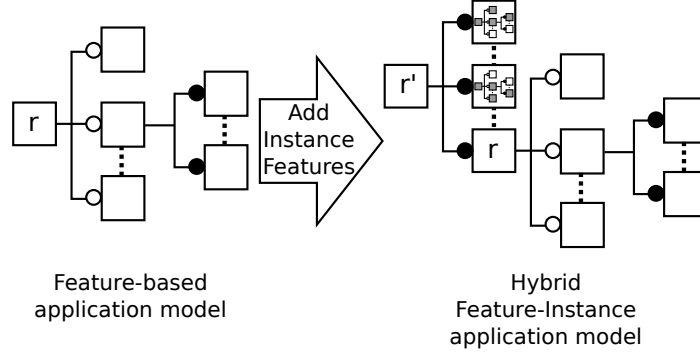


Figure 2.6: By adding specific application customizations as separate instances to the feature model, a hybrid approach using both FBB and ABB approaches can be used.

instance. By adding the original root feature of the feature model r' and the instance features I as using an alternative relation, a new feature model can be constructed. This feature model contains all the relations of the original feature model with the addition of the new **Alternative**($r', I \cup \{r\}$) relation. Using this approach, a model is constructed where every instance is either an instance of the feature-based approach, or of one of the chosen ABB instances.

This hybrid approach, further illustrated in Figure 2.6 combines properties of both the FBB and ABB approaches. The hybrid multi-tenancy approach still results in a runtime feature model with multiple services, and is thus equivalent to the FBB approach with a larger feature model. By combining hybrid application models with feature placement algorithms and open variation points interesting synergies can be discerned: the feature placement algorithm can decide the optimal application configuration by either deploying the application as a single instance or by deploying the application as a collection of feature instances depending on which option results in the lowest resource cost.

2.5 Analysis

We compare the ABB, FBB and hybrid approaches theoretically by comparing the number of instance types that may be generated by them. As discussed in the previous section, resources can only be shared between identical instances. Therefore, there will be less multi-tenancy when more different instance types may be generated, making it preferable to have fewer possible instance types. We refer to the maximum number of instances that may be generated by a management approach for a feature model as the Maximum number of Instance Types (MIT). Note that the MIT results in a worst case scenario for a given feature model, and the number of instances active at any time may be lower: in the ABB approach there

will never be more different instance types than there are applications and in the FBB the management algorithm may exploit open variation points to reduce the number of instantiated instance types.

2.5.1 ABB instance count

In the ABB approach, the MIT is limited by the possible number of application variants as every variant is statically compiled and then allocated as a separate instance. Thus, to determine the potential number of different instances, the total number of valid feature selections of the application feature model must be determined. We represent the MIT in the ABB approach of a family of applications with feature model F by $\mathcal{A}(F)$.

\mathcal{R} represents the collection of all relations within the feature model, and $\mathcal{R}(f, \cdot)$ represents all relations in \mathcal{R} with parent feature f . In our analysis we focus, as stated previously, on feature models consisting of four types of relations: **Mandatory**, **Optional**, **Alternative** and **Or**. For a feature x , $\mathcal{A}^f(x)$ represents the MIT of the feature model containing x as the root feature and all of the subfeatures of w within the original feature model. For a relation x , $\mathcal{A}^r(x)$ represents the number of variations introduced by this specific relation. The total MIT is then computed by determining the number of variants of the root feature r , implying that $\mathcal{A}(F) = \mathcal{A}^f(r)$.

When a feature f only has a single relation r in which it is the parent, $\mathcal{A}^f(f) = \mathcal{A}^f(r)$. It may however occur a feature is a parent in multiple relations. In this case, each of these relations results in a collection of possible variants. As all of these relations are independent, the total number of variants can be computed combinatorially by multiplying the individual MIT counts. This is expressed formally in Equation (2.1).

$$\mathcal{A}^f(a) = \prod_{r \in \mathcal{R}(a, \cdot)} \mathcal{A}^r(r) \quad (2.1)$$

The MIT counts can be computed for the various relation types as follows. **Mandatory** relations do not cause additional variants as they must always be included if the parent feature is included. The child feature c will however itself result in multiple variations $\mathcal{A}^f(c)$, which is expressed in Equation (2.2). **Optional** relations either result in including the child feature (resulting in all possible variants of the child feature c , $\mathcal{A}^f(c)$) or in not including the child feature (resulting in an additional configuration and increasing the number of variants by 1); this is expressed in Equation (2.4). **Alternative** relations always result in exactly one child feature being included resulting in causing all of the variability of its child nodes to be included, which is expressed in Equation (2.4). The formula for the **Or** relation, shown in Equation (2.5), can be easily derived from Equations (2.1)

and (2.3) by observing that an **Or** is equivalent to a collection of **Optional** relations where one case, that where none of the child features are included, is removed.

$$\mathcal{A}^r(\text{Mandatory}(\mathbf{a}, \mathbf{b})) = \mathcal{A}^f(b) \quad (2.2)$$

$$\mathcal{A}^r(\text{Optional}(\mathbf{a}, \mathbf{b})) = \mathcal{A}^f(b) + 1 \quad (2.3)$$

$$\mathcal{A}^r(\text{Alternative}(\mathbf{a}, \mathbf{S})) = \sum_{s \in S} \mathcal{A}^f(s) \quad (2.4)$$

$$\mathcal{A}^r(\text{Or}(\mathbf{a}, \mathbf{S})) = \left(\prod_{s \in S} (\mathcal{A}^f(s) + 1) \right) - 1 \quad (2.5)$$

2.5.2 FBB instance count

For feature based applications the MIT, which is represented by $\mathcal{F}(F)$, is limited by the number of features. Therefore, $\mathcal{F}(F)$ equals the number of features in the feature model F . This value can also be computed making use of the feature hierarchy, similar to how this was done for application-based multi-tenancy, which is useful for comparing the theoretical performance of FBB with ABB. $\mathcal{F}(F)$ can be easily computed based on the same approach we previously used to compute MIT for the ABB approach. The MIT of a feature x is represented as $\mathcal{F}^f(x)$; the MIT of a relation x is represented by $\mathcal{F}^r(x)$. $\mathcal{F}(F) = \mathcal{F}^f(r)$ with r the root feature. The formulations for the various relation types can be computed trivially based on their definition:

$$\mathcal{F}^f(a) = 1 + \sum_{r \in \mathcal{R}(a, \cdot)} \mathcal{F}^r(r) \quad (2.6)$$

$$\mathcal{F}^r(\text{Mandatory}(\mathbf{a}, \mathbf{b})) = \mathcal{F}^f(b) \quad (2.7)$$

$$\mathcal{F}^r(\text{Optional}(\mathbf{a}, \mathbf{b})) = \mathcal{F}^f(b) \quad (2.8)$$

$$\mathcal{F}^r(\text{Alternative}(\mathbf{a}, \mathbf{S})) = \sum_{s \in S} \mathcal{F}^f(s) \quad (2.9)$$

$$\mathcal{F}^r(\text{Or}(\mathbf{a}, \mathbf{S})) = \sum_{s \in S} \mathcal{F}^f(s) \quad (2.10)$$

2.5.3 Hybrid instance count

The hybrid multi-tenancy approach extends the FBB approach and adds an additional root element r' , an **Alternative** relation, and at least one feature linked to an instance of a specific application customization. Therefore the hybrid multi-tenancy approach will always result in at least two more instance types than the FBB approach. Thus, the number of instance types $\mathcal{H}(F)$ of the hybrid approach can be computed by $\mathcal{H}(F) = \mathcal{F}(F) + n + 1$ where n is the number of added ABB

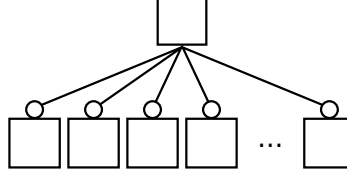


Figure 2.7: The worst case feature model for the ABB approach resulting in the maximum possible instance type count.

applications. In the evaluation and analysis, we will focus mainly on the ABB and FBB approaches as the hybrid approach can be considered as a special case of the FBB approach with a additional features and thus a slightly higher MIT.

2.5.4 Worst-case comparison

FBB and ABB multi-tenancy behave very differently depending on the feature models they are used with. The FBB approach always results in the same number of features for every feature model irrespective of the relations within the model as it is only dependent on the number of features within the model. Therefore, we consider the worst case for this approach to be the one where the model only results in few variations while requiring many features. For a feature model with n features containing only **Mandatory** features, FBB results in n different feature instance types, while ABB results in an MIT of only 1 as there is no customization. In this case, it would be preferable to restructure the feature model to reduce the number of features, possibly reducing the feature model to a single feature. This is however an edge case as there is no customizability in this scenario. Ignoring **Mandatory** relations, the **Alternative** relation performs worst for the FBB approach compared to the ABB approach. A feature model containing only a single **Mandatory(r, S)** relation results in $|S| + 1$ instance types in the FBB approach and only in $|S|$ variants in the ABB approach.

The worst case scenario for the ABB approach is when it is used for a flat feature model containing only **Optional** relations, as illustrated in Figure 2.7. This model results in 2^{n-1} possible variants with n the number of features within the model.

Theorem 1. *The worst case feature model, resulting in the maximum MIT for a given number of features, only contains **Optional** relations.*

Proof. We assume that a model F exists with the maximum $\mathcal{A}^f(r)$ that contains relations other than **Optional** relations. This implies there must be at least one relation r that is not an **Optional** relation. This relation can be one of three types:

1. **Mandatory(a, b)**: By replacing **Mandatory(a, b)** with **Optional(a, b)**, $\mathcal{A}^r(r)$ can be increased. This in turn increases the model MIT.

2. **Or(a, S)**: In this case, $\mathcal{A}^r(r) = (\prod_{s \in S} (\mathcal{A}^f(s) + 1)) - 1$. By replacing the **Or** relation by a set of **Optional(a, s)** relations for every $s \in S$, the number of variants will be increased by one, increasing $\mathcal{A}^f(a)$ and in turn increasing the MIT of the new feature model.
3. **Alternative(a, S)**: From Equations (2.4) and (2.5), and the fact that the set S must always contain at least two features, it can be concluded that replacing the relation **Alternative(a, S)** by **Or(a, S)** the MIT can be increased.

In each of the cases a new feature model F' can be constructed for which $\mathcal{A}(F) < \mathcal{A}(F')$, contradicting the assumption. \square

Theorem 2. *The feature model resulting in the maximum MIT for a given number of features is flat and consists of a root feature r and a set of relations **Optional(r, s)** for all features $s \in \{F/r\}$.*

Proof. Suppose the contrary that a feature model F exists that is not flat and that results in a higher MIT. This model must only contain **Optional** relations as proven in Theorem 1. This model must therefore contain a relation **Optional(n^r , a)**, with n^r the root node, and a feature a that is itself parent in one or more **Optional(a , s)** relations with $s \in S$ as child features. The contribution to the total MIT by the feature a and its subfeatures, represented as C^a can be determined using Equations (2.1) and (2.3):

$$C^a = \mathcal{A}^r(\text{Optional}(a, s)) = 1 + \prod_{s \in S} (\mathcal{A}^f(s) + 1)$$

An alternative feature model F' can be constructed where the features $s \in S$ are attached directly to the root node instead of to the feature a . The contribution of a and the features in S is then represented by C'^a (in this model, a no longer has child nodes, ensuring $\mathcal{A}^f(a) = 1$):

$$\begin{aligned} C'^a &= (\mathcal{A}^f(a) + 1) \times \prod_{s \in S} (\mathcal{A}^f(s) + 1) \\ &= 2 \times \prod_{s \in S} (\mathcal{A}^f(s) + 1) \end{aligned}$$

As $\mathcal{A}^f(f) \geq 1$ for all features f , we can conclude that $C^a < C'^a$, which due to Equation (2.1) ensures that $\mathcal{A}(F) < \mathcal{A}(F')$. Therefore, this new model results in more variations than the original model F , contradicting the assumption that the feature model F resulted in the maximum MIT given the number of features. \square

From this analysis, we conclude that if a model has many **Optional** or **Or** relations, or if a single feature is the parent in many relations, the FBB approach will generally result in fewer instance types than the ABB approach. **Mandatory** and **Alternative** relations may however work better in an ABB approach.

2.6 Evaluation results

As noted in the previous section, it is trivial to construct feature models where one approach is better than the other. It is however important to determine when which approach is preferable for realistic feature models. For our evaluations we make use of three feature models of commercial SaaS applications, a composed feature model, and randomly generated feature models based on the structure of the previous models.

We use the feature models of three commercial SaaS applications as a baseline for our evaluations. The models are that of a Medical Communications (*MC*) application, a Document Processing (*DP*) applications and a Medical Data Management (*MDM*) application. These models contain 12, 22 and 16 features respectively. Based on the application feature models we also define a composed feature model *Full* which contains all of the features of the *MC*, *DP* and *MDM* models composing them using an **Alternative** relation. This represents the situation where all of the previous applications are executed on the same application platform. The composed model contains in total 51 features.

We also defined a collection of randomly generated feature models of varying sizes to compare the approaches for feature models of differing sizes. The models are generated ensuring they are similar to the *MC*, *DP*, *MDM* and *Full* models in terms of structure and frequency of relation types. First the set of features is generated and one feature is selected as root of the feature model tree. Next the other features are iteratively added to the feature model by selecting a random node as the parent and one or more of the remaining features are added as child nodes. We use an equal probability for selecting any of the relation types, with **Optional** and **Mandatory** relations having (by definition) one child while **Alternative** and **Or** relations have between 2 and 6 features as child nodes (chosen uniformly at random).

Figure 2.8 compares the number of instance types that may have to be deployed within a cloud environment for the four application feature models. For the four cases the MIT is lower for the FBB approach than it is for the ABB approach. The largest difference is observed for the *DP* where the ABB results in 22 times the number of possible instances while the *MDM* case results in the smallest increase (4.5 times as many possible instances as the FBB approach).

By using randomly generated feature models, we can further evaluate the behavior of both approaches. Figure 2.9 compares the number of variants for the ABB and FBB approaches for varying numbers of features. For the FBB approach the number of instance types always equals the number of features, for the ABB approach the number of instance types depends on the used feature model. For every data point 10000 randomly generated feature models were used, which results

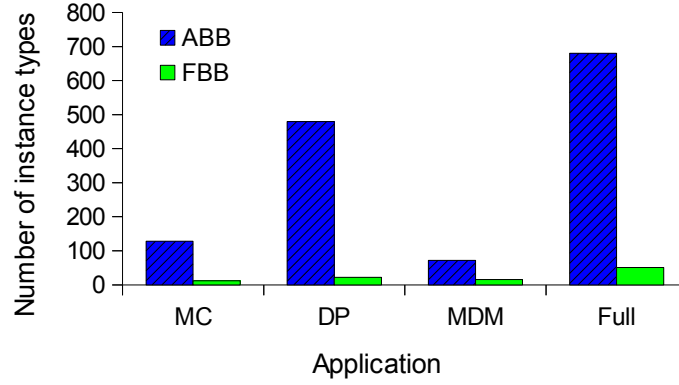


Figure 2.8: The MIT for the ABB and FBB approaches for three commercial applications, MC, DP and MDM, and a composed model containing all three models, Full.

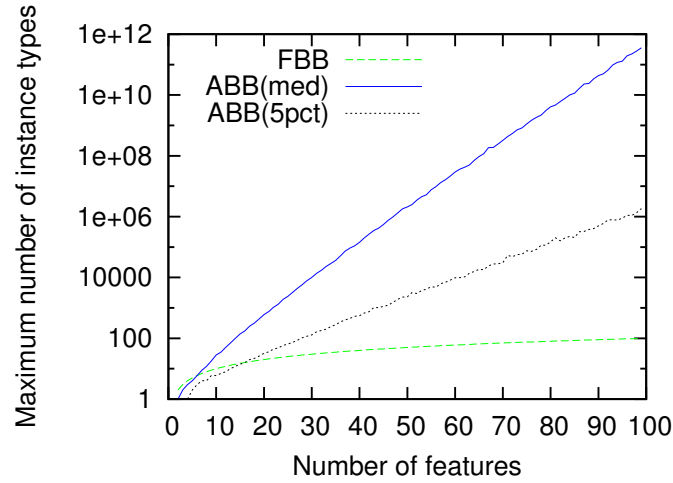


Figure 2.9: The median of MIT for the ABB and FBB approaches for varying numbers of features (10000 feature models per data point). The 5th percentile of the ABB values is shown to give an indication of the distribution of the number of instance types for the ABB approach.

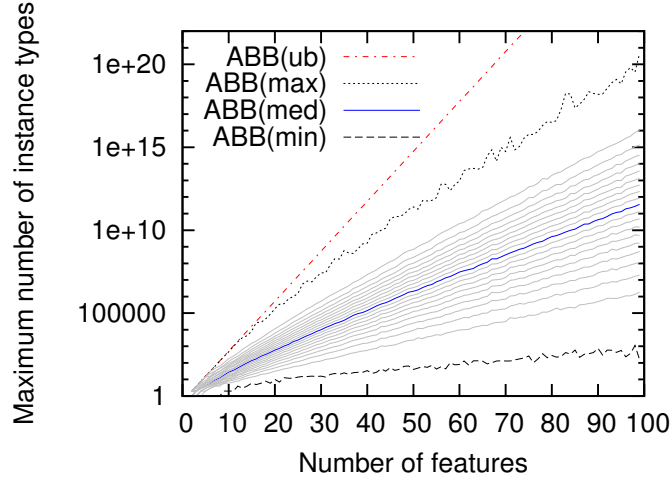


Figure 2.10: The distribution of MIT for the ABB approach for varying numbers of features (10000 feature models per data point). The theoretical upper bound (ub), maximum (max), median value (med) and minimum (min) are shown. The gray lines show the percentiles of the distribution (5th until 95th in increments of 5).

in a large spread of resulting values. Figure 2.10 shows the distribution of the number of instance types for the evaluation.

As the number of features increases, the FBB approach results in much fewer instance types than the ABB approach. On average, the ABB approach only results in fewer instance types for feature models containing 4 or less features, but the average is heavily skewed by outliers. For the median feature model, the ABB approach performs best for models with less than 7 features. For larger models, the FBB approach results in fewer variants.

2.7 Discussion

The analysis and evaluations show that the FBB generally results in fewer distinct application instances compared to the ABB approach, as measured by the MIT metric. For very small feature models containing 6 or fewer features, the ABB approach may however be preferable. If the feature model contains many mandatory and alternative relations, the ABB approach may also result in a lower MIT value.

The hybrid approach is an extension to the FBB approach, adding specific application configurations implemented using the ABB approach. Because of this, the hybrid approach can be used in all scenarios where the FBB approach may be

used, resulting in a slightly higher MIT (depending on the number of application configurations that are added using the ABB approach). In a scenario where the ABB results in fewer instances than FBB, however, these variants can be added to the model, resulting in fewer instantiated instances at runtime compared to the FBB approach. The hybrid approach may also be useful for application configurations that are used by many tenants if the single-instance application requires fewer resources, which may be the result of a reduction in communication overhead.

The most important disadvantage of the FBB approach is that it becomes more difficult to manage applications, as more communication between components must be taken into account. This disadvantage is shared with the hybrid approach. A second disadvantage of the FBB approach is that more application instances are needed to offer an application, which may be disadvantageous if some components are rarely used causing the instances to be underutilized. This problem can be mitigated using the hybrid approach as, in such cases dedicated instances can be defined.

2.8 Conclusions

In this chapter, we formalized two approaches for managing variability in multi-tenant SaaS environments: FBB, a SOA-based which composes the application out of multiple multi-tenant components and ABB which statically generates multiple distinct multi-tenant applications based on a common feature model. We described how applications can be developed, deployed and managed using both approaches. We also presented a hybrid approach, combining beneficial properties of both the ABB and FBB approaches.

We found that, for feature models with more than 6 features requiring compile-time changes, the FBB approach results in fewer possible runtime instance types, which in turn results in more opportunities for exploiting multi-tenancy and lower costs. For models with fewer features, the ABB approach will perform better.

Acknowledgments

Hendrik Moens is funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT).

References

- [1] B. Morin, F. Fleurey, N. Bencomo, J.-M. Jézéquel, A. Solberg, V. Dehlen, and G. Blair. *An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability*. In *Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 782–796. Springer Berlin Heidelberg, 2008. doi:10.1007/978-3-540-87875-9_54.
- [2] S. Hallsteinsen, M. Hinchey, and K. Schmid. *Dynamic Software Product Lines*. *Computer*, 41(4):93–95, apr 2008. doi:10.1109/MC.2008.123.
- [3] M. Hinchey, S. Park, and K. Schmid. *Building Dynamic Software Product Lines*. *Computer*, 45(10):22–26, oct 2012. doi:10.1109/MC.2012.332.
- [4] R. Capilla, J. Bosch, P. Trinidad, A. Ruiz-Cortés, and M. Hinchey. *An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry*. *Journal of Systems and Software*, 91(0):3–23, may 2014. doi:10.1016/j.jss.2013.12.038.
- [5] C. Parra, X. Blanc, and L. Duchien. *Context awareness for dynamic service-oriented product lines*. In *Proceedings of the 13th International Software Product Line Conference (SPLC 2009)*, pages 131–140. Carnegie Mellon University, aug 2009.
- [6] R. Mietzner, T. Unger, R. Titze, and F. Leymann. *Combining Different Multi-tenancy Patterns in Service-Oriented Applications*. In *Proceedings of the 2009 IEEE International Enterprise Distributed Object Computing Conference (EDOC 2009)*, pages 131–140. IEEE, sep 2009. doi:10.1109/EDOC.2009.13.
- [7] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl. *Variability modeling to support customization and deployment of multi-tenant-aware Software as a Service applications*. In *Proceedings of the ICSE Workshop on Principles of Engineering Service Oriented Systems (PESOS 2009)*, pages 18–25. IEEE, may 2009. doi:10.1109/PESOS.2009.5068815.
- [8] L. Baresi, S. Guinea, and P. Liliana. *Service-Oriented Dynamic Software Product Lines*. *Computer*, 45(10):42–48, aug 2012. doi:10.1109/MC.2012.289.
- [9] S. Walraven, D. Van Landuyt, E. Truyen, K. Handekyn, and W. Joosen. *Efficient customization of multi-tenant Software-as-a-Service applications with service lines*. *Journal of Systems and Software*, 91:48–62, may 2014. doi:10.1016/j.jss.2014.01.021.
- [10] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. D. Turck. *Feature Placement Algorithms for High-Variability Applications*

- in Cloud Environments*. In Proceedings of the 13th Network Operations and Management Symposium (NOMS 2012), pages 17–24. IEEE, apr 2012. doi:10.1109/NOMS.2012.6211878.
- [11] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. *Cost-Effective Feature Placement of Customizable Multi-Tenant Applications in the Cloud*. Journal of Network and Systems Management, 22(4):517–558, oct 2014. doi:10.1007/s10922-013-9265-5.
- [12] K. Pohl, G. Böckle, and F. Van Der Linden. *Software product line engineering: foundations, principles, and techniques*. Springer-Verlag New York, Inc., 2005.
- [13] pure-systems GmbH. *pure::variants User’s Guide* [online]. 2012. Last accessed: December 2012. Available from: <http://www.pure-systems.com/Documentation.116.0.html>.
- [14] W. Sun, X. Zhang, C. J. Guo, P. Sun, and H. Su. *Software as a Service: Configuration and Customization Perspectives*. In IEEE Congress on Services Part II (services-2), pages 18–24. IEEE, sep 2008. doi:10.1109/SERVICES-2.2008.29.
- [15] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici. *A scalable application placement controller for enterprise data centers*. In Proceedings of the 16th International Conference on World Wide Web (WWW 2007), pages 331–340. ACM, may 2007. doi:10.1145/1242572.1242618.
- [16] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. *Developing and Managing Customizable Software as a Service Using Feature Model Conversion*. In Proceedings of the 3rd IEEE/IFIP Workshop on Cloud Management (CloudMan 2012), pages 1295–1302. IEEE, apr 2012. doi:10.1109/NOMS.2012.6212066.
- [17] C. Adam and R. Stadler. *Service Middleware for Self-Managing Large-Scale Systems*. IEEE Transactions on Network and Service Management, 4(3):50–64, dec 2007. doi:10.1109/TNSM.2007.021103.
- [18] B. Urgaonkar, A. L. Rosenberg, and P. Shenoy. *Application Placement on a Cluster of Servers*. International Journal of Foundations of Computer Science, 18(05):1023–1041, oct 2007. doi:10.1142/S012905410700511X.

3

Cost-Effective Feature Placement of Customizable Multi-Tenant Applications in the Cloud

H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt and F. De Turck

Published in Journal of Network and Systems Management (JNSM), 2014

In Chapter 2 the Feature-Based Binary (FBB) approach for modeling and managing customizable multi-tenant SaaS applications was introduced. In this chapter, we use this approach to define a feature-based cloud resource management model. The focus of this chapter is on how feature-based multi-tenant SaaS applications can be allocated in a cost-effective way in a datacenter, a problem which we refer to as the feature placement problem. A formal description of this problem, which is used to allocate resources in a cost-effective way, is provided. Both the cost of failure to place features, and the cost of using servers are considered, making it possible to take energy costs or the cost of using public cloud infrastructure into consideration during the placement calculation. Four algorithms for solving the static feature placement problem, which takes all applications into account at once, are defined, evaluated, and compared with an optimal solution. In Chapter 4 the concepts introduced in this chapter are extended to support dynamic datacenter scenarios and in Chapter 5 these management algorithms are framed within an overarching development and management approach.

3.1 Introduction

In recent years, there has been an increasing interest in cloud computing [1]. By moving applications to cloud platforms, and making use of multi-tenancy, where multiple end users utilize the same application instances and hardware, administrators can consolidate hardware and save costs. Cloud-hosted applications can also react faster to sudden changes in demand. Different obstacles to the widespread adoption of cloud computing do however still exist. One of the issues with contemporary cloud Software-as-a-Service (SaaS) offerings is that the applications generally offer a one-size-fits-all package, with only limited customizability. Often it is only possible to add minor changes using configuration changes. Software customizability, where entirely separate code paths are executed in different software versions, significantly changing the behavior of applications, is difficult to add to SaaS applications.

Often, applications must however be tailored for specific customer needs, offering similar but slightly differing functionality for different end users. The CUSTOMSS [2] project seeks to create solutions to develop, deploy and manage highly customizable software and services on multi-tenant cloud infrastructures, by incorporating management of the variability of applications into the cloud platform itself. Within the project we focus on applications from three domains: 1) document processing, in which large batches of documents are processed and managed using a web interface; 2) medical information management, where medical data and patient information are stored and processed; and 3) medical communication systems, where communication between patients and nurses is coordinated based on a management system using advanced ontologies. While we focus our evaluation on these three use cases, the presented approach could be applied to all cloud-based applications that require high variability, provided the applications can be split into interacting components. The techniques can either be applied on top of an Infrastructure as a Service (IaaS) or Platform as a Service (PaaS) platform, or can be integrated into existing PaaS platforms. In this chapter, we will discuss how customizable multi-tenant applications can be managed by a cloud platform.

Software Product Line Engineering (SPLE) [3] concepts are often used to develop customizable applications. In this approach, the software is modeled as a collection of features. By selecting and deselecting features, different software variants can be created. Features themselves are organized by relating them to each other in a *feature model*. These techniques can however not easily be adapted to a cloud context, as most approaches in SPLE focus on the develop-

ment of statically configured products, where changes are compiled into the application. In this approach, all variations are instantiated and compiled before a product is delivered to customers and, once the decisions are made, it is difficult for users to alter them. When used in a cloud context, this implies that every software variant would be an entirely separate application, making it impossible to use multi-tenancy where instances are shared between users if these users do not use the same variant, thereby greatly reducing the potential cost savings of a migration to the cloud.

An alternative approach [4], where the software is split up into separate services using a Service-Oriented Architecture (SOA), and where the individual services are multi-tenant alleviates this shortcoming, but in this case, some services risk being underutilized, especially if many features and variants exist. Furthermore, as these services are dependent on each other, failure of a single service can result in performance degradations for the entire application, which can not be taken into account by current cloud resource allocation mechanisms.

By adding variability information to the applications running on a cloud platform, and managing variability at this platform level, developing highly customizable SaaS applications becomes easier. One very important functionality of the platform, is to decide which applications are executed where. This is known as the application placement problem [5, 6]. Current application placement techniques are however inadequate for this purpose, as they do not take relationships between services, introduced by variability modeling, into account. For our purposes, the placement must take the variability of the managed applications into account, ensuring all application components are allocated sufficient resources. Furthermore, the cost of using servers for running applications must be taken into account as well, as this makes it possible to either minimize the carbon footprint of the managed cloud, or to reduce costs when part of the application is executed on public cloud infrastructure. Within this chapter, we consider two costs: the cost of failing to provision capacity for application components (determined e.g. by a service level agreement) and the operational cost of using a server.

In this chapter, we focus on the design of algorithms for placing high-variability applications on cloud infrastructure, extending the methods and evaluations from our previous work [7], adding energy efficiency and server usage costs, and incorporating relations between applications components. The applications are composed from a set of multi-tenant feature instances using a SOA. For this purpose we designed a variation of the application placement problem [8], which we refer to as the *feature placement problem*. An overview is

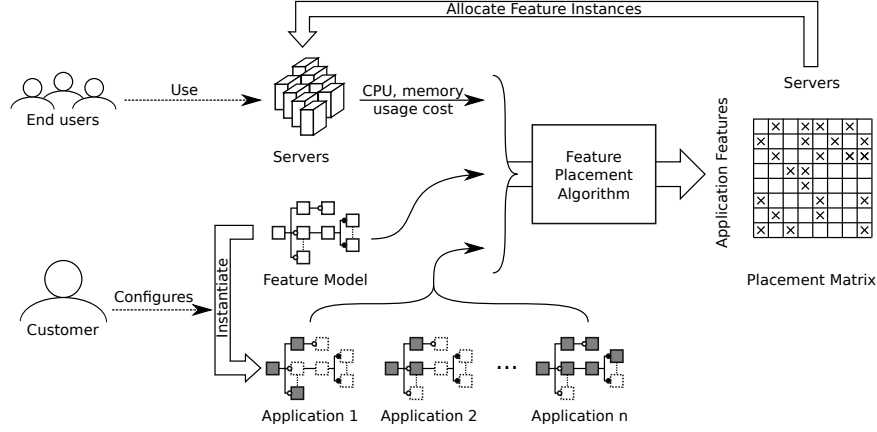


Figure 3.1: A schematic representation of the feature placement problem. Applications instantiate a feature model. Feature instances are placed on physical servers and can be used by multiple applications.

shown in Figure 3.1. The resulting feature placement determines which servers will execute which feature instances, taking into account the datacenter server configuration, applications to be placed, and the feature model of which the applications are instantiations. A single feature instance is capable of serving multiple applications, ensuring applications composed of a set of features are themselves multi-tenant. We address the following research questions: (i) How can we define the feature placement problem, and what information is required to define it? (ii) How can the feature placement problem be formally modeled? (iii) Which algorithms can be designed to solve this problem in an efficient way? and (iv) Which performance is achieved by the algorithms compared to the optimal solution, in terms of placement quality and execution speed, and what is the impact of model parameters on the obtained performance?

The contribution of this chapter is two-fold: (1) we describe how SPLE techniques can be combined with cloud application placement techniques to facilitate the management of high-variability applications; and (2) we formally define the feature placement problem, define optimal and heuristic algorithms, and evaluate them. In the next section, we will discuss related work. Afterwards, in Section 3.3 we explain the feature modeling approach, and how it can be applied to cloud applications. We then formally define the feature placement problem in Section 3.4. This is followed by Section 3.5, where we outline different approaches to solve the placement problem. In Section 3.6 we describe the set-up of the evaluation. Subsequently, in Section 3.7 we evaluate the algorithms. Both the quality of the results of the algorithms, and their execution speeds are discussed. Finally, Section 3.8 contains our conclusions.

3.2 Related work

This work builds on two research areas: SPLE and feature-oriented application development, and application placement.

3.2.1 Software Product Line Engineering

SPLE is used to manage the variability of applications, making it easier to build and manage groups of similar applications, with different feature sets. Managing a separate codebase for every software variant family would introduce a large overhead. Instead, only a single codebase is used, in which the variability is managed using SPLE techniques. Research has been done on configuration policies and methodologies to support customizations of SaaS. In [9], Zhang et al. discuss a policy-based framework for publishing customization options of web services and building customizations on top of this, enabling clients to build their own customizations. They however do not take multi-tenancy and runtime aspects into account, nor do they propose a software development methodology to create the customizable applications. Sun et al. [10] propose an approach choosing configuration over customization to create modifiable applications, and propose a software development methodology to develop such applications. We, by contrast, focus on the customization aspect by using SPLE methods in combination with a SOA development approach. In Mietzner et al. [4] an approach for modeling customizable applications built using SOA is described. The application is linked to a feature model, allowing automatic generation of deployment scripts. Our approach is similar in its use of SOA in the proposed development approach. We however focus on the resource allocation of customizable applications, proposing optimal and heuristic algorithms to determine where to run specific features. Recent work in the SPLE community [11–14] further progresses towards the development of customizable SaaS applications, but mainly focuses on the design-time variability of these applications, and not on their runtime management. Work on the dynamic adaptation of SOA applications was conducted in [15], but it does not address how these applications must be placed on physical infrastructure.

3.2.2 Application Placement

The application placement problem is used within clouds and clusters to determine which services to execute on which servers, and has previously been formally described [5, 6, 8, 16–18]. Many different approaches to application placement in clouds have been developed over the recent years. Specific requirements have however led to the creation of many extensions to the application placement problem, each focusing on different parameters. Whalley et al. [19] extended a Virtual Machine (VM) management system to take into account the complexities of soft-

ware licensing. In a similar way, Breitgand et al. [20] added the consideration of Service Level Agreements (SLAs) to the placement problem. The consideration of energy consumption and carbon emissions was added in [21] using a system that works in parallel with existing datacenter brokering systems. We extend the generic application placement problem formulation to place the features of applications in a cloud environment. Our approach further differs from the traditional application placement problem formulation and its variants, as we consider an application to be a set of interacting services, and not just a single service. By contrast to the existing work surrounding application placement, our placement approach not only takes these services, but also the relations between them into account during the placement calculation.

The algorithm we describe within this chapter has similarities with the linear application placement algorithm described in [16]. Our work however adds the concept of software variability. Furthermore, our application-based feature placement algorithm aims to place all application components at once and adds a backtracking phase to the algorithm if placement of an application fails, lowering the cost of placements.

Energy efficiency and server usage costs are incorporated in an application placement system in [22]. The authors however focus on the placement at a VM level, while our approach focuses on managing multi-tenant applications where multiple applications can make use of a single instance, meaning more fine-grained control is needed. Furthermore, our algorithm also adds explicit support for software variability. This enables the management system to dynamically fill in undecided variability, known as open variation points, at runtime.

In [23], the concept of application component placement is introduced, where applications consisting of separate components are placed within datacenters, and an integer linear programming algorithm to solve the problem is introduced. Our approach similarly focuses on applications consisting of multiple components, but we by contrast add support for multi-tenancy, making it possible for multiple tenants to make use of individual application components. Additionally, we also take relations between application components, modeled using SPLE, into account during the placement.

We have previously discussed the runtime management of, and resource allocation for highly customizable applications [7]. In this chapter we extend the problem description, generalizing the inputs, and add a server use cost, ensuring energy efficiency and hybrid cloud scenarios can be taken into account. We also incorporate requirements that improve the problem applicability, ensuring the algorithm can better handle scenarios where memory requirements increase when the loads increase, and situations where features depend on each other to function. We also present and evaluate an improved, application-centric placement algorithm yielding more cost-effective resource allocations than the algorithms described in our previous work.

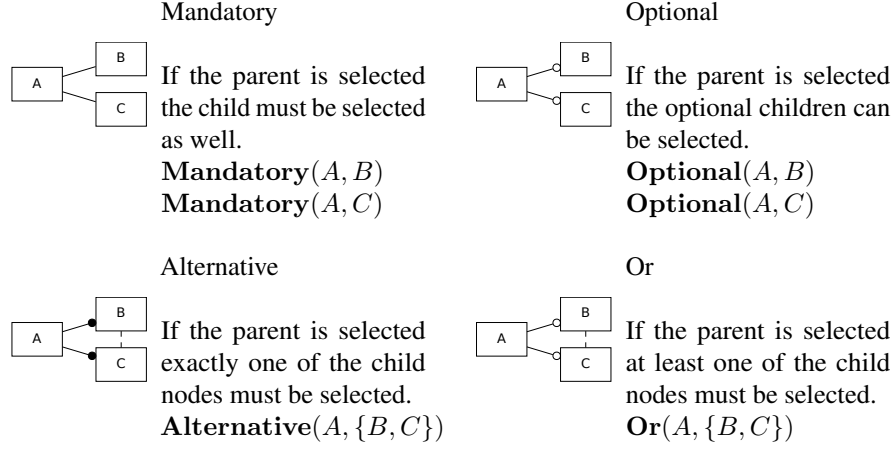


Table 3.1: Graphical representation of feature models, description of relations, and formal representation. The nodes on the left are parent features, those on the right are child features.

In literature, most application placement algorithms make use of specific resources, usually taking into account CPU and memory limitations [5, 8, 24, 25], application bandwidth requirements [26], or generalized load-dependent and load-independent resources [27]. Our approach generalizes these inputs, as done in [27], but goes further by allowing for the definition of multiple resources. This is achieved by making use of concepts we previously described in [28], enabling the management of high-variability applications with heterogeneous resource demands.

In [29] and [30], a management system for services composed of multiple VMs is presented. These works focus on the definition and deployment of composed cloud services. Our work is complementary with this approach, as it focuses on the relations between the different services using SPLE techniques and not on how these relations are represented. We also focus on the physical location where the instances are executed, rather than how they are deployed.

3.3 Feature placement concepts

Using SPLE, an application is modeled as a collection of features and relations between these features. The features are then linked to actual code modules or configuration files. Sometimes the inclusion of a feature can imply the inclusion or exclusion of other features, which is represented using relations in the feature model. A software variant can then be generated by selecting and excluding features from this feature model.

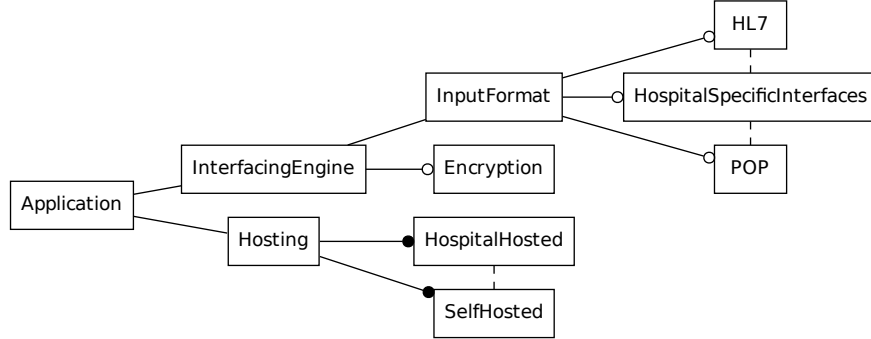


Figure 3.2: A feature model fragment for a medical data processing application.

To facilitate reasoning on these relations, feature models are often created hierarchically. Table 3.1 contains the different relation types, a description, a graphical representation, based on the notation used in [31], and a formal notation which will be used later on in this chapter. An example feature model is shown in Figure 3.2. The figure shows an illustrative fragment of the feature model for a medical data processing application. The application contains an *interfacing engine* feature to connect to individual hospitals, which is capable of handling input in one or more different formats. Additional *encryption* can optionally be added to the interfacing engine. Finally, parts of the application can be hosted at the hospital or they can be hosted by the application provider. An application created for a hospital using their own datacenter and a hospital specific interface will differ significantly from the application created for a hospital using public cloud infrastructure and a standard medical data interface.

Sometimes a feature can be implemented by simply updating configuration files. This could for example be changing the logo of an application. More complicated changes can be created by adding code changes. The most complicated changes lead to completely different modules being used by the applications. The first method is variation by configuration, the latter two variation types are referred to as customization [10]. In this chapter, we only consider customization, which leads to the creation of applications that are different at the code level. Configuration-based features can already be adapted into a cloud context using existing software development techniques [10]. The feature models used further on in this chapter will only contain features that cause changes at a code level in the deployed services.

The development of applications will be driven using the feature model, building an application using a SOA, in which the individual services map to the different features defined in the feature model. An example of this is shown in Figure 3.3a. Deploying the application then comes down to allocating feature instances and con-

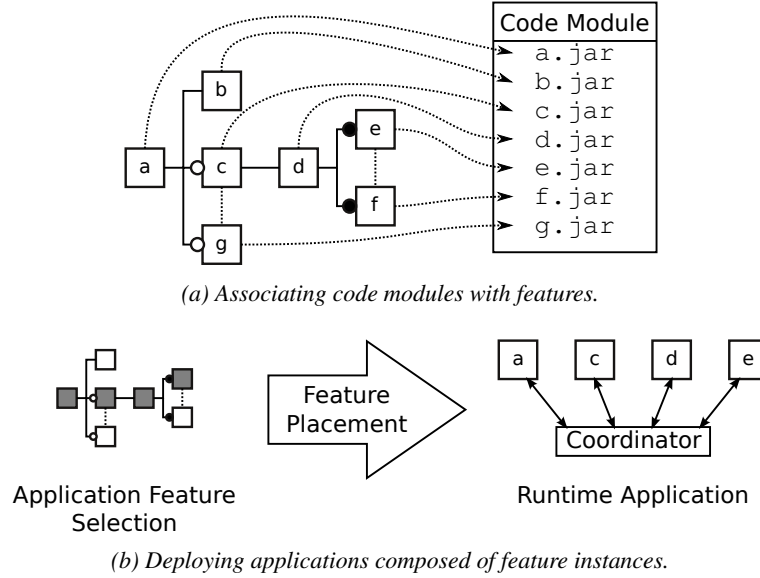


Figure 3.3: Features are associated with code modules. Applications containing the features are created by instantiating these features and linking them together.

necting them either to each other or by using a coordinator component, illustrated in Figure 3.3b. To determine where these instances are placed, a feature placement algorithm is used. We assume that the individual services are multi-tenant and can serve multiple applications. In our use cases, the various feature instances are developed, managed and tested by the platform provider, ensuring the components can be trusted and that calls to a service respect the tenant resource limits. This in turn minimizes performance interference between tenants which can be caused by sharing a feature instance between different tenants. The allocation of the different feature instances, taking into account relations as defined by the feature model, is the main focus of this chapter. We assume the application has already been split up into components, and that data isolation issues are resolved using existing techniques [32, 33]. We also assume the performance of the various components has been evaluated using performance models such as those in [34], possibly grouping components that often communicate together to guarantee they are colocated, which ensures good performance is achieved.

When configuring a SPLE application, part of the variability can be left undecided, creating open variation points [4]. When two applications with different feature configurations exist, and some have open variation points, this information can be used to reduce the cost of the full placement. This makes it particularly interesting to take these points into account during the placement of applications.

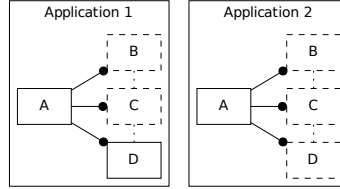


Figure 3.4: Different feature model selections for two applications. Features with a solid border are selected, features with a dotted border are undecided and remain open variation points. By selecting Feature D for Application 2 during placement, the total resource requirement of both applications can potentially be decreased.

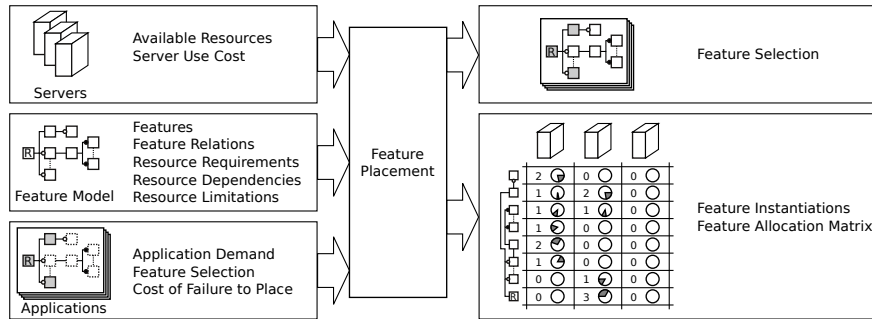


Figure 3.5: A detailed overview of the feature placement inputs and outputs. We use a small example feature model to illustrate the inputs. The root of this feature model is marked using the letter R.

An application with, e.g., regular availability requirements can use high availability instances when such instances exist with remaining capacity, rather than creating new instances with lower reliability, effectively lowering the total resource usage. This is further illustrated in Figure 3.4, where two applications are shown. The first application uses Feature D, and the second application requires either Feature B, C, or D. If the placement function is unaware of these open variation points, it would simply choose the cheapest alternative, while a choice for the, possibly more expensive Feature D might be preferable as this choice reduces the total number of feature instances used in the application.

The inputs and outputs of the feature placement problem are shown in Figure 3.5. Input for the placement problem comes from three sources: the servers, on which the application are executed, the feature model, that defines the structure of applications that are to be placed, and the applications themselves. More specifically:

- The *servers* contain resources, such as CPU, memory, disk space and bandwidth. Each of these are limited, and it is impossible to allocate more of these resources to feature instances than available. Using a server also incurs

a server use cost. This is the operational cost of using the server, and can represent an energy cost, to determine an energy-efficient placement, or an instance cost in a hybrid or public cloud environment.

- The *feature model* describes the different features, and the relations between them. These must be expressed, ensuring they can be taken into account during the placement process. For every feature, instance resource requirements are needed. The different feature instances may also impact each others resource requirements. For example, the addition of a security feature can increase the load on other services, that then have to use encryption in their communications. To get a realistic view of the actual resource need of features, the impact features have on each others resource demands is added as an input. Finally, it is possible that a single instance of a feature, with a fixed amount of e.g. memory, can only use a limited amount of resources, e.g. CPU. If this is the case, these limitations are also added as an input of the feature placement problem. Then, multiple instances of a feature can be instantiated on a single server to handle higher resource demands.
- Each *application* is an instantiation of the feature model, with a specific selection of features. Applications add three parameters to the feature placement: (1) the demand, that varies depending on the load on the application, (2) a feature selection, that indicates the selected and excluded features, and (3) the cost of failing to place the application. In some cases an additional cost for the failure to provision specific features can be added, for example a feature providing a minimal service. If so, this feature failure cost is also added as an input. The cost of failing applications and features can either be an actual economical cost, defined in a SLA, or it can be an estimated cost such as the potential cost of losing customers due to a bad service.

Using these inputs, the feature placement will generate two outputs:

- For every application, a *feature selection* will be returned. This contains all the features that were selected in the feature selection input variable, but any remaining open variation points are filled in.
- A *placement*, that contains for every server the number of instances of a feature that are executed on it, and the amount of resources allocated to them. Each instance of a feature uses part of the available resources on the server on which it is executed (represented using pie charts in Figure 3.5). When no services are allocated on a server, it can be turned off, reducing the operational cost of the placement.

When a resource conflict occurs, and more resources are needed by applications than available, the algorithm handles this conflict by choosing the best configuration

based on its resulting cost. In this case, some of the application features will not be placed. The cost used within the optimization is composed of the cost incurred by a failure to place applications and the cost of using a server, with the cost of failure of placing applications typically significantly larger than the cost of using servers. An optimal solution to the feature placement problem is a placement that minimizes the total cost.

A feature placement algorithm will be used as one of the central components of a cloud management system. The system architecture of this management system contains three components that are responsible for determining feature placement inputs: 1) a staging environment where new configurations are tested, and where the impact of features on other features can be measured in a controlled environment; 2) a monitoring system, that can be used to dynamically improve estimated demand and impacts during execution; and 3) an admission controller, limiting the number of applications admitted into the system.

3.4 Formal problem description

The variables used in the model are listed in Table 3.2. We begin by discussing the optimization objective. This is followed by a description of the input variables. Three variable types can be distinguished: input variables, decision variables and auxiliary variables. Finally we will discuss the constraints used within the model.

3.4.1 Optimization objective

The objective of a placement algorithm is to minimize two costs: the cost of failure to place an application or feature, which we refer to as the cost of non-realized demand, and the cost of using servers, referred to as the server use cost. When multiple applications contend for resources, the configuration with the lowest cost according to this objective function will be chosen.

More formally, the goal of the model is to minimize the cost, C , of the placement. This cost is determined by two factors: the cost of non-realized demand, C_D , which is incurred due to failure to place applications, and a server use cost, C_U which is incurred when servers are used. We express this cost using Equation (3.1).

$$C = C_D + C_U \quad (3.1)$$

C_D is defined in Equation (3.2).

$$C_D = \sum_{a \in A} \left(p_a \times C^V(a) + \sum_{f \in \text{sel}(a)} p_{f,a} \times C^V(f, a) \right) \quad (3.2)$$

Equation (3.2) uses binary variables to indicate when the provisioning of features or applications fail, and multiplies these binary variables with the cost that this

Input Variables		
Symbol	Type	Description
Γ		The set of considered resource types (eg. memory, CPU, bandwidth).
Γ_s	$\wp(\Gamma)$	Resource types for which the demand is strict: they must be allocated for each feature instance for the instance to be usable.
$\Gamma_{\bar{s}}$	$\wp(\Gamma)$	The resource types for which the demand is non-strict: the goal of the optimization process is to allocate as much of this demand as possible, but a configuration in which not all of these resources have been allocated is still valid.
S		The set of servers.
Ra_s^γ	$[0, +\infty)$	The available resources on a server s for a resource type $\gamma \in \Gamma$.
\mathcal{F}		The feature model used by the applications.
F		The set of features contained in \mathcal{F} .
\mathcal{R}		The set of relations contained in \mathcal{F} , using relations as described in Table 3.1.
A		The set of applications.
$\text{sel}(a)$	$\wp(F)$	The features that must be included for application a .
$\text{excl}(a)$	$\wp(F)$	The features that must not be included for application a .
$FI_{f_1}^\gamma(f_2)$	$[0, +\infty)$	The impact on the resource requirement for feature f_2 if feature f_1 is included in the selected features of an application, for a resource type $\gamma \in \Gamma_{\bar{s}}$.
D_a	$(0, +\infty)$	The demand for an application a .
IR_f^γ	$[0, +\infty)$	The resource requirement of a single instance of a feature f for a resource type $\gamma \in \Gamma_s$.
L_f^γ	$(0, +\infty)$	The instance limitations indicate the maximum amount of non-strict resource type γ that can be allocated to a single instance of a feature f .
$CV(f, a)$	$[0, +\infty)$	The cost of failing to place a feature f for an application a .
$CV(a)$	$[0, +\infty)$	The cost of failing to place an application a .
$CU(s)$	$[0, +\infty)$	The cost of using a server s .
Decision Variables		
Symbol	Type	Description
$M_{s,f,a}^\gamma$	$[0, +\infty)$	The amount of a resource $\gamma \in \Gamma_{\bar{s}}$ to be allocated for a given server s , feature f and application a .
$\Phi_{f,a}$	$\{0, 1\}$	A binary variable indicating whether application a includes feature f .
$IC_{s,f}$	\mathbb{N}	The instance count is an integer variable, indicating how many instances of a feature f are instantiated on a server s .
Auxiliary Variables		
Symbol	Type	Description
$AI_{f,a}^\gamma$	$[0, +\infty)$	The application impact, containing the actual resource impact per feature f of a specific application a , for a resource $\gamma \in \Gamma_{\bar{s}}$.
p_a	$\{0, 1\}$	A binary variable that has value 1 if an application a is not correctly placed, that is when any of its features are not placed.
$p_{f,a}$	$\{0, 1\}$	A variable that has value 1 when the resource demand of a single feature f of an application a is not placed.
$p_{f,a}^\gamma$	$\{0, 1\}$	A variable indicating whether the resource demand of a single feature f of an application a is not placed, for a specific resource $\gamma \in \Gamma_{\bar{s}}$.
U_s	$\{0, 1\}$	A binary variable indicating whether a server s is used.

Table 3.2: The different symbols used in Section 3.4.

failure causes. The variable $p_{f,a}$ takes on value 1 if the feature f of an application a is not provisioned sufficient resources, and 0 otherwise. Similarly, a binary variable p_a is used to express the failure of *any* feature of an application a . To determine the total costs, these binary variables are then combined with the cost of failing to provision individual features $C^V(f, a)$, and the cost of failing to provision an application $C^V(a)$.

Note that within our approach, any feature can fail, including those that are considered mandatory; feature failure is handled by assigning a cost to it. This is done to ensure the feasibility of results: by enforcing the inclusion of selected features using constraints, some inputs could lead to an infeasible result to which no solution exists. It is better for a single application or feature to fail, than for there not to be a feasible solution at all. More importantly, if no feasible solution can be determined, it is important that the application or feature that fails incurs the lowest possible cost.

The cost of using a server is expressed in Equation (3.3). The equation makes use of a server usage cost $C^U(s)$, denoting the cost of using a server, and binary variables U_s indicating whether a server is used.

$$C_U = \sum_{s \in S} U_s \times C^U(s) \quad (3.3)$$

3.4.2 Input variables

In literature, application placement techniques are generally designed to place application instances on servers, ensuring a global CPU demand is met. Each of these application instances requires a fixed amount of memory for it to work. Some works, however, make use of different resource types, e.g. bandwidth [26], or sometimes the resource types are abstracted [27]. To ensure maximum applicability of the formal model, we will define it making use of two generalized resource types, and we will allow multiple resources of these types to occur:

- The first resource type is associated with individual instances, and these resources are needed to create a valid instance. Every instance needs exactly the right amount of these resources to function correctly. We refer to these resources as *strict resources*, as a given amount of them is needed to create a valid feature instance. This in turn implies these requirements are enforced as *constraints*. The memory resource, in a VM placement scenario, has this behavior, as an instance needs a fixed amount of memory to run. Similarly, disk space is also a resource of this type, as each VM requires disk space for its image. In some cases, a fixed amount of bandwidth is required per instance, making it a resource of this type.
- The second resource type behaves differently. For these resource, there is a *global demand*, that must be fulfilled, and fulfilling as much of this demand

as possible is the goal of the optimization process. To succeed, instances must be created that handle part of the resource demand. We refer to these resources as *non-strict resources*. The traditional example of this resource requirement is the CPU demand, that is often used in application placement. In some cases other resource types can occur, such as bandwidth, if fulfilling a given bandwidth demand is the optimization goal.

We have previously made a similar distinction between resources in [28], and a similar approach was used in [27]. Within the model, we define Γ as the collection of all resource types, and we use Γ_s and $\Gamma_{\bar{s}}$ to denote strict and non-strict resource types respectively. Note that in practice less strict resources than needed could be allocated to an instance: a virtual machine can for example function with less memory at the cost of performance degradation. Characterizing this performance degradation is however service-specific, and as every service is used by multiple applications this performance degradation impacts the quality of multiple applications. By using conservative fixed resource requirement estimates, these issues can be avoided. For these reasons, strict resources are defined as a fixed value requirement within this chapter.

Sometimes a pure separation between the two resource types is difficult to achieve, as an increase in for example CPU use can sometimes cause increasing memory utilization. To linearize this problem, we will introduce instance limitations further in this section. These ensure a limit is added to the amount of work a single instance can process, ensuring that memory-intensive applications can also be modeled using this formulation.

Each problem also has a set of servers S with an amount of available resources. For a server $s \in S$ the available resources are given by Ra_s^γ , for the different resource types $\gamma \in \Gamma$. The goal of the optimization is to allocate the required non-strict resources for applications at a minimal cost, while ensuring the created instances have the required strict resources they require to execute.

The problem statement also contains a set of applications A that must be placed. Each of the applications is a specific instantiation of a global feature model \mathcal{F} . This feature model contains a set of features F and a collection of relations \mathcal{R} , formally describing the feature model tree. The possible relations are described in Section 3.3 and Table 1. This approach still allows the placement of entirely distinct application types with separate feature models \mathcal{F}_i by creating a set containing the roots of every feature model, R , and linking these different feature models in a global feature model by the addition of a new root feature r and a relation **Alternative**(r, R). This ensures that each application executed on the cloud is an instance of exactly one of the separate feature models, and that an arbitrary amount of different application types can be placed using the model. An example of this will be shown in Section 3.7.3.2.

Every application $a \in A$ contains a set $\text{sel}(a) \in F$ with features that must be selected in the application and a set $\text{excl}(a) \in F$, containing features that must be excluded. The configuration of both is assumed to be valid according to \mathcal{F} . Features contained in neither set are considered open variation points as described in Section 3.3.

It is possible for features to impact the resource needs of other features. For instance, adding the *encryption* feature to the application in Figure 3.2 can increase the CPU load on the *interfacing engine*, and applications hosted by the application provider will require more CPU resources than applications partially hosted at the client site. We assume that applications with similar feature selections will have similar load characteristics, as this is the case for the three application use cases, and represent this using a feature impact matrix FI . $FI_{f_1}^\gamma(f_2)$ represents the impact of feature f_1 on feature f_2 for a non-strict resource type γ . The resource requirement of a feature f can be expressed using the feature's impact on itself $FI_f^\gamma(f)$. By including a feature f , it's own feature impact $FI_f^\gamma(f)$ is added, representing the resource requirement of the feature itself, and it's impact is added to all other features f' for which $FI_{f'}^\gamma(f) \neq 0$. When two applications make use of the same feature, they will both require resources allocated to this feature, and thus both resource requirements will be counted to determine the total resource demand for this feature. As the demand for an application varies in time, we also add a D_a variable denoting the user demand for an application a . This variable impacts the resource need for the entire application. If load characteristics can vary for individual applications, the approach could be extended ensuring a FI matrix is defined per-application, but in such a case detailed measurements would be needed to determine this matrix for every individual application. Every instance of a feature f also requires a specific amount of strict resources IR_f^γ .

In many situations, it is unrealistic to assume that a single instance with limited strict resources allocated to it, would be able to use an unlimited amount of non-strict resources. Because of this, we introduce resource limitations: A single instance of a feature f cannot use more than L_f^γ of non-strict resource $\gamma \in \Gamma_{\bar{s}}$. E.g. an application component that is memory intensive will have a low limit, ensuring only a limited amount of CPU can be used by it. These limitations make the model more applicable to real-life applications, ensuring the ratio between allocated non-strict and strict resource types remains realistic.

The optimization process is used to minimize the cost of failed placement and the server use cost. Two variables are needed to represent the cost of failing to place specific features and applications:

- The cost of failing to reserve the capacity for a specific feature f of an application a is given by $C^V(f, a)$. This can be used if failure of specific features needs to be taken into account.

- The cost of failing to reserve the capacity for *any* feature of an application is given by $C^V(a)$.

Finally, for every server s , the cost of using the server $C^U(s)$ can be determined. This cost can be the energy cost of using the server, or the cost of using a server from a remote IaaS provider. This parameter allows the system to take energy-efficiency of the cloud into account, and could also be used to differentiate between the cost of using the local datacenter and a remote IaaS cloud in a hybrid cloud scenario.

3.4.3 Decision variables

The output of the formulation is a placement, indicating which applications are executed where, and the amount of resources allocated for each of these applications. The resource types behave differently, leading to two separate expressions. For strict resource types, we determine how often an application is instantiated on a server, a value that can be used to determine the required strict resource requirement. As the amount of non-strict resources that can be allocated for a given feature can be limited, it is possible for there to be multiple instances of a feature on a single server. For non-strict resources we make use of a matrix representing the amount of resources allocated on a server. This yields two variables:

- The variable $IC_{s,f}$ determines the number of instances of feature f on server s . This integer variable can be used to determine the total strict resource usage of features on servers, by multiplying it with the, fixed, per-instance resource requirements IR .
- For non-strict resources we use an allocation matrix M . For a server s , feature f , and application a , $M_{s,f,a}^\gamma$ contains the amount of non-strict resources of a type γ that need to be allocated.

Another output is the feature selection matrix Φ , indicating which applications are selected and excluded for a given application. For application a and feature f , $\Phi_{f,a} = 1$ if the application contains the feature and $\Phi_{f,a} = 0$ if it does not. At the start of the algorithm this matrix can be partially filled in by using $\text{sel}(a)$ and $\text{excl}(a)$, the remaining features are assigned values during the optimization process.

3.4.4 Auxiliary variables

Up until now, we have not yet defined a variable that determines the actual resource requirement of a feature of an application. For this, we define the application impact matrix $AI_{f,a}^\gamma$ which contains, for every feature f of application a , the actual resource requirement for a given non-strict resource γ . This matrix can be constructed using the selected features and the feature impact matrix.

A set of binary variables is needed to express whether an application is correctly provisioned. We do this by introducing variables denoting application failure, feature failure, and the failure to provision specific resource demands for an application:

- For every application a there is a variable p_a , indicating whether the provisioning of an application has failed. If $p_a = 1$, a feature of a exists that has not been allocated sufficient non-strict resources.
- For every application a and feature f there is a variable $p_{f,a}$. This variable indicates whether a specific feature of the application is insufficiently provisioned.
- For every application a , feature f and non-strict resource γ , there is a variable $p_{f,a}^\gamma$, which has value 1 when too few resources of type γ were allocated for a feature of an application.

Finally, a collection of variables is needed to determine whether a server is active. For every server s there is a binary variable U_s , indicating whether the server is used. If $U_s = 0$ the server is not used and can be turned off.

3.4.5 Constraint details

In the following sections we will discuss the different constraints included in the model.

3.4.5.1 Feature-based constraints

The feature selection matrix Φ is used to indicate whether a feature f is present in an application a , $\Phi_{f,a}$ being 1 if f is included in a , and 0 if it is not. For an application a we add the constraints $\Phi_{f,a} = 1$ if $f \in \text{sel}(a)$ and $\Phi_{f,a} = 0$ if $f \in \text{excl}(a)$. If the feature does not occur in either set, the value of $\Phi_{f,a}$ remains undecided, creating open variation points, which will be filled in during the optimization process.

The relations between features \mathcal{R} must also be converted into constraints. Elements of \mathcal{R} define relations between individual features. As the constraints of the feature model affect all applications, they must be applied to all application features in the feature selection matrix. Because of this, we define $f_i = \Phi_{i,*}$ a row of the feature selection matrix. We describe the conversion for the relation types to constraints in Table 3.3. This conversion is required as the logical constraints defined by the relations must be converted into linear expressions for them to be formally used within the model. When we, for example, apply this conversion to the **Alternative**($f_A, \{f_B, f_C\}$) relation, this yields the constraint $\Phi_{f_A,a} = \Phi_{f_B,a} + \Phi_{f_C,a}$, for every $a \in A$.

Relation	Conversion
Mandatory (f_A, f_B)	$f_A = f_B$
Optional (f_A, f_B)	$f_A \geq f_B$
Alternative ($f_A, \{f_B, f_C\}$)	$f_A = f_B + f_C$
Or ($f_A, \{f_B, f_C\}$)	$f_A \geq f_B$
	$f_A \geq f_C$
	$f_A \leq f_B + f_C$

Table 3.3: Conversion of relations in the feature model \mathcal{F} to constraints.

3.4.5.2 Application resource requirement constraints

Each feature f can have resource requirements, but it can also impact resource requirements of other features. If feature f is selected, its impact matrix, FI_f^γ will be added to the total resource requirement for the application. A feature f_i can only affect a feature f_j if f_i requires f_j according to the feature model. Otherwise the feature impact matrix would be able to add feature constraints not included in the feature model, which could in turn lead to inconsistencies.

Using the selected features Φ and the feature impact matrices FI_f^γ , an application impact matrix $AI_{f,a}^\gamma$ can be constructed. This application impact matrix, expressed in Equation (3.4), displays the resource requirements for individual features f , of an application a , for a given non-strict resource γ , and additionally takes into account the global application demand variable D_a for the application.

$$AI_{f,a}^\gamma = D_a \times \sum_{f' \in F} \Phi_{f',a} \times FI_{f'}^\gamma(f) \quad (3.4)$$

3.4.5.3 Resource constraints

Resource constraints are expressed for every server s , but this is done differently for strict and non-strict resources. For non-strict resources, the used resources are expressed using the allocation matrix M^γ , of which the requirement is aggregated over all features and applications. This is done, for every $\gamma \in \Gamma_s$, in Equation (3.5). Strict resource limitations follow from the instance count IC for the service, indicating the number of times a service is allocated, and the required amount of strict resources per-instance, as shown in Equation (3.6), which is added for every $\gamma \in \Gamma_s$.

$$\sum_{f \in F} \sum_{a \in A} M_{s,f,a}^\gamma \leq Ra_s^\gamma \quad (3.5)$$

$$\sum_{f \in F} IR_f^\gamma \times IC_{s,f} \leq Ra_s^\gamma \quad (3.6)$$

As discussed earlier in Section 3.4.2, we assume that single feature instances are only capable of using limited amounts of resources. This is expressed using Equation (3.7). The equation expresses that the total resource allocation, for a non-strict resource type γ , of a given feature f , on server s , must not exceed the amount of resources the instances can handle.

$$\sum_{a \in A} M_{s,f,a}^\gamma \leq L_f^\gamma \times IC_{s,f} \quad (3.7)$$

3.4.5.4 Application provisioning constraints

Additional constraints are needed to ensure the variables $p_{f,a}^\gamma$, $p_{f,a}$ and p_f , introduced in Section 3.4.4, correctly express whether the application and features are insufficiently provisioned. Logically, we can express the $p_{f,a}^\gamma$ this using Equation (3.8):

$$p_{f,a}^\gamma \equiv \sum_{s \in S} M_{f,s,a}^\gamma < AI_{f,a}^\gamma \quad (3.8)$$

This statement can be turned into constraints using the transformation of Equation (3.9) to Equation (3.10), with $x \in \{0, 1\}$, and M a number larger than any possible value of **expr**. If $x = 0$, it follows from Equation (3.10) that **expr** ≤ 0 , while $x = 1$ yields the constraint **expr** $\leq M$, which is always true. Consequently, this transformation holds only in optimizations where the objective function value improves when $x = 0$, which is the case here as a placement in which no applications fail ($p_{f,a}^\gamma = 0$) is preferred by the optimization objective function.

$$x \equiv \mathbf{expr} > 0 \quad (3.9)$$

$$\mathbf{expr} \leq x \times M \quad (3.10)$$

Applying the transformation to Equation (3.8), $\mathbf{expr} = AI_{f,a}^\gamma - \sum_{s \in S} M_{f,s,a}^\gamma$. To determine a minimal value for M , we must find a maximum value for the first term, and a minimal value for the second term of **expr**. For the first term, the definition of AI , Equation (3.4), can be used with an application that contains all features. For the second term, an empty allocation matrix can be used. This leads to $M = 1 + \sum_{f' \in F} FI_{f'}^\gamma(f)$, ensuring $M > \mathbf{expr}$ for all possible values of **expr**.

