Foundations of efficient virtual appliance based service deployments.

Gabor Kecskemeti

School of Electronics and Computer Science

FOUNDATIONS OF EFFICIENT VIRTUAL APPLIANCE BASED SERVICE DEPLOYMENTS

GABOR KECSKEMETI

A thesis submitted in partial fulfilment of the requirements of the University of Westminster for the degree of Doctor of Philosophy

December 2011

ABSTRACT

The use of virtual appliances could provide a flexible solution to services deployment. However, these solutions suffer from several disadvantages: (*i*) the slow deployment time of services in virtual machines, and (*ii*) virtual appliances crafted by developers tend to be inefficient for deployment purposes. Researchers target problem (*i*) by advancing virtualization technologies or by introducing virtual appliance caches on the virtual machine monitor hosts. Others aim at problem (*ii*) by providing solutions for virtual appliance construction, however these solutions require deep knowledge about the service dependencies and its deployment process.

This dissertation aids problem (*i*) with a virtual appliance distribution technique that first identifies appliance parts and their internal dependencies. Then based on service demand it efficiently distributes the identified parts to virtual appliance repositories. Problem (*ii*) is targeted with the Automated Virtual appliance creation Service (AVS) that can extract and publish an already deployed service by the developer. This recently acquired virtual appliance is optimized for service deployment time with the proposed virtual appliance optimization facility that utilizes active fault injection to remove the non-functional parts of the appliance. Finally, the investigation of appliance distribution and optimization techniques resulted the definition of the minimal manageable virtual appliance that is capable of updating and configuring its executor virtual machine.

The deployment time reduction capabilities of the proposed techniques were measured with several services provided in virtual appliances on three cloud infrastructures. The appliance creation capabilities of the AVS are compared to the already available virtual appliances offered by the various online appliance repositories. The results reveal that the introduced techniques significantly decrease the deployment time of virtual appliance based deployment systems. As a result these techniques alleviated one of the major obstacles before virtual appliance based deployment systems.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank Dr. Gabor Terstyanszky, my director of studies, for his support throughout the past seven years I have known him. During this period first, he supervised me as a foreign exchange student at the University of Westminster. Then, he approached me with an offer that launched my scientific career with a job and a PhD position at the university. This dissertation would never exist without his extremely detailed comments on my research papers and this dissertation.

Next, I would like to thank Prof. Peter Kacsuk, my third supervisor, for offering me a job in the MTA-SZTAKI, Hungary. Even though I left the UK, this job allowed me to still work on my PhD studies and frequently visit the University of Westminster in the early stages of my research.

Afterwards, I would like to express my gratitude to Prof. Stephen Winter (my second supervisor) and Dr. Zsolt Nemeth (a colleague in MTA-SZTAKI). They both largely contributed to my scientific writing skills. Without the support of Zsolt Nemeth, this dissertation would have taken twice the time to write.

Then, I appreciate the support of Dr. Laszlo Kovacs from the University of Miskolc, Hungary. He introduced me to the scientific methodologies and gave me three years leave from my teaching job at the university to fast track my PhD studies.

I would like to acknowledge the countless hours the infrastructure team of the Centre of Parallel Computing has spent on preparing the various testbed requirements I have asked for. I would like to emphasize the assistance of Zsolt Lichtenberger who has spent months of his precious time to prepare and support my Eucalyptus testbed.

Finally, I would like to thank my wife, my mother and my grandmother their encouragement throughout my research and writing up periods that lead to this thesis.

London, December 2011
Gabor Kecskemeti

CONTENTS

ABSTRACT ii
ACKNOWLEDGEMENTS iii
Contents iv
List of Figures vii
List of Tables x
Notation Index xi
Glossary xviii

I RESEARCH OVERVIEW 1
1 Introduction 2
  1.1 Service deployment overview 4
    1.1.1 The appliance model 5
  1.2 Requirements for Automating Service Deployment 8
    1.2.1 Taxonomy of Related Deployment Systems 9
  1.3 Contributions 11
    1.3.1 Approach for initial Virtual Appliance creation (C1) 12
    1.3.2 Parallel Algorithm for Virtual Appliance Size Optimization (C2) 12
    1.3.3 Distributed virtual appliance storage and delivery (C3) 13
    1.3.4 Minimal Manageable Virtual Appliance (C4) 13
  1.4 The structure of the thesis 13
2 Related Works 15
  2.1 Infrastructure as a service clouds 15
  2.2 Service Deployment overview 17
  2.3 Virtual appliance size optimization 21
  2.4 Virtual appliance distribution optimization 23

II ACHIEVEMENTS 25
3 Overview of the Architecture 26
  3.1 Introduction 26
  3.2 Virtual Appliance Management 28
  3.3 Basic system definitions 29
    3.3.1 Package types and relations 31
4 Automatic Virtual Appliance Creation Service 39

4.1 Introduction 39
4.2 The AVS Client Interface 41
4.2.1 Virtual Appliance extraction 41
4.2.2 Playground operations 43
4.2.3 The upload operation 45
4.3 The Infrastructure as a Service System Interface 48
4.4 Repository Interface and Metadata Collection 50
4.4.1 Virtual Appliance Representation 51
5 Virtual Appliance Size Optimization Facility 55
5.1 Virtual Appliance Optimization Principles 55
5.1.1 The Virtual Appliance Optimization Facility 56
5.1.2 Appliance Contents Removal 58
5.2 Implementation of Virtual Appliance Optimization 62
5.2.1 Implementation of the Item Selection 62
5.2.2 Parallel Validation 68
5.2.3 Virtual Machine Management Strategy 71
6 Partial Virtual Appliance Replication 75
6.1 Introduction 75
6.2 Active Repository Functionality 76
6.2.1 Package decomposition 77
6.2.2 Package merging 80
6.2.3 Package replication 83
6.2.4 Package destruction 88
6.3 Virtual Appliance Rebuilding 91
6.3.1 Rebuilding scenarios and algorithm 91
6.3.2 Rebuilding in active repositories 95
6.3.3 Rebuilding in the IaaS system 99
6.3.4 Reconstructing the virtual appliance 100
7 The Minimal Manageable Virtual Appliance 102
7.1 Introduction 102
7.2 Definition of the manageable virtual appliance 103
7.2.1 Maintaining the Management Capabilities of Virtual Appliances 105
7.2.2 Minimal Manageable Virtual Appliances 106
7.3 Architectural developments 109
7.3.1 Effects on the rebuilding algorithms 109
7.3.2 MMVA based virtual appliance transformation 114
7.3.3 MMVA and the Optimization facility 116

III ANALYSIS 121
8 Methodology 122

8.1 Introduction 122
8.2 Appliance classification 124
8.3 Correctness of the architecture 125
8.4 Basic deployment efficiency 128
8.4.1 Baseline and post-optimization deployments 129
8.4.2 Deployments with rebuilding 130
8.4.3 Deployments utilizing the MMVA 131
8.5 Estimating the cost of the architecture 131
8.5.1 Evaluating the optimization time 132
8.5.2 The cost of initial upload 132
9 Testbed 134
9.1 Introduction 134
9.2 The generic testbed 135
9.3 Testbed with Nimbus 138
9.4 Testbed with Eucalyptus 140
9.5 My proprietary testbed 143
9.6 Summary 144
10 Evaluation 147
10.1 Used virtual appliances 147
10.2 The cost of applying the architecture 148
10.2.1 The cost of initial upload 149
10.2.2 Evaluating the cost of size optimization 150
10.3 The influence of the AVS architecture on deployment 157
11 Conclusions 161
11.1 Future research directions 164

12 PUBLICATIONS DURING RESEARCH 166
BIBLIOGRAPHY 168

IV APPENDIX 176
A VIRTUAL APPLIANCE DEFINITIONS 177
A.1 The SSH Virtual Appliance 178
A.2 The Apache Virtual Appliance 179
A.3 The GEMLCA Virtual Appliance 180

LIST OF FIGURES

Figure 1.1 Relations of deployment tasks 4
Figure 1.2 Basic layout of a virtual appliance and the way it encapsulates a software system 6
Figure 1.3 Generic view of an IaaS system 7
Figure 1.4 The process of appliance based deployment on an IaaS system 8
Figure 1.5 Effects of the different contributions on the service deployment time 11
Figure 3.1 Architectural connections of the Automatic Virtual appliance creation Service 27
Figure 3.2 Hypothetical dependency graph of 6 services 32
Figure 3.3 Possible rebuilding paths of σ_6 from Figure 3.2 33
Figure 3.4 Basic package relations 35
Figure 3.5 Virtual appliance package relations 36
Figure 4.1 Use cases and relations of the AVS subsystem 40
Figure 4.2 Virtual appliance extraction scenarios 42
Figure 4.3 Different states of a Virtual Appliance playground in the local AVS repository 44
Figure 4.4 Options on initiating format transformation of virtual appliances 48
Figure 4.5 The virtual appliance representation of the AVS 51
Figure 5.1 Basic appliance optimization technique 58
Figure 5.2 Overview of the proposed optimization technique 63
Figure 5.3 Average grouping failure rate of the directory based grouping algorithm during optimization 65
Figure 5.4 Parallelism in the validation process 69
Figure 5.5 Number of groups formed from the available items during the optimization of the Apache appliance (see Sections 8.5.1 and A.2) 70
Figure 5.6 Handling virtual machine instances 71
Figure 5.7 Virtual machine management states 72
Figure 6.1 The lifecycle of a virtual appliance with the use of active repositories 76
Figure 6.2 Splitting graph of two virtual appliances 78
Figure 6.3 Basic steps of package replication 83

Figure 6.4 Identifying unused packages 89
Figure 6.5 Activity diagram of the first phase of package destruction 90
Figure 6.6 Options on embedding the rebuilding algorithm 92
Figure 6.7 Rebuilding algorithm utilizing both IaaS systems and active repositories 94
Figure 6.8 Package availability in a specific repository 95
Figure 6.9 Example repository layout with availability information and stored packages 97
Figure 6.10 Virtual appliance rebuilding options 100
Figure 7.1 Minimal interfaces of a manageable virtual appliance 103
Figure 7.2 Interfacing between MMVAs and the AVS architecture 107
Figure 7.3 Deployment client using an MMVA during VA rebuilding 109
Figure 7.4 Transformation applied only on MMVAs 114
Figure 7.5 Handling virtual machine instances of MMVA enabled virtual appliances 118
Figure 7.6 Virtual machine management states with virtual appliances embedding an MMVA 119
Figure 8.1 Classification and relationship of the evaluation scenarios 123
Figure 8.2 Correctness checking scenario of the architecture 125
Figure 8.3 Basic deployment measurements 128
Figure 9.1 The generic testbed used for evaluating the AVS service and its components 136
Figure 9.2 Nimbus based testbed 139
Figure 9.3 Eucalyptus based testbed 141
Figure 9.4 The proprietary testbed 143
Figure 10.1 Comparison of initial upload phases with and without using Algorithm 4.1 149
Figure 10.2 The process of optimization demonstrated on the size of the appliance 151
Figure 10.3 Effect of intermediate virtual appliance creation in the optimization facility 152
Figure 10.4 Comparing the stability of the *remaining* and *reduction* completion conditions – see Section 5.2.1.4 153
Figure 10.5 Effects of *remaining* size exit condition on execution time 154
Figure 10.6 Effects of MMVA usage during optimization 155

Figure 10.7 The number of future deployments required before size optimization becomes profitable 156
Figure 10.8 Deployment phases 158
Figure 10.9 Effects of rebuilding 160

LIST OF TABLES

Table 2.1 Classification of the different deployment solutions 18
Table 4.1 Size comparison of virtual appliances and item hashes 46
Table 4.2 Comparison of basic transformation initiators 49
Table 5.1 Size dependent virtual appliance start timings 55
Table 6.1 Collected availability information 96
Table 6.2 Evaluation of the rebuilding options for p_1 98
Table 7.1 Comparison of the introduced rebuilding scenarios 111
Table 7.2 Comparison of all proposed transformation initiators 115
Table 9.1 Comparison of the testbed infrastructures 146
Table 10.1 Basic virtual appliance properties of the experimental appliances 148
Table 10.2 Deployment times [in seconds] before and after the optimization process 157
Table 10.3 Basic virtual appliance properties of the optimized experimental appliances 158
Table 11.1 Comparison of the proposed architecture to related works 162

NOTATION INDEX

Notation	Description
$\alpha(i_9)$	The aging coefficient of item i_9
$baseva(p_{14})$	True, if p_{14} is a base virtual appliance
$BW(h_3, h_4)$	Network bandwidth between hosts
$BW_P(p_{33}, r_{12})$	The bandwidth required serving the requests for package p_{33} from repository r_{12}
$BW_R(p_{34}, r_{16})$	The bandwidth utilized while serving the queries for package p_{34} of repository r_{16}
C_φ	The set of IaaS services in φ
c	A particular IaaS service
$configurator(p_{23})$	The configurator component of package p_{23}
$contents(r_1)$	The set of packages stored in a particular repository – r_1
$correct(p_{25}, \varphi)$	Checks whether p_{25} can be correctly extracted, optimized, decomposed and rebuilt by the architecture in infrastructure φ
$correlated(p_{30}, r_2)$	The set of packages that are downloaded in correlation with p_{30} from r_2
$dep(p_4)$	The direct package dependency set of package p_4
$delta(p_{13})$	Evaluates if package p_{13} is a delta package
$D(p_6, n)$	The nth dependency set of package p_6
$D_{con}(p_{48}, h_8)$	The construction path: the dependency set of package p_{48} that offers the smallest total rebuilding time on host h_8
$DG(p_{31})$	The group of packages with the common ancestor p_{31}
$executable(X)$	Evaluates to true when the virtual appliance image $X \in \mathcal{I}_\varphi$ can be executed on the service-based system
$evaluate(f, arg)$	Evaluates the function f with arguments arg

Notation	Description
$ext(P_3, r_{28})$	Defines the subset of P_3 with those elements that are not available in the $contents(r_{28})$
F	The set of supported virtual appliance formats
f	A particular virtual appliance format
$\gamma(i_{10})$	The prior group participation coefficient of item i_{10}
h	A host in a service-based system
h_{AVS}	The host of an AVS service
h_{cli}	The host of the deployment client
h_{s,q_2}	The source host of the download query q_2
$H(p_{21})$	The hash values of all items in package p_{21}
$hash(i_1)$	The calculated hash value of a particular item
$hosted(\xi_1)$	The set of virtual machines hosted on a particular virtualization-enabled host of an IaaS system
I_φ	The set of possible items in a particular service-based system – φ
i	A single item – the smallest building block of a virtual appliance
$IE(r_{11})$	The combined non-optimality value of repository r_{11}
$ie_{ed}(r_9)$	The non-optimality value of external dependencies required for packages in r_9
$ie_{fr}(r_8)$	The non-optimality value of frequently requested content of r_8
$ie_{nsc}(r_5)$	The non-optimality value of non-stored content of r_5
$X \in Y$	Set Y contains element X
$items(p_1)$	The set of items that form package p_1
$\mathcal{I}_\varphi$	The set of possible virtual appliance images in a particular service-based system – φ
$im(p_{12}, f_1)$	Defines the virtual appliance image stored in package p_{12} with the format f_1
$initVM(p_{17}, \varphi)$	Initiates a virtual machine with the virtual appliance provided by p_{17} in the service based system φ

Notation	Description
$jeos(p_{19}, p_{20})$	True, if p_{19} contains the support environment of p_{20}
$\kappa(i_8)$	The previous removal success rate of i_8 according to the knowledge base
$l(h_1, h_2)$	Network latency between hosts
$latgroup(\epsilon, r, r_s)$	Defines a group from the users of r_s, who share similar connection properties as r
$M(i_5)$	The number of validated siblings of item i_5
$M_{faulty}(i_6)$	The number of successfully validated siblings of item i_6
$M_{success}(i_7)$	The number of faulty validated siblings of item i_7
$measure(X(Y))$	The median execution time of function X with arguments Y
$manageable(p_{49})$	Determines the if package p_{49} offers management interfaces
n	The index of the possible dependency sets
$N_{dep}(p_{27}, \varphi)$	The number of deployments needed to overcome the cost of optimization
$N_F(i_{14})$	The number of unsuccessful removal attempts of item i_{14} prior a given size optimization operation
$N_T(i_{15})$	The total number of removal attempts on item i_{15} prior a given size optimization operation
$\varnothing$	The empty set
$online(vm_1)$	True, if vm_1 is executed currently
$optimalsize(p_{18})$	Determines if the given package (p_{18}) is minimal
Φ	The set of service-based systems
φ	A particular service-based system
$\wp(X)$	Power set of X
P_φ	The set of all available packages in a service-based system
p	A single package
p_β	A base virtual appliance
$p_c(r_{14})$	The package selected for replication from repository r_{14}

Notation	Description
p_{Δ}	A delta package
p_{σ}	A service package
p_{Ω}	A self-contained package
$p_{\Omega,common}$	The common parts of two virtual appliance packages after applying the decomposition algorithm on them
$p^{*}_{\Omega,nm}$	A non-manageable self-contained virtual appliance package
p_{req,q_4}	The package downloaded as a result of download query q_4
$PC(p_7)$	The number of possible rebuilding paths starting from package p_7
$pkgsize(p_3)$	The size of the package p_3
$pkgforms(p_{22})$	The set of appliance image formats available in package p_{22}
Q_{φ}	The set of download queries in the service-based system φ
q_1	An arbitrary download query
$Q_{amg}(r_6, r_7)$	The set of queries where repository r_7 was requested to download a package from repository r_6
$Q_{All}(r_3)$	The set of queries where repository r_3 behaves as the requested repository
$Q_{Ex}(r_4)$	The set of queries where repository r_4 behaves as the query source
Q_{latg}	The set of queries that were initiated by the users of a specific latency group
$Q_{pac}(p_{34}, r_{13})$	The set of queries that request package p_{13} from repository r_{13}
R_{φ}	The set of repositories in φ
$R_{dep}(p_{45}, n)$	The set of repositories that contain parts of the dependency set $D(p_{45}, n)$
r	A specific repository in the service-based system
r_{AVS}	The virtual appliance playground storage
r_c	The assumed repository for the biggest latency group

Notation	Description
$r_{ext}(p_{43}, r_{26})$	The repository that offers the minimal external rebuilding time for p_{43} when the package was originally requested from r_{26}
$r_{ler}(p_{35}, r_{18})$	The least expensive location for rebuilding package p_{35} in a repository if the requested repository was r_{18}
r_{req,q_3}	The repository that serves the download query q_3
r_s	The repository selected as the source for replication
r_t	The repository selected as the target for replication
$related(i_3, P_1)$	Defines the subset of packages $P_1 \in P_\varphi$ that contain the item i_3
$S(p_{26}, \varphi)$	The speedup on deployment time achieved by the architecture after the optimization of p_{26} for size or delivery
$s_N(X)$	Sample standard deviation of function X
$SD(p_8, X)$	The index of the element in the direct dependency set of p_8 used by the Xth rebuilding path
$selfcontained(p_{10})$	Evaluates to true if p_{10} is a self-contained package
$servicepkg(p_{15})$	True, if package p_{15} contains the target functionality for the users
$size(i_2)$	Size of a specific item – i_2
$stsize(p_{50})$	The storage size of package p_{50}
$t_{comp}(p_{40}, h_7)$	The time of applying the composition rule on p_{40} and its dependency set on the host of h_7
$t_{exr}(p_{42}, r_{24}, r_{25})$	The external rebuilding time of package p_{37} when the rebuilding is accomplished by repository r_{24} even though the package was originally requested from r_{25}
t_{f,q_6}	The time the download query q_6 has completed
t_{IUBL}	The baseline measurement for initial upload time
t_{IUOpt}	The optimized initial upload time

Notation	Description
$t_{oplor}(p_{36}, r_{19})$	The total optimal rebuilding time of the self-contained package ($\sum D(p_{36}, x)$) if it is rebuilt entirely in r_{19}
$t_{opexr}(p_{37}, r_{20})$	The minimal external rebuilding time of package p_{37} when the rebuilt package is requested by r_{20}
$t_{opreb}(p_{41}, r_{23})$	The minimal local rebuilding time of package p_{41} when the rebuilt package is requested by r_{23}
$t_{oir}(p_{46}, \xi_2)$	The minimal IaaS based rebuilding time of package p_{46} on execution host ξ_2
$t_{reb}(p_{38}, r_{21}, h_5)$	The rebuilding time of p_{38} on the host of h_5 when the package is downloaded from r_{21}
t_{s,q_5}	The time the download query q_5 was received
$T^{ar}_{totreb}(p_{44}, n, r_{27})$	The total rebuilding time of package p_{44} when r_{27} selects the rebuilding location for every package in the dependency set of $D(p_{44}, n)$
$T^{iaas}_{totreb}(p_{47}, n, \xi_3)$	The total rebuilding time of package p_{47} when ξ_3 downloads and rebuilds every package in the dependency set of $D(p_{47}, n)$
$t_{trans}(p_{39}, r_{22}, h_6)$	The transfer time of p_{39} to the host of h_6 from repository r_{22}
$t_{TS}(i_{16})$	The number of the last optimization iteration in which item i_{16} was evaluated for removal
τ	The pre-transfer measurement threshold
$\Theta(p_9)$	All possible dependency sets of p_9
$X \cap Y$	Intersection of two sets
$X \cup Y$	Union of two sets
$X \backslash Y$	The set of those elements in X that are not present in Y
$\|X\|$	The cardinality of set X
U_φ	The service users of the system φ
u	A single service user
$used(r_1, P_2)$	Filters the package set P_2 to contain only those packages that were downloaded from repository r_1

Notation	Description
$unused(r_{15})$	Specifies the set of packages that were not downloaded from repository r_{15} within a pre-specified time period
$va(p_{11})$	Determines whether p_{11} is a virtual appliance that can be instantiated in a virtual machine
$VAforms(c_2)$	The set of the virtual appliance formats supported by IaaS servcie c_2
$valid(p_{16}, h_x)$	Evaluates the validators of p_{16} on host h_x
$validator(p_{24})$	The validator component of package p_{24}
vm	An arbitrary virtual machine in the service-based system
$vms(c_3)$	The set of virtual machines managed by IaaS service c_3
$w(i_4, p_{26})$	The evaluated value of weight function w for item i_4 in package p_{26}
$w_A(i_{11}, p_{27})$	The weight function for volatile item i_{11}
$w_P(i_{12}, p_{28})$	The pattern matching weight function of i_{12}
$w_S(i_{13}, p_{29})$	The basic, size based, weight function of i_{13} in package p_{29}
$\Xi_\varphi(c_1)$	The set of virtualization-enabled hosts managed by a specific IaaS service – c
ξ	The target host for deployment where the desired virtual appliance is instantiated

GLOSSARY

A

amazon machine image (AMI) Amazon's proprietary virtual appliance format, p. 15.

application content service (ACS) An Open Grid Forum standard proposal for application repositories in Grids, p. 134.

automated virtual appliance creation service (AVS) The proposed solution to create and optimize virtual appliances, p. 26.

D

domain (Dom) Single virtual machine provided by Xen, p. 7.

E

elastic compute cloud (EC2) The IaaS offering of Amazon, p. 15.

H

hypervisor (HV) The Virtual machine monitor of xen virtual machines, p. 7.

I

infrastructure as a service (IaaS) A cloud computing platform offering the outsourcing of hardware resources, p. xviii.

J

just enough operating system (JeOS) All the support system of an application – including the OS, but excluding the application itself, p. 28.

M

minimal manageable virtual appliance (MMVA) A size optimized embeddable virtual appliance with management capabilities, p. 107.

O

open grid services architecture (OGSA) The vision of the Open Grid Forum about the service grid concept, p. 19.

open virtual machine format (OVF) A standard proposed by Distributed Management Task Force (DMTF) for unified representation of Virtual Machine states, p. 39.

P

platform as a service (PaaS) A cloud computing platform offering the outsourcing of software platforms, p. 2.

S

simple storage service (S3) Amazon's repository implementation, primarily referred as a virtual appliance repository in this dissertation, p. 15.

software appliance The software system is packaged and delivered with the software environment supporting its execution (including the OS and the necessary libraries and third parties), p. xix.

software as a service (SaaS) A cloud computing platform offering the outsourcing of applications, p. 2.

V

virtual appliance (VA) A software appliance prepared for virtualized environments, p. 26.

virtual machine (VM) Virtualization of hardware resources of an entire computer, p. 6.

virtual machine monitor (VMM) Offers virtual machine management and execution capabilities, p. xviii.

virtual workspace service (VWS) The IaaS offering of the Nimbus project, p. 39.

W

web services resource framework (WSRF) A group of OASIS standards for unified state management of Web Services, p. 11.

X

xen virtual machines (Xen) A hardware virtualization solution proposed in [7], p. xviii.

Part I

RESEARCH OVERVIEW

1 INTRODUCTION

Services abstract system functionalities from the applied technology and they enable the access of these functionalities through predefined interfaces. In service-based systems [25] these interfaces, defined by service descriptions, allow the users to access the services transparently without knowing the exact details of the used service instance. The vision of these service-based systems incorporates highly dynamic environments [23] where service instances are deployed, utilized, and decommissioned on demand. Service deployment [78] prepares the service code on the infrastructure of the service provider for later usage. Dynamism in the service-based systems requires the automation of the service deployment process.

Cloud computing [5, 14, 31, 85, 86] promises a simple and cost effective outsourcing of applications (Software as a Service – SaaS), software platforms (Platform as a Service – PaaS) or hardware resources (Infrastructure as a Service – IaaS). PaaS and IaaS systems can be used to extend service-based systems by deploying services on their resources. On PaaS cloud systems the services are developed specially for the given platform. In contrast, IaaS systems use hardware virtualization to support a wider variety of applications. These systems require the encapsulation of the services in virtual appliances. Therefore, services are deployed by instantiating a virtual machine in the IaaS system.

Virtual appliances combine services and their support environment in a form executable by virtual machines (see Figure 1.2). In current IaaS systems users either apply virtual appliances already published in a virtual appliance marketplace [20, 53, 56, 82] or they have to *create the required virtual appliance* on their own. However, these newly created virtual appliances are not specifically designed with their frequent deployments in mind and this seemingly minor issue can seriously hinder exploiting the dynamic features of the system. For example some IaaS systems charge for the network usage during deployment, therefore, improperly created virtual appliances entail hidden deployment costs for their users. Hence, one of the main aspects of this research is providing the appliance developers an automated mechanism and support for *creating virtual appliances*.

In highly dynamic service environments a service request might impose a service deployment before the request can be executed (pre-execution deployment). However, the inclusion of the deployment has a negative effect on the apparent execution time of the request. Current appliance based deployment systems (e.g. [74]) try to *reduce the effect of pre-execution deployment* by replicating the virtual appliances within the IaaS system. However, IaaS systems regularly charge for the extra storage needs of the owner of the replicated virtual appliance. The other main aspect of this work means to reduce turnaround time of service requests and to keep deployments transparent to the users.

In this dissertation, I propose the automated virtual appliance creation service (AVS) that supports the service developers in the *virtual appliance creation* and publication process. This service supports most of the deployment tasks (except selection that is the most independent task of the deployment and I have discussed it outside of the scope of this dissertation in [44, 45, 46]). The service also offers solutions for acquiring, distributing and optimizing virtual appliances. The AVS creates the service's virtual appliance based on an already operable service installation on the developer's system. After the initial virtual appliance is created the developer can request the service to prepare and publish the appliance for execution in various IaaS systems.

This dissertation also introduces an optimization facility that aims to minimize the size of the virtual appliances before they are published for widespread use. The smaller sized appliances allow faster deployment than the initially created virtual appliances, therefore, this technique targets the *reduction of the effect of pre-execution deployment*. The facility requires the appliance developer to specify validator algorithms for the virtual appliance, these algorithms are intended for automated use and they test the functionality of the service encapsulated in the appliance. If these algorithms are present, then the facility iteratively removes parts of the initial virtual appliance and after each iteration it ensures the success of the removal by validating the now smaller appliance.

I also present active repositories as another approach for *reducing the effect of pre-execution deployment*. These repositories automatically decompose the stored virtual appliances to smaller parts, thus they allow the partial replication of the appliances. The proposed approach only replicates virtual appliance parts that are common in the stored virtual appliances. As a consequence to decomposition, my solution incorporates a set of rebuilding algorithms for reconstructing the original virtual appliances on the target site of the deployment.

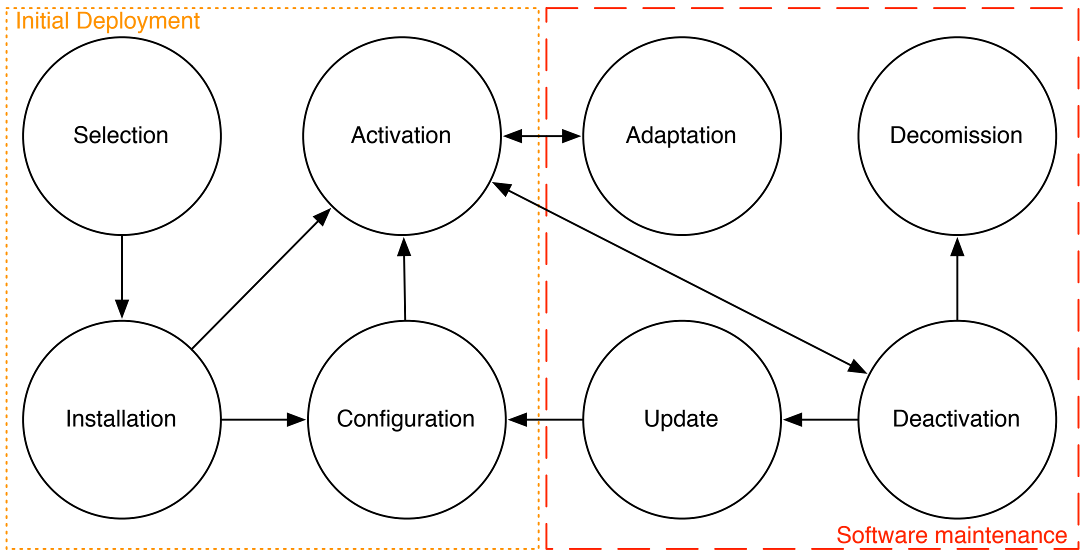


Figure 1.1: Relations of deployment tasks

Finally, this dissertation presents the concept of minimal manageable virtual appliances. These special virtual appliances can form the base of the future appliances. If the appliance developers decide to incorporate these virtual appliances into theirs, then the proposed architecture – based on the AVS service, the size optimization facility and the active repositories – utilizes the management capabilities of the appliance. I reveal that based on these capabilities the architecture becomes more *widely adoptable* and more effective in *reducing the effect of pre-execution deployment*.

The rest of this chapter introduces the basic concepts on which this thesis is based. First introducing the process of service deployment in highly dynamic service environments, then identifying the requirements of an architecture supporting service deployments and provides a taxonomy for the classification of the related works.

1.1 SERVICE DEPLOYMENT OVERVIEW

Service deployment [78] is the process of making a service instance available for the users. I define deployment as a complex process that is composed of following deployment tasks (see Figure 1.1): selection, installation, configuration, activation, adaptation, deactivation, update, and decommission. Systems that offer at least the triplet of installation-configuration-activation are referred as *service deployment systems*. The next paragraphs detail the purpose of these tasks.

First, the *Selection* task chooses the appropriate (hardware and software) target system to deploy the service on. Then the *Installation* task manages the addition of the new software components to the target system. These components include the service itself and its dependencies that are described in [9, 26, 70]. The installed components are adjusted to fit the target system's specific needs with the *Configuration* task [47, 78]. During the *Activation* task the deployment system makes the service available for the users, and usually executes it.

The remaining deployment tasks are directly related to software maintenance. While the target system is still running and serving requests, its previously installed and configured software components are fitted with the *Adaptation* task to its current needs. Then the *Deactivation* task enables those maintenance related operations that cannot operate on an active service. The *Update* task replaces previously installed software components and initiates a reconfiguration on them. Finally, the *Decommission* task removes the service's software components from the target system, and then issues a reconfiguration task for the remaining software systems.

I have identified three types of service deployment systems that can accomplish the previously mentioned deployment tasks: (*i*) *manual* service deployment systems, (*ii*) *container based* deployment systems and (*iii*) *appliance based* deployment systems. Manual deployment systems require continuous user interaction during the deployment process, therefore, they are not suitable in highly dynamic service environments. Container based systems (e.g. [15, 62]) predefine an execution platform for the services that includes the deployment system. Therefore, services have to be developed for this specific platform and they are not portable between the different container based deployment systems. Finally, appliance based deployment systems (e.g. [42, 44, 48, 63]) require the encapsulation of the deployable services and their specific support environments in virtual appliances. Thus, service deployment is achieved by instantiating a virtual machine that executes the virtual appliance with the service. The next section elaborates the concept of virtual appliances.

1.1.1 *The appliance model*

During their life cycle, software systems may have several versions and subversions. The diversity of their different installations means their vendors have to support an unforeseen number of software environments. These software environments are built up from several software components interfaced with each other. However, the fact that they are installed on the same system means they could also interfere with each other. In several cases, the software vendor cannot identify the cause of an irregular situation because the

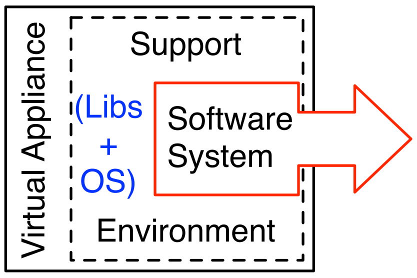


Figure 1.2: Basic layout of a virtual appliance and the way it encapsulates a software system

vendor is not capable of reproducing the exact same software environment its software components were deployed in.

To reduce the irregularities in deployments the vendor maintains a specialized environment, and the software system is executed in this controlled environment only. There are two distinct ways to offer this environment:

SOFTWARE AS A SERVICE [18] approach allows the vendor to deploy and maintain the software system within the administrative domain of the vendor. Users access the functionality of the software through well-defined interfaces over the network. Depending on the target audience, this could mean a web based user interface or a service interface, etc. Since the software is deployed and maintained internally by the developer, this scenario is not discussed further in this thesis.

SOFTWARE APPLIANCE [3] model requires the vendor to package the software system and its specialized support environment (including the OS and the necessary libraries), and then the model expects the vendor to offer the previously packaged system with specific hardware requirements. By buying the appliance, the user gets all the necessary licenses for the included software packages (for the OS, also the libraries and the offered high level software system). The resulting appliance (*i*) simplifies the support for the service vendor by constraining the service's environment and (*ii*) it also allows easy installation, update and decommission of the service. For example, when the appliance is provided on a hard disk drive then the installation of the service only requires its connection to the machine that meets the predefined requirements.

Virtual appliances (or VAs - [71]) are software appliances prepared to run in virtualized environments. A virtual appliance defines a virtual machine (VM) state that contains the software system and its support environment (see Figure 1.2). Virtualized environments [76] range from software to full hardware

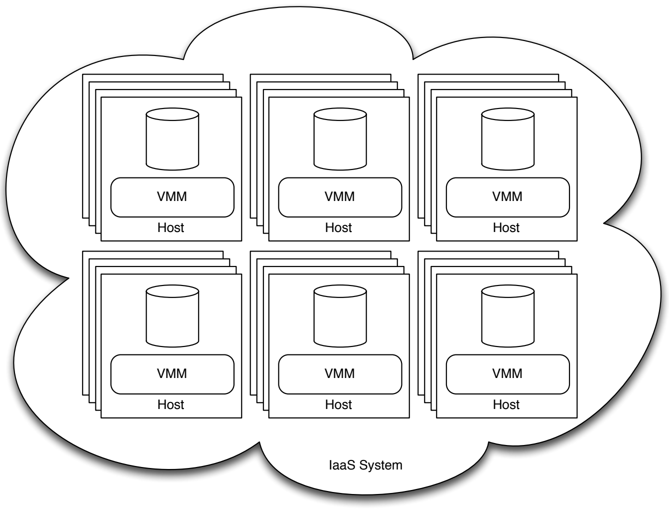
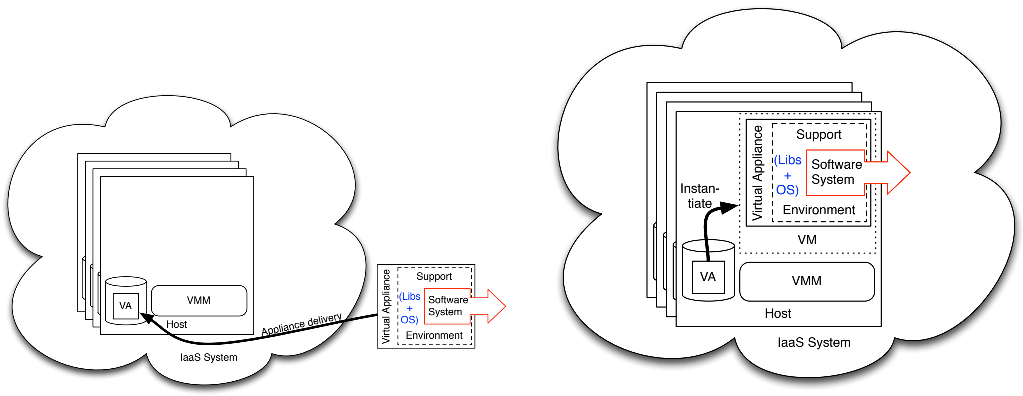


Figure 1.3: Generic view of an IaaS system

virtualization. Software virtualization encapsulates the software appliance and isolates its execution from the OS by concealing its interfaces (e.g. the Wine project [2] provides virtual Windows API on top of UNIX systems). As an opposite, hardware virtualization offers virtual machines limiting the utilized resources (e.g. maximum processing power, network bandwidth) by the software installed on them. VMs can also provide different instruction sets as the host hardware. Hardware virtualization is provided by virtual machine monitors (VMM – [1, 7, 11]) that offer virtual machine management functionalities, including their creation, start up and shutdown procedures on their hosting machines.

In the scope of this thesis, a good example for hardware virtualization is Xen [7] that uses the hypervisor as its VMM. In Xen terminology, a virtual machine is called a domain that represents a part of the physical machine (or a subset of physical resources). Domains are numbered, and the number 0 domain is a privileged domain that can control the hypervisor. Xen specific virtual appliances are prepared to run in one of the Xen domains, therefore, they store all the information required to instantiate a domain. If a software system is installed, configured and activated in a domain, then saving the state of that particular domain will result a Xen specific virtual appliance of the software system.

Cloud computing [5, 14] is an emerging concept that is based on the Software as a Service paradigm. Thus cloud computing provides services that are managed centrally by software vendors. *Infrastructure as a service clouds (IaaS)* provide services to remotely access VMM functionalities (see Figure 1.3). IaaS systems offer the functionality to create, manage and destroy virtual ma-



(a) Appliance delivery (b) Virtual Machine instantiation

Figure 1.4: The process of appliance based deployment on an IaaS system

chines on multiple host machines. On creation, the user provides a virtual appliance (see Figure 1.4a) that the IaaS system uses to instantiate a virtual machine (see Figure 1.4b). This appliance therefore has to be created taking into account the requirements set by the virtual machine provided by the IaaS. Therefore, the developer of the appliance has to prepare its software system as a software appliance that can run on the virtual machines of the IaaS system.

1.2 REQUIREMENTS FOR AUTOMATING SERVICE DEPLOYMENT

I have assessed the related works (later detailed in the next Chapter) to identify the approaches, issues, common requirements and deployment tasks they implemented. This section distills these requirements and forms a taxonomy based on them. To define the requirements, first, I have collected the possible actors who would need automation from the deployment system. These actors include service brokers, orchestrators, composition engines, service containers, etc.

Afterwards, I have identified the requirements an automated deployment system planning to support these actors should comply with: (*i*) externally controllable deployment tasks, (*ii*) scalability with the size of the service-based system, (*iii*) reduced data storage, (*iv*) minimal deployment time, (*v*) minimal disruption of the other services in the service-based system. The next paragraphs discuss these requirements in detail.

First, automation of service deployment requires that the various deployment tasks (see Section 1.1) should be exposed with their interfaces to the outside world (allowing their *external control* and manipulation). As a result, even those components (e.g. service brokers) that are not directly related (or *external*) to the deployment system of the service-based system can influence the entire deployment process.

Service-based systems are massively distributed environments, thus the deployment system should be able to *scale* to the size of the service-based system. For example, larger sized service-based systems involve more frequent deployments. However, the increase in the number of deployments should not affect the normal operation of the system.

Frequent deployments intensify the amount of data transferred to the target sites that could affect the network connections of already deployed services. Large-scale service-based systems involve a large number of deployable services that are stored in repositories. The deployment system should *reduce* the size of the deployable service components in order to achieve their effective storage and transfer.

Highly dynamic service environments require the deployment system to frequently perform service deployments and decommissions. Service deployment often precedes a service call to the newly deployed service instance, because the controlling components of the service-based system regularly instantiate deployments when the timely execution of a service call requires it. The deployment system should *minimize the deployment time* in order to reduce the total time of the service call on the newly deployed service instance.

Finally, new service deployments should not *disrupt* the service-based system's overall behavior. The newly deployed service should not be able to obstruct the ongoing tasks of the previously deployed services.

1.2.1 *Taxonomy of Related Deployment Systems*

To provide further details on the requirements of the proposed deployment system I established the following categorization for service deployment systems: (*i*) *isolation level* defines the separation between services installed on the same host; (*ii*) *repository support* enables the storage of the code of the available services for deployment; (*iii*) *universal* service support increases the number of deployable services in the system; (*iv*) *non-invasiveness* requires no modifications on the service code in order to support deployment; finally, (*v*) *state transfer* enables the newly deployed system to resume from the state of a remote service.

Service deployment solutions are categorized depending on their *isolation level* that defines the level of service separation during the deployment on a host already offering services. The lowest isolation level means all the other services are stopped and their states are lost. The highest isolation level does not decrease the turnaround time of the other service invocations during and after deployment or even when malicious code gets activated with a newly deployed service. There are two main approaches to tackle the isolation problem. The first built on the fact that *service containers* offer basic isolation, however service containers do not separate the services entirely (e.g. newly deployed services can exhaust system resources, thus degrading the previously deployed services). The second approach provides isolation with *virtualization* that offers the highest isolation levels. Isolation ensures that improper deployment decisions do not influence the overall system performance, therefore, *virtualization based isolation is a key requirement* for an automated deployment system.

Service deployment solutions are also categorized depending on their support of *repositories*. Repositories could act as the primary source of trust if they enclose security information (e.g. a signature of the service's developer) or enforce different registration policies – for example they require the validation of services. Without repositories, service code has to be collected before every deployment operation. This gets even worse when the same service has to be collected and deployed several times, instead of downloading it from an already prepared location – the repository. The continuous repetition of the expensive service code collection tasks reduces the overall performance of the service-based system even though deployments were initiated to avoid performance drop. Therefore, *repositories are required* for the automation of deployment by acting as the sources of deployable services.

The next categorization is the *universality* of the deployment solution that defines the generality of the deployment solution with regards to the deployable services. Specialized service deployment solutions are optimized for a specific service. They can support all the deployment tasks from installation towards adaptation and decommission; however, to simplify the deployment problem they specialize these tasks for a given service. Container based isolation techniques jeopardize universality by supporting services only compatible with the chosen container. In service-based systems, every service is represented with its interface that cannot be used to differentiate between deployable and non-deployable services. With specialized deployment solutions only few services are deployable, therefore, highly dynamic service environments *require universal deployment* solutions that are not differentiating the services in the service-based system.

Another categorization is based on the level of *invasiveness* the deployment solution enforces on service developers. Invasive systems require the service code to be modified to support the service's deployment. E.g., these systems require the service to implement an additional interface or they require the use of special libraries and solutions that enable further deployments. As an opposite, non-invasive systems are more compatible with the existing service-based systems, therefore, a *non-invasive deployment solution is required*.

Finally, there are deployment solutions with *state transfer* capabilities. State transfer in service-based systems require that a service suspended on a site can be resumed on a different one. In this case, the whole process depends on the state representation of the service [59]. The deployment system might introduce new interfaces for state transfer, it might require the developer to implement the state transfer mechanisms for its own system, and finally, it could use a standardized state representation mechanism (like the Web Services Resource Framework – WSRF – [6]).

1.3 CONTRIBUTIONS

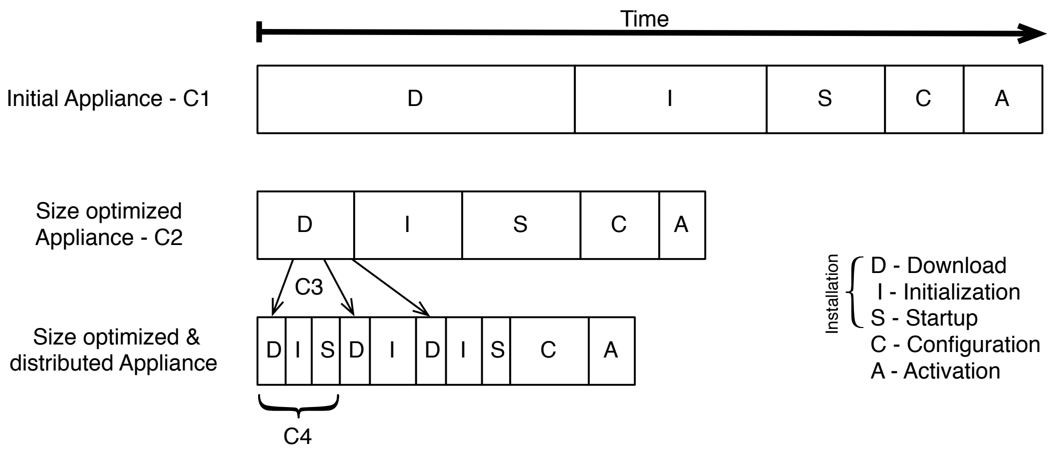


Figure 1.5: Effects of the different contributions on the service deployment time

This chapter highlights my contributions to the knowledge. Figure 1.5 gives an overview on the effects and relations of my contributions. The figure presents the various deployment tasks that are affected by my contributions. As this research is mainly focused on the installation deployment task, I have identified three subtasks that can be used to present the results of this dissertation.

Installation time is composed of three separate components. The first one is the *download* time (see "D" in Figure 1.5) required for transferring the virtual appliance from the repository to the target host. This time mainly depends on the size of the virtual appliance. The second one is the *initialization* time (see "I" in Figure 1.5) needed for the IaaS provider to instantiate a virtual machine that hosts the transferred virtual appliance. The initialization time only depends on the IaaS provider. Finally, the third one is the *startup* time (see "S" in Figure 1.5) spent during the initial startup of the initiated virtual machine (until the service is ready for configuration). It mainly depends on the virtual appliance developer.

The figure reveals that my initial contribution allows the creation of the initial virtual appliance (with the help of my first contribution). This initial virtual appliance and the service that allows its creation are used as the baseline for my later research. My second contribution targets the size optimization of the initial virtual appliance in order to allow shorter deployment time. My third contribution offers a technique to automatically decompose and replicate appliance contents among repositories. The third row of Figure 1.5 reveals the effects of the rebuilding algorithms used during the installation task by showing the repeated execution of the download and initialization subtasks during a single deployment. Finally, my last contribution offers the concept of minimal manageable virtual appliances that can be used to increase the efficiency of the rebuilding and size optimization techniques.

1.3.1 *Approach for initial Virtual Appliance creation (C1)*

My initial contribution facilitates the virtual appliance creation process by allowing the developer of the virtual appliance to work in the environment they are used to. Therefore, I have defined an architecture and a methodology for creating virtual appliances. I have presented this methodology through a scenario that introduces the three cornerstones of this contribution. First, I have designed two approaches for the initial creation of the virtual appliance. Then, I have contributed with the identification of the metadata supporting the deployment time optimization processes (see contributions C2-C3). Finally, I have contributed the initial upload algorithm that automatically selects, decomposes and uploads the acquired virtual appliance to a repository with the shortest estimated upload time.

1.3.2 *Parallel Algorithm for Virtual Appliance Size Optimization (C2)*

My research revealed that reducing the service's encapsulating virtual appliance could significantly influence the service deployment time. Consequently, my second contribution provides a parallel active fault injection based algorithm that reduces the size of a virtual appliance. The algorithm removes sections of the virtual appliance while it ensures the target functionality of the encapsulated service is still provided by the remainder of the appliance. I reveal that the proposed algorithm is independent from the applied virtual appliance sectioning. The foundations of the algorithm are the removable selection algorithms and the gradually increasing granularity of the removable parts during the optimization process.

1.3.3 *Distributed virtual appliance storage and delivery (C3)*

The third contribution offers a method for storing virtual appliances distributed among repositories. My research shows that appliances stored in repositories could have common components and elaborates an algorithm to identify, separate and replicate them depending on the demand of the different appliances. I have defined the metadata to be shared among repositories in order to allow automated selection of the target sites for the replicas. I have identified the role of the active repository that autonomously organizes its contents with the help of the previous algorithms. Finally, I have defined various algorithms that rebuild the decomposed appliances before deployment.

1.3.4 *Minimal Manageable Virtual Appliance (C4)*

My final contribution provides the foundation for virtual appliance delivery and size optimization techniques by increasing their efficiency. First, I have elaborated the requirements for the management capabilities that a virtual appliance should meet to support these techniques. Then, I have defined an algorithm that captures the substance of virtual appliances matching the requirements. This algorithm creates the minimal manageable virtual appliance (MMVA) that can act as the base for developer provided appliances. This research also reveals a methodology for the developers to incorporate the MMVA in their own appliances to enable the previously mentioned deployment tasks.

1.4 THE STRUCTURE OF THE THESIS

The rest of the thesis is organized as follows. First, in Chapter 2, I discuss the related works in the context of the requirements and taxonomy discussed in Section 1.2. Throughout the related works I present those research results that are directly related to my contributions. Next, the thesis is subdivided into two parts: the achievements and the analysis. In these parts, every chapter starts with a short overview aligning its contents with my research results. Chapters finish with a short summary that reveals their most important achievements. The next paragraphs discuss the chapters about the two main parts of the thesis.

Throughout the *achievements* part, I reveal the research background (see Chapter 3) of my findings including the outline of the proposed architecture and the theoretical basis for later chapters. Afterwards, I have dedicated a chapter for each of my contributions. First, a detailed description my first contribution can be found in Chapter 4. Next, I reveal the research results about my Parallel algorithm for Virtual Appliance optimization in Chapter 5. Then, my third contribution (with the principal topics on active repositories and virtual appliance rebuilding) is detailed in Chapter 6. Finally, Chapter 7 discusses my contribution on minimal manageable virtual appliances and how they affect the behavior of my proposed architecture.

In the *analysis* part, I present the evaluation of the proposed architecture in three chapters then I conclude the thesis with the last chapter. The three chapters that are dedicated to evaluation first start with the discussion on the applied methodology in Chapter 8. This methodology provides an overview on the evaluation scenarios used in later parts of the thesis. Chapter 9 utilizes the proof of concept scenario and evaluates the architecture from infrastructure independence by providing independent implementations on three different testbed infrastructures. Finally, evaluation concludes with Chapter 10, where I present my measurements on the testbed to reveal the cost effects and deployment time reduction capabilities of the proposed architecture.

2 RELATED WORKS

This chapter aims at discussing the various research areas that can be associated with the contributions of this dissertation. First, the chapter discusses several IaaS systems that can form the basis of the architecture introduced in Chapter 3. Later, some of these IaaS systems are used as the basic fabric on which my findings are demonstrated. Next, this chapter also discusses the research efforts related to the topic of grid and web service deployment. In this chapter, I also analyzed the identified deployment systems based on their support for the different deployment tasks and classified them according to the taxonomy I have revealed in the previous sections (see Sections 1.1 and 1.2.1). Finally, the current approaches for virtual appliance delivery and size optimization are revealed in Sections 2.3 and 2.4.

2.1 INFRASTRUCTURE AS A SERVICE CLOUDS

One of the most established and widespread commercial infrastructure as a service cloud provider is Amazon. They offer a wide range of web services based on their *Elastic Compute Cloud* (*EC2* – [50]). This service offers SOAP and http query interfaces to handle basic virtual machine management functionalities such as creation, registration, termination, restart and firewall configuration. EC2 supports the creation of virtual machines from Amazon Machine Images (AMIs) – the Amazon notation for Virtual Appliances. These AMIs are stored in Amazon's own repository service called Simple Storage Service or S3. Advanced users can publish AMIs [53], however the creation of the AMIs is really left to the developers themselves, because Amazon only provides basic facilities to upload an already prepared Amazon compatible disk image as an AMI. The lack of support for AMI creation reveals that Amazon's primary concern is serving already available appliances to large user communities.

Several open-source solutions implement Amazon's IaaS interfaces. First, *Eucalyptus* [58] has implemented the EC2 and S3 services as Cloud Controller and Walrus respectively. The second implementation is *Nimbus* [42] that has been used for service deployments with manually created virtual appliances as described in [63]. However, Nimbus provides multiple interfaces on its IaaS functionality (one based on Amazon EC2 and one on their own proprietary WSRF service). Finally, *OpenNebula* [29] offers only EC2 interfaces and no repository (S3) implementation; as a result, it cannot be used as fully compatible open source replacement for Amazon's IaaS system. OpenNebula's Amazon API implementation is partial, however they support virtual machine management on top of various VMMs and other IaaS systems. Current IaaS systems offer the ways to reach the infrastructure, however they lack the tools to effectively create virtual appliances to utilize their services.

Eucalyptus [58] is an open source implementation of the public Amazon EC2 and S3 interfaces. Their EC2 implementation is referred as Cloud Controller, while their S3 implementation is Walrus. Eucalyptus is capable to build a hierarchical infrastructure where a single cloud controller can control multiple clusters hosting basic Eucalyptus services. In this hierarchy, each cluster runs a cluster controller service that acts as the single point of entry to the cluster's virtual machine monitors. Cluster controllers are also responsible for the most distinguishing feature of Eucalyptus: they can form virtual networks to separate the virtual machines of different users. Finally, each VMM is exposed through the node controller service that is capable of the basic virtual machine management functionalities on individual hosts. The virtual appliance creation tools of Eucalyptus do not differ from Amazon's, because Eucalyptus uses the client software Amazon provides.

Nimbus [54] was developed as an incubator project for the Globus Toolkit. From the early stages [43] of its development, Nimbus was designed to support multiple types of virtual machine monitors (for example Xen [7] or kvm). In [63] they have also demonstrated that their service is capable of deploying services packaged as virtual appliances. They provide an Amazon EC2 compatible interface and one that is inherited from the Globus Toolkit 4 – WSRF. Later on their focus moved towards the contextualization or deployment time configuration of virtual clusters [41]. Until recently the Nimbus service was supporting local clusters only, however with the current developments [42] locally formed cloud infrastructure services can be interconnected as meta-cloud services similarly to the Cloud Controller feature of Eucalyptus. Similarly to Eucalyptus, Nimbus also offers an S3 implementation (called Cumulus). However, their repository support is not restricted to this implementation, since they also provide generic interfaces to support third party repositories. Nimbus recommends the use of third party virtual appliance creation tools; as a result, they leave the delivery optimization for those tools.

VMPlants [48] goes a step further with appliance-based deployment and offers faster delivery of the virtual appliances by constructing it on site with the help of directed acyclic deployment graphs. These deployment graphs let the deployment system build the service from smaller parts coming from a distributed repository called VM Warehouse. Analogous to Nimbus it has two main interfaces, one for managing the virtual machine itself (VMPlant) and one for creating a new one (VMShop). Inside the VMPlant the VM Warehouse component operates a cache for virtual machine construction. Meanwhile the VMShop's main task is to estimate the cost of deployment on a given system and with this information it decides which machine should initiate the new VMPlant for the requested service. VMPlants defines a framework for the management of virtual machines that includes techniques for representing VM configurations in a flexible manner, for efficient instantiating of VM clones, and for composition of services to support the synchronous creation of large number of VMs. The performance results encourage the use of this technique for on-demand provisioning of VMs, showing that these flexible execution environments can be dynamically cloned often in less than a minute. As an extra over the virtual workspace service updating a service is simple with the deployment graphs, because the service is updated with only the portion of the code that really needs change.

2.2 SERVICE DEPLOYMENT OVERVIEW

This section gives an overview on the related works in deployment. First, I start with the two overview Tables (2.1a and 2.1b). These tables review the various service deployment solutions according to the taxonomy and the service deployment tasks discussed in the previous chapter (see Sections 1.1 and 1.2.1 in particular).

Infrastructure as a service systems were not developed for service deployment, however it has been demonstrated (e.g. in [63]) that they are also capable for deploying services packaged as virtual appliances. Their *isolation level* is high since they support hardware virtualization. Service repositories are not necessarily part of an IaaS system. For example, Amazon and Eucalyptus support only their own internal *repositories*. In contrary, Nimbus offers a generic and extensible way to download virtual appliances before initiating VMs. Because of the nature of the virtual appliances, IaaS systems do not have a limitation on the supported services (they meet the *universality* criterion). They only require the service to be fitted into a single appliance executable in the IaaS system. IaaS systems currently do not generally support *state transfer* by migrating virtual machines among hosts. As for the deployment tasks, the IaaS systems are not generally targeted for deployment,

Deployment Systems	Isolation	Repository support	Universality	Non Invasiveness	State transfer support
IaaS systems	Virtualised	✓	✓	✓	—
Hot Deployment Service	Container	—	✓	✓	—
HAND	Container	—	✓	✓	—
WSPeers	Container	—	✓	✓	—
Dynagrid	Container	—	✓	—	✓
CDDLM implementations	N/A	—	—	—	—

(a) Requirement based classification according to the taxonomy in Section 1.2.1

Deployment Systems	Selection	Installation	Configure	Activate	Adapt	Deactivate	Update	Decommission
IaaS systems [40, 48, 58]	—	✓	—	✓	—	✓	—	✓
HDS [75]	—	✓	Partial	✓	—	✓	—	✓
HAND [62]	—	✓	Partial	✓	—	✓	—	✓
WSPeers [35]	✓	✓	Partial	✓	—	✓	—	✓
Dynagrid [15]	✓	✓	Partial	✓	—	✓	—	✓
CDDLM [81]	—	✓	✓	✓	✓	✓	—	✓

(b) Deployment task based classification (see Section 1.1 for details)

Table 2.1: Classification of the different deployment solutions

therefore, they cannot support the selection and configuration related tasks. Most IaaS systems allow little configuration options on prepared virtual machines – e.g. they can configure network interfaces and other hardware constraints for them. Configuration is a recent research topic in this field and frequently referred as virtual machine contextualization [41].

IaaS systems allowed the rise of the appliance based deployment. In contrast, container based deployment systems existed before the cloud computing era. In cloud computing terminology, these systems are close to the Platform as a Service (PaaS – [37, 38, 85]) concept: these systems provide a unified software environment for the services building on top of them. PaaS systems are frequently encapsulating a container based deployment system in a virtual appliance, then allow the instantiation of these virtual appliances that can encapsulate user provided services. However, the properties and features of the various container based deployment systems are inherently different, because every system provides its unique platform. Therefore, the following paragraphs analyze the various container-based systems based on the requirements and taxonomy identified in Section 1.2. As a result, the soon identified differences between the container-based deployment systems are highlighted in the comparison Tables 2.1a and 2.1b.

Hot Deployment Service (HDS – [75]) introduces a new service for the Open Grid Services Architecture (OGSA – [30]) based service containers. This new service is called ServiceFactory and it enables the dynamic deployment model on the otherwise static containers. With this extension, their system is capable to deploy and decommission services while the service container is still running. With HDS, the service container needs to be changed (at configuration level) to support the operations for new service registration and class loading. This solution ensures inter-service security by running each service in its own java sandbox in order to make sure they do not have direct access to the running code of other services. This means the isolation level is low, since the execution of a malicious service can interfere with the other one by modification of its underlying files or just simply consuming more CPU or network than the others. The HDS allows the deployment of any service code from any source (no repository support) if the code is received from a trusted party. They use grid security infrastructure to identify the user of the deployment request that usually incorporates the service's code itself. The deployment solution they propose is universal as long as the service can be deployed within their chosen OGSA container – this practically reduces the universality criteria to Java-based services. The ServiceFactory does not depend on the deployed service's behavior therefore this solution is non-invasive. State transfer is not addressed even in case of OGSA compliant services. As for deployment tasks, the HDS supports Web Service

Deployment Descriptor that allows the configuration of the service before activation. HDS handles the service's code as a black box therefore it is not possible to update parts of it.

Highly Available Dynamic Deployment Infrastructure (HAND – [62]) discusses isolation issues occurring while a newly deployed service is activated. HAND extends the Java web service core container of the Globus Toolkit 4 [32] with a thin layer responsible for deployment. Thus, they require the container to be delivered with extra services not available in the Globus distribution. In the original Globus web services core container services are delivered in a gar file encapsulating both the service's code and configuration details. Therefore, the original solution only supports deployment time configuration. For reconfiguration it requires the decommission and redeployment of the service. In contrast, HAND supports several deployment approaches and provides two deployment solutions called HAND-C (container level solution that requires the container to be restarted after a service injection), and HAND-S (a service level solution that leaves all other services unaffected during deployment). Between HAND and HDS the only difference is that HAND supports the latest Web Services Resource Framework (WSRF – [6]) container and offers container and service level deployment. The more advanced service level approach is matching the capabilities of the HDS and therefore fulfils the deployment requirements on the same level.

WSPeer [35] defines a message-oriented interface for service registration, deployment, discovery, and invocation. The authors introduce two implementations; one for regular web services based software and other for P2P Simplified protocol. The deployment in the WSPeer system is accomplished by passing a class file to the WSPeer service that registers this class as a web service, or in a more complicated case, the WSPeer automatically generates a proxy for a given service used by the clients. This second case is used to interface the WSPeer environment with the Triana workflow environment. Because the WSPeer uses a lower level service paradigm than the previously mentioned solutions, this increases the universality of the solution; however it still requires that services should use Java technologies. The P2P technology applied in WSPeer enables the deployment task of selection. With the P2P selection solution, the WSPeer can identify possible deployment targets in localized environment.

Dynagrid [15] offers an all in one solution for deployment that covers all the aspects of WSRF service deployment on some level and even reaches out of the boundaries of deployment. It offers a solution for a unified invocation interface (called ServiceDoor), and with the help of the ServiceDoor, the system even offers service state transfers. Deployments are done with the help of the dynamic service launcher (DSL) components that need to be deployed in all the WSRF containers (similarly to HDS and HAND). With DSL they also get a unified service interface, so all the services are hidden behind the invokeMethod function of the DSL service. This approach implies the most changes in the current service-based systems but offers scheduling, migration and invocation support over the deployment capabilities.

Finally, [10] summarizes the development and standardization activities about Configuration Description Deployment and Lifecycle Management (CDDLM) that is a collection of standard proposals from the Open Grid Forum (OGF) focusing on service deployment and management. The CDDLM API provides management and deployment interfaces and the CDL language [79] offers a generic way to configure the deployed application. According to the final report of the CDDLM working group [81], there are four available implementations (two open source, two closed source). For instance, the implementation called SmartFrog from HP Labs provides a framework to create deployment solutions for specific software components. The other implementations only implement parts of the CDDLM recommendation. The flexibility of the CDDLM is shown by the Softcity implementation that is not even using java. Because there are multiple implementations, the isolation level varies in every one of them, however the CDDLM specifications allow even virtualization-based isolation. The generic nature of the CDL language does not allow *universal* implementations, every service will have its own extensions for the CDL and therefore the deployment solution should be extended for every service it plans to support. In case of CDDLM, the *invasiveness* criterion of deployment is reflected to the deployment service itself, because the deployment system might need to be updated to support the new kinds of services. The CDDLM remains in the scope of the traditional deployment tasks therefore it does not support the *state transfer* of services. *Selection* is not a traditional task for deployment systems, and therefore CDDLM does not support it even though the automation of the entire deployment process is not possible without its support. The *update* task is also problematic because even the deployment system could need changes to support the updated service's new configuration details (result of the reflected invasiveness).

2.3 VIRTUAL APPLIANCE SIZE OPTIMIZATION

Several solutions *optimize virtual appliance distribution* by reducing the appliance size. Based on their input requirements these solutions use two distinct approaches for the optimization procedure. First, the *pre-optimizing* approach requires the appliance developer to provide the application and its known dependencies that should be offered by the appliance. Second, the *post-optimizing* approach uses already existing virtual appliances and optimizes their size by altering and removing their contents.

With *pre-optimizing* algorithms, the dependencies of the user applications are prepared as reusable virtual appliance segments. Appliance developers select from these segments so that they can form the base of their planned service. These algorithms then form the virtual appliance with the selected virtual appliance segments and the service's code itself. This algorithm is used by rBuilder [68] with an extension that supports the creation of custom virtual appliances by building from the application source codes.

The extreme case of pre-optimizing algorithms follows a minimalist pre-optimizing approach that offers optimized virtual appliances with known software environments. To support this approach several OS and reusable application vendors offer the minimalist version of their product packaged together with their just-enough operating system (JeOS – [34]) in virtual appliances [20, 53, 56, 82]. For example, there are several virtual appliances available prepared to host a simple LAMP (Linux, apache, MySQL, php) project. However, this approach requires the appliance developer to manually install its application to a suitable optimized virtual appliance. The advantage of these algorithms is the fast creation of the appliances with the price that the developer has to trust the optimization attempt of the used virtual appliance's vendor. If the appliance is not well optimized, or the vendor offers a generic appliance for all uses then the descendant virtual appliances cannot be optimal without further effort.

Other pre-optimizing algorithms determine dependencies within the virtual appliance by using its source code. Software clone [8] and dependency [70] detection techniques identify all of the required underlying software components by analyzing the sources. Once the dependencies are detected, these algorithms leave only those components that are required for serving the target functionality of the virtual appliance. Optimizing a virtual appliance with these techniques require the source code of all the software encapsulated within the appliance. Thus, they also need to analyze the underlying systems (e.g. the operating system) of the application. Unfortunately, this last requirement renders these techniques unfeasible in most cases.

The most widely used *post-optimizing algorithms* [16, 49, 55] are optimizing the free space in the disk images of the virtual appliance. If virtual appliances are created from previously used software systems then their disk images contain their available free space fragmented throughout the entire image. Before publication, virtual appliances are usually compressed for easier transfer. However, the fragmented free space is harder to compress because it might contain leftover data from previously erased content. Therefore, these kinds of post optimizing algorithms analyze the available free space and first they fill their content with easily compressible data. Next, they offer their users the option to defragment the disk images. As a result, these disk images can be shrunk so they are not only more compressible but they do not even store the free space in the disk image if they are not required. The advantage of this algorithm is that it can operate on any virtual appliances so long it can understand and use its file system.

2.4 VIRTUAL APPLIANCE DISTRIBUTION OPTIMIZATION

Virtual appliances frequently require large storage space, thus, their pre-deployment transfer can be optimized by replicating their content in the various repositories of the service-based system. However, in a highly-dynamic service environment there could be thousands of virtual appliances so their replication is not feasible. To overcome this issue researchers try to minimize the storage requirements of these virtual appliances with the help of *data de-duplication* algorithms, e.g. [73]. De-duplication algorithms came from the field of data mining. These algorithms identify the data entries that represent the same items in the various virtual appliances. As a result, systems applying data de-duplication only store partial virtual appliances and they only recover the appliances if a specific appliance is requested. In the field of disk image distribution, there are several approaches for the identification of the common parts of virtual appliances. These approaches are mainly coming from system administration issues like how to install and maintain almost identical machines and their software configurations or how to efficiently store full system backups.

First, [87] discusses the concepts of the *disk-based de-duplication* storage as a new-generation storage system for enterprise data protection. This article uses de-duplication to remove redundant data portions to compress data into a highly compact form. The newly introduced disk based de-duplication algorithms are analyzed from the perspective of performance. The new algorithms are aimed at providing efficient de-duplication by optimizing the memory usage of the proposed system.

Next, [66] introduces algorithms for partition cloning and *partition repositories*. The article proposes techniques to handle the evolution of software installations and the customization of installed systems independently from the applied operating systems. The partitions can be replicated and transferred to a large number of PCs with the Dolly cloning tool. The various distribution strategies applied during the replication phase are discussed in [65].

Later, [72] introduces the so-called *capsules* to build hierarchies of the virtual appliance images. The article introduced techniques to reduce the amount of data sent over the network while migrating capsules: copy-on-write disks track just the updates to capsule disks, “ballooning” eliminates the unused memory, demand paging fetches only needed blocks, and hashing avoids sending blocks that already exist at the remote end. As a result, efficient capsule migration improves the user mobility and system management.

Afterwards, [57] discusses the *Virtual Cluster Installation System* for providing virtual clusters that scale with the number of applied virtual machines. This approach allows fine-grained virtual machine customization with the help of virtual appliance repositories. The proposed system allows the creation and customization of virtual machines on the fly. To improve scalability the installation system pipelines the data transfers and caches the virtual appliances to save software installation time.

Finally, [24] identifies the main problem of current virtual appliance distribution solutions as the lack of customization. The article proposes a new framework that reduces the provisioning time of customized virtual appliances by staging them on in a repository near the customer. As a result, the new framework calculates an optimal staging schedule, according to network bandwidth, pending reservations, and customer value.

Part II

ACHIEVEMENTS

3 OVERVIEW OF THE ARCHITECTURE

CHAPTER OVERVIEW. This chapter provides the first insight of the architecture proposed in this dissertation. Then offers a short overview of the inner workings of virtual appliance management with the architecture in order to offer a foundation for the later chapters. Finally, the chapter outlines the formal background of the thesis.

3.1 INTRODUCTION

This chapter discusses the different components of my proposed virtual appliance creation architecture – automated virtual appliance creation service (AVS). This architecture contributes to an appliance based, universal and non-invasive deployment system that supports all deployment tasks (outlined in Figure 1.1) on Infrastructure as a Service cloud systems. The deployment tasks supported by the architecture are represented with interface lollipops in Figure 3.1 and presented in the conclusions Chapter in Table 11.1. The proposed components of the architecture are presented in Figure 3.1. These components all target the ultimate goal of this dissertation to minimize the time and cost of virtual appliance based service deployments. This chapter gives a general overview and discusses the relationships between the different components.

Virtual appliances are *extracted* and managed with the help of the *automated virtual appliance creation service (AVS)*. The service's main functionality *extracts* virtual appliances from donor systems maintained by the service developers. The service also supports the following virtual appliance *management* tasks: (*i*) transformation of the appliance between various virtual machine formats, (*ii*) size optimization of virtual appliances (called the *optimization facility* – see Chapter 5 for details) and (*iii*) initial upload of a new virtual appliance to a repository. The general properties and behavior of the AVS are discussed in Chapter 4.

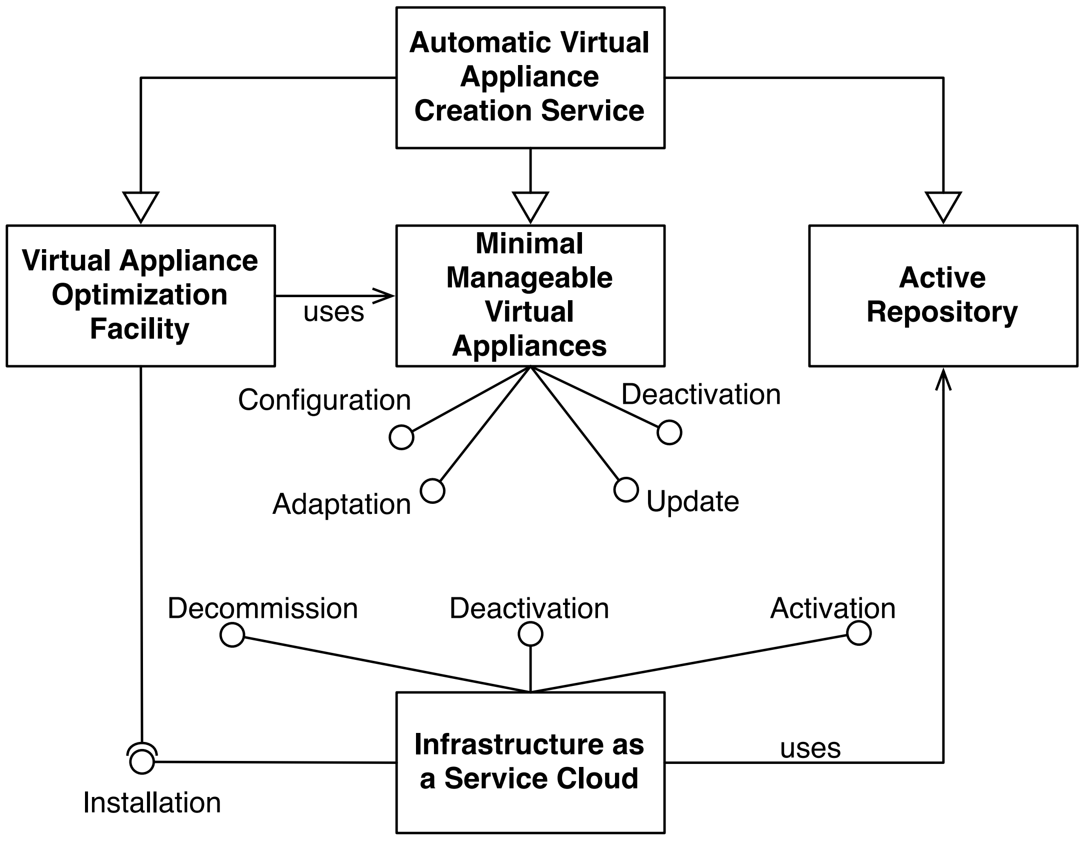

Automatic Virtual Appliance Creation Service
Virtual Appliance Optimization Facility
uses
Minimal Manageable Virtual Appliances
Active Repository
Configuration
Deactivation
Adaptation
Update
Decommission
Deactivation
Activation
Infrastructure as a Service Cloud
uses
Installation

Figure 3.1: Architectural connections of the Automatic Virtual appliance creation Service

In related works, repositories are represented as local file-systems or file servers (e.g. FTP, HTTP). The only task of today's repositories is to safely store and provide access to their entries for authorized parties. Therefore, they act passively and only user actions change their contents. To minimize the download time of repository entries during the *installation task* I propose the extension of these repositories with automated entry management algorithms. In the proposed architecture, virtual appliances are stored in *active repositories* that are defined in Section 6.2. These repositories are active because they optimize the delivery of their contents by (*i*) decomposing the virtual appliances to smaller parts and (*ii*) replicating the commonly used portions of the stored virtual appliances to other repositories.