Once the different $p_{f,a}^\gamma$ variables are determined, we can use these to determine the value of $p_{f,a}$ by expressing, for all of the non-strict resource types γ , that $p_{f,a} \geq p_{f,a}^\gamma$, as the failure for a single resource type ($p_{f,a}^\gamma = 1$) implies the failure of the entire feature ($f_{f,a} = 1$). We also add the constraint $p_a \geq p_{f,a}$ for every feature f and application a , using a similar logic.

3.4.5.5 Cascading failure of features

Child features are dependent on their parent features, and require the parent feature to be selected for them to be used. This implies that, should the parent feature

fail, the child feature will fail as well. This is easy to add to the model by, for every parent feature f and child feature c related in the feature model, and every application a , adding the following constraint:

$$p_{f,a} \leq p_{c,a} \quad (3.11)$$

Equation (3.11) expresses that if a parent feature fails for an application, the child features must fail as well.

3.4.5.6 Server usage constraints

The variable U_s expresses whether a server s is used. Logically, a server is used if any resource $r \in \Gamma$ is allocated on the server. We express this using Equation (3.12).

$$U_s \equiv TSU_s \neq 0 \quad (3.12)$$

$$TSU_s = \sum_{\gamma \in \Gamma_s} \sum_{f \in F} \sum_{a \in A} M_{f,s,a}^\gamma + \sum_{\gamma \in \Gamma_s} \sum_{f \in F} IR_f^\gamma \times IC_{s,f} \quad (3.13)$$

Equation (3.13) describes the total server use (TSU) for a server s , and calculates the sum of all resources used on the server. This adds values for all non-strict resource types, by summing them over the allocation matrix M , and for all the strict resource types by multiplying the instance counts IC with the instance requirements IR . This summation adds elements with different unit types, so the actual resulting value is of little use, but as soon as a single resource is used on the server, TSU_s will be non-zero, ensuring $U_s = 1$.

The transformation from Equation (3.14) to Equation (3.15) transforms these logical statements into constraints, and only holds if **expr** is non-negative, which is the case here as negative resource requirements are impossible. If $x = 1$, then **expr** $\leq \mathbf{M}$, which is always true. If $x = 0$, it follows that **expr** ≤ 0 , which taking into account that **expr** ≥ 0 implies that **expr** $= 0$. Again, this transformation holds only if the placement quality benefits when $x = 0$, as otherwise this option will not necessarily be taken, but this is the case as switching off servers ($U_s = 0$) lowers the cost of execution.

$$x \equiv \mathbf{expr} \neq 0 \quad (3.14)$$

$$\mathbf{expr} \leq x \times \mathbf{M} \quad (3.15)$$

Like in the previous section, we can determine a minimal value for \mathbf{M} , again by finding a maximal value for **expr**. Here this can be done by observing that **expr** equals the sum of all resources used on a server, which can never be larger than the sum of all available resources. Thus, we choose $\mathbf{M} = 1 + \sum_{\gamma \in \Gamma} Ra_s^\gamma$.

Function	Description
place	This recursive function forms the main part of the feature placement algorithm, and is responsible for placing a collection of features on a collection of servers.
placeFeature	This function is responsible for placing a single feature on a collection of servers.
featureConversion	A function used to fill in open variation points in feature models.
groupStrategy	A function determining whether an application features are placed at once or in multiple steps.
featureOrder	Determines the order in which features or applications are placed.
serverOrder	The order in which servers are considered during placement.

Table 3.4: The different functions used in Section 3.5.2.

3.5 Solution techniques

We consider an optimal algorithm, based on an Integer Linear Programming (ILP) solver, and several heuristic algorithms to solve the feature placement problem.

3.5.1 Integer Linear Programming (ILP)

The formulation, discussed in the previous section, can be used to define an ILP. This program can be solved using a commercial ILP solver, and yields the optimal problem solution using Simplex and Branch and Bound algorithms. As the model contains integer values, the ILP algorithm can not be run in polynomial-bound execution time. Therefore, we will define heuristic algorithms that approximate the optimal solution generated by the ILP solver.

3.5.2 Heuristic algorithms

We first define a single meta-heuristic, consisting of two recursive functions: an inner function **placeFeature**, placing individual features and a **place** function that does the actual feature placement. The meta-heuristic as we define it makes use of four functions that are left open. We then present different approaches for filling in these functions. The combination of the algorithm with different function implementations can be used to define different algorithms with varying performance and properties. The different functions used in this section are shown in Table 3.4.

Data: problem P
Data: Instance Count for a feature on a server $IC_{s,f}$
Data: current placement matrix $M_{s,f,a}^\gamma$
Data: a feature f of an application a to place
Data: list of servers with remaining resources $Servers$
 sort $Servers$ using **serverOrder**;
 $s \leftarrow \text{findServer}(f, Servers)$;
if no s found then
 return \emptyset ;
else
 if no remaining capacity for f on s then
 $IC_{s,f} = IC_{s,f} + 1$;
 end
 Update $M_{s,f,a}$ for all resource types;
 Adjust remaining resources on s ;
 if all non-strict resource demand placed then
 return $(IC, M, Servers)$;
 else
 Update f , decreasing its resource demand;
 return $\text{placeFeature}(P, IC, M, f, Servers)$;
 end
end

Algorithm 1: The **placeFeature** function used by the algorithm.

Algorithm 1, describes the **placeFeature** function, responsible for the placement of a single feature. As input, this function requires different parameters: (1) the problem configuration P , containing all the input variables of the formal problem formulation, (2) the instance count matrix $IC_{f,s}$, which contains the number of instances of a features each server has, (3) the placement matrix $M_{s,f,a}^\gamma$, which specifies the amount of resources allocated to a feature and application on a server, (4) the feature f that must be placed, and (5) a list $Servers$ containing all the servers in the system and their remaining resource capacities.

The algorithm sorts the list, using a given **serverOrder**, and uses a **findServer** operation to find the first server s in the sorted list on which either a feature instance exists with remaining free space, or on which enough resources remain to create a new instance of the feature. In the latter case, a new instance is created. The **serverOrder**, which determines the order in which servers are considered, is essential for the performance of the algorithm, and will be elaborated on later on in the chapter. Subsequently, the maximum amount of resources possible, taking into account instance resource limitations, are allocated for the feature that is to be placed, by adding them to $M_{s,f,a}^\gamma$. The server information of s is also updated, to reflect the decrease in available resources on the server. If the entire feature f is

placed, the updated allocation IC and M is returned, along with the updated server list $Servers$ are returned. If the feature is not fully placed yet, the **placeFeature** function is repeated recursively, and is given as an argument the residual demand of feature f . The **placeFeature** function will always either return a placement where the feature is placed in its entirety, or not placed at all.

The main body of the heuristic is listed in Algorithm 2, which displays the **place** function. The function is responsible for placing a list of applications or features. It requires five parameters: (1) the problem model description P , (2) the instance count IC , (3) the current placement matrix M , (4) a list $Servers$, containing all the servers, (5) a list $AppFeatures$, of which every entry is either an application or a feature, and (6) a collection $Failed$ containing the applications for which the placement of the application as a whole has failed. The first four parameters are also used for the **placeFeature** function. The fifth parameter determines the order in which features and applications are added, and makes it possible to place features as either applications, or as individual instances. The sixth parameter maintains a list of applications and features that could not be placed successfully.

The formulation of the **place** function makes use of two additional functions:

1. The **dependingFeatures**(f) function returns the set of all features that depend on the feature f . All the features present in the subtree with root f of the feature model tree, except the feature f itself, are included in this set.
2. The **dependentFeatures**(f) function is the opposite of the previous relation, and returns the collection of all features upon which the feature f is dependent. This set can be constructed by, within the feature model tree, selecting the parent feature of f , and subsequently recursively adding all of the parent features of the features present in the set.

The **place** function starts by choosing the first element of the $AppFeatures$ list. If this element is a feature, it first checks whether the feature should be added. If any feature upon which the feature depends has already failed to be placed for this application, the selected feature is not placed, as it would automatically fail because of the cascading of failure constraint described in Equation (3.11). If the application has already failed to be placed, and there is no additional cost for the failure of this feature or any of the child features, the feature is not placed either. This rule is added, as placing these features would increase both the load on the system and the cost of used servers, without decreasing the cost of failed placement. If neither condition is met, the algorithm continues by using the **placeFeature** function to place the feature on the infrastructure. The **place** function is then repeated with the remaining elements of the $AppFeatures$ list and, if the feature was correctly placed, the server configuration returned by the **placeFeature** function. Otherwise the initial server configuration is reused.

Data: problem P
Data: Instance Count for a feature on a server $IC_{s,f}$
Data: current placement matrix $M_{s,f,a}^\gamma$
Data: list of servers with remaining resources $Servers$
Data: list of applications and features to place $AppFeature$
Data: collection of failed application placements $Failed$
if $AppFeature$ is empty **then**
 | **return** $(IC_{s,f}, M_{s,f,a}^\gamma, Servers)$
else
 $fa \leftarrow$ take first element of $AppFeature$;
 $AppFeatures' \leftarrow$ tail of $AppFeature$ list;
 if fa is a feature f of an application a **then**
 $failedDependent \leftarrow Failed \cap \text{dependentFeatures}(f)$;
 $failCost \leftarrow C^V(f, a) + \sum_{f' \in \text{dependingFeatures}(f)} C^V(f', a)$;
 if $failedDependent \neq \emptyset$ **then**
 | Do not place feature;
 else if $a \in Failed \wedge failCost = 0$ **then**
 | Do not place feature;
 else
 $(IC', M', Servers') \leftarrow$
 | $\text{placeFeature}(P, IC, M, Servers, f)$;
 end
 if feature fa placed **then**
 | $\text{place}(P, IC', M', AppFeature', Servers', Failed)$;
 else
 | $\text{place}(P, IC, M, AppFeature', Servers, Failed \cup fa)$;
 end
 else if fa is an application **then**
 $features \leftarrow$ features of fa ;
 sort $features$ using **featureOrder**;
 $(IC', M', Servers') \leftarrow \text{place}(P, IC, M, features, Servers)$;
 if fa is correctly provisioned **then**
 | $\text{place}(P, IC', M', features, Servers', Failed)$;
 else
 $AppFeatures' \leftarrow AppFeatures' + features$;
 sort $AppFeatures'$ using **featureOrder**;
 $\text{place}(P, IC, M, AppFeatures', Servers, Failed \cup fa)$;
 end
 end
end

Algorithm 2: The place function.

If the head element of the *AppFeatures* list is an application, the list of all features in the application will be determined, and this list will be placed by recursively calling the **placeFeature** function. If this succeeds, and all the features of the application can be placed, the algorithm will continue by processing the tail of the *AppFeatures* list. If this fails, the changes are undone, and the individual features of the applications are added to the *AppFeatures* list, which will be sorted again, and then placed using a recursive call to the **placeFeature** function. This ensures that, if an application can not be placed in its entirety, the algorithm will still make an effort to place individual application features. As the cost of failing to place the application is incurred by this, only features that further add to the cost of failure will still be considered for placement.

Algorithm 3 shows how the initial parameters are generated, and contains the complete feature placement algorithm.

Data: problem P
 $features \leftarrow \text{featureConversion}(P);$
 $list \leftarrow \text{groupStrategy}(P, features);$
 $AppFeature \leftarrow \text{sort } list \text{ using } \text{featureOrder};$
 $Servers \leftarrow \text{sort servers in } P \text{ using } \text{serverOrder};$
 $IC_{s,f} \leftarrow 0;$
 $M_{s,f,a}^\gamma \leftarrow 0;$
 $Failed \leftarrow \emptyset;$
 execute **place**($P, IC, M, AppFeature, Servers, Failed$);
Algorithm 3: The feature placement algorithm.

The algorithm contains four components that we have not yet elaborated on. At the start of the algorithm, the open variation points of the feature model are filled in using a **featureConversion** function. This function ensures that for every application, all features are either selected or excluded, eliminating open variation points. The **groupStrategy** is used to determine whether all the application features should be considered as a whole, or whether they should be placed independently. The result of this function is a list containing a mix of features and applications: the *AppFeature* list. The **featureOrder** is used to sort the *AppFeature* list, and can compare features and applications to determine the order in which they are placed. Finally, the order in which servers are considered is determined by the **serverOrder** function. The effectiveness of the meta-heuristic is largely determined by the **featureConversion**, **groupStrategy**, **featureOrder**, and **serverOrder** functions. We will now present different implementations for these functions.

3.5.2.1 Feature ordering

The order in which features are considered significantly impacts the quality of the final result, as it determines which features are placed first, and thus assigns a

priority to the features. We make use of an application-based ordering, where applications and features with a higher cost of failure are placed first. For applications, we define the cost of failure as the sum of feature failure costs, and the application failure costs. For features, we define this cost as the sum of the cost of failure of the feature, the application, and the cost of failure of all features dependent on the feature. When according to this ordering no preference is achieved, we consider the number of instances required to place the feature or application. The instance requiring the smallest number of instances is placed first.

3.5.2.2 Grouping strategies

As explained above, the list *AppFeature* can contain either entire applications, individual application features or a combination of both. The **groupStrategy** function determines for each application present in the problem definition whether it must be considered as a whole, or as a group of features. We consider two versions:

- Feature grouping, where every application is split up into features, and the features are placed independently. This corresponds to the approach we previously described in [7].
- Application-based grouping, where applications are always grouped, their features are placed at the same time. Should the placement of an application fail, the algorithm will still try to place individual features, as described in the Algorithm 2.

It is important to note that in both cases, the algorithm will place multi-tenant feature instances, and allocate part of their capacity to the placed applications. Using application-based grouping, the algorithm will however start by trying to place all of the feature instances of a given application at once.

3.5.2.3 Server ordering

We consider two different server orderings:

- Instance Based (IB) ordering, which orders servers according to the best fit for the feature f that is to be placed. This ordering prefers servers that have instances of the feature placed on it, that are not fully utilized by the current allocation. If multiple servers comply, the server with the best fit will be selected. If, using this approach, two servers score the same, the server with the lowest utilization cost is used.
- Cost Based (CB) ordering, where servers are ordered according to their utilization cost.

Note that the IB ordering of nodes changes for every invocation of the place method, whereas the CB ordering does not change. This ensures the sort in Algorithm 1 does not have to be executed for the CB ordering. Both of the approaches take the cost of using servers, C^U , into account, but only in the IB case is it the primary selection criterion.

3.5.2.4 Feature model conversion

The **featureConversion** function is used to fill in open variation points. This function determines the features that must be included in the placed applications. We make use of an approach in which the cheapest feature combination in terms of resource requirements is determined in two steps. First, ten cheap combinations of feature models are determined for every application. As the number of combinations increases exponentially, at each point in time the list of possibilities is shortened. We have shown before that shortening to ten elements is sufficient for improving the placement [7]. This can be determined as soon as an application is added, rather than when application placement is executed. Within the evaluation section, we will refer to this as the preparation step of the algorithm.

Secondly, when the list of all applications is known, the best total configuration is determined. This is done by incrementally iterating all applications, and creating a partial list containing a the best configuration for the subset of applications that has already been considered. In every step, an application is added, and each of its ten feature combinations is combined with the list of best applications from before. From the resulting collection, the ten best elements are retained and passed on to the next iteration.

3.5.2.5 Heuristic algorithms

The described functions can be combined with the meta-heuristic to create different algorithms. For this chapter, we use the two grouping strategies, *feature* and *application* based, and the two server orderings, *IB* and *CB*. This creates four algorithms: **IB_application**, **IB_feature**, **CB_application** and **CB_feature**.

3.6 Evaluation setup details

We implemented the ILP problem and the heuristics using Scala. The ILP solver uses CPLEX [35] as its back-end. Within the evaluations, we will make use of two types of problem models: 1) problem models based on the three real-life applications studied in the CUSTOMSS project, and 2) problems created using a generator capable of creating a wide range of random problems.

3.6.1 CUSTOMSS problem model

The full model, used by the CUSTOMSS project, is shown in Figure 3.6, and contains the features and relations as they are currently defined in the project. The feature names have been replaced by numbers. Each feature entry also contains an estimated CPU requirement and a CPU use limitation (in the form $CPU = requirement/limit$), and an instance memory requirement ($Memory = requirement$). The relations between features are expressed in the format as described in Section 3.3. As discussed earlier, features can impact each other. This is illustrated by the arrows between nodes, the number on the arcs represents the impact on CPU requirement other nodes. For example, the addition of Feature 7 increases the CPU demand for Feature 1 by 100.

The presented model groups the models for the three real-life applications, with application roots Feature 1, Feature 16 and Feature 28 into a single model by adding a new root node, modeling a cloud that executes these distinct applications. As the nodes are grouped using an *Alternative* relation, every application will be an instance of exactly one of the CUSTOMSS applications. This approach for running multiple distinct applications was discussed previously in Section 3.4.2. Note that some of the features do not have any CPU or Memory requirement. These features are either used for grouping other features, improving the structure of the complete feature model, or as they do not create new feature instances but significantly impact the demand for other features. When these features are included, they are automatically satisfied, provided that their parent nodes are correctly provisioned.

Applications feature selections are randomly generated by creating random valid selections, where all features are either selected or excluded. Open variation points are then randomly added. The application failure cost is set to 10. Some features are selected in the feature model and are considered as being critical: if they are selected, they must be correctly provisioned, or an additional cost of 5 will apply. The features in the model that can be considered as critical are Features 13, 21, 22, 26 and 40. The energy cost is chosen as 1. This ensures applications will always be placed if possible, the desired behavior, and that, if an application does fail, only its critical features are placed.

In practice, a realistic cost could be determined by utilizing the actual economical cost of failure of applications. This cost would however vary throughout time, based on previous placement performance. Practically, it is better to assign relative costs that are maintained by the management system. In general, the cost of failure of applications is always bigger than the cost of using servers, and specific features exist that significantly increase the cost, on top of application placement failure cost, if they fail to be placed.

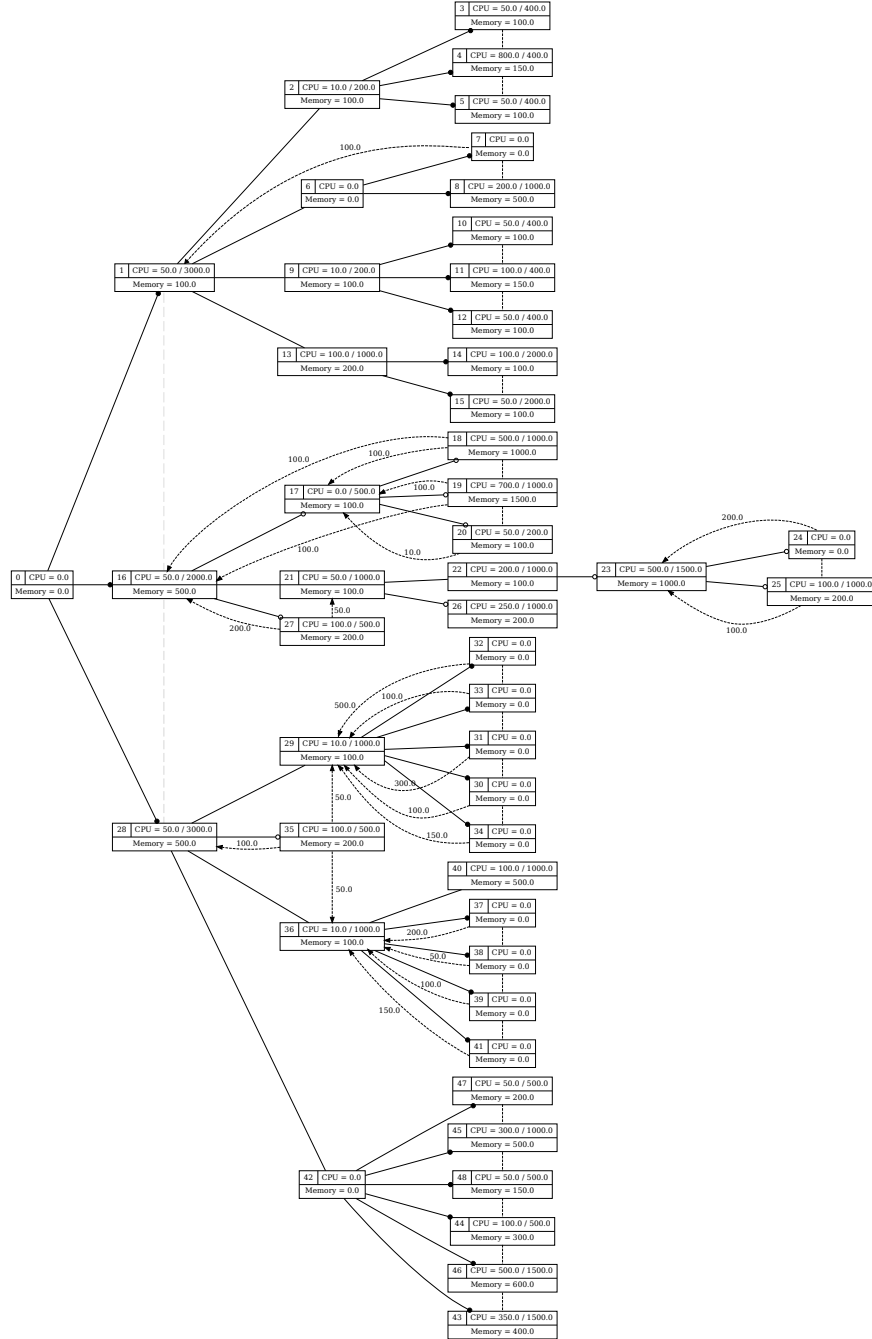


Figure 3.6: The combined feature model containing the CUSTOMSS applications. Each entry in this model corresponds to a feature in one of three real-life applications.

3.6.2 Generated problem models

To evaluate the algorithms for differing problem sizes and varying features models, we generated different problem models. These models are similar in structure to those of the applications studied in the CUSTOMSS project, but the number of features in the feature models can be varied.

The generator creates a collection of servers, a feature model, and a set of applications. For the purposes of the evaluation, we use $\Gamma_s = \{CPU\}$ and $\Gamma_s = \{Memory\}$. First, the servers S are generated. For these evaluations we assume a uniform server configuration with 4000MiB memory and a 2000MHz processor. We also use a uniform server use cost of 1. The costs of failure are chosen relative to this cost.

To create a random feature model \mathcal{F} , first, a collection of features F is generated with random memory requirements from a set $\{500MB, 1GB, 2GB, 2.5GB\}$. Subsequently a feature model tree \mathcal{R} is created. This is done by iteratively selecting nodes that are not in the tree yet and adding them in a relation with a node in the tree as the parent node. To start this process, a random feature is selected as root of the feature tree. There is an equal chance of picking any of the four relation types, and *Alternative* and *Or* relations have between two and six child nodes. Feature models generated in this fashion are similar in structure to those used for the applications in the CUSTOMSS project.

Next, we generate the impact matrix FI^{CPU} . Each feature impacts itself and has a chance of impacting any feature required by it. This is enforced by only letting a feature impact parent features. The CPU impact of a feature on itself is randomly chosen from the set $\{100MHz, 200MHz, 500MHz, 1000MHz\}$, the CPU impact of a feature on a parent feature is added with a probability of 50%, and chosen randomly from the same set. As stated earlier, we assume a homogeneous host capacity.

Selecting features is done by randomly selecting or excluding features, and checking the validity of the resulting feature model with SAT4J [36], an open source SAT solver. This ensures that the selection is feasible according to feature model \mathcal{F} . Features are randomly removed from either the collection of selected features, or from the collection of excluded features. All dependent features are removed as well, ensuring an open variation point is added.

Finally, random applications A are generated using the generated feature selections. Each application and application feature is also assigned costs for failure, randomly chosen from a given set. We use four different scenario's, shown in Table 3.5 for the evaluations, each with a different application failure cost. The Varying Costs (VC) scenario makes use of varying costs for both application and feature failure, and represents the realistic case where the failure of some applications or features can incur a much larger cost than the failure of others. The Identical Costs (IC) scenario by contrast only considers a single cost for both application

Scenario	Application Failure Costs	Feature Failure Costs
Varying Costs (VC)	$\{2, 4, 8, 16, 32\}$	$\{2, 4, 8, 16, 32\}$
Identical Costs (IC)	$\{2\}$	$\{2\}$
Application Costs (AC)	$\{2\}$	$\{0\}$
Feature Costs (FC)	$\{0\}$	$\{2\}$

Table 3.5: The costs for the different evaluation scenarios.

and feature failure. Finally, the Application Costs (AC) and Feature Costs (FC) scenarios consider situations in which either only application failure, or only feature failure are considered. Costs are defined relative to each other, the VC scenario representing the case where the costs of different applications differ by a large order of magnitude.

3.6.3 Evaluation methodology

We will now discuss the different evaluation strategies, and the different quality metrics used in these evaluations.

3.6.3.1 Load-based evaluation

We use a large number of randomly generated problem models in our evaluations. As each of the randomly generated problem models can have very different properties, we need a common parameter to represent the difficulty of finding a good solution. For this, we use the *problem model load*. The problem model load is determined by filling in the feature model for every application, and determining the cheapest possible application. We sum the CPU load for all features and all applications, to determine the total application demand. We then divide this by the sum of all available resources. This variable is indicative of the problem difficulty, as higher load values imply that it becomes more difficult to place all applications.

Load-based evaluation of feature placement algorithms is done by first generating a large batch of problem models: a model is generated for every value of $(s, a, f) \in \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}^3$, thus creating 1000 problem models. We subsequently removed all problem models with a load > 3 . In these cases, it would be preferable to filter applications using admission policies, such as those described in [37], in the management system, ensuring some applications are not accepted by the system.

Due to the nature of ILP solvers, some problems require large amounts of memory or computing time. Additionally, the CPLEX solver allows slight constraint violations, in the order of 10^{-9} , making the solutions to a minority of the problem models violate constraints when the values are rounded. In both cases, the models

causing problems are excluded from the test. The placement quality comparisons were performed using the Stevin Supercomputer Infrastructure at Ghent University, a hardware cluster containing quad core Intel Xeon L5420 nodes with 16 GB ram. This ensures almost no problem models are filtered due to resource constraints.

For the each of the evaluations in this chapter, we repeat this process three times, retaining on average 150 entries for every test set, most being excluded due to the load limitations.

3.6.3.2 Execution time evaluations

The execution speed evaluations of the algorithms were executed on a Linux server with a Dual-Core AMD Opteron(tm) Processor 2212 with 4GiB of memory, and using Scala version 2.9.0.1. For these evaluations, the different versions of the algorithm are executed for varying server, application and feature count.

3.6.3.3 Quality evaluation metrics

The results of a placement can be evaluated in different ways:

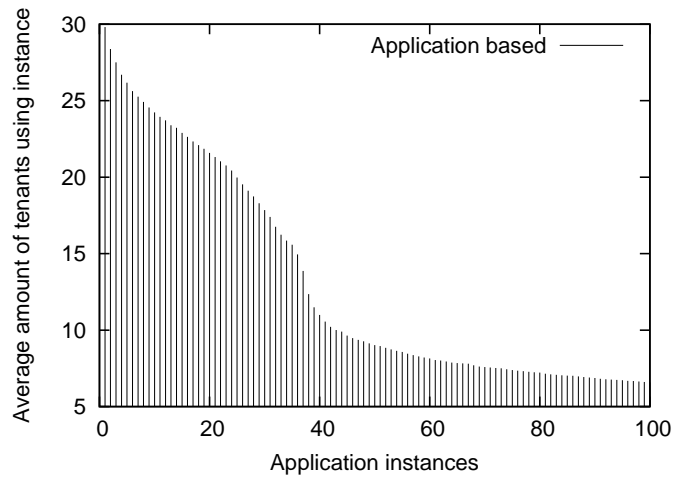
- Cost of Non-Realized Demand (NRD): This metric measures the cost caused by the failure to provision applications. It corresponds to the C_D variable in the formal model, defined in Equation (3.2).
- Cost of Non-Realized Demand Simple (NRDs): This measure is similar to NRD, but does not take cascading failure of features into account.
- Full: Measures the total cost function as defined by the formal model. This corresponds to the total cost C defined in Equation (3.1).

3.7 Evaluation results

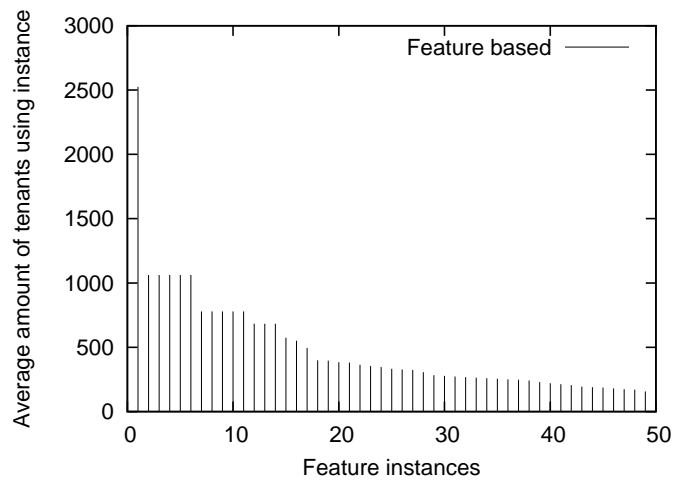
First, we evaluate the feature-based approach by comparing the degree of multi-tenancy that can be achieved compared to an approach where every variant would be provisioned its own instance. We then evaluate the impact of some of the design decisions, determining the maximal amount of quality that can be achieved by the placement when the various constraints are taken into account, and the importance of including these constraints. Then, we evaluate the placement quality of the algorithms compared to an optimal solution, and finally we evaluate the execution speed of the algorithms.

3.7.1 Degree of multi-tenancy

As discussed earlier, there are two approaches to build high-variability applications in clouds: (1) by generating a binary application for every variant and (2) by



(a) Using application instances.



(b) Using feature instances.

Figure 3.7: The maximum number of tenants sharing instances for an application-based approach, where variants are created as monolithic instances, and a feature-based approach, where applications are composed from feature instances.

splitting the application into separate components, which we have referred to as feature instances. In the former case, tenants can only share an application instance if they require the same variant, in the latter case, individual feature instances are shared by tenants. Note that the application-based grouping in Section 3.5.2.2 still makes use of the second approach, and ensures that the different feature instances are considered at the same time during placement.

To compare the degree of multi-tenancy that can be achieved using an application instance approach to that of a feature based approach, we use the CUSTOMSS problem model and count the number of instances that make use of the same application variant for the former, and the number of applications that make use of the same feature for the latter.

By counting the number of identical applications appearing within the problem models used in the next subsections, we can determine how many tenants make use of the same application. We then sort these values, showing frequently used applications first, and average the results over the different problem models used in the CUSTOMSS model evaluation, which will be discussed more in depth later on in Section 3.7.3.2. The results, shown in Figure 3.7a, show that in average problems, 100 different applications must be provisioned that are each used by 5 to 30 tenants. Many of these instances share less than 10 tenants.

When the focus is shifted from application instances to feature instances, and we count how many applications require specific features, we see that a much higher degree of multi-tenancy can be achieved, as seen in Figure 3.7b. Only 49 feature instances are needed, and each instance is shared between 150 to 2500 different tenants.

Our more fine-grained approach where applications are composed using multi-tenant services, thus increases the achievable level of multi-tenancy while at the same time decreasing the number of different instances that must be provisioned.

3.7.2 Impact of the model constraints

We will now determine the impact of various constraints that are defined in the model on the maximum amount of quality that can be achieved by the optimization algorithm. Compared to our earlier work [7], three additional constraints have been added: (1) resource limits, expressing the limited amount of resources that can be used by single application instances, (2) the cascading failure of features, which expresses that parent features, upon which the feature relies for its correct execution, need to be correctly provisioned for the feature to be allocated, and (3) the consideration of server usage costs. Each of these additional constraints will have an impact on the complexity of the problem, and on the minimal achievable cost.

We consider three variants of the ILP formulation:

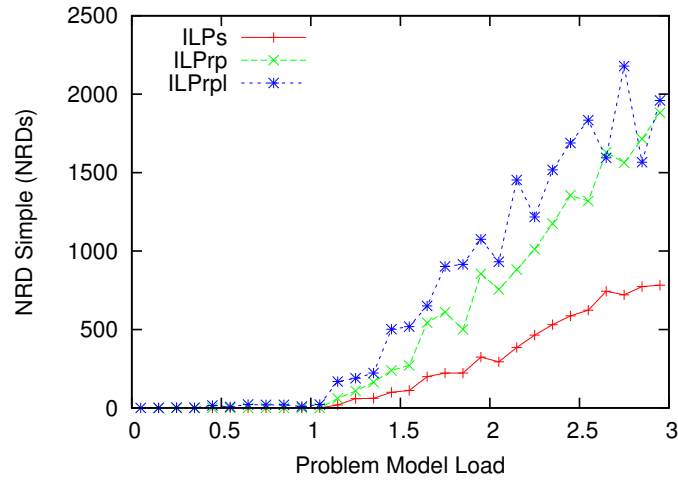
- The first formulation, ILP Simple (ILPs) represents a simple variant of the ILP formulation, where no resource limits, energy requirements or cascading failure are taken into account.
- ILP Requires Parent (ILPrp) is a variant of the ILP formulation that adds the cascading failure of applications, but not the energy requirement of servers nor the resource limitations.
- ILP Requires Parent Limited (ILPrpl) is a variant of the ILP formulation, considering both resource limitations and cascading failure of features.

We will now evaluate the impact of these requirements on the quality of the resulting feature instance allocations using a load-based evaluation using randomly generated problem models. We will do this by comparing the algorithms using the two evaluation functions, NRD and NRDs. The results of these evaluations are shown in Figure 3.8a and Figure 3.8b.

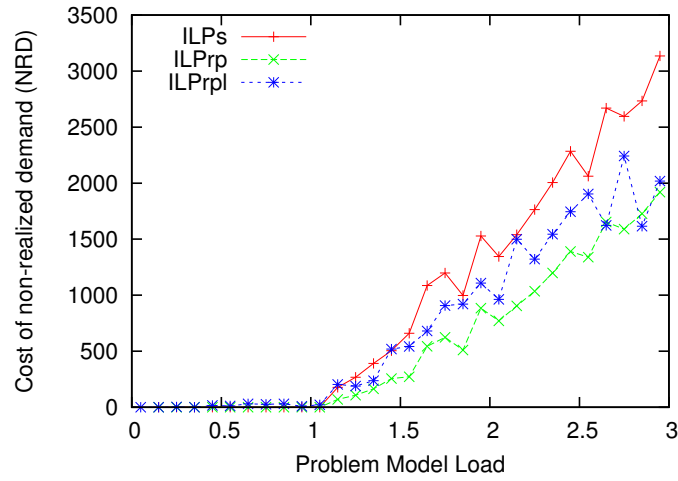
In Figure 3.8a we show the performance of the three variations of the ILP solution with respect to NRDs metric. The addition of the different constraints increases the number of applications that fail. Introducing the cascading failure of features greatly increases the cost of placing applications. This is to be expected, as constraints are added to the ILP formulation that complicate placement, but that are not taken into account by the NRDs evaluation function. Adding resource limitations further increases the cost, as more instances are required to meet the required demand.

When we add the effect of cascading failure in the evaluation, and measure the performance using NRD, the performance of the different ILP formulation changes drastically, as shown in Figure 3.8b. Here, the ILP solution performs badly, which is again to be expected as it allocates features with a high cost of failure, without taking into account whether the parent features are correctly allocated. The performance of both ILPrp and ILPrpl remains identical for both evaluation mechanisms, as no new failed features are introduced.

From this evaluation we can conclude that the use of cascading failure and resource limitations comes at a cost, making it more difficult to find a satisfactory placement on given infrastructure as more requirements are taken into account. This disadvantage is in addition to the increased computational cost incurred by these constraints. If, however, the constraints are required for an accurate representation of the application, they must be considered during placement, as these results show that otherwise the quality of the eventual placement result will be significantly worse.



(a) Performance of ILPs, ILPrp and ILPrpl using the NRDs metric.



(b) Performance of ILPs, ILPrp and ILPrpl using the NRD metric.

Figure 3.8: Performance of ILPs, ILPrp and ILPrpl using different quality evaluation metrics using the VC scenario.

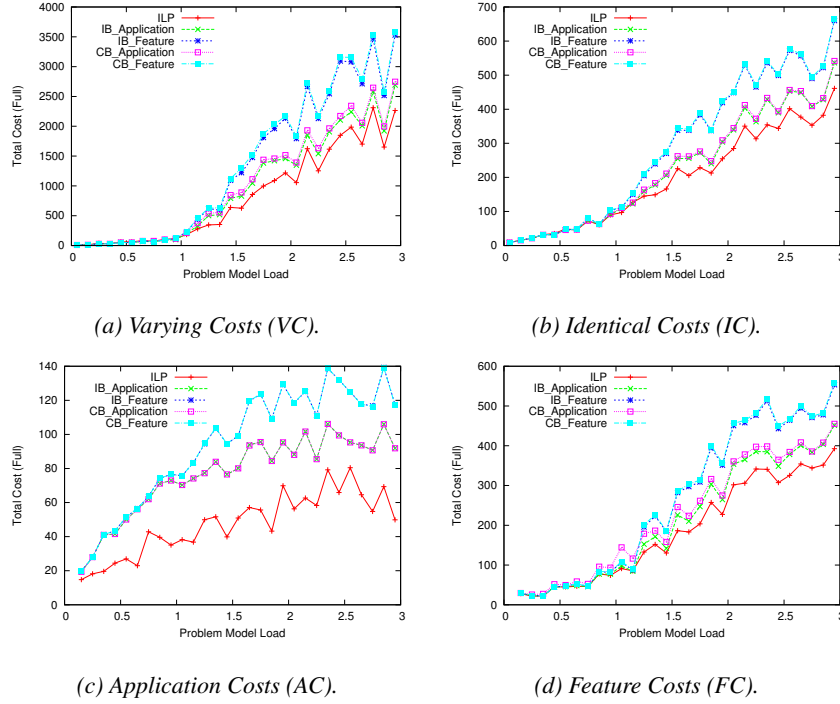


Figure 3.9: Performance of the heuristics and the ILP algorithm for the different scenarios.

3.7.3 Placement quality

We evaluate the placement quality using both the load-based approach as discussed in the previous section, and the CUSTOMSS model.

3.7.3.1 Generated problem models

We use the load-based evaluation with generated problem models for the different scenarios which we defined in Table 3.5, and evaluate the performance of the algorithms using the total cost evaluation function (Full) defined earlier. Figure 3.9a shows that in the Varying Costs (VC) scenario with varying costs, the **IB_application** and **CB_application** algorithms perform significantly better than the **IB_feature** and **CB_feature** algorithms. This indicates that an application-based feature grouping strategy performs well in practice. In both cases, the IB server ordering strategy performs slightly better than the CB approaches, but these differences are less significant.

As shown in Figure 3.9b, the application-based approach works remarkably well when both features and applications impact the cost of placing features, as

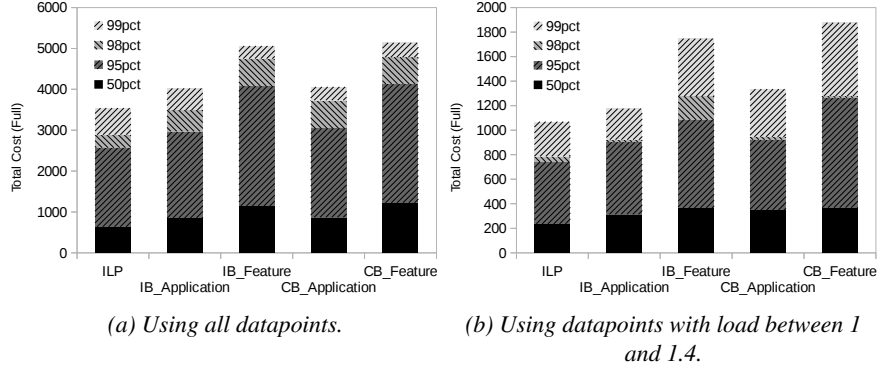


Figure 3.10: Percentiles for the performance of the heuristic and ILP algorithms for the VC scenario.

we do in the IC scenario, even if these costs are kept constant, yielding significant improvements when compared to a purely feature-based approach.

The results for the AC scenario are shown in Figure 3.9c. In this scenario, the different algorithm variations all perform more or less the same, and significantly worse than the ILP results. While this may seem counterintuitive, considering the feature placement focuses on applications in their entirety, this is not unexpected: this phenomenon is caused by the homogeneous nature of the different applications, which makes each application equally important in the placement, making it difficult to decide on which applications to exclude.

Even when no costs for application failure are taken into account, as in the FC scenario, an application-based approach again performs best, as we show in Figure 3.9d. It is however to be noted that the combination of a purely CB approach for servers, combined with an application-based approach for grouping, can sometimes result in bad performance, as seen in the $[1, 1.2]$ region of the plot.

As explained previously, these evaluations make use of randomly generated problem models. While a load-based approach groups elements according to their difficulty, variations in feature models can still cause large differences in the eventual quality of the placement. We show the percentiles for the VC scenario in Figures 3.10. Figure 3.10a shows the percentiles for the entire test set, with loads between 0 and 3. Figure 3.10b shows the percentiles for the problem entries with loads in $[1, 1.4]$, an interesting region as it represents a situation that could potentially occur when application demand spikes. In both charts, the tendencies explained previously reoccur: an application-based approach performs better than a feature-based approach, and an instance based order for servers performs better than a purely cost based approach. On average, the application-based approach performs $\pm 25\%$ better than a feature-based approach, in the $[1, 1.4]$ region, while when

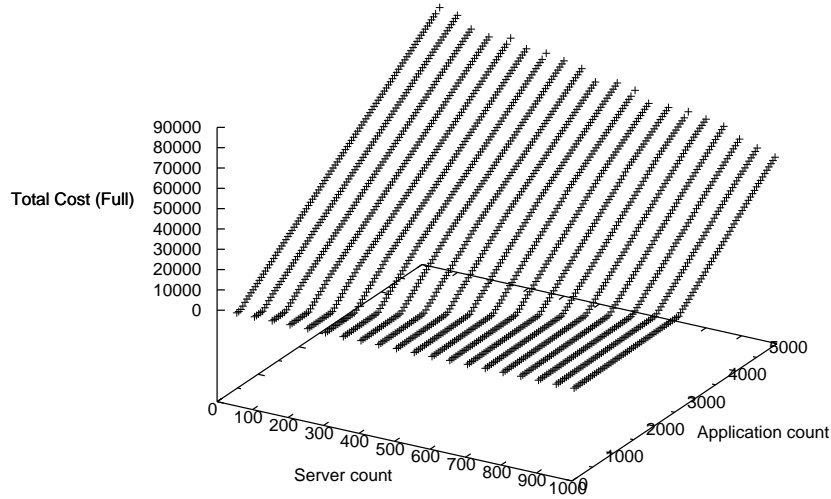


Figure 3.11: The quality of feature placement as a function of the number of servers and applications.

all results are considered, this difference increases to $\pm 42\%$. Using a cost-based approach rather than a feature-based approach also slightly increases placement quality, by ± 12 in the $[1, 1.4]$ range, but only by a negligible $\pm 1\%$ when all evaluation results are included.

It is noticeable that, globally, the instance-based approach has a similar worst-case performance as the cost-based approach, but it occurs less often. In an overload situation, the difference between both approaches is only noticeable in the 99th percentile. It is of note that, for loads in the $[0.5, 1]$ range, not shown here, the different algorithms often perform slightly better than the ILP-based algorithm due to the assignment of non-integer values to integer variables that occurs in CPLEX.

The results in this section demonstrate that an application-based approach to feature placement, where the algorithm tries to place all of the features of applications at once, performs significantly better compared to a feature-based approach, where the features are considered separately. Note that in both cases, the placed feature instances are multi-tenant services and shared between applications, as discussed previously. When servers are selected, it is best to take into account how well applications fit on the server, as it is done with the IB approach, but the improvements of this choice compared to a purely cost-based approach are limited, and this change only impacts problem models in the 98th and 99th percentiles.

3.7.3.2 CUSTOMSS problem model

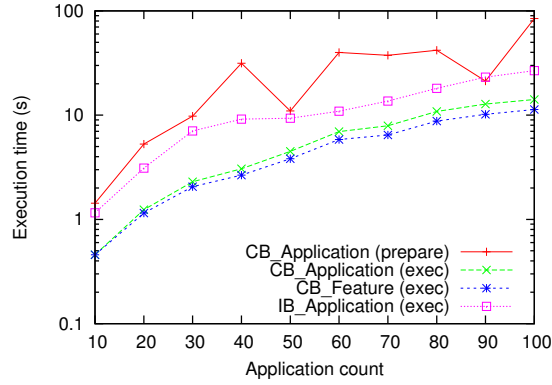
Using the CUSTOMSS model, we assessed the costs when the number of servers is varied using the total cost evaluation function (Full) and using the **CB.application** algorithm. Figure 3.11 shows the quality of placement considering varying application and server counts. These graphs were generated by using the CUSTOMSS feature model, and creating applications as described in Section 3.7. To generate the problem for n applications, a single application is generated and added to the problem generated using $n - 1$ applications. Because of this, the graph shows the impact of iteratively adding applications and servers to a cloud.

We see that as applications are added, the cost for failed placement increases. The plot consist of two intersecting planes. One plane is nearly flat, with only a slight slope as application counts increase, as this ensures more servers that need to execute applications, increasing the server use cost. The second plane shows a steeper increase in costs, as in these points application failure occurs, incurring a larger cost. The intersection of the planes shows the point at which too many applications are allocated, and a cost of failure is incurred.

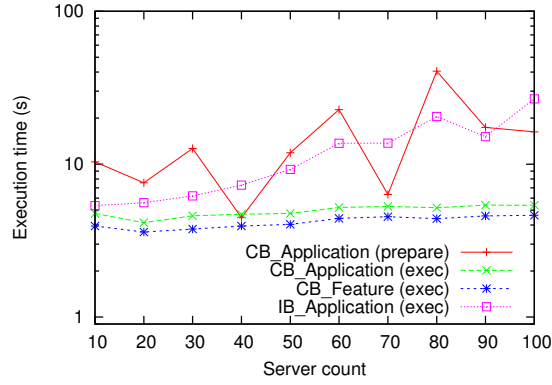
3.7.4 Execution speed evaluation

We consider the execution speed of the algorithm for increasing application counts, server counts and feature counts, using randomly generated problem models. In the graphs, we separate the total execution time of the algorithms into two parts: a preparation time and an execution time. Each data point is an average of 20 executions using randomly generated feature models with the parameters discussed in the previous section. The preparation time is the part of the computation that can be executed when an application is added, and is needed only once. This mainly comprises of the time required to create feature model configurations using the feature model conversion as discussed in Section 3.5.2.4. We only show the preparation cost for the **CB.application** algorithm, but these costs are identical for the other three algorithms. The execution time is the time required to execute feature placement, provided the preparation step has been executed in advance. This step must be executed when applications are actually running, to take changing application demands and the addition of new applications into account.

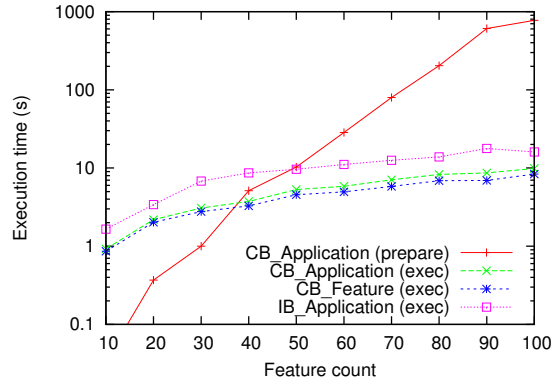
As shown in Figure 3.12a, increasing the number of applications increases the execution duration of the algorithm in a more or less linear fashion. It is notable that the application-based approach performs a bit worse than a feature-based approach, as some applications will be considered twice by it, once in an application-based fashion and once in a per-feature order, while the feature-based approach only considers each feature once. Similarly, an IB approach to server ordering also increases the execution duration w.r.t. a CB approach, as it requires an additional sort operation. As the number of applications doubles, so does the preparation



(a) Varying application counts, using $|S| = 50$ and $|F| = 50$.



(b) Varying server counts, using $|F| = 50$ and $|A| = 50$.



(c) Varying feature counts, using $|S| = 50$ and $|A| = 50$.

Figure 3.12: The execution speed of the feature placement algorithm as a function of varying application counts, server counts and feature counts.

duration, as this preparation runs for each application. Such a trend is noticeable in the plot, but the preparation duration does vary significantly from problem model to problem model.

In Figure 3.12b, we show the performance of the algorithm in the face of varying server counts. Once more, the high variability in preparation execution time is demonstrated, as for each data point the same number of applications are considered. We notice that the CB algorithms are largely independent from the number of servers considered. This opposed to the IB approach, for which the required computational time increases with the server count.

Figure 3.12c demonstrates the execution speed considering varying feature counts. Here we notice a significant increase in preparation time as feature counts increase, but only limited impact on the execution time of the algorithms. We again observe that an application based approach requires more time to execute, as do the IB algorithms.

Feature models become more complicated to manage as the number of features increases, especially as within this chapter we only consider customization changes, in which a change implies the use of a different code module that must be maintained. Because of this, we do not expect the models to become prohibitively large, ensuring the preparation duration will remain acceptable. Furthermore, this has little impact on the execution time of the algorithm. We can conclude that the CB algorithms scale well in terms of application and server counts, and that, due to the possibility of preparing applications before execution, increasing feature counts can be managed as well. The IB algorithms do not scale as well when server counts increase as the CB approach. This implies that, while the IB algorithms perform slightly better than the CB algorithms, it can be preferable to make use of the latter in large server configurations to improve the speed with which placements can be determined. The presented algorithms still make use of a centralized approach, implying they could become a bottleneck as the size of the cloud increases. To address this, techniques such as those we presented in [28] could be used to increase the scalability of the algorithms in larger clouds, by reusing the centralized algorithms within a hierarchically structured management infrastructure.

3.8 Conclusions

In this chapter, an approach for managing highly customizable applications using feature modeling and SPLE techniques was presented. We first presented the feature placement problem, determining the different inputs, outputs and requirements, which we subsequently formalized. Then, heuristics were developed and compared to the optimal ILP-based algorithm. In this evaluation we used the feature models from existing applications, ensuring the presented techniques are applicable to

realistic cases, and using generated feature models, ensuring the performance remains similar for different cases.

For the considered cases, using feature instances rather than application instances greatly increased the achievable level of multi-tenancy. In the former approach, each instance can be shared between up to at least 150, while in the latter approach some instances can only be used to serve ± 5 tenants. We found that an application-centric approach to feature placement, where the services corresponding to application features are placed at once, performs 25% to 40% better than a feature-based approach, where the features are placed independently without taking their relations into account. We also conclude that an approach where servers are chosen based on a best fit approach performs best, albeit with a penalty to execution times. For three out of four scenarios, the application-based approach to feature placement performs close to the optimal algorithm, failing only when no differences between applications occur. The presented heuristics scale well, with execution times remaining under 10s for the considered cases.

In future work we will extend the discussed approach to achieve dynamic application placement, and we will incorporate the designed algorithms in a cloud management platform as a proof-of-concept.

Acknowledgement

Hendrik Moens is funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). This research is partly funded by the iMinds CUSTOMSS [2] project. This work was carried out using the Stevin Supercomputer Infrastructure at Ghent University.

Addendum

The dynamic application placement which is referenced in the conclusion of this chapter is discussed in Chapter 4.

References

- [1] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. *Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility*. Future Generation Computer Systems, 25(6):599–616, jun 2009. doi:10.1016/j.future.2008.12.001.
- [2] *CUSTOMSS: CUSTOMization of Software Services in the cloud* [online]. 2013. accessed on 1/2013. Available from: <http://www.iminds.be/en/projects/overview-projects/p/detail/customss>.
- [3] K. Pohl, G. Böckle, and F. Van Der Linden. *Software product line engineering: foundations, principles, and techniques*. Springer-Verlag New York, Inc., 2005.
- [4] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl. *Variability modeling to support customization and deployment of multi-tenant-aware Software as a Service applications*. In Proceedings of the ICSE Workshop on Principles of Engineering Service Oriented Systems (PESOS 2009), pages 18–25. IEEE, may 2009. doi:10.1109/PESOS.2009.5068815.
- [5] F. Wuhib, R. Stadler, and M. Spreitzer. *Gossip-based Resource Management for Cloud Environments*. In Proceedings of the 6th International Conference on Network and Service Management (CNSM 2010), pages 1–8. IEEE, oct 2010. doi:10.1109/CNSM.2010.5691347.
- [6] Y. Li, F.-H. Chen, X. Sun, M.-H. Zhou, W.-P. Jiao, D.-G. Cao, and H. Mei. *Self-Adaptive Resource Management for Large-Scale Shared Clusters*. Journal of Computer Science And Technology, 25(5):945–957, sep 2010. doi:10.1007/s11390-010-1075-6.
- [7] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. D. Turck. *Feature Placement Algorithms for High-Variability Applications in Cloud Environments*. In Proceedings of the 13th Network Operations and Management Symposium (NOMS 2012), pages 17–24. IEEE, apr 2012. doi:10.1109/NOMS.2012.6211878.
- [8] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici. *A scalable application placement controller for enterprise data centers*. In Proceedings of the 16th International Conference on World Wide Web (WWW 2007), pages 331–340. ACM, may 2007. doi:10.1145/1242572.1242618.
- [9] K. Zhang, X. Zhang, W. Sun, H. Liang, Y. Huang, L. Zeng, and X. Liu. *A Policy-Driven Approach for Software-as-Services Customization*. In Proceedings of the 9th IEEE International Conference on E-Commerce and the 4th

- IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, (CEC/EEE 2007), pages 123–130. IEEE, jul 2007. doi:10.1109/CEC-EEE.2007.9.
- [10] W. Sun, X. Zhang, C. J. Guo, P. Sun, and H. Su. *Software as a Service: Configuration and Customization Perspectives*. In IEEE Congress on Services Part II (services-2), pages 18–24. IEEE, sep 2008. doi:10.1109/SERVICES-2.2008.29.
- [11] M. Abu-Matar and H. Gomaa. *Feature Based Variability for Service Oriented Architectures*. In Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture (WICSA 2011), pages 302–309. IEEE, jun 2011. doi:10.1109/WICSA.2011.47.
- [12] M. Abu-Matar and H. Gomaa. *Variability Modeling for Service Oriented Product Line Architectures*. In Proceedings of the 15th International Software Product Line Conference (SPLC 2011), pages 110–119. ACM, aug 2011. doi:10.1109/SPLC.2011.26.
- [13] S. T. Ruehl and U. Andelfinger. *Applying Software Product Lines to create Customizable Software-as-a-Service Applications*. In Proceedings of the 15th International Software Product Line Conference (SPLC 2011), pages 16:1–16:4. ACM, aug 2011. doi:10.1145/2019136.2019154.
- [14] G. H. Alférez and V. Pelechano. *Context-Aware Autonomous Web Services in Software Product Lines*. In Proceedings of the 15th International Software Product Line Conference (SPLC 2011), pages 100–109. ACM, aug 2011. doi:10.1109/SPLC.2011.21.
- [15] H. Gomaa, K. Hashimoto, M. Kim, S. Malek, and D. a. Menascé. *Software adaptation patterns for service-oriented architectures*. In Proceedings of the 2010 ACM Symposium on Applied Computing (SAC 2010), pages 462–469. ACM Press, mar 2010. doi:10.1145/1774088.1774185.
- [16] B. Urgaonkar, A. L. Rosenberg, and P. Shenoy. *Application Placement on a Cluster of Servers*. International Journal of Foundations of Computer Science, 18(05):1023–1041, oct 2007. doi:10.1142/S012905410700511X.
- [17] C. Adam and R. Stadler. *Service Middleware for Self-Managing Large-Scale Systems*. IEEE Transactions on Network and Service Management, 4(3):50–64, dec 2007. doi:10.1109/TNSM.2007.021103.
- [18] J. Rolia, A. Andrzejak, and M. Arlitt. *Automating Enterprise Application Placement in Resource Utilities*. In M. Brunner and A. Keller, editors, Self-Managing Distributed Systems, volume 2867 of *Lecture Notes in Computer*

- Science*, pages 118–129. Springer Berlin Heidelberg, 2003. doi:10.1007/978-3-540-39671-0_11.
- [19] I. Whalley and M. Steinder. *Licence-aware management of virtual machines*. In Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011), pages 169–176. IEEE, may 2011. doi:10.1109/INM.2011.5990688.
- [20] D. Breitgand and A. Epstein. *SLA-aware Placement of Multi-Virtual Machine Elastic Services in Compute Clouds*. In Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011), pages 161–168. IEEE, may 2011. doi:10.1109/INM.2011.5990687.
- [21] C. Peoples, G. Parr, and S. McClean. *Context-Aware Characterisation of Energy Consumption in Data Centres*. In Proceedings of the 3rd IEEE/IFIP International Workshop on Management of the Future Internet (ManFI), pages 1246–1253. IEEE, may 2011. doi:10.1109/INM.2011.5990573.
- [22] J. Xu and J. a. B. Fortes. *Multi-Objective Virtual Machine Placement in Virtualized Data Center Environments*. In Proceedings of the 2010 IEEE/ACM International Conference on Green Computing and Communications & International Conference on Cyber, Physical and Social Computing, pages 179–188. IEEE, dec 2010. doi:10.1109/GreenCom-CPSCoM.2010.137.
- [23] X. Zhu, C. Santos, D. Beyer, J. Ward, and S. Singhal. *Automated application component placement in data centers using mathematical programming*. International Journal of Network Management, 18(6):467–483, nov 2008. doi:10.1002/nem.707.
- [24] D. Carrera, M. Steinder, I. Whalley, J. Torres, and E. Ayguadé. *Utility-based placement of dynamic web applications with fairness goals*. In Proceedings of the 11th Network Operations and Management Symposium (NOMS 2008), pages 9–16. IEEE, apr 2008. doi:10.1109/NOMS.2008.4575111.
- [25] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. *Dynamic placement for clustered web applications*. In Proceedings of the 15th International Conference on World Wide Web (WWW 2006), pages 595–604. ACM, may 2006. doi:10.1145/1135777.1135865.
- [26] C. Low. *Decentralised Application Placement*. Future Generation Computer Systems, 21(2):281–290, feb 2005. doi:10.1016/j.future.2003.10.003.
- [27] T. Kimbrel, M. Steinder, M. Sviridenko, and A. Tantawi. *Dynamic Application Placement Under Service and Memory Constraints*. In S. Nikolettseas, editor, Experimental and Efficient Algorithms, volume 3503 of *Lecture Notes*

- in Computer Science*, pages 391–402. Springer Berlin Heidelberg, 2005. doi:10.1007/11427186_34.
- [28] H. Moens, J. Famaey, S. Latré, B. Dhoedt, and F. De Turck. *Design and Evaluation of a Hierarchical Application Placement Algorithm in Large Scale Clouds*. In Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011), pages 137–144. IEEE, may 2011. doi:10.1109/INM.2011.5990684.
- [29] L. Roderio-Merino, L. M. Vaquero, V. Gil, F. Galán, J. Fontán, R. S. Montero, and I. M. Llorente. *From infrastructure delivery to service management in clouds*. Future Generation Computer Systems, 26(8):1226–1240, oct 2010. doi:10.1016/j.future.2010.02.013.
- [30] C. Chapman, W. Emmerich, F. G. Marquez, S. Clayman, and A. Galis. *Elastic service definition in computational clouds*. In Proceedings of the 12th IEEE/IFIP Network Operations and Management Symposium Workshops (NOMS 2010), pages 327–334. IEEE, apr 2010. doi:10.1109/NOMSW.2010.5486555.
- [31] pure-systems GmbH. *pure::variants User’s Guide* [online]. 2012. Last accessed: December 2012. Available from: <http://www.pure-systems.com/Documentation.116.0.html>.
- [32] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao. *A Framework for Native Multi-Tenancy Application Development and Management*. In Proceedings of the 9th IEEE International Conference on E-Commerce and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, (CEC/EEE 2007), pages 551 – 558. IEEE, jul 2007. doi:10.1109/CEC-EEE.2007.4.
- [33] H. Cai, N. Wang, and M. J. Zhou. *A Transparent Approach of Enabling SaaS Multi-tenancy in the Cloud*. In Proceedings of the 6th World Congress on Services (SERVICES 2010), pages 40–47. IEEE, jul 2010. doi:10.1109/SERVICES.2010.48.
- [34] M. Hajjat, X. Sun, Y.-W. E. Sung, D. Maltz, S. Rao, K. Sripanidkulchai, and M. Tawarmalani. *Cloudward Bound: Planning for Beneficial Migration of Enterprise Applications to the Cloud*. SIGCOMM Computer Communication Review, 40(4):243–254, aug 2010. doi:10.1145/1851275.1851212.
- [35] *IBM ILOG CPLEX 12.2* [online]. 2011. Available from: <http://www-01.ibm.com/software/integration/optimization/cplex-optimizer>.
- [36] *SAT4J 2.2.2* [online]. 2011. Available from: <http://www.sat4j.org/>.

-
- [37] N. Leontiou, D. Dechouniotis, and S. Denazis. *Adaptive Admission Control of Distributed Cloud Services*. In Proceedings of the 6th International Conference on Network and Service Management (CNSM 2010), pages 318–321. IEEE, oct 2010. doi:10.1109/CNSM.2010.5691214.

4

Allocating Resources for Customizable Multi-Tenant Applications in Clouds Using Dynamic Feature Placement

H. Moens, B. Dhoedt and F. De Turck

**Accepted for publication, Elsevier Future Generation Computer Systems
(FGCS)**

In Chapters 2 and 3, a feature-based approach for modeling and managing customizable multi-tenant SaaS applications was introduced. In this chapter, we describe dynamic feature placement algorithms that minimize migrations between subsequent invocations. These algorithms extend the algorithms presented in Chapter 3, adding migration-awareness, and use similar principles to achieve their results. We evaluate the algorithms in dynamic scenarios where applications are added and removed throughout the evaluation scenario. We find that the developed algorithm achieves a low cost, while resulting in few resource migrations. In our evaluations, we observe that adding migration-awareness to the management algorithms reduces the number of instance migrations by more than 77% and reduces the load moved between instances by more than 96% when compared to a static management approach. Despite the reduction in the number of migrations, a cost that is on average less than 3% more than the optimal is achieved.

4.1 Introduction

In recent years there has been a growing interest in using cloud computing as a means of offloading applications and reducing costs. An efficient way in which costs of cloud deployments may be reduced is through multi-tenancy. In a traditional model, every client is provided with a separate application instance. In multi-tenant environments, however, a single instance can be used by multiple clients. Every client of the application is referred to as a *tenant* and is considered to be an organization with its own end users. The major advantage of this approach is that it makes it possible to use less application instances to provision the service to each of these tenants, reducing the cost of offering the service. Additionally, this approach makes it easier to scale applications, as sudden increases in numbers of users result in smaller increases of the number of required instances. Spikes in the numbers of end users of one tenant can also be compensated by decreasing numbers of end users of other tenants.

Building customizable multi-tenant applications is however difficult, and it is often hard to make changes that are not just cosmetic configuration changes. Therefore, multi-tenant applications are often offered as a take it or leave it package, with only limited customizability. This approach works well for many application types, especially when tenant needs are very similar, but there are use cases where a very high degree of customizability is required. This is the case in various domains, such as for example document processing, medical communications and medical information management. These application cases are all characterized by the fact that the offered platform is used by a relatively small number of large tenants that each have a large number of end users. Each of these tenants may request its own customizations to the application platform, and as many tenants are large, it is difficult to deny these requests. Currently such customizations are often developed on an ad-hoc basis. This however poses difficulties concerning the management of these customizations and as separate tenants have custom tailored codebases, it becomes impossible to share resources between end users. This problem becomes even more complex when clients are also split up into multiple departments that each require specific customizations and when the application platform is offered to other clients using resellers. An illustration of the various tenant types is shown in Figure 4.1.

Using feature modeling [1], this issue can be addressed. Feature modeling is an approach where the variability of an application is modeled using a *feature model*. The customizability of the application is represented by a collection of features, a representation of specific functionality that may or may not be added to the application, and their relations. Features can be implemented using aspect oriented programming [2], as configuration changes, or as custom code modules. While feature modeling is an interesting approach for managing the codebase of

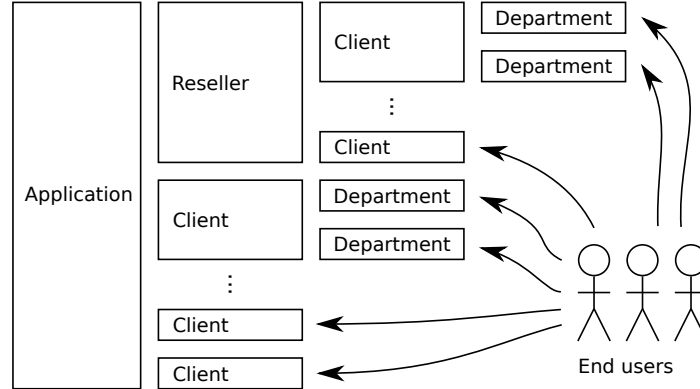


Figure 4.1: An illustration of a scenario where the application is offered to end users by a hierarchy of the three types of tenants: resellers, clients, and client departments. Resellers can also sell the application to other resellers, and departments may also be further divided into smaller departments. At every level, different application customizations may be required.

customizable applications, this still results in customized application binaries, making it impossible to use multi-tenancy in the resulting applications. We previously proposed an approach where applications are separated into multiple interacting components, effectively making sure every feature is implemented in its own service [3]. The entire application is then composed from the various components, thus forming a service oriented architecture. As every code module is itself multi-tenant, the advantages of multi-tenancy can be attained.

Splitting applications into multiple components however impacts the performance of the applications, complicating cloud management. Additionally, the chosen features should be taken into account by the management system. It may e.g. be cheaper to use an existing high-performance instance for a tenant that does not pay for such an instance rather than to allocate a low-performance instance specifically for this tenant. We previously addressed resource allocation taking this information into account, referred to as feature placement, in [4] and [5], but the approach however resulted in a static resource allocation, that has to be recomputed periodically. In doing so, the number of migrations is not taken into account, which adversely impacts the performance of the system when services are migrated. Furthermore, adding applications is relatively expensive and slow as they can only be added whenever the algorithm is invoked rather than immediately when they are added.

In this chapter, we focus on dynamic feature placement algorithms that relocate and reconfigure features when changes occur. In computing these changes, the previous state of the system is taken into account, minimizing the number of application changes and instance migrations. We present both ILP-based algorithms

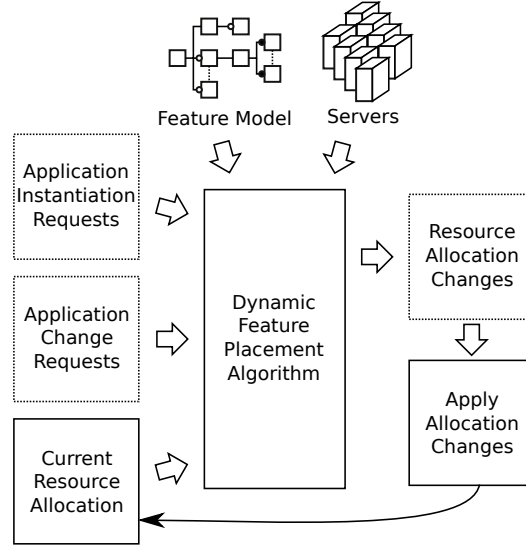


Figure 4.2: The dynamic feature placement, its inputs and its function within a management system.

and a heuristic algorithm, the Dynamic Feature Placement Algorithm (DFPA). Figure 4.2 shows the algorithm inputs and how it functions within a cloud management system.