The last component is the *minimal manageable virtual appliance* (MMVA) that is a special virtual appliance designed to be embedded into the service's appliance in order to allow the management of the service and its software environment. These embedded appliances enable several advanced features of the proposed system: (*i*) the online reconstruction of decomposed or partially available virtual appliances, (*ii*) the reuse of the successfully validated virtual machines during size optimization, (*iii*) reducing the need for appliance type transformations. The details of these features and the definition of the MMVAs are discussed in Chapter 7.

3.2 VIRTUAL APPLIANCE MANAGEMENT

I have identified five basic operations for virtual appliance management: (*i*) *extraction* of the appliances from preinstalled donor systems, (*ii*) *publication* of the extracted appliances to allow their deployments, (*iii*) *optimization* of the appliances for faster delivery on arbitrary IaaS systems (*iv*) *decomposition* of the published appliances to optimize their storage and replicate their highly demanded parts, (*v*) *rebuilding* of the decomposed appliances to allow their faster delivery to the target site before deployment.

Extracting the virtual appliance is the first task in every appliance based deployment system. The automated virtual appliance creation service (AVS) is designed to support the process of extraction and publication. Compared to the frequent deployment requests virtual appliance creation is a rare task, therefore, earlier systems leave it as a manual task. With the help of the AVS, the deployable virtual appliance is automatically extracted from the developer's system. This operation is further discussed in Section 4.2.1.

The AVS also supports the *publication* of the extracted appliances in repositories as it is discussed in Section 4.2.3. After the virtual appliance is extracted, it is stored in a repository to enable further deployments. Repositories accept virtual appliances using different policies, for example they require third party validation of the uploaded content (e.g. user rating) or repository owners manually select the publicly available appliances. If automated deployments occur, target sites decide on allowing deployments by including the acceptance policies of the repositories in their decision, e.g. they only allow the deployment of highly rated appliances.

Virtual appliance delivery is a sub-task of the installation deployment step (see Section 1.1). The delivery time is measured between the time of the initial request to the IaaS system and the time when the entire virtual appliance is available for initiation of the virtual machine for the service. Transferring a virtual appliance requires more time than its configuration and activation (see Table 5.1). The system aims to *optimize the delivery* time by minimizing the size of the virtual appliance in a way that it is still capable of serving its target functionality; however, with a lower footprint. The resulting VA is offered as a single entity that only holds the required service and its support system (the *just enough operating system (JeOS)*). This technique is detailed in Chapter 5.

Virtual appliances are large by nature. Thus, my approach prefers to download them from a high bandwidth and low latency party. Higher bandwidth is achieved by *decomposing* the virtual appliance to smaller portions, then the commonly used parts are spread widely allowing their parallel download and consequently faster delivery time. This decomposition algorithm is detailed in Section 6.2.1.

Finally, initiating a virtual machine is straightforward only if its state – the virtual appliance – is available entirely. However, due to decomposition, virtual appliances are no longer available in a single package. Therefore, they have to be *rebuilt* before they are used to instantiate a virtual machine. This technique is detailed in Section 6.3.

3.3 BASIC SYSTEM DEFINITIONS

The proposed architecture will be applicable on various service-based systems (or infrastructures). The set of these systems are represented with $\Phi := \{\varphi_1, \varphi_2, \dots\}$. Throughout this thesis a particular service-based system ($\varphi \in \Phi$) is represented with its interacting hosts ($\varphi := \{h_1, h_2, \dots\}$). Each one of these hosts ($h_x \in \varphi$) is a networked entity and their relationships can be described with network latency ($l : \varphi^2 \to \mathbb{R}$) and bandwidth ($BW : \varphi^2 \to \mathbb{R}$) between them. The latency and bandwidth functions are the result of measurements taken to evaluate the network relationship properties of the hosts in the current service-based system. The network latency is measured as the time between the first message is sent from the one host ($h_i \in \varphi$) and received at the another ($h_j \in \varphi$). The bandwidth is measured as the maximum amount of data that can be sent between the same hosts in a given time period. If $h_i = h_j$ then these functions are evaluated as follows:

$$l(h,h) := 0 \tag{3.1}$$
$$BW(h,h) := \text{local storage bandwidth} \tag{3.2}$$

Where *local storage bandwidth* represents the bandwidth the actual host can reach its persistent storage. Depending on the host configuration the local storage bandwidth is evaluated differently. For example, hosts with local disk drives will use the bandwidth of their disk I/O subsystem for local storage bandwidth. Alternatively, hosts with networked storage options will use the bandwidth they reach their networked file system.

In case of appliance based service deployment, I have identified five different types of hosts: (i) the IaaS services, (ii) service users, (iii) deployment clients, (iv) repositories, and (v) the deployment hosts. First, *IaaS services* ($C_\varphi := \{c_1, c_2, \dots\}$) offer the management of virtual infrastructures – $\Xi_\varphi : C_\varphi \to \wp(\varphi)$. Where $\Xi_\varphi(c)$ represents the virtualization-enabled hosts in the service-based system – φ – accessible through a particular IaaS service – c.

On these virtualization-enabled hosts the IaaS service can host a fixed set of virtual machines – $vms_\varphi : C_\varphi \to \wp(\varphi)$. However, these virtual machines ($vm \in vms_\varphi(c)$) are executed occasionally, therefore, when they are not running ($online : vms_\varphi(c) \to \{true, false\}$) their network relationship properties are evaluated as follows:

$$\begin{array}{rl} \text{if} & online(vm) = false \\ \text{then} & \forall h \in \varphi : (l(vm,h) = \infty) \\ \text{and} & \forall h \in \varphi : (BW(vm,h) = 0) \end{array}$$

The currently executed virtual machines of a specific virtualization enabled host are defined as: *hosted* : $\Xi_\varphi(c) \to \wp(vms_\varphi(c))$. The host of the IaaS service and its managed infrastructure together forms an IaaS system – $(c \cup \Xi_\varphi(c) \cup vms_\varphi(c)) \subset \varphi$.

The IaaS system hosts various services processing the requests of the *service users* ($U_\varphi := \{u_1, u_2, \dots\}$ where $u_x \in \varphi$ represents the host of a particular service user). If the current set of services cannot fulfil the actual demand then the users or the system can initiate the deployment of new service instances with the help of a *deployment client* – $h_{cli} \in \varphi$. The deployment client looks up the virtual appliance that encapsulates the target service. Then the client requests the IaaS service – c – to instantiate a virtual machine based on a virtual appliance available in a particular *repository* ($R_\varphi := \{r_1, r_2, \dots\}$ where $R_\varphi \subset \varphi$ is the set of all available repositories in the service-based system). Finally, the virtual appliance is instantiated and run on the *deployment host* – $\xi \in \Xi_\varphi(c)$.

Repositories store virtual appliances in one or more packages ($p \in P_\varphi$), where P_φ represents all the available packages in a service-based system. Each repository stores an arbitrary subset of all the available packages that can be described with the *contents* : $R_\varphi \to \wp(P_\varphi)$ function. Consequently, the set of available packages in the service-based system can be defined as:

$$P_\varphi := \bigcup_{\forall r \in R_\varphi} contents(r) \qquad (3.3)$$

Package content is derived from items ($i \in I_\varphi$ where I_φ symbolizes all possible items in the service-based system). The function *items* : $P_\varphi \to \wp(I_\varphi)$ provides the set of items offered by a particular package. In this thesis, items are described with their hash value (*hash* : $I_\varphi \to \mathbb{N}$) and their storage size (*size* : $I_\varphi \to \mathbb{N}$). The hash value of each item has to be unique in the entire service-based system, therefore, this thesis uses $hash(i)$ interchangeably with the item – i – itself when other properties of the item are not important.

The findings of this dissertation are indifferent from the chosen hash value if they fulfill the requirements later discussed in Section 4.4.1.3. Finally, I have defined the hash set $(H : P_\varphi \rightarrow \wp(\mathbb{N}))$ of a particular package as follows:

$$H(p) := \bigcup_{i \in items(p)} hash(i) \tag{3.4}$$

Items are also used to identify relations between packages. Therefore, items can be used to filter package sets to define the set of related $(related : I \times \wp(P_\varphi) \rightarrow \wp(P_\varphi))$ packages as:

$$related(i, P) := \{\forall p \in P : i \in items(p)\} \tag{3.5}$$

I have derived the size of packages $(pkgsize : \wp(P_\varphi) \rightarrow \mathbb{N})$ from the size of their individual items:

$$pkgsize(P_s) := \sum_{p \in P_s} \sum_{i \in items(p)} size(i) \qquad \text{where } P_s \subset P_\varphi \tag{3.6}$$

Package content can be represented in multiple forms depending on the virtual appliance format ($f \in F$ where F depicts the set of supported appliance formats). The items of a package can form virtual appliance images based on the appliance format:

$$im : P_\varphi \times F \rightarrow \mathcal{I}_\varphi$$

Where $\mathcal{I}_\varphi$ represents the possible virtual appliance images in the service-based system. The supported formats of a particular IaaS service are described with $VAforms : C_\varphi \rightarrow \wp(F)$. The supported formats of a particular package are described with $pkgforms : P_\varphi \rightarrow \wp(F)$.

3.3.1 *Package types and relations*

Virtual appliances can be stored in multiple packages because of the active repository functionalities of the architecture (see Section 6.2). This thesis uses the set of direct package dependencies – $dep : P_\varphi \rightarrow \wp(P_\varphi)$ – to specify the related packages. Therefore, $dep(p)$ defines the package alternatives on which p is directly dependent. Consequently, before deployment, p requires the presence of one of the packages from $dep(p)$ in the same virtual machine host. This dissertation represents direct package dependencies as directed edges in its Figures. The edges are directed from the dependent package – p – towards its direct dependency set – $dep(p)$:

$$\forall p_x \in dep(p_1) : \; p_1 \overset{\frown}{\longrightarrow} p_x$$

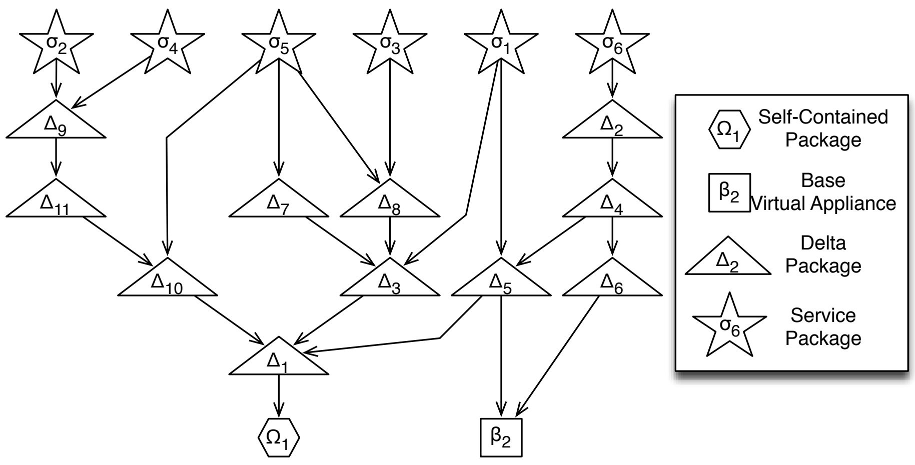


Figure 3.2: Hypothetical dependency graph of 6 services

Figure 3.2 depicts a hypothetical package hierarchy revealing the possible package types in the system and their direct package dependencies.

Direct package dependencies are used to identify possible ways to compose two packages as defined with the following rule:

$$\begin{aligned} p_3 &:= p_1 + p_2 \text{ where } p_2 \in dep(p_1) \\ &\text{therefore} \\ items(p_3) &:= items(p_1) \cup items(p_2) \\ dep(p_3) &:= dep(p_2) \end{aligned} \tag{3.7}$$

In this equation, the dependency package (p_2) is an arbitrary choice from $dep(p_1)$. This equation also reveals that a package (p_3) can be defined by specifying its items ($items(p_3)$) and its direct dependencies ($dep(p_3)$).

The composition rule can be applied repeatedly with the composed packages (shown as p_3 in the previous equation) and their direct package dependencies until the last composed package (p'_3) has no further direct dependencies: $dep(p'_3) = \varnothing$. The dependency packages, selected throughout the repeated composition, describe a possible *rebuilding path* of the package p_1. Figure 3.3 reveals all possible rebuilding paths of package σ_6 (introduced in Figure 3.2).

This dissertation describes a particular rebuilding path with a dependency set that contains all packages required to form a self-contained package based on a specific package. For each rebuilding path I have assigned an identifier (n). As a first step towards the dependency set definition, I have

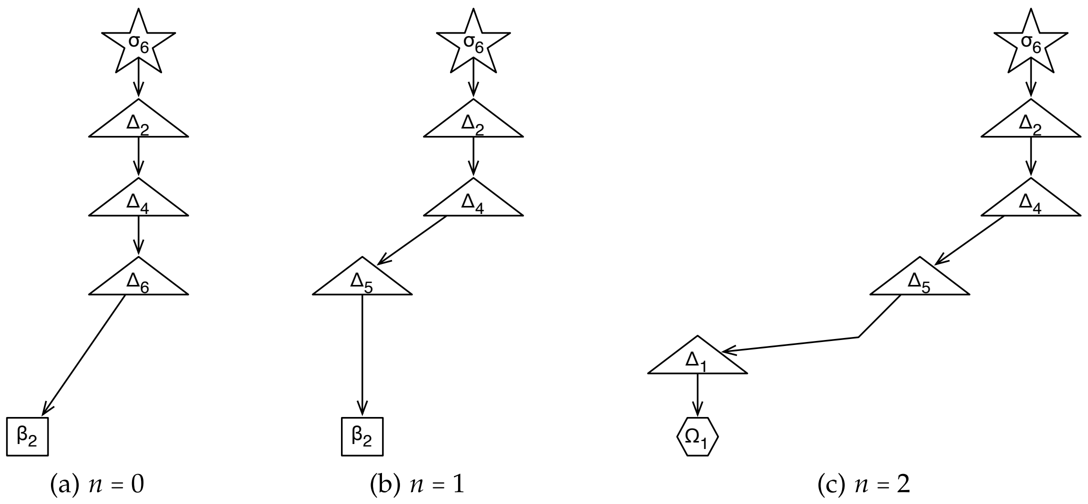


Figure 3.3: Possible rebuilding paths of σ_6 from Figure 3.2

identified the maximal value for the identifier as the possible rebuilding path count for a specific package – $PC : P_\varphi \to \mathbb{N}$:

$$PC(p) := \begin{cases} \sum\limits_{p_x \in dep(p)} PC(p_x) & \text{if } dep(p) \neq \varnothing \\ 1 & \text{otherwise} \end{cases}$$

Then, I have calculated the $PC(p)$ values for several example packages in Figure 3.2:

$$\begin{array}{lclclcl} PC(\beta_2) & = & 1 & PC(\Delta_2) & = & 3 & PC(\Delta_5) = 2 \\ PC(\Delta_{11}) & = & 1 & PC(\sigma_5) & = & 3 & PC(\sigma_6) = 3 \end{array}$$

Next, I have defined the selection rule of the dependency packages based on the rebuilding path identifier ($0 \leq n < PC(p)$). This rule assumes that direct dependency sets are totally ordered (e.g. they can be ordered by the cumulative hash values of their items). Consequently, the function $SD : P_\varphi \times \mathbb{N} \to \mathbb{N}$ returns the index of the selected dependency package:

$$\begin{aligned} SD(p,n) &:= \begin{cases} n & \text{if } |dep(p)| < n \\ i & \text{otherwise} \end{cases} \\ \text{and} & \\ i &:= \min_i \left(\sum_{j=1}^{i-1} PC(p_j) \leq n \right) \end{aligned} \tag{3.8}$$

Where $0 < i \leq |dep(p)|$ and $p_j \in dep(p)$. Accordingly, the $SD(p,n)$ function selects the smallest dependency package identifier – i – that offers the largest cumulative path count still smaller than n. Consequently, I have defined the dependency set ($D : P_\varphi \times \mathbb{N} \rightarrow \wp(P_\varphi)$) using the selection rule as follows:

$$D(p,n) := \begin{cases} p \cup D\big(p_i, n - \sum_{j=1}^{i-1} PC(p_j)\big) & \text{if} \quad dep(p) \neq \varnothing \wedge n \leq PC(p) \\ p & \text{if} \quad dep(p) = \varnothing \end{cases} \tag{3.9}$$

Where $i := SD(p,n)$ and $p_i, p_j \in dep(p)$.

Finally, I have defined the set ($\Theta : P_\varphi \rightarrow \wp(\wp(P_\varphi))$) of all possible dependency sets for a given package in order to allow later operations to select an arbitrary dependency set:

$$\Theta(p) := \{D(p,i) : 0 \leq i < PC(p)\} \tag{3.10}$$

Dependency sets extend the package composition rule as follows:

$$\begin{array}{lrcl} \text{if} & p_3 & := & p_1 + p_2 \text{ where } \exists D(p_1,x) \in \Theta(p_1) : (p_2 \in D(p_1,x)) \\ \text{then} & D(p_2,y) & := & D(p_2,z) \in \Theta(p_2) : \\ & & & \quad \big((D(p_2,z) \cap D(p_1,x)) \backslash D(p_1,x) = \varnothing\big) \\ & D'(p_1,x) & := & D(p_1,x) \backslash D(p_2,y) \\ & items(p_3) & := & \bigcup\limits_{p \in D'(p_1,x)} items(p) \\ & dep(p_3) & := & dep(p_2) \end{array} \tag{3.11}$$

Using the example presented in Figure 3.2, the resolution of the various dependency sets of package σ_6 can be calculated as follows:

$$\Theta(\sigma_6) = \{\underbrace{\{\sigma_6, \Delta_2, \Delta_4, \Delta_6, \beta_2\}}_{D(\sigma_6,0)}, \underbrace{\{\sigma_6, \Delta_2, \Delta_4, \Delta_5, \beta_2\}}_{D(\sigma_6,1)}, \underbrace{\{\sigma_6, \Delta_2, \Delta_4, \Delta_5, \Delta_1, \Omega_1\}}_{D(\sigma_6,2)}\}$$

These dependency sets are also presented in Figure 3.3.

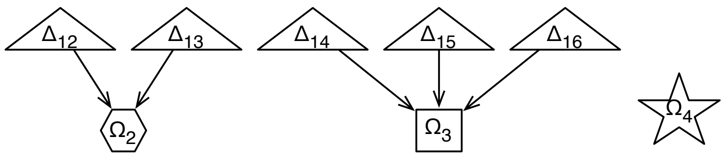
Based on the package *dependency sets* ($D(p,n)$) I have identified the following package types:

SELF-CONTAINED PACKAGES (p_Ω) are those packages that are not dependent on other packages (see Figure 3.4a):

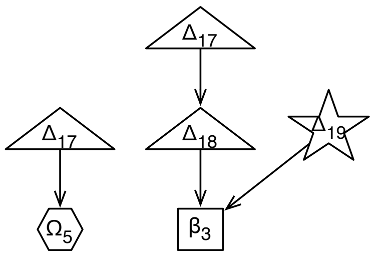
$$selfcontained(p) := \begin{cases} true & \text{if } dep(p) = \varnothing \\ false & \text{otherwise} \end{cases} \tag{3.12}$$

Self-contained packages not necessarily store virtual appliances therefore, their contents have to be evaluated if they can be used to instantiate a virtual machine:

$$va(p_\Omega) := \begin{cases} true & \exists f \in F : (executable(im(p_\Omega, f)) = true) \\ false & \text{otherwise} \end{cases} \tag{3.13}$$



(a) Self-contained packages



(b) Delta packages

Figure 3.4: Basic package relations

Where $executable : \mathcal{I}_\varphi \to \{true, false\}$ evaluates if an appliance image can be used to instantiate a virtual machine. This definition does not mean that the virtual appliances of self-contained packages provide the target services for the users.

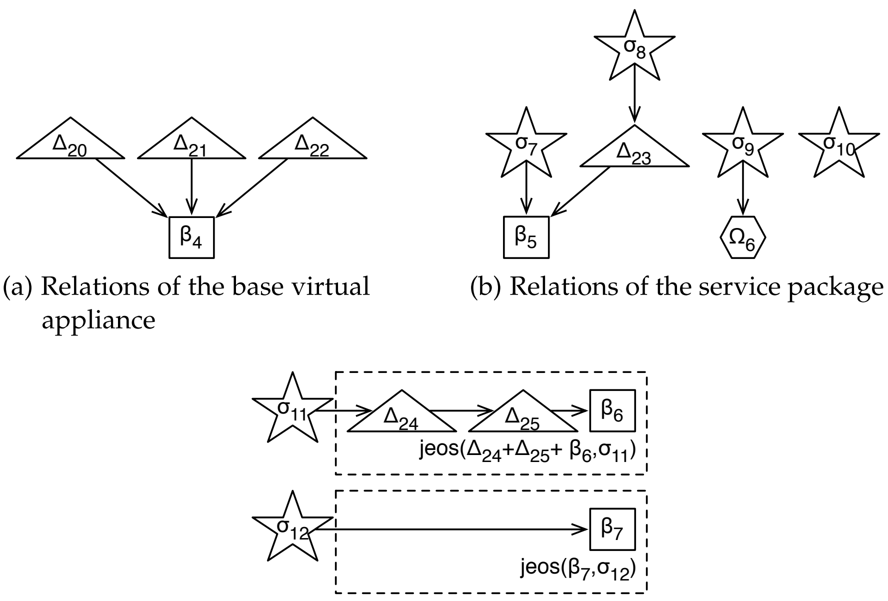
Repositories service self-contained packages immediately without any processing on the package contents. However, these kinds of packages are larger than delta packages introduced below.

DELTA PACKAGES (p_Δ) are dependent on other packages, consequently, they cannot function without deploying their dependencies first – see Figure 3.4b:

$$delta(p) := \begin{cases} true & \text{if } dep(p) \neq \emptyset \\ false & \text{otherwise} \end{cases} \tag{3.14}$$

BASE VIRTUAL APPLIANCES (p_β) are self-contained virtual appliance packages that are used as dependencies by other packages – see Figure 3.5a.

$$baseva(p) := \begin{cases} true & selfcontained(p) = true \\ & \wedge va(p) = true \\ & \wedge \exists p_x \in P_\varphi : \big(p \in dep(p_x)\big) \\ false & \text{otherwise} \end{cases} \tag{3.15}$$



(a) Relations of the base virtual appliance

(b) Relations of the service package

(c) Packages of the just enough operating systems

Figure 3.5: Virtual appliance package relations

If a base virtual appliance is composed with one of its dependent packages ($p_\beta \in dep(p_x)$) then the composite package will maintain the properties of base virtual appliances:

$$\begin{aligned} \text{if } p' &= p_\beta + p_x \\ \text{then} \quad & baseva(p') = true \end{aligned} \tag{3.16}$$

SERVICE PACKAGES (p_σ) provide the target functionality for the users. Other packages do not refer to these packages as their dependencies – see Figure 3.5b:

$$servicepkg(p) := \begin{cases} true & \text{if } \nexists p_x \in P_\varphi \text{ where } p \in dep(p_x) \\ false & \text{otherwise} \end{cases} \tag{3.17}$$

Composing all the dependencies of a service package results in a self-contained virtual appliance that contains all necessary data for the execution of a service incorporated in the appliance:

$$\begin{aligned} \text{if } p' &= \sum_{p_x \in D(p_\sigma, n)} p_x \\ \text{then} \quad & servicepkg(p') = true \wedge va(p') = true \\ & \wedge selfcontained(p') = true \end{aligned} \tag{3.18}$$

Where $0 \leq n < PC(p_\sigma)$ refers to an arbitrary rebuilding path. Two rebuilt service packages are said to be equivalent if they meet with the following requirements:

$$\begin{array}{ll} \text{if} & valid(p_1, initVM(p_2, \varphi)) = true \wedge valid(p_2, initVM(p_1, \varphi)) = true \\ \text{then} & p_1 \equiv p_2 \end{array} \tag{3.19}$$

Where function $initVM : P \times \Phi \to \varphi$ returns with a newly created virtual machine ($vm \in \varphi$) in the service-based system ($\varphi \in \Phi$) that executes the virtual appliance described by package $p \in P_\varphi$. The virtual machine is managed by one of the IaaS services of φ:

$$vm \in \bigcup_{c \in C_\varphi} vms(c)$$

Then, the $valid : P \times \varphi \to \{true, false\}$ evaluates if the execution host (vm) offers the target functionality of a service package p.

Finally, a self-contained service package is optimally sized when there are no items in its image that can be dropped without losing the target functionality for the users:

$$optimalsize(p) := \begin{cases} true & \nexists i \in items(p) \text{ where} \\ & selfcontained(p) = true \\ & \wedge servicepkg(p) = true \\ & \wedge items(p_x) = items(p) \backslash i \\ & \wedge valid(p, initVM(p_x, \varphi)) \\ false & \text{otherwise} \end{cases} \tag{3.20}$$

JUST ENOUGH OPERATING SYSTEM. When composition rule (see Equation 3.16) is applied on decomposed and size optimized virtual appliances, then the largest base virtual appliance, which still does not behave as a service package, is the Just enough Operating System (JeOS – see Figure 3.5c).

$$jeos(p, p_\sigma) := \begin{cases} true & baseva(p) = true \\ & \wedge p \in dep(p_\sigma) \\ & \wedge servicepackage(p_\sigma) = true \\ & \wedge delta(p_\sigma) = true \\ & \wedge optimalsize(\sum_{p_x \in (D(p_\sigma, n))} p_x) = true \\ false & \text{otherwise} \end{cases} \tag{3.21}$$

Where n refers to an arbitrary rebuilding path and p specifies the just enough operating system (or the support environment) of the service package p_σ.

CHAPTER SUMMARY. This chapter revealed the main components of the proposed architecture and shortly discussed their high level connections. Then the chapter progressed through the introduction of the fundamental usage scenarios of the proposed architecture (appliance extraction, publication, optimization, decomposition and rebuilding). The chapter provided the foundation of the later chapters by offering the formal representation of the relevant IaaS and deployment system components and their relations to package delivery and representation. Finally, the chapter identified several types of packages that are later frequently referred and utilized: *self-contained, delta, base virtual appliance, service* and *just enough operating system.*

4 AUTOMATIC VIRTUAL APPLIANCE CREATION SERVICE

CHAPTER OVERVIEW. Only the basic functionality of the automated virtual appliance creation service (AVS) is discussed here in detail. In this chapter, I present the extraction scenarios and algorithms that are initially creating a virtual appliance. I also cover the metadata collected during the extraction process. Later, I summarize the various operations the AVS is capable of executing on the extracted appliances. In this chapter, I detail the appliance transformation and initial upload procedures of the AVS service. The package and delivery optimization subsystems of the service are discussed later (see Chapters 5, 6 and 7).

4.1 INTRODUCTION

The task of the AVS service is to extract a virtual appliance from a virtualized host and package it for later deployments. This package is stored in a virtual appliance (VA) repository. Therefore, the AVS (hosted on $h_{AVS} \in \varphi$) has to interact with the three main actors from the outside world (see Figure 4.1): (i) the client actor (h_{cli}), (ii) the IaaS system actor (via one of the IaaS services – $c \in C_\varphi$) and (iii) the repository actor $(r \in R_\varphi)$.

For the *AVS client actor* the AVS offers an interface that exposes the VA extraction functionality. The AVS has to be installed on the same host as the virtual machine monitor (VMM) to ensure the AVS accesses the virtual machine control functionality of the VMM (see Section 1.1.1). Therefore, on client request, the AVS acquires the state of a virtual machine to form a virtual appliance.

For the *IaaS system actor* the AVS offers two actions. First, the AVS enables the *transformation* of a VMM specific virtual appliance format (that can be used to instantiate a virtual machine on a specific VMM) to a platform independent one (e.g. the open virtual machine format (OVF) specification of the Distributed Management Task Force (DMTF) [52] or the Workspace metadata of virtual workspace service [40]). This functionality is used when the IaaS

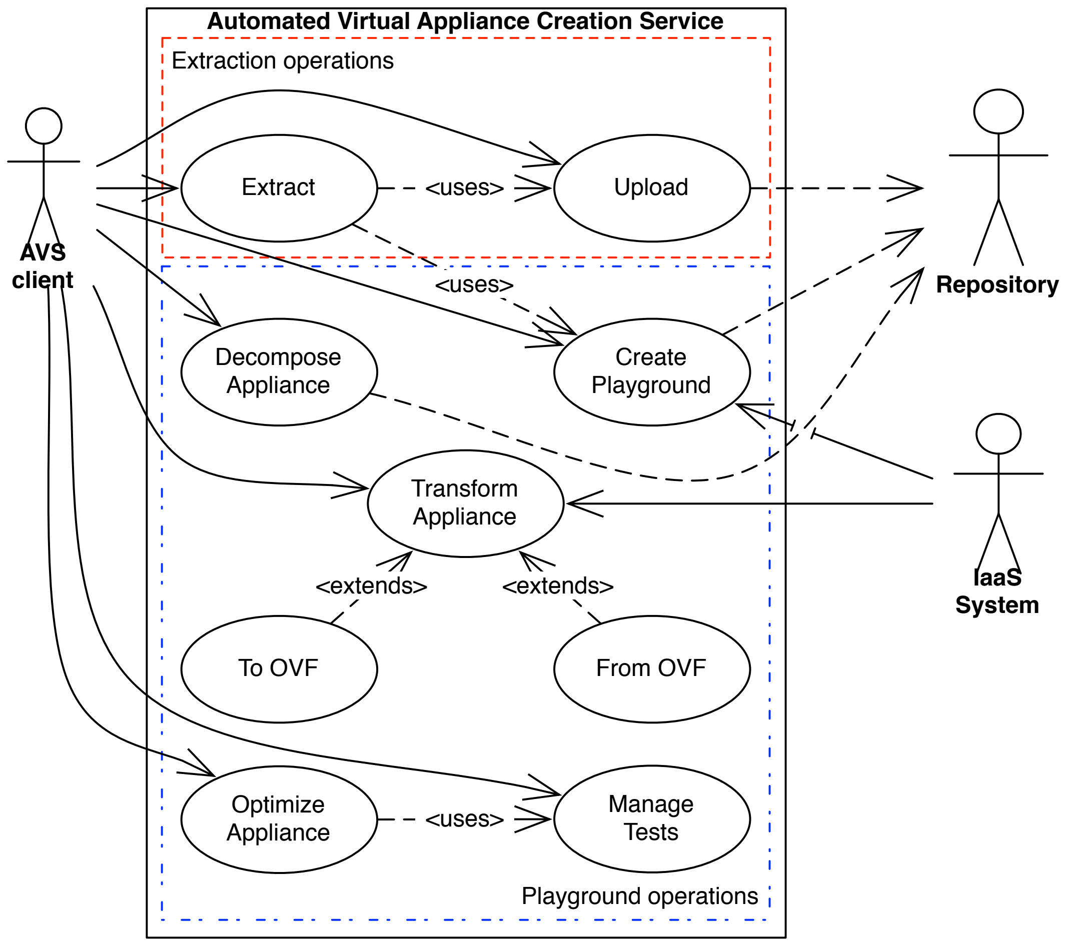


Figure 4.1: Use cases and relations of the AVS subsystem

system receives a request to deploy a virtual appliance that is not supported by the current VMM. The second, *create playground*, operation for the IaaS system is initiated by the AVS itself when it optimizes the virtual appliances with the optimization facility (see Chapter 5).

Finally, the AVS uses a package *repository actor*. The AVS uploads the packaged VA in the repository specified by the appliance developer and then the package replicates it to several other repositories on request. Next, it is capable to search through multiple repositories to find similarities between a VA package and the appliances stored in the given repositories. These similarities form the basis of the decomposition functionality of the architecture detailed in Chapter 6.

4.2 THE AVS CLIENT INTERFACE

The AVS has two basic functionalities (see Figure 4.1). First, it provides an operation to extract and publish an initial virtual appliance from a running system (*Extraction operation*). Second, it provides further operations on just created or previously available (created by third parties) virtual appliances (*Playground operations*). The playground operations either use a previously created virtual appliance or one that the extraction operation has just created.

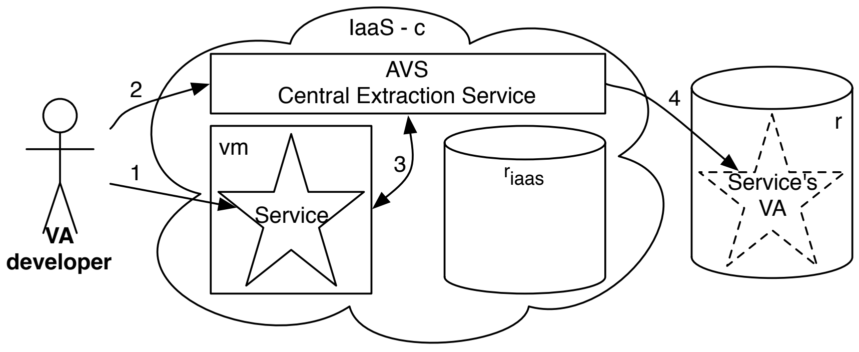
4.2.1 *Virtual Appliance extraction*

The *extract* operation creates the initial virtual appliance by taking a snapshot of the developer specified machine (either virtual or physical). If a virtual machine is specified as the source of the virtual appliance then the extraction operation can create a snapshot of both running and stopped virtual machines. The operation creates the virtual appliances with the file-systems of the specified virtual machine and optionally with the memory state of the running virtual machines.

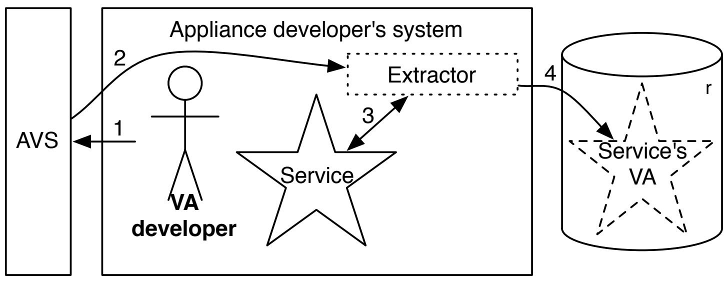
The extract operation supports different virtual machine monitors and virtual machine representations, therefore, it requires the identification of the specific virtual machine to be extracted. If an entire site uses the same virtual machine monitor on all of its nodes then a single AVS service handles the entire site. Therefore, the VM under extraction is identified with a data triplet: (i) the host that runs the VMM ($\xi \in \Xi_{\varphi}(c)$), (ii) the type ($f \in F$) of the VMM the AVS should interface with, and finally, (iii) the virtual machine ($vm \in hosted(\xi)$) identifier in a format specific to the VMM.

In dynamic service environments [23], the caller of the service is not aware if a deployment precedes the service call. Therefore, the startup time of the virtual appliance is critical because it is added to the time of the first service invocation on the newly deployed service instance. Virtual machines are started up differently depending on the content of the previously created virtual appliance: (i) appliances of running VMs are resuming their previous state by loading their system memory from the appliance, (ii) in contrast, appliances of stopped VMs execute the entire boot procedure before the activation of the embedded service in the appliance. The last step of the extraction algorithm temporarily creates two virtual appliances (one with memory state and one without it). Then it measures the startup time of both virtual appliances and only publishes the appliance with faster startup time.

Depending on the location of the execution of the extraction operation, I have defined two scenarios that the AVS has to support (i) the central extrac-



(a) Centrally managed



(b) Decoupled

Figure 4.2: Virtual appliance extraction scenarios

tion service and (*ii*) the decoupled extractor component. These scenarios are detailed in Figure 4.2 and discussed in the following two sections.