The remainder of this chapter is structured as follows. In the next section we discuss related work. Afterwards, in Section 4.3 we describe how the system in which the feature placement algorithm is executed is structured, and how feature modeling is used within the approach. A formal problem representation is presented in Section 4.4. In Section 4.5, we present the DFPA. The evaluation setup is presented in Section 4.6, and the algorithms are then evaluated in Section 4.7. Finally, we state our conclusions in Section 4.8.

4.2 Related Work

To manage variability when building applications, Software Product Line Engineering (SPLE) [6] techniques are used. Instead of managing multiple codebases for different application variants, a single codebase is used, and different variants are generated using SPLE tools. In traditional SPLE applications, the application configuration is however generally decided at compile-time, making it ill-suited for cloud environments. Dynamic SPLE [7] can be used to configure and reconfigure software variants at runtime, making it more suited for cloud environments. This makes it possible to characterize runtime variability and reconfigure applications at

runtime. SPLE has been used in cloud environments [8–10], but the approaches tend to focus mostly on development, deployment and configuration. We however focus specifically on runtime resource allocation for customizable SPLE applications by adding awareness of application variability to the cloud management algorithms. Similarly, other work [11–14] focuses on the design-time variability of the applications rather than on their runtime management, the latter being the focus of this chapter.

In this chapter, we focus on cloud resource allocation [15] and design dynamic management algorithms that are aware of application customizability. In particular, we focus on extending the generic application placement problem [16] and cloud application placement problem [17–19] to incorporate both multi-tenancy and software variability. The approach we use for multi-tenancy uses component-based applications composed using a Service-Oriented Architecture (SOA), making the relations between components another important consideration.

Our approach is similar to application component placement approaches [20–23], where applications consisting of multiple components, represented as a set of Virtual Machines (VMs), are placed within a datacenter taking the relation between components into account. These approaches typically focus on colocation, anti-colocation and other placement constraints used to impact application security, performance, and reliability. These approaches however do not take multi-tenancy on a VM-level into account, meaning the approaches do not support sharing components between different multi-component applications. Our approach further differs by the inclusion of SPLE principles within the management system, making it possible to take application variability into account during resource allocation.

[24] both focuses on VM placement taking energy efficiency into account. Our approach also incorporates server use costs, but differs in that we focus on managing multi-tenant applications where multiple applications can make use of a single instance. Additionally, our algorithm also adds support for software variability within the management algorithm itself. Energy efficiency and server usage costs are incorporated in an application placement system in [25]. The authors however focus on the placement at a VM level, while our approach focuses on managing multi-tenant applications where multiple applications can make use of a single instance, meaning more fine-grained control is needed. Furthermore, our algorithm also adds explicit support for software variability. This enables the management system to dynamically fill in undecided variability, referred to as open variation points [10], at runtime.

Application placement algorithms typically focus on server CPU and memory resources [19, 26–28] or bandwidth limitations [29]. In this chapter we make use of a generalized approach where arbitrary resources of different types can be used. Such an approach was previously used in [30], but our approach goes further as it allows the definition of multiple resources rather than just supporting two

arbitrary resource types, enabling the management of high-variability applications with heterogeneous resource demands.

This chapter extends our previous work related to feature placement [4, 5], which focused on the static feature placement problem, and describes and evaluates new dynamic feature placement algorithms that can be used in a context where applications are added and removed through time. The modeling approach we use is further based on our work on feature model conversion [3], which focuses more on how the code modules themselves are defined and how customizable applications within our approach can be designed rather than on how these modules are managed at runtime, which is the focus of this chapter.

An overview of how the DFPA, which is introduced in this chapter, compares to the most relevant previous work is shown in Table 4.1. In the table, we focus specifically on approaches using and supporting multi-component cloud applications. We compare multiple properties for the various publications. These properties are the following:

1. Multi-component applications: Whether the approaches support the management of applications consisting out of multiple components.
2. Application variability: Whether application variability and customizability is considered in the work.
3. Resource management: Whether the work addresses the runtime management and resource allocation of these applications.
4. Dynamic cloud management: Whether the management of applications is dynamic (i.e. resource demand can vary over time and application components can be migrated).
5. Service management: Whether the management focuses on services instead of VMs. This makes it possible to consider how application resources are allocated within VMs in addition to how the VMs themselves are allocated.
6. Service and VM migrations: Whether both service and VM migration is supported (i.e. VMs can be moved between nodes and application load can be shifted between VMs without moving the VMs themselves).
7. Generalized resources: Whether the approach supports generalized arbitrary resources, and not just CPU and memory capacity.
8. Server use minimization: Whether the approach takes server utilization into account, enabling energy-efficient resource management.

¹Focuses on modeling of applications, not on managing them at runtime.

²On-line algorithm, but does not migrate instances over time.

	[10], [14]	[17]	[20]	[21]	[22]	[23]	[4], [5]	DFPA
Multi-component applications	+	+	+	+	+	+	+	+
Application variability	+	-	-	-	-	-	+	+
Resource management	- ¹	+	+	+	+	+	+	+
Dynamic cloud management	N/A	- ²	+	+	-	-	-	+
Service Management	N/A	-	-	-	-	-	+	+
Service and VM migrations	N/A	-	-	-	-	-	-	+
Generalized resources	N/A	-	+	+	+	+	+	+
Server use minimization	N/A	-	-	-	-	-	+	+

Table 4.1: An overview of the relation between the DFPA introduced in this chapter and previous work.

4.3 Feature Placement

SPLE [6] techniques can be used to model an application as a collection of features and relations between features. Both the features themselves, which encapsulate specific functionality that may or may not be included in an application, and their relations are important. It may for example be the case that the inclusion of a feature implies that other features must be included or conversely that the inclusion of a feature prevents another feature from being selected. To make it easier to reason using these relations, feature models are often defined hierarchically. The relation between child nodes can be chosen as one out of four types:

- **Mandatory(a, b):** If a feature **a** is included, the feature **b** must be included as well.
- **Optional(a, b):** If a feature **a** is included, the feature **b** may be included. Conversely, the feature **b** must not be included if **a** is not included.
- **Alternative(a, S):** If a feature **a** is included exactly one of the features contained in the set **S** must be included. If **a** is not included, none of the features in **S** may be included.
- **Or(a, S):** If a feature **a** is included, at least one of the features contained in the set **S** must be included. If **a** is not included, none of the features in **S** may be included.

In our feature placement approach, applications are constructed using a service oriented architecture and are composed out of various feature instances. Only features that refer to actual code modules are used, while other features such as smaller configuration changes are handled at runtime. The code module of a feature can then be instantiated as a VM in a cloud. We assume that feature instances are multi-tenant, meaning they can be used to serve multiple applications for different clients. For more information as to how features are represented and how feature

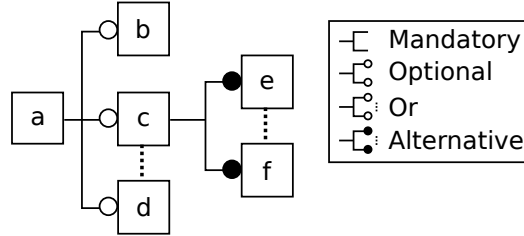


Figure 4.3: An illustrative example of a small feature model using the Pure::Variants notation.

models containing non-code changes can be mapped to feature models containing feature models we refer to [3]. To support this transformation, and add versatility to the feature model representation, two additional non-hierarchical relation types are used:

- **Excludes(a, b)**: If a feature **a** is included, the feature **b** must not be included and vice versa.
- **Requires(a, b)**: A feature **a** may only be included if the feature **b** is included as well.

Figure 4.3 shows an illustrative example a simple feature model consisting of six features. The relations between the various features are expressed using the Pure::Variants notation [31] which we also used in our previous work [3, 5]. This model corresponds to three relations: **Optional(a, b)**, **Or(a, {c, d})**, and **Alternative(c, {e, f})**. Based on this model, a specific configuration of features can be selected to be used in an application. A feature can either be selected, excluded or undecided. A selected feature must always be included for the feature placement of the application to be successful. An excluded feature may not be included in the application under any circumstance. Finally, undecided features may either be included or excluded at runtime, based on what results in the lowest placement cost. It may, e.g. be cheaper to add encryption for a client, even if he has not selected the feature, if only instances with encryption exist than to create a new instance specifically for this client. These undecided features are referred to as *open variation points* [10]. If, for the model in Figure 4.3, {a, c, d} are selected and {b} is excluded, e and f remain as open variation points. Only when the application is deployed will it be decided whether e or f are included.

An overview of the workflow when new requests are added to the system is shown in Figure 4.4. As application placement requests enter the system, the viability of adding them is first evaluated by an application request filter. This filter can make decisions based on multiple factors:

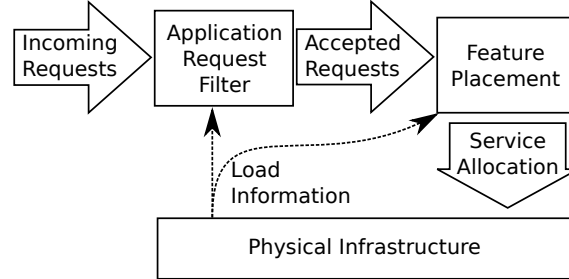


Figure 4.4: An overview of the relevant components within the cloud management system and the communication flow between the components.

- The amount of system resources required for the request can be determined and compared to the remaining resources available within the datacenter. If there are not sufficient remaining resources, the application request should be rejected.
- Another factor that is important when application allocations are initiated is the consideration of network capacity. This is especially important if some features are allocated in different networks and part of the communication must pass over the Internet. Filters such as this can be created by determining the impact on the underlying network of instantiating these services, which we previously discussed in [32].

This filter is invoked whenever a new application configuration is instantiated, which is when a seller enters the feature configuration for a client within the system. Accepted requests are sent to the feature placement system, which runs algorithms responsible for determining where the services out of which the application consists should be allocated on the physical infrastructure. These allocation changes are then enforced by the feature placement system resulting in a change of the physical allocation. System load information is then sent back to the application request filter and feature placement components, which can be used to update allocations and to allow or reject future application instantiation requests.

4.4 Feature Placement Model

We previously presented a formal problem formulation for the static feature placement problem in [5]. In this section we briefly describe the problem model, focusing on the variables that are needed to add migration-awareness to the model. For a more detailed discussion of the model parameters and formulation we refer to [5]. We will end this section by discussing how migration-awareness can be added to the model to make it useful for dynamic application placement scenarios.

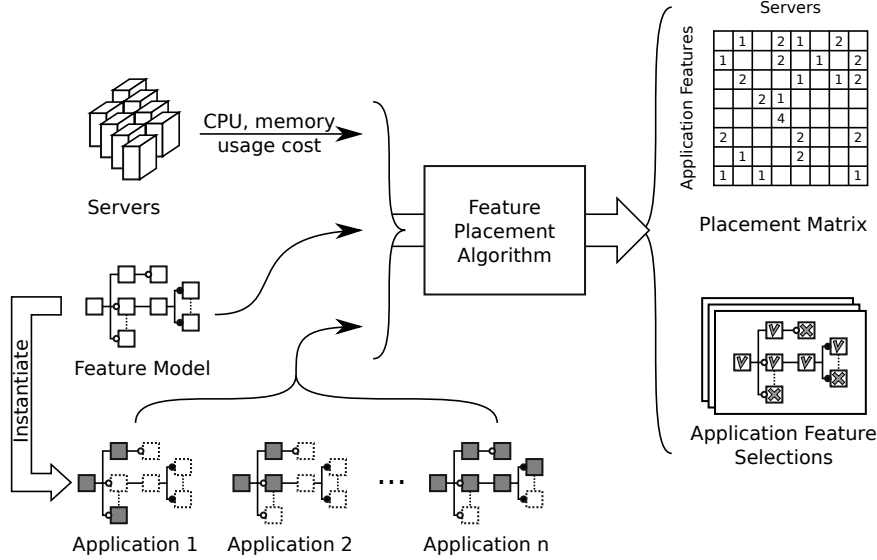


Figure 4.5: The input and output of the feature placement.

Figure 4.5 shows the inputs and outputs of feature placement. As an input, the model defines servers, with associated CPU, memory and usage cost information. Additionally, a feature model is defined containing all of the application features and their relations. Individual features, comparable to VMs in traditional application placement scenarios, can be instantiated on a server. Thus, every instance has specific memory requirements. Additionally, we limit the amount of processing that a single instance can do by limiting the amount of CPU resources that can be processed by a single instance. These instance limits are used as it is unrealistic for a VM with limited memory to be able to process an infinite amount of requests (represented by their CPU use). Finally, applications are composed out of multiple features. They are represented by a set of selected features, and a set of excluded features. Some features may be neither selected nor excluded for an application, leaving these features as open variation points. During the execution of the feature placement algorithm, these open variation points are filled in by either selecting or excluding them based on the feature model. The above results in three sets of input variables: servers, features and applications. These are respectively contained in the sets S , F and A .

The model has two outputs: a placement matrix that defines where features are instantiated and for which applications they are used, and the application feature selection which indicates which features are used for which applications, filling in open variation points. The placement matrix itself consists of two separate parts. First, it describes on which servers features are instantiated. As the amount of CPU

resources that can be used by a single feature instance is limited, it is possible for multiple instances of a feature to exist on a server. Secondly, the placement matrix also describes which applications make use of which feature instances and how much CPU capacity of a feature instance is used for every application. The above results in three output variables:

- The instance count, $IC_{s,f}$, represents the number of instances of a feature $f \in F$ that are instantiated on a server $s \in S$ in the solution.
- The placement matrix, $M_{s,f,a}^{CPU}$, shows how much CPU resources are allocated on a given server $s \in S$ for a feature $f \in F$ of an application $a \in A$.
- The binary variables $\Phi_{f,a}$ are used to represent the feature selections: $\Phi_{f,a} = 1$ if the feature $f \in F$ is included for application $a \in A$. Conversely, feature is excluded if $\Phi_{f,a} = 0$.

The objective of the feature placement optimization is to minimize the cost of the placement. This cost consists of two separate components: the cost of using servers and the cost of failing to place applications or their components. The cost of using a server represents an operational cost that can either be an energy cost or the cost of renting a server. The failure cost represents a Service Level Agreement (SLA) cost of failing to place an application correctly. Two separate failure costs are taken into account: an application failure cost is incurred whenever any feature of an application fails to be placed correctly, while the feature failure cost is incurred when a single feature of an application is not placed correctly. These separate costs make it possible to define an additional cost for failing to provide specific essential features of the applications. These costs may be represented as a monetary cost, but also as a virtual cost determined by the management system that varies during the execution of the management system. E.g. failing to place an application may in reality result in a cost of 0 if the service interruption is short but should still have an assigned cost within the management system to prevent it from happening at all.

In the previous discussion, we focused specifically on memory and CPU resources for simplicity. In practice, it is however possible for there to be additional relevant resource types, such as disk space and bandwidth. The model supports this by making an abstraction of the concept of resource, and making a distinction between two resource types:

- *Strict resource* types are resources that are required to create an instance of the feature. The typical example of this resource type when studying resource allocation problems is application memory. Another possible example is disk space. To create a VM, this memory and disk demand must be satisfied; otherwise it is impossible to create an instance of the feature. Every server

has a specified amount of available strict resources, and every feature instance has a specific strict resource demand.

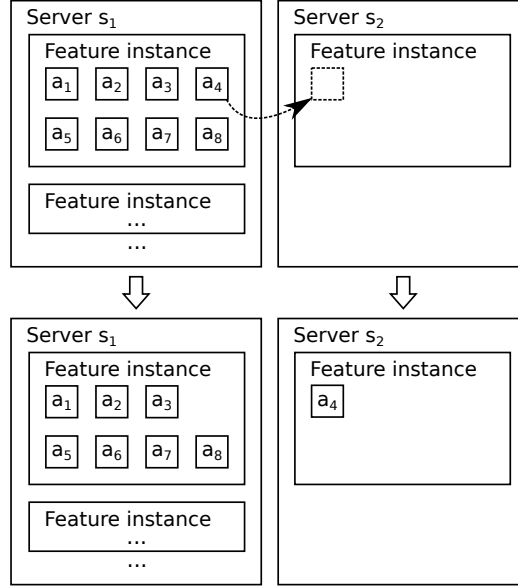
- *Non-strict resource* types behave differently. The goal of the optimization is to ensure that all of the requested resources of this type are allocated for applications. To achieve this, instances are created and that are responsible for providing (part of) the resource demand of an application. The common example here is CPU capacity: a certain amount of CPU capacity must be allocated to an application to ensure all requests for the application can be handled, but not all of the calls must always be handled on the same instance, especially for larger tenants for whom the application must be distributed over multiple nodes anyway because of their scale. While every server has a specified available non-strict resource demand, feature instances do not have a non-strict resource demand. Instead, applications have a specific non-strict resource demand that must be handled by one or more feature instances. As stated previously, it is possible for feature instances to be limited in the amount of non-strict resources that they can process, which is represented using instance non-strict resource limits.

All of the resources are contained within the set Γ , strict resources are contained in the set Γ_s , while non-strict resources are contained in the set $\Gamma_{\bar{s}}$. Generalizing the output variables, we obtain a placement matrix $M_{s,f,a}^\gamma$, where γ is any of the non-strict variables in $\Gamma_{\bar{s}}$.

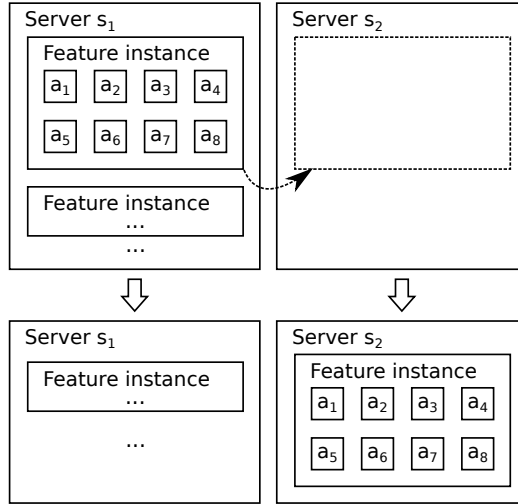
4.4.1 Dynamic feature placement: resource migrations

The static problem model as summarized above can be used to minimize the cost of a placement, in terms of both failed placements and server use, but it does not take the current application allocation into account. By adding the current resource allocation as an input, the number of resource migrations that are required to apply the placement can be determined. Based on how the model is defined, there are two types of resource migrations that should be considered. On one hand, the number of instance count changes can be determined. This represents the number of virtual machines that must be started and stopped. We refer to this migration type as an *instance count change*. Due to the multi-tenant nature of the instances, a second measure for migrations can be considered as well: the amount of resources used for applications that are allocated using other instances. This type of migration, which we refer to as *resource shift*, can also cause network traffic as application data present in one instance may have to be transferred to another instance. An illustration of instance count change and resource shift is shown in Figure 4.6.

To model the migrations caused by a placement, the current allocation must be added as an input to the model. This current allocation can be represented using two additional inputs:



(a) An example of a resource shift migration. No new instances are created, but resources that were allocated for an application a_4 within a feature instance on server s_1 are moved to another feature instance on server s_2 .



(b) An example of an instance count change migration. An instance is removed on server s_1 and a new instance is created on server s_2 . (Note that in this example the resources allocated for the applications a_1 to a_8 are shifted as well.)

Figure 4.6: An illustration of the difference between resource shift migrations and instance count change migrations.

1. The previous instance count $IC'_{s,f}$, which indicates the number of instances of a feature f that are currently allocated on a server s . This input is required to determine the change in instance count on every instance, which can in turn be used to measure the number of instance count change migrations.
2. The previous resource allocation, represented as $M'^\gamma_{s,f,a}$ must be included as an input variable as well. By determining changes between M' and the currently computing allocation M , the number of resource shift migrations can be modeled.

The number of instance count increases IC_+ can be determined as shown in Equation (4.1). The increase can be determined by, for every server and feature type, calculating the difference in the number of feature instances between both allocations. We are only interested in feature instance count increases, as only creating a new feature instance incurs a cost, in the form of network load and delays. Therefore decreasing values (when $IC_{s,f} - IC'_{s,f} < 0$) are ignored and replaced by 0 within the sum.

$$IC_+ = \sum_{s \in S} \sum_{f \in F} \max(IC_{s,f} - IC'_{s,f}, 0) \quad (4.1)$$

To characterize resource shift migrations, we make use of a similar formulation. The difference in resource use between the current placement matrix M and the previous placement matrix M' are determined. Like in the definition of IC_+ negative values are removed, and only positive changes are counted. Equation (4.2) shows how this resource shift can be computed for non-strict resource types. As there may be migrations of resources of different types, this results in a value for every resource type. By normalizing the M_+^γ values based on the total resource load these different resource types can be combined into a single measure M_+ . This is shown in Equation (4.3). In this equation, T^γ is the total load for non strict resource γ in the previous iteration, which is used for the normalization.

$$M_+^\gamma = \sum_{s \in S} \sum_{f \in F} \sum_{a \in A} \max(M_{s,f,a}^\gamma - M'^\gamma_{s,f,a}, 0) \quad (4.2)$$

$$M_+ = \frac{1}{|\Gamma_{\bar{s}}|} \times \sum_{\gamma \in \Gamma_{\bar{s}}} \left(\frac{1}{1 + T^\gamma} \times M_+^\gamma \right) \quad (4.3)$$

These model extensions add IC_+ , which is a measure of instance count change migrations, and M_+ , which is a measure of resource shift migrations.

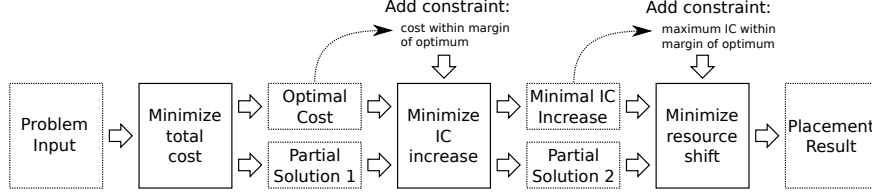


Figure 4.7: The different optimization steps of the migration minimizing model.

4.4.2 Iterative migration minimizing model

The migration minimizing model makes use of an iterative approach for minimizing the cost, instance count increase IC_+ and resource shift M_+ . Within the optimization, we define use of nearness parameter $\alpha \geq 1$, and ensure the cost of the solution will always be within a factor α of its optimal value. The migration minimizing model is illustrated in Figure 4.7 and consists of three steps:

1. First, the base model is optimized, minimizing the total cost. This results in an optimal cost C^* .
2. Subsequently, the instance count increase IC_+ is minimized. During this minimization, an additional constraint on the cost is added: $C \leq \alpha \times C^*$. This results in an optimal instance count increase value IC_+^* .
3. Finally, the resource shift migrations M_+ are minimized. During this optimization, two constraints are added, limiting both the cost and instance count increase migrations: $C \leq \alpha \times C^*$ and $IC_+ \leq \alpha \times IC_+^*$. This last optimization returns a placement result where the total cost C the number of instance count migrations IC_+ and the amount of resource shift migrations M_+ are all taken into account.

4.5 The Dynamic Feature Placement Algorithm

We designed the Dynamic Feature Placement Algorithm (DFPA) heuristic to solve the problem discussed in the previous section. To create a dynamic algorithm that minimizes the number of migrations between iterations, we design an algorithm that, using the current placement state and a specific change, results in a new application placement. These changes can either be the addition of an application, the removal of an application, or a change to the resource requirement of an application. The latter can be achieved by sequentially removing and re-adding the application with different resource requirements, which is why we focus specifically on application start and stop events. The management algorithm dynamically generates a new solution whenever applications are added or removed, and bases

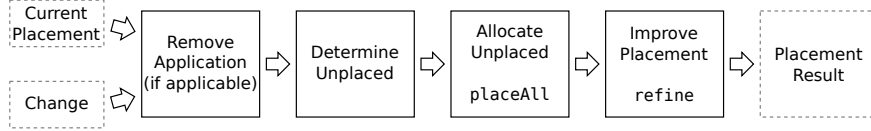


Figure 4.8: A high level overview of the Dynamic Feature Placement Algorithm (DFPA) steps. The functions `placeAll` and `refine` are discussed in Sections 4.5.1 and 4.5.2 respectively.

itself on the previous allocation to achieve good placement results with limited resource migrations.

The algorithm is invoked whenever an application is instantiated or halted. A high level overview of the algorithm steps is shown in Figure 4.8. The algorithm maintains a solution to the placement problem in-memory, which it updates during each of the steps. The initial state of the algorithm is the current placement, before the changes are taken into account. The main steps of the algorithm are as follows:

1. Two inputs are used by the DFPA: the current placement and a change to the current state. The current placement is represented using a placement matrix M and is used as the initial algorithm state. The provided change is either a start or a stop event for an application.
2. If the change is the removal of an application, the application is removed from the current solution, possibly reducing the number of feature instances for some of the application features if this causes feature instances to become unused.
3. Subsequently, the applications that must be placed are determined. This may be an application that must be instantiated now if the change is the addition of an application, but this list may also be larger if an application was not successfully placed in a previous algorithm iteration.
4. In a next step, the applications are placed iteratively. This is done using a `placeAll` function that is described in detail in Section 4.5.1.
5. Then, an improve operation is used to refine the placement and improve its quality. This is done by selectively removing and re-adding applications, reducing the number of servers used and making it possible to place some applications that would otherwise have failed. This solution refinement process, executed using a `refine` function, is discussed in Section 4.5.2.
6. Finally, the placement result that has been determined during the previous steps is returned as the placement result.

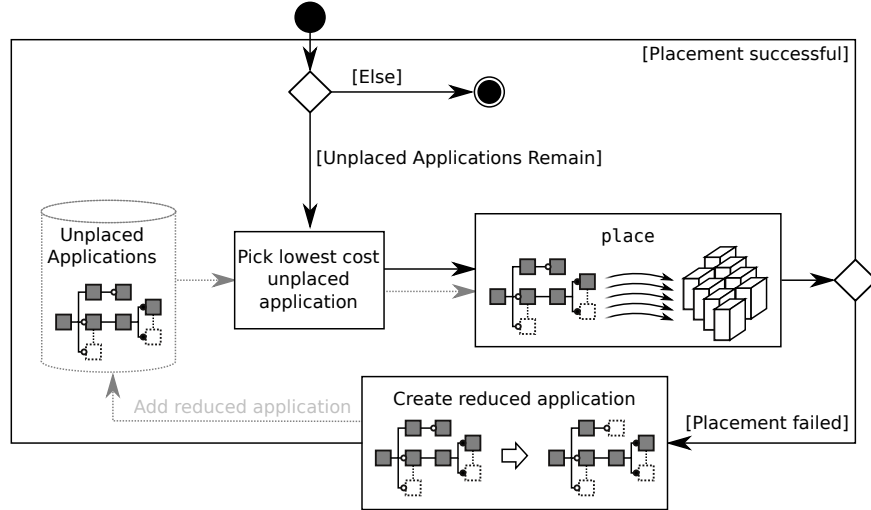


Figure 4.9: A high level overview of the `placeAll` function which is responsible for allocating all unplaced applications. To achieve this, the function iterates over all unplaced applications and allocates resources for them using the `place` function. If this succeeds, the next unplaced application is allocated. Otherwise, a reduced version of the application is created and added to the collection of unplaced applications. This application will then be placed in a later iteration of the algorithm.

4.5.1 Placing applications

The `placeAll` function, illustrated in Figure 4.9 and shown in Algorithm 4 is responsible for allocating resources for a collection of applications. This function iterates over all of the applications, and allocates resources one by one. The order by which applications are selected is defined by the total cost of failing to place the application, and can be computed by adding the cost of failing the application to the cost of failing of its individual features. Applications with a higher cost are placed first, which is done using a `place` function which will be explained later on. If it is impossible to place the application due to insufficient resources, a reduced version of the application is created: for this version of the application the cost of failing the application itself is 0 (as at this point it has already failed) and only the features that incur a separate failure cost are included. Alternatively, if the application failure cost is already 0, the reduced application is created by removing the feature with the lowest cost of failure. The reduced application is then re-added to the collection of applications that must be placed, ensuring critical application components will be allocated in a later iteration.

Using this approach, applications are either placed in their entirety, or if this is impossible, an effort is made to place a reduced version of the application. This ensures that important features, for which the cost of failure is high, will still be

included even if not all other features can be made available. This approach for allocating collections of applications is based on the application-based feature placement algorithm which we previously presented in [5].

4.5.1.1 The place function

The `place` function is illustrated in Figure 4.10 and shown in detail in Algorithm 5. First, the function determines the different possible feature configurations where the open variation points are filled in. Then, the algorithm evaluates the possible configurations, using a `chooseBestFeatureSelection` function to select the best configuration. While doing so, it takes into account the current allocation, minimizing the number of new instances needed and maximizing the use of currently existing instances. Once the features that are chosen have been determined, the resources that must be allocated can be determined. This resource demand is then allocated on the existing feature instances using a `placeResidualCapacity` function. If, after allocating resources on existing instances, not all of the resource demand is handled, new feature instances are created using a `doCreateInstance` function. These last two steps are repeated until all demand has been allocated.

In the first step of the `place` function, it determines all of the different possible alternative feature allocations. This is done by at first selecting all of the features that are logically implied by the current feature selection (e.g. by adding parent features of selected features to the collection of parent features) and pruning optional features that are not implied as being included by adding them to the set of excluded features. If at this point there are still features that are neither included nor excluded, new configurations are generated where, in every new configuration, one of the undecided features is added to the selected set. This process is repeated until only

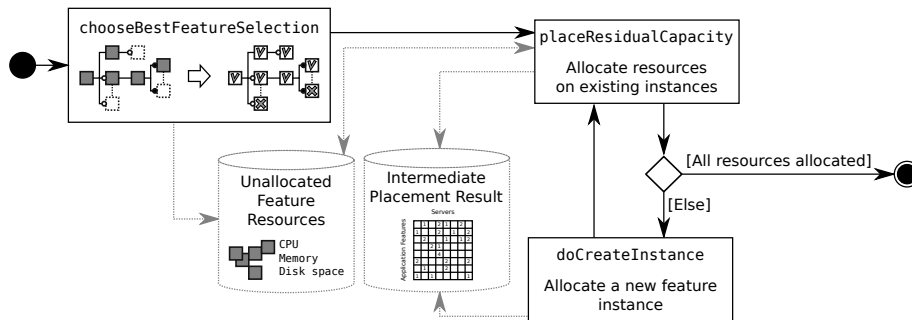


Figure 4.10: A high level overview of the `place` function which is responsible for allocating a single application. The function first determines a feature selection for the application, resulting in a collection of application features for which resources must be allocated. Resources are then allocated using existing feature instances. When there is no capacity left on existing feature instances, new instances are created.

Data: The current allocation matrix M
Data: A collection of applications $toPlace$ for which resources must be allocated

```

while  $toPlace$  not empty do
   $a \leftarrow$  application with highest cost;
   $toPlace \leftarrow toPlace \setminus \{a\}$ ;
   $M' \leftarrow \text{place}(a, M)$ ;
  if placement of  $a$  unsuccessful then
     $a' \leftarrow$  create a reduced version of application  $a$ ;
    if  $a'$  still contains features with failure cost then
       $toPlace \leftarrow toPlace \cup \{a'\}$ 
    end
  end
end
return  $M$ ;

```

Algorithm 4: The placeAll function.

Data: The current allocation matrix M
Data: An application a for which resources must be allocated
 Determine alternative feasible feature configurations $\phi(a)$;
 $F \leftarrow \text{chooseBestFeatureSelection}(M, \phi(a))$;
 $d \leftarrow$ resources required for features F ;

```

while  $d$  not empty do
   $(M, d) \leftarrow \text{placeResidualCapacity}(M, d)$ ;
  if  $d$  not empty then
     $(M, d) \leftarrow \text{doCreateInstance}(M, d)$ ;
  end
end
return  $M$ ;

```

Algorithm 5: The place function.

feature selections where every feature is either selected or excluded are left³. This results in a collection $\phi(a)$ containing the possible feature configurations of a .

4.5.1.2 Choosing the best feature selection

Next, the `chooseBestFeatureSelection` function is used to determine the best selection of features given the selected application features and the current allocation. In this function, the alternative allocations in $\phi(a)$ are compared using two criteria: the strict resource increase (*SRI*), and the total resource requirement (*TR*). The *SRI* represents the amount *additional* strict resources that are needed to instantiate this application, thus not counting already existing application instances. The *TR* measure computes the total resource demand of a configuration, showing its general resource requirement.

A low *SRI* implies that the cost of allocating the specific feature selection is low, taking into account the current placement, as few additional instances are needed. If the feature selection can be instantiated entirely on existing instances, the *SRI* value will be zero. A low *TR* on the other hand implies that the cost of the selection is low in general. When instantiating new applications, we prioritize a low *SRI* value, and *TR* values are only minimized when *SRI* values of two configurations are equal.

The *SRI* of a feature selection represents the amount of strict resources that must be added to instantiate a specific feature selection taking into account the current placement state. This value is determined in three steps:

1. First, determine how many of the non-strict resources required for this allocation can be allocated on existing instances.
2. Based on this, determine the number of new feature instances that can be allocated to place the application.
3. Finally, the amount of strict resources needed to create these instances can be calculated. The resulting value is the *SRI* for the analyzed application feature configuration.

Equation (4.4) shows how the *TR* value for an application is computed. This value is composed out both non-strict resource demands $TR_{\bar{s}}$ and strict resource demands TR_s . As strict resource demands are more constraining than non-strict demands, the impact of $TR_{\bar{s}}$ is divided by 2, making it impact the total *TR* value less than the TR_s value. Non-strict resource demand is calculated by directly measuring the resource demand of an application feature configuration. Strict resource demand is calculated by determining the number of instances required for

³Note that this process is only dependent on the application and feature model. Thus, these alternatives can be computed once when the application is added and do not have to be computed every iteration.

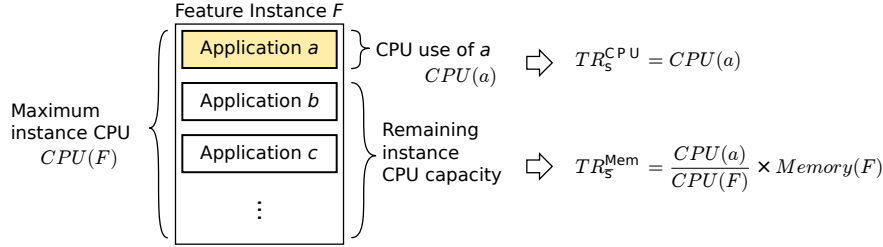


Figure 4.11: An illustration of how TR values can be determined for an application a using CPU and memory resources. TR_s^{CPU} can be calculated directly based on the application demand. The memory requirement, TR_s^{Mem} is calculated indirectly as the feature instance's memory demand is shared between the different applications using the instance. The ratio of CPU resources used compared to the total available resources is used to determine the share of the total memory that is dedicated to the application.

the configuration. Both TR_s and $TR_{\bar{s}}$ values are normalized by dividing them by the maximum value for any application.

$$TR = \sum_{\gamma \in \Gamma_s} \frac{TR_s(\gamma)}{\max_{\gamma} TR_s(\gamma)} + 0.5 \times \sum_{\gamma \in \Gamma_{\bar{s}}} \frac{TR_{\bar{s}}(\gamma)}{\max_{\gamma} TR_{\bar{s}}(\gamma)} \quad (4.4)$$

In Figure 4.11 an illustrative example is shown of how the TR_s and $TR_{\bar{s}}$ values can be computed for CPU and memory resources. More formally, $TR_{\bar{s}}$, as defined in Equation (4.5), can be computed in a straightforward way: the total resource demand is the sum of the demand for the individual selected features (represented by the variable $required(f, \gamma)$). The TR_s value of a feature selection is computed, as shown in Equation (4.6), by taking into account both the amount of resources needed for its instances and the share of the instance that is used for this specific configuration. This share, for a given feature f , is computed by determining the total amount of non-strict resources that are required for the feature, and comparing this value to the amount of these resources that can be provided by a single feature instance. This is expressed in Equation (4.7).

$$TR_{\bar{s}}(\gamma) = \sum_{f \in \text{selected}} required(f, \gamma) \quad (4.5)$$

$$TR_s(\gamma) = \sum_{f \in \text{selected}} share(f) \times IR_f^\gamma \quad (4.6)$$

$$share(f) = \max_{\gamma \in \Gamma_{\bar{s}}} \frac{required(f, \gamma)}{L_f^\gamma} \quad (4.7)$$

As mentioned previously, the feature configuration with the lowest SRI value is chosen. If, for two configurations the SRI metrics are equal, the alternative with the lowest TR value is preferred.

4.5.1.3 Allocating resources

Once a feature configuration has been selected, the `placeResidualCapacity` function, shown in Algorithm 6, is used to allocate as much of the demand as possible on existing feature instances. This is done by iterating over every feature that must still be allocated and every feature instance with remaining capacity that is currently allocated.

Data: The current allocation matrix M
Data: The demand d that is not yet allocated
Data: The application a for which the demand must be allocated
for every feature f in d do
 for every feature instance of i in M do
 $s \leftarrow$ the server on which i is running;
 $fCap \leftarrow$ remaining resource capacity of feature f ;
 $sCap \leftarrow$ remaining resource capacity of server s ;
 for $r \in \Gamma_s$ do
 $toAssign \leftarrow \min(fCap(r), sCap(r), d(f, r))$;
 `allocate`($a, f, s, r, toAssign$);
 Update the remaining unallocated demand d ;
 end
 end
end
return (M, d);

Algorithm 6: The `placeResidualCapacity` function allocates the remaining demand d on currently existing feature instances.

If not all of the demand is provided after allocating resources on existing instances, additional feature instances must be instantiated. To achieve this, the `doCreateInstance` function, shown in Algorithm 7, is used. This function first determines the features with unassigned capacity, for which new feature instances must be instantiated. The algorithm then iterates over these features, allocating resources for them sequentially. For every feature f that must be instantiated, the number of times the feature must be instantiated, N , is determined using the remaining resource demand d . The servers within the datacenter that are capable of running the service and have sufficient free resources are then filtered and put in the set S' . The best server in S' for instantiating the feature is then selected, and as many instances as needed and possible are instantiated on the server. This process is repeated until either N new instances have been created, or until no servers are left to allocate instances on. If not all required instances can be created, it is impossible to fully allocate the desired applications. If this happens, the current `place` operation is aborted, and a new feature selection is determined as previously discussed in the description of the `placeAll` function.

Data: The current allocation matrix M
Data: The demand d that is not yet allocated
Data: The application a for which the demand must be allocated
 $F_{unassigned} \leftarrow$ features with remaining unassigned capacity;
for every feature $f \in F_{unassigned}$ **do**
 $N \leftarrow$ determine number of features needed;
 $S' \leftarrow \{s \mid s \in S \wedge s \text{ has sufficient free resources}\};$
 while $N > 0$ **and** $S' \neq \{\}$ **do**
 $s \leftarrow \text{selectBestServer}(S', f);$
 $N^s \leftarrow$ determine maximum possible number of instances of f on s ;
 $\text{createInstance}(M, f, s, \min(N, N^s));$
 $N \leftarrow \min(N, N^s);$
 end
 if $N > 0$ **then**
 Abort: Insufficient resources, allocation is impossible;
 end
end
return M ;

Algorithm 7: The `doCreateInstance` function creates additional feature instances for the demand d that is not yet allocated and allocates these resources.

4.5.1.4 Server selection

The `doCreateInstance` function compares different servers and makes use of the `selectBestServer(S', f)` function to determine the best server in S' to create instances of f on. This function selects the server s with the highest quality $Q(s)$. To ensure that the algorithm avoids using additional servers, the quality of an unused server is defined as 0. For other servers, this quality is composed of three factors: 1) the quality of remaining resources $QR(s)$ characterizes the desirability of using a server based on the remaining resources on the server; 2) the quality of the fit $QF(s)$ determines whether it is desirable to instantiate the feature on the server by determining the amount of resources that would be remaining on the server afterwards; finally 3) a bonus is added through a binary server use variable $SU(s)$ if an instance already physically exists. Equation (4.8) shows how this server quality can be computed by combining these three factors with different weights.

$$Q(s) = \begin{cases} 0.3 \times QR(s) + 0.5 \times QF(s) + 0.2 \times SU(s) & \text{if } s \text{ used} \\ 0 & \text{otherwise} \end{cases} \quad (4.8)$$

It is preferred to use servers with little remaining resources if possible, as this ensures other servers, with more remaining resources may be used for later, more complicated tasks. This is expressed, for a server, using the quality of remaining resources $QR(s)$ metric. This metric is shown in Equation (4.9), and represents the

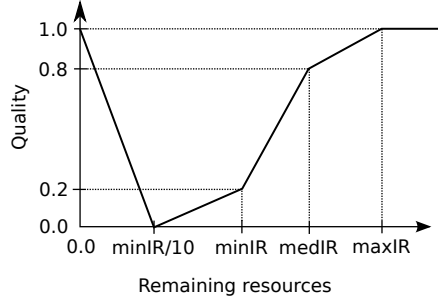


Figure 4.12: The piecewise linear function used to determine the quality of a fit. minIR is the minimum resource requirement of an instance, medIR is the median resource requirement and maxIR is the maximum resource requirement. If there are more remaining resources than maxIR , any other feature can be instantiated on the server, indicating the fit is good. When there are nearly no remaining resources (less than $\frac{\text{minIR}}{10}$), the fit is good as well, as in this scenario few resources are wasted.

desirability of using a server regardless of the instance type that is to be instantiated on it. Within this equation, $\text{remaining}(\gamma, s)$ refers to the amount of resources γ of s that are currently remaining on the server, while Ra_s^γ shows the amount of available resources on the server s .

$$QR(s) = 1 - \min_{\gamma \in \Gamma_s} \frac{\text{remaining}(\gamma, s)}{Ra_s^\gamma} \quad (4.9)$$

$QF(s)$ characterizes the quality of the fit of the chosen instance on the server, and represents the quality of remaining resources *after* an instance is allocated. This remaining capacity should either be enough to support new instances or alternatively, it should be very low ensuring few resources are wasted. This value is determined by, for every strict resource type, determining a separate $QF^\gamma(s)$ value which in turn is determined by the amount of resources left after allocating the feature on the server. To these remaining resources, a piecewise linear function, shown in Figure 4.12, is applied. If after the allocation any other application instance may be instantiated, a high $QF^\gamma(s)$ will be achieved. If after the allocation none of the other application instances may be instantiated, the remaining server capacity will be wasted, so a low $QF^\gamma(s)$ value is achieved, unless the amount of residual resources becomes very low which again indicates a good fit. The final $QF(s)$ value, defined in Equation (4.10), is the minimum of all $QF^\gamma(s)$ values.

$$QF(s) = \min_{\gamma \in \Gamma_s} QF^\gamma(s) \quad (4.10)$$

The $SU(s)$ variable is added to prevent instance migrations. During the execution of the DFPA algorithm multiple feature instances may be removed and added. $SU(s) = 1$ if in the original problem model there was already an instance of the

feature on the server, otherwise $SU(s) = 0$. This makes it more likely to select a server on which an instance of the feature already exists, even though this instance may have already been removed previously by the *remove application* step of the algorithm.

4.5.2 Refining placements

After a change to the placement has been made, there are multiple ways in which the placement may be improved. First, it is possible to remove some applications to free resources on a server with low utilization, making it possible to turn it off, and to reallocate the removed applications elsewhere. A second possible improvement can be determined by reconsidering the placement of applications that have a relatively expensive feature configuration. These expensive configurations can occur as the placement algorithm chooses a feature configuration that requires the least new feature instances, even if it requires slightly more resources. After adding and removing other applications, this configuration may however no longer be ideal, so it may be beneficial to re-place this application at a lower cost.

The placement refinement operation starts with an allocation M , and generates a new allocation M' by removing and re-adding a collection of applications \hat{A} . Subsequently, the quality of M' is compared to the quality of M . If the quality of M' is better than that of M (i.e. the placement has a lower cost), this refined placement is returned. Otherwise, the original placement M is returned. The *refine* function is designed to reconsider the placement of a limited collection of applications by ensuring it only moves about as many applications as one server can handle. The number of migrations is further limited by ensuring the refined placement is only enforced if it improves the resulting allocation.

The *refine* function is entirely defined by how the collection of applications \hat{A} is determined. Applications within this set are chosen in two ways: by determining applications that are running on underutilized servers and by determining applications of which the current allocation requires a high amount of resources compared to its minimum resource use. The final application set is determined in three steps:

1. The collection of underutilized servers \hat{S} is determined. This is a subset of all servers in S with a utilization higher than 0% and lower than 70% for all resource types. The collection of applications of which features are allocated on a server $s \in \hat{S}$ is represented by $app^{on}(s)$.
2. The currently allocated applications are evaluated based on the cost feature configuration used to allocate them. For every application $a \in A$, the relative cost of their resource configuration can be computed by comparing their minimum and maximum resource demand. This results in a scalar value where 0 represents an application allocated using minimal resources, and 1

represents the maximum possible resource demand, making the application use more resources than needed. A set of applications $A^{maxCost}$ containing the two highest cost applications is then created.

3. Using the $A^{maxCost}$ and \hat{S} collections the final application set \hat{A} is determined. This is done by, for every server $s \in \hat{S}$, building a collection of applications $app^{on}(s) \cup A^{maxCost}$. Out of all of these collections, the collection containing the fewest applications that results in the highest number of freed servers is chosen as \hat{A} .

4.6 Evaluation Setup

4.6.1 Evaluated algorithms

Based on the formal model discussed in Section 4.4, two algorithms can be designed using Integer Linear Programming (ILP). These algorithms were implemented using the CPLEX [33] solver, which is capable of optimally solving ILP problem formulations. The following algorithms were implemented:

- The ILP-based Feature Placement Algorithm (FPILP) is based on the formal model presented in the previous section without any considerations for the number of migrations. This algorithm yields an interesting baseline for the lowest possible cost. The results achieved by this algorithm may however not be practical as it does not take migration counts into account.
- The ILP-based Dynamic Feature Placement Algorithm (DFPILP) makes use of the iterative approach discussed in the previous section to determine a solution for the feature placement problem. As discussed, it first minimizes the cost, then the instance count increase, and finally the resource shift migrations.

We compare these ILP-based algorithms to the DFPA presented in the previous section.

4.6.2 Simulation parameters

For our evaluations we base ourselves on an application use case containing three applications. We evaluate the algorithms for two scenarios with a varying datacenter load. Within the scenario, we use applications defined by the feature model presented in [5], containing the features of three applications: document processing, medical communications and a medical information management application. In total, this model defines 49 different features. Within the evaluations, we make use of a uniform server configuration with a 3GHz CPU and 4GB of memory. Every

server is assigned an energy cost of 1. As previously mentioned in Section 4.4, this energy cost is not necessarily a direct cost but rather a cost defined by the management system.

Application feature configurations are generated at random by randomly selecting features to include and exclude. For every application, a random cost of failure is chosen within the set $\{2, 4, 8, 16, 32\}$, representing the idea that some applications may have a much higher cost of failure than other. The application demand determined based on the application features and configuration is multiplied with a variable that is randomly chosen using a uniform distribution in the interval $[0.1, 10]$, ensuring there is a variation in application load. Feature failure costs are determined by defining essential features associated with services that must continue functioning, even if the rest of the application fails. If the essential feature is included and correctly provisioned a minimal service is delivered to the end users. If not all of the features can be placed correctly, and placement of an application fails, this minimal service should still be provided. A failure cost out of the set $\{2, 4, 8, 16, 32, 64\}$ is assigned for all essential features. Like for the application fail costs, this value indicates that the costs of feature failure may vary greatly depending on client demands. The maximum feature failure cost is higher than the maximum application failure cost as failing this feature causes the interruption of critical services, while merely failing the application causes a service degradation.

A schedule is generated to determine when applications are started and stopped. To achieve this, a very large number of applications is generated and shuffled randomly. This ordered list of applications represents the order in which the applications are added. To ensure the datacenter does not become overloaded, as the total resource load of all of the generated applications greatly exceeds the total capacity of the cloud, we define a maximum datacenter resource load. If instantiating an application would result in exceeding the maximum datacenter load, an other application is first chosen to be removed. The resulting schedule will ensure that the datacenter is not overloaded, and will ensure that the application start events would always be accepted by an application request filter as described in Section 4.3. To choose the application to instantiate, first, the applications are stored as a list a_i where the order of applications is determined by the order in which they were instantiated. The index of the application to be removed is then determined using a Gaussian probability with $\mu = n$ and $\sigma^2 = n/2$, with n the number of applications that are instantiated. This ensures that very long running applications and very new applications have a smaller probability of being removed, which corresponds to client behavior: if an application has been running for a tenant for a very long time, it is a reliable client and there is a lower probability of him leaving, and if a tenant is very new, the application is often used for a period of time while its usefulness is determined and it is thus not stopped very quickly.

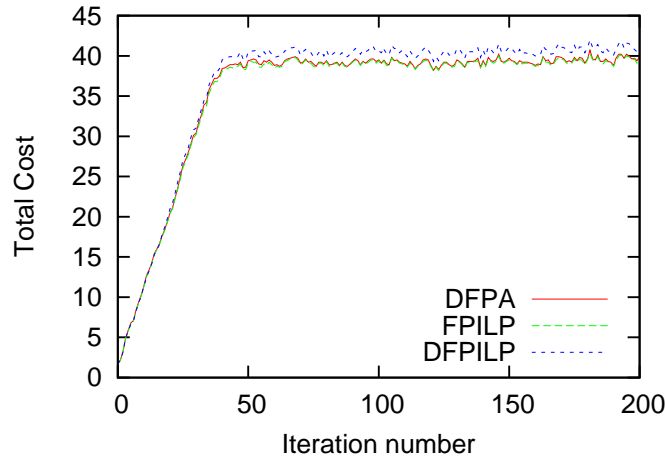


Figure 4.13: Total placement cost in an underloaded datacenter. Every iteration represents an application start or stop event. The quality achieved by the DFPA heuristic is the same as that achieved by the optimal FPILP algorithm.

The first scenario represents an underloaded datacenter, with a maximum load of 90% for all datacenter resources. This results in an “easy” placement with a low probability of failed placements occurring, which ensures the cost should be almost entirely caused by server use costs. To ensure the ILP-based algorithms can function, the number of server remains limited to 50. In a second scenario, a maximum load of 110% is chosen. In this overloaded datacenter scenario, placement becomes more complex, as more applications will fail to be placed. As this higher maximum load causes the number of applications that are active at any time to increase, the number of servers was reduced to 40 to ensure the ILP-based algorithms could compute solutions acceptably fast. For both scenarios 10 simulations were executed and the results were averaged.

4.7 Evaluation Results

4.7.1 Underloaded datacenter

Figure 4.13 shows the total cost throughout the simulation for the three solvers for the underloaded datacenter. In this scenario, there is no significant difference between the performance of the DFPA heuristic and the FPILP algorithm, which always results in the optimal total costs. The DFPILP algorithm, which is based on multiple ILP optimizations performs worst, as it tolerates a slight decrease in solution optimality to reduce the number of migrations. Averaging the cost between

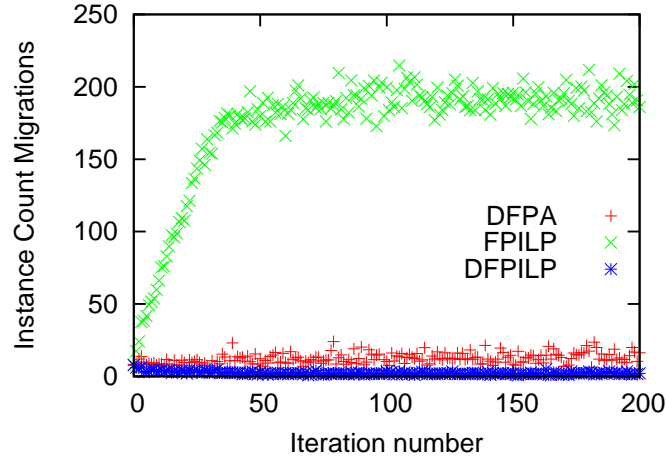


Figure 4.14: The number of instance migrations for every iteration of the dynamic placement algorithms in an underloaded datacenter. The number of IC migrations of DFPA is much lower than that of the FPILP algorithm which is unaware of migrations and slightly higher than that achieved by the DFPILP algorithm.

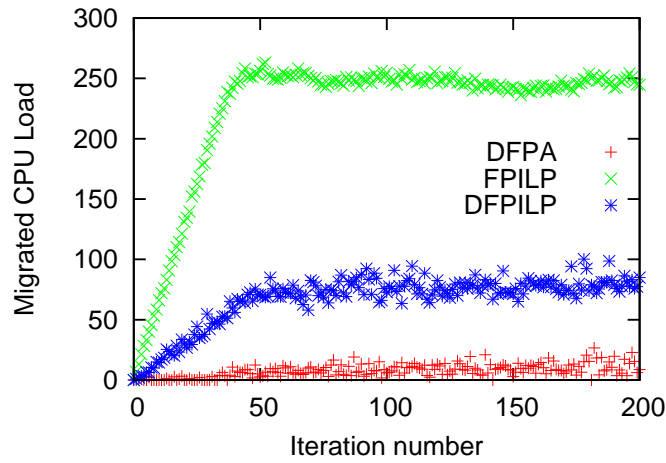


Figure 4.15: The amount of CPU resources that are moved between instances in subsequent algorithm invocations in an underloaded datacenter. The DFPA places applications while resulting in very few CPU load migrations, even when compared to the DFPILP algorithm.

iterations 50 and 200 (after the start up period of the evaluation), we observe that the DFPA heuristic on average only results in a cost that is 0.4% higher than the optimal cost.

Comparing the number of instance migrations, shown in Figure 4.14, we observe that the FPILP which does not take migrations into account results in very high numbers of migrations, making it unusable in dynamic scenarios where applications are started and halted at runtime. Both the DFPA heuristic and the DFILP algorithms result in fewer instance migrations. Due to the higher cost observed previously, the DFILP algorithm is capable of achieving fewer instance migrations than the DFPA heuristic. When the load shift is compared, shown in Figure 4.15, the FPILP algorithm again performs worst. Here, the DFPA heuristic outperforms the DFILP algorithm, migrating noticeably less resources between servers in consecutive algorithm invocations. Compared to the non-dynamic FPILP algorithm, the DFPA algorithm results in a 77.5% reduction in IC migrations and a 96.5% reduction in resource shift migrations.

From this, we conclude that the DFPA performs well in an underloaded datacenter scenario: it achieves a similar cost to the optimal FPILP, but results in much less load shift and instance count migrations. Compared to the iterative ILP-based DFILP algorithm it results in a lower total cost, and less resource shift at the cost of a slightly higher number of instance migrations.

4.7.2 Overloaded datacenter

The total cost of the algorithms in the overloaded datacenter scenario is shown in Figure 4.16. Generally, the performance of the DFPA heuristic remains similar to that of both the FPILP and DFILP algorithms, but there are multiple outliers where performance decreases and a higher cost occurs. Usually, these peaks are temporary and they decrease in the next iterations. Table 4.2 shows variation of the total cost throughout the evaluation data points (using all entries between iterations 50 and 200, thus taking only the data points into account where the datacenter is fully utilized). Based on this information we observe that in 95% of the algorithm invocations a cost similar to that of the ILP-based algorithms is achieved. In the remaining cases, a performance notably worse than the ILP-based algorithms is observed. On average, the DFPA heuristic on average only results in a cost that is 2.8% higher than the optimal cost. The results for instance count migrations and load shift are comparable to those shown for the underloaded datacenter as can be observed in Figures 4.17, and 4.18. Comparing the non-dynamic FPILP algorithm to the DFPA heuristic, we observe a 92.7% reduction in IC migrations and a 96.1% reduction in resource shift migrations.

Based on our evaluation, we observe that the DFPA algorithm achieves good results in an underloaded datacenter scenario, but performs less consistently in an

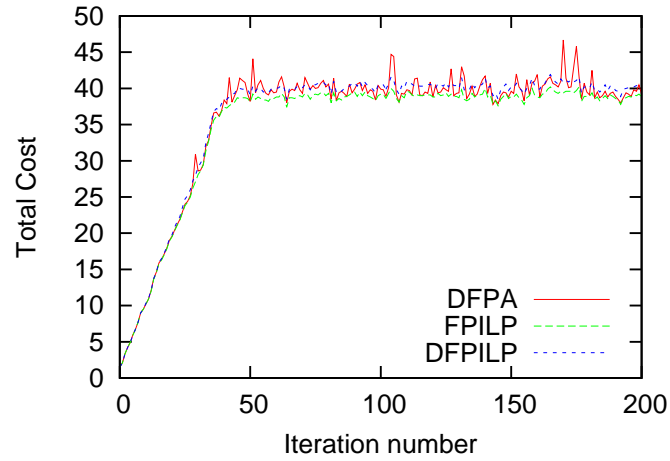


Figure 4.16: Total placement cost in an overloaded datacenter. The cost of the DFPA is less consistent than that of the ILP-based algorithms: at times the same quality of the FPILP algorithm is achieved, but often spikes in cost occur that are only solved in subsequent iterations.

Algorithm	50th pct	95th pct	98th pct	99th pct
DFPA	39	48	52	58
FPILP	39	42	42	43
DFPILP	40	44	44	45

Table 4.2: An analysis of the distribution of the total cost of the algorithms. The percentiles were determined using all of the entries between the 50th and 200th iteration.

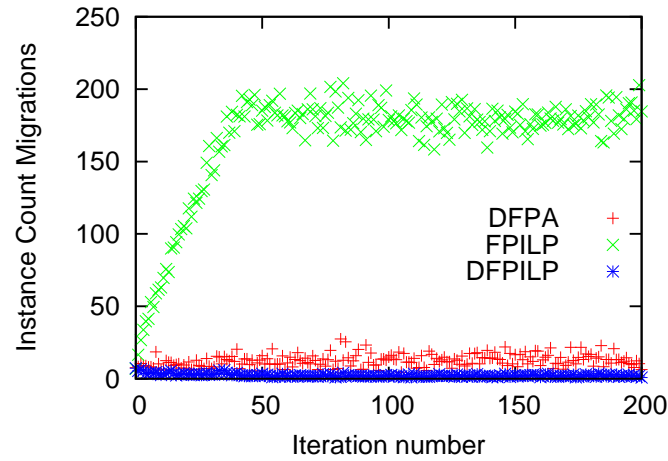


Figure 4.17: The number of instance migrations for every iteration of the dynamic placement algorithms in an overloaded datacenter. The DFPA results in slightly more IC migrations than the DFPIILP algorithm, and significantly less migrations than the FPILP algorithm.

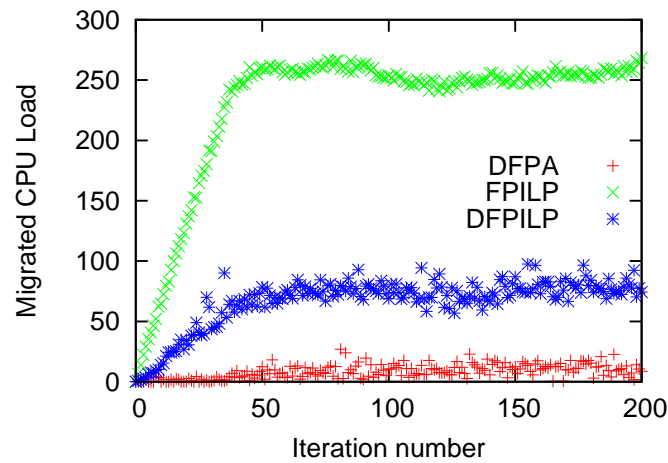


Figure 4.18: The amount of CPU resources that are moved between instances in subsequent algorithm invocations in an overloaded datacenter. Like in the underloaded datacenter scenario, the DFPA algorithm results in very few CPU load migrations.

overloaded datacenter scenario. In practice, it is however best to use an access filter that does not permit datacenter overload, as otherwise some application components are bound to fail, making the DFPA an interesting approach as it does not make use of ILP formulations, making it scale much better than the FPILP and DFILP algorithms.

4.8 Conclusions

A challenge in contemporary cloud platforms is that it is difficult to create and manage highly customizable applications while still achieving resource sharing through multi-tenancy. In this chapter we presented the concept of dynamic feature placement, an approach where customizable applications are composed using multiple interacting components and where individual application components can be shared between multiple applications and end users. The presented models and algorithms were designed to take into account dynamic scenarios where applications are started and stopped through time, taking migrations between the various steps into account.

We presented two new algorithms: the DFILP algorithm, an iterative ILP-based algorithm, and the DFPA heuristic. We analyzed the performance of the algorithms comparing them to a static optimal algorithm that is unaware of service migrations. In our evaluations, we found that adding migration-awareness to the management algorithms reduces the amount of instance migrations by more than 77% and reduces the load moved between instances by more than 96%. Despite this, the heuristic DFPA algorithm results in a cost that is on average less than 3% more than the optimal cost.

Acknowledgment

Hendrik Moens is funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT).

References

- [1] K. C. Kang, J. Lee, and P. Donohoe. *Feature-oriented product line engineering*. IEEE Software, 19(4):58–65, aug 2002. doi:10.1109/MS.2002.1020288.
- [2] R. E. Filman, T. Elrad, S. Clarke, and P. M. Aksit. *Aspect Oriented Software Development*. Addison-Wesley Professional, 2004.
- [3] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. *Developing and Managing Customizable Software as a Service Using Feature Model Conversion*. In Proceedings of the 3rd IEEE/IFIP Workshop on Cloud Management (CloudMan 2012), pages 1295–1302. IEEE, apr 2012. doi:10.1109/NOMS.2012.6212066.
- [4] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. D. Turck. *Feature Placement Algorithms for High-Variability Applications in Cloud Environments*. In Proceedings of the 13th Network Operations and Management Symposium (NOMS 2012), pages 17–24. IEEE, apr 2012. doi:10.1109/NOMS.2012.6211878.
- [5] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. *Cost-Effective Feature Placement of Customizable Multi-Tenant Applications in the Cloud*. Journal of Network and Systems Management, 22(4):517–558, oct 2014. doi:10.1007/s10922-013-9265-5.
- [6] K. Pohl, G. Böckle, and F. Van Der Linden. *Software product line engineering: foundations, principles, and techniques*. Springer-Verlag New York, Inc., 2005.
- [7] M. Hinchey, S. Park, and K. Schmid. *Building Dynamic Software Product Lines*. Computer, 45(10):22–26, oct 2012. doi:10.1109/MC.2012.332.
- [8] K. Zhang, X. Zhang, W. Sun, H. Liang, Y. Huang, L. Zeng, and X. Liu. *A Policy-Driven Approach for Software-as-Services Customization*. In Proceedings of the 9th IEEE International Conference on E-Commerce and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, (CEC/EEE 2007), pages 123–130. IEEE, jul 2007. doi:10.1109/CEC-EEE.2007.9.
- [9] W. Sun, X. Zhang, C. J. Guo, P. Sun, and H. Su. *Software as a Service: Configuration and Customization Perspectives*. In IEEE Congress on Services Part II (services-2), pages 18–24. IEEE, sep 2008. doi:10.1109/SERVICES-2.2008.29.

- [10] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl. *Variability modeling to support customization and deployment of multi-tenant-aware Software as a Service applications*. In Proceedings of the ICSE Workshop on Principles of Engineering Service Oriented Systems (PESOS 2009), pages 18–25. IEEE, may 2009. doi:10.1109/PESOS.2009.5068815.
- [11] M. Abu-Matar and H. Gomaa. *Feature Based Variability for Service Oriented Architectures*. In Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture (WICSA 2011), pages 302–309. IEEE, jun 2011. doi:10.1109/WICSA.2011.47.
- [12] M. Abu-Matar and H. Gomaa. *Variability Modeling for Service Oriented Product Line Architectures*. In Proceedings of the 15th International Software Product Line Conference (SPLC 2011), pages 110–119. ACM, aug 2011. doi:10.1109/SPLC.2011.26.
- [13] S. T. Ruehl and U. Andelfinger. *Applying Software Product Lines to create Customizable Software-as-a-Service Applications*. In Proceedings of the 15th International Software Product Line Conference (SPLC 2011), pages 16:1–16:4. ACM, aug 2011. doi:10.1145/2019136.2019154.
- [14] G. H. Alférez and V. Pelechano. *Context-Aware Autonomous Web Services in Software Product Lines*. In Proceedings of the 15th International Software Product Line Conference (SPLC 2011), pages 100–109. ACM, aug 2011. doi:10.1109/SPLC.2011.21.
- [15] B. Jennings and R. Stadler. *Resource Management in Clouds: Survey and Research Challenges*. Journal of Network and Systems Management, pages 1–53, mar 2014. doi:10.1007/s10922-014-9307-7.
- [16] J. Rolia, A. Andrzejak, and M. Arlitt. *Automating Enterprise Application Placement in Resource Utilities*. In M. Brunner and A. Keller, editors, Self-Managing Distributed Systems, volume 2867 of *Lecture Notes in Computer Science*, pages 118–129. Springer Berlin Heidelberg, 2003. doi:10.1007/978-3-540-39671-0_11.
- [17] B. Urgaonkar, A. L. Rosenberg, and P. Shenoy. *Application Placement on a Cluster of Servers*. International Journal of Foundations of Computer Science, 18(05):1023–1041, oct 2007. doi:10.1142/S012905410700511X.
- [18] C. Adam and R. Stadler. *Service Middleware for Self-Managing Large-Scale Systems*. IEEE Transactions on Network and Service Management, 4(3):50–64, dec 2007. doi:10.1109/TNSM.2007.021103.

- [19] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici. *A scalable application placement controller for enterprise data centers*. In Proceedings of the 16th International Conference on World Wide Web (WWW 2007), pages 331–340. ACM, may 2007. doi:10.1145/1242572.1242618.
- [20] X. Zhu, C. Santos, D. Beyer, J. Ward, and S. Singhal. *Automated application component placement in data centers using mathematical programming*. International Journal of Network Management, 18(6):467–483, nov 2008. doi:10.1002/nem.707.
- [21] D. Breitgand and A. Epstein. *SLA-aware Placement of Multi-Virtual Machine Elastic Services in Compute Clouds*. In Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011), pages 161–168. IEEE, may 2011. doi:10.1109/INM.2011.5990687.
- [22] D. Jayasinghe, C. Pu, T. Eilam, M. Steinder, I. Whally, and E. Snible. *Improving Performance and Availability of Services Hosted on IaaS Clouds with Structural Constraint-Aware Virtual Machine Placement*. In Proceedings of the 2011 IEEE International Conference on Services Computing, pages 72–79. IEEE, jul 2011. doi:10.1109/SCC.2011.28.
- [23] L. Shi, B. Butler, D. Botvich, and B. Jennings. *Provisioning of requests for virtual machine sets with placement constraints in IaaS clouds*. In Proceedings of the 13th IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), pages 499–505. IEEE, may 2013.
- [24] G. Foster, G. Keller, M. Tighe, H. Lutfiyya, and M. Bauer. *The right tool for the job: Switching data centre management strategies at runtime*. In Proceedings of the 13th IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), pages 151–159. IEEE, may 2013.
- [25] J. Xu and J. a. B. Fortes. *Multi-Objective Virtual Machine Placement in Virtualized Data Center Environments*. In Proceedings of the 2010 IEEE/ACM International Conference on Green Computing and Communications & International Conference on Cyber, Physical and Social Computing, pages 179–188. IEEE, dec 2010. doi:10.1109/GreenCom-CPSCCom.2010.137.
- [26] D. Carrera, M. Steinder, I. Whalley, J. Torres, and E. Ayguadé. *Utility-based placement of dynamic web applications with fairness goals*. In Proceedings of the 11th Network Operations and Management Symposium (NOMS 2008), pages 9–16. IEEE, apr 2008. doi:10.1109/NOMS.2008.4575111.
- [27] F. Wuhib, R. Stadler, and M. Spreitzer. *Gossip-based Resource Management for Cloud Environments*. In Proceedings of the 6th International Conference

- on Network and Service Management (CNSM 2010), pages 1–8. IEEE, oct 2010. doi:10.1109/CNSM.2010.5691347.
- [28] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. *Dynamic placement for clustered web applications*. In Proceedings of the 15th International Conference on World Wide Web (WWW 2006), pages 595–604. ACM, may 2006. doi:10.1145/1135777.1135865.
- [29] C. Low. *Decentralised Application Placement*. Future Generation Computer Systems, 21(2):281–290, feb 2005. doi:10.1016/j.future.2003.10.003.
- [30] T. Kimbrel, M. Steinder, M. Sviridenko, and A. Tantawi. *Dynamic Application Placement Under Service and Memory Constraints*. In S. Nikolettseas, editor, Experimental and Efficient Algorithms, volume 3503 of *Lecture Notes in Computer Science*, pages 391–402. Springer Berlin Heidelberg, 2005. doi:10.1007/11427186_34.
- [31] pure-systems GmbH. *pure::variants User’s Guide* [online]. 2012. Last accessed: December 2012. Available from: <http://www.pure-systems.com/Documentation.116.0.html>.
- [32] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. *Network-Aware Impact Determination Algorithms for Service Workflow Deployment in Hybrid Clouds*. In Proceedings of the 8th International Conference on Network and Service Management (CNSM 2012), pages 28–36. IEEE, oct 2012.
- [33] *IBM ILOG CPLEX 12.4* [online]. 2014. Available from: <http://www-01.ibm.com/software/integration/optimization/cplex-optimizer>.

5

Developing and Managing Customizable Software as a Service Using Feature Model Conversion

H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt and F. De Turck

Published in proceedings of the 3rd IEEE/IFIP Workshop on Cloud Management (CloudMan 2012), co-located with IEEE/IFIP NOMS 2012

The Feature-Based Binary (FBB) approach introduced in Chapter 2, and used in Chapters 3 and 4 to build feature placement algorithms focuses on how multi-tenant SaaS is managed at runtime. This chapter focuses on how FBB applications can be designed, and presents an overarching management architecture consisting of both a development and execution platform. We analyze how feature models can be used during application development in addition to their runtime use which has been elaborated on in previous chapters. To this end, we introduce the concept of a development feature model, which is designed during application development, and analyze how this feature model can be converted into a runtime feature model. This approach makes it possible to use feature modeling throughout the different stages of the application life-cycle: development, customization and deployment. We specifically focus on how development feature models can be automatically converted into runtime feature models ensuring a one-to-one correspondence between features and services exists.

5.1 Introduction

While the adoption of cloud technologies is on the rise, limitations prevent their uptake in some markets. Current cloud frameworks do not support the creation of highly customizable Software-as-a-Service (SaaS) applications that retain the multi-tenant nature of these applications, where a single application instance is shared between users. This is especially difficult if both functional and Quality of Service (QoS) variation are required. An approach where changes are statically compiled into the application could be used to run such applications on a cloud, but this would lead to the generation of different applications for clients, losing the cost-advantages offered by multi-tenancy. Furthermore, the cloud management software would still need extensive support for QoS management to ensure varying non-functional application requirements are met.

To develop customizable applications, Software Product Line Engineering (SPLE) techniques are often used. The software is modeled as a collection of functionalities, which are referred to as features. An application can then be created by combining these features. As it is possible for some features to require other features, or for some features to conflict with others, relations between features must be modeled as well. Feature modeling can become an integral part of the software development process, where code modules, configurations and Aspect-Oriented Programming (AOP) [1] aspects can be created that realize specific features. These feature models can then be used to communicate and configure variability in applications for clients [2]. They can also be used in the deployment process [3], using a Service-Oriented Architecture (SOA) approach where individual features map to services, and in the management and provisioning of these services, using a feature placement algorithm [4] that takes feature relations into account to determine where services implementing these features are placed.

Feature models are suited for many different phases in the life cycle of customizable applications, but maintaining a single feature model that can be used in each of the different phases can be complex, or even impossible. In particular, the representation of information in feature models, as they are specified during application development are not suited as an input for resource allocation algorithms, such as feature placement algorithms. This restriction either forces developers to work using cumbersome restrictions, increases development complexity as a conversion between a development model and a runtime model must be determined, or complicates feature-aware management of multi-tenant applications.

We present an approach where a logical feature model is defined and configured during development using existing SPLE techniques. When the service is deployed, a runtime version of the feature model is generated, together with a mapping between the development feature model and this runtime feature model. Services can then be allocated with the feature placement techniques we described in [4],

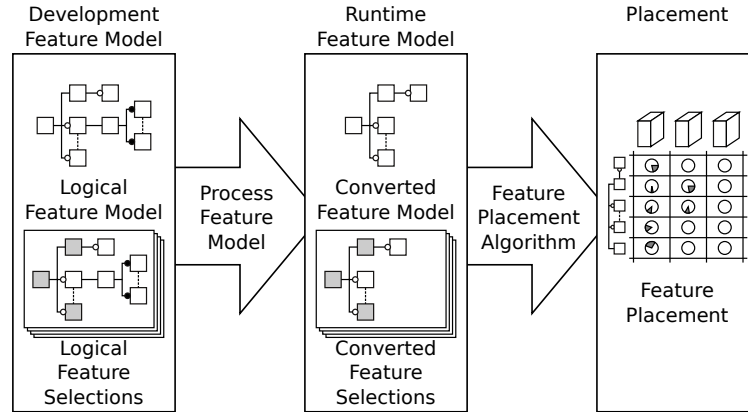


Figure 5.1: Runtime resource allocation for applications based on a logical feature model.

using this runtime feature model as an input. These steps are shown in Figure 5.1.

In this chapter we describe a system architecture for developing and managing such applications, focusing on the role of feature models within the management system, and on how a mapping between development and runtime feature models can be realized. To evaluate the presented approach, we use the application cases of the CUSTOMSS [5] project. Within the scope of this project a platform for developing and managing highly customizable cloud applications will be designed, with a focus on applications in the domain of document processing, medical information management and medical communication systems. We find that, using the presented conversion, development models can be kept significantly smaller than runtime models, while the overhead introduced by the transformation remains low.

In the next section we will discuss related work. In Section 5.3 we will discuss SPLE approaches for the development of SaaS applications. Subsequently, in Section 5.4 we will discuss the management architecture, and the role of feature modeling in it. We will elaborate on the conversion between feature models in a development and an runtime context in Section 5.5, after which we will evaluate the approach in Section 5.6. Finally, in Section 5.7, we will discuss our conclusions.

5.2 Related work

Within the domain of SPLE, a distinction is commonly made between problem-space and solution-space feature models, with mappings between the two [6, 7]. Our approach is complementary and adds an additional model, a runtime problem model, which is used to manage feature-based services. The runtime model is built using basic transformations applied on the development model, and is validated using a logical representation of feature relations, similar to those used in [8].

Recent work has been done in the field of service workflow variability. [9] focuses on how product lines spanning different organizational domains can be integrated, assuming each of the domains offers varying services. We on the other hand focus on how a single domain can offer these variable services. Closely related, in [10], a policy-based framework for publishing customization options of web services is proposed, enabling clients to build their own customizations. This work however focuses on the specification of variable applications, while we focus on the implementation and runtime aspects of developing customizable SaaS applications.

An approach to build variable SaaS applications is proposed in [11]. To achieve this, the authors however focus on configuration-based changes. We on the other hand focus on customization changes by using a SOA application development approach, making it possible to achieve greater customizability. Design-time variability management of SOA applications is discussed in [12] and [13]. We focus on how these development configurations are managed at runtime. The SOA approach in this chapter is similar to that proposed in [14], where a single application consisting of different components is proposed. Our approach, using different service instances is however more flexible, as it ensures services can have different, and sometimes incompatible dependencies.

A framework for native multi-tenancy is presented in [2]. In this work, security, isolation and software variability are considered. The authors focus on customization through configuration, laying the responsibility for managing the complexity of variability with the developer. Our approach on the other hand supports true customization changes using multi-tenant instances. Furthermore, the feature model conversion discussed in this work can be used to simplify the variability management done by the developers.

From a cloud management perspective, the described platform ensures underlying service instances are abstracted, effectively building a Platform-as-a-Service (PaaS) where entire services are managed at a higher level, as discussed in [15]. Many such platforms exist, both commercially and in literature. Our approach can be used to extend existing PaaS platforms, ensuring high-variability applications can be built on top of these platforms.

5.3 Variable Software as a Service development

SPLE techniques are often used to develop highly customizable applications. The variability in applications is modeled as a collection of features, where every feature represents a functionality of an application. These features are then related in a feature model. By including a collection of features in an application, and excluding other features, a specific *configuration* can be created. If this configuration is valid

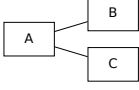
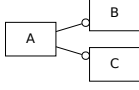
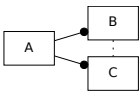
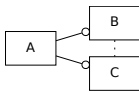
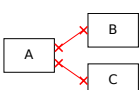
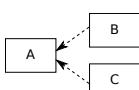
<p style="text-align: center;">Mandatory</p>  <p>If the parent is selected the child must be selected as well. Mandatory(A, B) Mandatory(A, C)</p>	<p style="text-align: center;">Optional</p>  <p>If the parent is selected the optional children can be selected. Optional(A, B) Optional(A, C)</p>
<p style="text-align: center;">Alternative</p>  <p>If the parent is selected exactly one of the child nodes must be selected. Alternative($A, \{B, C\}$)</p>	<p style="text-align: center;">Or</p>  <p>If the parent is selected at least one of the child nodes must be selected. Or($A, \{B, C\}$)</p>
<p style="text-align: center;">Excludes</p>  <p>The selection of one node makes it impossible to select the other. Excludes(A, B) Excludes(A, C)</p>	<p style="text-align: center;">Requires</p>  <p>The feature can only be selected if the required feature is selected too. Requires(B, A) Requires(C, A)</p>

Table 5.1: Graphical representation of feature models, description of relations, and formal representation.

according to the relations defined in the feature model, it can be used to specify a *variant* of the software application.

Features can depend on each other, they can be incompatible with each other, or they can relate in different ways. To make the complexity of feature models manageable, they are often specified in a hierarchical way [7, 8], where features have parent features and can only be included in an application when the parent feature is selected. These hierarchical feature models can be represented graphically, for which we use the notation as used in [16].

Four hierarchical relations, defined in Table 5.1 are typically used: *mandatory*, *optional*, *alternative* and *or*. In the table, we also define two additional relations that can occur between arbitrary features, making it possible to express more complex, non-hierarchical feature relations: the *excludes* and *requires* relations.

The different relations can be expressed logically using basic propositional calculus, where in a configuration for a software variant a feature in an application is either included (the value corresponding to the feature is assigned value 1) or excluded (the corresponding value is 0). The logical definition of the different

Relation	Conversion
Mandatory (A, B)	$A \leftrightarrow B$
Optional (A, B)	$B \rightarrow A$
Alternative ($A, \{B, C\}$)	$A \leftrightarrow B \vee C$ $\neg(B \wedge C)$
Or ($A, \{B, C\}$)	$A \rightarrow B \vee C$
Excludes (A, B)	$\neg(A \wedge B)$
Requires (A, B)	$A \rightarrow B$

Table 5.1: Conversion of feature model relations to logical statements.

relation types is shown in Table 5.1.

At this point, we need to make a distinction between configuration and customization changes [11]. Configuration changes do not impact the code of applications, and are caused by changes in configuration files or application metadata. An example of a configuration change is the logo that should be displayed for branding an application. Customization changes impact the code of an application, and change the code that must be executed. The inclusion of a feature that adds encryption to a process is an example of a customization change.

To enable the configuration of SaaS applications, existing techniques can be used [11]. Customization of SaaS applications can be achieved by splitting an application into separate services, that work together to realize the functionality of the application using a SOA [3, 4]. In this approach, every feature that leads to a customization change is realized by associating it with a multi-tenant service.

Figure 5.2 illustrates a feature model where all features are linked to a code module and a placement configuration. The placement configuration can be used to add additional restrictions that are to be taken into account when placing the service on infrastructure, for example ensuring that the feature can only be placed within the local datacenter or on a high-reliability instance. It is possible for two features to have the same code module, but a different placement configuration, resulting in different non-functional run-time behavior, as illustrated for the features e and e' .

Formally we introduce a relation **codeMap** that contains the relations between a feature and its linked code, and a relation **configMap** that links a feature to its placement configuration. For the feature model in Figure 5.2, the expression **codeMap**($a, a.jar$) links the feature a with its code module, and the expression **configMap**($a, Default$) indicates the feature is placed using the default placement configuration.

Within this chapter we will make a distinction between development feature models, which contain all variation types, and runtime feature models that retain only customization changes. During runtime, the specified configuration changes are added as application metadata.

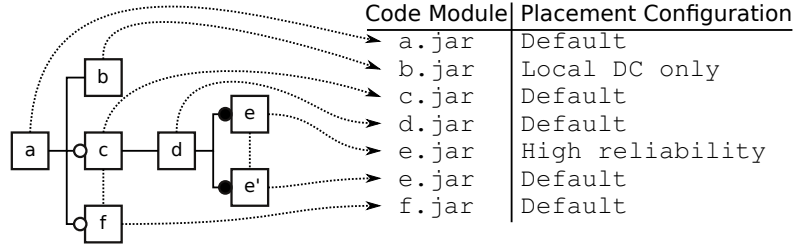


Figure 5.2: Linking code modules and configuration to features.

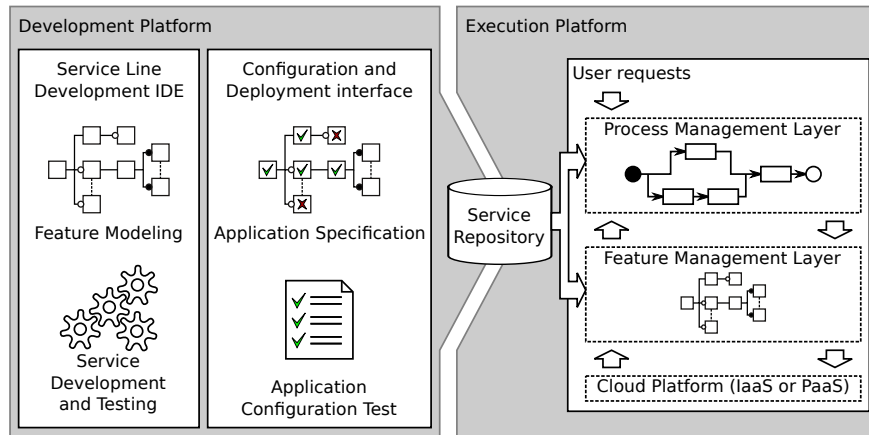


Figure 5.3: The system architecture.

5.4 System architecture overview

A general overview of the system architecture is shown in Figure 5.3. The system consists of two major components: a development platform, and an execution platform. These platforms are connected by means of a *service repository*. This component contains information of the currently deployed services, the work-flows that must be executed, and quality rules concerning these work-flows.

In the next sections we will give a general overview of both platforms, followed by an in-depth discussion of the management platform. Finally, we will discuss the role of feature models in the application architecture.

5.4.1 Development platform

The development platform contains a *service development Integrated Development Environment (IDE)*, which can be used to develop customizable applications. It allows the development of multi-tenant services, and provides the required feature modeling tools to build the application feature model. The IDE also provides a

testing environment where the functionality of individual services and composed applications can be evaluated.

A second part of the development platform is the *configuration and deployment interface*. This component can be used by either tenants or in-house vendors to create composed applications. This component is also responsible for testing the final service composition, as not all combinations can be tested during development. Should any tests fail, feedback concerning the configuration will be reported to the developers.

5.4.2 Execution platform

The execution platform is built using three layers. At the base there is a *cloud platform* layer, which contains an existing Infrastructure-as-a-Service (IaaS) or PaaS platform, and forms the foundation on which the execution platform is built. On top of this, a *feature management* layer manages the services allocated on the infrastructure. These services implement features selected in the application feature model, and can be used by multiple end users, and in multiple feature compositions. It is the responsibility of the feature management layer to ensure that the right server instances are created on the correct locations, given expected QoS requirements. Finally, a *process management* layer responds to individual user requests. When a request enters the system, a process, making use of feature instances, is instantiated and executed. The process management layer manages a work-flow, composed of the services that are controlled by the feature management layer and running on the cloud platform. Services are chosen by the process management layer, and updated during workflow execution based on monitoring information, ensuring the requested QoS is achieved.

5.4.3 Execution platform components

The execution platform, shown in Figure 5.4, manages and coordinates the communication between *multi-tenant feature instances*. The feature instances are executed on a cloud platform. By using a combination of these feature instances in a work-flow, a complete service can be constructed.

In the feature management layer, a *service allocation engine* is responsible for placing these feature instances on the cloud platform. The information stored in the service repository is used as an input for a feature placement algorithm [4], which returns a placement indicating which features are executed on which servers, and the amount of resources allocated to each of these features. A *monitoring component* collects input from the various components in the system, and is used to determine when the service allocation engine should reallocate services.

The process management layer consists of four components:

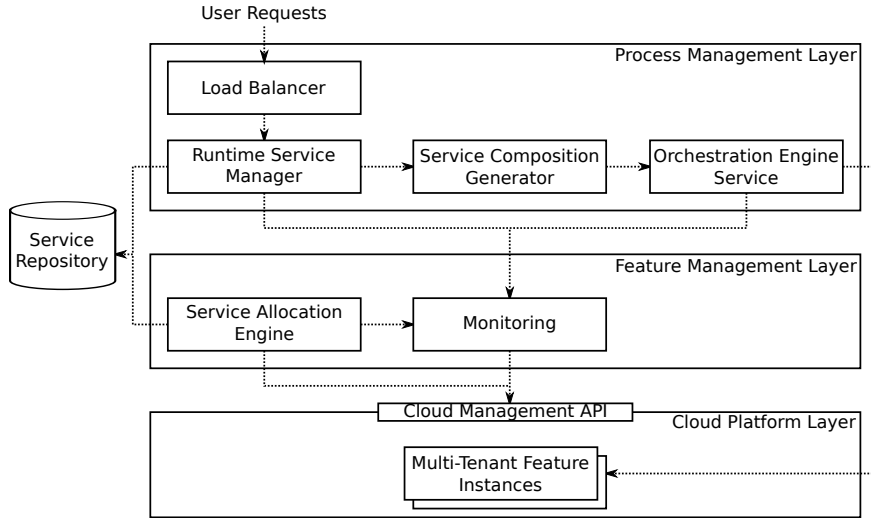


Figure 5.4: A detailed overview of the components in the execution platform.

- A *load balancer* is used to balance incoming requests, and forwards requests to a runtime service manager.
- The *runtime service manager* stores the location of different service compositions. It also maintains a register of services that are reserved for executing work-flows. This information enables the component to predict when QoS requirements risk being violated.
- A *service composition generator* is used to create and adapt service compositions. This component is used to generate a work-flow, and to bind concrete services to separate steps in the work-flow. This component is QoS-aware, and can dynamically update the composition based on feedback of the runtime service manager to ensure the QoS demands of the work-flow are met.
- An *orchestration engine service* is a multi-tenant service, running on the cloud platform, that contains a process execution engine. This service uses this execution engine to coordinate the different feature instances, combining them to create the desired process.

5.4.4 Role of feature models in the application architecture

Feature models of application are key resources at several points in the architecture:

- During development, the features of the application are modeled, and code and configuration information is associated with the feature model.

- During the specification of applications, a salesman or client fills in the feature model, specifying the features that are included in the application.
- During deployment and execution, the feature model of an application is used to determine an optimal placement of the service by solving a feature placement problem [4].

While, logically, these different phases make use of the same feature model, the view on this feature model is different:

- The developer sees the entire feature model, and links code modules to specific features.
- Salesmen are only interested in the changes that are externally visible. Implementation issues should be hidden. It is preferable to display the choices using a wizard-like interface as mentioned in [2], or using staged configuration [17] allowing the client to gradually fill in the feature model based on simple choices, without requiring a view on the entire software architecture or feature model.
- At runtime, only customization changes matter, and configuration changes can be completely removed from the feature model. A mapping between features and services must exist. At runtime it is impossible for features impacting the implementation of other features to exist, as only one service can be linked with a given feature. If features impacting others exist, the runtime mode must contain two separate versions of the impacted feature: one where the change is included, and one where it is not.

The properties of the three different feature model types are shown in Table 5.2. Within the subsequent sections, we focus on how the conversion between development feature models and runtime feature models can be implemented.

5.5 Feature model conversion

During development, all application features are included in the feature model. These features can imply both configuration changes and customization changes. At runtime, only customization changes should be considered, as configuration changes do not impact the feature model at a code level. Part of the feature model conversion is thus the *removal of configuration changes* from the runtime feature model.

A specific property of the feature placement algorithm is that it expects features to be linked to a code module and configuration. When two features share the same code but different configurations, for example a single-tenant and multi-tenant version of a component, this implies that both are considered as different features.

	Development Feature Model	External Feature Model	Runtime Feature Model
Customization changes	✓	✓	✓
Configuration changes	✓	✓	
Features change other features	✓	✓	
Wizard configuration interface		✓	
Implementation details hidden		✓	
Contains instance information	✓		✓

Table 5.2: Properties of the different feature model types.

A similar problem is the static addition of AOP aspects: when a feature causes the addition of an aspect to a set of components, the management system must ensure that two versions of the component are created as features, one with, and one without the modifications. This kind of behavior can be required when dynamic weaving of aspects would negatively impact the quality properties of the changed feature instances, decreasing the quality of other work-flows that make use of this modified feature. We call the creation of modified features, based on development-time features but statically including specific changes, either to the code modules or to the placement configuration, *feature expansion*.

We will define techniques for turning a development feature model, \mathcal{D} , into a runtime feature model, \mathcal{R} , by applying the two aforementioned transformations. We will also define a mapping between the two feature models, to represent how features selected in the development model correspond to those in the runtime feature model. This is represented using a mapping relation $\mathbf{Map} : \mathcal{D} \rightarrow \mathcal{R}$ that maps features from one model to the other.

Initially, before any transformations take place, $\mathcal{D} = \mathcal{R}$, and \mathbf{Map} contains for every feature f the relation $f \mapsto f$, mapping every feature from the development feature model to its corresponding feature in the runtime feature model.

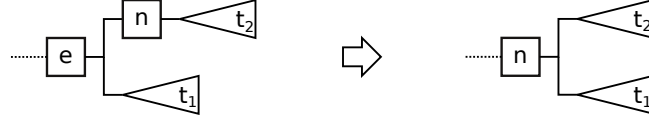


Figure 5.5: Elimination of empty mandatory features.

5.5.1 Removal of configuration changes

Removal of configuration changes is easy to achieve by simply turning the chosen feature into a dummy feature with nothing associated with it. This *empty feature* has no code nor placement information associated with it, so it will not be placed by the feature placement system. The inclusion of this empty feature only impacts the application metadata, ensuring the configuration change is realized.

Using these empty features ensures the logical relations of the feature model with respect to the feature are retained, but it also adds useless features to the feature placement process, increasing its computational cost. Transforming feature models, and thereby removing these useless features, the performance of the placement algorithm can be improved. It is however important that the transformed model is logically equivalent to the original model, implying not all empty features can at all times be removed. We call the process of removing redundant empty features *feature elimination*.

5.5.2 Feature elimination

Specific transformations can be used to, in some cases, eliminate empty features created by the removal of configuration changes discussed above. The elimination of mandatory empty features is shown in Figure 5.5, and is applicable whenever an empty feature e has a mandatory child n (as shown in the figure), or when a node n has a mandatory empty child e .

Formally, this transformation replaces any mappings referring to e in **Map** with references to n . The mapping $e \mapsto e$ is for example replaced by $e \mapsto n$. In the runtime feature model \mathcal{R} , all references to e in the relations are replaced by references to n . Finally, the effects of including e to the application metadata are added to n . The logical correctness of this transformation follows directly from the logical definition of **Mandatory**, which in this case ensures that $e \leftrightarrow n$.

A second transformation, the or elimination, is shown in Figure 5.6. This transformation can be used to eliminate empty features in **Or** constructs, if the empty features are themselves parent in an **Or** relation. The equivalence between both constructs shown in the figure follows directly from the fact that from $n \leftrightarrow e_1 \vee e_2$, $e_1 \leftrightarrow a_1 \vee a_2$, and $e_2 \leftrightarrow a_1 \vee a_2$, one can deduce that $n \leftrightarrow a_1 \vee a_2 \vee b_1 \vee b_2$. In the runtime model \mathcal{R} , all **Excludes** and **Requires** relations referring to any

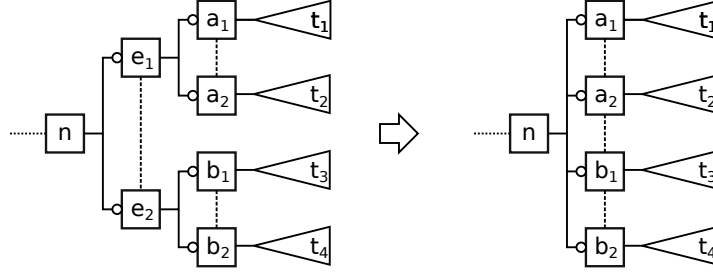


Figure 5.6: Elimination of empty or features.

of the removed nodes, are removed, and replaced by an equivalent relation for each of the children.

Applying the or elimination, unlike mandatory elimination, does remove information, as it becomes impossible to select the features e_i in a development model, as no corresponding feature in the runtime model remains. This limitation is taken care of by introducing a set of *artificial* features. This set represents features that are logically needed for the structuring of the model, but that can not be selected. These features are also removed from **Map**.

The or elimination as presented here can be generalized for arbitrary amounts of children. An identical transformation can be used, where all **Or** relations are replaced by **Alternative** relations.

5.5.3 Feature expansion

It is possible that, in the development feature model \mathcal{D} , the inclusion of a feature a changes the implementation of other features, ensuring either the code module linked to the feature changes, or causing the placement configuration of the feature to change. This approach can be used to add, for example a security feature: this can cause changes to the service implementation and can add server placement restrictions.

We introduce a construct **FeatureChanges**(a, f), which can be used in the \mathcal{D} model, indicating that the addition of a feature a changes the implementation of a feature f . Within the \mathcal{R} model, this relation can not exist, and must be removed.

The **FeatureChanges** relation is converted in the runtime feature model by creating two features f' and f_a . The mapping **codeMap**(f, c) in \mathcal{R} is replaced by a mapping **codeMap**(f', c) and an additional mapping **codeMap**(f_a, c_a) is added. In this case, c_a represents the code where the change caused by the feature f has been included. The **configMap** relation is changed similarly.

When the **FeatureChanges**(a, f) relation is included, this implies that three

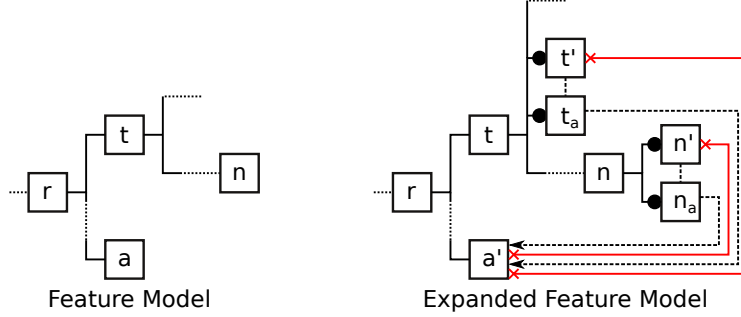


Figure 5.7: Feature expansion when it is applied to a subtree.

relations must hold in the runtime feature model:

$$a \wedge f \leftrightarrow f_a \quad (5.1)$$

$$\neg a \wedge f \leftrightarrow f' \quad (5.2)$$

$$\neg f \leftrightarrow \neg f_a \wedge \neg f' \quad (5.3)$$

Equation (5.1) expresses that if the feature f and a are selected, the f_a feature, which links to the code modified by the feature a must be selected too. Equation (5.2) ensures that if a is not selected, but f is, the feature f' without the code will be added. Finally, we ensure that, if the feature f is not selected, neither f_a nor f' is selected using Equation (5.3).

The **FeatureChanges**(a, f) relation can be expressed by combining the relationships described in Table 5.1. For clarity, the logical relations as described in Table 5.1 are also included.

$$\mathbf{Or}(f, \{f', f_a\}) \quad f \leftrightarrow f' \vee f_a \quad (5.4)$$

$$\mathbf{Excludes}(a, f') \quad \neg(a \wedge f') \quad (5.5)$$

$$\mathbf{Requires}(f_a, a) \quad f_a \rightarrow a \quad (5.6)$$

It can be proven that, together, these three relations are equivalent to the **FeatureChanges** relation, which can thus be expressed using these basic relations. This enables us to express this concept in a feature model that supports only a limited amount of relations between features. The feature expansion approach is illustrated in Figure 5.7, where a feature a impacts an entire subtree.

It is of note that the newly created features f' and f_a are not referenced in the \mathcal{D} feature model, making them part of the set of artificial features as discussed earlier, making them candidates for or elimination.

5.5.4 Feature model conversion algorithm

The techniques described above are combined into a feature model conversion algorithm, which we use to convert between development and runtime feature models. The algorithm takes a development feature model as an input, and first applies feature expansion. The amount of features in the expanded feature model is then reduced using feature elimination, resulting in the runtime feature model.

This feature model conversion algorithm is implemented within the service repository of the architecture discussed in Section 5.4, where it bridges between the development feature model and the runtime feature model.

5.6 Evaluation Results

We evaluated the algorithm to convert runtime feature models to runtime feature models using three feature models of applications used in the CUSTOMSS project. These applications are in the field of Document Processing (DP), Medical Information Management (MIM) and Medical Communications (MC).

Figure 5.8 shows the feature models of a MC application, in the various stages of the algorithm. The feature names have been replaced by numbers, where every feature is represented using the same number throughout the different images.

Figure 5.8a shows the development feature model of the application. For this model, Feature 21 impacts Feature 6, 9, 12, 15, and 18. This is noticeable after expansion, as shown in Figure 5.8b, where these features have new children, related to Feature 21 using **Excludes** and **Required** relations. Finally, after the elimination stage, some features are removed, lowering the total depth of the feature model, as shown in Figure 5.8c.

Feature expansion increases the amount of features, while feature elimination again decreases the present features in the runtime model. This is shown in Figure 5.9, where the feature counts for the three applications after the different phases are shown. We make a distinction between empty features, filled features and a total feature count.

Filled features are features linked to code and configuration, and are thus required during the deployment process. Empty features are only used to structure the feature model, and yield overhead during the placement. The increase of the amount of filled features during the feature expansion phase can not be prevented, and is indicative of why the transformation is useful: otherwise this variability needs to be managed during development. The increase in empty features is however not desirable, but as seen in the bar chart, this overhead significantly diminishes due to the feature elimination stage.

It is of note that the most significant increases in feature counts occur in the Document Processing application, where some features are impacted by multiple

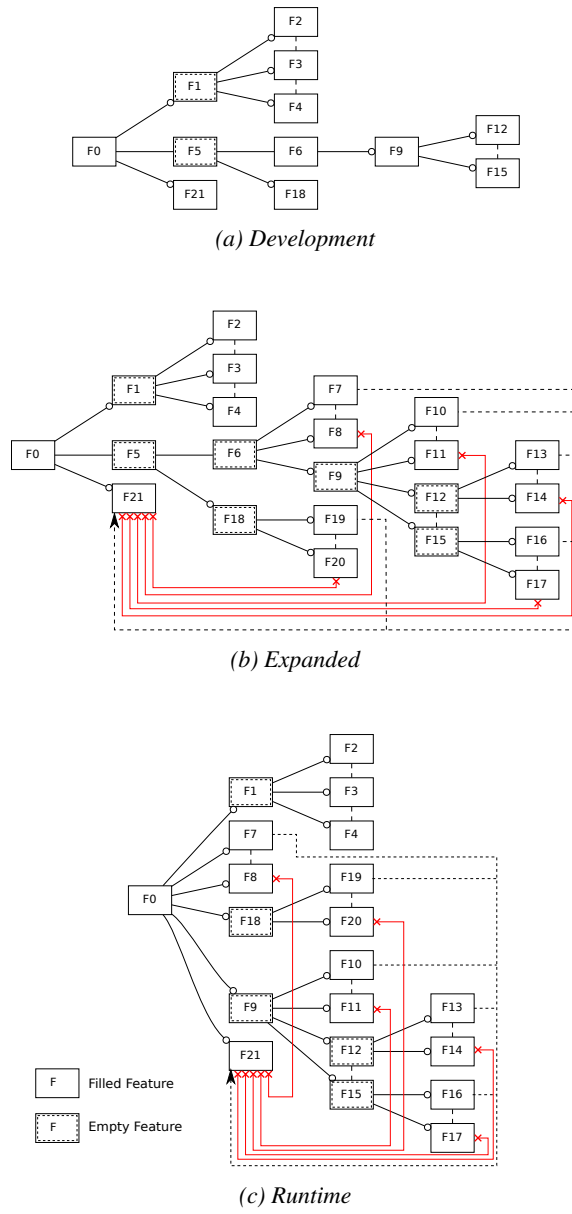


Figure 5.8: The feature model of a Medical Communication application in the development stage, after expansion and in the runtime stage.

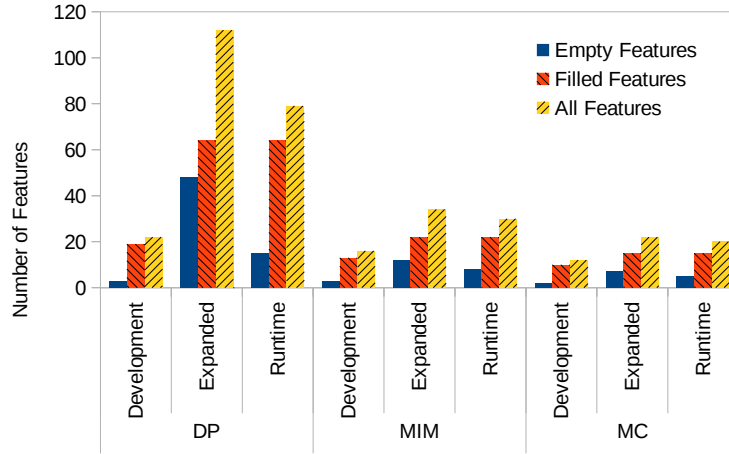


Figure 5.9: The features present in the models of the Document Processing (DP), Medical Information Management (MIM) and Medical Communications (MC) applications in the different feature expansion phases.

features, exponentially increasing the amount of variations. Most of the excess empty features created in this process are again removed in the feature elimination process. The exponential increase in features does however indicate a need for limiting the use of the discussed approach to those features that can not be implemented using alternative approaches, and maximally using configuration-based approaches for handling variability.

The execution speeds of the different algorithm phases were determined for the different application feature models. The results, shown in table Table 5.3, were determined using 1000 executions on an Ubuntu server with Intel Core 2 Duo T9400 CPU and 4GiB of memory. The results show that both algorithm phases can be executed fast, the actual execution speed is mainly influenced by the number of features generated.

To evaluate the impact of the transformation on the management complexity, we used the expanded and runtime feature models as input for the feature placement algorithm discussed in [4]. We considered a scenario containing 100 applications and servers. As seen in Table 5.4, where average times and percentiles are shown for 100 executions, placement consistently executes faster for the runtime models. For the MC and MIM cases the execution duration decreases by $\pm 5\%$, while for the DP model a reduction by $\pm 17\%$ is observed.

Phase	Model	Time (ms)	σ
Expand	Document Processing	3.56	2.19
	Medical Information Management	0.32	0.47
	Medical Communications	0.16	0.37
Eliminate	Document Processing	3.42	1.19
	Medical Information Management	0.34	0.47
	Medical Communications	0.13	0.34

Table 5.3: Execution speed of the different operations.

Input Model	AVG	σ	99 pct	98 pct	97 pct	50 pct
DP Expanded	2.19	0.31	2.75	2.74	2.71	2.20
DP Runtime	1.81	0.27	2.33	2.26	2.26	1.79
MIM Expanded	2.09	0.34	2.66	2.65	2.65	1.91
MIM Runtime	1.98	0.35	2.56	2.54	2.52	1.84
MC Expanded	2.08	0.38	2.72	2.67	2.64	2.00
MC Runtime	1.98	0.34	2.57	2.56	2.56	1.77

Table 5.4: The execution time of the feature placement algorithm for different models. Times are measured in seconds.

5.7 Conclusions

We presented a software architecture for the development, execution and management of highly customizable SaaS applications, and described how SPLE techniques and feature modeling can be utilized within this framework, focusing on how feature models are used at different stages in the software life cycle. The proposed feature model conversion algorithm can be utilized to convert development feature models to runtime feature models, increasing maintainability and expressiveness of development feature models. The transformation can be executed fast, in the order of a few milliseconds, and techniques are proposed to significantly limit the overhead of empty features that is added during the transformation, improving the execution speed of placement algorithms by 5 to 17% depending on the application case.

The proposed approach does come at a cost when features exist that are impacted by many different features, as an exponential amount of feature variations has to be created. This can however be remedied by only using this approach when it is impossible to resolve the variability using other techniques, such as configuration changes or AOP techniques.

In future work, we will extend the execution platform, which uses the feature model conversion and feature placement algorithms, and develop a process management layer enabling QoS aware composition of services.

Acknowledgement

Hendrik Moens is funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). This research is partly funded by the IBBT CUSTOMSS [5] project.

Addendum

The management and runtime platforms mentioned as future work in the conclusions of this chapter were partially implemented within the scope of the CUSTOMSS project using two separate proof-of-concepts focusing on different aspects of the project. Appendix B discusses how a medical communications platform was partially migrated to one of these platforms.

Proof of equivalence

Due to space constraints, the equivalence of the relations in Equations 5.1, 5.2 and 5.3, and the relations in Equations 5.4, 5.5 and 5.6 was omitted in Section 5.5.3 of the paper on which this chapter is based. For completeness, we now show a detailed formal proof of this equivalence. The **FeatureChanges**(a, f) relation is equivalent to the following logical equations:

$$a \wedge f \leftrightarrow f_a \quad (5.1)$$

$$\neg a \wedge f \leftrightarrow f' \quad (5.2)$$

$$\neg f \leftrightarrow \neg f_a \wedge \neg f' \quad (5.3)$$

For convenience, we define **Rels**(a, f), which is equivalent to the relations defined in Equations 5.4, 5.5 and 5.6:

$$\mathbf{Or}(f, \{f', f_a\}) \quad f \leftrightarrow f' \vee f_a \quad (5.4)$$

$$\mathbf{Excludes}(a, f') \quad \neg(a \wedge f') \quad (5.5)$$

$$\mathbf{Requires}(f_a, a) \quad f_a \rightarrow a \quad (5.6)$$

We prove this equivalence using natural deduction with the Fitch notation. The proof is split up into two separate parts, first showing that **Rels**(a, f) follows from **FeatureChanges**(a, f) and subsequently showing the reverse. From this, it follows that **Rels**(a, f) \leftrightarrow **FeatureChanges**(a, f).

FeatureChanges(a, f) \rightarrow **Rels**(a, f):

1	FeatureChanges (a, f)	Assumption
2	$a \wedge f \leftrightarrow f_a$	(1), definition FeatureChanges
3	$\neg a \wedge f \leftrightarrow f'$	(1), definition FeatureChanges
4	$\neg f \leftrightarrow \neg f_a \wedge \neg f'$	(1), definition FeatureChanges
5	$f \leftrightarrow \neg(\neg f_a \wedge \neg f')$	(4), $x \leftrightarrow y \leftrightarrow \neg x \leftrightarrow \neg y$, $\neg\neg x \leftrightarrow x$
6	$f \leftrightarrow f_a \vee f'$	(5), $\neg(x \wedge y) \leftrightarrow \neg x \vee \neg y$, $\neg\neg x \leftrightarrow x$
7	$a \wedge f'$	Assumption
8	a	(7), \wedge elimination
9	f'	(7), \wedge elimination
10	$\neg a \wedge f$	(3), \leftrightarrow elimination
11	$\neg a$	(10), \wedge elimination
12	\perp	(8, 11), \perp introduction
13	$\neg(a \wedge f')$	(7–12), \neg introduction
14	f_a	Assumption
15	$a \wedge f$	(2), \leftrightarrow elimination
16	a	(15), \wedge elimination
17	$f_a \rightarrow a$	(14–16), \rightarrow introduction
18	Rels (a, f)	(6, 13, 17), definition Rels

From this we can conclude that **FeatureChanges**(a, f) \rightarrow **Rels**(a, f) using implication introduction.

Rels(a, f) \rightarrow **FeatureChanges**(a, f):

1	Rels (a, f)	Assumption
2	$f \leftrightarrow f' \vee f_a$	(1), definition Rels
3	$\neg(a \wedge f')$	(1), definition Rels
4	$f_a \rightarrow a$	(1), definition Rels
5	$\neg f \leftrightarrow \neg f_a \wedge \neg f'$	(2), $x \leftrightarrow y \leftrightarrow \neg x \leftrightarrow \neg y$, $\neg(x \wedge y) \leftrightarrow \neg x \vee \neg y$
6	$a \wedge f$	Assumption
7	f'	Assumption
8	a	(6), \wedge elimination
9	$a \wedge f'$	(7, 8), \wedge introduction
10	\perp	(3, 9), \perp introduction
11	$\neg f'$	(7–10), \neg introduction
12	$\neg f_a$	Assumption
13	$\neg f_a \wedge \neg f'$	(11, 12), \wedge introduction
14	$\neg f$	(5, 13), \leftrightarrow elimination
15	f	(6), \wedge elimination
16	\perp	(14, 15), \perp introduction
17	f_a	(12–16), \neg introduction, $\neg\neg x \leftrightarrow x$
18	f_a	Assumption
19	a	(4, 18), \rightarrow elimination
20	$f' \vee f_a$	(18), \vee introduction
21	f	(2, 20), \leftrightarrow elimination
22	$a \wedge f$	(19, 21), \wedge introduction
23	$a \wedge f \leftrightarrow f_a$	(6–17, 18–22), \leftrightarrow introduction

24	$\neg a \wedge f$	Assumption
25	$\neg a$	(24), \wedge elimination
26	f	(24), \wedge elimination
27	$f' \vee f_a$	(2), \leftrightarrow elimination
28	f'	Assumption
29	f'	(28)
30	f_a	Assumption
31	a	(4), \rightarrow elimination
32	\perp	(25, 31), \perp introduction
33	f'	(32), \perp elimination
34	f'	(27, 28–29, 30–33), \vee elimination
35	f'	Assumption
36	$f' \vee f_a$	(35), \vee introduction
37	f	(2, 36), \leftrightarrow elimination
38	a	Assumption
39	$a \wedge f'$	(35, 38), \wedge introduction
40	\perp	(3, 39), \perp introduction
41	$\neg a$	(38–40), \neg introduction
42	$\neg a \wedge f$	(37, 41), \wedge introduction
43	$\neg a \wedge f \leftrightarrow f'$	(24–34, 35–42), \leftrightarrow introduction
44	FeatureChanges (a, f)	(5, 23, 43), definition FeatureChanges

Errata

Figure 5.7 shows the use of an **Alternative** relation when expanding the feature model, while the text references an **Or** relation instead. Both options are correct, as the additional relations in the **FeatureChanges**(a, f) relation prevent both options from being selected, even when they are related only using an **Or** relation.

In this case, the **Alternative** relation is semantically clearer. It however introduces redundant logical constraints. The **Or** relation results in a logically equivalent result, with fewer logical constraints, which is why **Or** relations were used in the evaluation.

References

- [1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-oriented programming*. In M. Akşit and S. Matsuoka, editors, ECOOP'97 — Object-Oriented Programming, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997. doi:10.1007/BFb0053381.
- [2] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao. *A Framework for Native Multi-Tenancy Application Development and Management*. In Proceedings of the 9th IEEE International Conference on E-Commerce and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, (CEC/EEE 2007), pages 551 – 558. IEEE, jul 2007. doi:10.1109/CEC-EEE.2007.4.
- [3] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl. *Variability modeling to support customization and deployment of multi-tenant-aware Software as a Service applications*. In Proceedings of the ICSE Workshop on Principles of Engineering Service Oriented Systems (PESOS 2009), pages 18–25. IEEE, may 2009. doi:10.1109/PESOS.2009.5068815.
- [4] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. D. Turck. *Feature Placement Algorithms for High-Variability Applications in Cloud Environments*. In Proceedings of the 13th Network Operations and Management Symposium (NOMS 2012), pages 17–24. IEEE, apr 2012. doi:10.1109/NOMS.2012.6211878.
- [5] *CUSTOMSS: CUSTOMization of Software Services in the cloud* [online]. 2013. accessed on 1/2013. Available from: <http://www.iminds.be/en/projects/overview-projects/p/detail/customss>.
- [6] A. Metzger, P. Heymans, K. Pohl, P.-Y. Schobbens, and G. Saval. *Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis*. In Proceedings of the 15th IEEE International Requirements Engineering Conference, 2007 (RE 2007), pages 243–253. IEEE, oct 2007. doi:10.1109/RE.2007.61.
- [7] F. Sanen, E. Truyen, and W. Joosen. *Mapping problem-space to solution-space features: a feature interaction approach*. ACM SIGPLAN Notices, 45(2):167–176, feb 2009. doi:10.1145/1837852.1621633.
- [8] D. Batory. *Feature Models, Grammars, and Propositional Formulas*. In H. Obbink and K. Pohl, editors, Software Product Lines, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer Berlin Heidelberg, 2005. doi:10.1007/11554844_3.

- [9] D. Dhungana, D. Seichter, and G. Botterweck. *Configuration of Multi Product Lines by Bridging Heterogeneous Variability Modeling Approaches*. In Proceedings of the 15th International Software Product Line Conference (SPLC 2011), pages 120–129. ACM, aug 2011. doi:10.1109/SPLC.2011.22.
- [10] K. Zhang, X. Zhang, W. Sun, H. Liang, Y. Huang, L. Zeng, and X. Liu. *A Policy-Driven Approach for Software-as-Services Customization*. In Proceedings of the 9th IEEE International Conference on E-Commerce and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, (CEC/EEE 2007), pages 123–130. IEEE, jul 2007. doi:10.1109/CEC-EEE.2007.9.
- [11] W. Sun, X. Zhang, C. J. Guo, P. Sun, and H. Su. *Software as a Service: Configuration and Customization Perspectives*. In IEEE Congress on Services Part II (services-2), pages 18–24. IEEE, sep 2008. doi:10.1109/SERVICES-2.2008.29.
- [12] M. Abu-Matar and H. Gomaa. *Feature Based Variability for Service Oriented Architectures*. In Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture (WICSA 2011), pages 302–309. IEEE, jun 2011. doi:10.1109/WICSA.2011.47.
- [13] G. H. Alf  rez and V. Pelechano. *Context-Aware Autonomous Web Services in Software Product Lines*. In Proceedings of the 15th International Software Product Line Conference (SPLC 2011), pages 100–109. ACM, aug 2011. doi:10.1109/SPLC.2011.21.
- [14] S. T. Ruehl and U. Andelfinger. *Applying Software Product Lines to create Customizable Software-as-a-Service Applications*. In Proceedings of the 15th International Software Product Line Conference (SPLC 2011), pages 16:1–16:4. ACM, aug 2011. doi:10.1145/2019136.2019154.
- [15] L. Rodero-Merino, L. M. Vaquero, V. Gil, F. Gal  n, J. Font  n, R. S. Montero, and I. M. Llorente. *From infrastructure delivery to service management in clouds*. Future Generation Computer Systems, 26(8):1226–1240, oct 2010. doi:10.1016/j.future.2010.02.013.
- [16] pure-systems GmbH. *pure::variants User’s Guide* [online]. 2012. Last accessed: December 2012. Available from: <http://www.pure-systems.com/Documentation.116.0.html>.
- [17] K. Czarnecki, S. Helsen, and U. Eisenecker. *Staged Configuration Using Feature Models*. In R. Nord, editor, Software Product Lines, volume 3154 of *Lecture Notes in Computer Science*, pages 162–164. Springer Berlin / Heidelberg, 2004. doi:10.1007/978-3-540-28630-1_17.

6

Shared Resource Network-Aware Impact Determination Algorithms for Service Workflow Deployment

H. Moens and F. De Turck

Published in Journal of Network and Computer Applications (JNCA), 2015

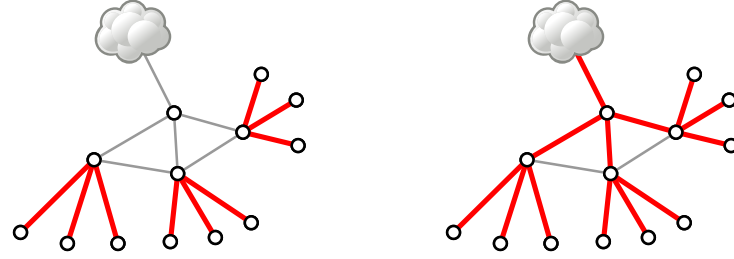
Some services use physical devices, terminals and servers that are installed at the customer's site. While the servers can be migrated to a remote environment during a cloud migration, locally installed devices must remain on-site. This implies that the communication between the various components must pass over the Internet, and can therefore no longer be isolated in a private subnet that is specifically dimensioned for its purpose. The client network is however not necessarily designed for these network flows, making it essential to determine how these networks are impacted when the service migration is executed, and when subsequent changes are made to the service. In this chapter we focus on how the impact of service workflows can be determined, ensuring service workflows do not negatively impact each other's execution. In particular, we determine an impact analysis strategy to evaluate the degree to which a given set of service workflows can be guaranteed in a given network topology. This impact analysis framework is designed as an access filter to be used in conjunction with the Feature-Based Binary (FBB) approach discussed in the previous chapters, validating whether sufficient network capacity is available before FBB applications are accepted and deployed.

6.1 Introduction

Many service providers install and maintain servers and physical devices on a customer's site to provide a service. These devices and servers work together, executing workflows that provide the service. When the service must be upgraded the devices must be replaced. Sometimes, the servers have hardware failures or have insufficient processing power, and must be replaced as well. This implies the service provider must dispatch technicians to carry out these changes on-site, making upgrading and maintaining the service offering costly for large customers, and prohibitively expensive for smaller customers. An example of such a service can be found in medical communications, where terminals are installed in hospital rooms and physical management servers are installed in the hospital. These devices are installed and maintained by a third-party service provider.

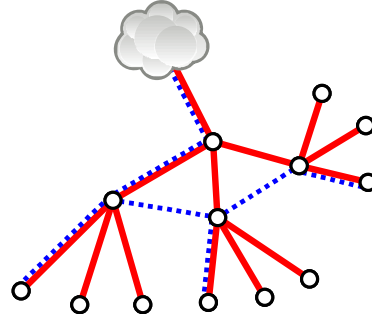
Traditionally, the on-site devices have been single-purpose, and built specifically to provide a small set of functionality. Currently, there is a trend to migrate to more generic devices, that can support a larger collection of features, and that can be mass-produced. This ensures the service offering can be updated without the need to replace any devices, and makes it possible to activate additional features at a later time. By moving management infrastructure to cloud environments rather than customer-site servers, it becomes easier manage and upgrade the service offering. This also makes it possible to make use of multi-tenancy for the management system, sharing a single management system instance between multiple clients. This can further lower the costs of providing services, as in such a system there no longer has to be a dedicated management instance running at all times for every individual client.

Offering multi-functional hardware and operating them using cloud computing, makes it easier and cheaper to upgrade the offered services, add new functionality, or to enable new services for customers. This in turn makes it possible to offer these services to smaller customers, for whom the cost of upgrading would have been prohibitive. This approach however also has an important downside. Originally, the devices, servers and network were specifically dimensioned to provide a specific service in a controlled environment. In this new configuration three major changes occur: 1) the load on the local network depends on the selected services, which can be changed during the system runtime, 2) as services may make use of cloud infrastructure, some service workflows must pass over a larger part of the customer network, and finally 3) as the configuration is done remotely, it is possible to deploy new services at runtime, without the need to visit the customer to install new servers or other devices. New services with varying network impacts can be rolled out more frequently, and these changes impact a larger part of the customer network. Because of this, it is important to create an access filter that can determine the impact on existing deployed services before new services are initialized, or upgraded. This problem is illustrated in Figure 6.1.



(a) In a network where all services are local, the network communication is restricted to a small part of the network. (The thick red lines represent the service communication flows.)

(b) By migrating multiple services to a remote cloud environment, the strain on the client network increases as services are no longer constrained to a small part of the network.



(c) Migrating services to a cloud makes it easier to upgrade them or to add new services as fewer changes are needed on-site. (In this figure, the dotted blue line represents a newly instantiated service.)

Figure 6.1: An illustration of the impact on a client network of service instantiations, migrations and changes.

For the considered cases the bottleneck is, both in terms of server capacity and network bandwidth, assumed to be in the customer network and the uplink between the customer network and the cloud environment. This assumption is made because the on-premise network and uplink typically have lower bandwidth than the network within the datacenter, even more so if high-bandwidth cloud instances are used, e.g. Amazon Cluster Compute Instances [1].

We refer to the problem of determining the impact on the network of deploying services as the Network-Aware Impact Determination (NAID) problem. We considered this impact analysis problem for the specific use case where services may either not fail at any cost, or when they are continuously processing information [2].

This implies that services may only be activated if all service workflows can be activated at the same time. When it is rare for all services to be active at the same time, this is an excessive requirement, and greatly limits the number of services that can be permitted on a network. In such a case it would be preferable to support resource sharing within the network, ensuring more services can be activated. This makes it possible to activate more services on the network, but it may also cause resource conflicts. Therefore, an approach is needed where resource sharing is maximized while at the same time the number of resource conflicts is minimized. In this chapter, we present Shared Resource Network-Aware Impact Determination (SRNAID), an extension to NAID which can be used to hierarchically structure workflows and achieve fine-grained control over resource sharing. We also propose a prioritization mechanism based on the NAID model to ensure important flows do not fail in case such resource conflicts occur.

In this chapter, we focus on network-aware impact analysis with support for resource sharing. We address three research questions: 1) How can resource sharing be incorporated when determining impacts of service workflows on a network during service deployment? 2) How can the resource conflicts occurring due to the sharing of resources be addressed at runtime, ensuring important services are impacted minimally? 3) How does the proposed approach perform, both in terms of service failures and in terms of performance? To this end, we first describe a SRNAID model for determining the impact of adding services with support of network and server resource sharing. To add more fine-grained control over the resource sharing, we define an approach where workflows are grouped in a hierarchy, and where resources are shared at multiple levels of this hierarchy. We then evaluate the SRNAID model using two use cases: a use case based on a Medical Communications (MC) application deployed in a hospital setting and a use case containing a varied collection of generated workflows.

While we focus on a medical communications use case in a partial cloud migration scenario, the designed algorithms can also be applied to general communications use cases, for instance in access control and home automation systems, and in scenarios where data is stored client-side due to compliance rules while (part of) the processing is done in a remote datacenter. Furthermore, the algorithms described may also be of use in multi-cloud deployment scenarios, and as a request admission filter in network function virtualization deployments.

In the next Section, we discuss related work. Afterwards, in Section 6.3, we discuss the basic NAID problem, which we extend to incorporate support for resource sharing in Section 6.4. In Section 6.5 we describe the conflict mitigation used to resolve resource conflicts during the simulation. The simulation setup is described in Section 6.6, and the results of our evaluations are presented in Section 6.7. Finally, we state our conclusions in Section 6.8.

6.2 Related work

Our approach to impact analysis is based on the use of network flow problems [3]. Multi-commodity flow problems [3] are a specific class of network problems that can be used to model various network-problems. Because of this, multi-commodity problems are commonly used in network management for solving various problems such as network routing problems [4–6], virtual network allocation [7], and design of fault-tolerant networks [8]. These approaches however work on the network level, and focus on routing flows from one network node to another. We on the other hand add service information to the input network, and focus on service-to-service routing: only the service that is executed matters, not where this service is executed, as long as server resource constraints are respected.

In our previous work [2], we have described a similar network impact analysis framework. In this chapter, we extend the presented approach and algorithm to incorporate support for network resource sharing, significantly reducing the amount of bandwidth needed to accept multiple services and thus increasing the number of services that can be provisioned on a given network. In this chapter we also propose a hierarchical model to better structure and control the various groups of services between which resources are shared, improving the quality compared to a flat resource sharing approach. The conflict resolution algorithm, which is described in this chapter and is used in the evaluation of the resource sharing aware algorithm, also makes use of the NAID algorithm described in [2].

Our approach differs from service selection as discussed in [9], where the authors describe an approach for selecting third-party services, such as cloud infrastructure, based on cost and other quality metrics, which focuses on the viewpoint of a client requiring the best-fitting service. We by contrast focus on the viewpoint of a service provider, who needs to select which services can be provided to a client based on the client's network and server capacities.

The approach described in this chapter has similarities with the application placement problem [10]. Application placement is used to determine the location of applications within networks [10–12] or clouds [13–15], taking into account the demand for each application. Application placement is used to coordinate applications. This work however focuses on the coordination of service workflows, rather than the management of individual services. In [16] network-aware placement of services is discussed, but the focus is the management of datacenters with specific layouts, so the techniques discussed cannot be directly applied to customer networks. Furthermore, the system assumes bandwidth is the only limitation, ignoring CPU limitations. Our approach however incorporates CPU limitations and can be applied to varying network layouts. In [11] and [17], an application placement algorithm based on a conversion to a network problem is discussed, but the physical network is not taken into account. Our work by contrast specifically focuses on the underlying

network.

Our approach further differs from application placement approaches as we assume that the services are already placed. We rather focus on determining which service workflows can be successfully executed, given a specific configuration. Thus the approach discussed in this work can be used in conjunction with existing application placement techniques, the application placement techniques being used to determine the service locations, and the SRNAID algorithms to determine the achievable workflows taking into account these service locations. As our techniques can be used to determine bottlenecks, it could also be used to enrich existing placement techniques. Alternatively, it would be possible to extend the described formal model using decision variables to signify instantiation of services on servers to directly use it as an application placement system, but this is not the focus of this chapter.

The NAID problem is similar to the service matching problem [18]. As in [19], we assume the service specification is known, but while the authors relax the capacity limit to achieve a polynomial time algorithm, we on the other hand focus specifically on these capacity constraints. By focusing on whether the required capacity for offering the services is present in the network, rather than on which specific service instances are used within the compositions, we similarly achieve polynomial time algorithms.

The problem we describe is the opposite of network dimensioning [20, 21] problems. Network dimensioning is used to determine the required network capacities for a given collection of services. Rather than dimensioning the network to be able to use a given number of services, we focus on determining the services that can be executed given a fixed network configuration. It would be possible to modify the model to use it to determine the minimal required network bandwidth for a given collection of services, and we use this approach to generate difficult problems in our evaluations. This is however not the focus of this chapter, and does not make it possible to incorporate network survivability, an important aspect for network dimensioning problems.

There are also similarities to the capacity assignment problem [22, 23], where network capacity is assigned to services within networks. We however do not focus on how capacities are divided, but rather on whether there is sufficient capacity at any point in time to provide the service. Additionally, we also consider both the services themselves, and the servers on which they are executing, ensuring there is both sufficient network and service capacity.

6.3 Network-Aware Impact Determination

NAID can be used to determine the impact of workflows on each other. In this context, a workflow is defined as a chain of communicating services between which

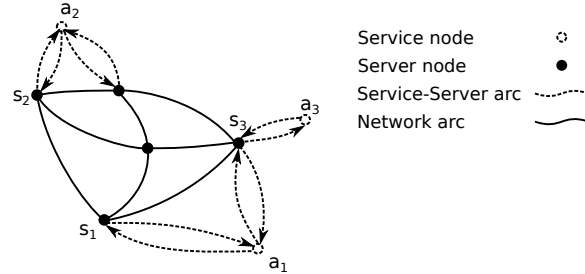


Figure 6.2: The NAID problem as a flow network. Each service-server arc in the Figure consists of two directed edges, while network arcs can either be bidirectional edges or two directed edges (this is e.g. useful for an uplink where download and upload capacities differ).

there is network demand. We have described the NAID problem and proposed various algorithms in our previous work [2]. In this section we present a brief summary of the basic NAID model. Some notational changes are introduced compared to our previous work to enable a more consistent presentation in the remainder of the chapter where we extend the model to support resource sharing.

We model the NAID problem by transforming the input into a graph problem where both servers and services are represented as nodes within the graph. This makes it possible to model the problem as a variation of the multi-commodity flow problem [4]. An advantage of this approach is that this problem can be solved using a Linear Programming (LP) solver, which as opposed to using Integer Linear Programming (ILP) solvers, do not result in exponential execution time complexity. Additionally, using an LP model, it becomes easier to make modifications to the model, e.g. to support non-linear service configurations. Within the graph, servers are interconnected based on their physical topology. Services are connected to servers on which they are running using two directed edges to represent incoming and outgoing streams of data to the service. We note that in this approach, routers and switches can be represented as server nodes on which no services are running. An example network is shown in Figure 6.2.

When services communicate, there is a network flow between both, where one service is the source of the flow, while the other is the sink. A network flow from the source service to the sink service is then determined. This flow will always move over at least one server, even when both the source and sink service are running on the same server. When the services are running on different servers, the flow moves over the physical network between the servers on which the services are running. In the example in Figure 6.2, a flow between services a_1 and a_2 must always move over the physical network (e.g. $a_1 \rightarrow s_1 \rightarrow s_2 \rightarrow a_2$), while a flow between a_1 and a_3 does not have to move over the physical network as both nodes are hosted on server s_3 , but the flow still moves over the server on which the services are running,

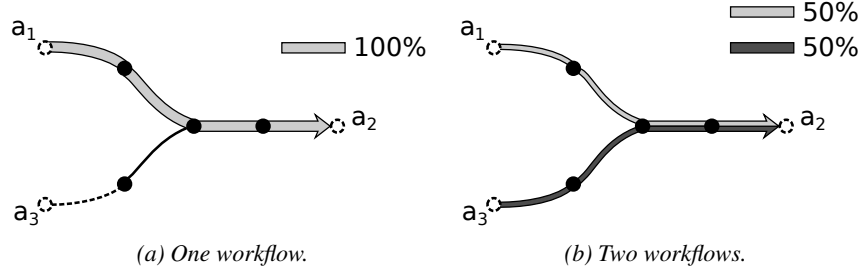


Figure 6.3: An illustration of the effect of adding workflows on the guaranteed network share of workflows.

resulting in the flow $a_1 \rightarrow s_3 \rightarrow a_3$.

The NAID problem focuses on the available network capacity for services, and the amount of capacity that can be guaranteed for services. Figure 6.3a shows an illustrative workflow between services a_1 and a_2 for which 100% of the capacity can be guaranteed at all times. Adding a second flow, as illustrated in Figure 6.3b shows the impact of adding a second flow: in this example, only 50% of the network demand of both services can be guaranteed at all times. This guaranteed network share is formalized as the value z , a value that is commonly used for this value in multi-commodity flow literature. This z value represents the share of the demand that can be guaranteed for *all* workflows within the network. We define a separate z_w value for every workflow w that can be guaranteed within the given network and service configuration. Whether or not a z_w value of less than 100% is acceptable for workflows such as those in Figure 6.3b depends on multiple factors:

- If one or both workflows may gracefully degrade, e.g. by reducing video quality if scalable video sources are used, or if delays are acceptable, a lower z value may be permitted.
- If the demand for the workflows fluctuates through time, it may be the case the probability of both requiring their maximum demand at the same time is small. In this scenario, resource conflicts could still occur, but depending on the probability of these conflicts occurring and their consequences the risk may be acceptable.
- If two workflows are guaranteed to never occur at the same time, e.g. as both use the same device at the same time for different purposes, there will never be a problem. While statically it is impossible to guarantee 100% capacity for both workflows in this scenario without taking this information into account, there will in practice never be resource conflicts making the configuration acceptable.

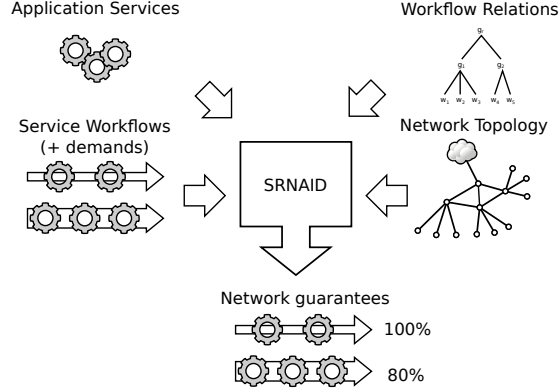


Figure 6.4: An illustration of the SRNAID problem input and output. The inputs and outputs of the NAID algorithm are similar, but NAID does not take workflow relations into account, making it incapable of sharing resources between workflows resulting in lower achieved network guarantees.

Making it possible to support these types of workflow interactions is an important motivation for the development of the SRNAID algorithm presented later on in the chapter. The inputs and outputs of the SRNAID and NAID algorithms are illustrated in Figure 6.4. In the remainder of this section we will formalize the NAID algorithm and model. In Section 6.4 we will then describe the SRNAID algorithm that extends the NAID model, taking workflow relations into account.

6.3.1 Base model: determining service workflow network impact

The NAID model contains a collection of nodes \mathcal{N} that are either server nodes, contained in a subset \mathcal{S} or application services \mathcal{A} . These nodes can be connected using either directed edges, contained in the set \mathcal{D} , or bidirectional edges, contained in the set \mathcal{B} . These bidirectional edges are implemented as two directed edges, and for every edge $b \in \mathcal{B}$, there is a corresponding edge $\text{cor}(b) \in \mathcal{B}'$, the set of corresponding edges. Together, these three collections of edges are contained in a collection of edges \mathcal{E} .