4.2.1.1 *Central extraction service*

As depicted in Figure 4.2a, central extraction service provides a unified interface for all virtual appliance developers. In this scenario, first, they request a virtual machine where they can prepare a service instance as the base for the virtual appliance to be published. Next, appliance developers deploy and configure their service (in *step 1*) so it can operate as a standalone application without any further adjustments. If the service is participating in a service composition or the service requires configuration during deployment, then appliance developers have to design configuration algorithms that can adapt the service's deployment with as little information as the IP address of the newly created virtual machine. Finally, the appliance developer has to give the extraction order (see *step 2*) to the AVS that resides in the same IaaS system as the prepared service's virtual machine.

As a result, in *step 3*, the AVS orders the IaaS system to suspend the service's VM and acquires its suspended state from the IaaS system. Alternatively, if the disk image of the VM is stored in the repository of the IaaS system then the appliance developer has the option to offer the link to the image to be used as the initial virtual appliance. Finally, the appliance developer can decide on applying transformation or optimization operations on the initial appliance (see Section 4.2.2) before publishing it in *step 4*.

4.2.1.2 *Decoupled extractor component*

As a drawback of the central extraction service, the appliance developer has to deploy a service instance on a third party's site. To overcome this issue the AVS offers a decoupled extractor component that can be executed on the appliance developer's own system (as seen in Figure 4.2b). This component is capable of uploading the initial virtual appliance to an arbitrary repository or to the AVS's local repository (r_{AVS}) – as a result, it creates a playground for further appliance operations.

If the appliance developer plans to use the decoupled extractor, then before contacting the AVS service, he prepares his own service development system to offer the instance of the service that should be deployed later on. Then, if necessary, the appliance developer prepares the same configurator scripts as he would with the central service. Next, the developer contacts the AVS service for the extractor component in *step 1*. The AVS prepares an extractor component that is a bootable single purpose system capable of connecting to the Internet and uploading the content of the development system's disk images. Before the AVS allows its download (see *step 2*) for the appliance developer, the extractor component is preconfigured to upload the initial virtual appliance to the same AVS service where the extractor was downloaded from. The AVS also provides a list of available repositories for the extractor to enable its multi repository upload functionality. After the extractor component is downloaded from the AVS, the appliance developer executes it. Therefore, it calculates the hash values for all the items on the development system and creates a repository package ready for upload (see *step* 3-4).

4.2.2 *Playground operations*

The AVS provides a local repository ($r_{AVS} \equiv h_{AVS}$ and $r_{AVS} \in R_{\varphi}$) in order to support its operations. Packages are temporarily created in this repository for the duration of the AVS operations. After the AVS finishes its tasks on these packages and publishes them in a third party repository, these packages are automatically removed from the local repository (r_{AVS}). I refer to these temporary packages as the playground for virtual appliances. The different uses of the playground are depicted in Figure 4.3 and discussed in the following paragraphs and subsections.

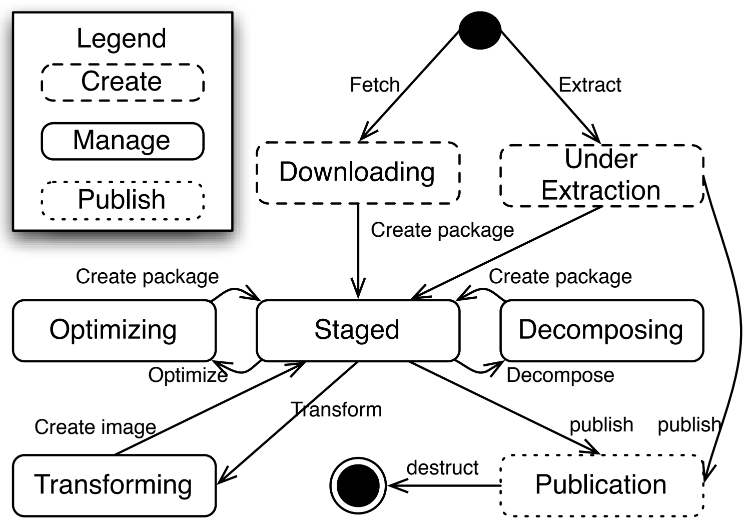


Figure 4.3: Different states of a Virtual Appliance playground in the local AVS repository

At the end of the extraction process, the AVS directly *publish*es the *extract*ed appliance in a repository or alternatively it creates a *playground* for further optimization of the extracted virtual appliance. The playground can be created as a result of the VA *extraction* process ("under extraction" state), or alternatively it can be created from an appliance *downloaded* from a remote repository ("downloading" state). The AVS supports the optimization of already published virtual appliances by this second creation operation. After the playground has been created it is "staged" for the three management operations: (*i*) *optimization*, (*ii*) *decomposition* and (*iii*) *transformation*.

4.2.2.1 *The optimization operation*

The optimization operation (while the playground is in the "optimizing" state, see Figure 4.3) is accomplished in two phases. First, in the test case upload phase, the appliance developer has to add the validation methods for the virtual appliance it plans to optimize, and then the optimization phase can proceed. Figure 4.1 depicts these two phases with the "*manage tests*" and "*optimize appliance*" use cases.

The manage tests use case enables the virtual appliance developer to specify the validation methods and the means of their execution. The AVS provides an extensible interface to support different validation algorithms. E.g. in the current implementation the AVS allows the appliance developer to specify shell scripts or validator virtual appliances. Simple stand-alone test cases are evaluated with shell scripts. However, AVS supports complex test scenarios with validator virtual appliances that can be instantiated while initializing the optimization procedure. This procedure is discussed in detail in Chapter 5.

4.2.2.2 *The decompose operation*

The *decompose* operation (see the "decomposing" state of the playground in Figure 4.3) splits the appliance to smaller parts to allow the delivery optimization of the virtual appliance (this operation is further detailed in Section 6.2.1). Using the search interfaces of the repositories ($r \in R_\varphi$) packages with similar dependency patterns or file contents are looked up. Then, the decomposition operation uses these similarities to form packages of the common parts and allow their wider distribution in order to lower the future deployment times. This operation usually relies heavily on the network; therefore, it is advised to use it when lower network usage is guaranteed.

4.2.2.3 *The transformation operation*

Next, the two *transformation* operations (*fromOVF* and *toOVF*) enable the AVS service to operate on VMM specific appliance representations by applying the Open Virtualization Format (OVF – [52]) as an intermediary. The playground is in "transforming" state (see Figrue 4.3) while executing these operations. The use of these operations are detailed in Section 4.3.

4.2.3 *The upload operation*

Finally, after all operations were executed on the playground of the virtual appliance, the AVS offers the *upload* operation (see the "publication" stage in Figure 4.3). The upload process has two objectives: (*i*) *distribute the contents* of the playground to single or multiple repositories for permanent storage and (*ii*) *optimize the bandwidth* utilized during the upload operation.

Efficient *content distribution* increases the accessibility of the initially uploaded virtual appliance. The AVS handles the upload process to the first selected repository. Further distribution of the created package is a well-discussed topic, and existing replica managers [17, 36] can handle the task. Therefore, the scope of this thesis does not extend in this direction.

As a result, I aim at optimizing the bandwidth utilization of the initial upload process. The AVS optimizes the bandwidth usage during the upload process (*i*) by decreasing the size of the uploaded package (p_{new}) and (*ii*) by selecting the target repository (r_{tg}) with the highest bandwidth. I propose to target these two optimization tasks in parallel by Algorithm 4.1. This algorithm is based on two assumptions: (*a*) all the public repository packages contain the pre-calculated hash values of the items encapsulated in them,

Algorithm 4.1 Initial upload

```
Require: $overhead, minSize$
Require: $p_{new} \in contents(r_{AVS}) : \big(va(p_{new}) = true\big)$
 1: $H(p_{new}) \leftarrow calculateHashes(p_{new})$
 2: $hashes \leftarrow compHashes \leftarrow compress(H(p_{new}))$
 3: if $filesize(compHashes) < minSize$ then
 4:     $hashes \leftarrow createFile(H(p_{new}))$
 5: end if
 6: $maxRepos \leftarrow \frac{overhead \cdot size(\{p_{new}\})}{filesize(hashes)}$
 7: $R_{nom} \leftarrow discoverRepos(maxRepos)$
 8: for all $r \in R_{nom}$ do
 9:     $H(p^*_r) \leftarrow H(p_{new}) \cap \Big(\bigcup_{p \in contents(r)} H(p)\Big)$
10:     $items(p_{diff,r}) \leftarrow \Big\{\forall i \in items(p_{new}) : \big(hash(i) \notin H(p^*_r)\big)\Big\}$
11:     $estTime(r, p_{new}) \leftarrow \sum_{i \in items(p_{diff,r})} \frac{size(i)}{BW(r_{AVS}, r)}$
12: end for
13: $r_{tg} \leftarrow r \in R_{nom} : \big(estTime(r, p_{new}) = \min_{r_x \in R_{nom}} (estTime(r_x, p_{new}))\big)$
14: $dep(p_{diff,r_{tg}}) \leftarrow \{p^*_{r_{tg}}\}$
15: $upload(r_{tg}, p_{diff,r_{tg}})$
```

	Compressed size	
Name	Appliance	Item hashes
SSH	122580kB	328kB
Apache	168932kB	440kB
Gemlca	308692kB	636kB
SSH′	6872kB	4kB
Apache′	12356kB	16kB

Table 4.1: Size comparison of virtual appliances and item hashes

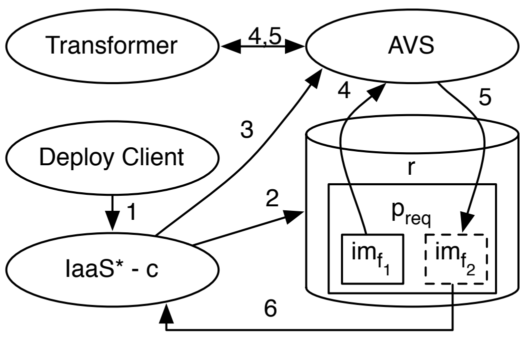
and (b) repositories offer an operation to identify the intersections between a hash set received from a third party repository and the hash values of all their stored content.

The algorithm starts with the calculation of the item hashes ($hash(i) \in H(p_{new})$) of the initial virtual appliance (p_{new}). In order to identify the common contents of the repositories ($r \in R_{\varphi}$) and the initial appliance, the AVS has to upload the hash set ($H(p_{new})$) of the appliance to the repositories for evaluation. The hash set upload operation is utilized to measure the bandwidth ($BW(r_{AVS}, r)$) between the AVS and the repositories. As bandwidth measurement requires larger sized data transfers, the AVS tries to send hash-sets with a minimum size (*"minSize"*). If the hash-set is smaller than *minSize* the AVS sends it uncompressed, otherwise the hash-set is sent out in a compressed form. Smaller values for *minSize* will result more unpredictable bandwidth estimations. Therefore, according to my measurements, *minSize* should be equivalent to the amount of data transferrable in two seconds based on the outgoing bandwidth of the network interface in r_{AVS}. The larger the value the less impact it has on the behavior of the algorithm. Because even uncompressed hashes require small amount of storage compared to the actual size of a package under initial upload.

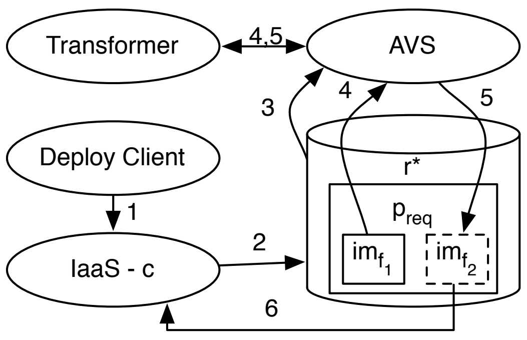
The size of item hashes is negligible compared to the size of the virtual appliance itself (see Table 4.1), therefore, the algorithm can send the item hashes to several *candidate repositories* (R_{nom}) where the AVS plans the initial upload. The algorithm determines the number of candidate repositories ($maxRepos$) based on a system administrator defined maximal *overhead* value. This value is used by the $discoverRepos$ operation that filters the set of available repositories (R_φ) based on the latency between the local repository (r_{AVS}) and the remote ones. Consequently, R_{nom} forms the set of repositories with the lowest latencies.

Next, the initial upload algorithm uploads the hash sets to the candidate repositories. The repositories compare the received hash values to the already stored ones – $\bigcup_{p \in contents(r)}(H(p))$ – and return with the hash set of the hypothetical package (p^*_r). This package is not jet created by the repositories, they only return its hash set. The algorithm uses this hash set to identify those items of the package that are not part of the repositories. For every candidate repository the algorithm calculates the set of items ($items(p_{diff}, r)$) that are not present in the repositories. Then the algorithm estimates the time ($estTime : R_\varphi \times P_\varphi \to \mathbb{N}$) required to upload these unique items to the actual candidate repository.

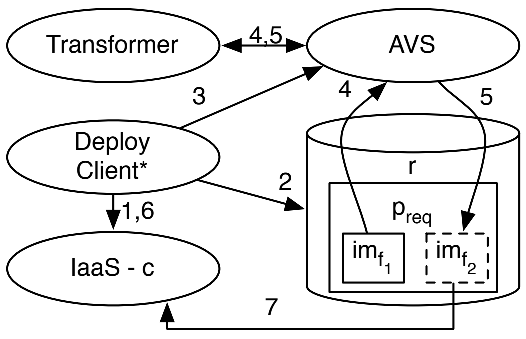
Finally, the AVS determines the target repository (r_{tg}) by minimizing the estimated upload times for the various item sets. Next, the AVS forms the package $p_{diff,r_{tg}}$ with the items not present in the target repository and with a direct package dependency on the hypothetical package $p^*_{r_{tg}}$. This newly formed package is uploaded to the repository r_{tg} instead of p_{new}. When the repository receives the package $p_{diff,r_{tg}}$ then it automatically creates the $p^*_{r_{tg}}$ also.



(a) IaaS initiates



(b) Active repository initiates



(c) Deployment client initiates

Figure 4.4: Options on initiating format transformation of virtual appliances

4.3 THE INFRASTRUCTURE AS A SERVICE SYSTEM INTERFACE

The result of the *extract operation* (see Section 4.2) is a virtual appliance in a format specific to a virtual machine monitor. The AVS service utilizes the Open Virtualization Format (OVF – [52]) as a generic intermediary form between the different VMM specific appliance formats. The transformation between the different virtual appliance formats has been solved by several systems (e.g. VMWare – [1], QEMU – [11]). The AVS uses the functionality of the QEMU allowing the use of the different IaaS systems during deployments.

If a deployment request is made for a package (p_{req}) not supported by the target IaaS system ($pkgforms(p_{req}) \cap VAforms(c) = \varnothing$), then the deployment system should support the transformation of the stored appliance images to the IaaS supported form. In Figure 4.4, I have identified three actors (marked with an asterisk in the figures) that can initiate these transformations with the AVS system: (i) the IaaS system, (ii) the active repository and finally, (iii) the client program of the deployment system.

First, if the IaaS system is capable to initiate the transformation, then the IaaS is allowed to receive requests to deploy non-supported virtual appli-

	IaaS	Repository	Client
User transparency	√	√	√
No IaaS change	–	~	√

Table 4.2: Comparison of basic transformation initiators

ances (see *step 1* in Figure 4.4a). After such a request, the IaaS system first recognizes the transformation need in *step 2*. Next, in *step 3* it queries the AVS service to transform the non-supported appliance to a supported format. As a result the AVS publishes the transformed appliance in the same repository where the non-supported appliance is located (see *steps 4-5*), then notifies the IaaS system to allow its further progression with the deployment in *step 6*.

Second, if the *active repository* is the initiator of the transformation then the IaaS is still allowed to receive deployment requests for non-supported appliances (see *step 1* in Figure 4.4b). However, in this case the IaaS is not aware of the various non-supported appliance formats. Therefore, in *step 2* it queries the repository for p_{req} even though its image im_{f_1} is not supported. During the request the repository determines the supported appliance format (f_2) of the query source (the IaaS system). Then, the repository contacts the AVS service for image transformation in *step 3*. The virtual appliance request is delayed (see *step 6*) until the AVS publishes the transformed appliance (see *steps 4-5*).

Finally, if the *deployment client* is the initiator (see Figure 4.4c), then the IaaS system will not receive deployment requests with non-supported appliances. To avoid using non-supported VAs, the client identifies the format of the requested package and the supported formats by the IaaS system in *steps 1-2*. After the client realizes the need for transformation, the AVS is ordered to transform the requested package in *step 3-5*. When the transformation is completed, the client requests the IaaS to deploy the now supported package storing the new image (see *steps 6-7*).

Table 4.2 highlights the analysis of the different initiators. The major drawback of the IaaS based solution that it has to be implemented for every IaaS system planning to support the AVS architecture. The repository-initiated transformation cannot be applied when repositories are centrally managed by the IaaS systems (e.g. Amazon EC2). Therefore, in these centrally managed repositories the transformation has to be applied before publishing the appliance. Consequently, only the deployment client can enable the use of the transformed images without changing the infrastructure. All three solutions consume the same amount of storage by publishing entire transformed virtual appliances. Therefore, the AVS's typical transformation requestor is the deployment client.

4.4 REPOSITORY INTERFACE AND METADATA COLLECTION

Repositories store arbitrary artifacts with their metadata description. However, throughout this dissertation I only consider storing virtual appliances or parts of the virtual appliances in the repositories. Therefore, from the repository point of view, I define a *virtual appliance* as a storage artifact that encapsulates the disk and the optional memory image of a virtual machine in a VMM specific format. Before storing disk images, the AVS optimizes them for compression by defragmenting them then erasing their unused portions. The appliance is stored as several compressed file-systems and a swap-like area containing the memory state of the virtual machine just before the image was extracted. This subsection defines the metadata the AVS service assigns to the appliances to enable the deployment tasks using the stored appliances.

The *repository package* is defined as an artifact encapsulated with the corresponding metadata. Different repository package formats use different metadata descriptions, however they always include the following items: (*i*) *Dependency description* to specify other packages required to successfully configure and run the virtual appliance packaged together with the description, (*ii*) *Configuration description*, used by the configuration task, specifies the series of configuration and decision steps that result in an executable service, (*iii*) *Human readable description* to support searching and indexing of the software packages within repositories, (*iv*) *Version information* to support exchangeability tests between different repositories.

Since the repositories are not aware of the kind of artifacts they store, the AVS defines the structure of the repository package that will be used for later deployments (see Section 4.4.1). The appliance developer of the AVS service defines the configuration description, the human readable description and version information for the repository package manually. During extraction, the AVS automates the collection of the installation task related information, e.g. (*i*) the first VMM specific image of the virtual appliance and also (*ii*) the item details as a result of the itemization procedure (later introduced in Sections 5.1.2.1 and 5.2.1.1).

In earlier systems, virtual appliances were offered as stand alone packages. Therefore, there was no need to couple them with dependency information. A typical example can be seen in the open virtual machine format (OVF) specification [52] that covers all the relevant metadata mentioned before, except dependencies. However, the AVS automates the dependency related metadata collection during the decomposition operation.

Based on the encapsulated dependency description I have defined two categories of repository packages: (*i*) the *self-contained virtual appliance packages*

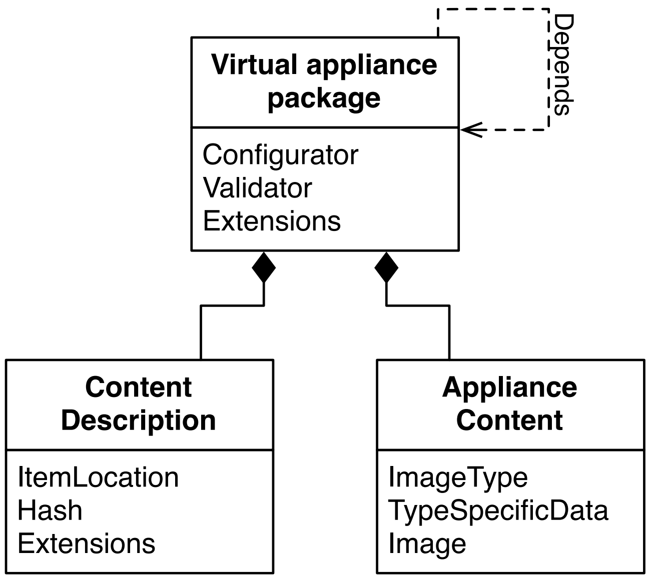


Figure 4.5: The virtual appliance representation of the AVS

and (*ii*) the *delta packages*. A *self-contained virtual appliance package* encapsulates deployment related metadata and a virtual appliance readily usable for immediate VM creation without further modifications. The *delta package* holds parts of a virtual appliance and is dependent either on other delta packages, or on a self-contained package. The decomposition algorithm automatically creates delta packages and identifies their dependencies (detailed in Section 6.2). Virtual appliances are reconstructed from delta packages with the VA rebuilding algorithm that is discussed in Section 6.3.

During the virtual appliance extraction operation, the AVS service collects metadata supporting the *installation* task. The collected metadata defines the VMM, the virtualized hardware and network requirements for the virtual machine hosting the virtual appliance. This metadata is also used by the selection task of the deployment systems. During selection, deployment systems could rank the different IaaS systems based on their support for the requirements of the virtual appliance. Later, the installation task is performed by the IaaS system using the previously collected requirements available in the repository.

4.4.1 *Virtual Appliance Representation*

To allow the efficient storage and delivery of virtual appliances the AVS defines each virtual appliance with the virtual appliance packages defined in Figure 4.5. These packages are capable to support the different deployment tasks and the advanced functionalities of the AVS (e.g. optimization or decomposition). The basic building blocks of these packages are (i) the package specific metadata (*dependencies, configurators and validators*), (ii) the *appliance content*s and (iii) their detailed description (*content description*).

4.4.1.1 *Package Specific Metadata*

The AVS identifies the direct package dependencies ($dep(p)$ see "*Depends*" in Figure 4.5) by the initial upload and decomposition algorithms discussed in Sections 4.2.3 and 6.2.1 respectively. Based on the package types (defined in Section 3.3.1) the AVS handles the package construction blocks differently.

The virtual appliance developer has to define the *configurator* as a component that accomplishes the configuration deployment task (see Section 1.1). In related works [41, 48], configuration is either executed before the instantiation of the virtual appliance (by altering its contents on the host of the VMM) or by runtime contextualization (where the instantiated appliance contacts a specific service that can define the runtime context for the VM, see Section 2.2). As only the first option requires repository side storage thus this paragraph focuses on it exclusively. In order to enable the optimization facility, the modification of the software environment in the deployed services is not allowed for configurators. Without this restriction, the optimization facility should be executed for all possible software environment configurations of the virtual appliance. Configurators are allowed to be defined for all package types. However, during extraction the AVS attaches the configuration to the service package. The configurator of a package can be accessed with the function: $configurator(p)$.

The *validator* component specifies the test cases and algorithms required to utilize all use cases of the target functionality of the virtual appliance. Appliance developers have to define validators for their service packages if they plan to use the optimization facility. The further requirements and uses of the validator component are detailed among the discussion of the optimization facility, see Sections 5.1.2.2 and 5.2.2. The validator component is used during the evaluation of the $valid(p, vm)$ function defined in Section 3.3.1. The validator algorithm of a package can be accessed with the function: $validator(p)$.

4.4.1.2 *Appliance Contents*

Appliance data represents the body of the stored virtual appliance. This data holds the actual contents of the appliance in the *Image* component ($im(p, f)$). The AVS allows multiple appliance data entries for a single appliance package. The *ImageType* ($f \in F$) attribute differentiates between these data entries and allows the AVS to support different VMM specific appliance formats or different IaaS solutions.

In case of self-contained packages, the *ImageType* attribute defines the VMM the *Image* is usable with. However, for delta packages I have defined two special Image types based on the two basic approaches a virtual appliance can be rebuilt before deployment (see Section 6.3).

First, the *offline reconstruction strategy* (see Section 6.3.4) requires the deployment system to understand the *Image* contents, therefore, the AVS also specifies the algorithm the *ApplianceData* can be used for the extension of the partially reconstructed appliance. For example, the *ApplianceData* can specify a *TypeSpecific* so called "algorithm" attribute of the *Image* as *patching*, *merging* or *appending* the *Image* contents to the already existing data.

Second, the delta packages might build on top of a minimal manageable virtual appliance (MMVA – see Chapter 7), therefore, the *online reconstruction strategy* can be applied. In such cases, the used MMVA downloads and installs the *Image* contents on an already running virtual machine. Therefore, the AVS has to identify that the package will be handled externally by the MMVA.

4.4.1.3 *Content description*

Even though a virtual appliance could be stored in different image formats ($|pkgforms(p)| > 1$), after instantiation, the differences between the various formats disappear because they all represent a virtual machine with the same functionality. The AVS service uses the itemization technique discussed in Section 5.2.1.1 to identify the different parts of the virtual appliance independently from the image format. After the *items* ($i \in items(p)$) are identified by their locations (see *ItemLocation* in Figure 4.5) in the appliance, the AVS collects temporal and permanent metadata about them. The temporal metadata collection is applied before the application of the various algorithms introduced later (like size optimization). This metadata is not stored in the repository packages because of their inexpensive collection method and their specific usage.

However, the AVS collects and stores metadata permanently in the repositories if they are utilized in the entire deployment system or when their collection is non-repeatable or too expensive. Currently the AVS only collects item *hash* ($hash(i) \in H(p)$) values for permanent storage to enable their use during dependency detection between different virtual appliances. The AVS is independent from the hashing function used; however, to ensure the compatibility of the different AVS functionalities and services, the same hashing function has to be used throughout the entire service-based system. Since these hash values are used for dependency detection, they have to be calculated with a function with low collision probability [51]. In the current implementations of the AVS service, I have used `SHA1` [28].

These hash values require expensive calculations on the entire content of the *Image* ($im(p, f)$), therefore, they are collected before the initial upload. Even though the calculation lengthens the upload process, it has several advantages e.g. (*i*) repository independence (the system is not bound to a particular hashing algorithm), (*ii*) distributed calculation (the hashes are calculated on the site of extraction) and (*iii*) early availability – the hashes can be used even during the initial upload procedure, and afterwards by the decomposition algorithm of the active repository functionality.

CHAPTER SUMMARY. This chapter first presented the use cases of the AVS architecture. These use cases were discussed from the point of view of the main actors in the system. During the discussion, I first revealed the two approaches for virtual appliance extraction. Then, I revealed that virtual appliances could reside in the AVS repository after extraction via the playground operations: *optimization, decomposition* and *transformation*. Next, the chapter provided the definition and detailed description of the initial upload algorithm that enables appliance developers to reduce their bandwidth usage while publishing their appliances in multiple repositories. Finally, the chapter concluded with the definition of the published virtual appliance packages (including the required metadata to be collected).

5 VIRTUAL APPLIANCE SIZE OPTIMIZATION FACILITY

CHAPTER OVERVIEW. This chapter introduces a technique that can optimize the size of virtual appliances. First, the chapter identifies the download time as most influential part of the various deployment tasks. Then proposes the use of active fault injection to remove contents of the appliance that are suspected to be unnecessary. The chapter discusses the use of various weight functions to identify the contents that are more likely removable. The proposed technique validates reduced appliances with the appliance developer provided validator algorithms. Finally, the chapter reveals a method to parallelize the removal and validation processes in order to decrease the apparent execution time of the virtual appliance size optimization process.

5.1 VIRTUAL APPLIANCE OPTIMIZATION PRINCIPLES

As the first step towards the optimization algorithm, this section starts with the identification of the deployment time reduction options for services encapsulated in virtual appliances. The deployment time is defined as the time between the deployment decision was made and the service was activated on the selected host. Therefore, deployment time is the sum of the execution times of the various deployment tasks required to activate the service. As

Table 5.1: Size dependent virtual appliance start timings

	Appliances		
Operation	SSH	Apache	Optimization party
Download	24.06s	33.68s	Size optimizer
Initialization	0.5s	0.5s	IaaS
Startup	1.69s	1.9s	Developer
Activation	1.12s	0.82s	Developer

it was defined in Section 1.1 these tasks include the installation, the configuration and finally, the activation. Within these tasks, the installation is the most time consuming thus it is subdivided into several subtasks according to Section 1.3. The time required for these tasks and subtasks can be seen in Table 5.1, the different virtual appliances used in the table are discussed in Section 10.1.

It can be observed in Table 5.1 that installation time mainly depends on the download time of the virtual appliance from the repository to the selected host. This time can be optimized in two ways: (*i*) minimizing the size of the virtual appliance while still maintaining its target functionality and (*ii*) storing the virtual appliance in a repository with the smallest latency and largest transfer rate to the selected host. This chapter only aims at the first option because it supports a wider range of IaaS systems; later, Chapter 6 will focus on the second option.

Throughout this chapter, I did not make any assumptions on the IaaS behavior; therefore, this approach is applicable to any IaaS system. This approach modifies the virtual appliance in a way that it is still capable of serving its target functionality, however, with a smaller size. A virtual appliance contains a disk image and if present a memory snapshot. However, the volatile nature of memory snapshots makes them undesirable for optimization. As a result, an optimal virtual appliance holds a disk image of two basic components: the Just-enough Operating System (JeOS) and the service itself. These components are distinguished only in theory. Appliance optimization is done by the VA optimization facility that is a subsystem of the AVS service (previously discussed in Chapter 4).

5.1.1 *The Virtual Appliance Optimization Facility*

Active fault injection uses fault injection approaches [4, 19, 80] that generate hardware and software level faults to test the fault tolerant behavior of the software. However, for virtual appliance optimization, the system does not really test for fault tolerant behavior. Instead, the optimization facility uses fault injection to determine those components of the original appliance ($p_\sigma \in P_\varphi : (servicepkg(p_\sigma) = true \wedge va(p_\sigma) = true)$) that are not needed for the target functionality of the appliance.

Therefore, first, I define the faults that can be injected in order to achieve size reduction. Having a virtualized environment enables the simulation of both software and hardware level faults. Software level faults can be the misbehavior of the file system – e.g. simulating the corruption of the file system by removing a file or some of its portions. Hardware faults can be the misbehavior of the memory or the disk subsystem. Simulating hardware faults require the modification of the virtual machine monitors. Thus, a hardware fault based optimization facility should ship with modified virtual machine monitors for all the supported virtualization platforms. This requirement seriously decreases the adaptability of the optimization algorithm; therefore, this work only considers software fault injection.

After the injection of a software fault the facility creates the temporarily *reduced virtual appliance* (for the conclusive definition of these virtual appliances see Section 5.1.2.2). The optimization facility requires a set of validation algorithms in order to ensure the reduced appliance still offers the target functionality of the original appliance. The optimization facility initiates a virtual machine based on the reduced virtual appliance. Validation algorithms have to evaluate these virtual machines ($validate : \varphi \rightarrow \{true, false\}$), whether they provide the target functionality. As it was defined in Section 4.4.1.1, every virtual appliance is accompanied with its own validation algorithm, therefore this dissertation does not aim to define validation algorithms, it is the task of the appliance developers to do so. Software development may involve unit and integration tests that can be also used as the validation algorithm. However, it is required that these algorithms evaluate the entire functionality of the appliance, therefore they can be used by the $valid(p, vm)$ function as follows:

$$valid(p, vm) := evaluate(validator(p), vm)$$

Where the validator algorithm of package p is evaluated with the argument vm.

During fault injection, the removable parts could have arbitrary granularity (e.g. software packages, files). If a software package manager (e.g. the debian package manager [12] – `dpkg`) is present in the appliance, then the optimization facility utilizes it to remove those packages that are independent from the target functionality of the appliance. Later on, when the facility cannot purge more packages from the VA with the manager, it switches to file level removal. During this level, the facility even drops the software package manager itself if the appliance can be validated without it.

Virtual appliance providers may offer updates to the original appliances being not aware that their appliance has been already optimized. The optimization facility avoids the re-optimization of the entire appliance by identifying the updates and reusing the previous optimization results. The facility takes snapshots of the optimization process that create *intermediate appliances*. When an update has to be propagated to the optimized virtual appliance the facility selects an intermediate appliance that can accept the updates. Therefore, only the snapshots taken after this intermediate appliance are lost because of the update.

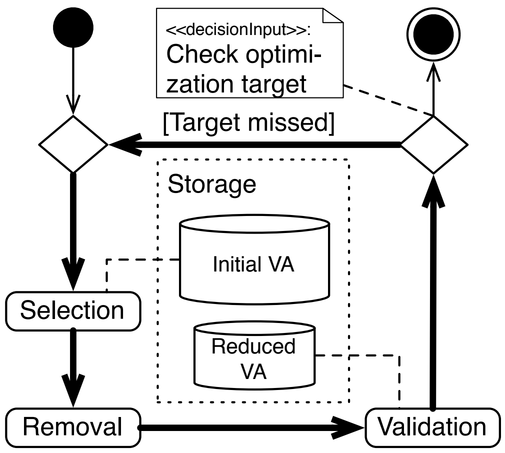


Figure 5.1: Basic appliance optimization technique

5.1.2 *Appliance Contents Removal*

This section describes the basic algorithm of the optimization procedure that is split into three distinct tasks as seen in Figure 5.1. The first task is the *selection* of the virtual appliance's removable parts. The selection task is the most complex and critical task of the optimization approach presented here. This task analyzes the package ($p_\sigma \in P_\varphi$) of the virtual appliance ($va(p_\sigma) = true$) and proposes how to partition the appliance. The proposed partitioning is based on the knowledge base and the relations and properties of the different parts of the appliance. The selection task also assigns a weight value to each part of the appliance. Parts with the highest weights are temporarily dropped by the *removal* task from the appliance. Finally, the third and last task is the *validation* of the modified appliance. With an appliance developer provided test, this last task decides whether the dropped parts should be permanently removed from the VA. Later, I refer to the triplet of selection, removal and validation as the *optimization iteration* (see the bold arrows in Figure 5.1). The algorithm decides on initiating further optimization iterations if the appliance developer provided optimization target constraints have not been achieved after validation. After the final iteration, the algorithm publishes the optimized appliance without the successfully removed parts. These three tasks are further outlined in the next sub-sections.

5.1.2.1 *Selection*

As it was discussed previously, the main task of selection is to identify parts to be removed. From the selection point of view the granularity of the parts in the VA is not important. The only requirement is that the selection algorithm has to use the same kind of VA parts that the removal task uses. Virtual appliance parts can range from single bytes, sectors, file contents, files to even directories or software packages. Later on, the removal task tries to drop all these parts from the virtual appliance. The identification of the different parts and their metadata is called *itemization*. Different itemization algorithms use different kinds of parts as items. *Items* ($i \in items(p_\sigma)$) are the internal representation of the virtual appliance parts with metadata. Items are the smallest entities handled by the selection and removal algorithms. In the implementation the currently used itemization algorithm partitions the virtual appliance on file boundaries.

The second subtask of the selection is *weighting* that prioritizes the different items. There are three basic kinds of items in the appliance: (i) the *core items* – $i_c \in coreitems(p_\sigma)$ –, (ii) the *volatile items* – $i_v \in volatileitems(p_\sigma)$ – and finally, (iii) the *fuzzy items* – $i_f \in fuzzyitems(p_\sigma)$. Those items that the selection algorithm does not have any prior knowledge about are called volatile and weighted regularly. In contrast to volatile items, the core and fuzzy items are identified by the knowledge base of the selection task. The core items are those that cannot be removed from an appliance under any circumstances (e.g. the `init` application in SystemV compatible UNIX systems); these items are predefined in the knowledge base of the selection task. Initially no fuzzy items are defined; they are identified by the algorithm when volatile items are repeatedly validated unsuccessfully. These items are weighted low but are available for removal if the optimization target cannot be reached without trying their removal (weight values assigned for volatile items are always higher than the largest weight for fuzzy items).

Weight functions ($w : I \times P_\varphi \to V$) assign weight values ($V = \{\forall v \in \mathbb{R} : (0 \leq v < 1)\}$) for each item of the virtual appliance under optimization ($i \in items(p_\sigma)$). The higher the weight value the more undesired the actual item is in the virtual appliance. Weight functions are utilizing the various metadata available about the items; therefore, details on the collected metadata and on the used weigh functions are discussed in Section 5.2. However, the optimization facility handles them independently of the utilized metadata.

The optimization facility decides on the use of the various weight functions based on the time and cost constraints specified in the optimization target criterion. The facility can even decide whether to use a single or multiple weight functions throughout the optimization process. If the facility decides on using multiple weight functions then their values are combined as follows: $w_{combined}(i, p) := w_1(i, p) \cdot w_2(i, p)$. Alternatively, different weight functions can be used during the different stages of the optimization process. This strategy ensures that the more expensive and more precise weight calculations are only used if their use would result faster reduction of the appliance size.

The facility also applies different weight functions depending on the kind of the item (core, fuzzy and volatile). Core items (i_c) always evaluated with the constant 0 weight value ($w(i_c) = 0$), therefore the system only has to identify them. These items are either manually defined in the knowledge base or the optimization facility could also apply the technique of runtime *item usage detection*. Fuzzy items (i_f) are defined based on the knowledge base or using the *pattern matching* weight function. Volatile items (i_v) are those items that are subject to active fault injection and their weight functions are discussed in Section 5.2. For any combination of core, fuzzy and volatile items the following statement is always true for their weight function evaluations:

$$0 = w(i_c, p) < w(i_f, p) < w(i_v, p) \tag{5.1}$$

Core items are detected with the technique of *item usage detection*. This technique requires the initiation of the original appliance prior the weight function evaluation – $vm = initVM(p_\sigma, \varphi)$. Then the resulting virtual machine is passed through validation ($valid(p_\sigma, vm)$). During validation, this technique collects the list of items utilized in the virtual machine to accomplish the target functionality of p_σ. Later, the facility uses this item list to specify the core items of the original virtual appliance. However, to detect the used items the virtual machine monitor has to be modified, therefore this detection technique is not used in this dissertation.

Fuzzy items are identified by the optimization facility with *pattern matching* that is an extension of an arbitrary weight function. It requires a set of sample virtual appliances that are *fully optimized* ($P_{sample} := \{\forall p \in P_\varphi : optimalsize(p)\}$), thus no further optimization is possible on them. These samples are either the results of previous optimizations or vendor provided pre-optimized appliances (see Section 2.3). The items found in fully optimized appliances were unsuccessfully removed before, therefore, the algorithm handles them as *fuzzy* items ($I_{fuzzy} := \left\{\forall i \in I_\varphi : \left(\exists p \in P_{sample} : (i \in items(p))\right)\right\}$). The pattern matching weight function uses the inverse appearance frequency ($f_a : I \to \mathbb{R}$) of the items within the fully optimized appliances as the base weight value and adjusts the basic weight provided by the active fault injection technique:

$$f_a(i) := \frac{1}{1 - \frac{|related(i, P_{sample})|}{|P_{sample}|}} \tag{5.2}$$

$$w_P(i, p) := f_a(i) \cdot \min_{j \in I}(w(j, p)) \tag{5.3}$$

The weight calculation with pattern matching is accomplished in two phases. First, before any optimization request could arrive, the optimization facility calculates the inverse appearance frequencies for every fuzzy item ($i \in I_{fuzzy}$). Then during the optimization process it utilizes the pre-calculated $f_a(i)$ values to evaluate the weight function of pattern matching. For the definition of the *related* function see Equation 3.5. Selecting the minimum value of the regular weight function ($w(j,p)$) as the maximum value for the pattern matching ensures that the calculated weight value will classify the item under evaluation as a *fuzzy item* ($i \in fuzzyitems(p_\sigma)$) according to Equation 5.1.

5.1.2.2 *Removal and validation*

The *removal action* sorts the items according to their weights and it removes the item with the highest weight from the virtual appliance. For the removal operation, the optimization facility has to understand the contents of the appliance to be able to remove the selected item. This is achieved with the removal technique – called *pre-execution removal* – that operates on the contents of the virtual appliance while it is not running. The *reduced virtual appliance* (p_{red}) is created ($p_{red} \in contents(r_{AVS})$) before its execution by removing the highest weighted items (i_{hw}) from the original virtual appliance:

$$i_{hw} := i \in items(p_\sigma) : \left(w(i) = \max_{j \in items(p_\sigma)} w(j)\right)$$

$$items(p_{red}) := items(p_\sigma) \setminus \{i_{hw}\}$$

After removal, *validation* instantiates the reduced appliance in a virtual machine (vm) and all the developer-supplied validation algorithms – $validator(p_\sigma)$ – are executed on the service it offers. If any of these algorithms fail, then the validation procedure is non-successful – $valid(p_\sigma, vm) = false$.

Non-successful validation leads to the restoration of the original virtual appliance and the optimization facility starts a new optimization iteration with the exclusion of the items that cannot be removed – $i_{hw} \in coreitems(p_\sigma)$. Successful validation is followed by a new iteration using the reduced appliance ($p_\sigma = p_{red}$). The reduced appliance is created with two approaches: (i) before the next validation starts, the original appliance is reduced with all the successfully validated parts, or (ii) when the cost of the *intermediate appliance* creation (see Section 5.1.1) allows, the reduced appliance is created as a snapshot of the optimization process and later this intermediate appliance is used instead of the original one.

Before starting a new optimization iteration, the facility decides if it has already reached the target that was set through the optimization request by the appliance developer. The optimization target is used to limit the resource usage of the optimization algorithm and leads to a virtual appliance with suboptimal size.

5.2 IMPLEMENTATION OF VIRTUAL APPLIANCE OPTIMIZATION

Before discussing the implementation of the optimization facility, I define the constraints of the optimization facility and system. The *optimization facility* is based on the algorithms I designed and described in this chapter. It is implemented as a web service that resides on a single host within the *optimization system*. The system also includes all the dependencies of the proposed algorithms. Depending on its configuration, the optimization system spreads to multiple hosts ($h_x \in \varphi$) within a single administrative domain (or an IaaS system controlled by c), where these nodes all host a virtual machine monitor for virtual machine management – $h_x \in \Xi_\varphi(c)$.

In the coming parts of this section, first, I present the overall optimization algorithm focusing on the problems raised by the selection task (see Section 5.2.1). Then I discuss the parallelization of the removal and validation steps by utilizing several validator virtual machines simultaneously (see Section 5.2.2). Finally, I present the virtual machine management features of the optimization facility (see Section 5.2.3).

5.2.1 *Implementation of the Item Selection*

Figure 5.2 depicts the entire optimization process, however it gives details of the selection related operations only. The other operations are discussed in the next sections. I have defined the *optimization iteration* as a single step in the loop that starts with *item pooling* task and ends with the *target check* operation.

5.2.1.1 *Virtual Appliance Itemization*

When the optimization facility receives a request for minimizing a virtual appliance (p_σ), it first *fetches the appliance* from the repository. As a result, it mounts the disk image ($im(p_\sigma, f)$) of the virtual appliance on the host of the optimization facility. The mount point is passed to the *file based itemization* algorithm that reads the file system and identifies the items (files in the current implementation) of the virtual appliance. The itemization procedure also collects and passes the following metadata for each file: (a) the *item size* ($size(i)$), (b) *dependencies* representing item relations (e.g. inclusion, parent/child relationships), (c) creation and modification *timestamps*. The metadata that is

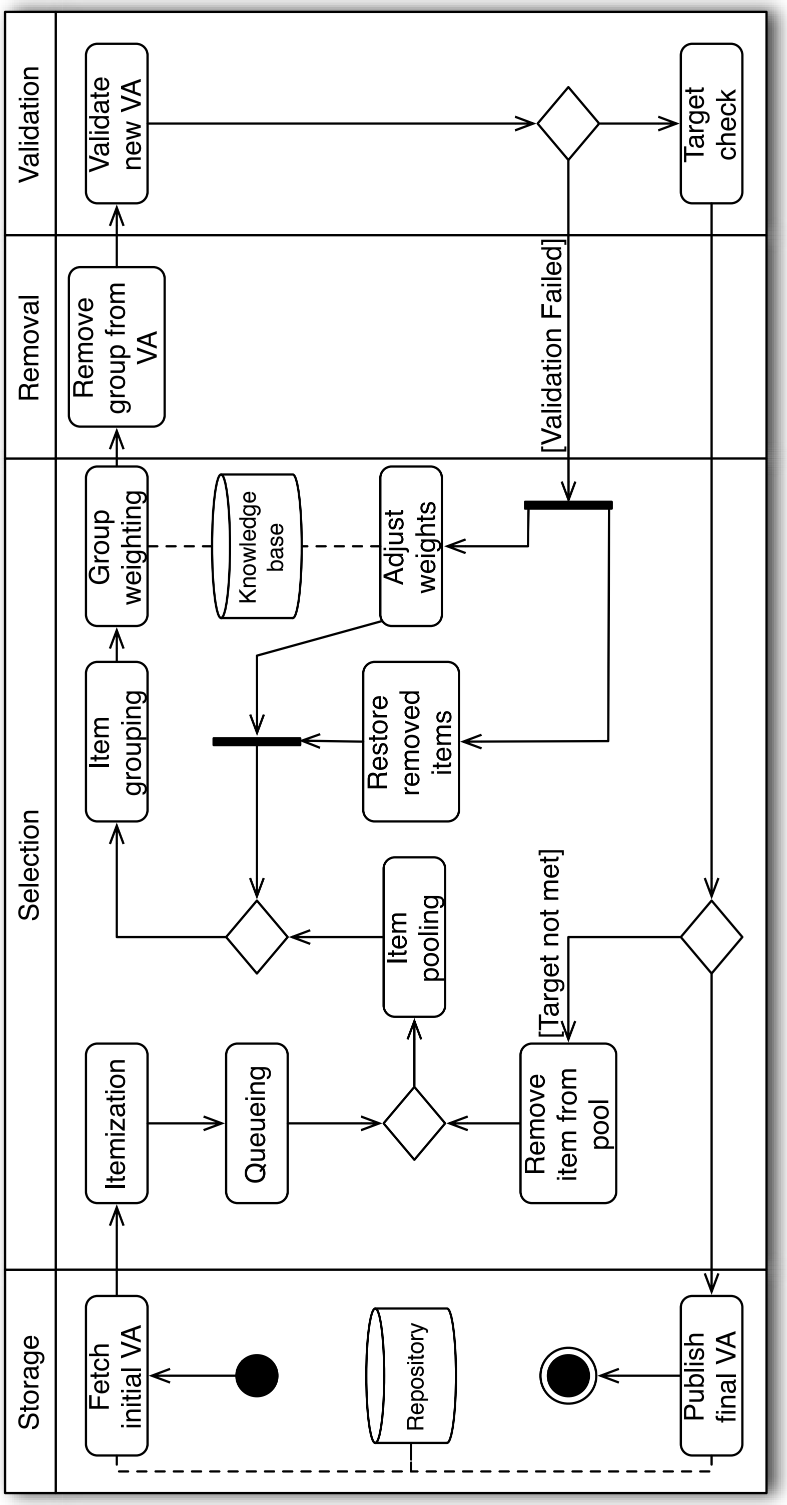


Figure 5.2: Overview of the proposed optimization technique

used by later phases of the algorithm is extensible, however it has to be collected in the itemization phase.

The optimization facility contains an item pool used as a metadata cache that avoids frequent queries on the appliance's file system in the later stages of the optimization algorithm. This *pool* stores metadata for all items that are ready for grouping and validation. The *item queue* is used as an intermediary between the file system and the item pool. The queue is the source to fill the unused item capacity of the pool. The length of this queue is automatically determined by estimating the amount of removable items during a single iteration. The maximum value is the same as the number of virtual machines used for validation as discussed in Section 5.2.3. If the queue is full, then the itemization procedure is blocked until the algorithm removes items from the pool. Therefore, item queue remains full while the itemization processes all parts of the appliance.

5.2.1.2 *Grouping*

The file-based itemization algorithm produces so many items ($|items(p_\sigma)| >> 1000$) from a virtual appliance (see Table 10.1) that creating a virtual machine for each item's removal and validation is not feasible. Therefore, the proposed algorithm groups these items together in order to decrease the number of removal and validation operations. The grouping algorithms have two common tasks: (*i*) form groups from items that are more likely to be removed together and (*ii*) aggregate the metadata attached to the individual items and present them as group metadata.

Item grouping forms groups from the items available in the pool. The algorithm waits until the pool is either full or the itemization has completed. As a result, the grouping algorithm always handles the maximum amount of items. If the removal of a group has failed then the group is split into smaller ones. The previous groupings of an item are also marked in its metadata section allowing the weight functions to adjust their weights on an even more detailed level.

I measure the efficiency of a grouping algorithm with the *grouping failure rate* that I defined as ratio between the number of groups successfully passing validation and the total number of groups that were validated in a specific iteration (see Figure 5.3). For example, when I have optimized the Apache Appliance (see Section A.2 for details) the facility validated 130 groups in total after the first iteration; among these, 26 groups did not pass the validation, therefore the grouping failure rate in this specific case is 20%. I only consider grouping failure rates for a specific algorithm comparable with others, after the number of groupings with that algorithm reaches a predefined threshold.

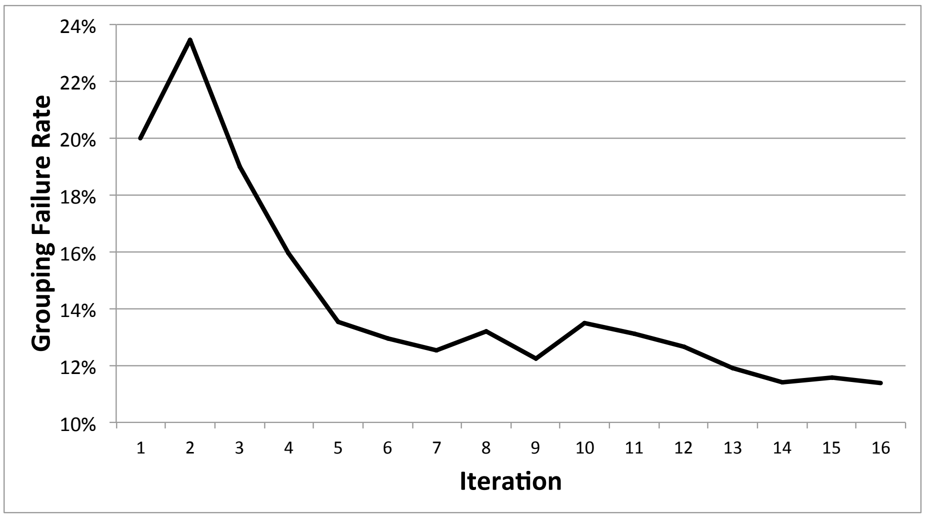


Figure 5.3: Average grouping failure rate of the directory based grouping algorithm during optimization

The first grouping algorithm is *based on the directory structure* in the appliance. Thus, items with the same parent directory are represented in a single group. This grouping algorithm is based on the assumption that files (for a given purpose) are located in the same folder.

The second grouping algorithm is based on *creation time proximity*. This algorithm assumes that files for a given purpose are placed on the file system during the same time period. Thus items within a given Δt time interval belong to a single group. The start of the time interval is selected in order to minimize the number of groups. The value of Δt is either a constant set up prior the execution of the grouping algorithm, or alternatively the system can calculate one for each appliance (e.g. Δt is set to a value that maximizes the sample standard deviation of the resulting group sizes). The implemented algorithm for time proximity uses 5 minutes as Δt.

The current implementation mainly uses the directory structure-based algorithm. However, I also use the creation time proximity based algorithm for appliances with high grouping failure rate (over 30%) for the directory-based algorithm.

Based on my experiments failure rates tend to be high in the early stages of the optimization procedure then sharply decrease afterwards. This can be observed in the grouping failure rate graph in Figure 5.3 that presents the average failure rates in every iteration for all the optimization procedures I have executed to present the findings of this thesis.

5.2.1.3 *Item Weight Calculation*

After grouping, I define *group weighting* that is one of the crucial steps in the selection phase. This step was already introduced in Section 5.1.2.1. Here I only describe the implemented weight calculation algorithm that is a composite of a base weight function and several coefficients:

$$w_A(i,p) = \gamma(i)\kappa(i)w_S(i,p) \tag{5.4}$$

Where $w_A : I \times P_\varphi \to V$ is the composite weight function of volatile items to be used with active fault injection, that is based on $w_S : I \times P_\varphi \to V$ the size based weight function. The facility alters the $w_S(i,p)$ value with the *prior group participation* ($\gamma : I \to V$) coefficient for a given item that prefers items with successfully validated group siblings. The $\kappa : I \to V$ coefficient uses the knowledge base to prefer items with high *removal success rates*.

First, I define the base weight function that assigns weights based only on the item size metadata ($size(i)$). This weight function forces the optimization facility to prefer the items that have larger impact on the overall appliance size:

$$w_S(i,p) = \frac{size(i)}{\max\limits_{j \in items(p)}(size(j))} \tag{5.5}$$

Therefore, without further coefficients the validation progresses from the largest items towards the smaller ones. The advantage of this weight value is that it can be calculated in every stage of the optimization.

The first coefficient uses the information about *prior group participation*. With this coefficient, items that were improperly grouped together affect each other's weight values. Previous groupings of an item are stored in the metadata with all group siblings. If an item already participated in a wrong grouping and some of its previous group siblings were already validated, then I propose the use of their validation success rate to alter the base weight function:

$$\gamma(i) = \begin{cases} M(i) > 0 & \min(1, 1 + \frac{M_{success}(i) - M_{faulty}(i)}{M(i)}) \\ M(i) = 0 & 1 \end{cases} \tag{5.6}$$

Where $M : I \to \mathbb{N}$ defines the number of already validated siblings in a previously faulty grouping where the current item was a member. $M_{faulty} : I \to \mathbb{N}$ specifies the number of those siblings that already failed the validation phase. Consequently $M_{success} : I \to \mathbb{N}$ is the number of successfully validated siblings.

The second coefficient alters the base weight value with previous *removal success rate* of the individual items. As a prerequisite, previous validation results of the items in other virtual appliances are stored in and restored from the knowledge base. With the help of the knowledge base, this coefficient encourages the removal of those items that were previously removed successfully.

$$\kappa(i) = 1 - \alpha(i)\frac{N_F(i)}{N_T(i)} \tag{5.7}$$

Where $\alpha : I \to V$ is the aging coefficient to be discussed in the following paragraph, $N_F : I \to \mathbb{N}$ is the number of unsuccessful removals of a given item, and $N_T : I \to \mathbb{N}$ is the number of trials made on the item.

Finally, the *aging coefficient* prevents the overestimation of the importance of past experiences. For example, in some cases, an important software library receives a replacement and later on, it is placed in software distributions for compatibility reasons. Thus as the virtual appliances evolve the original library loses its fuzzy state. I calculate the aging with the following formula:

$$\alpha(i) = 1 - \frac{t_{now} - t_{TS}(i)}{t_{now}} \tag{5.8}$$

Where t_{now} is the current number of optimizations requested, and $t_{TS} : I \to \mathbb{N}$ is the number of the last optimization iteration this item was tested for removal. These values are also stored in the knowledge base. If $\alpha(i)$ reaches a certain level I remove the stored $N_F(i)$, $N_T(i)$ and $t_{TS}(i)$ values from the knowledge base.

5.2.1.4 *Final Steps of the Optimization Iteration*

The removal and validation phases take place right after the evaluation of the weight functions for all current groups. Figure 5.2 reveals that the steps after validation are independent from the implementation of the removal and validation algorithms, therefore first, I discuss the remaining tasks related to the selection phase in the optimization algorithm. The removal and validation phases are detailed in Section 5.2.2.

In case of validation failure, the previously *removed items are restored* (this step is further detailed in Section 5.2.3). In parallel to the restoration operation, the algorithm *adjusts the weight* of the removed item or group. If the adjustment process receives a group then the algorithm ungroups the items and marks them in order to prevent their entire regrouping. These markings are also used by weight functions to calculate the prior group participation coefficient (γ). If the adjustment process receives an individual item then it saves the metadata of the item as a negative example in the knowledge base. Later, this knowledge base entry helps the calculation of the removal success rate coefficient (κ) during the optimization of other appliances.

Successful validation is followed by a decision whether to continue the optimization process. This task is accomplished by the *target check* action. Appliance developers can specify the completion criterion of the optimization iteration by making an arbitrary conditional expression based on five metrics: (i) the *number of optimization iterations* executed, (ii) the *current size* of the virtual appliance $(pkgsize(\{p_{red}\}))$, (iii) the *size reduction* achieved by the individual optimization iterations – $(pkgsize(\{p_{red}\})/pkgsize(\{p_\sigma\})$ – , (iv) the *wall time* for the entire optimization process and (v) the *size of the remaining* (or not validated) items of the virtual appliance – $remaining : P_\varphi \times P_\varphi \to \mathbb{R}$.

$$remaining(p_{red}, p_\sigma) = \frac{\sum\limits_{i \in (items(p_{red}) \setminus coreitems(p_\sigma))} size(i)}{pkgsize(p_\sigma)} \tag{5.9}$$

If the optimization completion criterion is not met, then the algorithm prepares another optimization iteration. As a result, the validated items or groups of items are removed from the item pool. The success of the removal process is also stored in the knowledge base for all group members and individual items. At the end of this step, the optimization system is ready for the next optimization iteration starting with the item pooling task.

The optimization process concludes with the *publication of the final virtual appliance* when the completion criterion is met or there are no more items to remove from the appliance. During this step, the optimization facility first fetches the original virtual appliance from the repository. Then facility attaches the appliance's image to the host machine allowing the optimization algorithm to remove the successfully validated items from the appliance. Finally, it uploads the locally altered appliance to the repository as an optimized version of the original one.

5.2.2 *Parallel Validation*

Based on Figure 5.4 this section provides a detailed description on how the item removal and validation is executed and parallelized. First, I discuss the item *removal* technique of the current implementation, and then I progress towards parallelization of the proposed technique.

The applied itemization technique determines the kinds of the items that the removal operation should handle, because removal techniques should erase the highest weighted items from the original virtual appliance. This dissertation uses a file-based itemization technique; therefore, the later introduced removal operations should also be capable of removing files from the virtual appliance images.

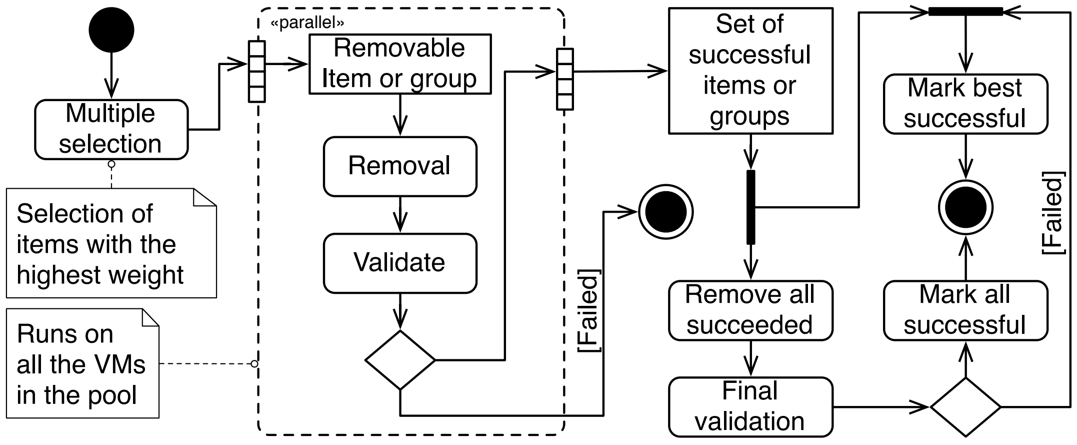


Figure 5.4: Parallelism in the validation process

As it was discussed in Section 5.1.2.2, the proposed optimization algorithm removes the appliance contents with the *pre-execution* removal technique. *Pre-execution* removal attaches the disk images of the original virtual appliance to the optimization facility's host, and removes the files with the highest weight. Then the validation algorithm uses this modified appliance to initiate a virtual machine and check the functionality of the appliance. IaaS systems like Amazon EC2 only initiate virtual machines with virtual appliances stored in their repositories. This requirement forces the pre-execution algorithm to upload the reduced virtual appliance to the repository of the IaaS system before the execution of the validation task. Therefore, I apply this removal technique if the used IaaS can initiate virtual machines from trusted locations like the host of the virtual appliance playground (r_{AVS}).

5.2.2.1 *Parallelism*

Figure 5.5 presents the effects of optimization on the available items and the groups formed from these items. During the individual iterations, the removal action has had more than 400 candidate groups for selection. With the previously presented removal and validation algorithms, each group requires an individual virtual machine for its validation. The most expensive operation in the algorithm is this virtual machine creation step (see Table 10.2 for deployment times of the original virtual machines). Therefore, I introduced the parallel execution of the removal and validation tasks.

As a result, the optimization system is required to be deployed on a cluster that is able to execute several virtual machines in parallel (see the "«parallel»" block in Figure 5.4). The removal and validation tasks are parallelized using these virtual machines. The parallelism in the algorithm starts after assigning the weight values. First, the system selects the highest weighted groups or

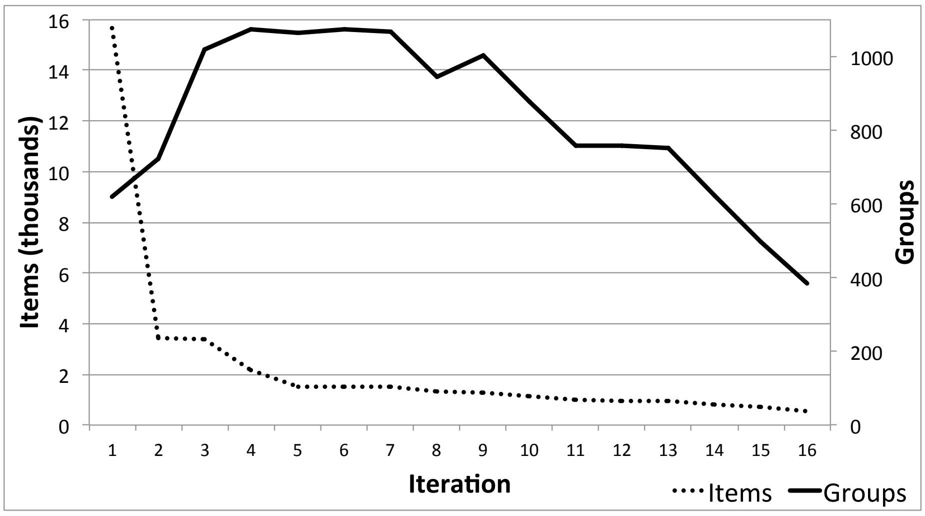


Figure 5.5: Number of groups formed from the available items during the optimization of the Apache appliance (see Sections 8.5.1 and A.2)

items, and for each one of them it initiates a removal and validation task in a dedicated virtual machine (see the "*multiple selection*" action in Figure 5.4). As a result, the facility receives the success reports on several validation tasks in parallel.

However, these validation reports are independent from each other. In order to avoid dependencies between the removed items the proposed algorithm creates a group that contains the union of each successfully removed item. This group is removed from the original appliance and validated again (*final validation* in Figure 5.4). Successful final validation ensures that there are no dependencies between all the previously successful removals. If the final validation is successful, all successful items and groups are removed from the item pool and the appliance.

If the final validation fails, the facility selects the highest weighted successful item or group for permanent removal. Other successfully validated items remain in the item pool with a successful validation marker. This enables their revalidation with the highest weighted element already removed from the appliance.

However, this selection approach leads to a suboptimal solution only. For the optimal selection, the optimization facility should evaluate all possible combinations of the successfully validated removals and mark the combination with the highest cumulative weight. The cost of evaluating the possible combinations renders the optimal selection procedure too expensive and therefore the proposed algorithm never applies it.

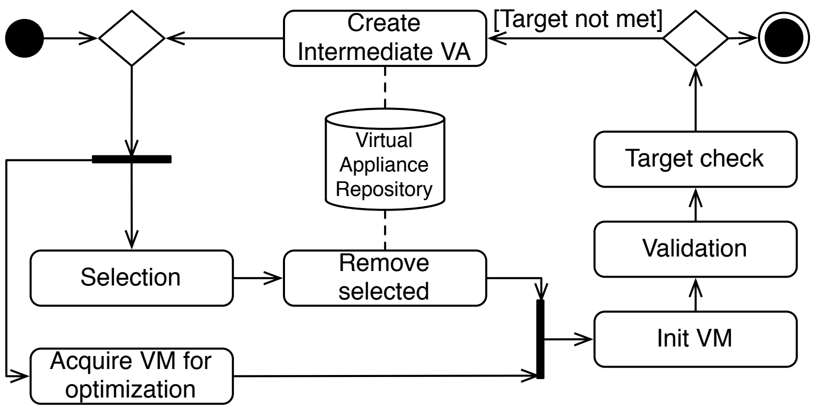


Figure 5.6: Handling virtual machine instances

5.2.3 *Virtual Machine Management Strategy*

The objective of the virtual machine management strategy is to enable the efficient parallel processing of validation tasks. Parallel validation requires multiple virtual machines ready to be validated. These VMs should be all initialized with partially optimized virtual appliances. However, initializing a virtual machine requires substantial amounts of bandwidth and time (see Tables 10.1 and 10.2 for details). Therefore, the proposed management strategy attempts to pre-initialize a pool of virtual machines before the validation process starts. As a result, the system executes the pre-initialization procedure while the optimization facility executes the initial itemization (until the item pool is first filled), grouping and weighting algorithms (see Figure 5.2). The algorithm determines the size of the virtual machine pool by the number of available virtualized CPUs in the optimization system (e.g. $|vms(c)|$). If the optimization system shares the underlying IaaS system with other services then the system administrator of the optimization facility can set up a maximum number and a maximum usage share from the entire system's resources.

Figure 5.6 presents the three main tasks during the management of available virtual machines for the optimization system (acquire VMs, remove contents, init VMs). The first task is to *acquire* as many VMs as possible from the IaaS system for the optimization.

Before the virtual appliances are instantiated, the system creates the temporary virtual appliances for each instantiation. The optimization facility downloads the virtual appliance under optimization from the repository. Subsequently, the system *removes* the item or group that has been selected for the current removal. Finally, the facility offers this temporarily created virtual appliance for the IaaS system to enable the *initiation* of the virtual machine

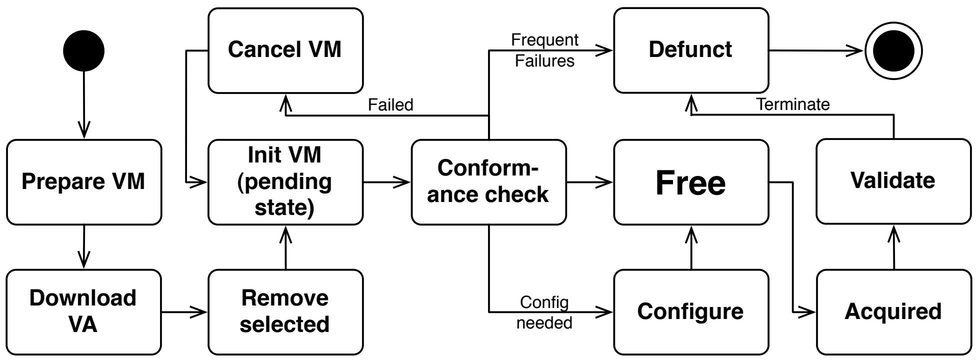


Figure 5.7: Virtual machine management states

for validation. In order to lower the network usage, the facility primarily uses a *virtual appliance playground* (see Section 4.2.2) to offer the temporary virtual appliance. If the IaaS requires virtual appliances to be stored in its own repository, then the optimization facility publishes the temporary virtual appliance in the required repository before the instantiation of a virtual machine.

After the virtual machine is instantiated, the system continues its *initiation* procedure with the configuration task. This step handles internal configuration provided by the appliance developer, then it also manages the external network configuration of the virtual machine (e.g. setting up the firewall). After the VMs are initialized, they are ready to be used for the validation task (detailed in Section 5.2.2).

To allow later validations on the infrastructure, the virtual machine is terminated after its validation task is finished. The parallel branches of the validation algorithm compete with each other for the now available VM slot. To increase the effectiveness of the next optimization iteration the manager *creates an intermediate VA* by removing the successfully validated items or groups from the appliance. As a result, the system reduces the time required for future virtual machine initiations. In other words, the optimization facility utilizes the effects of optimization before reaching the final optimized virtual appliance.

5.2.3.1 *Individual Virtual Machine Handling*

Virtual machines are pooled to support parallel validation, however individual virtual machines are not controlled by the algorithm discussed in the previous Section (see Figure 5.6). The instantiation, management and usage of individual virtual machines progress through different states that are discussed in detail in the following paragraphs.

Figure 5.7 defines the list of virtual machine management states a virtual machine passes through during its lifecycle. Before each validation could start, a new virtual machine has to be created with the virtual appliance according to the current group selection. A pool of virtual machines is created to support the various parallel group selections and removals from the virtual appliance under optimization. If the pool is not full and the current optimization iteration has not been completed, then the system automatically initiates new virtual machines. The first virtual machine management state – *Prepare VM* – acknowledges that the optimization facility started the process of creating a virtual machine for validation.

If the virtual machine pool requires a new VM, then the management algorithm starts with the *download* of the virtual appliance. If the virtual machine reached this state then the optimization facility already received the appliance under optimization from the repository. Next, the facility removes the item or group selected for removal and validation. The VM management state switches to *remove selected* state when the facility has finished the removal operation and already offers the temporary appliance for the IaaS system.

Afterwards, the management strategy requests a virtual machine instantiated with the temporary appliance. Subsequently, the virtual machine passes to the *initialization* state when the IaaS accepts the request. After the IaaS reports the availability of the requested virtual machine, its management state switches to *conformance check*. During this state the virtual machine handler checks the firewall setup of the virtual machine instance. The inspection of the firewall on the running virtual machine is essential to confirm the accessibility of the deployed service. This last inspection step avoids false failure reports on improperly initialized virtual machines when fault injection (item removals) could also cause failures. It ensures the validator will not falsely fail on improper initialization of the service's virtual machine.

If the conformance check fails, the virtual machine handler terminates the unusable virtual machine (*Cancel VM* state), and returns to the virtual machine preparation phase. In rare cases, the IaaS system might have troubles providing an accessible virtual machine that fulfils the minimal requirements of the service in the virtual appliance under optimization. The current virtual machine handler stops returning to the initialization phase after configurable amount of retries.

If the conformance check succeeds, then, if the appliance developer defined configurators ($configurator(p) \neq \emptyset$), the handler *configures* the newly deployed service residing in the virtual machine. Finally, the virtual machine becomes *free* when the virtual machine already runs an activated service usable for the validation processes.

The later states of the virtual machine all represent a state in the validation process. First, the *acquired* state designates the VM's participation in the procedure. This phase binds a validator and a virtual machine. Finally, comes the actual *validation* state. While in this state the optimization facility executes the test cases that validate the slightly modified service instance. After all test cases are executed, the validator reports the results to the facility then terminates the virtual machine. Finally, the virtual machine gets *defunct* and – if necessary – the manager initiates the creation of a new VM replacing the defunct one.

CHAPTER SUMMARY. This chapter first revealed that virtual appliance size heavily affects virtual appliance instantiation time. Next, I suggested that active fault injection techniques could be applied on the virtual appliance size optimization problem. Afterwards, the chapter provided the general overview of the *optimization iteration* (including removable item selection, removal and validation) on an extensible way. I have also revealed the first implementation of the size optimization algorithm utilizing a *pre-execution removal technique*. Later, with the introduction of the minimal manageable virtual appliance concept, the optimization iteration can be further extended by utilizing the technique of *execution during removal* (see Section 7.3.3 for details).

6 PARTIAL VIRTUAL APPLIANCE REPLICATION

CHAPTER OVERVIEW. This chapter focuses on the discussion of my third contribution (distributed virtual appliance storage and delivery). The chapter starts from the assumption that large-scale IaaS systems could store virtual appliances in several repositories, therefore deployment time could heavily vary depending on the connection properties of the repository storing the appliance and the virtualization enabled machine that will host the instantiated appliance. Thus the chapter aims at reducing the variance in deployment time by introducing of the concept of active repositories and appliance rebuilding.

6.1 INTRODUCTION

This chapter aims at revealing my findings about the applicable techniques and strategies to optimize the virtual appliance delivery time from the repositories to the IaaS system. I have achieved the delivery optimization through rearranging and replicating the content of the repositories so that they serve their users more efficiently. During the optimization process, I also aim at maintaining the smallest impact on the storage requirements of the affected repositories.

As a result, I propose that similarly to [48] virtual appliances should not be handled as monolithic entries, but should be decomposed into smaller packages. Contrary to the previously mentioned solution I propose an automatic decomposition algorithm that defines the different building blocks of virtual appliances. Then I introduce several algorithms that increase the storage and delivery efficiency of the newly created packages. These algorithms are dependent on the bandwidth available between the repository and the executor host of the algorithm. Therefore, I present them as an integral part of each repository. Later on, I refer to repositories with these new capabilities as *active repositories*.