A collection of service workflows \mathcal{W} is defined. Every workflow $w \in \mathcal{W}$ is represented as a chain of services and demands, where each service implements a part of the workflow functionality. The workflow starts from an initial service, a_1^w and goes to the next service until a final service a_n^w is reached¹. The bandwidth

¹Note that an additional initial service a_0 is defined. The initial service a_0 is the same for all workflows, and is used to ensure that each service requiring CPU resources has an input flow, which is used to model CPU load constraints shown further on in this section. For more information concerning the initial service we refer to [2].

demand between two services is defined by the variable $d_{i,i+1}^w$, which is specified for every pair of services a_i^w, a_{i+1}^w within the workflow as illustrated in Equation (6.1). Each pair of services within the workflow becomes a separate network flow going from a source to a sink node, which is referred to as a *commodity* within the multi-commodity flow problem.

$$w : a_0 \xrightarrow{d_{0,1}^w} a_1^w \xrightarrow{d_{1,2}^w} a_2^w \xrightarrow{d_{2,3}^w} \dots \xrightarrow{d_{n-1,n}^w} a_n^w \quad (6.1)$$

The approach used to determine the demand between two services of a workflow, $d_{j,j+1}^w$, is shown in Equations (6.2) and (6.3), and is dependent on a rate between incoming and outgoing flows, $r_{in}^{out}(a)$ which is defined for every service a .

$$\forall w \in \mathcal{W} : d_{0,1}^w = d^w \quad (6.2)$$

$$\forall w \in \mathcal{W} : d_{j,j+1}^w = d_{j-1,j}^w \times r_{in}^{out}(a_j^w) \quad (6.3)$$

The optimization objective is to maximize the share of workflow demand that is allocated. That is, for every workflow $w \in \mathcal{W}$ there is a decision variable $z_w \in [0, 1]$ which indicates which share of the demand for workflow w that can be provided. If, for a workflow w , $z_w = 1$, it means that the capacity demanded by it is available to it all of the time. If, on the other hand a lower z_w value occurs, such as 0.5 it means that only 50% of the demand can be guaranteed at all times. This is not necessarily a problem as mentioned in the previous section: NAID does not take resource sharing into account, causing there to potentially be more resources available at runtime. If varying demands during runtime cause more resources to be available, or if the workflow is capable of scaling back resource requirements, the problem may be mitigated at runtime. The objective of the optimization is to maximize these z_w values, resulting in the maximum possible quality. This is formulated using the following optimization criterion:

$$\max \sum_{w \in \mathcal{W}} z_w \quad (6.4)$$

The model defines decision variables $f(e, c)$, representing the network flow over an edge $e \in \mathcal{E}$ for a commodity $c \in \mathcal{C}$. Using these edge flows, the net flow for a specific node and commodity combination can be determined. This net flow is the sum of all incoming flows minus the sum of all outgoing flows, and is shown in Equation (6.5). For nodes that are not a source or sink, these net flows should be zero, as no flow may be lost within the network. For source nodes there is a negative flow as there is more outgoing flow than incoming flow. Similarly, sink nodes have a positive net flow. Every commodity flow has an incoming flow that is equal to the demand for the commodity, represented as $d(c)$ which can be calculated as explained previously, multiplied by the share of the total workflow that is achieved,

which is represented by z_w . This is expressed in the flow conservation constraints shown in Equation (6.6).

$$\forall n \in \mathcal{N} : \forall c \in \mathcal{C} : f(n, c) = \sum_{(m,n) \in \mathcal{E}} f((m, n), c) - \sum_{(n,m) \in \mathcal{E}} f((n, m), c) \quad (6.5)$$

$$\forall n \in \mathcal{N} : \forall c \in \mathcal{C} : f(n, c) = \begin{cases} -z_w \times d(c) & \text{If } n \text{ source of } c \\ z_w \times d(c) & \text{If } n \text{ sink of } c \\ 0 & \text{Otherwise} \end{cases} \quad (6.6)$$

Edges between servers are subject to a network link capacity constraint expressed in Equation (6.7) and Equation (6.8) for directional and bidirectional edges respectively. These capacities can be measured using existing bandwidth estimation approaches [24–26] and are used as an input for the model. Edges between servers and services represent the execution of a service on a server, and are subject to a CPU capacity constraint, which is determined for every server. This constraint is shown in Equation (6.9); note that only directed edges need to be taken into account here as service-server edges are always directed. In this expression, $r_{in}^{CPU}(a)$ represents a ratio converting the used network bandwidth to a measure of CPU use.

$$\forall e \in \mathcal{D} \cap \mathcal{S} \times \mathcal{S} : \sum_{c \in \mathcal{C}} f(e, c) \leq \text{cap}(e) \quad (6.7)$$

$$\forall e \in \mathcal{B} \cap \mathcal{S} \times \mathcal{S} : \sum_{c \in \mathcal{C}} f(e, c) + \sum_{c \in \mathcal{C}} f(\text{cor}(e), c) \leq \text{cap}(e) \quad (6.8)$$

$$\forall s \in \mathcal{S} : \sum_{(s,a) \in \mathcal{D}} \left(r_{in}^{CPU}(a) \times \sum_{c \in \mathcal{C}} f((s, a), c) \right) \leq CPU_s \quad (6.9)$$

Workflows are a sequence of different execution steps: the incoming flow in a service is processed, and results in an outgoing flow from the service that is sent the next service. This is expressed in the workflow chain relation constraint shown in Equation (6.10).

$$\forall w \in \mathcal{W} : r_{in}^{out}(a_n^w) \times f((s, a_n^w), c_i^w) = f((a_n^w, s), c_{i+1}^w) \quad (6.10)$$

The service-server edges may only be used to provide flow going to the specific services, and may not be used for any other services. Because of this, two additional constraints, Equation (6.11) and Equation (6.12) are added to restrict respectively outgoing and incoming flows. In this formulation, a_1 and a_2 represent the source and the sink service of the commodity c .

$$\forall c \in \mathcal{C} : \forall a \in \mathcal{A} : \forall s \in \mathcal{S} : f((a, s), c) = 0 \text{ (unless } a = a_1) \quad (6.11)$$

$$\forall c \in \mathcal{C} : \forall a \in \mathcal{A} : \forall s \in \mathcal{S} : f((s, a), c) = 0 \text{ (unless } a = a_2) \quad (6.12)$$

An additional artificial service, a_0 is also added to ensure all workflows have an input flow. This is needed to correctly specify the CPU capacity constraint. As

this service is artificial, the flow for any workflow commodity (a_0, a_{w_1}) which starts in this service may not pass over server-server links. To enforce this, the constraint in Equation (6.13) is added for all $e \in \mathcal{E}$ of the type $(s_1, s_2) \in S^2$, and all commodities $c \in C$ for which the flow starts in a_0 .

$$\forall c \in C : \forall (s_1, s_2) \in S \times S : f((s_1, s_2), c) = 0 \quad (6.13)$$

For performance reasons, we only consider the CPU constraints for servers active within the customer network, and not for physical devices or cloud nodes. This is possible as the considered physical devices are built to provide a limited collection of specific services, thus they are designed to be capable of providing these services. Within the cloud, additional computational capacity can be requested on-demand when insufficient resources are present.

6.3.2 Non-workflow service graphs

Within this chapter we focus on service workflows. In practice, coordinating services can however not always be represented as linear workflows, and arbitrary service configurations may need to be supported. These configurations can be represented as a graph where the nodes are services and the edges are the capacity demands between the services. For this reason, we refer to them as service graphs. A workflow is a service graph where a source and sink node have one outgoing and one incoming edge respectively, while all other nodes have both an incoming and an outgoing edge. Thus, service graphs are a generalization of the service workflow concept.

The approach discussed previously can be extended to support such configurations: in the model, it is possible to create multiple flows from a single source node, which is all that is needed to add support for arbitrary service graphs. Table 6.1 shows how three service graphs can be represented within the flow based model: the workflow (as presented in the previous section), a graph where one service interacts with multiple services, and a star service graph. The same approach can be used to represent more complex non-linear service graphs.

In general, these service graphs can be defined by defining a $r_{in}^{out}(s_a, s_b)$ variable for all services s_a and s_b that are connected in the service graph. This variable specifies the network demand between the two services. As illustrated in Table 6.1, this value can then be specified based on the incoming flow within the service s_a . If multiple flows enter a service, the outgoing flow can either be specified based on one of the incoming flows, or the sum of the incoming flows. Arbitrary service graphs can therefore be supported by the model. Within the evaluations in this chapter we however focus specifically on linear service workflows.

Service graph	Flow specification	Defining Variables
$s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4$	$d_{0,1} = d$ $d_{1,2} = d_{0,1} \times r_{in}^{out}(s_1)$ $d_{2,3} = d_{1,2} \times r_{in}^{out}(s_2)$ $d_{3,4} = d_{2,3} \times r_{in}^{out}(s_3)$	d $r_{in}^{out}(s_1)$ $r_{in}^{out}(s_2)$ $r_{in}^{out}(s_3)$
$s_1 \rightarrow s_2 \rightarrow s_3$ \downarrow s_4	$d_{0,1} = d$ $d_{1,2} = d_{0,1} \times r_{in}^{out}(s_1)$ $d_{2,3} = d_{1,2} \times r_{in}^{out}(s_2, s_3)$ $d_{2,4} = d_{1,2} \times r_{in}^{out}(s_2, s_4)$	d $r_{in}^{out}(s_1)$ $r_{in}^{out}(s_2, s_3)$ $r_{in}^{out}(s_2, s_4)$
$s_1 \leftarrow s_2 \rightarrow s_3$ \downarrow s_4	$d_{0,2} = d$ $d_{2,1} = d_{0,1} \times r_{in}^{out}(s_2, s_1)$ $d_{2,3} = d_{0,2} \times r_{in}^{out}(s_2, s_3)$ $d_{2,4} = d_{0,2} \times r_{in}^{out}(s_2, s_4)$	d $r_{in}^{out}(s_2, s_1)$ $r_{in}^{out}(s_2, s_3)$ $r_{in}^{out}(s_2, s_4)$

Table 6.1: An illustration of how various service graphs can be represented within the flow-based model. The defining variables column shows which input variables are needed to characterize the flow.

6.4 Shared Resource NAID

The model discussed in the previous section is useful in a context where service workflows are continuously transmitting or in contexts where a very high reliability is needed. For many use cases, not all services will be active at the same time, making it preferable to support resource sharing. This ensures more services will be allowed to be deployed, at the risk of resource conflicts occurring. In this section we discuss how resource sharing can be added to the NAID model by allowing workflows to partially ignore each others resource requirements. First, we will discuss network resource sharing, and subsequently we will discuss server resource sharing. In both cases we will make use of a hierarchical specification of services, where similar service workflows are grouped together. This hierarchical approach makes it possible to group workflows with similar properties together making it much easier to specify the relations between workflows, and making it possible to have fine-grained control over the degree by which workflows may ignore each others resource requirements.

6.4.1 Adding resource sharing: network edge sharing

To achieve network edge capacity sharing, we make use of two major changes to the previously specified NAID model: 1) capacity constraints are expressed per workflow and per edge, rather than just per edge, and 2) to achieve a finer-grained control over the sharing of network capacity, the workflows are grouped in

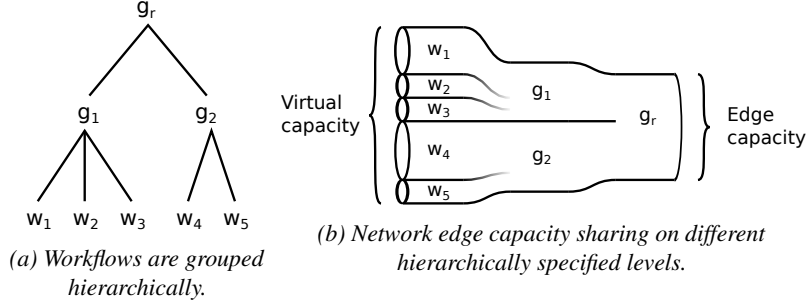


Figure 6.5: Hierarchical capacity sharing between edges ensures more capacity is available for individual workflows than physically present. The various workflows within a group share resources based on the characteristics of the group, such as the probability of occurring at the same time.

a hierarchy.

Within the NAID model, edge capacities are expressed using Equation (6.7). This expression is equivalent to Equation (6.14), where it is expressed for every workflow. By then adding an overprovisioning factor OF , as illustrated in Equation (6.15), a rudimentary system for sharing resources on edges can be defined. When $OF = 1$, no network resource sharing will occur, ensuring that the network is overprovisioned and that all resources will be able to be executed at the same time. If $OF < 1$ sharing of network capacity between service workflows occurs.

$$\forall e \in \mathcal{E} : \sum_{c \in C} f(e, c) \leq \text{cap}(e) \quad (6.7)$$

$$\forall e \in \mathcal{E}, w \in \mathcal{W} : \sum_{c \in w} f(e, c) \leq \text{cap}(e) - \sum_{c' \notin w} f(e, c') \quad (6.14)$$

$$\forall e \in \mathcal{E}, w \in \mathcal{W} : \sum_{c \in w} f(e, c) \leq \text{cap}(e) - OF \times \sum_{c' \notin w} f(e, c') \quad (6.15)$$

Using this approach, OF determines the degree to which other services are taken into account during optimization. An OF of 0 ensures that all other flows are ignored, while an OF of 1 results in the original equation of the NAID model where all workflows are fully taken into account and no sharing of resources occurs.

This approach is however not sufficient as all workflows are treated equally, offering only limited control. In practice different workflows can have different characteristics and probabilities of interfering with each other. To resolve this, we define a hierarchy containing workflows and groups of workflows. The leaves of this hierarchy are the workflows themselves, while inner nodes of the hierarchy group various workflows or other nodes together. An example is shown in Figure 6.5a, in this example a group g_1 contains three workflows w_1 , w_2 , and w_3 , group g_2 contains two workflows, and the root group g_r contains two groups g_1 and g_2 .

Every element of the hierarchy is assigned an individual capacity for every edge, and resource sharing occurs within individual groups. This concept is illustrated in Figure 6.5b, and makes it possible to control the process of resource sharing for every service and on multiple levels.

Formally, we define a collection of hierarchy nodes \mathcal{H} that contains both all workflows, contained in the collection \mathcal{W} , and all groups, contained in a collection \mathcal{G} . Each group $G \in \mathcal{G}$ can contain multiple workflows or flow groups. Every $h \in \mathcal{H}$, except for the root, is contained in exactly one group, creating the hierarchy. For the example in Figure 6.5a this results in the following collections:

$$\begin{aligned}\mathcal{W} &= \{w_1, w_2, w_3, w_4, w_5\} \\ \mathcal{G} &= \{\{w_1, w_2, w_3\}, \{w_4, w_5\}, \{\{w_1, w_2, w_3\}, \{w_4, w_5\}\}\}\end{aligned}$$

To make use of capacity constraints per group, we first define the capacity assigned to a workflow on an edge. This is shown in Equation (6.16) for directed edges and in Equation (6.17) for bidirectional edges.

$$\forall e \in \mathcal{D}, w \in \mathcal{W} : \text{cap}(e, w) = \sum_{c \in w} f(e, c) \quad (6.16)$$

$$\forall e \in \mathcal{B}, w \in \mathcal{W} : \text{cap}(e, w) = \sum_{c \in w} (f(e, c) + f(\text{cor}(e), c)) \quad (6.17)$$

We also define a new decision variable $\text{cap}(e, g)$ for all groups $g \in \mathcal{G}$ and all edges $e \in \mathcal{D} \cup \mathcal{B}$. Finally, a constraint is defined for every group in the hierarchy, linking the capacity assigned to its children to its own capacity. For this we use an expression based on that of Equation (6.15), but rather than defining it for the total set of commodities \mathcal{C} it is used within individual groups. The resulting expression is shown in Equation (6.18). This expression makes use of a factor OF_G which is defined within a group. Every group can be assigned a different OF_G value, depending on characteristics of the group.

$$\begin{aligned}\forall G \in \mathcal{G}, h \in G, e \in \mathcal{D} \cup \mathcal{B} : \\ \text{cap}(e, h) \leq \text{cap}(e, G) - OF_G \times \sum_{h' \in G \setminus \{h\}} \text{cap}(e, h')\end{aligned} \quad (6.18)$$

As mentioned previously, the workflows and groups form a hierarchy. The root group, G_r , has a capacity which is limited by the edge capacity. This is expressed in Equation (6.19). Additionally, a constraint to ensure the final capacity remains larger than that used for individual workflows is also added. This constraint, shown in Equation (6.20) ensures that no individual flow can ever be larger than the network edge capacity.

$$f(e, G_r) = \text{cap}(e) \quad (6.19)$$

$$\forall c \in \mathcal{C} : f(e, c) \leq \text{cap}(e) \quad (6.20)$$

6.4.2 Adding resource sharing: server capacity sharing

The server resource formulation which is included for all $s \in \mathcal{S}$ can be reformulated in a similar way to the transformation of network capacity limits. For this we first define an expression $L^{CPU}(s, w)$ which, for a workflow $w \in \mathcal{W}$, determines the CPU load on the server:

$$L^{CPU}(s, w) = \sum_{(s,a) \in \mathcal{E}} \left(r_{in}^{CPU}(a) \times \sum_{c \in w} f((s, a), c) \right)$$

By modifying the NAID model CPU constraint, Equation (6.9), to incorporate this L^{CPU} formulation, Equation (6.21) can be obtained. This formulation is similar to that shown previously in Equation (6.21), and like in the previous section, an overprovisioning OF factor can be added as expressed in Equation (6.22).

$$\forall s \in \mathcal{S} : \sum_{(s,a) \in \mathcal{E}} \left(r_{in}^{CPU}(a) \times \sum_{c \in C} f((s, a), c) \right) \leq CPU_s \quad (6.9)$$

$$\forall s \in \mathcal{S} : \sum_{(s,a) \in \mathcal{E}} \left(r_{in}^{CPU}(a) \times \sum_{w \in \mathcal{W}} \sum_{c \in w} f((s, a), c) \right) \leq CPU_s$$

$$\forall s \in \mathcal{S} : \sum_{w \in \mathcal{W}} \sum_{(s,a) \in \mathcal{E}} \left(r_{in}^{CPU}(a) \times \sum_{c \in w} f((s, a), c) \right) \leq CPU_s$$

$$\forall s \in \mathcal{S} : \sum_{w \in \mathcal{W}} L^{CPU}(s, w) \leq CPU_s$$

$$\forall s \in \mathcal{S} : \forall w \in \mathcal{W} : L^{CPU}(s, w) \leq CPU_s - \sum_{w' \in \mathcal{W} \setminus \{w\}} L^{CPU}(s, w') \quad (6.21)$$

$$\forall s \in \mathcal{S} : \forall w \in \mathcal{W} : L^{CPU}(s, w) \leq CPU_s - OF \times \sum_{w' \in \mathcal{W} \setminus \{w\}} L^{CPU}(s, w') \quad (6.22)$$

Analogous to how this was done in the previous section, we can now define this constraint for hierarchically specified groups. For every group $G \in \mathcal{G}$ and every server $s \in \mathcal{S}$ we define a new decision variable $L^{CPU}(s, G)$ representing the CPU capacity assigned to a group. The capacity of the root group G_r is defined as equal to the CPU capacity of the server:

$$L^{CPU}(s, G_r) = CPU_s \quad (6.23)$$

We now add flow group capacities and an overprovisioning factor, OF_G , for every group, similar to how this was done in this in Section 6.4.1. The resulting

constraint is shown in Equation (6.24).

$$\forall G \in \mathcal{G}, h \in G, s \in \mathcal{S} : \\ L^{CPU}(s, h) \leq L^{CPU}(s, G) - OF_G \times \sum_{h' \in G \setminus \{h\}} L^{CPU}(s, h') \quad (6.24)$$

6.5 Runtime conflict management

When resources are shared between workflows, and the above SRNAID algorithm is used to determine whether services are allowed to be deployed, it is possible for resource conflicts to occur. These resource conflicts occur when, at a given time during the execution, there is more demand for resources than there are available. How these resource conflicts are handled depends on the services and workflows themselves, and these conflicts can be handled in multiple ways: 1) the workflows may execute in a reduced quality mode (e.g. by using lower bitrate video and audio), 2) the workflow execution may be delayed until there is sufficient capacity for its execution, or 3) the workflow may fail causing it to not be executed at all. Only the last two approaches can be used generically, and often delaying workflows may cause unacceptable delays causing them to fail as well. Therefore we focus on the last scenario. Often, some workflows are more important than others, making it important to develop a strategy for dynamically failing less important workflows to prevent these more important flows from failing. Whenever a new workflow is started, the system state must be reevaluated: it must be determined whether the workflow can be added to the system, or whether it conflicts with already existing workflows. If there is a conflict this must be resolved. To achieve this, we make use of a modified NAID algorithm which is used in a workflow addition algorithm.

6.5.1 Adding quality levels: class-aware NAID

An important part of the conflict mitigation algorithm is a class-aware NAID algorithm. This algorithm is based on the NAID model without resource sharing outlined in Section 6.3, but incorporates the concept of a workflow class, indicating the importance of the workflow. We refer to this algorithm as *NAID**. By associating a class with workflows, we can prioritize the execution of important workflows compared to other, less important, workflows.

With every workflow $w \in \mathcal{W}$ we associate a value $\text{class}(w)$ which represents the quality level or importance of the flow. A workflow with a low class value is deemed more important than a workflow with a higher class value, with the highest workflow quality level being 0. The *NAID** algorithm takes one additional parameter compared to the regular NAID algorithm: w_{new} , the workflow which is the newest workflow to be added to the system. Additionally, a collection of new

constraints, shown in Equation (6.25), is added:

$$\forall w \in \mathcal{W} : \text{class}(w) \leq \text{class}(w_{new}) \Rightarrow z_w = 1 \quad (6.25)$$

This constraint ensures that, all workflows of the same or higher priority will be fully provided, and only lower-quality workflows may still fail. When running the *NAID** algorithm, there are three possible outcomes:

1. Every z_w value is 1. This implies all of the requested workflows can be provided.
2. Some z_w values are less than 1. In this case some lower-priority workflows can not be provisioned correctly, but all of the workflows of the quality of w_{new} and all workflows of higher quality can be fully provisioned.
3. The resulting model is infeasible and no configuration can be determined. This implies that it is impossible to provide the workflows of the current quality level and higher quality, and that at least one of these flows must be removed. In a runtime scenario where the *NAID** algorithm is executed iteratively this is an interesting case, as if at a given point in time the result of *NAID** execution is infeasible, and at the previous point it was feasible the infeasibility can be traced back to the last added workflow w_{new} .

Note that this algorithm does not take resource sharing into account as only workflows that are active at a given point in time are taken into account.

6.5.2 Workflow addition algorithm

To add workflows to a given configuration, Algorithm 8 is used. The algorithm takes as input a set of active workflows, and a new workflow that is to be placed, and returns three possible values: 1) *OK*, which indicates that the workflow can be added without impacting the active workflows; 2) *Fail*, which indicates that the workflow can not be added, as it would negatively impact high-priority active workflows; and 3) *Abort(abortedFlows)*, which indicates that the workflow can indeed be added, but that to do so the returned set of lower priority workflows must be aborted, causing those workflows to fail.

The *doPlace* function first solves the *NAID** algorithm which was mentioned in the previous section. If the *NAID** formulation is infeasible, this implies that it is impossible to achieve a configuration where all workflows with the quality level of *new* and higher are successful, even if all lower quality flows are eliminated. Thus, the new flow may not be added and *Fail* is returned.

If *NAID** succeeds, the workflows for which $z_w < 1$ are determined. If there are none, *OK* is returned and the algorithm finished. If some workflows fail, the conflict will have to be mitigated. These failed flows are stored in the set *fail*. At

Data: *active*: Set of currently active workflows
Data: *new*: New workflow to be placed
Result: Return state

```

1 Solve  $NAID^*$  for workflows  $active \cup \{new\}$ ;
2 if Infeasible then
3   | return Fail;
4 else
5   |  $failed \leftarrow$  flows for which  $z_i < 1$ ;
6   | if  $failed = \emptyset$  then
7     | return OK;
8   | else
9     |  $abortedFlows \leftarrow$  mitigateConflict(active, new, failed);
10    | return Abort(abortedFlows);
11  | end
12 end

```

Algorithm 8: The doPlace function. The algorithm can return three states: *OK* indicating the new workflow it can run; *Fail* indicating the new workflow can not run; and *Abort*(*abortedFlows*) indicating the new flow can run, but that some active flows, those contained in the set *abortedFlows*, must be aborted to do so.

this stage it would be possible to return all the failed workflows and abort their execution, which would result in a feasible configuration. This configuration could however be suboptimal in two ways: 1) it could occur that multiple workflows are not entirely successful in the $NAID^*$ solution, but that removing only a single flow is sufficient to achieve a feasible configuration; or 2) it may be possible to resolve the conflict by aborting lower-priority flows than those present in the set *fail*. For these reasons a separate mitigateConflict function is defined.

Algorithm 9 shows how resource conflicts are mitigated. First, the algorithm tries to resolve the conflict by removing the workflow that achieves the lowest z_w value, referred to as *worst*, and trying to place that workflow with doPlace, assuming all other workflows, including the new one, are active. This can result in two possible responses²:

1. *Abort*: It is possible that the recursive call leads to a list of failed flows that must be aborted. By aborting these flows a feasible configuration can be determined.
2. *Fail*: This implies that workflow *worst* can not be placed in the configuration. We then retry adding the original workflow *new* using doPlace, but remove workflow *worst* from the active set of workflows.

²The exit state *OK* of the doPlace function can not occur here as otherwise conflict mitigation would not have been initiated in the first place.

Data: *active*: Set of currently active workflows

Data: *new*: New workflow to be placed

Data: *failed*: The flows that failed in a previous placement

Result: *fail*: The workflows that must be removed to yield a feasible configuration

```

1 worst  $\leftarrow$  workflow with lowest  $z_i$  in failed;
2 reversedState  $\leftarrow$  doPlace(active  $\cup$  {new}  $\setminus$  {worst}, worst);
3 if reversedState = Abort(abortedFlows) then
    /* A feasible configuration exists by removing
       the selected set of workflows */
4     return abortedFlows;
5 else if reversedState = Fail then
    /* The workflow worst is incompatible with
       higher-class workflows, it must be completely
       removed */
6     restrictedPlace  $\leftarrow$  doPlace(active  $\setminus$  {worst}, new);
7     if restrictedPlace = Abort(abortedFlows) then
        /* Aborting the resulting flows and worst
           results in a feasible configuration */
8         return abortedFlows  $\cup$  {worst};
9     else if restrictedPlace = OK then
        /* By removing worst a feasible configuration
           was reached */
10        return {worst};
11    end
12 end

```

Algorithm 9: The mitigateConflict function used to determine the set of workflows that are to be removed to resolve a resource conflict.

6.5.2.1 Addition algorithm termination

The `doPlace` function makes use of the `mitigateConflict` function and vice versa, creating a recursive structure. To ensure the execution always terminates, we consider two values associated with every invocation of `doPlace` and `mitigateConflict`: $\text{class}(\text{new})$, the class of the newly added workflow, and $|\text{active}|$, the number of elements contained within the set *active*.

We first note that, if `mitigateConflict` is invoked from `doPlace`, the same *active* and *new* as used in the `doPlace` invocation are used, so we can focus specifically on the `mitigateConflict` function. The `doPlace` function is invoked at two points in `mitigateConflict`:

- The `doPlace` invocation on Line 2 tries to place the workflow *worst*. As *worst* had a z_i value less than 1, this implies *worst* has a lower priority than *new*, and thus a higher class, as otherwise $NAID^*$ would have assigned z_i value 1 to it because of the constraint in Equation (6.25). This new invocation of `doPlace` thus works on a class of $\text{class}(\text{worst}) > \text{class}(\text{new})$. The number of workflows that is to be placed, $|\text{active}|$ remains the same in this invocation.
- The `doPlace` invocation on Line 6 is executed using a set $\text{active} \setminus \{\text{worst}\}$. As the resulting set contains one less item, it is of lower cardinality than in the previous iteration. As the workflow *new* that is placed is the same, $\text{class}(\text{new})$ remains the same for the invocation.

As the number of workflows that can exist is finite, there is a workflow w_{mc} with the maximal class value $\text{class}(w_{mc})$ and thus the lowest priority. The minimal cardinality of a set is that of the empty set: $|\emptyset| = 0$. Every invocation of `mitigateConflict` either increases $\text{class}(\text{new})$ by at least one in the next invocation of `doPlace`, or decreases $|\text{active}|$ by one. As there is an upper limit to $\text{class}(\text{new})$ and a lower limit to $|\text{active}|$ the algorithm is guaranteed to terminate in at most $O(a + c)$ steps, where $a = |\text{active}|$, the number of active workflows, and $c = \text{class}(w_{mc}) - \text{class}(\text{new})$, the number of workflow classes that exist that are higher than that of *new*.

6.6 Evaluation approach

The algorithms presented in this chapter were implemented using Scala [27], the models were implemented using the CPLEX [28] LP solver. To evaluate the approach we make use of two evaluation scenarios, where we evaluate the quality of the resulting network using discrete event simulation: a Medical Communications (MC) use case and a randomized scenario.

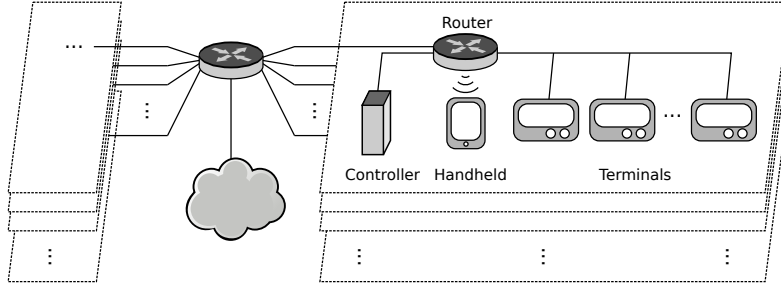


Figure 6.6: The network set-up used in the evaluation environment. The environment models two buildings with multiple floors and a collection of end-user devices on every floor. The entire set-up is connected to a remote cloud environment.

6.6.1 Medical communications use case

MC applications provide multiple communication services within hospitals and other care facilities such as nursing homes. While the cloud has many benefits related to manageability and scalability, a full migration is impossible as the services depend on locally installed hardware terminals. An important side effect of a cloud migration is the increasing impact on the client network, which often uses older hardware and has only limited capacity. Despite this, it is vital that services can be executed uninterrupted, making it important to determine whether services will impact each other before they are deployed. For this, we use the SRNAID approach. The considered MC use case provides three separate services, Nurse Call (NC), Voice over IP (VoIP) and video. The primary and most important functionality is NC, which is provided using terminals and buttons installed in rooms that can be used by patients to call hospital personnel. This important service is subject to strict quality and compliance rules, but requires only limited amounts of network bandwidth. VoIP and video are also offered as add-on services, but while they require significantly more network bandwidth, the impact of failing to provide the service is less severe. Within this system, various workflows exist to provide the three services. Each of these workflows represents a single part of the service, e.g. a patient initiating a NC or a nurse initiating a VoIP call.

To provide the services, three types of devices are installed at the customer's site: controllers, handheld mobile devices and terminals. These devices communicate with each other, and with a remote cloud environment. The customer network layout consists of two buildings, with different floors. Routers are provided per-floor and they are connected to a central uplink. The result is a tree-based structure, where inner nodes represent the network infrastructure and routers, while the leaf nodes are controllers, handhelds and terminal nodes. The network set-up is shown in Figure 6.6. For the quality evaluations a small hospital is used with two buildings, three floors per building, and ten leaf nodes.

Name	Load	Start	Duration
NC	2 KBps	Exponential $\lambda = 1/h$	Normal $\mu = 1s, \sigma = 100ms$
VoIP	10 KBps	Exponential $\lambda = 1/h$	Erlang $\mu = 30s, \sigma^2 = 50s^2$
Video	100 KBps	Exponential $\lambda = 1/5h$	Erlang $\mu = 50s, \sigma^2 = 90s^2$

Table 6.2: MC evaluation scenario workflows.

Within the system, there are three types of service workflows, which are shown in Table 6.2. The workflows connect end user terminals with a local controller, and for compliance and logging reasons a reduced version of the stream is sent to a cloud-hosted management component. For each of the workflows we determine a network load, a start distribution and a duration distribution. The start distribution determines the next time when the workflow will be started, while the duration distribution determines how long a workflow will require resources once it has been activated.

The start distribution of each of the workflows is exponential which is chosen as it describes events that occur continuously and independently at a defined rate. For NC, the duration of the workflow is normally distributed as this is an automated process. For the other workflows an Erlang distribution is used, as the distribution of the duration of VoIP and video workflows are similar to that of phone calls, for which an Erlang distribution is a good fit. The parameters for the various distributions are based on input from industry partners. These parameters can vary based on the type of facility, as e.g. hospitals typically have more frequent invocations compared to retirement homes.

The service hierarchy used in the SRNAID algorithm contains three levels. There is the root level *root*, which contains three groups for the three service types, *NC*, *VoIP* and *Video*. Finally, the three groups each contain all of the workflows that are defined for the respective services.

6.6.2 Generated use case

We also consider a second use case containing a larger variety of different randomly generated workflows to determine how the algorithms behave when there is a higher variety in workflows and when data and server intensive workflows interact. In this scenario, we generate a large collection of workflows which we deploy on the previously discussed client network. These workflows have random length and load characteristics, and can be divided into two types: Server Intensive (SI) workflows that require many server resources and limited network resources, and

	Parameters	
Number of workflows	50 of each type	
Flow length	Chosen using uniform distribution from $[2, 8]$	
Occurrence frequency	Exponential distribution. On average α times per interval t with $t \in [30, 90]$ minutes.	
Duration	Normal distribution. $\mu \in [10, 60]$ seconds, $\sigma = 10$ seconds.	
	Server Intensive	Data Intensive
Network demand	$[1, 50]$ MBps	$[10, 500]$ MBps
Server demand	$[100, 1000]$ MHz	$[10, 100]$ MHz

Table 6.3: Parameters used in the generated evaluation scenario workflows. Values within ranges are chosen at random using a uniform distribution.

Data Intensive (DI) workflows that require more network resources and less server resources.

The parameters used for both workflow types are shown in Table 6.3. Every workflow is generated with randomized length and resource requirements. For every service contained in a workflow, the number of nodes on which it may run is chosen at random using a non-negative normal distribution with mean 1 and $\sigma = 10$ ensuring it can only run on a limited number of servers. There is a 30% probability that a service may be allocated on the cloud, the other nodes on which the service may run are selected at random from the edge nodes present in the network. The occurrence frequency of the workflows is exponential and depends on a parameter α representing the average number of occurrences within an interval of duration t . The α parameters are equal for all workflows of the same type, and are thus specified separately for SI and DI workflows, represented as α_{SI} and α_{DI} . The time interval t is chosen randomly for every workflow. By modifying the α parameters, the occurrence frequency of DI and SI workflows can be modified. Half of the workflows of each type are marked as a high priority workflow, the other flows are marked as low priority.

Like in the MC scenario, the service hierarchy in the generated scenario contains three levels. At the top of the tree, there is a level *root* with contains the SI and DI service types represented as *SI* and *DI*. Finally, these two groups contain the corresponding workflow instances. This results in three parameters OF_{root} , OF_{SI} and OF_{DI} .

6.6.3 Simulation approach

The simulation environment contains a collection of *start* and *stop* events, that are queued. A timestamp is associated with each of the events, and they are processed

in-order. When a *stop* event is processed, a new *start* event is generated denoting when the service is activated next. When a start event is processed, the workflow is added using the doPlace algorithm explained in Section 6.5, and based on the result of the algorithm execution the workflow is either added to the set of active services, the workflow is not started, or other workflows are aborted. Based on the resulting state, the required collection of *start* and *stop* events are generated. The events can then be analyzed to determine the percentage of workflows that were correctly executed, that were aborted, and that were not executed can be determined.

The SRNAID problem is used to determine whether services can be added to the system, and whether their activation could result in problems. To evaluate the algorithm, we require a difficult problem model, where enough services are active, and where there is little remaining network capacity, as otherwise there would never be a conflict: activating a collection of low-bandwidth services in a high-bandwidth network will never be a problem. Because of this, we use the SRNAID model to determine a *minimally dimensioned network*. We achieve this by, rather than maximizing the achieved z_w values, instead minimizing the total network resources while every z_w is assigned value 1.

The resulting network resource configuration is the network with the smallest network capacities that, for the given services, still allows all of them to run. Using this hardest network, we can then simulate the use of the services for an extended period of time to determine the amount of violations, which can be used to determine the number of service workflows that fail. This approach can be used to determine the worst-case failures: any possible network that is accepted must have at least the same amount of capacity. If the network has higher capacity, the services will still be accepted, and failure rates will decrease.

6.7 Evaluation

6.7.1 Quality

6.7.1.1 Medical communication use case

We first evaluate the quality of the SRNAID algorithm by simulating the execution of a collection of workflows, and measuring the number of resource conflicts that occur. As mentioned, the networks are dimensioned using a modified SRNAID execution where network capacities are minimized, resulting in the tightest network where the regular SRNAID algorithm can still accept all workflows. In the simulation, four parameters can be changed: OF_{root} , OF_{NC} , OF_{VoIP} , and OF_{video} . These values correspond to the overprovisioning factors for the root of the hierarchy and each of the three services. We also compare our results with a simpler flat, non-hierarchical approach, where only a single OF is used. For every parameter

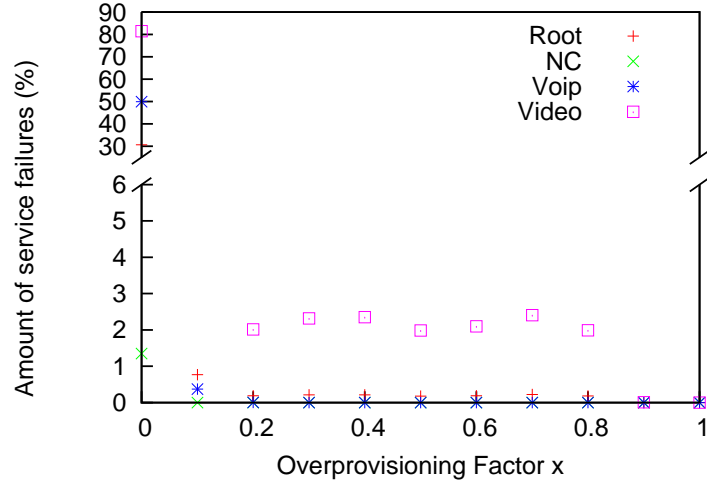


Figure 6.7: *EqualOF*: Fail rates using SRNAID in groups with varying overprovisioning factors. $OF_{root} = OF_{NC} = OF_{VoIP} = OF_{video} = x$

configuration, three weeks of execution were simulated based on the hospital layout described in the previous section.

In Figure 6.7, the failure rates are shown in a scenario where all OF factors are equal. We refer to this scenario as *EqualOF*. When this OF factor is zero, all resources are shared, implying every network edge only has enough capacity to provide for a single video workflow (as video requires most bandwidth). As can be observed, the failure rate for video flows is extremely high, and there is also a high failure rate of VoIP calls. This is to be expected: as there is only bandwidth for running a single video flow, starting any other workflow will cause the video flow to fail. Despite this very low capacity, there are still relatively few failures for the critical NC service due to its lower network demand and as the conflict management ensures it is preferred. As the common OF factor is increased, the number of failures decreases. Once $OF \geq 0.2$, all NC calls succeed, but VoIP failures only stop once $OF \geq 0.9$, and at this point there are still some rare video flow failures.

Another choice of OF values is the *FixedRoot* approach. This approach is similar to *EqualOF*, but the root factor OF_{root} is assigned value 1. Intuitively, we can explain the *FixedRoot* approach as follows: we share resources between services of every service type, to ensure less network resources are needed, but assign a fixed amount of resources to the services themselves to ensure they minimally interfere with each other. As can be seen in Figure 6.8 the fail rates are noticeably lower: once the OF values of the three services exceeds 0.3 no more failures occur in any service.

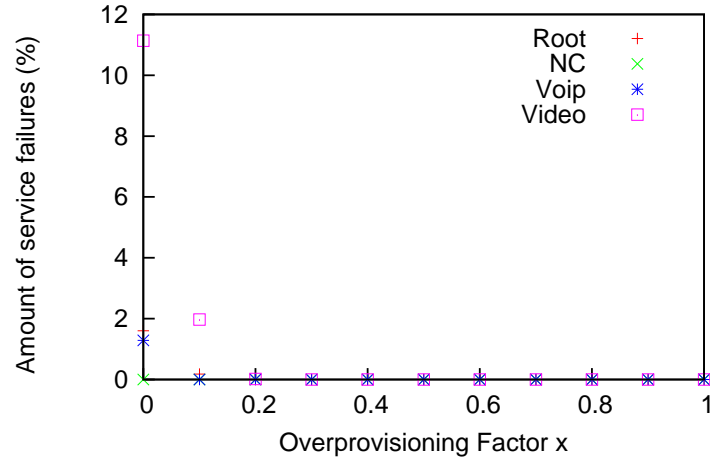


Figure 6.8: *FixedRoot*: Fail rates using SRNAID in groups with varying overprovisioning factors and fixed root overprovisioning. $OF_{root} = 1$, $OF_{NC} = OF_{VoIP} = OF_{video} = x$

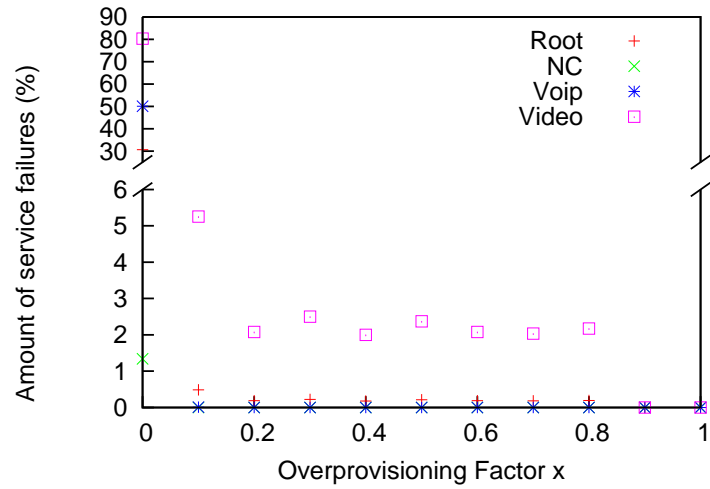


Figure 6.9: *Flat*: Fail rates using SRNAID without hierarchies. $OF = x$

Finally, we consider a last approach, *Flat*, which does not make use of service hierarchies and considers all services equally, making use of only a single overprovisioning factor OF . The fail rates of this approach are shown in Figure 6.9, and are similar to those of *EqualOF*.

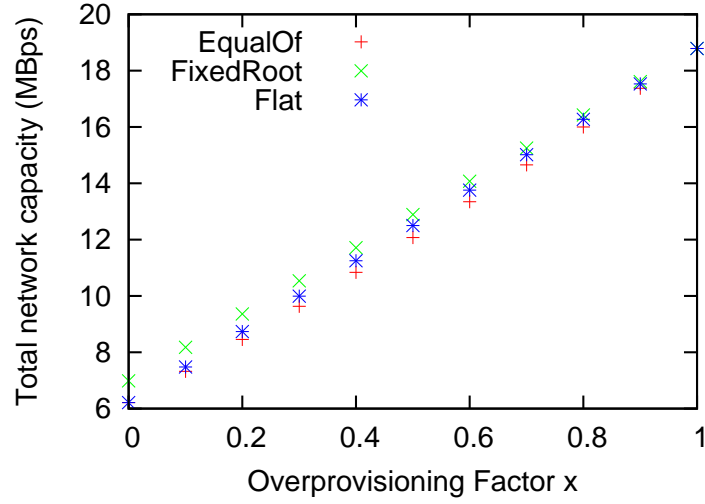


Figure 6.10: Required bandwidth for the three discussed SRNAID approaches with varying overprovisioning factors.

The required network capacity for the three approaches is shown in Figure 6.10. We observe that for the same choice of OF a similar cost is achieved, with the *EqualOF* approach requiring the lowest amount of capacity, while the *FixedRoot* approach requires most resources. This is to be expected as the *EqualOF* shares resources twice, once for level in the hierarchy, while the *FixedRoot* and *Flat* approaches only share resources once. As the $OF_{root} = 1$ in the *FixedRoot* approach, there is slightly less resource sharing compared to the *Flat* approach, which is why it requires more resources. The case where $OF_{root} = OF_{NC} = OF_{VoIP} = OF_{video} = 1$ corresponds to the regular NAID approach without resource sharing; in this case there can be no failure of workflows.

When combined with the failure rates, we can conclude that the hierarchical approach using fixed OF_{root} , *FixedRoot*, performs best, requiring only between 10 and 12MBps of network capacity to function without any failures, while the *Flat* and *EqualOF* approaches both require 18MBps of network capacity to achieve similar results. The latter network capacities are only slightly less than the required network capacity for the algorithm without resource sharing, which is 19MBps. For the considered cases, the *FixedRoot* approach requires 42% less resources than an approach without resource sharing, while still achieving similar qualitative results. The *FixedRoot* approach also requires 38% less resources compared to the *Flat* and *EqualOF* approaches to achieve similar results.

Using hierarchies to increase control of the resource sharing can thus greatly increase the number of services that can be allowed on a system, while at the

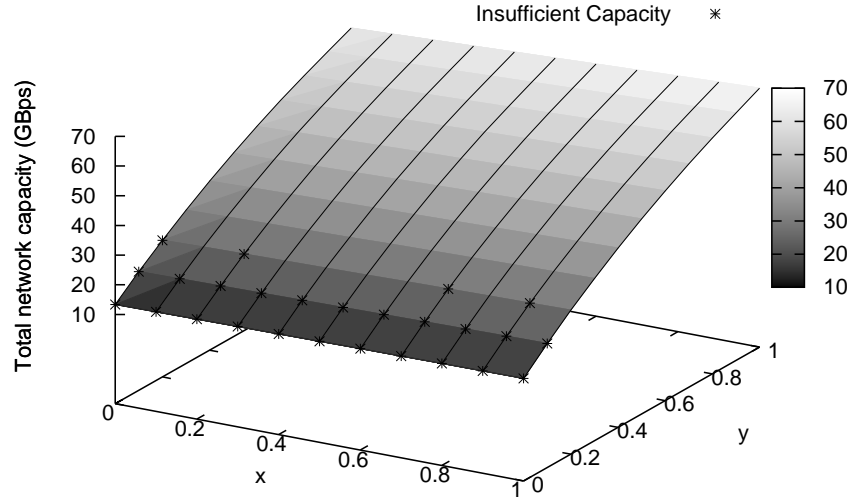


Figure 6.11: The required total network bandwidth for varying OF parameters in the generated scenario ($OF_{root} = x$, $OF_{SI} = OF_{DI} = y$). Markings show where workflow failures occurred during any of the simulation runs (DI, SI or balanced) due to insufficient network capacity.

same time also increasing the quality these services can achieve: the *FixedRoot* approach both needs less bandwidth and achieves better quality results than the *Flat* approach where no service hierarchies are used. The choice of parameters in the hierarchy is important, as evidenced by the worse results of the *EqualOF* approach.

6.7.1.2 Generated use case

We evaluate the SRNAID algorithm using the generated use case for various service frequencies α_{DI} and α_{SI} to determine the effect of the distribution of server and data intensive workflows. We consider three scenarios: (1) a Server Intensive (SI) scenario ($\alpha_{SI} = 9$, $\alpha_{DI} = 1$), (2) a Data Intensive (DI) scenario ($\alpha_{SI} = 1$, $\alpha_{DI} = 9$), and (3) a balanced scenario ($\alpha_{SI} = 5$, $\alpha_{DI} = 5$). In the first scenario, SI workflows activate on average 9 times every time interval while the DI workflows only activate once every time interval. In the second scenario, the opposite happens while in the final scenario both workflow types activate on average 5 times during every time interval.

In Figure 6.11, the total network cost for various OF_{root} , OF_{DI} and OF_{SI} combinations is shown. The markings show the OF values where any of the workflows failed during the simulation for any of the three scenarios (DI, SI or balanced). We observe that increasing OF_{root} has a limited impact on both cost and

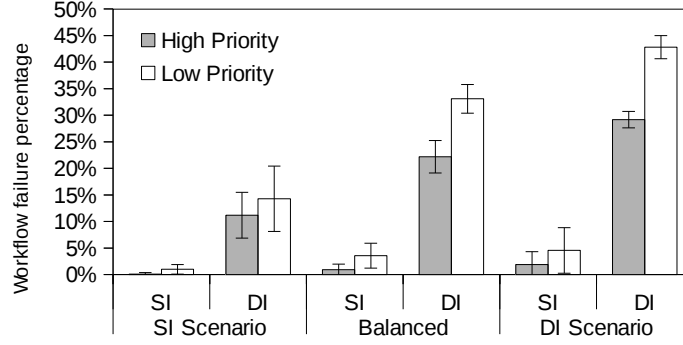


Figure 6.12: Failure rates of DI workflows in the DI, SI and balanced scenarios using a heavily underprovisioned network ($OF_{DI} = OF_{SI} = 0$).

quality compared to increasing OF_{DI} and OF_{SI} . Using $OF_{DI} = OF_{SI} = 0.3$ no failures occur in any of the simulations, irrespective of the chosen OF_{root} . With $OF_{DI} = OF_{SI} = 0.2$ failures still occur sporadically, but only for DI workflows in the DI scenario (with a failure rate of less than 1%). For lower OF values, failures occur frequently for DI workflows in all scenarios. SI workflows only fail during simulation in the cases where $OF_{DI} = OF_{SI} = 0$.

Figure 6.12 compares the workflow failure percentages for the three scenarios for a heavily underprovisioned network with $OF_{DI} = OF_{SI} = 0$ where failures of every workflow type occur. We observe that the SI workflows generally have lower failure rates compared to DI workflows, and that high priority workflows fail less frequently than low priority workflows. The SI scenario results in the lowest failure rate for all workflow types, while the DI scenario results in the highest number of failures. As was shown in Figure 6.11, increasing the OF values quickly reduces the failure rates until no failures occur during simulation.

Without resource sharing, 67GBps network capacity is needed for provisioning all workflows. Using resource sharing, a significant reduction in resources can be achieved resulting in a total network demand of 32GBps without any failures during simulation, resulting in a capacity reduction of $\pm 52\%$ compared to an approach without resource sharing. If infrequent workflow failure is tolerated, the total network capacity can be further reduced to 26GBps resulting in a capacity reduction of $\pm 61\%$ compared to an approach without resource sharing.

This evaluation shows that the model can incorporate both network and server resource demand. It is however clear that network constraints are more stringent and result in more failures than server resource constraints. This is not surprising, as when there is sufficient capacity for most DI flows, there will almost always be sufficient residual network resources for the lower network demand required for the SI flows, even if the services used in the workflow are allocated further apart. This

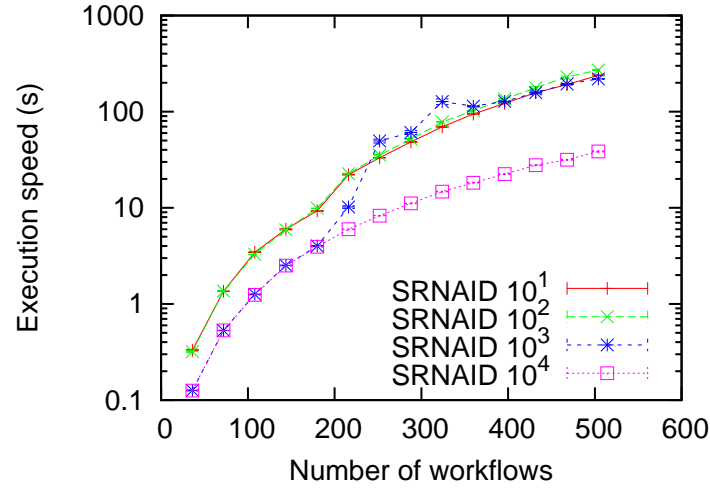


Figure 6.13: The execution speed of the SRNAID algorithm for varying network edge capacities. The SRNAID parameters are chosen based on the results of the quality evaluation: $OF_{root} = 1$, and $OF_{NC} = OF_{VoIP} = OF_{video} = 0.3$. 40 iterations per data point.

implies that, as long as there is a service where the service may run, wherever it is located within the network, there is likely to be sufficient network capacity between them for these workflows.

6.7.2 Execution speed

The execution speed of the SRNAID algorithm was evaluated using a server with dual-socket quad-core Intel Xeon L5420 processor and 16GB RAM. The base evaluation setup is the same as the that of the MC use case, but we incrementally increase the number of leaf nodes, resulting in an increase of active workflows. We make use of the *FixedRoot* approach as this yields the highest quality at the lowest resource requirement. The *OF* factors for the SRNAID algorithm are thus chosen as follows: $OF_{root} = 1$, and $OF_{NC} = OF_{VoIP} = OF_{video} = 0.3$.

The results of this evaluation are shown in Figure 6.13, where the performance of the SRNAID algorithm is shown for increasing numbers of workflows. The SRNAID algorithm is executed four times, for input problems with varying edge capacities in $\{10^1, 10^2, 10^3, 10^4 Kbps\}$. These varying capacities can impact the problem complexity, as it is easier to find a good solution if much network capacity is present. We incrementally increase the number of terminals per floor, which results in an increase in the number of possible workflows.

It can be seen in the Figure that the execution time of the algorithm increases as the number of workflows increases. The algorithms with limited network capacity (10^1 and 10^2 KBps) require more time to run than the algorithms with higher network bandwidths (10^4 KBps) as it is more difficult to determine a result that respects all of the constraints. This is especially noticeable for the scenario where there is 10^3 KBps network capacity is available: for smaller numbers of workflows, there is sufficient capacity making it perform similar to the case where there is 10^4 KBps of network capacity per edge. As the demand increases due to the increasing number of workflows, network capacity becomes a bottleneck, causing it to then start behaving like the problems with less available bandwidth.

When 500 workflows are active, the evaluations with more available bandwidth require only 16% of the execution time of the lower bandwidth algorithms. This difference can be explained, as when more bandwidth is available, it is much easier for the ILP solver to find a solution where every workflow can be completely executed. When there is insufficient bandwidth, it becomes more difficult to find the optimal solution, maximally satisfying the amount of allocated capacity. This is an interesting property, as when the SRNAID algorithm is used as an admission filter, it will mainly be used to validate the configuration: there will usually be sufficient capacity to execute all workflows, resulting in more favorable execution speeds. The worst-case performance will only occur in highly bandwidth constrained scenarios.

6.7.3 Algorithm scalability

As observed from Figure 6.13, the performance of the SRNAID algorithm is best when there is sufficient network and server capacity for all of the service workflows, making the impact analysis run faster. This has interesting consequences: if a workflow is deployed in a scenario where there is sufficient capacity, the SRNAID algorithm will finish quicker resulting in limited overhead. If there is less capacity, the algorithm will need more time to run. Thus, the SRNAID results in limited overhead if there is sufficient capacity, and only requires more execution time when it is actually important to run it to ensure there will be no service failures.

It is also important to note that the impact analysis is a step that is executed when new services are instantiated, a process which currently takes multiple days, making the process less sensitive to time constraints. A final consideration is that applying network impact analysis is more important for smaller client deployments, where the quality of the networks tends to be lower. In these cases, the number of workflows is usually limited to ± 100 workflows. These factors make the long algorithm execution duration not prohibitive for the algorithm's use.

Despite these considerations, it is desirable to improve the scalability of the SRNAID algorithm to make it possible to obtain faster feedback. To improve the scalability of the approach when larger networks are used, it is possible to

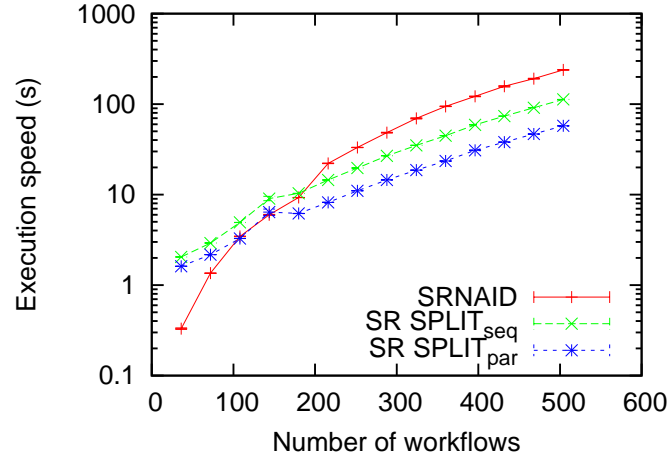


Figure 6.14: The execution speed of the SRNAID algorithm compared to the execution speed of the SR SPLIT algorithm. The algorithm parameters are chosen based on the results of the quality evaluation: $OF_{root} = 1$, and $OF_{NC} = OF_{VoIP} = OF_{video} = 0.3$. The edge capacity throughout the network is heavily constrained (limited to 10KBps per edge) to illustrate worst case performance. 40 iterations for each data point.

segment the input network, decomposing the problem network in separate networks, and statically dividing overlapping components between the networks. For the evaluation scenario, this can be easily done by considering both buildings within the simulation separately, leaving only the cloud uplink as a shared edge between both subproblems. Due to the symmetry in the problem model, the capacity of this shared edge can be divided equally between both subproblems. We will refer to this algorithm as the SR SPLIT algorithm. Two variants of the SR SPLIT algorithm can be discerned: SR SPLIT_{seq}, where the SRNAID algorithm is invoked sequentially on the subproblems, and the SR SPLIT_{par} algorithm where both algorithms are executed in parallel on separate computation nodes. This approach is generally applicable to all input networks, but simulations may be needed to determine an optimal division of shared edges between subproblems if the input network is not symmetrical.

Figure 6.14 compares the execution speed of the SRNAID algorithm with that of SR SPLIT_{seq} and SR SPLIT_{par}. For this scenario, both SR SPLIT algorithms execute faster than the SRNAID algorithm, with the SR SPLIT_{par} algorithm needing about half the time to run compared to the SR SPLIT_{seq} algorithm. The latter is to be expected as the network was split into two parts.

The impact on the total network requirement, calculated as discussed in Section 6.7.1 is negligible for this scenario: only 0.035% more network capacity is

needed. The impact on the shared edge itself is however larger: there, a capacity increase of 4.6% is observed, which implies that for these edges there will need to be 4.6% overprovisioning compared to the SRNAID approach. Splitting the network into more subnets further increases parallelism and reduces the subproblem size, and is expected to further increase the execution speed if needed. As in such a scenario there are more shared edges, the amount of resources needed by the algorithm to accept a service configuration will however further increase.

6.8 Conclusions

In this chapter, we discussed how resource sharing can be incorporated when determining impacts of service workflows on a network during service deployment by defining a model that allows service workflows to partially ignore other workflows. By hierarchically specifying groups of workflows, fine-grained control over the resource sharing can be achieved, increasing the quality of resulting configurations. We also specified a runtime conflict management algorithm that can be used to resolve resource conflicts when resource conflicts occur during runtime. By assigning a priority to different workflows, the successful execution of important workflows can be ensured. We found that, when suitable hierarchy parameters are determined through simulation, the presented hierarchical SRNAID algorithm requires $\pm 42\%$ and $\pm 52\%$ less resources than an approach without resource sharing for two evaluation use cases, without any workflow failures occurring during runtime. We also discussed how the scalability of the algorithm may be improved.

We focused on the application of the SRNAID algorithm as an admission filter, where it is used to determine whether services with high-availability requirements can be provided on a given network. In the future, the SRNAID approach could also be used for a what-if analysis, determining which additional hardware and network resources are required for providing a service, making it easier accurately predict the cost of implementing additional or new services. In future work, the presented model can also be adapted for resource allocation, determining which servers and devices are responsible for which services.

Acknowledgement

Hendrik Moens is funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). This work was carried out using the Stevin Supercomputer Infrastructure at Ghent University, funded by Ghent University, the Hercules Foundation and the Flemish Government – department EWI. We thank Televic Healthcare for their input on our evaluation use case.

Addendum

As mentioned in the conclusions, this model could be extended to be used for resource allocation. This is partially explored in Chapter 7, where resource allocation is discussed in network environments. The model in Chapter 7 uses a similar model to the model presented in this chapter for resource allocation. It does not however make use of a hierarchical structuring of network flows for performance reasons. The model in Chapter 7 could easily be adapted to include the presented hierarchies if the use case requires it.

References

- [1] Amazon. *Amazon EC2 Instance Types* [online]. Last accessed: February 2015. Available from: <http://aws.amazon.com/ec2/instance-types/>.
- [2] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. *Network-Aware Impact Determination Algorithms for Service Workflow Deployment in Hybrid Clouds*. In Proceedings of the 8th International Conference on Network and Service Management (CNSM 2012), pages 28–36. IEEE, oct 2012.
- [3] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows: Theory, Algorithms, and Applications*. Prentice Hall, New Jersey, 1993.
- [4] B. Awerbuch and T. Leighton. *Improved approximation algorithms for the multi-commodity flow problem and local competitive routing in dynamic networks*. In Proceedings of the 26th ACM Symposium on Theory of Computing (STOC 1994), pages 487–496. ACM, ACM, may 1994. doi:10.1145/195058.195238.
- [5] M. Pióro and D. Medhi. *Routing, Flow, and Capacity Design in Communication and Computer Networks*. The Morgan Kaufmann Series in Networking. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.
- [6] V. Kolar and N. B. Abu-Ghazaleh. *A Multi-Commodity Flow Approach for Globally Aware Routing in Multi-Hop Wireless Networks*. In Proceedings of the 4th Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM 2006), pages 308–317. IEEE, mar 2006. doi:10.1109/PERCOM.2006.3.
- [7] W. Szeto, Y. Iraqi, and R. Boutaba. *A multi-commodity flow based approach to virtual network resource allocation*. In Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM 2003), volume 6, pages 3004–3008. IEEE, dec 2003. doi:10.1109/GLOCOM.2003.1258787.
- [8] Y. L. Y. Liu, D. Tipper, and P. Siripongwutikorn. *Approximating optimal spare capacity allocation by successive survivable routing*. IEEE/ACM Transactions on Networking, 13(1):198–211, mar 2005. doi:10.1109/TNET.2004.842220.
- [9] L. Zhao, Y. Ren, M. Li, and K. Sakurai. *Flexible service selection with user-specific QoS support in service-oriented architecture*. Journal of Network and Computer Applications, 35(3):962–973, may 2012. doi:10.1016/j.jnca.2011.03.013.

- [10] J. Rolia, A. Andrzejak, and M. Arlitt. *Automating Enterprise Application Placement in Resource Utilities*. In M. Brunner and A. Keller, editors, *Self-Managing Distributed Systems*, volume 2867 of *Lecture Notes in Computer Science*, pages 118–129. Springer Berlin Heidelberg, 2003. doi:10.1007/978-3-540-39671-0_11.
- [11] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici. *A scalable application placement controller for enterprise data centers*. In *Proceedings of the 16th International Conference on World Wide Web (WWW 2007)*, pages 331–340. ACM, may 2007. doi:10.1145/1242572.1242618.
- [12] C. Adam and R. Stadler. *Service Middleware for Self-Managing Large-Scale Systems*. *IEEE Transactions on Network and Service Management*, 4(3):50–64, dec 2007. doi:10.1109/TNSM.2007.021103.
- [13] F. Wuhib, R. Stadler, and M. Spreitzer. *Gossip-based Resource Management for Cloud Environments*. In *Proceedings of the 6th International Conference on Network and Service Management (CNSM 2010)*, pages 1–8. IEEE, oct 2010. doi:10.1109/CNSM.2010.5691347.
- [14] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. *Cost-Effective Feature Placement of Customizable Multi-Tenant Applications in the Cloud*. *Journal of Network and Systems Management*, 22(4):517–558, oct 2014. doi:10.1007/s10922-013-9265-5.
- [15] H. Moens, J. Famaey, S. Latré, B. Dhoedt, and F. De Turck. *Design and Evaluation of a Hierarchical Application Placement Algorithm in Large Scale Clouds*. In *Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011)*, pages 137–144. IEEE, may 2011. doi:10.1109/INM.2011.5990684.
- [16] C. Low. *Decentralised Application Placement*. *Future Generation Computer Systems*, 21(2):281–290, feb 2005. doi:10.1016/j.future.2003.10.003.
- [17] D. Carrera, M. Steinder, I. Whalley, J. Torres, and E. Ayguadé. *Utility-based placement of dynamic web applications with fairness goals*. In *Proceedings of the 11th Network Operations and Management Symposium (NOMS 2008)*, pages 9–16. IEEE, apr 2008. doi:10.1109/NOMS.2008.4575111.
- [18] J. Brønsted, K. M. Hansen, and M. Ingstrup. *Service Composition Issues in Pervasive Computing*. *IEEE Pervasive Computing*, 9(1):62–70, jan 2010. doi:10.1109/MPRV.2010.11.
- [19] K.-T. Tran, N. Agoulmine, and Y. Iraqi. *Cost-effective complex service mapping in cloud infrastructures*. In *Proceedings of the 13th Network Operations*

- and Management Symposium (NOMS 2012), pages 1–8. IEEE, apr 2012. doi:10.1109/NOMS.2012.6211876.
- [20] C. Develder, J. Buysse, M. De Leenheer, B. Jaumard, and B. Dhoedt. *Resilient network dimensioning for optical grid/clouds using relocation*. In Proceedings of the 2012 IEEE International Conference on Communications (ICC 2012), pages 6262–6267. IEEE, jun 2012. doi:10.1109/ICC.2012.6364981.
- [21] B. Sansó, M. Gendreau, and F. Soumis. *An algorithm for network dimensioning under reliability considerations*. Annals of Operations Research, 36(1):263–274, 1992. doi:10.1007/BF02094333.
- [22] S. Verma, R. K. Pankaj, and A. Leon-Garcia. *Call admission and resource reservation for guaranteed quality of service (GQoS) services in internet*. Computer Communications, 21(4):362–374, apr 1998. doi:10.1016/S0140-3664(97)00169-2.
- [23] S. Sharafeddine. *Capacity assignment in multiservice packet networks with soft maximum waiting time guarantees*. Journal of Network and Computer Applications, 34(1):62–72, 2011. doi:10.1016/j.jnca.2010.09.004.
- [24] R. L. Carter and M. E. Crovella. *Measuring bottleneck link speed in packet-switched networks*. Performance evaluation, 27(28):297–318, oct 1996. doi:10.1016/0166-5316(96)00036-3.
- [25] N. Hu and P. Steenkiste. *Evaluation and Characterization of Available Bandwidth Probing Techniques*. IEEE Journal on Selected Areas in Communications, 21(6):879–894, aug 2003. doi:10.1109/JSAC.2003.814505.
- [26] M. Li, Y.-L. Wu, and C.-R. Chang. *Available bandwidth estimation for the network paths with multiple tight links and bursty traffic*. Journal of Network and Computer Applications, 36(1):353–367, jan 2013. doi:10.1016/j.jnca.2012.05.007.
- [27] *Scala 2.9.2* [online]. 2014. Available from: <http://www.scala-lang.org/>.
- [28] *IBM ILOG CPLEX 12.4* [online]. 2014. Available from: <http://www-01.ibm.com/software/integration/optimization/cplex-optimizer>.