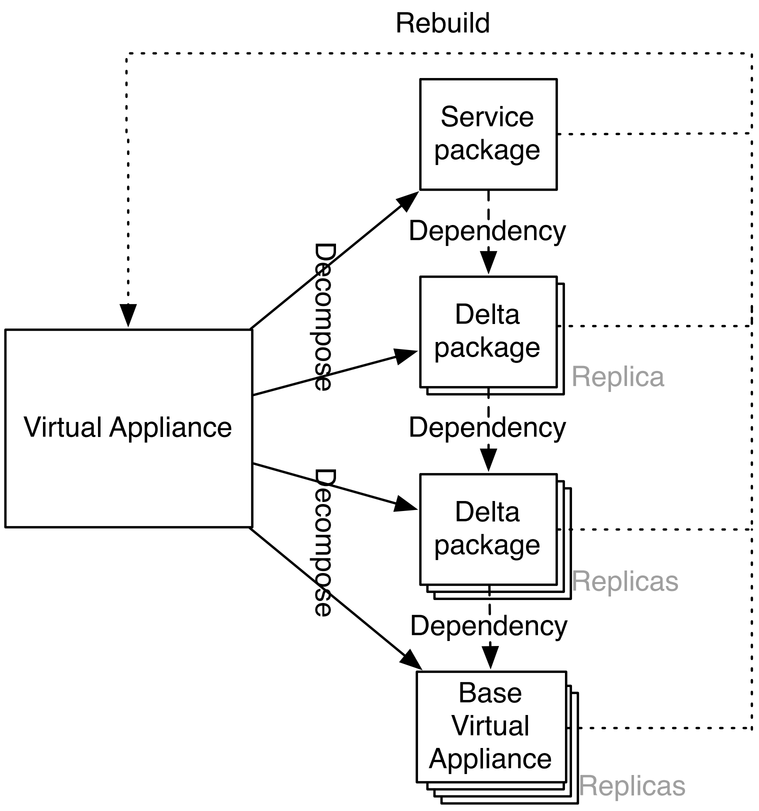


Figure 6.1: The lifecycle of a virtual appliance with the use of active repositories

Consequently, of these algorithms, repositories not only store stand-alone virtual appliances but they might also store some of their fragments. Therefore, I propose an algorithm that rebuilds the fragmented virtual appliances just before they are deployed on an IaaS system.

Finally, as it can be observed in Figure 6.1, the previously mentioned algorithms draw the entire lifecycle of a virtual appliance from the repository point of view. The following sections discuss how the appliance passes between these phases.

6.2 ACTIVE REPOSITORY FUNCTIONALITY

Repositories capable of automatic entry management are active entities in the service-based system because they automatically (*i*) create, (*ii*) merge, (*iii*) destruct and (*iv*) replicate their entries (the self-contained or delta repository packages). New entries are *created* with the proposed decomposition algorithm. Entries downloaded by the same appliance rebuilding process are *merged* to reduce the repository connections required during the rebuilding process. Low usage of the automatically created or merged entities initiates their *destruction*. Finally, frequently used entities are *replicated* to other repositories. All four active repository functionalities are executed as low priority background processes by repositories during low demand periods. The following sections discuss the automation of these management functions.

Active repository functionalities are aimed at minimizing the rebuilding time of the stored packages while minimizing the stored content in the various repositories in the service-based system. The four proposed automated functionalities are not capable to fulfil these aims on their own, however, their composite effect satisfies these aims.

6.2.1 *Package decomposition*

The goal of the decomposition algorithm is the identification of virtual appliance parts that can be effectively replicated between the different repositories. This algorithm is the core of the active repository functionality. The algorithm identifies the dependencies and the virtual appliance parts that can be shared between the different virtual appliances. The proposed technique defines the shared parts in repository packages to be used as building blocks for virtual appliances (for an example see Figure 6.2). Identification of the shared parts – future packages – can be done on several levels of granularity: sector, file and file content. The decomposition algorithm always uses the same level of granularity as the AVS and the optimization facility used for itemizing (see Sections 4.4.1.3 and 5.2.1.1) the virtual appliance during extraction.

The major challenge of this approach is that the virtual appliances are not necessarily available as self-contained packages, therefore, before their deployment they have to be rebuilt. Section 6.3 discusses this rebuilding algorithm, meanwhile Section 6.2.2 reveals a method to support the rebuilding process by merging the unnecessarily split packages prior to their deployment.

Active repositories host and apply the decomposition algorithm themselves. Alternatively, if the modification of a repository's code is not possible then the algorithm is offered as an external component (residing in the AVS). However, the extensive amount of download operations for content analysis requires this component to be installed on a host with high bandwidth connection towards the repository under analysis.

First, the *decomposition algorithm* (see Algorithm 6.1) starts processing right after a new package ($p_{new} \in contents(r)$) is added to the repository ($r \in R_{\varphi}$). Then, based on the various hash sets (e.g. $H_{p_{new}}$) in the repository, the algorithm identifies the package (p_{old}) that shares the most common content with p_{new}. Next, the algorithm determines the hash set of the hypothetical

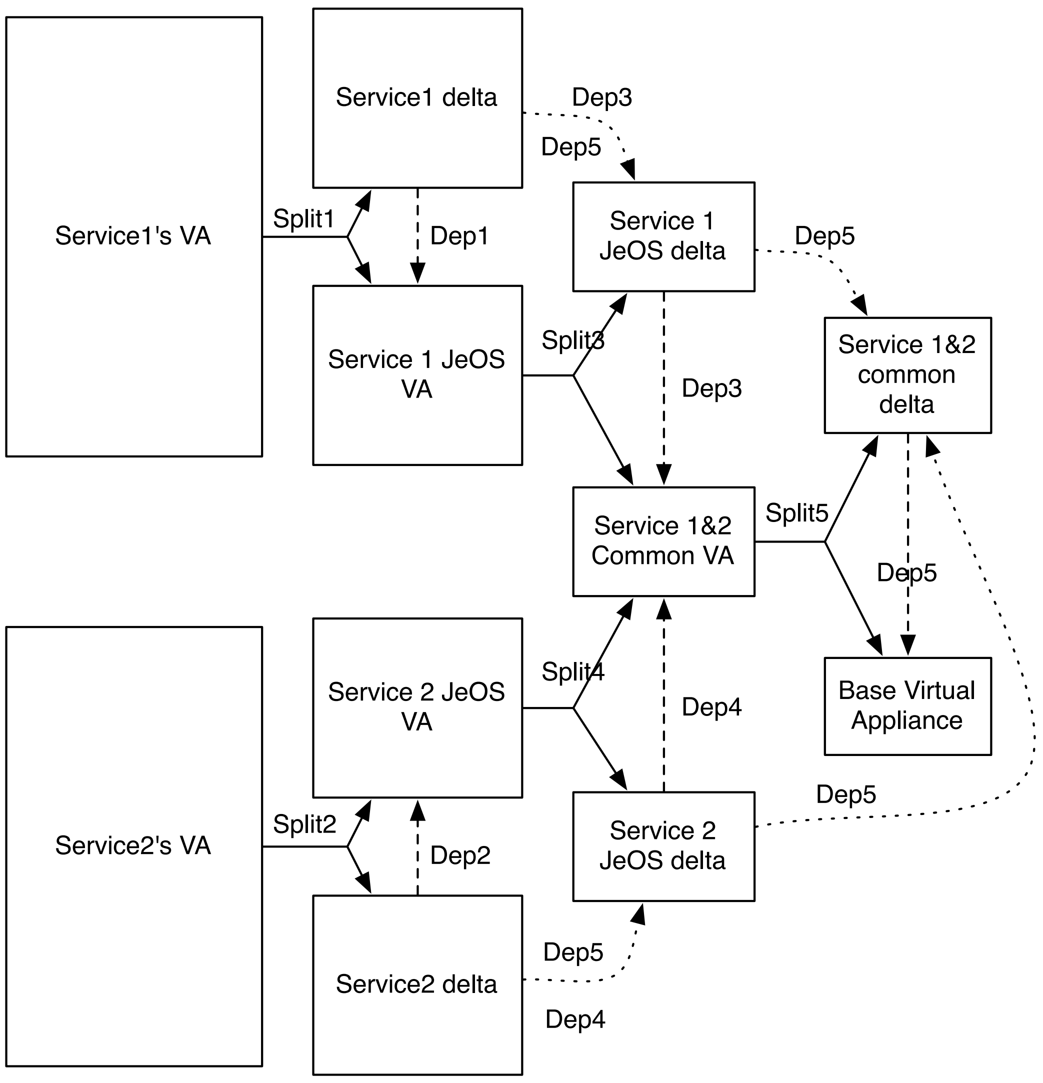


Figure 6.2: Splitting graph of two virtual appliances

package ($H(p^*)$) that stores the common contents between the two packages. If the algorithm could find common items, then it defines the package for storing them as p_{common}. The algorithm ensures that p_{old} shares common roots with p_{new} in *line 5*. Therefore, the direct package dependency set of p_{common} is inherited from p_{old} in *line 13*. Finally, if the package storing the common items can still form a base virtual appliance (see Equation 3.15), then two delta packages ($p_{\Delta,old}$ and $p_{\Delta,new}$) and the common package is published in the repository.

The publication of these three new packages ($p_{common}, p_{\Delta,new}, p_{\Delta,old}$) triggers the execution of the decomposition algorithm on all three of them (e.g. $p_{\Delta,old}$ will behave as p_{new} next time). As a result, when a new virtual appliance is added to the repository, the algorithm will be repeatedly executed

Algorithm 6.1 The proposed decomposition algorithm

```
Require: $r \in R_\varphi$
Require: $p_{new} \in contents(r)$
 1: $H(p_{\Omega,new}) \leftarrow \bigcup_{p \in D(p_{new},1)} H(p)$
 2: $H(p^*) \leftarrow \varnothing$
 3: for all $p_1 \in contents(r) : \big(H(p_1) \cap H(p_{new}) \neq \varnothing\big)$ do
 4:     $H(p_{\Omega,1}) \leftarrow \bigcup_{p \in D(p_1,1)} H(p)$
 5:     $H(p_1^*) \leftarrow H(p_{\Omega,new}) \cap H(p_{\Omega,1})$
 6:     if $(va(p_1^*) = true) \wedge (|H(p_1^*)| > |H(p^*)|)$ then
 7:         $H(p^*) \leftarrow H(p_1^*)$
 8:         $p_{old} \leftarrow p \in P_\varphi : \Big(\nexists p_x \in P_\varphi : \big(p \in dep(p_x) \wedge (H(p_x) \cap H(p^*)) \neq \varnothing\big)\Big)$
 9:     end if
10: end for
11: if $H(p^*) \neq \varnothing$ then
12:     $items(p_{common}) \leftarrow \big\{\forall i \in items(p_{new}) : \big(hash(i) \in H(p^*)\big)\big\}$
13:     $dep(p_{common}) \leftarrow dep(p_{old})$
14:     $p_{\Omega,common} = \sum_{p \in D(p_{common},1)} p$
15:     if $baseva(p_{\Omega,common}) = true$ then
16:         $items(p_{\Delta,new}) \leftarrow items(p_{new}) \backslash items(p_{common})$
17:         $copyPackageSpecificMetadata(p_{new}, p_{\Delta,new})$
18:         $items(p_{\Delta,old}) \leftarrow items(p_{old}) \backslash items(p_{common})$
19:         $copyPackageSpecificMetadata(p_{old}, p_{\Delta,old})$
20:         $dep(p_{\Delta,new}) \leftarrow dep(p_{\Delta,old}) \leftarrow \{p_{common}\}$
21:         $publishPackages(r, p_{common}, p_{\Delta,new}, p_{\Delta,old})$
22:     end if
23: end if
```

until no more common items are found between the already existing repository packages and the newly created packages. The algorithm also stops creating new packages when a common package cannot be used to initiate a virtual appliance. This last condition is applied to enable the online reconstruction strategy introduced in Section 7.3.1.2.

Finally, I present a simple example using Figure 6.2 to demonstrate the behavior of the algorithm. The demonstration starts with a repository that already stores packages "Service 2 JeOS VA" (p_{old}) and "Service 2 delta". Next, the package "Service 1 JeOS VA" (p_{new}) is added to the repository. As a result, the repository applies the decomposition algorithm on this newly added package. Consequently, the algorithm identifies the common portions of the two VA packages and creates it as p_{common} or "Service 1&2 Common

VA" (see "split 3-4" in Figure 6.2). Lastly, the algorithm creates the delta packages from the previous virtual appliance packages: "Service 1-2 JeOS delta".

6.2.2 *Package merging*

This section details the proposed *package merging* algorithm that decreases the virtual appliance rebuilding time by offering multiple repository packages in a single merged package. The architecture also applies merging to conserve repository storage space. Consequently, the algorithm finds the previously unnecessarily decomposed repository packages and merges them. Unnecessary decomposition is recognized when the decomposed packages are barely used individually.

The proposed merging algorithm aims to maximize the size of the merged packages. Before the system proceeds to package merging, it collects the necessary information for the merging decisions. Active repositories log their download queries and describe them with the following five parameters: (i) the source host of the query – $h_{s,q_i} \in \varphi$ –, (ii) the requested repository – $r_{req,q_i} \in R_\varphi$ –, (iii) the requested package – $p_{req,q_i} \in contents(r_{req,q_i})$ –, (iv) the start time of the download query processing – $t_{s,q_i} \in \mathbb{N}$ –, and finally, (v) the finishing time of the download query – $t_{f,q_i} \in \mathbb{N}$. A single query is denoted with the five parameters enclosed in brackets: $q_i := (h_{s,q_i}, r_{req,q_i}, p_{req,q_i}, t_{s,q_i}, t_{f,q_i})$. Consequently, parameters refer to their queries through their subscripts (e.g. h_{s,q_i} is the source host of query q_i). Therefore, the set of queries ($Q_\varphi := \{q_1, q_2, \dots\}$) contain all queries occurred in the service-based system (φ) up to the present. The query set enables the calculation of various derivative information used by the different techniques and algorithms introduced later in this section. In this subsection, the set of Q_φ is used to identify correlated package downloads from a given repository.

To allow determining correlations, I have identified the function $used : R_\varphi \times \wp(contents(r)) \to \wp(contents(r))$. This function defines the first derivative information of the query set used by the algorithms introduced later. This function lists all the packages from package set ($P \subset contents(r)$) that were downloaded from the repository by any user:

$$used(r, P) := \left\{\forall p \in P : \left(\exists q_i \in Q_\varphi : \left(\exists u \in U_\varphi : (h_{s,q_i} \in U_\varphi \wedge p_{req,q_i} \in P)\right)\right)\right\} \quad (6.1)$$

Next, I have defined Algorithm 6.2 to find the group of packages that are downloaded from a given repository in a specific sequence by multiple users. In lines 1-8 of the Algorithm 6.2, all those correlated package sets ($corrgroup(GU, p, r, n) \subset D(p, n)$ – shortened as "CG" in the algorithm description) are identified that include packages downloaded by multiple

Algorithm 6.2 Algorithm to find package correlations

```
Require: $r \in R_\varphi$
Require: $T_{corr} \in \mathbb{N}$
Require: $p \in contents(r)$
Ensure: $correlated(p,r) \subset P_\varphi$
 1: $getters(p,r) \leftarrow \{\forall u \in U_\varphi : (\exists q_i \in Q_\varphi : (h_{s,q_i} \in U_\varphi))\}$
 2: $getgroups(p,r) \leftarrow \{\forall GU \in \wp(getters(p,r)) : (|GU| \geq 2)\}$
 3: $t_{DL,o}(p,r) \leftarrow \infty$
 4: $correlated(p,r) \leftarrow \varnothing$
 5: for $n = 0$ to $PC(p)$ do
 6:     for all $GU \in getgroups(p,r)$ do
 7:         $corrgroup(GU,p,r,n) \leftarrow \{\forall p_x \in used(r, D(p,n)) :$
                $(\forall u \in GU : (|t_s(u,r,p) - t_s(u,r,p_x)| < T_{corr}))\}$
 8:         $CG \leftarrow corrgroup(GU,p,r,n)$
 9:         $p_s(GU,p,r,n) \leftarrow p_x \in CG : (\forall p_y \in CG :$
                $(\nexists u_1, u_2 \in GU : (t_s(u_1,r,p_x) > t_s(u_2,r,p_y))))$
10:         $p_b(GU,p,r,n) \leftarrow p_x \in CG : (\forall p_y \in CG :$
                $(\nexists u_1, u_2 \in GU : (t_f(u_1,r,p_x) < t_f(u_2,r,p_y))))$
11:         $t_{DL}(GU,p,r,n) \leftarrow \sum_{u \in GU} \frac{t_f(r,p_b(GU,p,r,n),u) - t_s(r,p_s(GU,p,r,n),u)}{|GU|}$
12:     end for
13:     $t_{DL,m}(p,r,n) \leftarrow \min_{GU_x \in getgroups(p,r)} t_{DL}(GU_x,p,r,n)$
14:     if $t_{DL,m}(p,r,n) < t_{DL,o}(p,r)$ then
15:         $t_{DL,o(p,r)} \leftarrow t_{DL,m(p,r,n)}$
16:         $corrGU(p,r,n) \leftarrow GU \in getgroups(p,r) :$
                $(t_{DL}(GU,p,r,n) = t_{DL,m}(p,r,n))$
17:         $correlated(p,r) \leftarrow corrgroup(corrGU(p,r),p,r,n)$
18:     end if
19: end for
```

users ($GU \subset U_\varphi$) in a particular sequence within a predefined time interval ($T_{corr} \in \mathbb{N}$). The time interval T_{corr}, in which the download sequences are looked for, is set up by the repository administrator depending on the average download speed of the typical user of its managed repository. In order to simplify the algorithm description, I have assumed that the triplet of $(h_{s,q_i}, r_{req,q_i}, p_{req,q_i})$ uniquely identifies a single download. Therefore, I intro-

Algorithm 6.3 The proposed merging algorithm

```
Require: $r \in R_\varphi$
 1: for all $p \in used(r, contents(r))$ do
 2:     $CPR \leftarrow correlated(p, r)$
 3:     if $CPR \neq \emptyset$ then
 4:         $p_{merged} \leftarrow \sum_{p_x \in CPR} p_x$
 5:         $publishPackages(r, p_{merged})$
 6:         $nodeps(p, r) \leftarrow \left\{\forall p_x \in CPR : \left(\nexists p_y \in CPR : \left(p_x \in dep(p_y)\right)\right)\right\}$
 7:         for all $p_x \in P_\varphi : \left(\left(dep(p_x) \cap nodeps(p, r)\right) \neq \emptyset\right)$ do
 8:             $dep(p_x) \leftarrow dep(p_x) \cup \{p_{merged}\}$
 9:         end for
10:     end if
11: end for
```

duced the $t_s : \varphi \times R_\varphi \times P_\varphi \to \mathbb{N}$ and $t_f : \varphi \times R_\varphi \times P_\varphi \to \mathbb{N}$ functions to specify the download timings for the triplet:

$$t_s(h_{s,q_i}, r_{req,q_i}, p_{req,q_i}) := t_{s,q_i} \qquad t_f(h_{s,q_i}, r_{req,q_i}, p_{req,q_i}) := t_{f,q_i}$$

Then in its second part, the Algorithm 6.2 calculates the average download time interval ($t_{DL} \in \mathbb{N}$) for each correlated package set. The calculation is based on the package with the earliest download start time ($p_s \in corrgroup(GU, p, r, n)$) and on the package with the latest download finish time ($p_b \in corrgroup(GU, p, r, n)$). Next, I define the final correlated package set as $correlated(p, r)$ that represents the correlated package set with the minimal download time ($t_{DL,o}$) within all possible user and dependency sets for a given package – $p \in contents(r)$.

Finally, I define the package merging algorithm (see Algorithm 6.3) using the final correlated package sets. First, in line 4, the merging algorithm employs the package composition rule (defined in Equation 3.11) on every member of the final correlated package set (referred as $correlated(p, r)$, or in its short form "CPR"). Then after publishing the merged package (p_{merged}), the algorithm updates the direct package dependencies of the delta packages ($nodeps(p, r)$) that were dependent on the correlated packages. For example, in Figure 6.2 "Service1 delta" and "Service 1 JeOS delta" can be merged if other – not shown – packages are not dependent on them.

The two last tasks of the merge operation enables the later evaluation of the merging decision by the package destruction technique introduced in Section 6.2.4.1. First, the merging algorithm does not destroy the individual packages of the final correlated package set. Therefore, these packages are

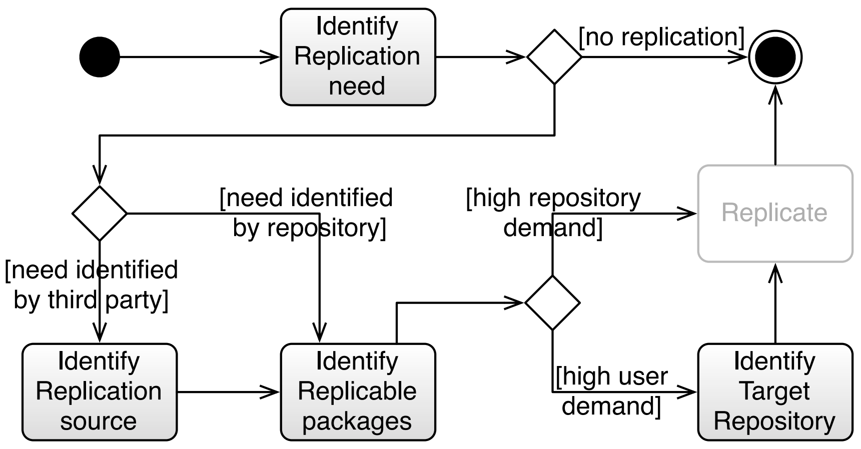

Identify Replication need
[no replication]
[need identified by repository]
[high repository demand]
Replicate
[need identified by third party]
Identify Replication source
Identify Replicable packages
[high user demand]
Identify Target Repository

Figure 6.3: Basic steps of package replication

still available as individual downloads; as a result, they could became unused by the time the analysis for package destruction is executed. Second, the merging algorithm marks the merged packages so they can be recognized as automatically created packages by the package destruction technique introduced later.

6.2.3 *Package replication*

This section introduces a technique that increases the availability of various repository packages in multiple repositories before their download. The proposed technique is based on the assumption that repository users download from the repository that offers their required packages with the highest bandwidth.

The proposed replication technique is executed in four phases (see Figure 6.3). First, it identifies the situations and repositories where there is room for improvement for increased efficiency in user downloads. Second, it identifies those packages that cause these situations. Later, my technique determines the ideal repositories in which the identified packages should have been located in order to avoid the non-efficient downloads. Finally, the proposed technique arranges the replication of the packages to the desired repositories. I am not discussing this last step because there are several already existing solutions that tackle the problem of data replication [17, 36]. Therefore, the following sub-sections detail the first three phases (marked with gray background in Figure 6.3).

6.2.3.1 *Identification of the replication need and source*

In this section, I discuss a method to classify different repositories based on their suitability to become replication sources. Therefore, first, I define replication sources as those repositories that are storing the least-efficient set of packages. The inefficiency of the repository content storage is identified through three cases: (i) when a repository is frequently *queried for non-stored content*, (ii) when a repository receives *frequent third party queries* to download parts of the stored appliances and by doing so it degrades its throughput for other queries, (iii) when there are *external dependencies* causing latencies for rebuilding virtual appliances.

NON-STORED CONTENT. If a locally available repository package is dependent on an externally available (i.e. non-stored) one, then its local rebuilding requires the download of the external package. The repository monitors the external package queries. Then it calculates the inefficiency for non-stored content ($ie_{nsc} : R_\varphi \to \mathbb{R}$) as the percentage of the external package queries ($Q_{Ex} : R_\varphi \to \wp(Q_\varphi)$) from all downloads ($Q_{All} : R_\varphi \to \wp(Q_\varphi)$) in a given repository:

$$\begin{aligned}
Q_{All}(r) &:= \{\forall q_i \in Q_\varphi : (r_{req,q_i} = r)\} \\
Q_{Ex}(r) &:= \{\forall q_i \in Q_\varphi : (h_{s,q_i} = r)\} \\
ie_{nsc}(r) &:= \frac{|Q_{Ex}(r)|}{|Q_{All}(r)|}
\end{aligned} \tag{6.2}$$

FREQUENT THIRD PARTY QUERIES. Repositories (r_1) log the source of the query for each download. If the query source is another repository (r_2), then I assume replicating the requested package could optimize the contents of the repositories in the service-based system. I defined the queries between two repositories with the function $Q_{amg} : R_\varphi^2 \to \wp(Q_\varphi)$. If a specific repository frequently queries packages from another one then this is a sign for inefficient package distribution. As a result, the sample standard deviation of the Q_{amg} sets can be used to identify if there are repositories that are *frequently requesting specific content* ($ie_{fr} : R_\varphi \to \mathbb{R}$).

$$Q_{amg}(r_1, r_2) := \left\{\forall q_i \in \left(Q_{All}(r_1) \bigcap Q_{Ex}(r_2)\right)\right\} \tag{6.3}$$

$$ie_{fr}(r) := \sqrt{\frac{1}{|R_\varphi| - 1} \sum_{r_x \in R_\varphi} \left(|Q_{amg}(r, r_x)| - \frac{1}{|R_\varphi|} \sum_{r_y \in R_\varphi} |Q_{amg}(r, r_y)|\right)^2}$$

EXTERNAL DEPENDENCIES. I have defined dependency groups ($DG : P_\varphi \to \wp(P_\varphi)$) using all packages with a common ancestor (p):

$$DG(p) := \{\forall p_x \in P_\varphi : (p \in dep(p_x) \vee p \in DG(p_x))\} \tag{6.4}$$

As a result, a dependency group can be defined for each package. Consequently, self-contained packages form a dependency group with all the service packages dependent on them, while, service packages form empty dependency groups. Active repositories use dependency groups to estimate the completeness of each virtual appliance accessible from the repository. Therefore, in each dependency group, repositories look for the external references – $ext : \wp(P_\varphi) \times R_\varphi \rightarrow \wp(P_\varphi)$:

$$ext(P,r) := \{\forall p \in P : (p \notin contents(r))\} \tag{6.5}$$

The factor of inefficiency for external dependencies ($ie_{ed} : P_\varphi \rightarrow \mathbb{R}$) is expressed by the number of external references ($|ext(DG(p),r)|$) within the different dependency groups:

$$ie_{ed}(r) := \frac{1}{|contents(r)|} \sum_{p \in contents(r)} \frac{|ext(DG(p),r))|}{|DG(p)|} \tag{6.6}$$

Finally, based on all the previously defined inefficiency values, each repository is assigned with a combined inefficiency value ($IE : R_\varphi \rightarrow \mathbb{R}$) that incorporates all the previously measured inefficiency values:

$$IE(r) := ie_{nsc}(r) \cdot ie_{fr}(r) \cdot ie_{ed}(r) \tag{6.7}$$

The replication need can be identified by (i) third parties (e.g. the AVS) and by (ii) the repositories themselves. In case of the AVS, the service investigates the global state of the repositories by evaluating the inefficiency value for all repositories. Then it calculates the sample standard deviation $s_N(IE(r))$ of the value set O by assuming its uniform distribution. If $s_N(IE(r))$ exceeds a predefined threshold, the AVS initiates the replication process on the repository with the highest inefficiency value. Later, I will refer to this repository as the *source repository* (r_s):

$$r_s := r \in R_\varphi : \left(IE(r) = \max_{r_x \in R_\varphi} IE(r)\right) \tag{6.8}$$

As part of the active repository functionality, the *repositories* themselves initiate the replication process locally (they can become source repositories – r_s). I propose that they maintain a historical database for their inefficiency values. Using this database the repository can evaluate the possible trends of its inefficiency values. If the repository identifies a specific trend, then they predict their future inefficiency values based on the identified trend. Next repositories use the predicted values to recognize the replication need if the inefficiency values reach a threshold specified by the repository administrator. In order to avoid the frequent rearrangement of repositories, changes on the repository content result in the reset of the historical inefficiency value database.

6.2.3.2 *Locating the replicable packages*

After the replication source repository has been selected, the system proceeds with the selection of the packages causing the inefficiency in the source repository. Unevenly downloaded packages are the main causes for inefficiency, therefore, the proposed technique measures the download frequency for each requested package. Then it estimates the required bandwidth ($BW_P : P_\varphi \times R_\varphi \to \mathbb{R}$) for each package, and uses this bandwidth information as the base measure to differentiate the optimality of a given package:

$$
\begin{aligned}
Q_{pac}(p,r) &:= \{\forall q_i \in Q_\varphi : ((p_{req,q_i} = p) \wedge (r_{req,q_i} = r))\} \\
BW_P(p,r) &= \frac{size(p) \cdot |Q_{pac}(p,r)|}{\max\limits_{q_i \in Q_{pac}(p,r)} t_{f,q_i} - \min\limits_{q_j \in Q_{pac}(p,r)} t_{s,q_i}}
\end{aligned} \tag{6.9}
$$

Where $Q_{pac} : P_\varphi \times R_\varphi \to \wp(Q_\varphi)$ defines the set of queries for a given package downloaded from a specific repository.

I propose to maintain a historical database of these bandwidth estimations for the packages of the repository. This database allows the detection of demand growth on a specific package by calculating the approximated gradient of the bandwidth estimation function. The replication should start on a package with steep increase on its demand. The list of proposed packages is established by sorting the current gradient values and selecting the package with maximal value ($p_c : R_\varphi \to P_\varphi$):

$$
p_c(r) := p \in contents(r) : \left(\frac{\Delta BW_P(p,r)}{\Delta t} = \max_{p_x \in contents(r)} \frac{\Delta BW_P(p,r)}{\Delta t} \right) \tag{6.10}
$$

6.2.3.3 *Determining the target repositories*

Then, in its last detailed phase, the package replication technique (see Figure 6.3) collects the request sources for all proposed packages. The request source is either another repository ($R_{rs} \in R_\varphi$) or a user ($u \in U_\varphi$) accessing the repository with an IaaS system. In order to equally handle repository and user requests, I propose to transform the user requests to equivalent candidate repositories. The candidate repository selection attempts to locate repositories that are more suitable for users downloading a specific package. Suitability is based on the behavior of current replica location algorithms (e.g. [64]) that choose repositories offering the user requested packages with higher bandwidth and lower latency than the source repository.

In order to estimate the available bandwidth and latency between the different repositories and the users, each repository measures and stores the network latency and bandwidth towards their query sources (h_{s,q_i}). Therefore, active repositories approximate the location of the users and the remote repositories by the network latency between the replication source repository (r_s) and the remote hosts: $l(r_s, h_{s,q_i})$.

The collected network latency values are used to group (called latency groups – $latgroup : \mathbb{R} \times P_\varphi \times R_\varphi^2 \to \wp(U_\varphi)$) the different users and repositories based on their latency differences. For each repository a latency group is formed by the user hosts with small latency differences to the given repository:

$$latgroup(\epsilon, r, r_s) := \begin{cases} \emptyset & \text{if } p_c(r_s) \in contents(r) \\ \forall u \in U_\varphi : (l(r_s, r) - l(r_s, u) < \epsilon) & \text{otherwise} \end{cases} \tag{6.11}$$

Later, the size of the latency groups is used to propose candidate repositories, therefore, the selection of ϵ is crucial. In order to provide the best selection ϵ is automatically selected so it produces the highest sample standard deviation for the cardinalities of the different latency groups:

$$\epsilon := k \in \mathbb{R} : \big(s_N(|latgroup(k, r, r_s)|) = \max_{k_x \in \mathbb{R}} s_N(|latgroup(k_x, r_i, r_s)|)\big) \tag{6.12}$$

The proposed technique determines the replication candidate repository by selecting the repository with the largest latency group:

$$r_c := r \in R_\varphi : (|latgroup(\epsilon, r, r_s)| = \max_{r_x \in R_\varphi} |latgroup(\epsilon, r_x, r_s)|) \tag{6.13}$$

After the candidate repository is selected, it is added to the set of the request source repositories ($r_c \in R_{rs}$). Next, in order to estimate the effect of the replication, the system calculates the required bandwidth for each request source repository ($BW_R : R_{rs} \to \mathbb{R}$):

$$\begin{gathered} Q_{latg} := \big\{\forall q_i \in Q_\varphi : \big((h_{s,q_i} \in latgroup(\epsilon, r_c, r_s)) \\ \wedge (r_{req,q_i} = r_c) \wedge (p_{req,q_i} = p_c(r_s))\big)\big\} \\ BW_R(p, r) := \begin{cases} \dfrac{size(p)|Q_{latg}|}{\max\limits_{q_i \in Q_{latg}} t_{f,q_i} - \min\limits_{q_i \in Q_{latg}} t_{s,q_i}} & \text{if } r = r_c \\ BW_P(p, r) & \text{otherwise} \end{cases} \end{gathered} \tag{6.14}$$

Where Q_{latg} denotes the set of queries initiated by the users in the latency group of r_c.

Based on the calculated bandwidth requirements the target repository is defined as follows:

$$r_t = r \in R_{rs} : \big(BW_R(p_c(r_s), r) = \max_{r_x \in R_{rs}} BW_R(p_c(r_s), r_x)\big) \tag{6.15}$$

Finally, all required information is available for the replication task, so the system orders the replication of the package $p_c(r_s)$ from the repository r_s to the repository r_t.

6.2.4 *Package destruction*

The proposed package destruction technique is the garbage collector of the active repository functionality. Package destruction frees up the repository storage of unused but automatically created packages. The active repository administrator can predefine a time interval (T_U) on which the unused criterion is defined. Using this constant, I have defined packages as unused (p_U) if they were not downloaded at all within the T_U interval:

$$unused(r) := \Big\{ \forall p \in contents(r) : \\ \Big(\nexists q_i \in Q_\varphi : ((p_{req_{q_i}} = p) \wedge (t_{s,q_i} < NOW - T_U)) \Big) \Big\} \tag{6.16}$$

Where NOW specifies the current timestamp in the system. Small T_U values (e.g., within a day) will result almost immediate disk cleanup in the repositories but will demand more bandwidth if the repository has to host the removed package again soon after deletion. Larger values will allow repositories to handle their disk space more gently thus the repository administrator have to decide the ideal T_U value depending on the available bandwidth and disk space for the repository. Throughout my experiments I have used 2 weeks for the value of T_U.

6.2.4.1 *Causes of unused packages*

Before proceeding with the package removal, I have investigated the possible causes of unused repository packages created by the active repository functionality. As proposed before, there are three automated ways for creating new packages in the repositories (see Figure 6.4): by decomposing a virtual appliance, by merging two packages, by replicating a package of an external repository. Any of these methods could falsely create a package that later on only consumes the storage space of the repository. During the investigation, I have identified the following five causes for unused packages:

1. Successful decomposition might lead to *diminishing demand for the decomposed virtual appliance packages* (see $p_{new} \in r_1$ in Figure 6.4). The demand decreases when virtual appliance rebuilding uses some of the decomposed packages from third party sources. In particular cases, the rebuilding algorithm does not choose the package containing the entire *original appliance*. Consequently, these situations render the original VA unused. Rebuilding decisions are further detailed in Section 6.3.

2. After the decomposition algorithm is applied, it splits two virtual appliances into three pieces (see Section 6.2.1 for details). If a *decomposed*

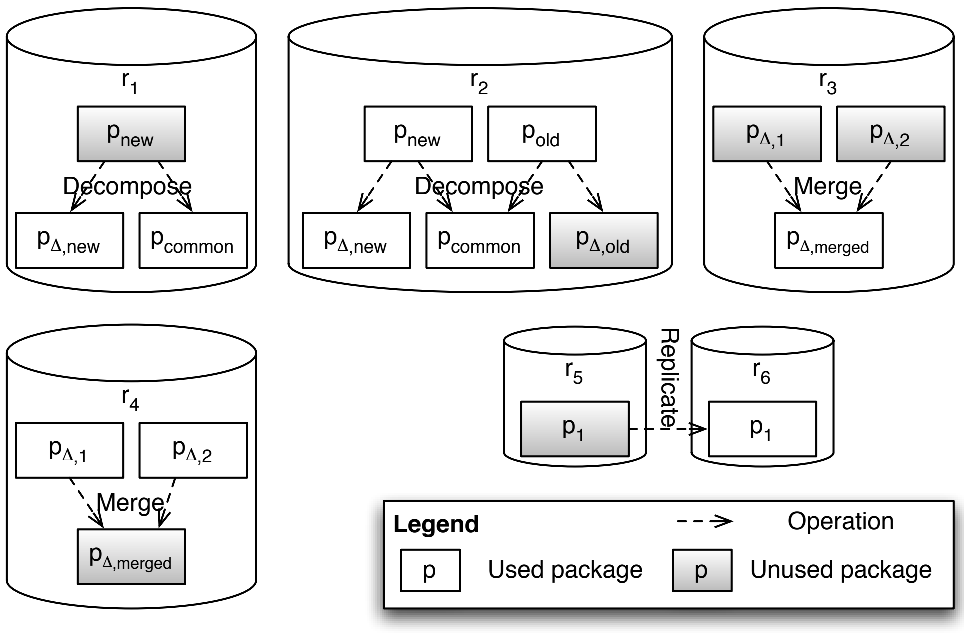


Figure 6.4: Identifying unused packages

virtual appliance attracts less demand (see $p_{\Delta,old} \in r_2$ in Figure 6.4) then the other appliance might be undesirably decomposed (see $p_{\Delta,new}, p_{common} \in r_2$). As a result, the three packages created during the decomposition become unused.

3. If the merging algorithm (see Section 6.2.2) successfully identifies correlations then it creates merged packages. After creating the merged package, the correlated packages (see $correlated(p_{\Delta,1}, r_3) = \{p_{\Delta,1}, p_{\Delta,2}\}$ in Figure 6.4) still remain in the system to allow their individual access. However, in some situations there is *no demand on the individual packages*, making them unused.