7

Customizable Function Chains: Managing Service Chain Variability in Hybrid NFV Networks

H. Moens and F. De Turck

Submitted to IEEE Transactions on Networking (ToN)

Network Functions Virtualization (NFV) is an upcoming paradigm where network functions are virtualized and split up into multiple building blocks that can be chained together to provide a network service. Often, a service chain can be allocated in different ways, making use of different physical or virtual Network Functions (NFs), and resulting in varying quality of service and deployment costs. In this chapter, we present Customizable Function Chains (CFCs) which model the services within a service chain and their variability, and present algorithms that can be used to deploy these services on NFV-based networks, making it possible to take service chain variability into account during the management of these service chains. This chapter builds on the Feature-Based Binary (FBB) approach presented in Chapter 2 and the feature placement algorithm presented in Chapters 3 and 4, and applies these concepts to NFV-based networks. The network model presented in this chapter bears resemblance to that presented in Chapter 6. The objective of the model differs however: the model in Chapter 6 is intended to be used as a request access filter, while the model presented in this chapter is used to allocate resources once requests have been accepted within the network.

7.1 Introduction

Using Network Functions Virtualization (NFV), Network Functions (NFs) can be migrated from costly hardware appliances to dynamically allocated virtualized instances deployed on generic servers using cloud technologies. This migration from Physical Network Functions (PNFs) to Virtual Network Functions (VNFs) increases network flexibility and scalability, as these virtualized instances can be instantiated and scaled on-demand using cloud scaling technologies. A Service Function Chain (SFC) [1] describes the various NFs and how they interact to provide a complete network service. While there are many similarities to cloud resource allocation, the NFV architecture is designed to be used within entire service provider networks, and not just within datacenters. In datacenters, high-capacity and high-speed networks are used to interconnect servers, making the specifics of the underlying network less important. In NFV deployments in networks outside of the datacenter, the importance of network constraints such as bandwidth and latency however increases. This is particularly important when NFs can be executed in multiple locations in the network, e.g. in multi-cloud scenarios, where multiple datacenters are present, or when physical devices spread throughout the network are part of the provided network service.

In practice, the more expensive dedicated hardware often performs faster and more efficiently than virtualized instances, even though the latter are more flexible. As dedicated hardware is currently widely deployed, it is likely that hybrid NFV deployments will be common, where part of the NFs are provided by PNFs while others are provided using VNF instances. This also makes it possible to use an approach analogous to a cloud burst: in an “NFV burst”, a base load is handled by physical hardware (the private cloud in a cloud burst scenario), while variation in load is handled by dynamically instantiating VNFs (the public cloud in a cloud burst scenario). This approach is illustrated in Figure 7.1. For these reasons, NFV management systems should support hybrid NFs that can be instantiated using both VNFs and PNFs.

When multiple PNFs or VNFs provide a similar functionality, there may be slight functional or qualitative differences. A software router or firewall may e.g. support newer protocol versions or more modern functionality than older, physical devices, but they may also be slower. In some cases, the functionality of NFs may also overlap. It may e.g. be possible to configure a Deep Packet Inspection (DPI) NF to offer firewall functionality. For some SFCs these differences could be more constraining than for others, meaning they must always make use of specific NFs, while other SFCs definitions could be more flexible, allowing for the use multiple different physical or virtual NFs with varying quality characteristics. This flexibility makes it possible to choose the most cost-efficient service chain configuration at runtime. Therefore it is important to model these potential variations, and take them

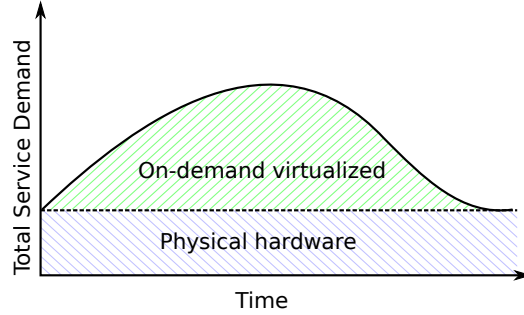


Figure 7.1: An NFV burst scenario: physical hardware is fully utilized by a base load, while spillover is handled by utilizing on-demand VNFs.

into account during the resource allocation process. This also makes it possible to model a degraded, lower quality fallback version of the SFCs for when service failures occur.

To manage NFV services that support both PNFs and VNFs, and to model service chain variability, a network and service-aware NFV management system must be developed. This management system should allow for the existence of both physical and virtual NFs, and should also take into account SFC variability. The management system can then make use of this information to minimize costs resulting from resource utilization and service failures.

In this chapter, we introduce Customizable Function Chains (CFCs), an extension to SFCs that takes service chain variability into account, and define a management approach that is capable of deploying CFCs in hybrid NFV networks. To model CFC variability, we make use of Software Product Line Engineering (SPLE) [2] concepts. SPLE techniques are often used to develop highly customizable software, of which multiple variants can exist. Using this approach, the software is modeled as a collection of features, that can be included or excluded. By selecting and deselecting features, different software variants can be created. Thus, a CFC not only specifies the used NFs and their interconnections, but also a feature model describing the relations between the NFs. To allocate CFCs on a network, we present a Customizable Function Chain Placement (CFC-P) model, which we compare to a Service Function Chain Placement (SFC-P) model which does not take variability into account. We then evaluate both models in a simulated service provider network.

In the next section, related work is discussed. In Section 7.3 we discuss an architecture for hybrid NFV resource management. Next, in Section 7.4 we describe how CFCs can be modeled and specified. We describe a model for CFC resource allocation in Section 7.5. In Sections 7.6 and 7.7 we discuss the evaluation setup and results. Finally, in Section 7.8 we state our conclusions.

7.2 Related Work

Resource allocation in NFV networks has some similarities to application placement approaches used within datacenters and clouds [3], specifically to network-aware application placement. Multiple publications [4–10] focus on either allocating collections of Virtual Machines (VMs), or on adding network-awareness to data-center resource management algorithms. These works, however, focus specifically on Infrastructure-as-a-Service (IaaS) clouds where VMs are allocated within data-center networks, meaning that often only datacenter-specific network topologies and hardware are considered. Therefore, these approaches are not suited for Wide Area Network (WAN) deployments, where a mix of general purpose hardware and dedicated hardware is present. This chapter contrasts with general datacenter management and application placement approaches by its focus on NFV resource allocation without any restrictions on the underlying network topologies, and its support for heterogeneous hardware. We define service chains that can contain NFs, which are managed by the service provider, and VMs, which are managed by clients. In addition, we also focus on how variability of the deployed service chains can be modeled and managed, making it possible to specify generic network services that can be implemented in multiple different ways at runtime.

SFC-P and CFC-P are also related to the problem of virtual network embedding in software defined networks [11]. Virtual network embedding focuses on how virtual network requests, in the form of a collection of VMs and their interconnections can be deployed on physical networks. In this chapter we extend this embedding approach by defining a model that makes it possible to specify both VM requests and service requests, the latter resulting in service provider managed services that may be shared between multiple tenants. We also incorporate the notion of hybrid networks containing both physical devices offering services and virtualized services. Similarly, [12] focuses on deployment of virtual network functions in pure NFV environments, while we also consider a hybrid environment where dedicated hardware for providing services is present. The work presented in this chapter further contrasts with existing network embedding approaches as we also define the concept of service chain variability, making it possible to reconfigure services at runtime to reduce management costs or to handle service degradations when insufficient resources are present.

The NFV Management and Orchestration specification [13] supports the definition of both PNFs and VNFs, respectively using Virtualised Network Function Descriptor (VNFD) and Physical Network Function Descriptor (PNFD) elements in the VNF Forwarding Graph Descriptor (VNFFGD). While this approach supports the specification of services containing both PNFs and VNFs, there is no generic NF description element that defines the common functionality of network functions, meaning that no hybrid NFs can be specified. The specification also has no support

for the specification of service chain customizability. In this chapter, we focus on how hybrid NFs and service chain variability can be modeled and managed.

SPLE can be used to develop customizable Software-as-a-Service (SaaS) applications [14–18], but the focus of these works is generally on how these applications can be developed and configured, and on not how they can be managed. In this chapter, we use these principles to model runtime service chain variations. Using this approach it is possible to specify multiple valid service chain configurations. This enables the management system to determine the included NFs at runtime, making it possible to reconfigure service chains to reduce costs or to manage service degradations.

This work extends our previous work [19], where we introduced an approach for SFC resource allocation in hybrid NFV networks. While this approach makes it possible to support both PNFs and VNFs within NFV networks, this only works when both services are identical, limiting its applicability. In this chapter, we introduce CFCs, which make it possible to specify the variability of service chains, solving this limitation. To manage CFCs, we introduce CFC-P, after which we compare the performance of CFC-P with that of SFC-P, which is based on the model from [19]. In addition, we present the architectural framework in which SFC-P and CFC-P can be used, and show how hybrid NFs and CFCs can be specified.

In our previous work on the management of customizable SaaS applications [20–22], we have developed an SPLE based approach for developing and managing customizable multi-tenant SaaS applications. The work focuses on managing small numbers of highly customizable applications within the context of a single cloud datacenter, which is achieved by splitting them up into components, and managing them using a Service-Oriented Architecture (SOA). We however focused on the allocation of these multi-tenant components within a single cloud datacenters, where devices are homogeneous, and where the underlying network is less important due to the high network capacities and low latencies. In this work, we use also use an SPLE based approach to model CFC variability, which we use to support customizability within a single service chain, but we add support for network-awareness, which is needed in WAN networks. Furthermore, we generalize the approach making it possible to deploy both VMs and NFs on networks containing hosts and physical network devices.

7.3 NFV resource management architecture

Figure 7.2 shows a high-level overview of the NFV resource management system architecture, which consists of two subsystems. A CFC provisioning system is responsible for allocating and managing CFCs by determining where and how CFCs are allocated, after which it configures the NFs and routing system. A CFC runtime system is responsible for ensuring network packets are routed to the correct

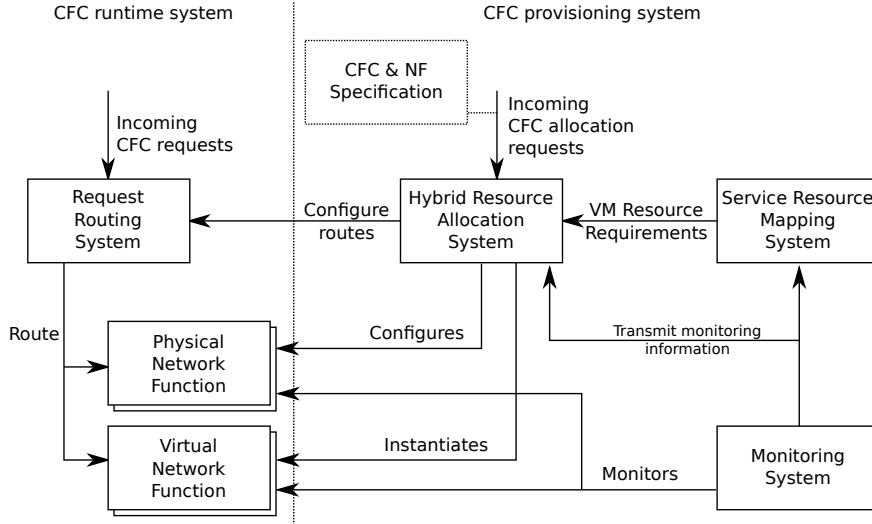


Figure 7.2: A high level architectural overview of how a hybrid NFV resource management system can be constructed.

NFs. Both subsystem respond to different network requests: CFC requests are requests for the network service that must be routed to the correct NFs by the CFC runtime system, while CFC allocation requests are requests for the allocation and instantiation of a new network service that are handled by the CFC provisioning system.

CFC requests are handled using the CFC runtime system, which makes use of the following components:

- A **request routing system** must be used to route incoming CFC requests and subsequent network messages to the correct NFs. Existing Software-Defined Networking (SDN) controllers such as OpenFlow [23] may be used for this.
- The **physical network functions** implement the functionality of a NF using physical hardware.
- The **virtual network functions** implement the functionality of a NF using virtualized service instances, which are allocated using VMs deployed within a datacenter.

Incoming CFC allocation requests are handled by the CFC provisioning system. This provisioning system which is responsible for determining how CFCs are deployed, and allocating the requested NFs using physical or virtual NF instances. In Section 7.4 we specify how CFCs and NFs should be modeled. Once the

management system receives CFC allocation requests, they are handled by the following components:

- The **hybrid resource allocation system** is a central component in the management architecture. This component receives incoming CFC requests and allocates the required NFs on the network. Depending on the CFC variability and the definition of the used NFs, the NFs may be allocated either using PNFs or VNFs. CFC-P is used by this component to determine how and on which PNFs and VNFs the CFCs are deployed.
- The **service resource mapping system** is used to maintain a mapping between *server resources* and *service resources*. NFs provide resources that are specific to the provided service, such as e.g. the number of requests per second (rps) that the service can provide. While PNFs offer such service resources, this is harder for VNFs, as VM resource requirements are expressed in terms of server resources such as CPU and memory. To correctly provision resources, it is important that the management system is aware of the amount of server resources that are needed to provide a given amount of service resources. Therefore, for every VNF an accurate mapping between server and service resources is needed. The service resource mapping system is responsible for storing this mapping and updating it by monitoring the resource use of deployed VNFs.
- A **monitoring system** is needed to monitor the performance of the various NFs and the network, ensuring the quality of the deployed CFCs is guaranteed, and enabling the hybrid resource allocation system to re-allocate resources when changes in resource demand or other problems are detected.

7.4 Modeling CFCs and NFs

A SFC defines a request for a complete network service which may be composed out of multiple NFs, and defines a service graph which shows how the NFs interconnect. To allow SFCs to define a large variety of network services, we assume that a client requesting it may also provide VMs that are part of the service chain. These client VMs must then be allocated on a host as part of the SFC allocation process. An SFC therefore contains a collection of *network asset requests* which can either be for VMs, or for NFs. These NF requests should then in turn be allocated using either a PNF or a VNF.

The key difference between client VM requests and NF requests is that a VM that is part of an SFC is managed by the client who requests the service chain, while a VM that provides a VNF is provided and managed by the service provider. The provider-hosted VNFs can therefore be shared between multiple clients. Requested

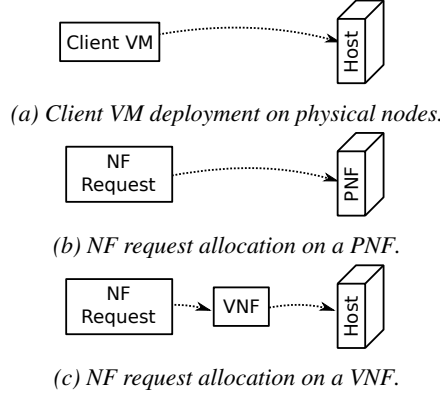


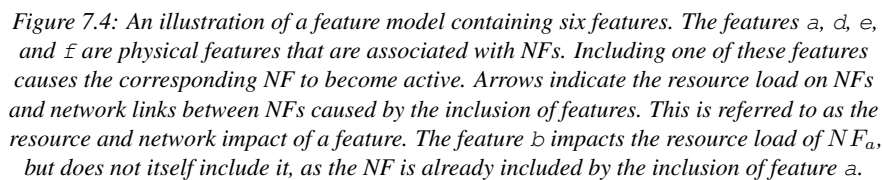
Figure 7.3: Network asset allocation approaches.

Relation	Definition
Mandatory (p, c)	If parent feature p is selected, child feature c must be included as well.
Optional (p, c)	The feature c may only be selected if p is selected.
Alternative ($p, \{c_1, c_2, \dots, c_n\}$)	When p is selected, exactly one of the features c_i must be selected.
Or ($p, \{c_1, c_2, \dots, c_n\}$)	When p is selected, at least one of the features c_i must be selected.
Requires (f_1, f_2)	For feature f_1 to be included, f_2 must be included as well.
Conflicts (f_1, f_2)	If f_1 is included, f_2 must not be included.

Table 7.1: The relation types used to structure feature models.

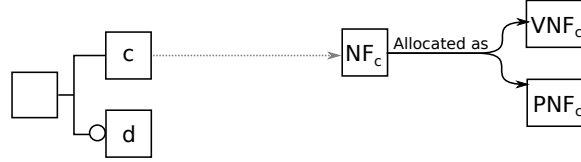
network assets may be allocated in three different ways depending on their type (illustrated in Figure 7.3): (1) client VMs must be deployed on physical server nodes, (2) NF requests can be allocated on PNF nodes, and (3) alternatively NF requests can be allocated on a VNF instance which is itself allocated on a physical server.

A CFC extends the SFC concept by defining a *feature model* instead of a service graph, making it possible to specify multiple alternative versions of a network service. A feature model defines a collection of features and their relations. A feature represents a single functionality which can be included in the service chain. The service chain itself is then composed by including and excluding features while taking their relations into account. This makes it possible to create multiple related service chains with differing functionalities using a common model. The

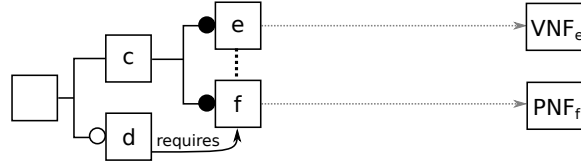


When modeling CFCs using a feature model, we assume that every requested asset is associated with a feature within the feature model. The feature model will then be used to determine the valid NF and VM combinations, and their network and resource demands. The inclusion of a feature may result in the inclusion of specific asset requests, impact the resource load of NFs and VMs, or result in an increase of the network demand between NFs and VMs. Within the feature model we discern two types of features: physical features and virtual features. A physical feature is directly associated with a specific asset request. Including the physical feature will therefore cause the associated NF or VM to be instantiated. Virtual features are not directly associated with a NF, but may still impact the configuration of NFs and VMs, and therefore their resource demand. They may also be used to add structure to the feature model. Figure 7.4 shows an illustrative feature model containing six features that map to four different NFs. Any SFC can be trivially converted into a CFC by creating a feature model containing all asset requests and only mandatory relations, making CFCs a superset of SFCs.

A hybrid NF can be specified within a CFC using two different approaches. Either a single NF, which can be allocated on either a VNF or on a PNF instance,



(a) A feature implemented using a hybrid NF. The CFC model is unaware of the different implementation methods.



(b) A feature implemented by explicitly linking features within the feature model to a VNF or a PNF using an **Alternative** relation. The CFC is aware of the existence of different NF implementations and may e.g. specify features that are dependent on one implementation.

Figure 7.5: An illustration of how hybrid NFV scenarios can be modeled using CFCs.

can be defined. Alternatively, separate VNF and PNF instances may be defined, which are explicitly linked to the feature model using an **Alternative** relation. Using this model-based approach, no overarching hybrid NF is needed. Both approaches are illustrated in Figure 7.5. While both approaches can be used to represent hybrid NFs, they have different advantages and disadvantages. The feature model based approach ensures the CFC is aware of the various hardware and software NF implementations. This increases the model complexity, and consequently the number of possible configurations, but also makes it possible to define features that depend on a specific NF implementation. Using hybrid NFs, the complexity of the feature model decreases, but the model itself does not contain any information pertaining to the various NF implementations. Consequently, hybrid NFs are preferred when the physical and virtual implementations of the NF behave similarly and offer identical functionality, while the more complex feature model based approach may be needed otherwise.

7.5 Function chain resource allocation

We construct the CFC-P model in two steps. First, we describe an approach for allocating SFCs in hybrid NFV networks. Subsequently, we extend the SFC-P model and add awareness of service chain customizability by adding a feature model.

Symbol	Description
G	The graph $G = (N, E)$ representing the network.
N	The nodes within the network. This collection can be partitioned into a set of computational nodes N^c and a set of PNF nodes N^p .
E	The edges within the network.
E_n^{in}	The edges that are incoming in node $n \in N$.
E_n^{out}	The edges that are outgoing from node $n \in N$.
$C(e)$	The bandwidth capacity of an edge $e \in E$.
\mathcal{NF}	The collection of all NFs that exist within the model.
Γ^n	The resource types that are available on a node $n \in N$, or the resource types provided by a NF $n \in \mathcal{NF}$.
$C^\gamma(n)$	The resource capacity of a node or NF n . For computational nodes, $\gamma \in \Gamma^{VM}$. For PNF nodes offering an NF type s , $\gamma \in \Gamma^s$. A VNF of an NF type s provides $C^\gamma(n)$ resources of types $\gamma \in \Gamma^s$.
NC_n	The cost of using a node $n \in N$ during resource allocation.
NC_n^γ	The cost of using resources of type $\gamma \in \Gamma^n$ on node $n \in N$.

Table 7.2: Model parameters: network and service specifications.

7.5.1 General model parameters

Both the SFC-P and CFC-P models require information about the topology of the network, and information about the PNFs and server nodes. These common parameters are shown in Table 7.2. The network is represented as a graph $G = (N, E)$, consisting of a collection of nodes N that represent physical network nodes and edges E between these nodes. We assume that these edges can be either bidirectional or unidirectional. Incoming and outgoing edges from a node $n \in N$ are represented by E_n^{in} and E_n^{out} respectively, and every edge e has a capacity $C(e)$. A node can be any device in the network, such as network switches, computational nodes on which VMs can be deployed, or PNFs such as e.g. routers, hardware firewalls, or access points. We make a distinction between computational nodes, contained in N^c , and PNF nodes N^p that offer a specific NF.

A collection of NFs that can be provisioned using either PNF nodes or VNFs allocated on computational nodes must be defined. This set, \mathcal{NF} should contain all of the NFs that can be allocated. Nodes n offer multiple types of resources, which are contained in the set Γ^n . The types of resources contained in this set are dependent on the type of the node: if a node is a computational node it will offer server resources, while if it is a PNF node it will offer service-specific resources. Similarly, for any NF $s \in \mathcal{NF}$ the offered service-specific resource types are contained in the set Γ^s .

Every node $n \in N$ has a resource capacity $C^\gamma(n)$ for each of its offered resource types $\gamma \in \Gamma^n$. For every node, a usage cost NC_n , and a node resource use

cost NC_n^γ can be specified. The former cost is incurred whenever a node n is used in an allocation, while the latter is incurred for every resource used on a node (e.g. a cost incurred for every CPU core used in a cloud node).

7.5.2 SFC-P

To be able allocate SFCs, they must first be specified. The required parameters for this are shown in Table 7.3. A collection \mathcal{SFC} contains all service chains that must be allocated within the network. A service chain $C \in \mathcal{SFC}$ consists of a collection of asset requests, where the requested asset can either be an NF or a VM. The NF and VM requests of an SFC C are contained in the $R^{NF}(C)$ and $R^{VM}(C)$ collections respectively. All asset requests in an SFC are grouped in the collection $R(C) = R^{VM}(C) \cup R^{NF}(C)$. The collection \mathcal{R} contains all asset requests in the model, aggregated over all of the service chains in \mathcal{SFC} .

The resource demand of an asset request r is represented by $D^\gamma(r)$, and is specified for all resources $\gamma \in \Gamma^r$ that are provided by the requested asset. The network demand between requested VMs and NFs is represented by $D(r_1, r_2)$ for any two asset requests $(r_1, r_2) \in \mathcal{R}^2$ that are part of the SFC.

When an SFC C can not be allocated because there is insufficient network or server capacity it may not be allocated. In this case, a service failure cost FC_C is incurred, which represents a management cost associated with this failure.

Symbol	Description
\mathcal{SFC}	The SFCs that must be allocated on the network.
$R(C)$	The asset requests that are part of service chain $C \in \mathcal{SFC}$. $R^{VM}(C)$ contains only the VM requests, while $R^{NF}(C)$ contains the NF requests. $R(C) = R^{VM}(C) \cup R^{NF}(C)$.
\mathcal{R}	A collection containing all of the asset requests of all service chains in \mathcal{SFC} . \mathcal{R}^{VM} and \mathcal{R}^{NF} contain only the VM and NF requests respectively.
$D^\gamma(r)$	The resources of type γ needed for a request r . This request can either be an NF request or a VM request.
$D(r_1, r_2)$	The network demand between asset requests r_1 and r_2 . The collection D contains all pairs of asset requests between which there is network demand.
FC_C	The cost of failing to allocate any of the services of a service chain $C \in \mathcal{SFC}$.

Table 7.3: Model parameters: SFC-P.

7.5.2.1 Resource management constraints

We define a binary decision variable, M_n^r , that determines whether an asset request $r \in \mathcal{R}$ is allocated on a node $n \in N$. If a service request may not be allocated on a given node, $M_n^r = 0$. An integer decision variable IC_n^s specifies the number of times that an NF $s \in \mathcal{NF}$ is instantiated on a computational node $n \in N^c$. We use an integer variable as we assume that it is possible for there to be multiple instances of the same VNF on a single physical node. If this is not desired, an upper bound of 1 can be specified for the IC_n^s variable preventing this. Using these decision variables, and the parameters specified previously, a capacity constraint can be defined. Equation (7.1) shows the capacity constraint, which specifies that the total resource use $U(n, \gamma)$ for all nodes $n \in N$ and resources $\gamma \in \Gamma^n$ should be less than the resource capacity $C^\gamma(n)$ of the node. The total resource use is computed as shown in Equation (7.2), and is specified separately for server nodes and physical nodes. For server nodes, the total amount of used resources is composed out of the resources used for handling VM requests, represented as U^{VM} and shown in Equation (7.3), and out of the resources used to allocate VNFs, represented as U^{VNF} (shown in Equation (7.4)). For physical nodes, only the resources used by the allocated NFs need to be taken into account, as shown in Equation (7.5).

$$\forall n \in N : \forall \gamma \in \Gamma^n : U(n, \gamma) \leq C^\gamma(n) \quad (7.1)$$

$$U(n, \gamma) = \begin{cases} U^{VM}(n, \gamma) + U^{VNF}(n, \gamma), & \text{if } n \in N^c. \\ U^{PNF}(n, \gamma), & \text{if } n \in N^p. \end{cases} \quad (7.2)$$

$$U^{VM}(n, \gamma) = \sum_{r \in \mathcal{R}^{VM}} M_n^r \times D^\gamma(r) \quad (7.3)$$

$$U^{VNF}(n, \gamma) = \sum_{s \in \mathcal{NF}} IC_n^s \times D^\gamma(s) \quad (7.4)$$

$$U^{PNF}(n, \gamma) = \sum_{r \in \mathcal{R}^{NF}} M_n^r \times D^\gamma(r) \quad (7.5)$$

It is important to ensure that the IC_n^s decision variables take on the correct values. This is done using Equation (7.6), which ensures that the number of instances on a node, IC_n^s , offers sufficient service resources for the VNFs that are allocated on the node. This equation makes use of the total available service resources $T^\gamma(s, n)$ and the total needed service resources $N^\gamma(s, n)$ on the node. The expressions for $T^\gamma(s, n)$ and $N^\gamma(s, n)$ are shown in Equations (7.7) and (7.8) respectively.

$$\forall n \in N : \forall s \in \mathcal{NF} : \forall \gamma \in \Gamma^s : T^\gamma(s, n) \geq N^\gamma(s, n) \quad (7.6)$$

$$T^\gamma(s, n) = IC_n^s \times C^\gamma(s) \quad (7.7)$$

$$N^\gamma(s, n) = \sum_{r \in \mathcal{R}^{NF}} M_n^r \times D^\gamma(r) \quad (7.8)$$

When there are insufficient resources, it may be possible that not every SFC will be allocated. We define the binary decision variable Φ^C which takes on value 1 if the service chain $C \in \mathcal{SFC}$ is allocated and 0 otherwise. Equation (7.9) is used to ensure that a function chain is allocated when all of its requests are.

$$\forall C \in \mathcal{SFC} : \forall r \in R(C) : \sum_{n \in N} M_n^r = \Phi^C \quad (7.9)$$

7.5.2.2 Network constraints

To add network-awareness to the model, the flow between two requests $(r_1, r_2) \in \mathcal{R}^2$ over the edges $e \in E$ of the network is modeled using a collection of binary flow decision variables $F(e, r_1, r_2)$. If $F(e, r_1, r_2) = 1$, the edge e is used for the flow (r_1, r_2) , otherwise the value of this decision variable must be 0. These flow variables are subject to a flow conservation constraint, shown in Equation (7.12). For all nodes except the source and sink nodes, the incoming flow must equal the outgoing flow. For the source node, the flow must exceed the out flow, while for the sink node the opposite holds.

$$OUT(n, r_1, r_2) = M_n^{r_2} + \sum_{e \in E_n^{out}} F(e, r_1, r_2) \quad (7.10)$$

$$IN(n, r_1, r_2) = M_n^{r_1} + \sum_{e \in E_n^{in}} F(e, r_1, r_2) \quad (7.11)$$

$$\forall (r_1, r_2) \in D : \forall n \in N : \quad (7.12)$$

$$OUT(e, r_1, r_2) = IN(e, r_1, r_2)$$

Two additional constraints, shown in Equations (7.13) and (7.14) are added to make sure there is no incoming flow in source nodes or outgoing flow in sink nodes. Finally, a capacity constraint, expressed in Equation (7.15), is needed to ensure that the total flow over an edge does not exceed the available edge capacity.

$$\forall (r_1, r_2) \in D : \forall n \in N : M_n^{r_2} + \sum_{e \in E_n^{out}} F(e, r_1, r_2) \leq 1 \quad (7.13)$$

$$\forall (r_1, r_2) \in D : \forall n \in N : M_n^{r_1} + \sum_{e \in E_n^{in}} F(e, r_1, r_2) \leq 1 \quad (7.14)$$

$$\forall e \in E : \sum_{(r_1, r_2) \in D} F(e, r_1, r_2) \times D(r_1, r_2) \leq C(e) \quad (7.15)$$

7.5.2.3 Optimization objective

The objective of the model is to minimize the total cost of the SFC allocations. This cost is composed of a resource utilization cost, CR , and a management cost associated with the failure to provision resources for an SFC, CF . Combining both costs, the optimization objective can be specified as shown in Equation (7.16).

$$\min (CR + CF) \quad (7.16)$$

The resource utilization cost is composed out of two costs: a static node use cost and a linear node resource use cost. The static node use cost is incurred whenever a node is used to allocate any asset. Therefore, this cost can be e.g. utilized to represent the energy cost of instantiating a node. By contrast, the linear node resource cost incurs a cost for every resource used on the node. This cost therefore increases as the resource utilization on a node increases, and may e.g. be used to incur a cost for every CPU core used in a datacenter. The complete utilization cost computation is expressed in Equation (7.17). In this definition, the binary decision variable U_n is used to determine whether a node n is used. Equation (7.18) ensures that U_n takes on value 1 as soon as anything is allocated on the node n .

$$CR = \sum_{n \in N} U_n \times NC_n + \sum_{\gamma \in \Gamma^n} U(n, \gamma) \times NC_n^\gamma \quad (7.17)$$

$$\forall n \in N : \forall r \in \mathcal{R} : U_n \geq M_n^r \quad (7.18)$$

When insufficient network or hardware resources are present to accomodate all requests, some SFCs will fail to be allocated. Such service interruptions should be prevented, but if this is not possible, the cost of these failures should be minimized. Therefore, a service failure cost is associated with every SFC, and when it is not allocated this cost is incurred. Equation 7.19 shows how the total service failure cost can be computed. Note that this cost is a management cost, which may be defined by the management system based on SFC Service Level Agreements (SLAs) and previous SFC behavior (e.g. if the service has previously achieved high availability a short failure may be tolerable resulting in a lower cost of failure).

$$CF = \sum_{C \in \mathcal{SFC}} FC_C \times (1 - \Phi^C) \quad (7.19)$$

Symbol	Description
\mathcal{CFC}	The CFCs that must be allocated on the network.
\mathcal{F}_C	The feature model of a service chain $C \in \mathcal{CFC}$. This model contains a collection of features $f \in \mathcal{F}_C$ and also defines the relations between these features.
\mathcal{F}_C^{phy}	This collection contains the physical features in the feature model of service chain $C \in \mathcal{CFC}$, i.e. the features that are directly linked to an asset request.
$FI_{f_1}^\gamma(f_2)$	The feature resource impact $FI_{f_1}^\gamma(f_2)$ defines the impact on the resource requirements for feature f_2 when feature f_1 is included.
$FI_{f_1}(f_s, f_t)$	The feature network impact $FI_{f_1}(f_s, f_t)$ defines the impact on the network demand between source associated with feature f_s and the sink linked with feature f_t .
FC_C	The cost of failing to allocate the entire service chain $C \in \mathcal{CFC}$.
FC_C^f	The cost of failing to provide a single feature $f \in \mathcal{F}_C$ that is part of a service chain $C \in \mathcal{CFC}$.

Table 7.4: Model parameters: CFC-P.

7.5.3 CFC-P

The general concepts of SFC-P can be extended to support CFCs. The parameters specific to the CFC-P are shown in Table 7.4. The general parameters specifying network and service information, listed in Table 7.2, are still required. As the CFC-P extends the SFC-P, all of the SFC-P parameters from table Table 7.3 must also be defined, but they are no longer inputs to the model as they are either transformed into decision variables, or alternatively the associated values are determined based on the CFC-P parameters.

A collection of CFCs is provided in the set \mathcal{CFC} . Every service chain $C \in \mathcal{CFC}$ has an associated feature model \mathcal{F}_C . This feature model contains a collection of features, f , and defines the relations between these features. The resource demand for a feature, and the network demand between features is dependent on which features are included. When a feature f_1 is included, it may impact the resource demand for another feature f_2 for a resource type γ using a given feature resource impact $FI_{f_1}^\gamma(f_2)$. The feature may also impact the network demand between two features f_s and f_t , resulting in a feature network impact $FI_{f_1}(f_s, f_t)$.

To be able to extend the SFC-P model, all of its parameters must be defined. First, the feature models must be linked to asset requests. This is done by linking physical features to asset requests, as explained in Section 7.4 and illustrated in Figure 7.4. Physical features, contained in the set \mathcal{F}_C^{phy} , are linked to asset requests. For a physical feature $f \in \mathcal{F}_C^{phy}$, the matching asset request is defined as \hat{f} . Using this mapping, the collection of all asset requests can be determined:

Relation	Constraint
HardSelected (f)	$\Phi_f^C = \Phi^C$
Mandatory (f, c)	$\Phi_f^C = \Phi_c^C$
Optional (f, c)	$\Phi_f^C \geq \Phi_c^C$
Alternative ($f, \{c_1, c_2, \dots, c_n\}$)	$\Phi_f^C = \Phi_{c_1}^C + \Phi_{c_2}^C + \dots + \Phi_{c_n}^C$
Or ($f, \{c_1, c_2, \dots, c_n\}$)	$\Phi_f^C \leq \Phi_{c_1}^C + \Phi_{c_2}^C + \dots + \Phi_{c_n}^C$ $\forall i = 0 \dots n : \Phi_f^C \geq \Phi_{c_i}^C$
Conflicts (f_1, f_2)	$\Phi_{f_1}^C \leq 1 - \Phi_{f_2}^C$
Requires (f_1, f_2)	$\Phi_{f_1}^C \leq \Phi_{f_2}^C$

Table 7.5: The constraints associated with the various feature model relations.

$R(C) = \cup_{f \in \mathcal{F}_C^{phy}} \hat{f}$. The demand $D^\gamma(r)$ for requests and the network demand between these requests $D(r_1, r_2)$ are redefined as decision variables. The network demand decision variables are only defined if any feature exists that impacts the network between both requests. Finally, the chain fail cost FC_C is redefined as the cost of completely failing the service chain, while a newly defined feature fail cost FC_C^f represents the cost of failing to provide a specific feature f .

For every CFC, a valid feature configuration must be determined. This is done by adding constraints for all relations that are contained in the feature model. These constraints make use of binary decision variables Φ_f^C , which are used to express whether the feature f is included in service chain C . Table 7.5 shows how all relation types can be modeled. The **HardSelected**(f) relation is used to express that the CFC can only be included if the feature f is included. This is e.g. always used for the root of the feature model. When a feature should be included, but may fail in some cases, a feature fail cost should be assigned to it. As shown in Equation (7.20), CFC features may only be included when the CFC itself is included.

$$\forall C \in \mathcal{CFC} : \forall f \in \mathcal{F}_C : \Phi^C \geq \Phi_f^C \quad (7.20)$$

When physical features are included in the CFC, the linked asset request must be included. This is shown in Equation (7.21). The demand for these asset requests is dependent on the features that are included. Equation (7.22) shows how the feature resource impact is used to compute the demand for a given feature.

$$\forall C \in \mathcal{CFC} : \forall f \in \mathcal{F}_C^{phy} : \sum_{n \in N} M_n^f = \Phi_f^C \quad (7.21)$$

$$\begin{aligned} \forall C \in \mathcal{CFC} : \forall f \in \mathcal{F}_C^{phy} : \forall \gamma \in \Gamma^{\hat{f}} : \\ D^\gamma(\hat{f}) = \sum_{f' \in \mathcal{F}_C} \Phi_{f'}^C \times FI_{f'}^\gamma(f) \end{aligned} \quad (7.22)$$

7.5.3.1 Network constraints

The network demand, like the demand for asset requests, is dependent on the features included in the CFC. Equation (7.23) shows how the network demand can be computed using the feature inclusion variables.

$$\begin{aligned} \forall C \in \mathcal{CFC} : \forall (f_s, f_t) \in \left(\mathcal{F}_C^{phy} \right)^2 : \forall \gamma \in \Gamma^{\hat{f}} : \\ D(\hat{f}_s, \hat{f}_t) = \sum_{f \in \mathcal{F}_C} \Phi_f^C \times FI(\hat{f}_s, \hat{f}_t) \end{aligned} \quad (7.23)$$

By using a feature model, there can be multiple possible feature configurations for a CFC, resulting multiple possible sets of included asset requests. This complicates the SFC-P flow conservation constraint (Equation (7.12)) as it depends on the source and sink nodes being both included or both excluded. To resolve this, a binary decision variable $FA(r_1, r_2)$ is specified which expresses whether a flow between two asset requests r_1 and r_2 is active. Equation (7.24) ensures the FA variables take on the correct value. Using these variables, the flow conservation constraint can be redefined using Equations 7.25 and 7.26.

$$\begin{aligned} \forall C \in \mathcal{CFC} : \forall f \in \mathcal{F}_C^{phy} : \forall (s, t) \in (\mathcal{F}_C)^2 : \\ FI_f(f_s, f_t) \neq 0 \rightarrow \Phi_f^C \leq FA(\hat{f}_s, \hat{f}_t) \end{aligned} \quad (7.24)$$

$$\begin{aligned} \forall (r_1, r_2) \in D : \forall n \in N : \\ OUT(n, r_1, r_2) \leq IN(n, r_1, r_2) + (1 - FA(r_1, r_2)) \end{aligned} \quad (7.25)$$

$$IN(n, r_1, r_2) \leq OUT(n, r_1, r_2) + (1 - FA(r_1, r_2)) \quad (7.26)$$

7.5.3.2 Optimization objective

In CFC-P, the cost of failing specific features is separated from the cost of failing the entire service chain. Because of this, the failure cost is redefined as shown in Equation (7.27). This cost is composed of the cost for failing to allocate the entire

CFC (i.e. at least the **HardSelected** features), and the separate costs for failing to allocate individual features.

$$CF = \sum_{C \in \mathcal{SFC}} \left(FC_C \times (1 - \Phi^C) + \sum_{f \in \mathcal{F}_C} FC_C^f \times (1 - \Phi_f^C) \right) \quad (7.27)$$

7.6 Evaluation Setup

The SFC-P and CFC-P models were implemented as an Integer Linear Programming (ILP) using the IBM Ilog CPLEX ILP solver [24] and Scala [25]. Note that the CFC-P model as specified in this chapter contains multiplications of binary decision variables with binary and continuous decision variables, making the model quadratic instead of linear. We linearized these operations by replacing them with logically equivalent linear equations that make use of intermediary decision variables, ensuring both models are implemented as pure ILPs.

We compare the two optimal ILP-based algorithms implementing the SFC-P and CFC-P models, and also compare their performance with time-limited versions of the ILPs. While during a normal execution, CPLEX will continue until a provably optimal solution has been found using simplex and branch and bound algorithms, the time-limited CPLEX invocations stop the execution of the optimizer after a given execution duration, and return the highest quality feasible result. This results in a suboptimal result, but guarantees a given execution duration, making it more feasible to use these algorithms within a dynamic management system.

We consider a connectivity service where a source and sink node are interconnected. This connection may be direct, but optionally, the packets may be intercepted and analyzed. This analysis can be achieved using a firewall, or alternatively using a DPI service. As a DPI service requires more computational resources, a sampled DPI service where a fraction of the requests are analyzed using DPI and the other packets are analyzed using a regular firewall can be supported as well.

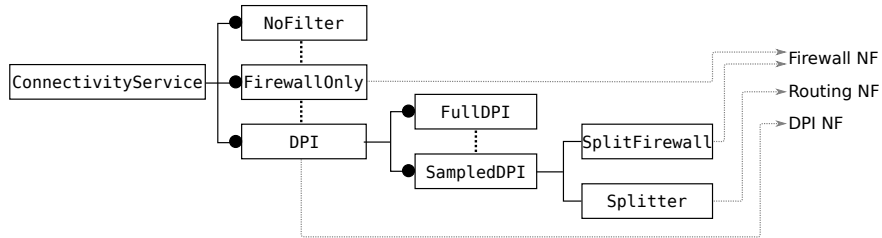


Figure 7.6: Connectivity service feature model and associated NFs.

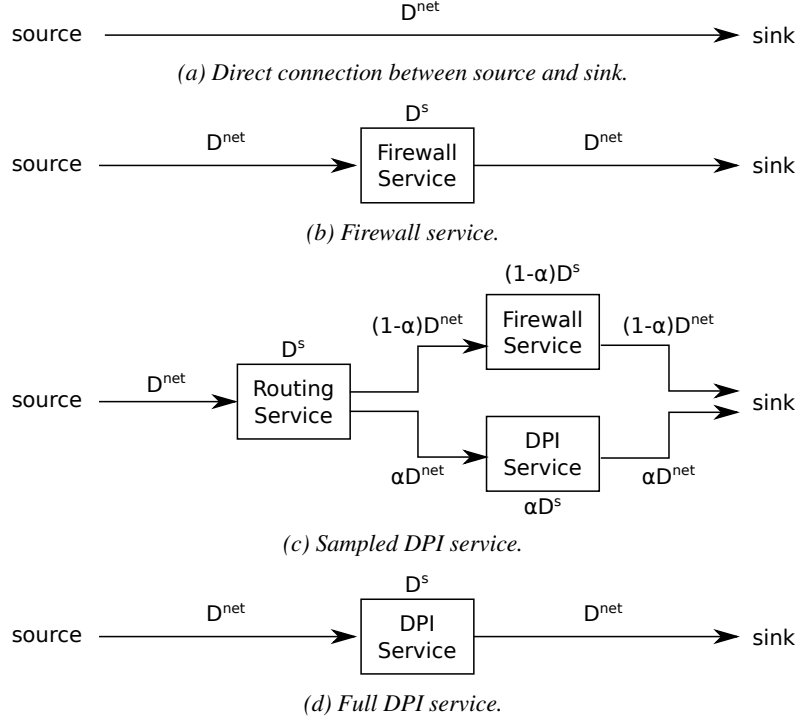


Figure 7.7: An illustration of the various possible connectivity service CFC configurations. D^{net} represents the network demand while D^s represents the service demand. α represents the fraction of requests that must be sent to the DPI service.

A feature model for the connectivity CFC is shown in Figure 7.6, together with the associated services. Figure 7.7 shows the alternative CFC deployments that this feature model results in. Service resource demand and service network demand are represented by D^{net} and D^s respectively, and can vary for different applications. Table 7.6 shows the resource and network impacts that the inclusion of features within the model result in. Note that, while the DPI feature results in the inclusion of the DPI service, the load on this service depends on how the service is instantiated (i.e. as a full DPI service or as a sampled DPI service). The structure of this model makes it possible to create *open variation points* [14], that leave some variability decisions undecided when the CFC is specified, allowing the management system to decide at runtime how the service chains are deployed. Using this approach, a service chain requiring only sampled DPI can be implemented using either the FullDPI feature or the SampledDPI feature when only the DPI feature is selected. This can potentially result in a reduction of the number of instances when there is sufficient capacity on existing DPI NFs, as then this DPI instance could be used instead of routing and firewall NF instances.

Feature	Impacts	Demand
FirewallOnly	FirewallOnly	D^s
SampledDPI	Splitter	D^s
	DPI	$\alpha \times D^s$
	SplitFirewall	$(1 - \alpha) \times D^{net}$
FullDPI	DPI	D^s
NoFilter	<i>source</i> \rightarrow <i>sink</i>	D^{net}
FirewallOnly	<i>source</i> \rightarrow FirewallOnly	D^{net}
	FirewallOnly \rightarrow <i>sink</i>	D^{net}
SampledDPI	<i>source</i> \rightarrow Splitter	D^{net}
	Splitter \rightarrow DPI	$\alpha \times D^{net}$
	Splitter \rightarrow SplitFirewall	$(1 - \alpha) \times D^{net}$
	DPI \rightarrow <i>sink</i>	$\alpha \times D^{net}$
	SplitFirewall \rightarrow <i>sink</i>	$(1 - \alpha) \times D^{net}$
FullDPI	<i>source</i> \rightarrow DPI	D^{net}
	DPI \rightarrow <i>sink</i>	D^{net}

Table 7.6: Resource and network impacts in the connectivity service scenario. D^s represents service demand (in rps), D^{net} represents network demand, and α represents the share of requests that require DPI.

Type	Selected	Excluded
Firewall	FirewallOnly	-
StrictFirewall	FirewallOnly	NoFilter
SampledDPI	DPI	NoFilter
FullDPI	FullDPI	NoFilter
StrictFullDPI	FullDPI, DPI	NoFilter, SampledDPI, FirewallOnly

Table 7.7: Evaluation CFCs.

We use multiple instances of the connectivity CFC within the evaluation scenario. Five different CFC types are defined. These types differ in the selection and exclusion of specific features, and are shown in Table 7.7. Selected features should be included in the CFC. If a valid feature model configuration exists where the selected features are not included, this model configuration is however permitted, while a cost of failing the selected feature is incurred. Excluded features may never occur in a valid configuration of the CFC. These types result in two firewalled connectivity services, one where the firewall may be disabled during short periods of time due to service overload, and one where this may not occur, and three DPI services, one using sampled, one with full DPI with fallbacks, and one without fallbacks. The source and sink nodes are chosen randomly from a set containing all of the edge nodes of the network and a VM using 4 cores and 8GB of memory. This causes the majority of services to represent an interconnection of two sites, while some will represent the connection of a remote site with a cloud-hosted VM. For CFC and feature failure, a random management cost in the set $\{8, 16, 32, 64, 128\}$ respectively $\{4, 8, 16, 32, 64\}$ is chosen, reflecting that failing to provide a degraded service is generally worse than failing to provide part of the service chain functionality, and that some services or features may be significantly more important than others resulting in much higher failure costs. Network demand is chosen uniformly between 10 and 1000Mbps, while service demand is chosen uniformly between 100 and 1000 rps. These demands are multiplied with a multiplier m , making it possible to change the application load throughout the evaluations. For sampled DPI, 10% of the requests are handled by the DPI NF.

The evaluation network is shown in Figure 7.8, and represents a small service provider, containing 27 edge nodes, 9 switches, 4 core routers, a hardware firewall, a small cloud datacenter, and three smaller edge datacenters which are located closer to the edge nodes. We also consider three variants of this network: one without edge clouds, one pure NFV variant (i.e. without PNFs) and a pure NFV network without edge clouds. The network hardware specifications used during the simulation are shown in Table 7.8, while the specifications of the used VNFs are shown in Table 7.9.

The experiments were conducted on a HPC cluster running Scientific Linux 6.1. Every compute node contains dual Intel Xeon CPU E5-2670 octo-core processors, and 64GB of physical memory. For the CFC-P and SFC-P an entire node was used¹. The time limited algorithm versions were limited to a single CPU core and 12GB of memory. All experiments were repeated 30 times.

¹The execution time was limited to 72 hours on a complete node if no provably optimal solution was found, which occurred for 5 entries in total.

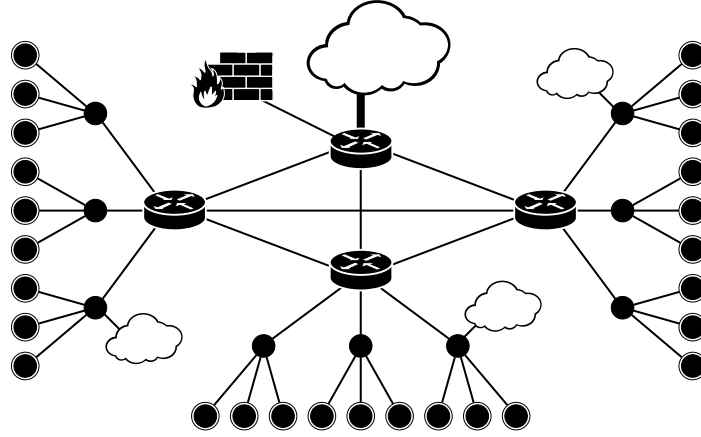


Figure 7.8: The evaluation network contains a backbone network and multiple edge nodes that are interconnected using switches. A cloud and hardware firewall are present in the network and accessible from the backbone network. Smaller edge clouds, that are located close to the edge nodes are also present in the network.

Parameter	Value
Network link capacity	10 Gbps
Cloud uplink capacity	20 Gbps
Cloud CPU	1000 cores
Cloud memory	100000 GB
Cloud core use cost	0.1
Edge cloud CPU	100 cores
Edge cloud memory	10000 GB
Edge cloud core use cost	0.2
Hardware firewall capacity	50000 rps
Hardware router capacity	100000 rps

Table 7.8: Evaluation network hardware parameters.

VNF	CPU	Memory	Provided resources
Routing VNF	1 core	100MB	1000 rps
Firewall VNF	1 core	100MB	1000 rps
DPI VNF	1 core	500MB	100 rps

Table 7.9: Evaluation VNF specification.

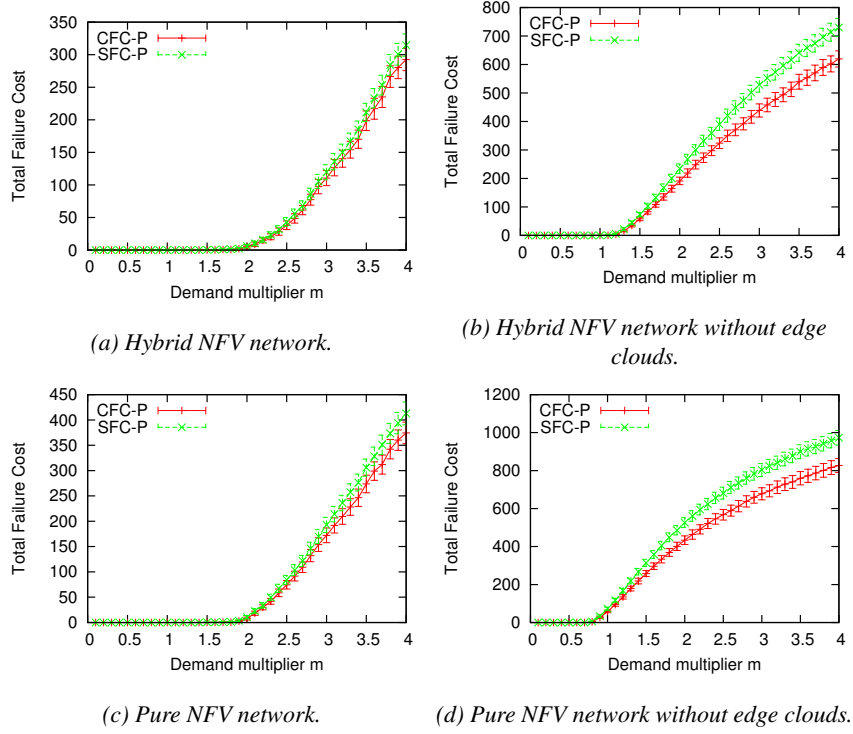


Figure 7.9: Total failure cost comparison for increasing network and service demand. Values averaged over 30 iterations.

7.7 Evaluation Results

7.7.1 Total cost comparison

We compare the total cost making use of an evaluation scenario containing 5 instances of every CFC type shown in Table 7.7, and making use of the four network variants discussed in the previous section. Figure 7.9 shows how the total service failure costs compare for the various evaluation networks when the multiplier m is varied. In all results, CFC-P consistently results in a lower total cost than SFC-P. This is to be expected, as the CFC-P model extends SFC-P, meaning any solution for the latter is also a solution for the former. Therefore, an optimal solution for the CFC-P model will return a result with an identical or lower cost than the SFC-P solution. In all three scenarios, the total cost is dominated by the cost of failing to allocate service chains and features.

Figures 7.9a and 7.9b show two hybrid NFV networks, one where edge clouds are present, and one where they are not present. In both scenarios, network capacity

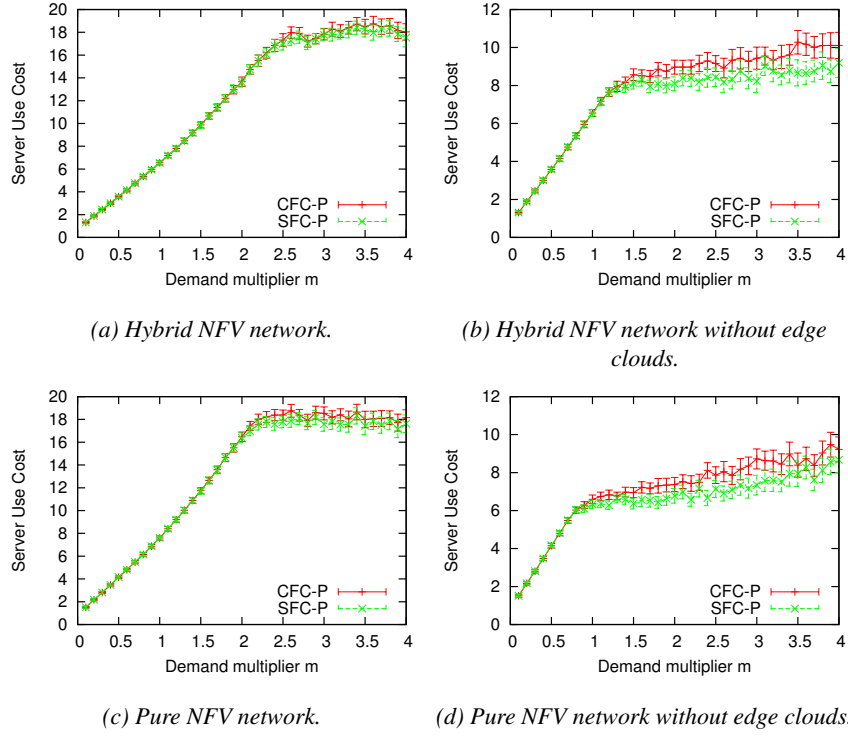


Figure 7.10: Server use cost comparison in network-constrained evaluation scenarios. Values averaged over 30 iterations.

becomes a bottleneck when m increases. When multiple clouds are present in a hybrid NFV network (Figure 7.9a) the benefits of CFC-P compared to SFC-P are limited, only resulting in marginal decreases in the cost. This is the result of the many cloud and PNF nodes spread throughout the network, resulting in many alternative ways to allocate the various service chains using SFC-P. As the cloud nodes are spread throughout the network, there will be limited network capacity, leaving little capacity for adding degraded network flows that the CFC-P could succeed in placing. Because of this, the CFC-P solution is unable to significantly reduce the total cost, as sufficient network capacity is needed to support the degraded fallback services. When no edge clouds are present however (Figure 7.9b), a clear difference between CFC-P and SFC-P costs can be observed. In this scenario, the SFC-P algorithm is severely restricted in how the various service chains are allocated, while CFC-P can improve the total cost by changing the feature configuration of the CFCs and partially allocating the service chain, causing it to fall back to a degraded service.

Comparing the results for a hybrid NFV network with those for a pure NFV network we find that, in pure NFV networks, the difference between CFC-P and

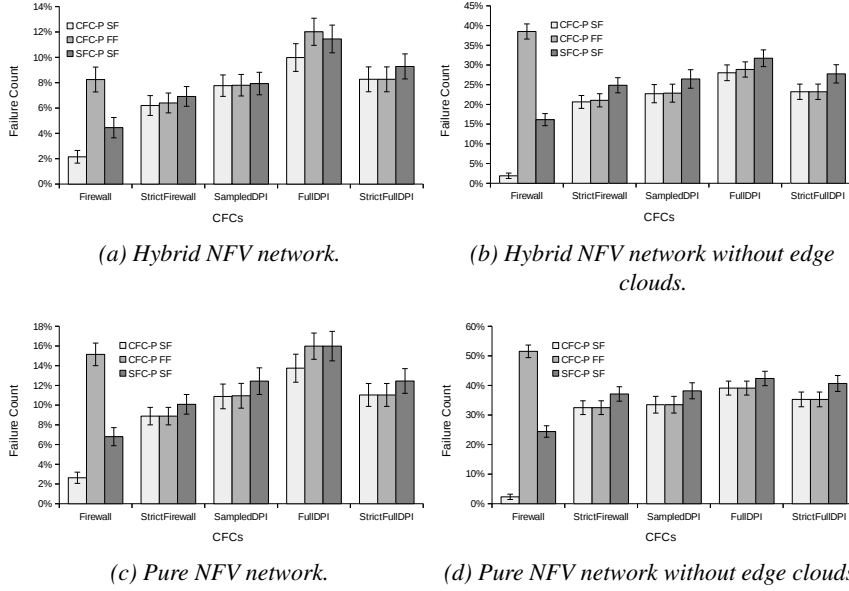


Figure 7.11: A breakdown of the failures occurring during the evaluation scenario by CFC and failure type. For every CFC, the failure rate averaged over the entire evaluation scenario is shown. A distinction is made between service chain failures (SF), which lead to a complete service outage, and feature failures (FF), which result in service degradations. ($m \in [0 \dots 4]$)

SFC-P can be observed both when edge clouds are present, as shown in Figure 7.9c, and when there are no edge clouds, as shown in Figure 7.9d. As in the hybrid scenario, the difference between CFC-P and SFC-P is more pronounced when no edge clouds are present.

Figure 7.10 shows the evolution of the server utilization cost in the same scenario. We observe that CFC-P and SFC-P result in indistinguishable server use costs when no failures occur. Once the load becomes high enough to start causing failures, the server use cost of CFC-P becomes higher than that of SFC-P. This shows that CFC-P is better able to fully utilize the server capacity, which results in the corresponding lower cost of failure discussed previously. This difference becomes more pronounced for the network environments that do not have edge clouds.

Figure 7.11 analyses the service chain and feature failures, and shows the average failure rate of every CFC for the various network scenarios. For CFC-P, this failure rate is split into a service chain failure percentage and a feature failure percentage. When a service chain failure occurs, the features of the service fail as well. For the SFC-P, only the service chain failure percentage is shown, as

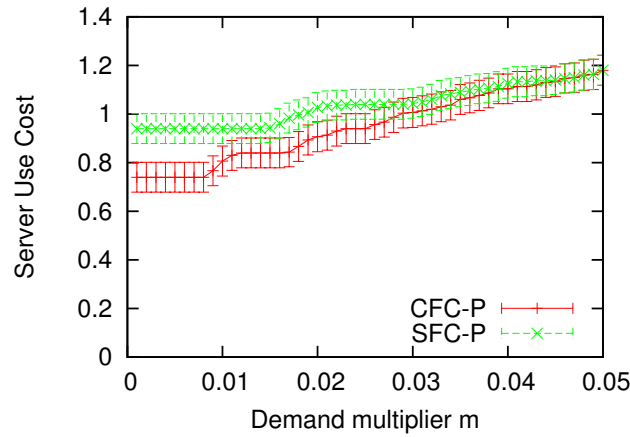


Figure 7.12: Server use cost comparison of in an underloaded scenario containing only *SampledDPI* and *StrictFullDPI* CFCs in the pure NFV network.

it does not support partial service chain failures. We observe that, for the CFCs where service degradation is possible (Firewall and FullDPI), the number of feature failures exceeds the number of service chain failures, often resulting in more feature failures than there are failures of the entire service chain in SFC-P. This is particularly noticeable for the Firewall CFC, which rarely fails completely, but is frequently degraded to a lower quality version. This is caused by the fact that this CFC can fall back to using the `NoFilter` feature, which is very easy to allocate, as it does not require an intermediary service and therefore is more flexible in how the service chain is routed. For services that can not fall back to a lower quality version, a lower failure rate is observed compared to SFC-P. This is caused by an increase in resource capacity due to the partial failures of the services that can fall back to a lower quality service.

7.7.2 Server use costs

While most of the evaluation CFCs can be instantiated in different ways, most of these configurations incur a feature failure cost. Because of this, CFC-P will avoid using these CFC configurations, unless there is insufficient capacity which causes this to become impossible. The *SampledDPI* CFC can however be implemented using either the *FullDPI* or *SampledDPI* features without incurring any cost. When an underutilized DPI NF is present within the network, workload for the *SampledDPI* CFC can be moved to it, reducing the load on Firewall and Routing NFs. This can in turn reduce resource consumption and the server use cost.

Figure 7.12 shows this effect for a scenario containing 10 *SampledDPI* CFCs

and 10 StrictFullDPI CFCs in the pure NFV network. As the DPI NF within the evaluation scenario can only handle a limited number of requests per second, this behavior is only observable for low m values. When there is a low load on the system, server use cost reductions of up to 20% are observed when CFC-P is used instead of SFC-P.

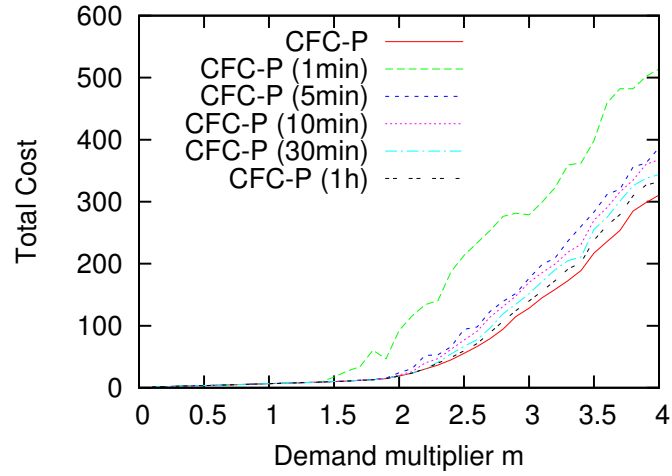
7.7.3 Time-limited heuristics

We compare the performance of CFC-P when its execution is time-limited using multiple different time limits. Figure 7.13 shows the results of this comparison for the hybrid NFV network and the pure NFV networks. We observe that longer execution durations lead to higher quality results. Despite this, a close to optimal quality can be achieved after one minute of computation for lower demand multipliers. For higher demand, longer computations may be required. For $m < 2$, the heuristic finds a solution which costs less than 5% more than the optimum. For higher multiplier values results within 20% to 30% are found in ten minutes. For the networks without edge clouds, which are not presented, the algorithms finish noticeably quicker: after one minute a solution within $\pm 5\%$ of the optimal result is found, and after five minutes, the optimal result is found in all cases.

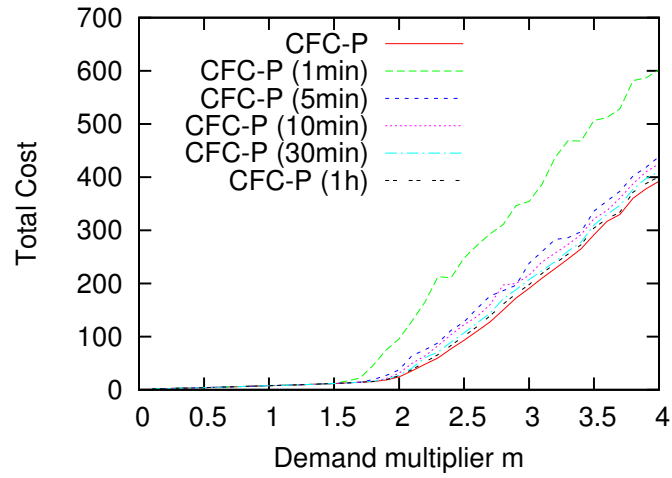
7.8 Conclusions

In this chapter, we introduced the concept of CFCs, a service chain modeling approach that extends SFCs by taking into account service chain variability. CFCs make it possible to model complex service chains in NFV networks that can be allocated in multiple alternative ways, e.g. using either physical or virtual NFs, or using different NFs with differing quality or functionality. This makes it possible to choose the best service implementation at runtime, reducing hosting costs, or to fall back to a degraded service when insufficient resources are available. We described how CFCs can be modeled and managed, and presented a formal CFC-P model that can be used to allocate CFCs on service provider networks.

We compared the CFC-P model with a SFC-P model which allocates SFCs, and therefore does not support service chain variability, using a connectivity service scenario executed in a simulated service provider network. In this scenario, CFC-P was shown to consistently result in a lower cost, indicating that taking variability into account during service chain placement can improve the quality of network services. CFC-P reduces costs in two ways: (1) by better handling service failures, making it possible to manage reduced quality versions of a service, resulting in cost reductions up to 15% depending on the scenario; and (2) by more efficiently using resources when low instance utilization occurs, showing server use cost reductions up to 20%.



(a) Hybrid NFV network.



(b) Pure NFV network.

Figure 7.13: Total cost comparison of time-limited CFC-P in the hybrid and pure NFV networks.

Acknowledgment

Hendrik Moens is funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). This work is partially funded by the FWO project G059615N. This work was carried out using the STEVIN Supercomputer Infrastructure at Ghent University, funded by Ghent University, the Flemish Supercomputer Center (VSC), the Hercules Foundation and the Flemish Government – department EWI.