4. In contrary to the previous case, *merged packages lose their demand* when the appliance rebuilding algorithm starts to choose an external repository for an element in the correlated package set (e.g. $p_{\Delta,2} \in r_4$ in Figure 6.4). The merged package ($p_{\Delta,merged} \in r_4$) becomes unused when the external selection becomes a usual practice for rebuilding.

5. Finally, replicated packages (see $p_1 \in r_6$ in Figure 6.4) are created based on past experience. If the *demand patterns change soon after the replication* took place then the local copy of the package ($p_1 \in r_5$) can easily become unused.

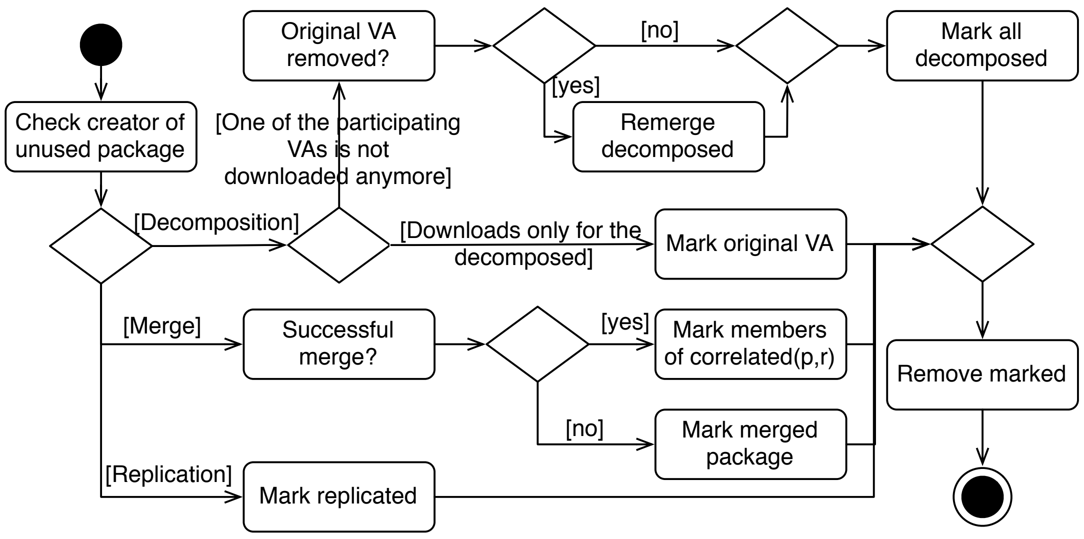


Figure 6.5: Activity diagram of the first phase of package destruction

6.2.4.2 *Two phased destruction*

Packages are removed from the repository in two phases: first, the system *removes appliance contents* only (see Figure 4.5), and then it *removes all package metadata* including the package definition. Appliance contents removal starts with the marking of removable packages.

The system starts the first phase of destruction by *marking* the identified unused packages (p_U) based on their cause. The proposed technique directly proceeds with the removal if the cause of the unused package is the original virtual appliance (cause 1 in Section 6.2.4.1 – "Mark original VA" step in Figure 6.5), merging (cause 3-4 – "Mark members of $correlated(p,r)$" and "Mark merged package" steps in Figure 6.5) or replication (cause 5 – "Mark replicated" step in Figure 6.5).

If a *decomposed virtual appliance attracts less demand* (cause 2) then, depending on the availability of the original virtual appliances, the repository behaves differently. If the original appliances are not available, then the repository reconstructs them by merging all the necessary ("*decomposed*") repository packages. Reconstruction brings the repository to the same state as if the original appliances would have been existed in parallel with the decomposed packages. The system *marks the decomposed packages* for removal after the original virtual appliances are available in the repository.

After the marking step, the appliance contents removal phase finishes with the removal action itself. The system *removes the marked appliance contents*, and places an expiration marker ($t_{Exp}(p)$) in the package specific metadata of the package. The technique leaves all other metadata intact in order to allow their later reuse. If the system identifies that a previous merging or decomposition decision should be taken again, then it is able to reuse the already specified packages. Therefore, their appliance contents are added again according to the metadata description. Hence, the system can save the often expensive metadata creation operations.

The second *metadata removal phase* is executed independently from the first phase. The system periodically (weekly in the current implementation) checks for packages (p_{WoC}) without appliance contents. The package definitions and all related metadata are destroyed when the actual time has past the expiration marker $t_{Exp}(p_{WoC})$.

6.3 VIRTUAL APPLIANCE REBUILDING

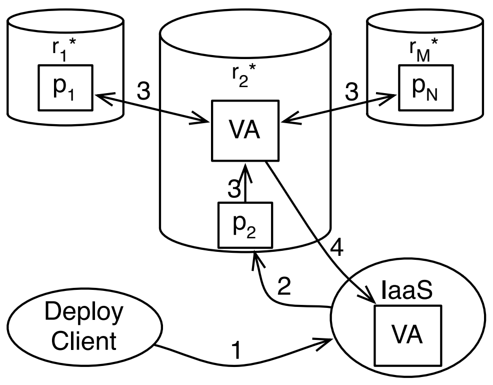
After applying the decomposition algorithm introduced in Section 6.2.1, virtual appliances are stored in multiple repository packages. The individual decomposed parts of a virtual appliance are not suitable for initiating a virtual machine. E.g. if the JeOS was separated from the service package (similarly to "split 1" in Figure 6.2) then the former could be used to initiate a virtual machine but without the target functionality for the users, even more, the service package cannot even function without the JeOS.

This section discusses the algorithm that rebuilds the original appliance before deployment if the appliance is offered as multiple packages. When a delta package is requested ($p_{req,q_i} \in contents(r_{req,q_i})$) in a query ($q_i \in Q_\varphi$) from repository r_{req,q_i}, then the rebuilding algorithm collects all packages the requested package depends on. As a result, when a service package is requested, its deployment will be preceded by the rebuilding of the entire virtual appliance from the collected packages.

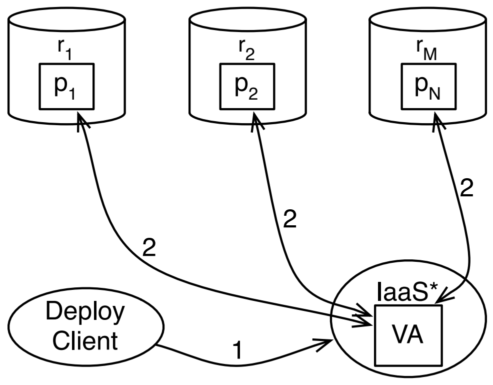
6.3.1 *Rebuilding scenarios and algorithm*

Similarly to the transformation algorithm introduced in Section 4.3, the proposed rebuilding algorithm can also be embedded in three different parties of the service-based system: (a) in the *active repository*, (b) in the *IaaS system* and (c) in the *deployment client*. All three embedding situations are presented in Figure 6.6. The components that incorporate the rebuilding functionality are emphasized in the figure with an asterisk.

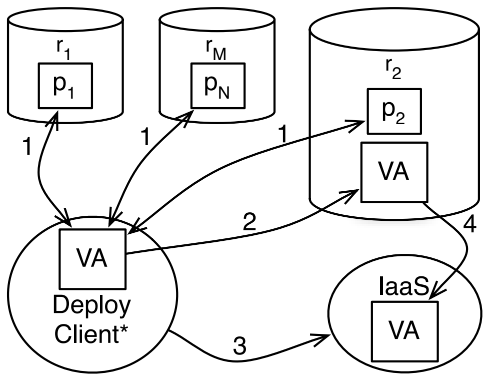
The differences between the embedding situations are highlighted with a rebuilding scenario, where the deployment client requests a service deployment on the specific IaaS system. The specifics of the different rebuilding scenarios are discussed in the following paragraphs. The scenarios are based on the following four basic assumptions: (i) the selection task of the deployment has already completed, (ii) the IaaS system has the highest bandwidth



(a) Active Repository



(b) IaaS



(c) Deployment Client

Figure 6.6: Options on embedding the rebuilding algorithm

connection towards r_2, (*iii*) at the beginning of the scenario the virtual appliance is not available in a self-contained package, and finally, (*iv*) only the component that embeds the rebuilding algorithm changes (see the starred components in Figure 6.6). The rebuilding algorithms introduced later are general and they are not built on these assumptions.

In case of *active repositories*, the IaaS receives the deployment request with the service package's identifier (see step 1 in Figure 6.6a). Therefore, the IaaS system only requests the service package before deployment (see step 2). However, the active repository does not allow the direct download of the service package. Instead, the repository collects the required packages to rebuild the virtual appliance of the service (see step 3). Finally, when the rebuilding has completed, the repository offers the rebuilt virtual appliance for download instead of the requested service package (see step 4).

If the *IaaS system* embeds the rebuilding algorithm, then it still receives the deployment request with the service package's identifier (see step 1 in Figure 6.6b). However, contrary to the previous solution, the IaaS directly collects all the dependencies of the service package and rebuilds its virtual appliance on the executor site (see step 2).

Finally, the *deployment client* could also embed the rebuilding algorithm. The client proceeds with the deployment with an entirely different approach. In step 1 (see Figure 6.6c), it downloads all required packages to its host to rebuild them. Then, in step 2, it publishes the rebuilt appliance in the repository with the highest bandwidth connection towards the target IaaS system. Afterwards, it requests the deployment of the service in step 3 by requesting the IaaS to initiate a VM with the rebuilt appliance. Finally, in step 4 the IaaS system downloads the rebuilt VA and deploys the service.

Even though the deployment client requires no modifications on the IaaS systems or on the repositories, the overhead of several extra transfers would make the rebuilding algorithm ineffective (assuming the client has low bandwidth connections). Therefore, later sections only discuss the first two embedding situations. Ideally, the IaaS systems should embed the rebuilding algorithm. However, they are usually commercially controlled, therefore, the introduction of active repositories is the most feasible extension on the current systems. Preferably, both IaaS systems and active repositories embed the rebuilding algorithm, and the system would choose between the rebuilding location based on package availability and network context.

6.3.1.1 *Basic rebuilding algorithm*

Figure 6.7 presents the overview of the proposed rebuilding algorithm. The figure follows the installation deployment task (see Section 1.1) from the request until the requested virtual appliance is ready for instantiation. The presented algorithm is applicable in all service-based systems indifferently from the entity that embeds the rebuilding algorithm. Therefore, an unmodified IaaS system could participate in the execution of the algorithm, because the algorithm utilizes the behavior of the IaaS systems – they request the packages to be deployed from an active repository that embeds the rebuilding algorithm.

The algorithm is composed of five stages: (*i*) rebuilding location selection, (*ii*) identification of the possible construction paths, (*iii*) selection of the ideal construction path, (*iv*) download of the required packages and (*v*) reconstruction of the virtual appliance.

During the *rebuilding location selection* (including actions marked with gray boxes in Figure 6.7), the algorithm decides between the IaaS system based rebuilding and the Active repository based one. If both the IaaS system and the active repository embed the rebuilding algorithm, then the system evaluates whether the IaaS system or an active repository has the *higher bandwidth connections* to transfer all required packages. If an external repository can

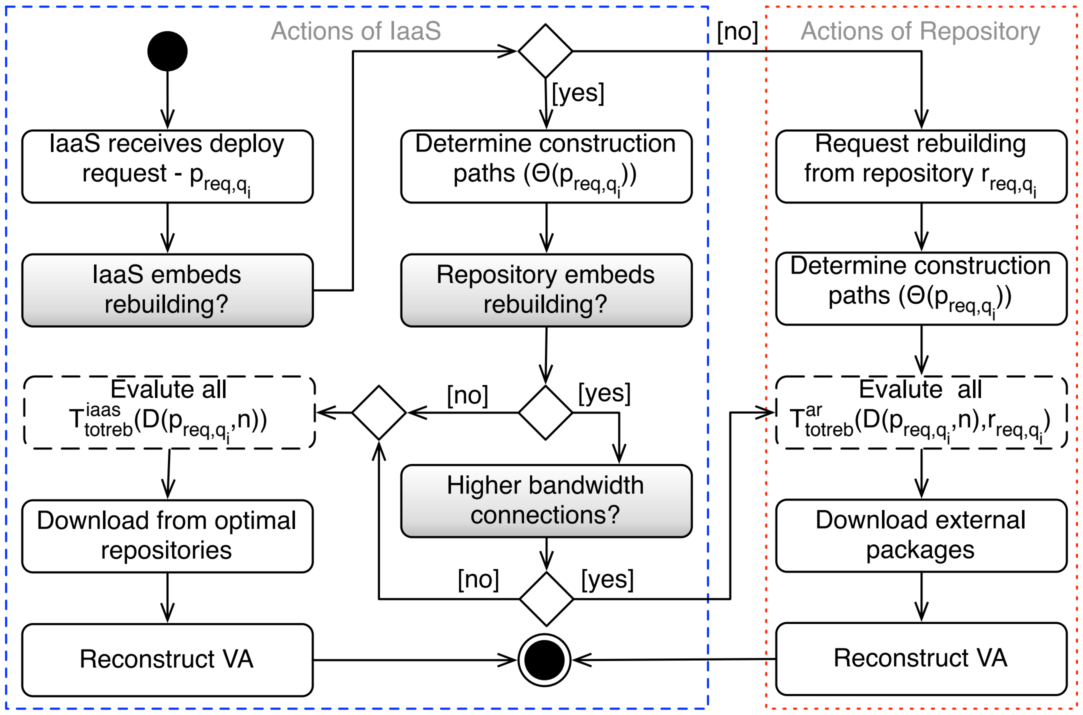

Actions of IaaS
Actions of Repository
IaaS receives deploy request - p_{req,q_i}
IaaS embeds rebuilding?
[yes]
[no]
Determine construction paths $(\Theta(p_{req,q_i}))$
Repository embeds rebuilding?
Evalute all $T^{iaas}_{totreb}(D(p_{req,q_i},n))$
Download from optimal repositories
Reconstruct VA
Higher bandwidth connections?
Request rebuilding from repository r_{req,q_i}
Evalute all $T^{ar}_{totreb}(D(p_{req,q_i},n),r_{req,q_i})$
Download external packages

Figure 6.7: Rebuilding algorithm utilizing both IaaS systems and active repositories

rebuild the virtual appliance and deliver it to the executor site faster than the IaaS system would, then the system rebuilds the appliance in the repository. In every other case, the system rebuilds the appliance locally in the IaaS system.

Construction paths are identified by populating the set of $\Theta(p_{req,q_i})$. The system calculates the possible dependency sets $(D(p_{req,q_i},n) \in \Theta(p_{req,q_i}))$ according to Equation 3.9. Each dependency set is equivalent to a construction path taking into consideration the dependencies among the packages in the set. Virtual appliance construction paths depict the reconstruction order of the original appliances, therefore, it is in reverse order of the dependencies starting from the self-contained package $(p_\Omega \in D(p_{req,q_i},n))$ of the set and finishing with the service package (p_{req,q_i}).

The core functionality of a rebuilding algorithm is the *selection of the ideal construction path*. The system computes rebuilding time estimates for each dependency set $(\forall n < PC(p_{req,q_i}) : D(p_{req,q_i},n)$ – shown with dashed boxes in Figure 6.7). These calculations are dependent on the location of embedding and are discussed in Sections 6.3.2 and 6.3.3. Afterwards, the selection of the ideal construction path, the download of the required packages and the reconstruction of the appliance are all independent from the location of rebuilding, therefore they are discussed separately in Section 6.3.4.

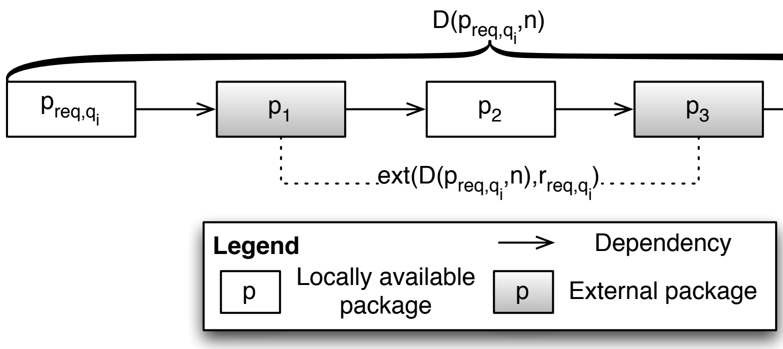


Figure 6.8: Package availability in a specific repository

6.3.2 *Rebuilding in active repositories*

The first discussed rebuilding technique utilizes the replication and merging functionalities of active repositories. The repositories incorporating the rebuilding algorithm always deliver self-contained packages. As a result, when a delta package ($p_{req,q_i} \in contents(r_{req,q_i})$) is requested then the repository ($r_{req,q_i} \in R_\varphi$) resolves its dependencies and automatically creates a self-contained package including the delta package and its dependencies. However, to spare storage space the repository does not publish this newly created self-contained package automatically. The merging algorithm (detailed in Section 6.2.2) manages the publication of these self-contained packages.

The algorithm attempts to use the locally available packages first. In order to determine the local packages usable during rebuilding, the algorithm distinguishes all external dependencies ($ext(D(p_{req,q_i},n),r_{req,q_i})$) of the requested delta package (for the definition of the dependency sets, see Equation 3.9). Figure 6.8 depicts the members of an example external dependency set ($ext(D(p_{req,q_i},n),r_{req,q_i}) \subset D(p_{req,q_i},n)$) with gray boxes. As seen in Figure 6.8, external dependencies (p_1 and p_3) could be scattered in the dependency set of the requested package. Therefore, the rebuilding algorithm decides between *downloading the individual dependencies* and reconstructing the entire self-contained package locally or requesting the external dependencies from other active repositories allowing their *external rebuilding*.

When the rebuilding technique reaches an external dependency ($p_{ext} \in ext(D(p_{req,q_i},n),r_{req,q_i})$) the algorithm estimates the cost of rebuilding locally ($t_{oplor} : P_\varphi \times R_\varphi \rightarrow \mathbb{R}$) – the optimal local rebuilding time) and externally ($t_{opexr} : P_\varphi \times R_\varphi \rightarrow \mathbb{R}$ – the optimal external rebuilding time). Based on the values of t_{oplor} and t_{opexr}, the system selects the least expensive rebuild-

Availability info	Collected for the following
$size(p)$	$\forall p \in D(p_{req,q_i}, n)$
$BW(r, r_{req,q_i})$	$\forall r \in R_\varphi : ((ext(D(p_{req,q_i}, n), r_{req,q_i}) \cap contents(r)) \neq \emptyset)$
$l(r, r_{req,q_i})$	$\forall r \in R_\varphi : ((ext(D(p_{req,q_i}, n), r_{req,q_i}) \cap contents(r)) \neq \emptyset)$

Table 6.1: Collected availability information

ing location ($r_{ler} : P_\varphi \times R_\varphi \rightarrow \mathbb{R}$) for the external dependency and the rest of the dependency set – $\exists m < PC(p_{ext}) : (D(p_{ext}, m) \subset D(p_{req,q_i}, n))$:

$$r_{ler}(p_{ext}, r_{req,q_i}) := \begin{cases} r_{req,q_i} & \text{if } t_{oplor}(p_{ext}, r_{req,q_i}) < t_{opexr}(p_{ext}, r_{req,q_i}) \\ r_{ext}(p_{ext}, r_{req,q_i}) & \text{otherwise} \end{cases} \tag{6.17}$$

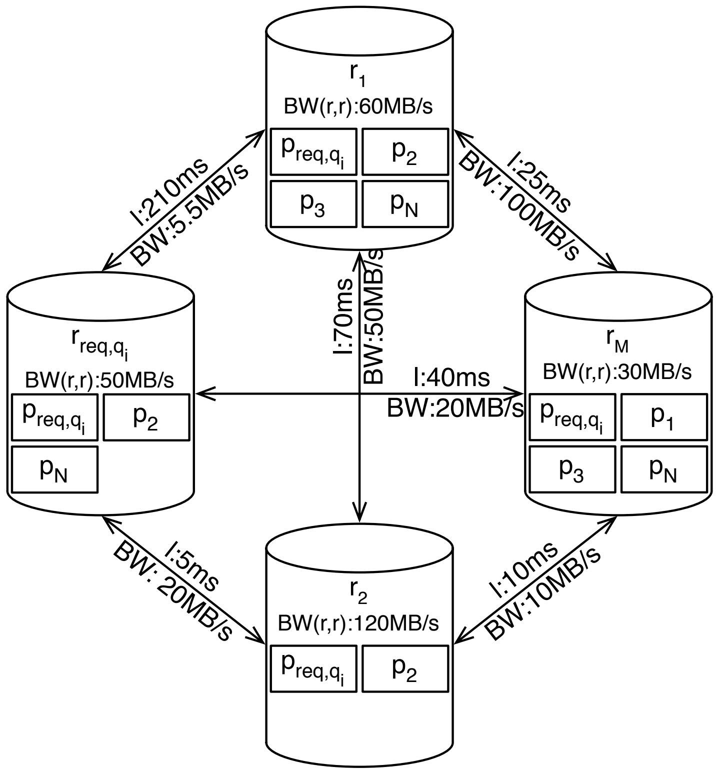
The following section discusses the two ways of rebuilding time estimation and also identifies the repository (r_{ext} – see Equation 6.22) used for the external rebuilding of the rest of the dependencies.

6.3.2.1 *Estimating rebuilding time*

Estimating the rebuilding cost requires the repositories to collect the availability information (see Table 6.1) on all packages in the used dependency set ($p \in D(p_{req,q_i}, n) : n < PC(p_{req,q_i})$). Figure 6.9 presents the availability information collected for several example repositories, their connections and the packages stored in them (the dependencies of the packages are shown in Figure 6.8). In the current system, the cost is approximated by the estimated time required for rebuilding. In general, rebuilding time ($t_{reb} : P_\varphi \times R_\varphi \times \varphi \rightarrow \mathbb{R}$) has two basic components, *the transfer time* – $t_{trans} : P_\varphi \times R_\varphi \times \varphi \rightarrow \mathbb{R}$ – of the package to the location of rebuilding and the time – $t_{comp} : P_\varphi \times \varphi \rightarrow \mathbb{R}$ – required to apply the package composition rule on the transferred package and its dependencies.

$$\begin{aligned} t_{trans}(p, r, h) &:= \begin{cases} l(r, h) + size(\{p\})/BW(r, h) & \text{if } r \neq h \\ 0 & \text{otherwise} \end{cases} \\ t_{comp}(p, h) &:= size(\{p\})/BW(h, h) \\ t_{reb}(p, r, h) &:= t_{trans}(p, r, h) + t_{comp}(p, h) \end{aligned} \tag{6.18}$$

Where $t_{trans}(p, r, h)$ specifies the transfer time of package p from repository r to the rebuilding host h. In case the package under rebuilding is available locally then the transfer time is ignored. Next, $t_{comp}(p, h)$ defines the composition time of package p in the rebuilding host h. For the estimation of



Package	$size(p)$
p_{req,q_i}	5MB
p_1	12MB
p_2	230MB
p_3	70MB
p_N	25MB

Figure 6.9: Example repository layout with availability information and stored packages

the composition time, the local storage bandwidth of the rebuilding host – $BW(h,h)$ – is used to represent the speed of the rebuilding host while it applies the composition rule on a delta package and a self-contained package using the package composition rule (see Equation 3.11).

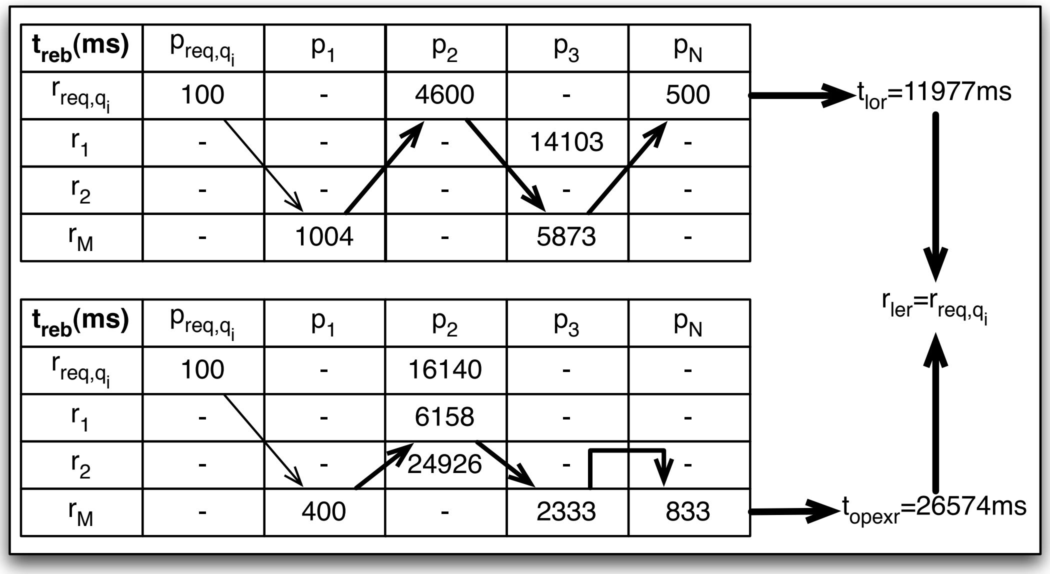
local rebuilding. Based on the previously defined rebuilding time values, the system calculates the optimal rebuilding time ($t_{opreb} : P_\varphi \times R_\varphi \to \mathbb{R}$) for a single package:

$$t_{opreb}(p, r_{req}) := \min_{r \in R_\varphi : (p \in contents(r))} t_{reb}(p, r, r_{req}) \tag{6.19}$$

Where the equation presents the optimal rebuilding time of package p in the repository r_{req}.

Then the system aggregates the optimal rebuilding times for the remaining dependencies in order to estimate the total *local optimal rebuilding time* – $t_{oplor} : P_\varphi \times R_\varphi \to \mathbb{R}$:

$$t_{oplor}(p, r) := \sum_{p_x \in P_\varphi}^{\exists n < PC(p) : (p_x \in D(p,n))} t_{opreb}(p_x, r) \tag{6.20}$$



t_{reb}(ms)	p_{req,q_i}	p_1	p_2	p_3	p_N
r_{req,q_i}	100	-	4600	-	500
r_1	-	-	-	14103	-
r_2	-	-	-	-	-
r_M	-	1004	-	5873	-

t_{reb}(ms)	p_{req,q_i}	p_1	p_2	p_3	p_N
r_{req,q_i}	100	-	16140	-	-
r_1	-	-	6158	-	-
r_2	-	-	24926	-	-
r_M	-	400	-	2333	833

t_{lor}=11977ms
$r_{ler}=r_{req,q_i}$
t_{opexr}=26574ms

Table 6.2: Evaluation of the rebuilding options for p_1

EXTERNAL REBUILDING. For each external package (p_{ext}) the system evaluates the *external rebuilding* time as follows:

$$t_{exr}(p_{ext}, r, r_{req}) := l(r, r_{req}) + \sum_{p \in P_\varphi}^{\exists n < PC(p_{ext}):(p_x \in D(p_{ext}, n))} \left(\frac{size(\{p\})}{BW(r, r_{req})} + t_{opreb}(p, r) \right)$$

$$t_{opexr}(p_{ext}, r_{req}) := \min_{r \in R_\varphi:(p_{ext} \in contents(r))} t_{exr}(p_{ext}, r, r_{req}) \tag{6.21}$$

External rebuilding requires p_{ext} to be reconstructed in a remote repository, therefore, the rebuilt package has to be transferred to the repository of the user request. Consequently, t_{exr} is calculated in two parts: (i) the download time estimation of the rebuilt package (the bandwidth dependent part – BW), and (ii) the rebuilding time of the package at the remote repository – $t_{opreb}(p, r)$. Similarly to t_{opreb}, t_{opexr} specifies the optimal rebuilding time for an external package if it is rebuilt entirely in the external repository (r_{ext}).

The system offers the most suitable repository for external rebuilding ($r_{ext} : P_\varphi \times R_\varphi \to R_\varphi$) as:

$$r_{ext}(p_{ext}, r_{req}) := r \in R_\varphi : \left(t_{exr}(p_{ext}, r, r_{req}) = t_{opexr}(p_{ext}, r_{req})\right) \tag{6.22}$$

Table 6.2 offers the example evaluation of the $t_{oplor}(p_1)$ and $t_{opexr}(p_1)$ values based on the availability information discussed in Figure 6.9. The bold arrows in Table 6.2 represent the decisions made between the different repositories before the reconstruction of p_1. The remaining arrows represent the decisions made before the algorithm was processing p_1. Therefore, the algorithm in the example arrives to the conclusion that the least expensive rebuilding location for p_1 is the repository that received the query (r_{req,q_i}).

Finally, the total rebuilding time for the requested package is calculated with the following formulae ($T^{ar}_{totreb} : P_\varphi \times \mathbb{N} \times R_\varphi \to \mathbb{R}$):

$$T^{ar}_{totreb}(p,n,r) := \min_{p_x \in ext(D(p,n),r)} \Big(t_{opexr}(p_x,r) + \sum_{p_{loc} \in (D(p,n) \setminus DG(p_x))} t_{reb}(p_{loc},r,r) \Big) \quad (6.23)$$

As a result, the system calculates the total rebuilding time based on the external packages. For each external package, the system calculates its external rebuilding time and the time required to apply the composition rule with the locally available packages.

6.3.3 *Rebuilding in the IaaS system*

This rebuilding algorithm constructs the original virtual appliance on the execution site ($\xi \in \Xi_\varphi(c)$ and $c \in C_\varphi$) using a self-contained package and several delta packages according to the dependency set of the requested package in the deployment query ($D(p_{req,q_i},n)$). All packages are downloaded from the various remote repositories directly to the executor site, where they are reconstructed. The IaaS system collects the list of repositories offering all dependent packages ($p \in D(p_{req,q_i},n)$), and determines their accessibility (their latency – $l(r,\xi)$ – and bandwidth – $BW(r,\xi)$). The accessibility information is collected prior the application of the rebuilding algorithm. The IaaS only initiates the virtual machine to host the virtual appliance after all packages in the dependency set are downloaded and the appliance is rebuilt on the execution site.

During the *repository selection* phase, the proposed IaaS based rebuilding algorithm first identifies all required packages and the repositories offering them ($R_{dep}(p,n) := \{\forall r \in R_\varphi : (D(p,n) \cap contents(r) \neq \emptyset)\}$). The algorithm identifies the repository offering the package with maximum bandwidth by estimating the rebuilding time ($t_{reb}(p,r,\xi)$) on the execution site for each package ($p \in D(p_{req,q_i},n)$) in the dependency set of p_{req,q_i}. As a result, the proposed technique computes the optimal download time for each package ($t_{oir} : P_\varphi \times \varphi \to \mathbb{R}$) as:

$$t_{oir}(p,\xi) := \min_{r \in R_\varphi} t_{reb}(p,r,\xi) \quad (6.24)$$

Using the values of $t_{oir}(p,\xi)$ function the algorithm calculates the total rebuilding time in the IaaS system with the following method ($T^{iaas}_{totreb} : P_\varphi \times \mathbb{N} \times R_\varphi \to \mathbb{R}$):

$$T^{iaas}_{totreb}(p,n,\xi) := \sum_{p_x \in D(p,n)} t_{oir}(p,\xi) \quad (6.25)$$

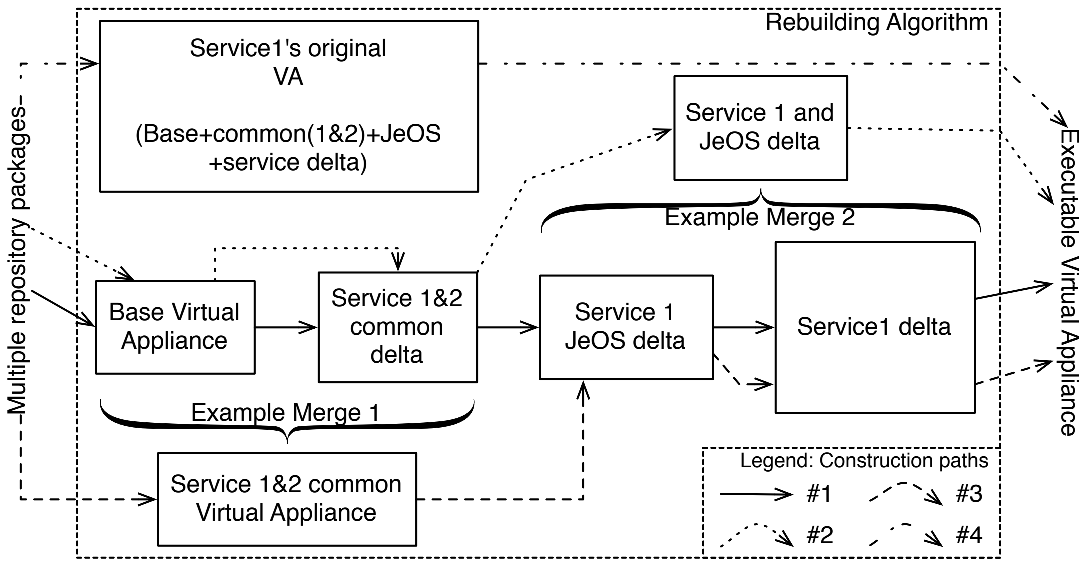

Rebuilding Algorithm
Service1's original VA (Base+common(1&2)+JeOS +service delta)
Service 1 and JeOS delta
Example Merge 2
Multiple repository packages
Base Virtual Appliance
Service 1&2 common delta
Service 1 JeOS delta
Service1 delta
Example Merge 1
Service 1&2 common Virtual Appliance
Executable Virtual Appliance
Legend: Construction paths
#1
#2
#3
#4

Figure 6.10: Virtual appliance rebuilding options

6.3.4 *Reconstructing the virtual appliance*

Figure 6.10 reveals that after several merge operations the repositories could contain the following packages: (i) packages representing the optimal decomposition (shown in Figure 6.2), (ii) packages resulted from previous merge operations ("example merge 1-2") and (iii) the original virtual appliance itself. Based on these packages and their dependencies (represented as arrows in Figure 6.10) repository selection algorithms should evaluate their selection procedures on all possible construction paths (the different dependency sets – $D(p_{req}, n) \in \Theta(p_{req})$ – of a specific package) starting from the decomposed packages and finishing at the rebuilt virtual appliance. Finally, because of package merging and destruction, there are several ways to construct a decomposed virtual appliance, therefore, the selection algorithm chooses the construction path ($D_{con} : P_\varphi \times \varphi \rightarrow \Theta(p_{req})$) with the lowest cumulative rebuilding cost.

$$D_{con}(p_{req}, h_{reb}) := D(p_{req}, n) \in \Theta(p_{req}) : \Big(\forall n, m < PC(p_{req}) : \\ \big(T_{totreb}(p, n, h_{reb}) \leq T_{totreb}(p, m, h_{reb})\big)\Big) \tag{6.26}$$

The system uses the T_{totreb} function according to the rebuilding location (e.g. in case of the IaaS system based rebuilding the function T^{iaas}_{totreb} is used). During the evaluation of T_{totreb}, the host of the rebuilding is depicted with $h_{reb} \in \varphi$.

After the algorithm selects the ideal reconstruction path, it executes the reconstruction in two steps. First, it downloads the packages that are not present at the location of rebuilding. In case of repositories, only packages in the $ext(D(p_{req}, n), r_{req})$ and their dependencies are considered for downloading (see Section 6.3.2 for details). However, if the rebuilding is applied by the IaaS system then all packages in the dependency set are downloaded.

6.3.4.1 *Reconstruction of the ideal reconstruction path*

The last phase of rebuilding is the reconstruction of the virtual appliance based on all the packages in the dependency set (D_{con}). The proposed *offline reconstruction strategy* downloads and constructs the original virtual appliance prior it is used for initiating a virtual machine. The strategy creates a disk image with the service's base virtual appliance (see Section 3.3.1 on page 35) and adds the contents of the delta packages to the image. Then the metadata of the original virtual appliance is attached to the disk image in order to allow its local deployment.

Delta packages could be added to the base virtual appliance's disk image on two levels. If the delta packages are stored as extensions of the disk image in the base virtual appliance then the system could apply *low-level addition* procedures where the base virtual appliance's disk image is extended with the contents of the delta package. Therefore, the system does not need to understand the file system of the base virtual appliance. However, the preparation of these kinds of base and delta packages would require special decomposition algorithms that are not discussed in this thesis.

Therefore, the current system applies a *high level addition* procedure that mounts the base package's file system on the rebuilding machine's host, then copies the files coming from the delta packages. This way the format of the delta packages is different from the format of the base packages, since the base packages contain a virtual appliance with a disk image. In contrast, the delta packages are simple file archives (e.g. zips).