References

- [1] IETF. *Service Function Chaining (SFC) Architecture (draft)* [online]. 2015. Last accessed: February 2015. Available from: <https://datatracker.ietf.org/doc/draft-ietf-sfc-architecture/>.
- [2] K. Pohl, G. Böckle, and F. Van Der Linden. *Software product line engineering: foundations, principles, and techniques*. Springer-Verlag New York, Inc., 2005.
- [3] B. Jennings and R. Stadler. *Resource Management in Clouds: Survey and Research Challenges*. Journal of Network and Systems Management, pages 1–53, mar 2014. doi:10.1007/s10922-014-9307-7.
- [4] C. Low. *Decentralised Application Placement*. Future Generation Computer Systems, 21(2):281–290, feb 2005. doi:10.1016/j.future.2003.10.003.
- [5] M. Rabbani, R. Pereira Esteves, M. Podlesny, G. Simon, L. Zambenedetti Granville, and R. Boutaba. *On tackling virtual data center embedding problem*. In Proceedings of the 13th IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), pages 177–184. IEEE, may 2013.
- [6] F. Wuhib, R. Yanggratoke, and R. Stadler. *Allocating Compute and Network Resources Under Management Objectives in Large-Scale Clouds*. Journal of Network and Systems Management, 23(1):111–136, jul 2013. doi:10.1007/s10922-013-9280-6.
- [7] M. Zhani, Q. Zhang, G. Simon, and R. Boutaba. *VDC Planner: Dynamic migration-aware Virtual Data Center embedding for clouds*. In Proceedings of the 13th IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), pages 18–25. IEEE, may 2013.
- [8] L. Shi, B. Butler, D. Botvich, and B. Jennings. *Provisioning of requests for virtual machine sets with placement constraints in IaaS clouds*. In Proceedings of the 13th IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), pages 499–505. IEEE, may 2013.
- [9] M. Alicherry and T.V. Lakshman. *Network Aware Resource Allocation in Distributed Clouds*. In IEEE INFOCOM, pages 963–971. IEEE, mar 2012. doi:10.1109/INFCOM.2012.6195847.
- [10] R. Esteves, L. Zambenedetti Granville, H. Bannazadeh, and R. Boutaba. *Paradigm-based adaptive provisioning in virtualized data centers*. In Proceedings of the 13th IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), pages 169–176. IEEE, may 2013.

- [11] R. Guerzoni, R. Trivisonno, I. Vaishnavi, Z. Despotovic, A. Hecker, S. Beker, and D. Soldani. *A Novel Approach to Virtual Networks Embedding for SDN Management and Orchestration*. In Proceedings of the 14th Network Operations and Management Symposium (NOMS 2014), pages 1–7. IEEE, may 2014. doi:10.1109/NOMS.2014.6838244.
- [12] S. Clayman, E. Maini, A. Galis, A. Manzalini, and M. Nicola. *The Dynamic Placement of Virtual Network Functions*. In Proceedings of the 14th Network Operations and Management Symposium (NOMS 2014), pages 1–9. IEEE, may 2014. doi:10.1109/NOMS.2014.6838412.
- [13] NFV ETSI ISG. *GS NFV-MAN 001. Network Functions Virtualisation Management and Orchestration V1.1.1* [online]. 2014. Last accessed: February 2015. Available from: <http://www.etsi.org/index.php/technologies-clusters/technologies/nfv>.
- [14] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl. *Variability modeling to support customization and deployment of multi-tenant-aware Software as a Service applications*. In Proceedings of the ICSE Workshop on Principles of Engineering Service Oriented Systems (PESOS 2009), pages 18–25. IEEE, may 2009. doi:10.1109/PESOS.2009.5068815.
- [15] M. Abu-Matar and H. Gomaa. *Feature Based Variability for Service Oriented Architectures*. In Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture (WICSA 2011), pages 302–309. IEEE, jun 2011. doi:10.1109/WICSA.2011.47.
- [16] M. Abu-Matar and H. Gomaa. *Variability Modeling for Service Oriented Product Line Architectures*. In Proceedings of the 15th International Software Product Line Conference (SPLC 2011), pages 110–119. ACM, aug 2011. doi:10.1109/SPLC.2011.26.
- [17] S. T. Ruehl and U. Andelfinger. *Applying Software Product Lines to create Customizable Software-as-a-Service Applications*. In Proceedings of the 15th International Software Product Line Conference (SPLC 2011), pages 16:1–16:4. ACM, aug 2011. doi:10.1145/2019136.2019154.
- [18] G. H. Alférez and V. Pelechano. *Context-Aware Autonomous Web Services in Software Product Lines*. In Proceedings of the 15th International Software Product Line Conference (SPLC 2011), pages 100–109. ACM, aug 2011. doi:10.1109/SPLC.2011.21.
- [19] H. Moens and F. De Turck. *VNF-P : A Model for Efficient Placement of Virtualized Network Functions*. In Proceedings of the 10th International

- Conference on Network and Service Management (CNSM 2014), pages 418–423. IEEE, nov 2014. doi:10.1109/CNSM.2014.7014205.
- [20] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. D. Turck. *Feature Placement Algorithms for High-Variability Applications in Cloud Environments*. In Proceedings of the 13th Network Operations and Management Symposium (NOMS 2012), pages 17–24. IEEE, apr 2012. doi:10.1109/NOMS.2012.6211878.
- [21] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. *Cost-Effective Feature Placement of Customizable Multi-Tenant Applications in the Cloud*. Journal of Network and Systems Management, 22(4):517–558, oct 2014. doi:10.1007/s10922-013-9265-5.
- [22] H. Moens and F. De Turck. *Feature-Based Application Development and Management of Multi-Tenant Applications in Clouds*. In Proceedings of the 18th International Software Product Line Conference (SPLC 2014), pages 72–81. ACM, sep 2014. doi:10.1145/2648511.2648519.
- [23] *OpenFlow* [online]. 2015. Last accessed: February 2015. Available from: <https://www.opennetworking.org/sdn-resources/openflow>.
- [24] *IBM ILOG CPLEX 12.6.1* [online]. 2015. Available from: <http://www-01.ibm.com/software/integration/optimization/cplex-optimizer>.
- [25] *Scala 2.11.2* [online]. 2015. Available from: <http://www.scala-lang.org/>.

8

Conclusions and research perspectives

In this dissertation, several contributions to the field of cloud and network management were presented. The central research question in this dissertation is “How can application and service customizability be handled in multi-tenant cloud and network environments?”. This question was addressed by detailing the following contributions: (1) an approach for modeling and managing customizable Software-as-a-Service (SaaS) applications by using a feature-based approach, which achieves multi-tenancy by composing applications based on multi-tenant service components, was detailed; (2) datacenter management algorithms that can be used to solve the feature placement problem which is used to allocate resources for customizable SaaS applications cost-efficiently were presented and evaluated; and (3) network-aware algorithms that can be used for resource management and as an access filter were developed. These contributions are summarized below, followed by a description of future perspectives.

8.1 Contributions

8.1.1 An approach for modeling and managing customizability of multi-tenant SaaS applications

In this dissertation, the Feature-Based Binary (FBB) approach for managing and provisioning customizable SaaS applications was introduced and evaluated. This approach, described in Chapter 2, splits an application into separate, multi-tenant components, from which the application is composed using a Service-Oriented

Architecture (SOA). Each of these components realizes part of the application, and therefore application customization can then be achieved by connecting different components. As every component out of which the application is composed can be shared between multiple tenants, a high degree of multi-tenancy can be obtained, while individual tenants can still receive a highly customized product.

This dissertation presented management algorithms to allocate resources for FBB applications in Chapters 3 and 4. These management algorithms were then framed within an application development and runtime platform in Chapter 5. This approach makes it possible to manage customizable, multi-tenant SaaS applications throughout their development, customization, and deployment using a feature modeling approach. To this end, algorithms are developed that can be used to automatically convert development feature models to runtime feature models, which can be used by the algorithms presented throughout this dissertation.

8.1.2 A resource allocation approach for managing customizable SaaS applications within datacenters

To manage customizable FBB SaaS applications in cloud datacenters, specialized resource management algorithms were presented. In Chapter 3, the feature placement problem was introduced. The feature placement problem extends the generic application placement problem, which is used to determine where applications are allocated within a datacenter, and takes the relations between application components into account. To achieve this, the feature placement algorithm is made aware of the application customizability by making use of a feature model. This dissertation presented a formal mathematical model, defining the feature placement problem, and presented multiple heuristic algorithms that can be used to quickly solve the problem.

The analysis in Chapter 3 solves the static feature placement problem, and finds a solution for allocating a collection of applications at once. In practice, applications may start and stop at various times, and their load may change through time, often making it necessary to use dynamic management algorithms. In Chapter 4, dynamic feature placement is introduced, which limits the number of migrations, making it suitable for use in dynamic management scenarios. When compared to a static management approach, it was shown that the dynamic algorithms reduce the number of migrations needed by 77%. The heuristic algorithms built to solve the dynamic feature placement problem, which were based on the approach presented in Chapter 3, achieve results that on average cost less than 3% more than the optimal cost in the evaluation scenarios.

8.1.3 Network-aware modeling and management algorithms for inter-cloud network environments

Focusing only on scaling managing applications within a datacenter, and ignoring the networks connecting the cloud to the end user and networks interconnecting clouds, could cause failures due to insufficient network capacity. This was addressed in Chapter 6, where an access filter was designed, and in Chapter 7, where feature placement concepts were extended, making it possible to use them to allocate resources within wide area networks.

The access filter presented in Chapter 6 works by modeling the service and network resource utilization within networks. By structuring the service chains formed by SaaS applications hierarchically, a resource sharing approach was achieved. This makes it possible to model the way in which services are executed in parallel. For the two evaluated use cases, it was shown that the developed hierarchical algorithm required $\pm 42\%$ and $\pm 52\%$ less resources than an approach without resource sharing, without any workflow failures occurring during the executed simulations.

Chapter 7 focuses on server provider networks, and shows how the feature placement approach introduced in this dissertation can be used in wide area networks. The focus on hybrid Network Functions Virtualization (NFV) makes the approach applicable for both traditional networks and NFV networks, making it possible to allocate customizable service chains containing multiple services and Virtual Machines (VMs) in these networks. The approach is shown to offer benefits compared to an approach which does not support variability, both when service failures occur due to insufficient resources, and when resource utilization costs can be reduced by efficiently making use of service variability. In addition, this approach makes it possible to define network services more generically, deferring customization choices until services are deployed.

8.2 Future perspectives

Using the approaches developed during this dissertation, it becomes possible to migrate highly customizable applications to cloud environments, and to develop novel customizable SaaS offerings. This enables service providers to benefit from the various advantages that clouds offer, such as increased flexibility and easier application management. This also makes it possible to offer the provided services at a lower cost, enabling service providers to sell these applications to clients for whom the cost would previously have been prohibitive.

The network-aware variability management approach in particular can be utilized to support complex services that can be provided using both physical hardware and virtual services. This approach makes it possible to use physical hardware in networks more cost-efficiently, by ensuring the hardware is fully utilized while

spillover is handled by virtualized instances. This also offers a cheaper migration path to NFV, as existing physical hardware can be used in conjunction with virtualized service instances.

8.2.1 Advanced variability modeling

The feature modeling approach used within this dissertation is based on a limited set of hierarchically structured relations. This hierarchical approach was already extended in Chapter 5 by introducing additional, non-hierarchical relations to increase the expressiveness of the feature models. Creating additional relations and additional ways to specify feature models can make it easier to specify some application configurations, for which otherwise complex and unwieldy feature models would be needed. In general, the approach used in this dissertation can be extended to support any feature modeling approach which can be expressed using logical operations.

Additional feature model flexibility could also be added by modifying the feature impact concept used throughout this dissertation. Feature impacts are used to determine the impact of the inclusion of features on the resource load of services and networks, and are expressed using scalar values. This approach ensures that a feature is either included or excluded, and that if the feature is included its impact is counted accordingly. When feature models are used to model application failure however, failing to provide resources for features incurs a failure cost. In these cases, it may be beneficial to consider partial feature failures, where features themselves are only partially allocated. Then, only a portion of the failure cost could be incurred, while only a portion of the feature impact is incurred. While similar behavior can be achieved by adding additional features to the models, this would result in a significant increase of complexity, both in the specification of the feature models and in their management. It would therefore be useful to add the concept of partial feature failure to the models and algorithms.

8.2.2 Federated management of customizable SaaS

In practice, network communication often traverses multiple administrative domains owned by different service providers. The proposed management approach discussed throughout this dissertation focuses on the point of view of a single service provider, who is in charge of managing, customizing and deploying the application. In this approach, the capabilities of other network and service providers are not taken into account. Instead, they are abstracted, and replaced by basic nodes, i.e. a cloud provider is replaced by a single node in a network management scenario, and a network service provider is similarly replaced by a single edge. This is why the network is not taken into account in Chapters 3 and 4, why the connection between access router and cloud is replaced by a single link in Chapter 6, and why the

datacenters are replaced by a single node in Chapters 6 and 7. Using this approach decreases the functional requirements for service providers, and ensures only one network domain must be aware of the SaaS variability, while the other domains can be managed using standard management approaches.

Limiting the awareness of service customizability to a single domain also limits the potential cost savings to this domain. In the future, it may be beneficial to make multiple domains aware of application customizability. Using this approach, multiple domains could cooperate, sharing the application customization information, and enabling them to cooperate to determine a cheapest configuration for a SaaS. Alternatively, the SaaS feature model could be split up into smaller segments, which are distributed to the various service providers who each provide their part of the service, while a single service provider remains in charge of providing the complete service.

8.2.3 Resilient management of customizable SaaS

Services provided using networks and clouds make use of various networked devices. Failure of any of these devices, or the links between them, may cause service interruptions. While the algorithms designed throughout this dissertation can react to these failures by determining new resource allocations that result in a low cost, there will still be a period of time between this failure and the time when the service is restored, impacting service quality.

When hardware failures are frequent, or have a significant impact, it would be beneficial to add failure protection to the management algorithms. Failure protection proactively reserves network and server resources to protect against failures. When hardware failure is detected, these backup paths are then already configured, making it possible to recover more quickly.

8.2.4 Streamlining the configuration process

In the presented approach, feature modeling is used throughout application development, customization and runtime. The disadvantage of orchestrating the customization process using feature models, is that this could make the clients aware of the structure of these internal models. Such an approach makes the process of requesting applications more complex than for non-customizable applications. This can be resolved by letting the configuration be handled by a service provider employee instead of by the client, or by creating a wizard-based configuration interface which is linked to the feature model.

Alternatively, the required features could be determined based on the chosen application capabilities. By analyzing the desired service configuration, it is possible to determine the functional capabilities that are needed to provide the applications. Hence, the features that are capable of providing these capabilities could be derived

automatically. Using this approach, the configuration process could be streamlined and simplified, making it easier for clients to request changes to the provided service.

8.2.5 Containerization

This dissertation focuses mainly on the management of customizable applications of which the components are contained in VMs, which make use of hardware virtualization to emulate a computer system. In recent years, the concept of operating-system-level virtualization has been gaining popularity as an alternative to hardware virtualization. In this approach, multiple software containers are deployed on a single, shared host operating system. This approach has different advantages and disadvantages compared to the more traditional approaches like hardware virtualization. On one hand this approach is slightly less flexible, as a container may put constraints on the used underlying operating system, and it is harder to isolate the containers from a performance and security point of view. On the other hand, the overhead of containers is smaller, and it is easier to scale them when demand varies.

These properties make it interesting to study how software containers could be used in conjunction with the VM-based approaches presented in this dissertation. Due to their more limited security and performance isolation, they may not be suitable to fully replace VMs in all cases. Sometimes, the quicker scaling and lower overhead of containers could however potentially reduce costs in a feature-based approach. Software containers could, for example, be used specifically to host rarely used features to reduce their overhead.

8.2.6 Managing the Internet of Things

The Internet of Things (IoT) envisions a world where a huge number of smart devices are interconnected, creating new and innovative ways for these devices to interact. Offering services in these highly heterogeneous environments will however be challenging: there may be multiple versions of a device offering similar but not the same functionality, or a service may need to be able to function both with and without some devices. In addition, computations which may be too complex to run on the low-powered devices, may be executed on different devices within the network: some can be done on the devices themselves, alternatively they can be executed on a cellphone, on set top boxes, or remotely using (edge) clouds.

The concepts developed during this dissertation deal with service variation, and could be applied to the IoT use case. Specifically, the network-aware customization management concepts could be applied to define and manage complex services in these heterogeneous environments.



Design and Evaluation of a Hierarchical Application Placement Algorithm in Large Scale Clouds

H. Moens, J. Famaey, S. Latré, B. Dhoedt and F. De Turck

Published in Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011)

This paper was awarded the best student paper award

As the requirements and scale of cloud datacenters increase, scalable management of the cloud is needed. Centralized solutions, like the ones presented in Chapters 3 and 4 are limited in their scalability. In this appendix a general approach is introduced for using centralized cloud resource management algorithms in a hierarchical context, increasing the scalability of the management system while maintaining a high placement quality. This approach can be used to improve the scalability of the feature placement algorithms when their performance becomes too slow due to their centralized nature. The proposed method uses aggregation and decoupling techniques to generate input values for a centralized application placement algorithm, which is executed in all management nodes of the hierarchy. The evaluation results show that a solution, within 5% of the optimum placement when using the centralized algorithm, can be achieved hierarchically in less than 25% of the time needed for execution of the centralized algorithm.

A.1 Introduction

In recent years the adoption of cloud computing has increased greatly. The increasing scale of clouds complicates management, which leads to scalability issues of the management system itself, compromising the scalability of hosted applications. Centralized management systems are being replaced by distributed management infrastructures, that are often fully decentralized, lacking a full overview of the system, and making it more difficult to achieve a global optimum.

Our previous theoretical work [1] indicates that a hierarchical structuring of control nodes enables good sharing of context, information of the current state of the system, with a relatively small communications cost, while at the same time having a high scalability.

In this appendix, we investigate how centralized algorithms can be modified to work in large-scale environments by using a hierarchical management system. The different levels of the hierarchy have a different view of the system, with highest-level controllers having an overview of the system based on aggregated values, increasing the scalability of a hierarchy compared to a purely centralized solution. To this end, we evaluate how centralized solutions for one specific problem, the cloud application placement problem, can be incorporated into a hierarchical management system.

One of the key challenges in cloud management is quickly adjusting application resource allocation in the face of changing demands, and doing this in a scalable way. Determining which servers in the cloud need to execute which applications is done by means of solving the application placement problem. This problem is NP-hard, and many different solutions have been proposed [2–5].

Currently the two common approaches to application placement are centralized and fully decentralized solutions. In centralized solutions, a controller gathers monitoring information, calculates an optimal placement, and enforces the configuration. These algorithms tend to be highly complex and slow to execute, which makes scalability an issue. Decentralized approaches on the other hand optimize using only local information, leading to suboptimal placement.

Our results demonstrate that a hierarchical approach leads to scalable and fast cloud application placement, as the structure scales better than the centralized approach, and has a higher-level overview of the total system compared to the fully decentralized model. A generic template for using centralized application placement algorithms in a hierarchical fashion is presented, based on information aggregation and decoupling of management levels. The aggregation and decoupling techniques demonstrated in this appendix will enable the use of various centralized algorithms in a hierarchical fashion, greatly increasing scalability of existing cloud management solutions.

In the next section, we will discuss related work. Afterwards, in Section A.3 we will give an overview of the system architecture. Section A.4 contains a generalized formal description of the application placement problem. Following this, we discuss the hierarchical management itself in Section A.5, where both the creation of a management hierarchy and the modification of centralized algorithms to function in the system are described in-depth. In Section A.6 we evaluate the proposed solution. Finally, Section A.7 contains our conclusions.

A.2 Related work

Much work has been done concerning the application placement problem. Most of the work can be divided into two categories: centralized and fully distributed.

One of the first articles on the subject of application placement was published in 2003 [5]. A solution is generated centrally using linear programming and genetic algorithms. The solution is more geared towards consolidation of datacenter resources rather than dynamic cloud provisioning in large scale clouds, as is the case here. Our solution is designed to scale where centralized solutions no longer function, be it because of CPU limitations, which would be the bottleneck when using linear programming, or bandwidth bottlenecks, which would occur when using genetic algorithms.

Multiple centralized solutions have been proposed [2, 6–8], and many of the algorithms use similar principles. The centralized solution proposed in [2] uses multiple min-cost max-flow problems to generate a suitable solution and has a complexity of $O(n^{2.5})$, better than older solutions. The algorithm was further expanded in [8], yielding slightly better performance and results. As centralized solutions have access to all information concerning managed nodes, the placement quality is generally very good. These solutions work well for smaller datacenters, but do not scale well for large datacenters. Our solution uses centralized algorithms on clusters which are limited in size, leading to a much higher scalability at a cost to placement quality due to the smaller overview of the system resources.

A distributed peer to peer system, used for cloud management was demonstrated in [9]. In this solution, every node contains a database of management information, which is selectively flooded towards other nodes. This ensures every node has the relevant information for its management, but due to its design, no single node has a full overview of the network. In our work, we focus specifically on a subproblem, the application placement problem, and we examine how application placement can be made to scale while maintaining a global overview.

In [3] a fully decentralized approach is used, based on a gossip-protocol. Here individual nodes manage themselves and continually exchange information. Nodes continually improve their configuration by exchanging information and shifting load between them. It is shown that this leads to an optimal configuration if memory

constraints are omitted. This decentralized approach has a very high scalability, but convergence to an optimum is slower when compared to the centralized approach, and as each step leading to a configuration causes the migration of applications, these steps are expensive. Another decentralized approach, proposed in [4] uses an economical approach in which every actor tries to maximize its own gain. In doing so, a good global solution is obtained, but like all fully decentralized solutions there is no higher-level overview of the network. In contrast to these fully decentralized solutions, ours executes application placement on different hierarchy levels in which higher-level nodes have better overview of the system and are able to achieve good placement quality, while still maintaining good scalability.

While most application placement approaches are based on CPU and memory requirements, [10] executes application placement based on the physical location and bandwidth requirements of the servers, trying to put as many application components as possible close to each other and taking into account the physical system configuration. The solution is based on a central placement manager of which multiple, synchronized instances can exist. It combines properties of both the centralized approach and decentralized approaches, but needs much synchronization between management node instances. In this appendix we propose a management system in which the different management nodes are more loosely coupled, needing only limited amounts of communication between nodes.

Hierarchical techniques in provisioning were used in [11], but this system is dependent upon hierarchical application and system descriptions and executes placement using only bandwidth information. By contrast, our solution uses hierarchies for the management itself without needing additional descriptions.

Automatic hierarchical node ordering in peer to peer systems was demonstrated in [12]. TreeP is used to hierarchically structure nodes in peer to peer systems, mainly used to structure object lookup, and its structure is inspired by that of B-Trees. Each node can occur at multiple levels in the tree. Our solution uses a similar approach, but has different goals and thus uses a different structure. We use dedicated management nodes, without strict ordering. Because of this our solution can grow and recover faster, as no node ordering is needed in our management system.

A.3 System Architecture

The cloud computing datacenter consists of multiple servers. All the servers have two applications installed on them: a cloud environment, such as OpenNebula, and management middleware, the functionality of which will be explained later in this section. Most of the servers are *execution servers*, used to execute application instances. Some servers are used by the management system to control the execution servers. These servers, used to manage the cloud system are *management*

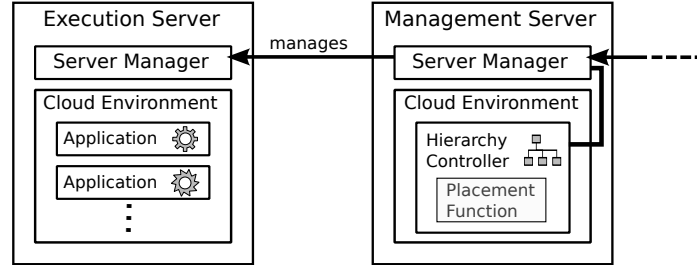


Figure A.1: The system components on management and execution servers

servers. They execute a dedicated management application in their cloud environment instead of regular applications. This approach allows every server to be either a management server or an execution server, enabling dynamic scaling of the management system itself. The servers of the datacenter are connected hierarchically, where the execution servers act as leaves and the management servers are nodes of the management tree.

The different system components are shown in Figure A.1. Each server in the system contains a *server manager*. This is a lightweight middleware component, responsible for maintaining a relationship with the servers parent, gathering management information and sending it to the parent management server. Its main function is abstracting the cloud infrastructure present on the server. As this component is present on all servers, it can also be used to increase robustness of the system by monitoring its parent, and initiating leader election should one of the management nodes fail. A cache of neighbours, nodes with which the server has communicated with recently, can be used to reconnect to the management system.

A second component, the *hierarchy controller* is only present on management servers and gathers service performance information of their children. These children are either application servers or other controllers. The controller executes application placement based on the locally present management information. It also aggregates it and forwards it to its own parent. The parent node executes application placement on a higher level, determining which clusters execute which applications, and how much of each application to execute.

This hierarchy controller, unlike the server manager is a heavyweight component, which is executed like the other applications in the cloud (e.g.: in the case of OpenNebula as a Virtual Machine (VM)). It is responsible for executing the application placement algorithm. As every server in the cloud can be used as a controller, the management system itself can scale when needed and can be made more reliable.

The server manager and hierarchy controller form the backbone of the architecture, but to enable full cloud functionality, additional supporting components would

Symbol	Description
A	The set of all applications.
S	The set of all servers.
Γ	The various resources considered by the system.
Ra	The resource availability of the various servers. Ra_s^r determines the available amounts of resource r on server s .
Rd	The resource demand of the applications. Rd_a^r is a value containing the demand of application a for resource r .
M	The placement matrix, $M_{s,a}^r$ contains the amount of resource r allocated to server s for application a .
Γ_s	Resources for which the demand is strict. They have a fixed demand per-instance and without this amount the placement is invalid.
Γ_l	Resources for which the demand is loose. The goal of the management system is to maximise loose demand fulfilment.
Rd_l	Resource demands for loose resource types.
Rd_s	Resource demands for strict resource types.

Table A.1: Symbols

be needed: An *application request router*, responsible for routing successive application requests to application instances and session management, an *application image repository* containing images of the various hosted applications, and a *policy repository* containing placement restrictions. In this appendix we focus on the application placement problem itself, so these components were not implemented.

A.4 Formal Problem Description

The application placement problem itself has previously been described formally [2–4, 9]. In this section we formally describe and generalize common inputs and outputs of centralized application placement algorithms [2, 8, 10]. An overview of the symbols used here is shown in Table A.1.

A cloud consists of a set of servers S on which a set of applications A are executed. The cloud management system considers a set of multiple resources types Γ , such as memory, CPU and bandwidth. For each resource type $r \in \Gamma$, every server $s \in S$ has available resources Ra_s^r , and every application $a \in A$ has a resource demand Rd_a^r . A centralized application placement algorithm takes a set of inputs and delivers as output a placement matrix M . For application a , server s and resource r , $M_{s,a}^r$ contains the amount resource r to allocate on server s for application a .

The inputs generally contain server resource information Ra , the current placement M' , and application resource demands Rd . When it comes to resource

demands, we differentiate between two resource types which we shall call *strict* and *loose*. The set of strict resources $\Gamma_s \subset \Gamma$ contains demands that are invariable and per-instance, such as memory use, and in some instances bandwidth. Loose demands $\Gamma_l \subset \Gamma$, such as CPU requirements and sometimes bandwidth, are total demands. The goal of the application placement problem is to optimize the fulfilment of loose requirements, while respecting the strict application requirements. Generally, strict requirements of applications are considered fixed for every application whereas loose requirements are variable [7]. Strict resource demands are indicated by Rd_s , loose resource demands by Rd_l .

The complete specification of the application placement function is:

$$aplace : Ra \times Rd_s \times Rd_l \times M' \rightarrow M$$

A.5 Hierarchical management

In this section we will describe the hierarchical management structure. First we will describe the management hierarchy itself, after which we will explain how the centralized algorithm is used in individual management nodes.

A.5.1 Hierarchical Management Structure

A hierarchical management node organisation is dynamically created by executing “add node” operations for each of the servers to add them to an existing management hierarchy. As more nodes are added, the hierarchy will automatically restructure itself. The structure of the management hierarchy is inspired by that of B-Trees [13]. B-Trees are datastructures, used mainly for ordering large amounts of data. The most important characteristics of B-Trees are the large number of children for every node and an equal depth of tree leaf nodes. Every node in a B-tree, except for the root, contains between n and $n/2$ entries. The root itself is allowed to have any number of nodes between 0 and n .

In the hierarchical management scenario, less restrictions are needed as the hierarchy is only used to structure nodes, and not to order them. Adding nodes can be realised simply by adding them to any controller at the lowest level. Deleting execution nodes can be done trivially, whereas deleting management nodes is achieved by performing a leader election amongst its children. Furthermore, we change the restrictions at the root node: an imbalanced tree is allowed, but only at this level. This enables us to require a higher minimum node count in the root than in a regular B-Tree, as using a dedicated server to manage only a small number of child servers would be a waste of resources.

As controllers gain more and more children, the execution time of *aplace* continually increases. Eventually the execution time exceeds a given threshold,

indicating that the node is overutilized. In the reverse case, if execution time gets too low, network delay overhead becomes a bigger concern than *aplace* execution time, which indicates the node is underutilized. By executing *aplace* for various server counts the number of children causing overutilization and underutilization can be determined. These values are C_{min} and C_{max} . If $C_{min} \leq \frac{C_{max}}{2}$, splitting and merging of nodes can be achieved using basic B-Tree operations. If $C_{min} > \frac{C_{max}}{2}$ techniques used in variations on B-Trees can be used where n nodes are considered and split into $n + 1$ nodes (as opposed to splitting a single node into two nodes in regular B-Trees).

The splitting of overutilized nodes is illustrated in Figure A.2. A node n_i chooses a node $n_j \in children(n_i)$, which it adds to $children(parent(n_i))$ it then chooses half the nodes remaining in $children(n_i)$, which it adds as children to n_j . The opposite situation is demonstrated in Figure A.3, where an underutilized node is deleted and its children are added to its neighbours.

In a bootstrap scenario server managers connect to each other and execute leader election to determine the root of the management hierarchy. Once this root is chosen the other nodes can be added to it using a default “add server node” operation. Nodes can find each other by providing an initial neighbour cache or by using separate approaches such as the Dynamic Domain Name System.

Because there are only a few restrictions, this structure can be created and updated swiftly, enabling dynamic restructuring of the management hierarchy. Furthermore, the equal depth of the various nodes ensures the various nodes at each level are either all management servers or all execution servers, making the set of managed servers at each level more homogeneous.

A.5.2 Algorithm Details

We will now demonstrate how the general *aplace* function can be executed in a hierarchical structure. As servers are grouped hierarchically, each server s has a parent $parent(s) = p$, $p \in S$ and a set of children $children(s) \in S$, except for the root control node, which has no parent, and execution nodes, which have no children. Every management server executes the *aplace* function. We will now consider a single cluster with management server m , which will execute the *aplace* function with modified inputs and outputs:

$$M'_m = \text{aplace}(\hat{Ra}, \hat{Rd}_s, \hat{Rd}_l, M'_m)$$

As Rd_s is generally taken to be static, it can be considered as application information, available to every management node involved in managing a specific application, so $\hat{Rd}_s = Rd_s$. M'_m is always present on each management node, except in bootstrap scenarios where the general bootstrap scenario of the specific *aplace* function needs to be used. This leaves \hat{Ra} and \hat{Rd}_l , which need to be aggregated across the management tree.

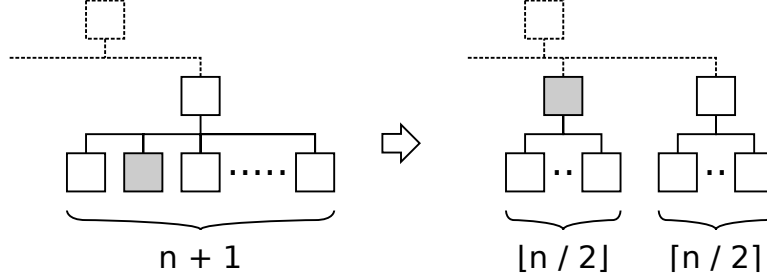


Figure A.2: Solving overutilization by splitting a node and promoting a child node (grey) to peer status.

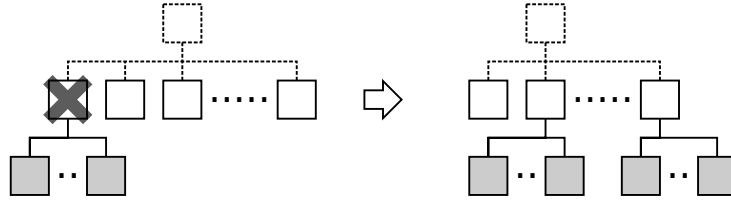


Figure A.3: Solving underutilization by removing a node and distributing its children (grey) amongst the node's peers.

A.5.2.1 Resource availability aggregation

Server resource information \hat{Ra} at the lowest level can be observed from the child nodes and can be used directly, so if a child $c \in \text{children}(m)$ is an execution server, $\hat{Ra}_c^r = \hat{Ra}_c^r$ for all resource types r . At higher levels, in every management server, an aggregated value of child resource information must be determined. Whereas a single server has a definite resource availability Ra_s , a cluster of servers does not. As the placement algorithm can only function using definite values for Ra , a value must be determined for the clusters by aggregating the values of its children. As a result, aggregated resource information \hat{Ra} needs to be determined which indicates how many resources its parent can realistically expect to allocate on the cluster. This estimation will have to be revised whenever underutilization of resources or unrealistic resource allocation occur. For a resource $r \in \Gamma_l$ and a cluster c an aggregated estimation \hat{Ra}_c^r can be determined:

$$\hat{Ra}_c^r = wE_{r,c}^{low} + (1 - w)E_{r,c}^{max} \quad (\text{A.1})$$

$$E_{r,c}^{low} = C_c^r + U_c^r \quad (\text{A.2})$$

$$E_{r,c}^{max} = \sum_{n \in \text{children}(c)} \hat{Ra}_n^r \quad (\text{A.3})$$

The complete estimation is denoted in the first equation, with E^{low} an estimate of available resources that will generally be close, but slightly too low and E^{max} an upper bound which will be too high. The actual ratio will be determined by a weight w , which will be determined experimentally. E^{max} is the sum of all available resources of the children. Finally, E^{low} combines two values: C_c^r , the amount of resource r currently allocated in the cluster, and U_c^r , an estimate of the remaining usable space on all servers on which more application instances can be instantiated:

$$C_c^r = \sum_{s \in children(c)} \sum_{a \in A} (M_c')_{s,a}^r \quad (A.4)$$

$$U_c^r = \sum_{s \in \Upsilon} (\hat{R}a_s^r - \sum_{a \in A} (M_c')_{s,a}^r) \quad (A.5)$$

$$\Upsilon = \{s \in S | \forall r' \in \Gamma_s : \hat{R}a_s^{r'} - (M_c')_{s,a}^{r'} < \min_{a \in A} \hat{R}d_a^{r'}\} \quad (A.6)$$

Equation A.4 determines the amount of resources allocated by the current placement matrix. This can be achieved by summing the allocations in the cluster's placement matrix for all servers and applications. Equation A.5 is used to evaluate the remaining amount of free space on the server. To this end, the remaining resource availability of all servers on which more applications can be executed, the set Υ , are summed.

To determine the set of usable servers Υ , minimal strict resource requirements for all applications are determined. If the resource availability on a server for a resource $r \in R_s$ is less than the minimal resource requirement, no additional applications can be instantiated on this server. Therefore, the remaining resources of this server are not added to the aggregated resource information. This is denoted formally in Equation A.6.

In a bootstrap scenario there is no current allocation, so $C_c^r = 0$. Hence, the first value of $E_{r,c}^{low}$ will be the sum of all available resources, the same as $E_{r,c}^{max}$ which will cause overallocation of applications on the selected cluster. The cluster will then place as much of the load as possible, maximizing the amount of allocated resources. A second execution will have much higher C_c^r , and severely reduced U_c^r , leading to a better second estimate. The value of C_c^r gives an accurate representation of the current resource allocation. The value of U_c^r only gives an upper bound on possible resource availability. As the share of U_c^r in the estimation decreases significantly after one allocation, the value of a second estimation will be much more accurate than the initial

estimation. Consequently, doing multiple placements will increase placement quality.

A.5.2.2 Demand decoupling

The other input value $\hat{R}d$ determines the application demand. Here, lowest-level clusters gather application demand information and send this to its parent nodes. Only the root has a total overview of the application demand Rd and schedules based on this information, so at the root $\hat{R}d = Rd$. These scheduling requirements are passed on to its children, who can then start scheduling based on this information. This method has the disadvantage that all management levels need to be passed to enable scheduling.

These levels can be decoupled however, by changing the type of information passed on between these levels. Application placement yields a placement matrix M . Instead of directly passing on application demands as per M , two values are propagated across the tree for every application a : an application share $\sigma_a \in [0..1]$, and total application demand D_a^r for resource r . Server resource demand can then be calculated as $Rd_a^r = \sigma_a \times D_a^r$. At the root level, $\sigma_a = 1$ for all applications, ensuring the full application demand will be met, leading to $\hat{R}d = Rd$ as expected. The advantage of this approach is that passing on σ is much cheaper than directly using $\hat{R}d$ as the latter requires placement calculation at higher levels, while the former only distributes a single value across the hierarchy. It allows management nodes to work independently, instantly reacting to changing demands.

This still requires the highest level of the hierarchy to be aware of every application that is active in the system, which makes application placement more expensive as an increase in applications leads to an increase in execution time. In realistic situations however, not every application needs to be known at every level as smaller applications can be managed by only a part of the management tree. We use application delegation to resolve this issue. A management node can delegate an entire application by assigning an application share of 1 to a specific application on single child. If a parent delegates an application, the child can remove all of the application's resources from its aggregated resource information, and the parent no longer needs to monitor the application's performance information and placement. If the child is no longer able to achieve the required placement, it can delegate the application upward towards its parent, once again making it responsible for the application's management.

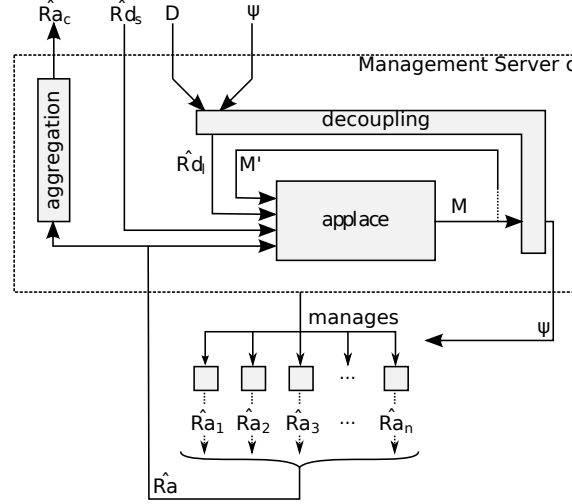


Figure A.4: The origin and destination of the different appplace in- and outputs. A single management server, containing the appplace-function, aggregation and decoupling mechanisms is shown.

A.5.2.3 Overview

In summary, to execute the centralized algorithm, we retain non-varying resource demand, we aggregate resource availability and we decouple the different management levels by using application shares instead of resource demand. Figure A.4 illustrates how the various inputs are combined in a management server. A single management server is shown, together with its various inputs and outputs.

A.6 Evaluation Results

We modified an algorithm from the literature [2] in a hierarchical fashion. The original algorithm operates in a centralized fashion with a complexity of $O(n^{2.5})$ with n the number of servers $|S|$, causing scalability issues as the server count increases. The hierarchical use of this algorithm solves these issues by introducing clusters grouping servers. The centralized algorithm is then executed on smaller clusters. We compared the performance of the hierarchical system using multiple values for C_{max} . We choose $C_{min} = \frac{C_{max}}{2}$. We considered application placement using memory and CPU as requirements with memory as loose requirement and CPU as strict requirement.

We used simulation of the cluster to evaluate the centralized and hierarchical algorithms where parallel tasks were executed sequentially and network overhead was simulated. Communication overhead between management nodes was simu-

lated using a Gaussian distribution with mean 30ms and variance 10. Execution times were measured using a Linux server with an Intel Core i3 CPU (2.93GHz) with 4GiB of memory.

For every datapoint, multiple (20) random datacenters and sets of applications were generated, after which the measurements for the different datacenters were averaged. A random server has a CPU capacity, randomly picked from the set $\{1\text{GHz}, 1.6\text{GHz}, 2.4\text{GHz}, 3\text{GHz}\}$ and a memory capacity from the set $\{1\text{GB}, 2\text{GB}, 3\text{GB}, 4\text{GB}, 8\text{GB}\}$. A set of random applications A is generated. Individual applications a are randomly generated by choosing a memory capacity from the set $\{400\text{MB}, 800\text{MB}, 1.2\text{GB}, 1.6\text{GB}\}$ and allocating a random size $\theta_a \in [0..1]$ to it. The total application size, $\Theta = \sum_{a \in A} \theta_a$ can then be used to determine the total application share $\frac{\theta_a}{\Theta}$. Using a total CPU load for the entire datacenter L_{CPU} and individual application shares, we can then determine the demand for the application $Ra_a^{CPU} = L_{CPU} \frac{\theta_a}{\Theta}$. We used $w = 0.9$ as experiments have shown this weight yields good results. High weights lead to good estimations giving a higher weight to E^{low} , as discussed in Section A.5.2.1. A weight of 1 would lead to a too low estimate and would make it impossible to improve on bad allocations.

We evaluated a server and application configuration where application demands remain static. We evaluate the speed and quality of a single allocation on a datacenter with a heavy load ($L_{CPU} = 1$). We measure allocation quality by comparing the average application satisfaction $Q_a = \frac{\sum_{s \in S} M_{s,a}^{CPU}}{Ra_a^{CPU}}$ of the different allocation strategies. Allocation quality is measured by comparing the desired amount of resources with the allocated amount of resources. A satisfaction of 1 means all demands are satisfied. For this test we used an equal number of randomly generated applications and servers. As our architecture assumes that the management system itself is executed on a cloud instance, and we work using $L_{CPU} = 1$ achieving full satisfaction will be impossible as part of the capacity will be consumed by the management system itself, pushing the centralized algorithm to its limits.

In Section A.5.2.1 we mentioned that the quality of \hat{Ra}_c^r increases as multiple placements are made. Figure A.5 illustrates the effect of this: subsequent placements increase in quality until a threshold is reached, after which placement quality stagnates. The first placement has a relatively bad quality, caused by the low quality of the initial estimate. The second placement greatly increases placement quality and after a third placement the quality of the placement has a value near its maximum. Because of this we will execute hierarchical placement three times in following experiments, leading to a high quality at the cost of higher execution times.

We compared the performance of the centralized approach with that of five different variants of the hierarchical approach, with $C_{max} = 50$, $C_{max} = 100$ and $C_{max} = 200$. The allocation speed of the different techniques is illustrated in Figure A.6. As we expected, the hierarchical approach executes faster than the centralized approach. As indicated in Figure A.7, the average satisfied demand of

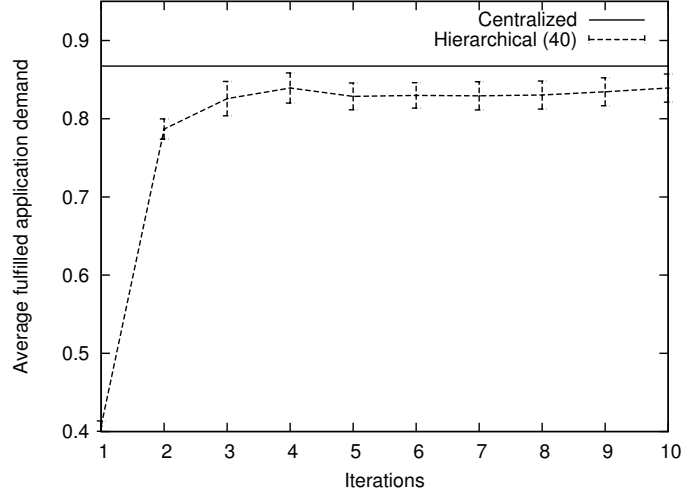


Figure A.5: The quality of the allocation after subsequent placement calculations ($C_{max} = 20$, $|S| = 50$). Standard errors are shown as well.

applications is best provided for by the centralized algorithm once $|S| \gg C_{max}$. As long as $C_{max} > |S|$ the hierarchical approach is the same as the centralized algorithm. Once branching occurs a tree is formed. This temporarily *increases* both placement quality and allocation cost, as the management system repeats the allocation multiple times. As the datacenter size increases, allocation performance decreases while the difference between centralized execution times and hierarchical execution times increases. The higher C_{max} , the higher the placement quality, but the slower the execution.

Branching causes two types of performance penalties. As additional servers are used for the management process, they can no longer be used to execute applications, decreasing maximum achievable application satisfaction. Furthermore, information is fragmented by the process, allowing for over- or under-allocation of application demand on clusters.

The effects of the different performance penalties in the hierarchical approach are illustrated in Figure A.8. Here we compare the satisfied demand of the centralized algorithm with that of the hierarchical approach and that of an adjusted centralized algorithm, where servers used in hierarchical management are kept idle. In this case we used a management tree with an unrealistically low C_{max} to illustrate both types of performance penalties. While the centralized approach uses all servers but one, where the algorithm itself executes, the adjusted centralized algorithm does not use any of the servers used in the management hierarchy. Because less servers can be used for actual application execution, the achievable

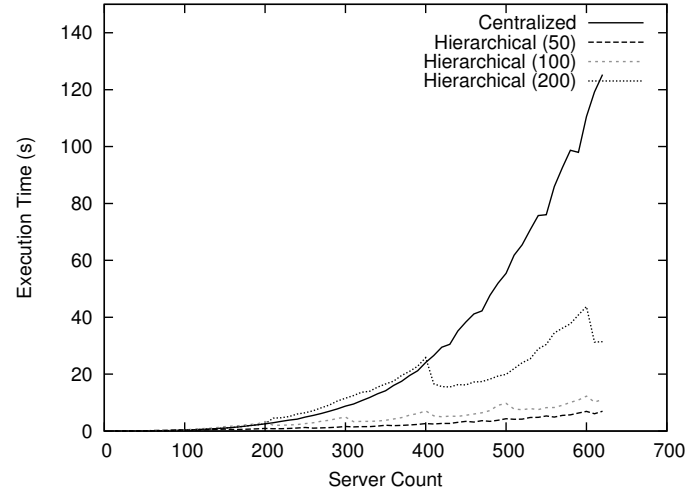


Figure A.6: Execution time of the hierarchical and centralized allocation strategies with varying server and application counts ($|A| = |S|$).

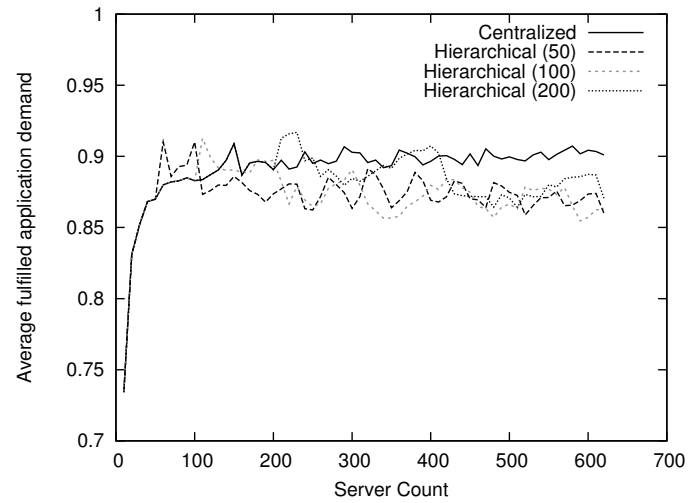


Figure A.7: Comparison of the average satisfied demand of the different allocation strategies ($|A| = |S|$).

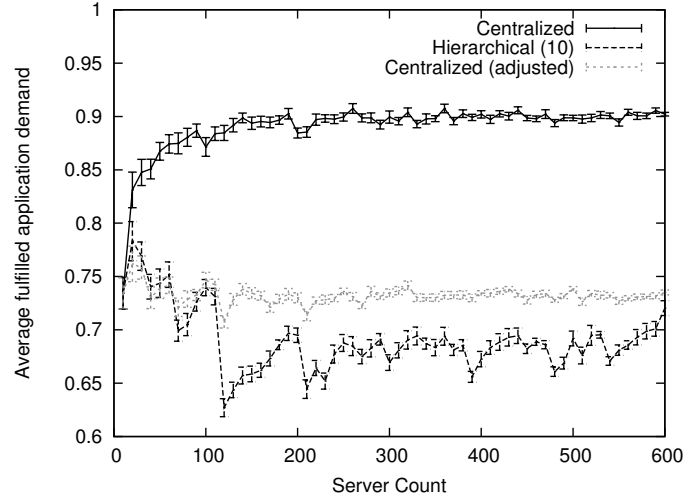


Figure A.8: Illustration of the management overhead and performance penalties induced by the hierarchy with a very low branching factor ($C_{max} = 10$).

placement quality is lower. As the hierarchical placement uses the same servers as management servers, the adjusted centralized placement offers an upper bound for the placement quality. The quality difference between the centralized approach and the adjusted centralized approach (in this specific case $\pm 20\%$) is caused by the management hierarchy itself and stems from the choice to place the management infrastructure on the cloud itself. Not doing so would require a separate management infrastructure which would be less dynamic. The quality difference between the adjusted centralized approach and the hierarchical approach is caused by fragmentation of information and under-allocation. Picking higher a C_{max} causes performance penalties to decrease significantly while increasing allocation performance by repeated execution of the allocation algorithm, as illustrated in Figure A.9, where hierarchical execution at times even surpasses centralized execution due to repeated executions.

The performance of the hierarchical system is still impacted by the number of applications, as shown in Figure A.10, where a constant number of applications is used ($|A| = 50$) and significantly better performance is achieved. The cause of this is that the number of applications also has an impact on the performance of the application placement algorithm. When considering a first placement, all applications need to be taken into consideration, leading to very high placement times. After this, some application responsibilities are delegated to specific instances, decreasing costs for subsequent placements. This implies that when many applications are in use, a lower branching factor should be chosen as this leads to more management

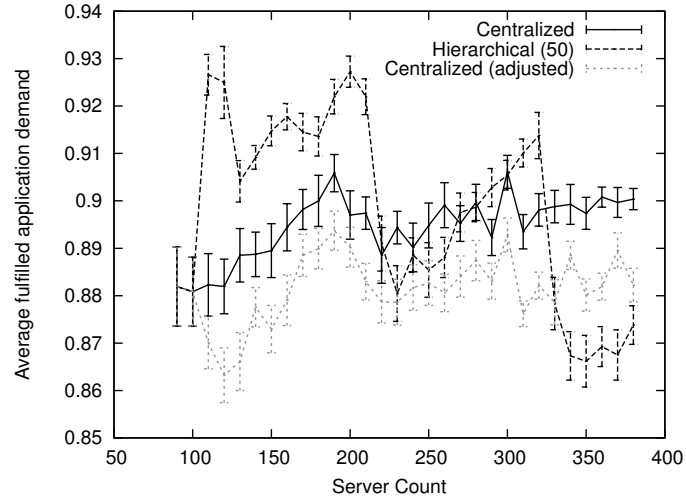


Figure A.9: Illustration of the management overhead and performance penalties with a higher branching factor ($C_{max} = 100$).

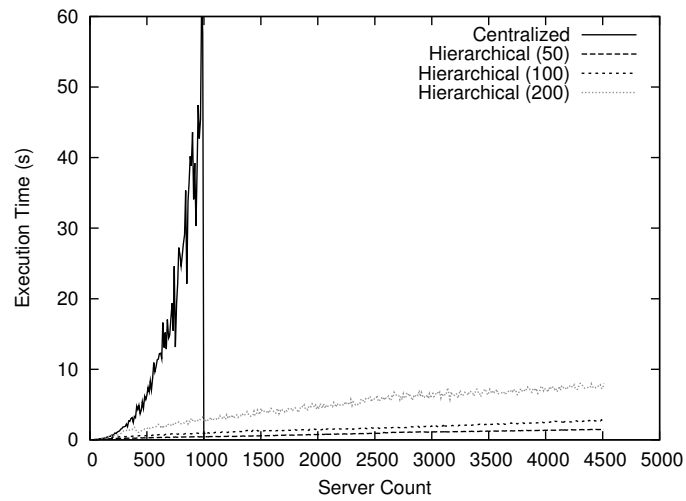


Figure A.10: Execution time of the hierarchical and centralized allocation strategies with fixed application counts ($|A| = 50$). Execution time of the centralized algorithm was measured up until 1000 servers.

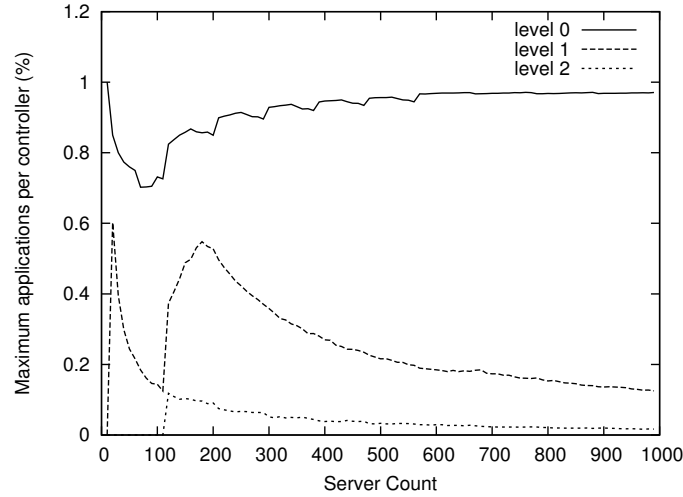


Figure A.11: The maximum number of applications known per node at different management levels ($C_{max} = 10$).

nodes and thus more delegation of applications.

Figure A.11 illustrates the number of applications used in application placement per level. The graph illustrates the maximum number of applications used in application placement at a given tree level. The lower this number, the faster a placement at this level can occur. From the graph, we see that a large part of applications continues to be managed at the root level, but at lower levels, significantly less applications are managed per-node. This implies lower-level nodes can execute placement much faster. This enables the system to react fast based on local data, swiftly yielding a local solution, while still maintaining a globally good solution once higher levels have executed application placement.

A.7 Conclusion

In this appendix we presented a hierarchical management system for cloud environments. Management nodes automatically order themselves in a structure inspired by that of B-Trees and each node executes a centralized placement algorithm for which inputs are generated by the management system.

The centralized approach leads to higher placement quality at the cost of higher execution times. As datacenters scale, the execution time also increases, leading to slow reactions to changing environments. At this point, using the hierarchical approach makes sense, as it is much faster and, thus, more scalable. The number of servers managed by each node has a large impact on the execution speed and on the

quality of allocation, as it directly influences both the number of servers used in the management system and the quality of the placement itself. These results are in accordance with our earlier work concerning hierarchical management. Introducing hierarchies increases scalability of management systems, and eases distribution of context, in this instance calculated application demands.

In future work, we intend to study and improve the robustness of the hierarchical management system. It would also be useful to add adaptiveness to the system by dynamically restructuring the management tree and adjusting system parameters, such as branching factor and estimation weights. These adjustments should lead to improved resource estimates, better resource allocations and faster placements. We also need to examine how the techniques demonstrated in this appendix can be applied to other centralized cloud management algorithms.

References

- [1] J. Famaey, S. Latré, J. Strassner, and F. De Turck. *A hierarchical approach to autonomic network management*. In Proceedings of the 12th IEEE/IFIP Network Operations and Management Symposium Workshops (NOMS 2010), pages 225–232. IEEE, apr 2010. doi:10.1109/NOMSW.2010.5486571.
- [2] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici. *A scalable application placement controller for enterprise data centers*. In Proceedings of the 16th International Conference on World Wide Web (WWW 2007), pages 331–340. ACM, may 2007. doi:10.1145/1242572.1242618.
- [3] F. Wuhib, R. Stadler, and M. Spreitzer. *Gossip-based Resource Management for Cloud Environments*. In Proceedings of the 6th International Conference on Network and Service Management (CNSM 2010), pages 1–8. IEEE, oct 2010. doi:10.1109/CNSM.2010.5691347.
- [4] Y. Li, F.-H. Chen, X. Sun, M.-H. Zhou, W.-P. Jiao, D.-G. Cao, and H. Mei. *Self-Adaptive Resource Management for Large-Scale Shared Clusters*. Journal of Computer Science And Technology, 25(5):945–957, sep 2010. doi:10.1007/s11390-010-1075-6.
- [5] J. Rolia, A. Andrzejak, and M. Arlitt. *Automating Enterprise Application Placement in Resource Utilities*. In M. Brunner and A. Keller, editors, Self-Managing Distributed Systems, volume 2867 of *Lecture Notes in Computer Science*, pages 118–129. Springer Berlin Heidelberg, 2003. doi:10.1007/978-3-540-39671-0_11.
- [6] T. Kimbrel, M. Steinder, M. Sviridenko, and A. Tantawi. *Dynamic Application Placement Under Service and Memory Constraints*. In S. Nikolettseas, editor, Experimental and Efficient Algorithms, volume 3503 of *Lecture Notes in Computer Science*, pages 391–402. Springer Berlin Heidelberg, 2005. doi:10.1007/11427186_34.
- [7] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. *Dynamic placement for clustered web applications*. In Proceedings of the 15th International Conference on World Wide Web (WWW 2006), pages 595–604. ACM, may 2006. doi:10.1145/1135777.1135865.
- [8] D. Carrera, M. Steinder, I. Whalley, J. Torres, and E. Ayguadé. *Utility-based placement of dynamic web applications with fairness goals*. In Proceedings of the 11th Network Operations and Management Symposium (NOMS 2008), pages 9–16. IEEE, apr 2008. doi:10.1109/NOMS.2008.4575111.

- [9] C. Adam and R. Stadler. *Service Middleware for Self-Managing Large-Scale Systems*. IEEE Transactions on Network and Service Management, 4(3):50–64, dec 2007. doi:10.1109/TNSM.2007.021103.
- [10] C. Low. *Decentralised Application Placement*. Future Generation Computer Systems, 21(2):281–290, feb 2005. doi:10.1016/j.future.2003.10.003.
- [11] S. Schneider, A. Meisel, and W. Hardt. *Communication-Aware Hierarchical Online-Placement in Heterogeneous Reconfigurable Systems*. In Proceedings of the 2009 IEEE/IFIP International Symposium on Rapid System Prototyping (RSP 2009), pages 61–67. IEEE, jun 2009. doi:10.1109/RSP.2009.23.
- [12] B. Hudzia, M.-T. Kechadi, and A. Ottewill. *TreeP: A Tree Based P2P Network Architecture*. In Proceedings of the 2005 IEEE International Conference on Cluster Computing, pages 1–15. IEEE, sep 2005. doi:10.1109/CLUSTER.2005.347022.
- [13] R. Bayer and E. McCreight. *Organization and Maintenance of Large Ordered Indices*. In Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control, SIGFIDET '70, pages 107–141. ACM, 1970. doi:10.1145/1734663.1734671.

B

Migrating Legacy Software to the Cloud: Approach and Verification by means of Two Medical Software Use Cases

P.-J. Maenhaut, H. Moens, V. Ongenae and F. De Turck

Accepted for publication in Wiley Journal of Software: Practice and Experience (SPE)

In this appendix, a cloud migration strategy is discussed and evaluated. In this approach, a distinction is made between the steps needed to migrate an application to a remote cloud environment, and the steps to add multi-tenancy to the migrated application are discussed. By executing these changes, a client-hosted application can be converted to a multi-tenant SaaS application. In the generic approach, the Feature-Based Binary (FBB) approach, which was presented in Chapter 2 is utilized to model application customization, and the impact analysis strategy presented in Chapter 6 is used to analyze the impact of the migration. The generic approach is verified by means of two case studies, a commercial medical communications software package mainly used within hospitals for nurse call systems and a schedule planner for managing medical appointments. Both case studies are subject to stringent security and performance constraints, which need to be taken into account during the migration.

B.1 Introduction

Cloud computing is a technology that enables elastic, on-demand resource provisioning. Over the last few years many companies have used clouds to build new highly scalable systems. However, legacy applications can also benefit from the advantages of cloud computing, and there is a general trend for moving applications to a cloud infrastructure, consolidating hardware, saving costs and allowing applications to react faster to sudden changes in demands. With the recent evolution of cloud computing [1] and Software-as-a-Service (SaaS) in particular, an elastic, scalable multi-tenant architecture has gained popularity [2]. Elastic systems are able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner. With cloud computing, an optimal usage of available resources is recommended to reduce operating costs, as the infrastructure provider usually charges for the number of instances used. SaaS is a software delivery model in which the software and associated data are centrally hosted on the cloud, and the end-users are typically accessing the software through the browser or by using a thin client. As the number of clients grows, a scalable architecture for both the application and data is needed.

Multi-tenancy [3] enables the serving of multiple clients or tenants by a single application instance. The major benefits include increased utilization of available hardware resources and improved ease of maintenance and deployment. Without a multi-tenant architecture, the cost savings using cloud computing are limited for applications requiring continuous availability, as for every new client (tenant), a separate Virtual Machine (VM) instance would have to be provisioned. This instance must then be available at all times, even if it is only used sporadically. Also, as every tenant has a dedicated instance, some resources would be wasted, especially for smaller clients. Using a multi-tenant architecture, a SaaS application could run on few instances that are shared between the different users, and the number of instances could dynamically grow with the current demand. Smaller tenants could be co-located on a single instance, minimizing costs and maximizing resource utilization.

Therefore, when migrating applications to the cloud, it is recommended to adapt the legacy software to support multi-tenancy. Some changes to the architecture will be necessary, coming at a one-time cost, but this cost is overruled by the long-term benefits. Apart from adapting the legacy software for supporting multi-tenancy, some other changes may be needed to support the migration to a public or hybrid cloud, as every Platform-as-a-Service (PaaS) or Infrastructure-as-a-Service (IaaS) provider will have its own limitations and possibilities.

In this appendix we propose an approach for both migrating applications to a hybrid or public cloud, and for adding multi-tenancy to the existing software with a minimal overhead. We verify our approach using two different case studies of legacy

medical applications which are migrated to the cloud, and discuss the required changes. We describe the advantages and disadvantages of moving components of the software to the public cloud, and evaluate the migration costs.

In the next section of this appendix we will discuss related work. Afterward, in Section B.3, we will present the approach for both migrating legacy software to the cloud and adding multi-tenancy. We verify this approach in Section B.4 and Section B.5 using two different case studies. In Section B.6, we discuss our approach and present our evaluation results. In Section B.7, we state our conclusions and discuss avenues for future research.

B.2 Related Work

In previous work [4], we described the steps required to migrate an existing .NET-based application to the Windows Azure public cloud environment, and proposed a specific approach for adding multi-tenancy to the application. In this appendix, we propose a generic migration approach for migrating legacy applications to the cloud. We describe the different steps of our approach in detail, and verify our approach by means of two case studies. In this appendix we also present an extended discussion and evaluation based on the results from the two case studies.

An approach for partially migrating applications to the cloud is presented in [5], together with a model to explore the benefits of a hybrid migration approach. The approach focuses on identifying components to migrate, taking into account various rules such as performance and security. We also focus on migration to a hybrid or public cloud, but extend their approach by going into detail about the complete migration process, and not only selecting the components to migrate. We also present an approach for adding multi-tenancy to the application to optimal benefit from the migration to a public cloud.

When migrating software to the cloud, some choices have to be made. Different cloud computing service models exist, each having their own advantages and limitations. Figure B.1 provides an overview of the different cloud service models. The legacy software could for example be fully migrated to a public cloud, or a hybrid approach could be used. When it comes to public cloud providers, CloudCmp [6] offers a system for comparing the performance and cost of the different providers. For the implementation, the authors use computation, storage and network metrics. For the storage metrics, they selected some benchmark tasks and measured the response times, throughput, time to consistency and cost per operation.

Cost savings and other organizational benefits and risks of migration to IaaS are discussed in [7]. We however don't limit our approach to migrations to an IaaS provider, but also consider migrations to a PaaS platform. When using an IaaS provider, the customer has full access to the operating system, middleware and runtime, hosted on a virtual machine. On the other hand, when using a PaaS

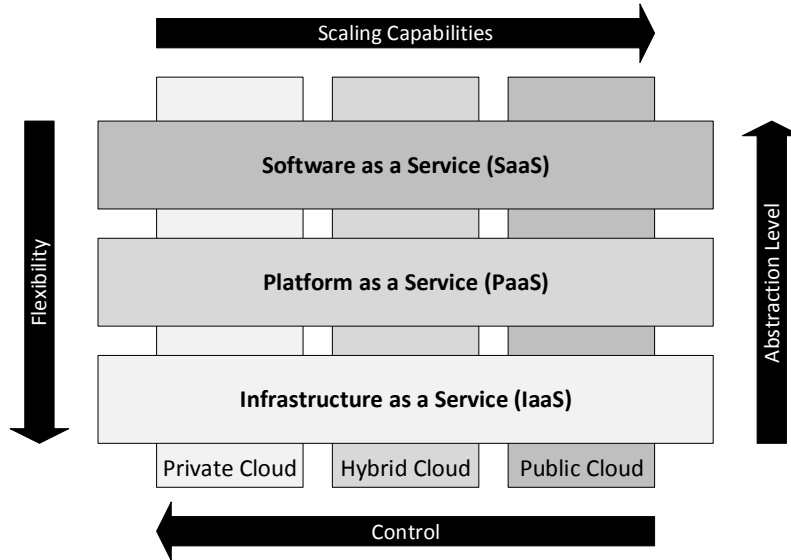


Figure B.1: An overview of the different cloud service models used in cloud computing.

provider, the customer only manages the application and data, which brings some limitations, such as the selected operating system and supported frameworks and libraries.

In [8] a checklist is presented that can be used to determine whether applications are compatible with a chosen PaaS provider. The approach is evaluated by three case studies where a Java application and two Python applications are migrated to Google App Engine. Three different and representative PaaS platforms are compared in [9], based on a practical case study, with respect to their support for SaaS application development. In this appendix, we focus on how complex applications can be executed on the public cloud, and for our case studies, we go into detail on migrating two different legacy applications. We don't limit our work by determining whether the applications are compatible with the selected provider, but also describe the different steps required in detail. Furthermore, we describe how multi-tenancy can be added, making it possible to better utilize individual application instances.

As our first case study handles a legacy application written in .NET, we selected Windows Azure to host some components of the legacy software. The migration of an on-premise web application to Windows Azure is described in [10], together with a comparison of the application's performance when deployed to a traditional Windows server versus its deployment to Windows Azure. While the cloud migra-

tion of a .NET application requires limited effort, Azure has no built-in support for multi-tenancy, so it must be added during the migration process. In this appendix, we discuss both the steps needed to migrate an application to the cloud, and the steps needed to add multi-tenancy to the application.

To support highly customizable SaaS applications, we use a software product line based customization approach, which we have previously discussed in [11], [12], [13] and [14]. In this approach, variability is modeled by defining multiple features and the relations between them. These features are then associated with separate code modules that are deployed separately. The application is then composed out of these multi-tenant components, resulting in an application that is both customizable and multi-tenant. For changes that do not impact the performance of the application, a multi-tenancy enablement layer can be used, which amongst others can be used for data isolation, feature management and tenant-specific customizations [15].

In [16] we focused on the scalability of tenant data in multi-tenant applications and the impact on the performance of the application. The outcome of this research is used in [17] to build an abstraction layer for achieving high scalability for the storage of tenant data. This layer uses data allocation algorithms to determine an acceptable allocation of tenant data to different databases. The presented solution can be used for decoupling the databases and the management of tenant data, two of the steps in the approach presented in this appendix.

B.3 Migration Strategy

In this section, we describe both the steps needed to migrate an existing application to the cloud, and to redesign the application to support multi-tenancy. We start this section by briefly describing the concept of multi-tier architectures, a popular software architecture used by many applications, which we will refer to later in this section. Next we discuss the different steps of our approach, as summarized in Figure B.2.

Many applications are designed using a multi-tier architecture, where the application is separated into multiple layers. A typical multi-tier architecture consists of 3 layers: the client layer, the business logic layer and the database layer. We refer to this basic layered architecture as the *3-Tier architecture*. Most layered applications will have more than 3 layers, as more layers can be easily added to the architecture if needed. For example, when working with multiple database instances, an extra data access layer can be added between the business logic and database layer, responsible for load balancing and selecting the correct instance. Other architectures are possible, but in the remainder of this section, we will start from the *3-Tier architecture*.

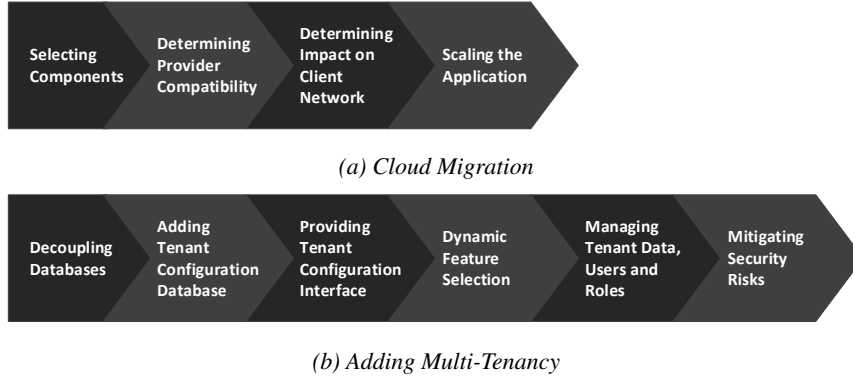


Figure B.2: A summary of the different steps required to migrate an existing application to the cloud, and to add multi-tenancy to the application. The different steps are described in detail in sections B.3.1 and B.3.2.

B.3.1 Cloud Migration

The process to migrate an existing application to a public or hybrid cloud can be summarized in a few steps, illustrated in Figure B.2a and described below.

B.3.1.1 Selecting Components

The first step during the planning phase should be to select the components of the software to migrate to the public cloud, as described in [5]. Both components of the business logic layer and the data access layers can be selected. The selection can happen based on the quality attributes of the application, to guarantee the required Quality of Service (QoS) and Service Level Agreements (SLAs). In case the whole application is being migrated, this step is quite straightforward, but when only some components of the application are selected, the architecture might need to be reviewed. Special attention has to be paid to the communication between the different components, as the communication between the dedicated servers and the public cloud might need extra security, extra bandwidth, and usage of standardized protocols. When using a Service-Oriented Architecture (SOA), communication between the different modules could for instance make use of SOAP or REST over HTTPS. Possible communication between the client layer and the components of the business logic layer should also be secured.

Figure B.3 illustrates an example of the possible communication between the different components after migration to a hybrid cloud. The components are represented by server instances. Dark arrows denote communication where extra attention has to be paid regarding security and available bandwidth.

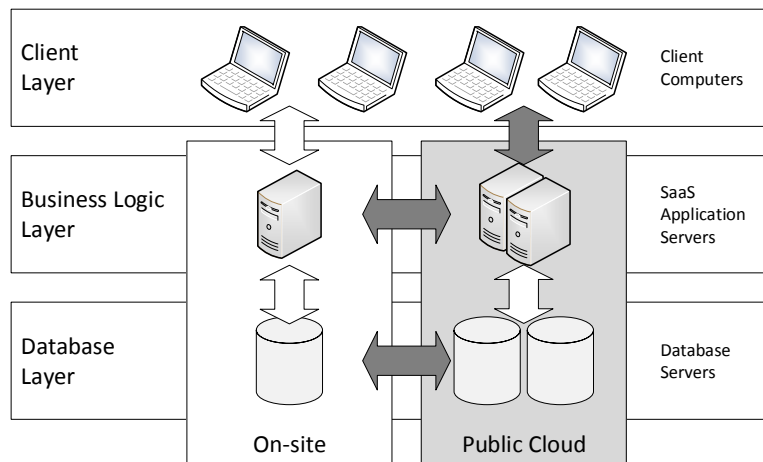


Figure B.3: An illustrative example of the possible communication between components after migration to a hybrid cloud. Dark arrows denote communication that should be secured.

B.3.1.2 Determining Provider Compatibility

Apart from selecting the components for migration, some extra changes might be needed for migrating the application to the public cloud. Every public PaaS provider will typically have its own limitations and possibilities, so during the planning phase of the migration, it is best to verify that the provider will support all features of the software. In case no suitable PaaS provider can be found, an IaaS provider could also be selected to host some components of the application, but this again results in more maintenance overhead for the application provider. When comparing different providers, for example by using CloudCmp [6], the balance should be made between the advantages of the selected provider and the overhead due to needed changes to the application. Different public cloud providers should be considered and evaluated, for example by using a small Proof of Concept (PoC), and the advantages of using a PaaS provider should also be weighted against the increased control gained when using an IaaS provider.

B.3.1.3 Determining Impact on Client Network

A side effect of the migration to cloud environments is that communication between some components of the software might need to pass over the Internet, especially

when migrating the software to a hybrid cloud. As a result, more traffic bandwidth at the client network might be required. Before deploying the service, it is important to perform an impact analysis whenever the client configurations are changed. We have previously covered this in-depth in [18].

B.3.1.4 Scaling the Application

When an instance is overloaded, extra instances can be added (up-scaling) and removed (down-scaling) in a few steps. This concept is often referred to as the elasticity of the (public) cloud. Some public cloud providers offer out of the box load-balancing and/or scaling, other providers only provide limited load-balancing possibilities, together with an Application Programming Interface (API) to support up-scaling and down-scaling from within the application.

When selecting the components to migrate, it is a good idea to take account the scalability of the application. Components which should be highly scalable could be good candidates for migration to the public cloud, as the public cloud offers an unlimited resource pool. The application should also support decoupling of the components, and handle synchronization and conflicts in data. Possible bottlenecks should be eliminated, as these could break the whole scalability of the application. Reviewing the architecture of the application to better support scalability will bring some overhead, but the advantages on the long term will outweigh this one-time investment.

B.3.2 Multi-Tenancy

In this subsection, the steps required to add multi-tenancy to an existing application are discussed. These steps are also summarized in Figure B.2b.

B.3.2.1 Decoupling Databases

As multiple tenants will use the same application instance, each tenant will have its own application data stored in a shared or dedicated database instance. Using shared database instances is cheaper, while dedicated databases will lead to better performance and higher security, but at a higher cost. To connect to the correct database, a connection string is associated with each tenant. These connection strings can for example be stored in a shared database.

The application database needs to be decoupled, and support for multi-tenancy needs to be added to the data tables in case of shared instances. Also, the application needs to be modified to support dynamic database binding. An extra component can be added to the application, the *data access component*, responsible for both the correct handling and access control of all data requests by the application. Figure B.4 illustrates a possible architecture of the application after decoupling the

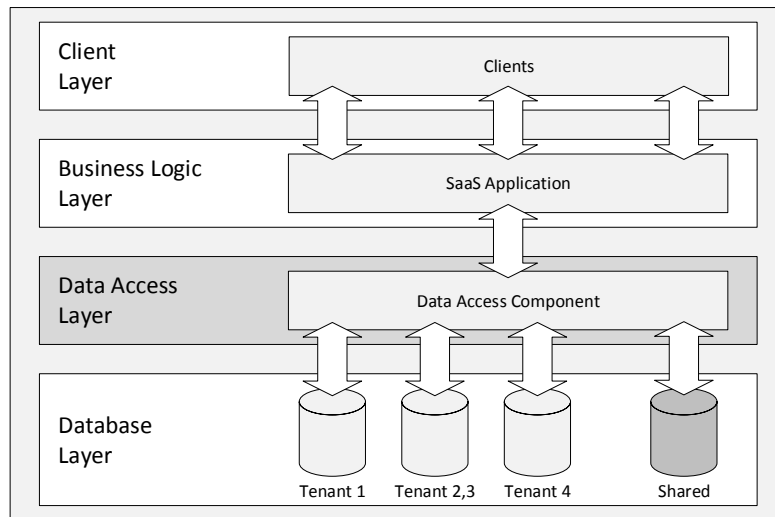


Figure B.4: Possible architecture of the application after decoupling the databases.

databases. The data access component is added to a new layer, the data access layer, situated between the business logic layer and the database layer.

For the design of the data access layer, the abstraction layer presented in [17] could be used. This abstraction layer mainly handles the security and isolation of tenant data, and the scalability of the database layer. In our approach, we partition tenant data over multiple database instances based on the tenant. By doing so, tenants can still store their data in a dedicated on-site database instance, for example to comply with regulatory policies on data. Partitioning the data based on the tenant also provides a clear separation of tenant data. For large tenants with a dedicated database instance, the performance of the database will not be influenced by other tenants. As the number of tenants using a shared database instance will be limited, the possible damage due to an information leakage is also minimized. Different SLAs can be provided, based on the scenario of a dedicated or shared database instance.

B.3.2.2 Adding Tenant Configuration Database

A new database, which we refer to as the *tenant configuration database*, needs to be added to store general information about all tenants. The connection strings introduced in the previous steps will be stored in this database, together with specific information and configuration parameters such as billing and contact information,

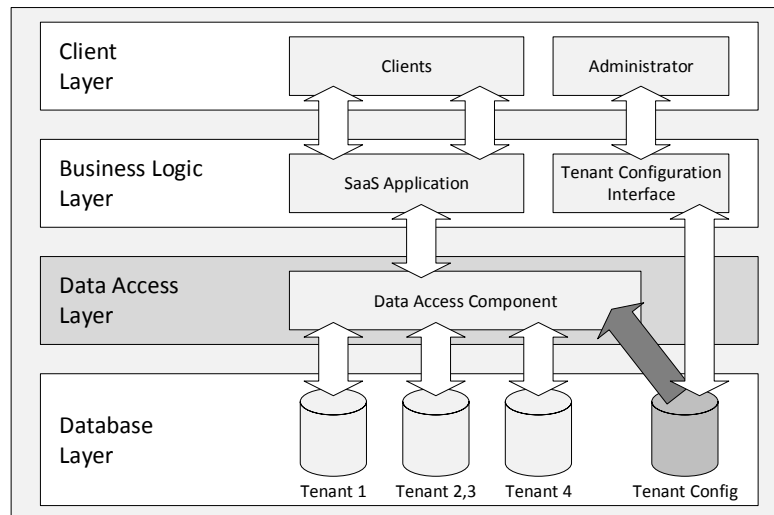


Figure B.5: Possible architecture of the application after adding the tenant configuration interface.

and the selection of features for the tenant as described in Section B.3.2.4. While this database is shared between all tenants, it only contains minimal information, and is only accessed sporadically as the information inside this database can be cached by the application, so it should not become a bottleneck [16] [18].

B.3.2.3 Providing Tenant Configuration Interface

Adding multi-tenancy to the application makes it possible to more flexibly select the application features used by different clients, as the tenant configuration is stored in the shared *tenant configuration database*. It is however also necessary to create a separate application, the *tenant configuration interface*, which can be used by tenant administrators to modify the tenant configuration. This interface will be used to create, modify and delete tenants in an easy way, and change the configuration of a single tenant, for example the selection and configuration of features and the connection string of the tenant.

Ideally, the *tenant configuration interface* is the only component which has read/write access to the *tenant configuration database*, as the legacy application should only require read access. Figure B.5 illustrates the changes to the architecture after adding this interface.

B.3.2.4 Dynamic Feature Selection

An application can have multiple features which will be dynamically loaded at start-up. As every tenant can have its own selection of features, and a tenant-specific configuration for these features, a tenant administrator should be able to select and configure these features using the *tenant configuration interface* introduced before.

The application itself needs to support the dynamic selection of features. For example, some features might require additional modules, and the application needs to support dynamic loading (and unloading) of the corresponding modules. Also, the user interface of the application might need to be automatically adapted for the different tenants, based on their configuration, representing the tenant's feature selection. The different features might run on the same instance of the application, or on dedicated machines. In the latter case, feature placement algorithms can be used to determine the optimal solution. We have previously covered this in [11].

B.3.2.5 Managing Tenant Data, Users and Roles

Every tenant using the application will typically have its own data, users and custom roles. The users could be stored in a shared common database or in the tenant database, or the application could support external identity providers. In case the users are stored in a global shared database, or when an external identity provider is used, the application could provide single sign-on scenarios. In case the users are stored in the tenant database, an administrator should be able to create and modify users and their corresponding roles from within the application. By introducing multi-tenancy, a tenant administrator role with permissions to create and modify the tenant configuration using the *tenant configuration interface* is required, different from the administration roles within a single tenant. These tenant administrators can be stored in the shared *tenant configuration database* and should have limited access to the multi-tenant application for every tenant if required. The management of users and roles could be moved to the *tenant configuration interface*, or could stay inside the application, depending on the application's requirement and the software license model. The question arises how and where to store the tenant data and the different users and roles. Different approaches are possible, and we have previously covered this in-depth in [16].

B.3.2.6 Mitigating Security Risks

A major disadvantage of using multi-tenancy is an increased security risk, as by definition multiple tenants will use the same application instance. These risks can be mitigated in multiple ways:

- **Implementing URL-based filtering of application requests, taking into account the permissions of the user and tenant.** Every tenant can have

its own URL, for example by having a customized sub-domain. When a client wants to access the data of a specific tenant, the access module of the application needs to verify if the authenticated user and its corresponding tenant have access to the requested data (the requested URL), to eliminate unauthorized access.

- **Separating the tenant configuration from tenant data.** Because the tenant data is stored in a different database instance as the tenant configuration, it is easier to configure tenant-specific access at the database level. Each tenant will have its own connection string, and the associated credentials will only have access to the tenant's database.
- **Offering single-tenant instances of specific components at a higher cost.** If the above methods are not deemed sufficient, tenants with a huge amount of confidential data can have single-tenant instances at a higher cost. Having a dedicated instance clearly improves security, as the tenant's data is not only virtual but also physical isolated from other tenants. Because the connection strings are stored separately for each tenant in the shared *tenant configuration database*, these connection strings can either point to a shared or dedicated database.

B.4 Case Study 1: Medical Communications System

B.4.1 Introduction

In this section we verify our presented approach using the case study of a Medical Communications (MC) system. The MC system is responsible for the correct functioning of all communication peripherals located in a medical environment. The central functionality of this system is the *nurse call* system. The basic concept of a *nurse call* system is simple: a call device is located in every room. When a button is pressed on the device, a message is sent to a controller after which nurses are notified of the call. This concept can be enhanced by using ontologies and semantic reasoning to identify the urgency of a call or select the nurses to notify in a more intelligent way [19] [20] [21].

A *nurse call* system consists of many different elements, installed within a hospital. These elements include amongst others 1. end user equipment installed in the rooms, which patients can use to contact hospital personnel, and terminals used by the personnel; 2. embedded servers, used to communicate between the terminals and management servers; and 3. servers for logging, registration and visualization. Figure B.6 illustrates an example of the architecture of a nurse call system, with the possible communication between the different components when a patients calls a nurse shown in arrows.

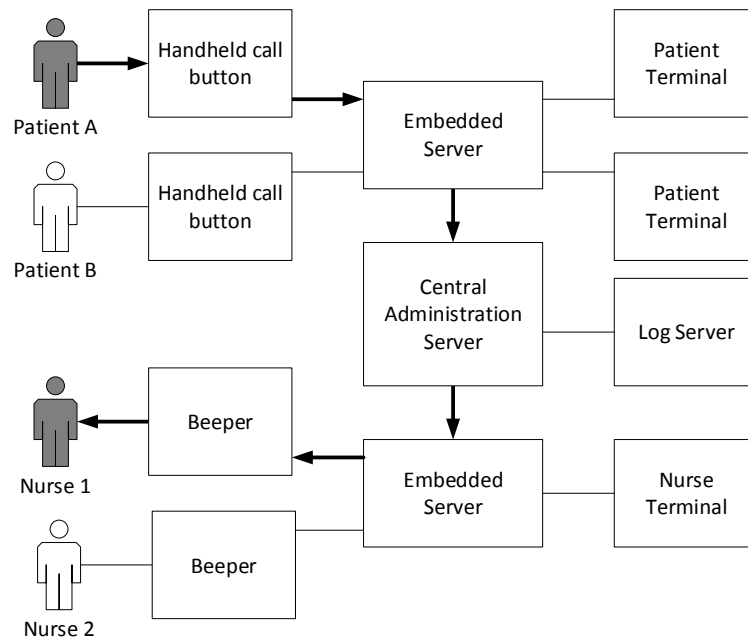


Figure B.6: An example architecture of a nurse call system, with the communication between the different elements when a patient makes a call.

While the center of the MC application is the *nurse call* system, additional services, such as intercom, video over IP, access control and other health services are being offered as well. Currently, the MC system is installed in multiple locations, ranging from big hospitals to small rest-houses. The cost of installing dedicated servers, and the corresponding maintenance is quite high. Migrating a part of the system to the cloud will minimize the cost, by eliminating the need of many of the dedicated servers, making it possible for smaller hospitals and rest-houses to afford the system. However, considering the medical use case, the MC application is subject to stringent security and performance constraints, which need to be taken into account when the components to migrate to the cloud are selected.

B.4.2 Cloud migration

B.4.2.1 Selecting Components

The MC software consists of two main components: the *device manager* and the *administration service*. The *administration service* is the main application, and is used to manage the different features and devices installed within the hospital. The *device manager* is a dedicated hardware box, running different modules mainly

written in C++ for communicating with the different peripherals installed within the medical environment. The modules for the different features are dynamically loaded on start-up, and can be configured from the *administration service*. Figure B.7a shows the initial architecture of the application. The device manager and the different peripherals communicate over Ethernet, using a custom proprietary secure protocol.

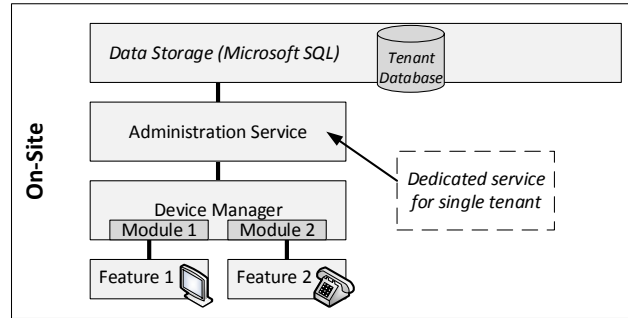
For our PoC, we selected the *administration service* and its corresponding database instances for migration to the public cloud. The MC system has a fall-back mechanism, allowing the *device manager* to operate standalone in case the *administration service* is not available. Because the *device manager* communicates directly with the devices, migrating this component to the cloud would be tricky, as the devices need to operate when no connection to the public cloud is available. Passing all communication between the device manager and the peripherals over the public Internet would also result in slow response times, and could make the system unreliable. However, as most of the processing is done in the devices, a single *device manager* will be sufficient to control all devices in a small or medium environment. If the peripherals could be adjusted to work standalone when the device manager is unavailable, migrating the *device manager* could also be an option in the future, but for this PoC, we started focusing only on decoupling the *administration service*.

After adding multi-tenancy to the application and migration to the cloud, a new component is introduced, the *tenant configuration interface*. The reviewed architecture after adding multi-tenancy and migration is shown in Figure B.7b. Because every tenant has its own features, the user interface of the *administration service* is automatically adapted for the different clients based on the tenant configuration.

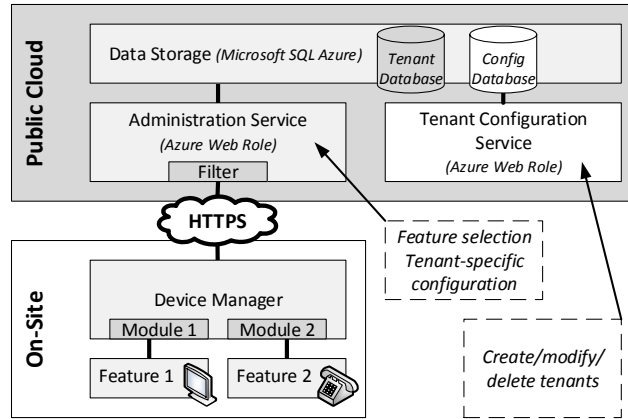
B.4.2.2 Determining Provider Compatibility

As the application is written in .NET, migrating the *administration service* to Microsoft Azure seemed like an evident choice. Microsoft Azure [22] currently offers two roles to choose from when creating an instance, web roles and worker roles, both based on Windows Server. The main difference between these two is that an instance of a web role runs IIS, while an instance of a worker role does not. In addition to the type of instances, Azure offers different sizes for both roles [23]. Table B.1 gives an overview of the different standard instances available on Azure.

Both the *administration service* and the *tenant configuration interface* will be running on an Azure web role. While preparing the application for migration, these Azure web roles need to be added to the .NET project, and can be tested in the Azure simulator. When using a third party assembly in the project, this assembly should be added as a reference to the project, with the Copy Local property set to true. A nice side effect of this process is that many deprecated libraries were removed or replaced in the project, making it much easier for developers to locally install the



(a) Initial architecture of the software before moving to the cloud.



(b) Architecture after migration to the cloud and adding support for multi-tenancy.

Figure B.7: Architecture of the application before and after migration to the cloud and adding support for multi-tenancy.

Name	Virtual Cores	Ram
Extra Small (A0)	Shared	768 MB
Small (A1)	1	1.75 GB
Medium (A2)	2	3.5 GB
Large (A3)	4	7 GB
Extra Large (A4)	8	14 GB

Table B.1: Overview of standard instances on Windows Azure.

application, as they no longer needed to configure and install third-party products on a clean environment before being able to compile and test the application.

The SQL databases will be moved to SQL Azure. As a result, the connection strings inside the application should be altered to point to the SQL Azure instance. SQL Azure has some limitations regarding a dedicated Microsoft SQL Server, but for most .NET applications, this shouldn't be an issue. Once the application is running correctly in the Azure simulator, the project can be packaged and deployed onto Windows Azure [24] [25].

B.4.2.3 Determining Impact on Client Network

The traffic between the *administration service* and the *device manager* now has to pass the public Internet, and the internal network is also loaded with traffic between the *device manager* and the different peripherals. The total amount of traffic is depending on the selection of features, as some of the features might require more bandwidth. Both the internal network as the public Internet connection need to have sufficient bandwidth to support the MC system to operate. The service described in [18] was customized to support this PoC, making it possible to predict if a custom selection of features would be able to run on the client network. For this PoC, the different topologies of the client networks were implemented statically, but we introduced the option to easily replace these static topologies by a dynamically generated topology, which could be generated by tools using existing network discovery protocols, such as Neighbor Discovery Protocol (NDP) and Link Layer Discovery Protocol (LLDP).

B.4.2.4 Scaling the Application

Azure allows the administrator to configure multiple instances with automatic load balancing, which will be required as the number of tenants grow. Recently, limited possibilities were added to Azure for automatic scaling, using the *Autoscaling Application Block* [26]. Alternatively, the creation and deletion of extra instances can be done manually (or in code) by the customer. Some third party products also exist, like AzureWatch [27], which will handle the scaling automatically, or the SaaS provider can create a customized system, for example by using advanced load prediction.

B.4.3 Multi-Tenancy

B.4.3.1 Decoupling Databases

In the initial single-tenant architecture, there is a dedicated relational database for every instance. The connection string to this database is hard-coded in the configuration file (Web.config). To support multi-tenancy, we introduced dynamic

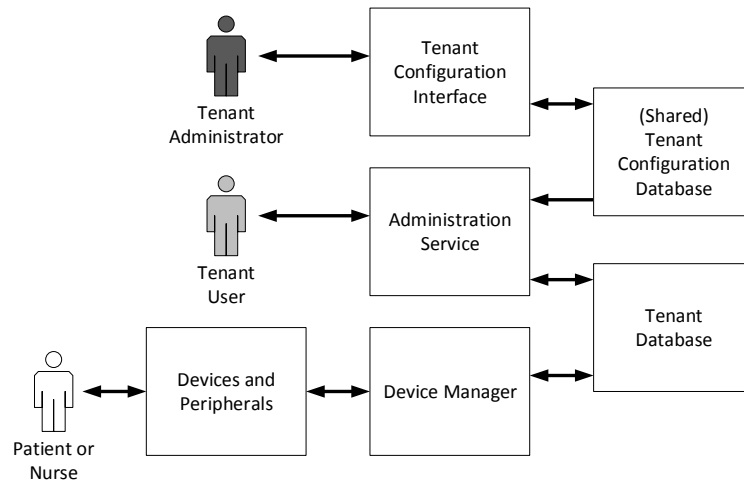


Figure B.8: An overview of the possible communication between actors and the different components of the medical communications system.

connection strings, stored in the *Tenant Configuration Database*. The connection string in the configuration file was replaced by a connection string to this shared database.

To support both shared and dedicated databases, we added an extra column in the data tables, holding the identification of the tenant (*tenantID*). By doing so, the application itself doesn't need to know if the database is shared or dedicated, as multiple tenants can share the same connection string. The *Data Access Component* introduced in Section B.3 is now responsible to select the correct tenant's data, for example by filtering on the corresponding *tenantID*.

B.4.3.2 Adding Tenant Configuration Database

The *Tenant Configuration Database* is introduced to store the general information about the different tenants. It holds the connection strings for each tenant, together with some contact and billing information, and the feature selection for the tenant. As the *administration service* only needs to get this information at start-up, read-only access to this database is sufficient for the main application. This also eliminates the risk of tenants modifying the configuration of other tenants. Figure B.8 illustrates this by giving an overview of the possible communication between actors and components within the system.

B.4.3.3 Providing Tenant Configuration Interface

A new application is introduced, the *Tenant Configuration Interface*, used by tenant administrators (like resellers or the application provider) to setup and configure the different tenants. This application has write access to the *tenant configuration database*, but as only tenant administrators have access to this application, there is no risk of tenants modifying the configuration of other tenants, or even their own configuration, making their system unusable. For this PoC we didn't spend too much time to build a full-blown interface, but in the final version enough time should be spent building this application, as it is a key component in the multi-tenant application which can dramatically minimize the time needed to configure and modify new or existing tenants. The tenant configuration interface was designed as a web application running on an Azure Web Role, but to mitigate security risks, this interface could also be developed as an internal mobile or desktop application, accessing the *tenant configuration database* through web services.

B.4.3.4 Dynamic Feature Selection

The nurse call feature is the core feature of our MC system, but some other features are also implemented, for example voice and video calling between different rooms using Voice over IP (VoIP), and door access control with badges used by the hospital personnel. The selection of features for a single tenant depends on the available hardware and peripherals within the hospital, and the available bandwidth of both the internal and external network. The selection of features and general/technical configuration is done by a tenant administrator through the *tenant configuration interface*, while the tenant-specific configuration of the features can be done by different tenant users through the *administration service*. The initial application (*administration service*) was designed to support dynamic loading of the required libraries and modules at start-up. The modules kept running during the lifetime of the application, but as this application was installed on a dedicated instance with a lot of available resources, this was not really an issue. Converting the application to a multi-tenant application however introduced some new challenges. As every tenant can have its own selection of features, all features might need to be loaded on the single machine, and if the multi-tenant application is not well designed, some features might even be loaded multiple times. To overcome this issue, some changes are needed to the application:

- The required libraries and modules for a specific tenant are loaded as soon as a user logs in to the *administration service*.
- Libraries and modules should be loaded only once, and hence can be shared between different tenants.

- Loaded libraries and modules should be freed as soon as they are not used anymore, for example after a timeout, to eliminate the usage of unnecessary resources.

B.4.3.5 Managing Tenant Data, Users and Roles

As already indicated in Figure B.8, there are a different users and roles used in the MC system:

- The tenant administrators (application provider, resellers, installers), having access to the *tenant configuration interface*. These users and their corresponding roles are stored in the *tenant configuration database*.
- The tenant users and their corresponding roles (mostly personnel of the different hospitals. Because every tenant can have its own users and roles, these are stored in the *tenant database*.
- The patients don't really require roles, but are in way guest users of the system. The peripherals however could count as visualized users with customized roles, and can also be stored in the *tenant database*, together with the tenant users and roles.

B.4.3.6 Mitigating Security Risks

Some of the security risks and a way to eliminate these risks are already described in the previous steps. To increase the security, we added URL-filtering to the application, and altered the access module to take into account the requested URL (and hence the identification of the specific tenant) and the authorized user and its corresponding tenant ID. The traffic between the *device manager* and the *administration service* and *tenant database* now passes the public Internet and is secured by using HTTPS over SSL/TLS. Every tenant can have a dedicated *tenant database*, increasing the isolation of data, but this comes at a higher cost. In practice, big hospitals will typically have a dedicated database, and data from smaller rest-houses belonging to the same entity (subtenants of the same tenant) will be co-located in shared databases. This way, we won't be mixing data from subtenants belonging to different tenants, and isolation of data is always guaranteed at tenant level.

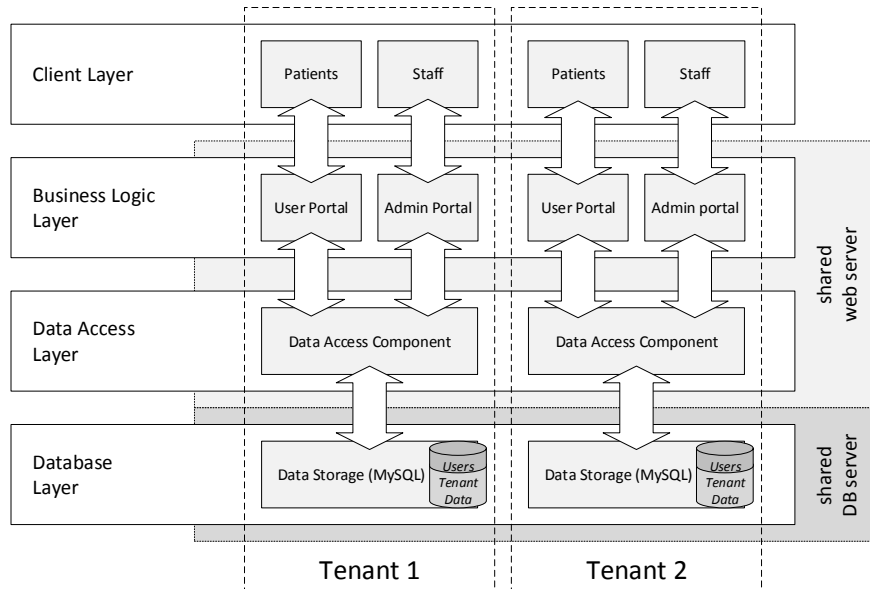


Figure B.9: Pre-migration single-tenant architecture of the medical appointments schedule planner.

B.5 Case Study 2: Medical Appointments Schedule Planner

B.5.1 Introduction

As a second case study, we migrated a medical appointments schedule planner to public cloud environments. This planner is used by both patients and medical staff to manage their appointments. The software was originally developed as a single-tenant application. Figure B.9 illustrates the original layered architecture of the application. The end-users (patients) access the web application through the user portal, in order to manage their medical appointments. The application is running on a shared web server, the appointments and patient data are stored in a dedicated database on a shared database server. Medical staff access the application through the admin portal in order to approve and review the requested appointments.

As multiple clients started using the software, multiple independent copies of the software were installed and configured, running different versions, increasing maintenance complexity. Independent copies were deployed on the same shared web server and the average load increased over time, resulting in an increase in page load times due to the large amount of data and the required amount of data processing by the application. For this case study, we added multi-tenancy to

the application to optimize the utilization of available hardware resources, and migrated the application to the public cloud environment in order to centralize the management and to increase the scalability. We deployed the application on two different cloud providers in order to compare the performance and the ease of deployment.

B.5.2 Cloud Migration

B.5.2.1 Selecting Components

The schedule planner consists of two main components: the user portal, used by patients to request medical appointments, and the admin portal, used by medical staff to approve and review the requested appointments and to manage their schedule. Both patients and medical staff can synchronize their appointments to their personal calendar using one of the available standard calendar formats, and confirmations and reminders are sent by email or by text message (SMS). Different departments are using this portal, but a department may only access patient information relevant for their appointments. Patients on the other hand can browse and request appointments at the different departments.

For this second case study, we selected the whole application for migration to a public cloud provider. This application is less sensitive for short downtime periods as the MC application from the previous case study, as both patients and medical staff have offline copies of their appointments. Therefore, no additional fallback mechanism is necessary inside the application.

The legacy application was developed to be used by a single medical department or an independent doctor (a single tenant), and independent copies of the software were installed and configured. After adding multi-tenancy to the legacy application, a single instance of the application is now shared between multiple tenants, and a new component is introduced, the *tenant configuration interface*, with a similar functionality as the interface from the previous case study.

B.5.2.2 Determining Provider Compatibility

The legacy web application is developed using HTML5, PHP and MySQL for the persistent storage of data, and is executed on a shared web server. For evaluating our approach, we migrated the application to both a PaaS and IaaS environment. We selected Google AppEngine [28] as PaaS provider as they provide a PHP Runtime Environment, and Amazon EC2 [29] as IaaS provider. Google AppEngine requires more changes to the legacy application as it puts more constraints on applications, while Amazon EC2 requires more maintenance as they provide full control over the virtual machine.

Migrating an application to Amazon EC2 is straightforward. Amazon currently offers two types of EC2 instances, the T2 instances which are burstable performance

Model	Virtual Cores	Ram
t2.micro	1	1 GB
t2.small	1	2 GB
t2.medium	2	4 GB

Table B.2: Overview of the available T2 instance types on Amazon EC2.

Model	Virtual Cores	Ram
m3.medium	1	3.75 GB
m3.large	2	7.5 GB
m3.xlarge	4	15 GB
m3.2xlarge	8	30 GB

Table B.3: Overview of the available M3 instance types on Amazon EC2.

instances for development environments and early product experiments, and the M3 instances, which provide a good balance of compute, memory, and network resources. The different types of T2 and M3 instances currently available are listed in Tables B.2 and B.3 respectively. For our PoC, we selected a t2.micro instance running Ubuntu Server 14.04 for both the database and web server. We configured Apache and MySQL on the instance and deployed the application, and except from some configuration settings, no changes were required in the application.

Migrating the legacy PHP application to Google AppEngine on the other hand required multiple changes to the application as summarized in Table B.4. First of all, as Google offers Cloud SQL instead of MySQL, some changes are required in the application to connect to the Cloud SQL database instance [30]. The original MySQL database can be exported to a file using a SQL dump, and this file can be used to import the data into a new Cloud SQL database instance. The mysqli extension introduced with PHP version 5.0.0 can still be used to connect to the database, but the connection string differs from a traditional connection string as illustrated in [30].

In AppEngine, the local file system that the application is deployed to is not writable. However, if the application needs to write and read files at runtime, AppEngine provides a built-in Google Cloud Storage (GCS) stream wrapper that allows many of the standard PHP file system functions. A PHP application running on AppEngine can read and write files by using buckets as illustrated in [31]. The legacy application was developed using the Smarty PHP Template engine [32], which requires different physical file directories for reading and writing templates and configuration files. As a result, the Smarty engine needs to be reconfigured to

Item	Description
Relational Data	Migrate MySQL databases to Google Cloud SQL and modify connection strings
Temporary Files	Replace local file storage by storage buckets on Google Cloud Storage
URL Rewriting	Replace mod_rewrite by a custom PHP script providing similar functionality

Table B.4: Overview of the most important changes for migration to Google AppEngine.

use the GCS for storing the compiled templates and files. One major difference between writing to a local disk and writing to GCS is that GCS does not support modifying or appending to a file after closing it. Instead, a new file can be created with the same name, which overwrites the original. For the Smarty PHP Template engine however this is not really an issue as it only creates temporary files which are not modified after creation. Using buckets to store temporary files can have an influence on the performance of the application. There is no straightforward way to measure this impact, as local file storage is not supported by Google AppEngine. In Section B.6 we however do compare the performance of the application running on Google AppEngine with other environments which are using traditional file storage.

Finally, the legacy application implemented URL rewriting by invoking the Apache mod_rewrite module. As Google AppEngine does not support this module, this functionality has to be simulated through the use of a PHP script referenced from the application's configuration file (app.yaml) that will in turn load the desired script, as described in [33]. The overhead introduced by this script is minimal, as it just parses the requested URI and executes a simple conditional statement. Google however recommends to rewrite the application to operate without the mod_rewrite module, but this requires more effort as more changes to the source code are required.

Once the application is running correctly in the simulated environment of Google App Engine Launcher (part of the Google App Engine SDK), it can be deployed onto the public cloud.

B.5.2.3 Determining Impact on Client Network

As the original application was already designed to be accessed over the web, and the full application is migrated to the cloud, there is no real impact on the client network after migration to the public cloud.

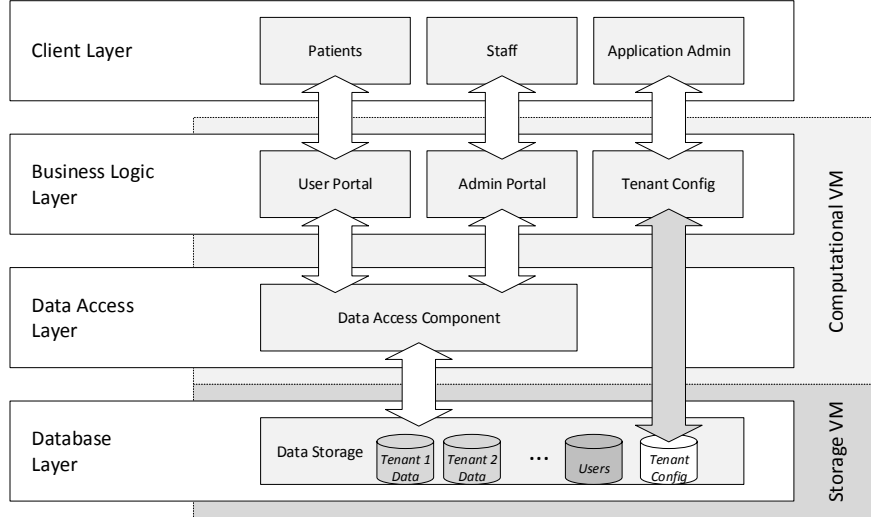


Figure B.10: Revised architecture of the schedule planner after adding multi-tenancy to the application and migration to the public cloud.

B.5.2.4 Scaling the Application

Amazon offers CloudWatch [34] to monitor AWS cloud resources and applications. This service provides a clear insight in the current demand using different metrics such as CPU utilization, data transfers and disk usage activity. Application developers can also create custom metrics, and customize automated actions and alarms. For a production-ready application on Amazon EC2, CloudWatch can be used to provide compliance with specific SLA targets, and to handle the automated scaling of both the computational resources.

Google AppEngine on the other hand has built-in support for high scalability. An application running on AppEngine can be deployed on multiple instances and instances are automatically created or removed depending on the current load. No action is required from the developer, but the developer has more limited control than with Amazon EC2. During our experiments as described in Section B.6, multiple instances were automatically created.

B.5.3 Multi-Tenancy

After adding multi-tenancy to the application, the original architecture was slightly modified. Figure B.10 illustrates the modified architecture after adding multi-tenancy and migration to a public cloud provider. These modifications are discussed in detail in the remainder of this section.

B.5.3.1 Decoupling Databases

In the initial single-tenant architecture, every tenant has a dedicated MySQL database on a shared database server. In order to support multi-tenancy, we introduced dynamic connection strings, as in the previous case study. All users are however stored in a single database, separated from the tenant databases. By doing so, a single user can access multiple tenants, and multiple copies of the same user object are eliminated.

As with the previous case study, we added an extra column to the data tables, holding the identification of the tenant (*tenantID*). By doing so, the application supports both shared and dedicated database instances, and multiple tenants can share a single database. The data access component of the data access layer was modified to support the dynamic behavior of the tenant databases, and to filter tenant data based on the *tenantID*. This filtering is required in order to provide transparent isolation of tenant data, especially when multiple tenants are sharing a single database instance.

B.5.3.2 Adding Tenant Configuration Database

The shared *Tenant Configuration Database* contains general information about the different tenants, and a connection string to the database instance where the tenant data is stored. This database is small in size, which is why it is also used to store the different user objects. However, should this database ever become a bottleneck, the user data can easily be decoupled from the general tenant information, as separate connection strings are used for the user database and the tenant configuration database. For our PoC these connection strings refer to the same tenant configuration database instance.

B.5.3.3 Providing Tenant Configuration Interface

A tenant configuration interface was added to the application, used to manage the different tenants. As in the previous case study, this configuration interface communicates directly with the tenant configuration database.

B.5.3.4 Dynamic Feature Selection

Tenants can have optional features enabled, for example notifications by text messages or export options to different calendar formats. A tenant administrator can configure these features through the tenant configuration interface. The feature configuration is stored in the tenant configuration database together with the general tenant information, and both the application's user portal and admin portal take the feature selection of the selected tenant into account.

B.5.3.5 Managing Tenant Data, Users and Roles

The user objects are stored in the shared tenant configuration database, the roles are stored together with the tenant data in a tenant database instance. This instance can either be a dedicated database instance or an instance that is shared between multiple tenants. Relevant medical information belonging to a certain patient is stored together with the role in the tenant database. By doing so, sensitive patient data is inaccessible by other tenants, as all data queries are filtered based on the *tenantID* by the data access component.

By using a single database instance to store all user objects, multiple roles for different tenants can be created for a single user object. This allows for a single sign-on, where the user object is loaded when the user logs in on the application, and the relevant roles are loaded when the user wants to access one of the tenant's restricted pages.

B.5.3.6 Mitigating Security Risks

As mentioned above, sensitive data belonging to a certain patient is stored together with the user role in the tenant database. As all data queries are filtered by the data access object based on the *tenantID*, queries can never return data belonging to different tenants.

Communication between a client computer and the web server is encrypted, as all communication uses HTTPS over SSL. This was already the case with the legacy application.

B.6 Discussion and Evaluation

Moving applications to the cloud and adding multi-tenancy introduces new opportunities for our presented use cases. First of all, there is the increased flexibility and elasticity. When the workload on the application increases, new instances can be created and deployed automatically. Similarly, when the workload decreases, instances can again be removed. For new customers, deployment times decrease as there is no need to physically install a new server. By using a PaaS platform instead of IaaS, there is no need to install, configure and manage the guest OS, further reducing the deployment times. The hardware maintenance cost is also eliminated as the virtual machines running in the cloud are automatically migrated when the hardware fails. Combining multi-tenancy and migration to a public cloud makes maintenance easier, as the software is deployed centrally, and no on-site intervention is needed, for example to install patches or updates. Adding multi-tenancy to the application also improves the efficiency of resource utilization, decreasing the costs, and eliminates the need for installing and configuring independent copies of the same software, sometimes running different versions of the software. In

this section, we will highlight some of the major advantages of the migration, and compare them with the overhead of the migration.

B.6.1 Increased flexibility and elasticity

As the amount of available resources in the cloud is quasi unlimited, application developers don't have to worry about selecting the right amount of resources to host the software. Novel multi-tenant applications can start with a single instance, and the number of instances can grow as the workload increases. In cloud computing, elasticity is defined as the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible. In our approach, we have presented some possibilities for building elastic applications in the step of *Scaling the Application*. For the two case studies, we also started with a single instance, and determined the possibilities to scale the application as the demand grows.

B.6.2 Decreased deployment time

The addition of multi-tenancy to the application and migration to a public cloud yields a significant decrease in deployment times, especially for new tenants. Tables B.5 and B.6 illustrate this for the MC software case study by giving an estimation of the needed deployment time for a new tenant, respectively before and after the migration process.

Before migration, a physical server was installed and configured on-site for every new tenant, together with the device manager and peripherals. A local copy of the *administration service* was installed and configured on the dedicated physical server together with a SQL Server instance. A total of 6 man-days was required to perform the installation and configuration of both the server and the *device manager* and peripherals.

Task	Time
Install and configure on-site server for application	1 day
Deploy administration service on on-site server	1 day
Configure SQL server and initial tenant database	1 day
Configure on-site hardware (device manager and peripherals)	2 days
Verification and testing	1 day

Table B.5: Initial configuration of new tenant before migration: time estimation for the MC software case study.

Task	Time
Create new tenant and initial tenant database	1 hour
Impact analysis on client network (automated)	1 hour
Configure on-site hardware (device manager and peripherals)	2 days
Verification and testing	1 day

Table B.6: Initial configuration of new tenant after migration: time estimation for the MC software case study.

Task	Time
Deploy administration service on Azure	2 hours
Deploy tenant configuration interface on Azure	2 hours
Create initial databases on SQL Azure	2 hours
Create tenant administrators	2 hours

Table B.7: Initial deployment after migration: time estimation for the MC software case study.

After migration, the initial configuration time is largely reduced. Only the *device manager* and peripherals need to be installed, and the configuration of the *device manager* can be done remotely by using the multi-tenant *administration service* running on the cloud. As only a new tenant needs to be created, no local copy of the *administration service* needs to be installed and configured. An estimated total of 3.5 man-days are required for the initial setup after migration, mainly for the installation and configuration of the on-site *device manager* and the peripherals, and for full testing.

Migrating the *device manager* to the cloud could further reduce the deployment time, but this however introduces additional challenges which were mentioned before in Section B.4.2.1. For completeness, Table B.7 shows an estimation of the initial deployment of the application on Microsoft Azure. This initial deployment needs to be done only once, and not for every new tenant.

B.6.3 Ease of Maintenance

Having the core of the MC software, the *administration service*, hosted in the public cloud makes maintenance a lot easier, as installers no longer need to go on-site to make small configuration changes. Eventually, one could argue that having VPN connections to the customer sites could also bypass this, but this requires a VPN setup to the hospitals, or public access to the internal network, which again introduces some security risks, together with a stable external connection at the

client side. Having a single multi-tenant application also has the advantage that every tenant uses the same version of the software, and software updates can be deployed centrally, for all tenants at once. For installing software updates, a second instance can be deployed and configured in an isolated environment on the public cloud, and switched with the current instance once the configuration and testing is done. Software updates and patches for the *device managers* can be pushed from the central *administration service*, as under normal circumstances, the *device manager* has a persistent connection to the *administration service* and will frequently check for updates.

For our second case study, the schedule planner, deploying the multi-tenant application on the public cloud results in similar benefits. Before adding multi-tenancy to the application, different independent versions were deployed, and updating and maintaining the application became harder as the number of tenants grew. After migration to the cloud, only one copy of the multi-tenant application is running on multiple instances in the cloud, and all tenants are running the latest software of the application.

B.6.4 Migration Cost

Migrating software to the cloud and adding multi-tenancy to the application comes at a cost. The architecture and code might need to be changed, and the software needs to be tested thoroughly. Extra attention needs to be paid to the security aspect, as the whole application or some components are now hosted remotely. For the MC software, we spent a total of six man-months to implement the changes described for in Section B.4. For a production ready application in the cloud, an estimated additional 14 man-months would be required, as summarized in Table B.8. The remaining tasks mainly focus on adapting the cloud application to support the existing SLAs, investigating possibilities for automatic backup and restore, providing training for installers and full testing.

For the schedule planner, our second case study, only two man-months were required to implement the changes described in Section B.5, as this application is less complex than the MC software. Adding multi-tenancy to the legacy application required a bit less than two man-months. For the migration to Amazon EC2 only 1 day was sufficient, whereas for Google AppEngine a few man-days were necessary to implement the required changes. For a production ready application in the cloud, an estimated additional 4 man-months would be required, for finishing the application and full testing.

B.6.5 Remaining Risks

As some components are now hosted on a public cloud, there is an increased security risk. For our first case study, the MC software, extra attention should be

Task	Time
Azure-specific tasks	3 man-months
Explore monitoring options	
Define backup and restore strategies	
Verify SLA constraints on Azure	
Adapt cost model	
Administration Service	8 man-months
Improve security of service and features	
Make complete mapping for all features	
Migrate remaining features to Azure	
Add support for newest hardware nodes	
Study impact of shared vs. dedicated database instances	
Other remaining issues	
Impact Analyzer	1 man-month
Support dynamic generated topologies	
Other tasks	2 man-months
Training and development courses for developers	
Training for installers, retailers, clients	
Full testing of application	

Table B.8: Tasks to be done for a production ready application in the cloud: time estimation for the MC software case study..

paid to the new risks introduced by moving the software to the public cloud. A side-effect of the migration is that the *Administration Service* is now hosted in the public cloud, and could become a bottleneck if the number of tenants grows significantly. Therefore, extra attention should also be paid to the scalability of this service. For our second case study, the main security risk is due to the addition of multi-tenancy, so the application need to guarantee isolation of sensitive data, for example by restricting all queries to filter data based on the *tenantID*.

B.6.6 Change in Cost Model

Migrating the software to the cloud brings a change in the cost-model of the MC software. Before the migration, the software and hardware was typically sold as a single package, including the necessary hardware, licenses for the software, initial installation and configuration. As most public providers work with monthly fees, the application provider should adapt the cost-model to reflect this cost model. Instead of selling a one-time license for the software, the end users can pay a monthly fee, depending on the size of the tenant, covering the hosting on Azure and future software changes and updates. The *Tenant Configuration Interface* can be

Label	Description	Estimated cost
local	A dedicated virtual machine with 1GB Ram and 1vCPU, running on a physical Linux server with an Intel Core i7 CPU (2.80 GHz) with 8 GiB of memory.	± 20.00 USD/month + one-time infrastructure cost
shared	The shared web server on which the legacy application was running before migration to the public cloud.	1.82 USD/month
EC2	Amazon EC2 t2.micro instance	14.28 USD/month
AppEngine	Instance running on Google AppEngine	depends on usage

Table B.9: An overview of the different deployment environments. The mentioned cost values are valid at the time this appendix was submitted as an article.

designed to support this cost-model, and could be linked to the financial software. This change in cost model introduces a new opportunity by expanding the customer market, as smaller clients (for example small rest-houses) are now able to start using the software at a lower cost, as the costs of implementing the system can now easier be spread over time, less hardware is required on-premise, and computational resources are shared between multiple tenants.

The schedule planner software was already being sold using a license-based cost model. Adding multi-tenancy and migrating the application to the cloud introduces no visible changes in cost model. Sales prices could however drop as utilization of available resources is optimized by adding multi-tenancy to the application, and the infrastructure cost is reduced by using a public cloud provider.

B.6.7 Performance Comparison

The performance of an application running on the cloud depends on both the selected cloud provider and the selected instance type. We selected the second case study, the medical appointments schedule planner, for evaluating the performance of the selected cloud providers, as this application is fully migrated to the public cloud, whereas the MC application is only partially migrated, making the performance depend on more factors such as the on-site client network topology and capacity and on-site available hardware.

In order to evaluate the performance of the medical appointments schedule planner, we deployed the application to four different environments, as summarized in Table B.9. We measured the average page generation time, this is the time needed for the PHP interpreter to generate the page, for different pages of the application. This metric does not take into account the network latency, as the generation time is measured at the server side by the PHP interpreter itself. We also measured the end-to-end transaction time, this is the total load time as perceived by the client, as this metric does include the network latency. The schedule planner application

	local		shared		EC2		AppEngine	
	\bar{y}	σ	\bar{y}	σ	\bar{y}	σ	\bar{y}	σ
Page 1	0.19588	0.00596	3.20709	0.81836	0.19557	0.00437	6.28543	0.84406
Page 2	0.20555	0.00688	3.00282	0.23948	0.19974	0.00372	6.38447	0.91441
Page 3	0.02882	0.00216	0.33663	0.03426	0.02183	0.00039	1.38429	0.56244

Table B.10: Average page generation times (in seconds) and standard deviations for 3 test pages over 50 iterations.

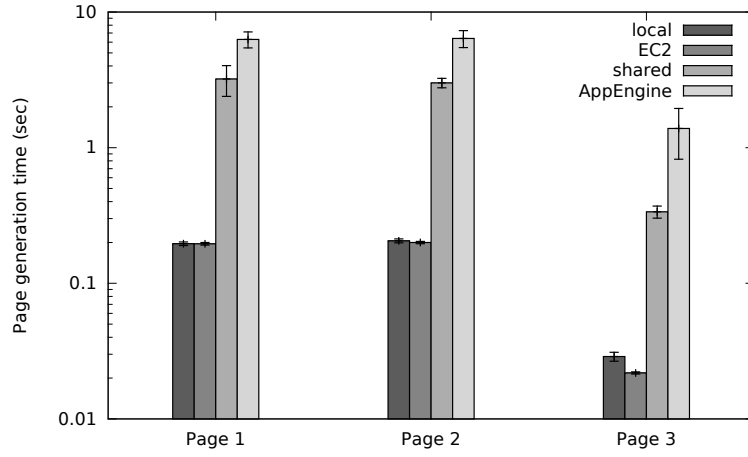


Figure B.11: A comparison of the average page generation times for 3 test pages over 50 iterations.

	local		shared		EC2		AppEngine	
	\bar{y}	σ	\bar{y}	σ	\bar{y}	σ	\bar{y}	σ
Page 1	0.39160	0.01721	3.39200	0.74018	0.54840	0.10075	6.68200	0.79333
Page 2	0.37520	0.00934	3.11800	0.23113	0.47580	0.08840	7.03000	0.83211
Page 3	0.22260	0.00439	0.43580	0.03662	0.40920	0.03746	1.57200	0.56211

Table B.11: Average end-to-end transaction times (in seconds) and standard deviations for 3 test pages over 50 iterations.

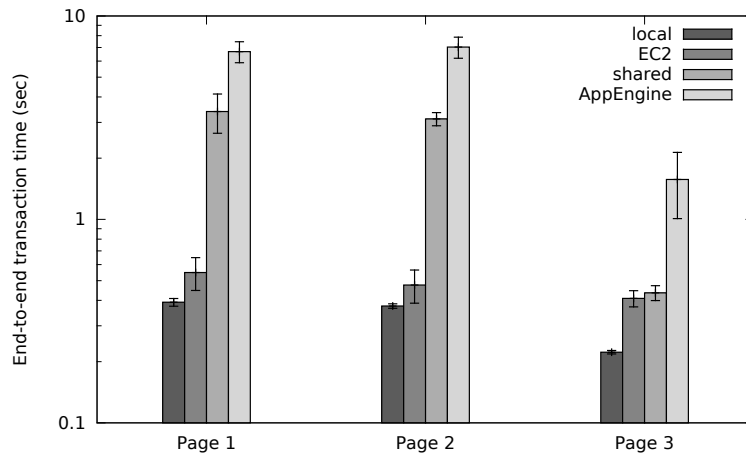


Figure B.12: A comparison of the average end-to-end transaction times for 3 test pages over 50 iterations.

was configured with a custom database based on data from existing production databases, combining historical data from different tenants over the last 3 years. We selected three specific pages for this experiment. The first two pages perform complex merge operations on the tenant data, as these pages were reported by existing users as being too slow. The third page is a normal page with an average load time. The experiments were executed on the cloud platforms in January 2015. Table B.10 provides the measured average page generation times together with the standard deviations for the selected test pages over 50 iterations, and Figure B.11 illustrates the same results graphically. Table B.11 and Figure B.12 are similar, but for the end-to end transaction times.

As can be seen from these results the Amazon EC2 Engine provides a good performance, as it is even faster than the local VM, even though we used the lightest instance available, a t2.micro instance. The Google AppEngine on the other hand is rather slow, as it takes up to 7 seconds to generate one of the selected heavy pages. A possible explanation for this is that PHP support by Google AppEngine is still experimental, and the engine is not yet optimized for PHP. We however would like to note that the performance of Google AppEngine has already improved considerably over the last months, as in September 2014 the similar experiments were executed, and the same page could then take up to 45 seconds to generate. For a production ready application in the cloud, Amazon EC2 will however be selected to host the application, as it currently is a clear winner in the executed experiments.

B.7 Conclusions

Cloud computing and multi-tenancy allow providers to improve the scalability of applications while reducing hosting costs. In this appendix, we presented a generic approach for migrating legacy software to the public cloud, and adding multi-tenancy to the application. We briefly described the different steps needed to convert the dedicated application to a cloud application, and the steps required to add multi-tenancy to the application. We verified our approach using two case studies from medical software: medical communications software and a medical appointments schedule planner. For the MC software, we migrated some components to a public cloud provider, creating a hybrid cloud, whereas for the schedule planner, we did a full migration of the legacy software to two different public cloud provider.

Migrating an application to the public cloud only requires a limited number of changes, while the conversion from a single-tenant to a multi-tenant application requires more steps as the latter requires limited changes to the application architecture. These modifications are however necessary to fully benefit from the opportunities of public cloud computing. We presented a proactive approach by identifying and eliminating possible future risks, for example by mitigating security risks and analyzing the architecture regarding its scalability.

In our evaluation, we described the advantages of both the cloud migration and the addition of multi-tenancy, and took into account the costs of the migration and remaining risks. After migrating the MC software, the time needed for the initial creation of a new tenant is strongly reduced (from 6 to an estimated 3.5 man-days, including the initial setup of the dedicated hardware), and maintenance has become much easier after migration. The reduction in initial costs and management costs also enables supporting smaller clients for which the costs used to be prohibitive. Supporting these additional clients may present new business opportunities in the long-term.

For scenarios where the performance is critical, different public cloud providers should be considered and evaluated, and within a single provider different instance types might exist. For our second use case, Amazon EC2 has a clear advantage over Google AppEngine for running the schedule planner, and yielded even better results than a dedicated virtual machine on a physical Linux server. The advantages of using a PaaS provider should also be weighted against the increased control gained when using an IaaS provider.

Migrating legacy software to the cloud comes at a cost, and some application components may need to be modified or rewritten. However, by following the multi-step migration approach presented in this appendix, the benefits of a cloud migration could outweigh the costs of implementing the described changes, as can be seen in the evaluation section of this appendix.

References

- [1] M. Armbrust, A. Fox, G. Rean, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. *Above the Clouds : A Berkeley View of Cloud Computing*. Technical report, University of California at Berkley, 2009.
- [2] D. Schatzberg, J. Appavoo, O. Krieger, and E. Van Hensbergen. *Scalable Elastic Systems Architecture*. In Proceedings of the ASPLOS Runtime Environment/Systems, Layering, and Virtualized Environments (RESOLVE) Workshop. ASPLOS, mar 2011.
- [3] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao. *A Framework for Native Multi-Tenancy Application Development and Management*. In Proceedings of the 9th IEEE International Conference on E-Commerce and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, (CEC/EEE 2007), pages 551 – 558. IEEE, jul 2007. doi:10.1109/CEC-EEE.2007.4.
- [4] P.-J. Maenhaut, H. Moens, M. Verheye, P. Verhoeve, S. Walraven, W. Joosen, and V. Ongenaes. *Migrating Medical Communications Software to a Multi-Tenant Cloud Environment*. In Proceedings of the 13th IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), pages 900–903. IEEE, may 2013.
- [5] M. Hajjat, X. Sun, Y.-W. E. Sung, D. Maltz, S. Rao, K. Sripanidkulchai, and M. Tawarmalani. *Cloudward Bound: Planning for Beneficial Migration of Enterprise Applications to the Cloud*. SIGCOMM Computer Communication Review, 40(4):243–254, aug 2010. doi:10.1145/1851275.1851212.
- [6] A. Li, X. Yang, S. Kandula, and M. Zhang. *Comparing Public-Cloud Providers*. IEEE Internet Computing, 15(2):50–53, mar 2011. doi:10.1109/MIC.2011.36.
- [7] A. Khajeh-Hosseini, D. Greenwood, and I. Sommerville. *Cloud Migration: A Case Study of Migrating an Enterprise IT System to IaaS*. In Proceedings of the 3rd IEEE International Conference on Cloud Computing, pages 450–457. IEEE, jul 2010. doi:10.1109/CLOUD.2010.37.
- [8] Q. H. Vu and R. Asal. *Legacy Application Migration to the Cloud: Practicability and Methodology*. In Proceedings of the 8th World Congress on Services (SERVICES 2012), pages 270–277. IEEE, jun 2012. doi:10.1109/SERVICES.2012.47.

- [9] S. Walraven, E. Truyen, and W. Joosen. *Comparing PaaS offerings in light of SaaS development*. Computing, 96(8):669–724, oct 2013. doi:10.1007/s00607-013-0346-9.
- [10] P. J. P. da Costa and A. M. R. da Cruz. *Migration to Windows Azure - analysis and comparison*. Procedia Technology, 5(0):93–102, 2012. 4th Conference of ENTERprise Information Systems – aligning technology, organizations and people (CENTERIS 2012). doi:10.1016/j.protcy.2012.09.011.
- [11] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. D. Turck. *Feature Placement Algorithms for High-Variability Applications in Cloud Environments*. In Proceedings of the 13th Network Operations and Management Symposium (NOMS 2012), pages 17–24. IEEE, apr 2012. doi:10.1109/NOMS.2012.6211878.
- [12] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. *Developing and Managing Customizable Software as a Service Using Feature Model Conversion*. In Proceedings of the 3rd IEEE/IFIP Workshop on Cloud Management (CloudMan 2012), pages 1295–1302. IEEE, apr 2012. doi:10.1109/NOMS.2012.6212066.
- [13] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. *Cost-Effective Feature Placement of Customizable Multi-Tenant Applications in the Cloud*. Journal of Network and Systems Management, 22(4):517–558, oct 2014. doi:10.1007/s10922-013-9265-5.
- [14] H. Moens and F. De Turck. *Feature-Based Application Development and Management of Multi-Tenant Applications in Clouds*. In Proceedings of the 18th International Software Product Line Conference (SPLC 2014), pages 72–81. ACM, sep 2014. doi:10.1145/2648511.2648519.
- [15] S. Walraven, E. Truyen, and W. Joosen. *A Middleware Layer for Flexible and Cost-Efficient Multi-tenant Applications*. In F. Kon and A.-M. Kermarrec, editors, Middleware 2011, volume 7049 of *Lecture Notes in Computer Science*, pages 370–389. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-25821-3_19.
- [16] P.-J. Maenhaut, H. Moens, M. Decat, J. Bogaerts, B. Lagaisse, W. Joosen, V. Ongenae, and F. De Turck. *Characterizing the Performance of Tenant Data Management in Multi-Tenant Cloud Authorization Systems*. In Proceedings of the 14th Network Operations and Management Symposium (NOMS 2014), pages 1–8, Krakow, Poland, may 2014. IEEE. doi:10.1109/NOMS.2014.6838232.

- [17] P.-J. Maenhaut, H. Moens, V. Ongenae, and F. De Turck. *Scalable User Data Management in Multi-Tenant Cloud Environments*. In Proceedings of the 10th International Conference on Network and Service Management (CNSM 2014), pages 268–271, Rio de Janeiro, Brazil, nov 2014. IEEE. doi:10.1109/CNSM.2014.7014171.
- [18] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck. *Network-Aware Impact Determination Algorithms for Service Workflow Deployment in Hybrid Clouds*. In Proceedings of the 8th International Conference on Network and Service Management (CNSM 2012), pages 28–36. IEEE, oct 2012.
- [19] F. Ongenae, P. Duysburgh, M. Verstraete, N. Sulmon, L. Bleumers, A. Jacobs, A. Ackaert, S. De Zutter, S. Verstichel, and F. De Turck. *User-driven design of a context-aware application: an ambient-intelligent nurse call system*. In Proceedings of the 6th International Conference on Pervasive Computing Technologies for Healthcare (PervasiveHealth), pages 205–210. IEEE, may 2012. doi:10.4108/icst.pervasivehealth.2012.248699.
- [20] F. Ongenae, M. Claeys, T. Dupont, W. Kerckhove, P. Verhoeve, T. Dhaene, and F. De Turck. *A probabilistic ontology-based platform for self-learning context-aware healthcare applications*. Expert Systems with Applications, 40(18):7629–7646, dec 2013. doi:10.1016/j.eswa.2013.07.038.
- [21] F. Ongenae, A. Hristoskova, E. Tsiporkova, T. Tourwé, and F. De Turck. *Semantic reasoning for intelligent emergency response applications*. In Proceedings of the 10th International Conference on Information Systems for Crisis Response and Management, Abstracts, pages 1–2, may 2013.
- [22] Microsoft. *Introducing Windows Azure* [online]. 2013. Last accessed: February 2015. Available from: <http://www.windowsazure.com/en-us/develop/net/fundamentals/intro-to-windows-azure>.
- [23] Microsoft. *Azure Pricing* [online]. 2013. Available from: <http://www.windowsazure.com/en-us/pricing/details/cloud-services/>.
- [24] D. Betts, S. Densmore, M. Narumoto, E. Pace, and M. Woloski. *Moving Applications to the Cloud on Microsoft Windows Azure*. Microsoft ; O'Reilly distributor, 2010.
- [25] Microsoft. *How to Create and Deploy a Cloud Service* [online]. Last accessed: February 2015. Available from: <http://www.windowsazure.com/en-us/manage/services/cloud-services/how-to-create-and-deploy-a-cloud-service>.

- [26] Microsoft. *How to Use the Autoscaling Application Block* [online]. Last accessed: February 2015. Available from: <http://www.windowsazure.com/en-us/develop/net/how-to-guides/autoscaling/>.
- [27] Microsoft. *Monitoring and Autoscaling features for Windows Azure with AzureWatch indepth - AzureWatch* [online]. Last accessed: February 2015. Available from: <http://www.paraleap.com/azurewatch>.
- [28] Google. *Google App Engine - Free Platform-as-a-Service (PaaS)* [online]. Last accessed: February 2015. Available from: <https://cloud.google.com/appengine/>.
- [29] Amazon. *AWS - Amazon Elastic Compute Cloud (EC2) - Scalable Cloud Hosting* [online]. Last accessed: February 2015. Available from: <http://aws.amazon.com/ec2/>.
- [30] Google. *Using Google Cloud SQL - PHP* [online]. Last accessed: February 2015. Available from: <https://cloud.google.com/appengine/docs/php/cloud-sql/>.
- [31] Google. *What is Google Cloud Storage? - Google Cloud Storage* [online]. Last accessed: February 2015. Available from: <https://cloud.google.com/storage/docs/overview>.
- [32] *PHP Template Engine - Smarty* [online]. Last accessed: February 2015. Available from: <http://www.smarty.net/>.
- [33] Google. *Simulate Apache mod_rewrite routing - PHP* [online]. Last accessed: February 2015. Available from: https://cloud.google.com/appengine/docs/php/config/mod_rewrite.
- [34] Amazon. *Amazon CloudWatch - Cloud & Network Monitoring Services* [online]. Last accessed: February 2015. Available from: <http://aws.amazon.com/cloudwatch>.