CHAPTER SUMMARY. This chapter argued about the need for active repositories that are special repositories behaving and controlling their contents autonomously. I defined the autonomous behavior through the introduction of package decomposition, merging, destruction and replication. However, as a result of the autonomous behavior, appliances are no longer available in self-contained packages. The chapter has concluded by proposing an appliance rebuilding technique that ensures appliances are always received by the deployment host in a self-contained package.

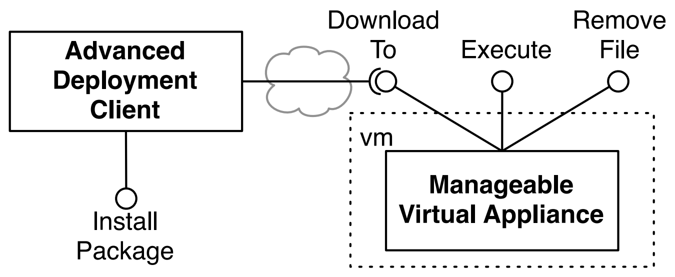
7 THE MINIMAL MANAGEABLE VIRTUAL APPLIANCE

CHAPTER OVERVIEW. This chapter focuses on the discussion of my fourth contribution. The chapter finds that the possible modification of virtual machine contents after deployment is an essential for efficient virtual appliance delivery and size optimization. Thus first it proposes a new concept (the minimal manageable virtual appliance – MMVA) that offers runtime appliance content modification options for the appliances embedding the MMVA. Then the chapter finishes with the discussion of how these new kind of appliances can be more efficiently rebuilt, transformed and how MMVAs streamline the appliance optimization technique introduced in Chapter 5.

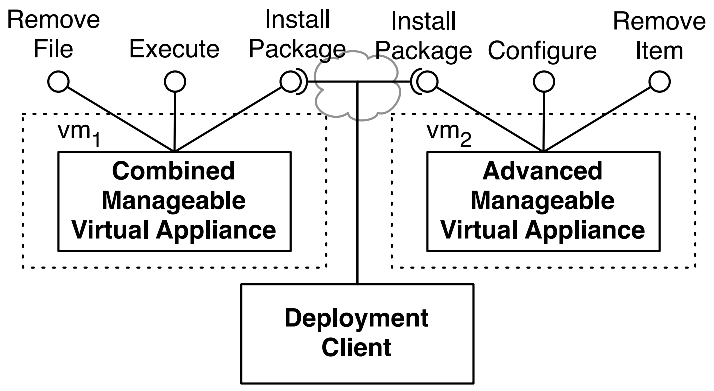
7.1 INTRODUCTION

For the last contribution (see Section 1.3.4) of this dissertation I have identified two challenges that were not targeted previously: (*i*) enable more wide spread adaptation of the proposed techniques by reducing the invasiveness of the AVS service on the already existing infrastructure and (*ii*) increase the effectiveness of the previously proposed algorithms by exploiting the capabilities of the deployed virtual appliances. Wider adaptation can be achieved by substantially extending the deployment clients. These extensions were discussed in Section 4.3 for appliance transformation and in Section 6.3.1 for embedding rebuilding algorithms. However, as it was revealed in the aforementioned sections, applying these extensions would raise serious efficiency issues in the system (e.g., require the transfer of entire transformed virtual appliances).

The optimization facility (introduced in Chapter 5) has raised two unanswered issues. First, the removal of an item from a virtual appliance might imply the upload of the modified virtual appliance (see Section 5.2.2 for details) before the system could proceed to validation. Second, as it was discussed in Section 5.2.3.1, virtual appliances cannot be modified after their



(a) Basic interfaces



(b) Combined & Advanced interfaces

Figure 7.1: Minimal interfaces of a manageable virtual appliance

instantiation, therefore, the system must create a new virtual machine for every proposed item removal.

Manageable virtual appliances offer a handful of operations that allow the modification of the virtual machine that executes the deployed appliances. In this chapter, I reveal that these special virtual appliances could solve the efficiency issues of the deployment clients and the bottlenecks in the optimization facility. Because of the increased efficiency of the deployment clients, these special virtual appliances enable the techniques proposed in this dissertation to be applied in already existing infrastructures (like the Amazon EC2).

7.2 DEFINITION OF THE MANAGEABLE VIRTUAL APPLIANCE

Manageable virtual appliances (MVA) can be defined in various ways. However, my virtual appliance creation architecture only requires that these virtual appliances are capable of (*i*) installing a package from a remote repository – required for rebuilding and transformation –, (*ii*) configuring the virtual machine according to the configuration deployment task (see Section 1.1) – required for all AVS operations –, and (*iii*) erasing items from the virtual machines based on them – required for size optimization. Instantiating a manageable virtual appliance results in a manageable virtual machine that allows the modification of its contents and state.

Based on these prerequisites, I have defined three interface sets (see Figure 7.1) for the manageable virtual appliances. Other interfaces could also meet these prerequisites (e.g. allowing the runtime adaptation of the service in the virtual appliance). However, this thesis only considers these three sets, because they are already capable of supporting the proposed architecture:

BASIC INTERFACES offer the minimal required operations to accomplish the prerequisites. These interfaces require an advanced deployment client capable of transforming the high level prerequisites of the architecture to the low level operations provided by the manageable appliances. Consequently, the client should be fully aware of the contents of the virtual appliance, and it should be capable of managing the appliance on file system level. As depicted in Figure 7.1a, these interfaces allow the following operations: (i) *downloading* a single file from a protocol restricted URL (e.g. only rsync, http URLs are allowed) to a designated location in the executor virtual machine, (ii) *executing* a configuration script in the virtual machine and (iii) *removing* a file from one of the file systems used by the executor virtual machine. The "Download To" operation introduces a bottleneck in the system, as it requires the advanced deployment client to download and itemize repository packages before they can be uploaded as files through the interface. Consequently, advanced deployment clients would require the double download of the packages – first between the repository and the host of the deployment client (h_{cli}), then between h_{cli} and the host of the manageable virtual appliance ($vm \in vms(c) : c \in C_\varphi$).

ADVANCED INTERFACES realize the prerequisites on an AVS conform approach. Therefore, these interfaces can download packages from repositories, can process install and configure these packages, and can operate on the same item type the AVS implementation is using. Figure 7.1b displays the advanced virtual appliance approach with the following operations: (i) "InstallPackage" downloads a repository package from an arbitrary repository, then unpacks the downloaded content and places the arrived items on their designated locations; (ii) "Configure" enables the virtual appliance to be configured through in an appliance specific way; finally, (iii) the "RemoveItem" operation drops a single or multiple items from the executor virtual machine.

COMBINED INTERFACES are the mixture of the advanced and basic interfaces (see Figure 7.1b). Combined interfaces are designed to allow the simplest implementation of the manageable virtual appliances with the use of regular deployment clients. Previously, I have identified the "Download To" operation of the basic interface as its bottleneck. Therefore, the combined interfaces replace the"Download To" operation of the basic interfaces with the "InstallPackage" operation of the advanced interfaces.

7.2.1 *Maintaining the Management Capabilities of Virtual Appliances*

The more advanced the MVAs become the higher their impact on the overall size of their appliance. However, Section 5.1 already revealed that the size of the appliance heavily influences the deployment time of the encapsulated service. As the management interfaces require new functionality of the virtual appliances, they inevitably increase the deployment time of the service, therefore, it is preferred to choose the smallest possible MVAs for each service. Manageable virtual appliances with combined interfaces provide the smallest sized virtual appliances that are still versatile enough for the optimal behavior of the AVS architecture. As the architecture tries to decompose and optimize the size of the virtual appliances, it might break the management functionality. Therefore, the following paragraphs discuss the built-in defense mechanisms of the AVS service.

To achieve the smallest possible MVAs the proposed architecture could optimize the size of the MVAs with the optimization facility. In order to avoid the removal of the management interfaces from the MVAs the validation of these appliances requires their service packages (see Figure 4.5 and Equation 3.17) to contain the test for the management interfaces ($manageable(p_\sigma) : P_\varphi \to \{true, false\}$).

The test of a combined interface is defined in the Algorithm 7.1 – for the operations of the interface see Section 7.2 on page 104. The algorithm tests whether the virtual appliance of p_σ incorporates the previously defined management capabilities for the AVS architecture. First, in lines 1-3, the algorithm selects a reference service package (p_{ref}) that can be installed in the virtual machine ($vm \in \bigcup_{c \in C_\varphi} vms(c)$) instantiated from p_σ. Next, it installs p_{ref} in the vm using the management interfaces in p_σ. As a result, if the management interfaces are available in the virtual machine then vm should successfully pass the validation in line 8. Afterwards, the items of p_{ref} are removed from the vm to test its $removeFile$ interface. Consequently, the vm should not pass the validation in line 12. Finally, the algorithm only accepts the management capabilities of p_σ if the first validation of p_{ref} succeeds and the last fails.

Algorithm 7.1 The test for the management capabilities

```
Require: $\varphi \in \Phi$
Require: $p_\sigma \in P_\varphi : (servicepkg(p_\sigma) = true)$
 1: $P_{refs} \leftarrow \Big\{\forall p \in P_\varphi : \Big(\exists n < PC(p) : \big((D(p,n)\backslash p) \subset \Theta(p_\sigma)$
                    $\wedge servicepkg(p) = true\big)\Big)\Big\}$
 2: $vm \leftarrow initVM(p_\sigma, \varphi)$
 3: $p_{ref} \leftarrow p \in P_{refs} : \Big(\big(pkgsize(D(p,0)) = \min_{p_x \in P_{refs}} pkgsize(D(p_x,0))\big)$
                    $\wedge valid(p, vm) = false\Big)$
 4: for all $p \in D(p_{ref}, 0)$ do
 5:     $installPackage(vm, p)$
 6:     $executeOn(vm, configurator(p))$
 7: end for
 8: $before \leftarrow valid(p_{ref}, vm) = true$
 9: for all $i \in items(p_{ref}) : (type(i) = \text{"}file\text{"})$ do
10:     $removeFile(vm, i)$
11: end for
12: $after \leftarrow valid(p_{ref}, vm) = true$
13: return $manageable(p_\sigma) \leftarrow before \wedge \neg after$
```

MANAGEABLE BASE VIRTUAL APPLIANCES are those base virtual appliances that offer one of the previously discussed management interfaces (see Section 7.2). These kinds of base virtual appliances are later used for the online reconstruction strategy (detailed in Section 7.3.1.2). Consequently, the system should stop applying the decomposition algorithm (introduced in Section 6.2.1) on manageable virtual appliances before the loss of management capabilities. Therefore, Algorithm 6.1 has to be extended to ensure base appliances always offer the management interfaces in order to allow their online reconstruction. The extension replaces the condition statement in Line 15 of Algorithm 6.1 with the following:

$$(baseva(p_{\Omega,common}) = true) \wedge (manageable(p_{\Omega,common}) = true) \tag{7.1}$$

7.2.2 *Minimal Manageable Virtual Appliances*

The management interfaces affect the deployment time of the virtual appliances they are embedded in. In case of optimally sized self-contained service packages these interfaces are not used during deployment. Therefore, the interfaces only increase the size and deployment time of the appliances. As an opposite, manageable virtual appliances stored in multiple packages could

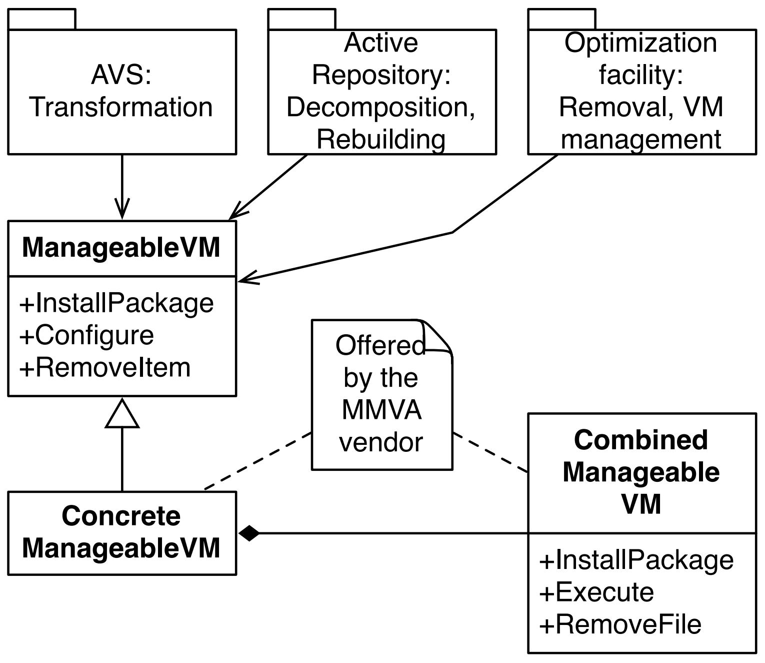


Figure 7.2: Interfacing between MMVAs and the AVS architecture

utilize their management interfaces to reconstruct the virtual appliances on site. This rebuilding technique could lower the deployment time. In both cases, the size of the management interfaces influences the deployment time of the encapsulating virtual appliances. Therefore, these interfaces are required to occupy minimal size from their hosting appliance. To ensure minimal impact on their hosts I propose to create predefined template virtual appliances exclusively for the management interfaces.

These template virtual appliances need to be delivered with the appliance creation service. Therefore, they could form the basis of the services published later with the AVS. I propose to call these templates "*Minimal Manageable Virtual Appliances*" or in short form MMVAs.

Minimal manageable virtual appliances (p_μ) are size optimized base virtual appliance packages that offer management interfaces only:

$$\begin{aligned} mmva(p) := (baseva(p) = true) \wedge (manageable(p) = true) \\ \wedge (optimalsize(p) = true) \end{aligned} \tag{7.2}$$

MMVAs are special base virtual appliances that are delivered with a validator (e.g. with the one defined in Algorithm 7.1) – for the details of packaging see Section 4.4.1. Consequently, the optimization facility can be used to create such virtual appliances from manageable base virtual appliances.

Since it is possible to define multiple minimal manageable virtual appliances, the appliance creation service has to offer a common way to interface with them. This common interface is detailed in Figure 7.2. The first row of the figure depicts the components of the AVS architecture and those functionalities that could be dependent on minimal manageable virtual appliances. Before these components use the MMVA, they always initiate a virtual machine based on it. The AVS architecture provides an abstract interface of this virtual machine called *ManageableVM*. This interface behaves as the common interface between the AVS and the various MMVAs by providing the advanced management interfaces specified in Section 7.2. The AVS allows the implementation of this common interface by third parties to support concrete manageable virtual machines. This implementation should translate the requests of the AVS components towards the manageable VM to a vendor dependent form. For example, the last row of the figure depicts my combined management interface set as the target of the AVS requests.

7.2.2.1 *Creating virtual appliances based on an MMVA*

The use of minimal manageable virtual appliances changes the system's invasiveness on the infrastructure to the invasiveness towards the virtual appliances. As a result, appliance developers should prepare their initial appliances to incorporate MMVAs.

I have identified two basic methods of creating an MMVA based virtual appliance. First, the appliance developers could *extend* the MMVAs by utilizing them as the foundation for their own services. Alternatively, they could *append* the files of the image in the MMVA ($im(p_\mu, f)$) to extend their virtual appliance with the minimal management interfaces.

If developers decide in favor of the *MMVA extension*, then they should first find a suitable MMVA to extend. To determine the most suitable MMVA, appliance developers investigate the offered functionality and contents of the various available MMVAs and select one that requires the least effort for extension. Then, they should instantiate a virtual machine with the selected MMVA. During instantiation, they should ensure that the new virtual machine could host both their service and the selected MMVA. Afterwards, they proceed with the installation of the service inside the newly created virtual machine as if they would do with the central extraction service (discussed in Section 4.2.1.1).

In case of the second scenario (previously referred as "*append*"), developers prepare their services for extraction as they prefer. Consequently, before progressing further they could already create the initial virtual appliance without management interfaces – I refer to this hypothetical appliance as $p^*_{\Omega,nm}$. However, as a final step before the appliance creation, appliance developers download and itemize a suitable MMVA – p_μ. As a result, they receive the items ($items(p_\mu)$) that ensemble the MMVA. Finally, they append

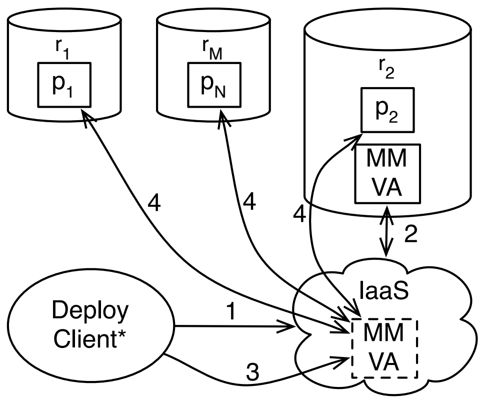


Figure 7.3: Deployment client using an MMVA during VA rebuilding

these items to their already prepared service and then proceed with the initial appliance (p_Ω) creation:

$$items(p_\Omega) := items(p^*_{\Omega,nm}) \cup items(p_\mu)$$

In this second scenario, appliance developers have to choose an MMVA that can operate in the hosting environment of their initial service installation: $manageable(p_\Omega) = true$. If the AVS detects that the MMVA is not successfully appended ($manageable(p_\Omega) = false$) then it cancels the publication of the initial virtual appliance allowing the appliance developer to select a more suitable MMVA.

7.3 ARCHITECTURAL DEVELOPMENTS

This section discusses the differences and extensions of the architecture when it is handling (rebuilds, transforms or optimizes) a manageable virtual appliance. As the system supports both manageable and non-manageable virtual appliances, the architecture automatically chooses between the previously presented solutions (in case of non-manageable VAs) and the new solutions introduced in this section.

7.3.1 *Effects on the rebuilding algorithms*

An embedded MMVA, in the virtual appliance of the service (p_{req}) under deployment, opens the feasibility of the deployment client based rebuilding (initially discussed in Section 6.3.1). As it is depicted in Figure 7.3, the deployment client can use the MMVA for its advantage and initiate it on the target IaaS system (see step 1). Afterwards, during step 2, the IaaS system downloads the MMVA from the repository r_2 then creates a virtual machine ($vm \in \{\bigcup_{c \in C_\varphi} vms(c)\}$) based on the MMVA. Next, in step 3, the client

Algorithm 7.2 Estimating connectivity details between the repositories and the virtual machine of an MMVA

```
Require: $timeout, 0 < \tau < 1, awaitedbw, maxtestsize$
Require: $vm \in \{\bigcup_{c \in C_\varphi} vms(c)\}$
Require: $p_{req} \in P_\varphi$
Require: $p_\mu \in P_\varphi : \Big(mmva(p_\mu) = true$
             $\wedge\big(\exists D(p_{req}, x) \in \Theta(p_{req}) : (p_\mu \in D(p_{req}, x))\big)\Big)$
 1: for all $r \in R_\varphi : \Big(\exists D(p_{req}, x) \in \Theta(p_{req}) : \big(contents(r) \cap D(p_{req}, x) \neq \varnothing\big)\Big)$ do
 2:     $l(r, vm) \leftarrow measure(latency(r, vm), timeout)$
 3:     $BW(r, vm) \leftarrow 0$
 4: end for
 5: $transfersize \leftarrow pkgsize(D(p_{req}, 0)) - pkgsize(\{p_\mu\})$
 6: $uptimeout \leftarrow transfersize/awaitedbw$
 7: $R' \leftarrow R_\varphi$
 8: $bwmeasures \leftarrow \tau\frac{transfersize}{maxtestsize}$
 9: for $i = 0$ to $bwmeasures$ do
10:     $r_{ml} \leftarrow r \in R' : \big(l(r, vm) = \min_{r_y \in R'} l(r_y, vm)\big)$
11:     $R' \leftarrow R' \backslash r_{ml}$
12:     $P_m \leftarrow \Big\{\forall p \in contents(r_{ml}) : \big(pkgsize(\{p\}) < maxtestsize\big)\Big\}$
13:     $p_{dummy} \leftarrow p \in P_m : \big(pkgsize(\{p\}) = \max_{p_x \in P_m} pkgsize(\{p_x\})\big)$
14:     $BW(r_{ml}, vm) \leftarrow measure(downloadbw(p_{dummy}, r_{ml}), uptimeout)$
15: end for
```

requests the MMVA to download the entire construction path ($D_{con}(p_{req}, vm)$) of the service package. Later, the MMVA rebuilds the service package according to the requests of the deployment client (see step 4). Finally, the deployment client restarts the virtual machine in order to activate the now rebuilt service.

In order to determine the rebuilding path, the deployment client uses a similar approach to the IaaS based rebuilding strategy discussed in 6.3.3. As opposed to the IaaS based strategy, the deployment client cannot store any connectivity history for the new virtual machine ($vm = initVM(p_\mu, \varphi)$) and the different repositories ($r \in R_\varphi$). Therefore, the connectivity data is estimated with Algorithm 7.2. The developer or the administrator of the deployment client can customize the behavior of the presented algorithm with the following constraints: (i) the maximum acceptable latency ($timeout$), (ii) the maximum amount of data used for bandwidth measurements (called the *pre-transfer measurement threshold* – τ), (iii) the minimal bandwidth between

	IaaS	Repository	Client	
			alone	MMVA
Measurements	prior	prior	during	during
No IaaS change	–	√	√	√
No repository change	√	–	√	√
Arbitrary VAs	√	√	√	–
Transfers	1	$1+E$	$3+\tau$	$1+\tau+M$

Table 7.1: Comparison of the introduced rebuilding scenarios

the repository and the virtual machine (*awaitedbw*) and finally, (*iv*) the maximum size of the package used for bandwidth estimation (*maxtestsize*).

Using this algorithm, the deployment client measures the latencies of the various repositories – $l(r, vm)$. Next, it estimates the number of bandwidth measurements (*bwmeasures*) the algorithm can make before reaching the pre-transfer measurement threshold. Then selects the repositories ($r_{ml} \in (R_\varphi \backslash R')$) with the lowest latencies to measure their bandwidth towards the host of the MMVA. Using these measurements all the data is available for the deployment client to estimate the construction path ($D_{con}(p_{req}, vm)$) that would have been chosen by the IaaS system (see Equations 6.25 and 6.26). As opposed to the IaaS based rebuilding solution, this algorithm can only use estimated construction paths because the connection details are not available for all the repositories in the service-based system (R_φ).

The algorithm is heavily dependent on the value of the *pre-transfer measurement threshold* (τ) that describes the amount of data used for bandwidth measurements expressed relative to the size of the required package (p_{req}). On one hand, values closer to zero reduce the amount of measurements the algorithm can make to estimate the bandwidth between the deployed MMVA and the various repositories in the system. On the other hand, increasing the value could reduce the effectiveness of the algorithm because it scarifies more time on finding the most suitable repository to download the required package from. Based on my initial experiments, the value of 0.1 offers a good balance between the two extremes. Several example deployments can be seen with this value in Section 10.3.

7.3.1.1 *Comparison of the rebuilding scenarios*

Table 7.1 compares the previously detailed rebuilding scenarios (see Sections 6.3.1 and 7.3.1) from five point of views: (*i*) "Measurements" denote if the system takes the measurements independently from the rebuilding process; (*ii*) "No IaaS change" depicts the approaches that does not require changes in the available IaaS systems for their operation; similarly (*iii*) "No repository change" describes the solutions independent of the repository implementation; (*iv*) "Arbitrary VAs" present if an algorithm is not dependent on a specific type of virtual appliance; finally, (*v*) "Transfers" represent the amount of data (expressed relative to the size of the appliance: $reltsize := tsize/pkgsize(D(p_{req},0))$) required for the reconstruction of the VA on the execution host.

Analyzing the table reveals that the IaaS and active repository based solutions can take measurements prior the actual rebuilding process starts, therefore, they do not require extra transfers to determine the latency and bandwidth values. These systems can measure and monitor the connection properties for previous transfers between the different parties of the service-based system. As an opposite, deployment client based solutions use extra transfers to estimate the connection properties just before the rebuilding takes place. However, according to Algorithm 7.2, these extra transfers are limited by the pre-transfer measurement threshold value (τ).

The final comparison of Table 7.1 is based on the total size of transfers. The table unveils the superiority of the IaaS based solution, where the virtual appliance is transferred directly to the execution host. Active repositories lose to IaaS based solutions because they have to transfer the external packages first to the repository (r_{req}) of the requested package (p_{req}) then, after rebuilding, the entire virtual appliance to the IaaS system. I have identified the size of the required transfers for active repositories ($tsize_{AR}, reltsize_{AR}$) as follows:

$$tsize_{AR} := \overbrace{pkgsize(ext(D(p_{req},n),r_{req}))}^{\text{transfer externals}} + \overbrace{pkgsize(D(p_{req},n)}^{\text{transfer rebuilt}}$$

$$reltsize_{AR} := 1 + \frac{pkgsize(ext(D(p_{req},n),r_{req}))}{pkgsize(D(p_{req},n))}$$

consequently:

$$E = \frac{pkgsize(ext(D(p_{req},n),r_{req}))}{pkgsize(D(p_{req},n))} \tag{7.3}$$

Deployment clients introduced in Figure 6.6c first estimate connection properties (τ), and then download all packages for rebuilding on the client's host (this is the first time the entire appliance is transferred). Later, the rebuilt appliance is published (second transfer from the client to a repository). Finally, the IaaS is requested to instantiate the published appliance, resulting the third and final transfer during deployment. As a result, I have defined the size of the required transfers for deployment clients ($tsize_{DC}, reltsize_{DC}$) with the following equations:

$$\begin{aligned} tsize_{DC} &= \overbrace{\tau \cdot pkgsize(D(p_{req}, n))}^{\text{measure}} + \overbrace{\sum_{p \in D(p_{req}, n)} pkgsize(\{p\})}^{\text{rebuild}} \\ &+ \underbrace{pkgsize(D(p_{req}, n))}_{\text{publish}} + \underbrace{pkgsize(D(p_{req}, n))}_{\text{deploy}} \end{aligned} \tag{7.4}$$

$$reltsize_{DC} = \tau + 3 \tag{7.5}$$

As an opposite, deployment clients utilizing MMVAs (p_μ) first instantiate the MMVA (the entire MMVA is transferred), then estimate connection properties using the MMVAs management interfaces (τ). Finally, after the client evaluates the optimal reconstruction path, the MMVA is requested to install the packages according to $D_{con}(p_{req}, vm)$. The size of the required transfers ($tsize_{VM}, reltsize_{VM}$) by the MMVA based rebuilding solution is defined as:

$$\begin{aligned} tsize_{VM} &= \overbrace{pkgsize(\{p_\mu\})}^{\text{init}} + \overbrace{\tau \cdot pkgsize(D(p_{req}, n))}^{\text{measure}} \\ &\underbrace{+ pkgsize(D(p_{req}, n)) - \sum_{i \in (items(p_\mu) \cap items(p^*_{\Omega,nm}))} size(i)}_{\text{reconstruct}} \\ reltsize_{VM} &= 1 + \tau + \frac{pkgsize(\{p_\mu\}) - \sum_{i \in (items(p_\mu) \cap items(p^*_{\Omega,nm}))} size(i)}{pkgsize(D(p_{req}, n))} \\ &\text{therefore:} \\ M &= \frac{pkgsize(\{p_\mu\}) - \sum_{i \in (items(p_\mu) \cap items(p^*_{\Omega,nm}))} size(i)}{pkgsize(D(p_{req}, n))} \end{aligned} \tag{7.6}$$

Where M represents the data that would not be transferred without the use of the MMVA, because, the non-manageable original virtual appliance ($p^*_{\Omega,nm}$) did not use them.

7.3.1.2 *Online reconstruction strategy*

I have extended the rebuilding algorithm to include two *virtual appliance reconstruction* strategies that are applied either before (offline) or after (online) the appliance's hosting virtual machine is initiated. The algorithm decides on the strategy based on the capabilities of the base virtual appliance. If the base appliance offers management interfaces to install, configure and update

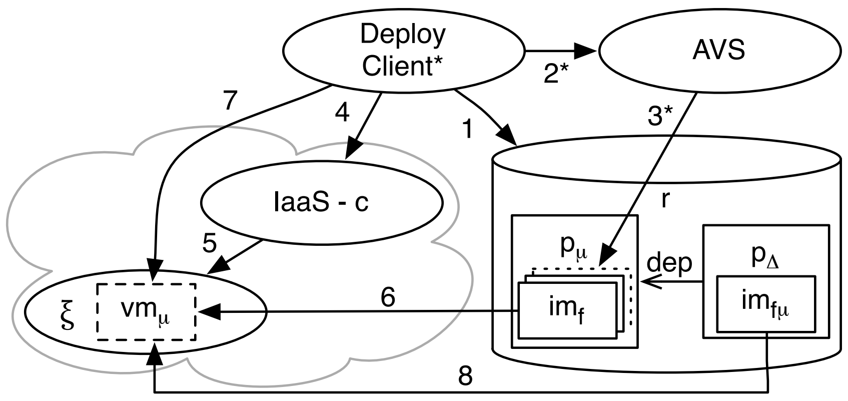


Figure 7.4: Transformation applied only on MMVAs

components of a virtual machine then the algorithm chooses the *online reconstruction strategy*, otherwise it uses the *offline reconstruction strategy*. The offline strategy was discussed in Section 6.3.4.1.

The *online reconstruction strategy* follows a new approach that requires the base virtual appliance ($p_\beta \in D_{con}(p_{req}, vm)$) to comply with the requirement of exposing management interfaces (e.g. its image contains an embedded MMVA – $manageable(p_\beta) = true$). This base virtual appliance is used to initiate a virtual machine ($vm = initVM(p_\beta, \varphi)$) that is transformed by the new reconstruction strategy to host the original virtual appliance. The rebuilding algorithm uses the management interfaces to install the required delta packages according to the reconstruction path ($D_{con}(p_{req}, vm)$) on the running virtual machine. Therefore, most of the installation task is executed within the virtual machine that is going to host the deployed service by the end of this procedure.

7.3.2 *MMVA based virtual appliance transformation*

When an IaaS system receives a deployment request for a virtual appliance (p_Δ) that is stored in a non-supported format, then an appliance transformation should be applied before the deployment could take place. I have introduced three basic transformation techniques in Section 4.3. This section introduces a new method for handling virtual appliance format transformation.

If a stored virtual appliance is based on a *minimal manageable virtual appliance*, then only the MMVA have to be transformed (see Figure 7.4). When the transformed MMVA is instantiated, the differences between the stored virtual appliances disappear because of the virtualized hardware (see Section 4.4.1.3). Therefore, the system stores all delta repository packages solely

	IaaS	Repository	Client	MMVA
User transparency	√	√	√	√
No IaaS change	–	~	√	√
Minimized storage	–	–	–	√
Arbitrary VAs	√	√	√	–

Table 7.2: Comparison of all proposed transformation initiators

in an appliance format agonistic way. However, it is required that one of the MMVAs with the *installPackage* interface should be able to handle the image format (f_μ) of the delta packages.

In case a deployment request is received for an MMVA based appliance, then as seen in step 1 in Figure 7.4, the system checks whether one of the images of the MMVA (p_μ) can be used with the target IaaS system – c. If there is no matching image ($pkgforms(p_\mu) \cup VAforms(c) = \varnothing$), then in steps 2* and 3*, the system initiates the transformation on the MMVA only – for details see the deployment client initiated transformation scenario in Section 4.3. To lower the need for transformations before deployment, the AVS checks for MMVAs in the repository and automatically transforms them to all the frequently used appliance formats.

If there is an image of the MMVA that is in the desired format, then the system instantiates a virtual machine ($vm_\mu \in hosted(\xi) : (\xi \in \Xi_\varphi(c))$) in the IaaS system for p_μ – see steps 4-6 in Figure 7.4. Next, in steps 7-8, the deployment client uses the management interfaces of vm_μ to install one of the dependency sets of p_Δ inside the newly instantiated VM.

Table 7.2 compares the MMVA based transformation to the previously discussed basic options. Previously identified transformation approaches required to store the virtual appliances in their IaaS system specific format even after decomposition. Consequently, previous approaches store the appliances in multiple instances if they need to be stored in multiple appliance formats. In contrast, minimal manageable virtual appliances allow the use of VMM format agonistic packages according to Section 4.4.1.2. As a result, only the MMVAs have to be stored in the various appliance formats of the service-based system (F). Consequently, MMVA based transformation *minimizes* the storage requirements of the AVS according to the following example:

$$\begin{aligned}
p_1 &= p \in P_\varphi : (mmva(p) = true)\\
F_{p_1} &= \{\forall f \in F : (\exists im(p_1, f))\}\\
p_2 &= p \in P_\varphi : \Big(\nexists z < PC(p) : (manageable(\sum_{p_x \in D(p,z)} p_x) = true)\Big)\\
F_{p_2} &= \{\forall f \in F : (\exists im(p_2, f))\}
\end{aligned} \tag{7.7}$$

$$\begin{aligned} p_3 &= p \in P_\varphi : \big(p_1 \in D(p,x) : (x < PC(p))\big) \\ stsize(p_3) &= (|F_{p_1}| - 1) pkgsize(\{p_1\}) + pkgsize(D(p_3,x)) \\ stsize(p_2) &= |F_{p_2}| pkgsize(D(p_2,1)) \end{aligned}$$

Where $stsize : P_\varphi \to \mathbb{N}$ depicts the storage size requirements of a given package according to the multiple images stored within the package and its dependencies. As depicted in the definition of p_3 in the previous equation, the MMVA based solution cannot function on *arbitrary virtual appliances*, because this transformation technique cannot be applied to non-manageable virtual appliances (e.g. p_2 in the previous example).

7.3.3 *MMVA and the Optimization facility*

The optimization facility (introduced in Chapter 5) increases its efficiency if there is an embedded minimal manageable virtual appliance in the VA under optimization. When the facility meets such an appliance then it enables a new removal algorithm and a different virtual machine management strategy. The following sections discuss the differences between the already discussed optimization process and the one based on the use of MMVAs.

7.3.3.1 *Removal during virtual machine execution*

First, an embedded MMVA enables a new removal technique, called *removal during execution*. This approach starts up the original virtual appliance before the removal action takes place. Then, removes the highest weighted items or groups by utilizing the management capabilities of the virtual machine under optimization. Consequently, the reduced virtual appliance (p_{red}) is created right inside its hosting virtual machine.

This technique removes contents from already instantiated virtual machines. Therefore, this removal technique could optimize the virtual appliances until they would never boot again (because the memory of such systems contains some of the removed contents). However, this requires storing not just the disk state of the virtual machine but the virtual appliance should also include the entire memory state of the virtual machine. Therefore, this algorithm is not optimal, unless the management interfaces plus the saved memory state is smaller than the overall size of all the removed items.

The proposed technique avoids this problem by restarting the virtual machine after the removal action. This step simulates the boot procedure of a virtual machine with the reduced appliance. If the restart is unsuccessful the validation automatically fails, otherwise the validation algorithms are executed using this already running virtual machine. As a result, this technique cancels effects of the virtual machine's memory.

The issue with this new removal technique is the requirement of the embedded MMVA in the original virtual appliance. As a result, after the optimization finishes, the reduced virtual appliance still offers the management interfaces among the target functionality. To reach the optimal size the facility offers the appliance developer the option to remove the MMVA functionality and their dependencies with the pre-execution method (for details see Section 5.2.2). However, this option is only recommended for use in appliance based deployment systems not capable of utilizing the embedded MMVA (e.g. in systems without online virtual appliance reconstruction capabilities).

The algorithm also offers two major advantages over *pre-execution removal*:

- This approach is the only realistic solution in IaaS systems (see Section 2.1) similar to Amazon EC2 [50] where the sole virtual appliance source is their internal repository (e.g. S3 in case of Amazon). With this approach, the removal of an item or group does not require the system to publish the appliance with the removed contents for the time of the validation task.

- Removal during execution also enables the use of a different virtual machine management strategy that does not terminate the virtual machines after every validation (introduced in the following section).

7.3.3.2 *MMVA based Virtual Machine Management Strategy*

An embedded MMVA enables the repeated use of the removal algorithm on a single virtual machine, because, virtual machines that successfully pass the validation phase can be reused. As a result, the optimization facility does not need to initiate new virtual appliances for every validation. The following virtual machine management strategy manages the virtual machine instances and allows their reuse.

In contrast to the previous strategy (introduced in Section 5.2.3), this new strategy initializes pool of virtual machines with the same partially optimized virtual appliance. Then the MMVA based strategy uses the previously introduced removal algorithm to remove one of the highest weighted items or groups from the running VM and prepare it for validation. Therefore, the tasks of the new management strategy do not stop at preparing the VMs for validation.

Figure 7.5 presents the three main tasks (allocate, recover or synchronize VMs) during the management of available virtual machines for the optimization system. The first task is to *allocate* as many VMs as possible for the optimization (the actions of this task are circled with a dashed line in Figure 7.5). Each acquired virtual machine is instantiated with the appliance under optimization. Then the appliances are configured to run inside their

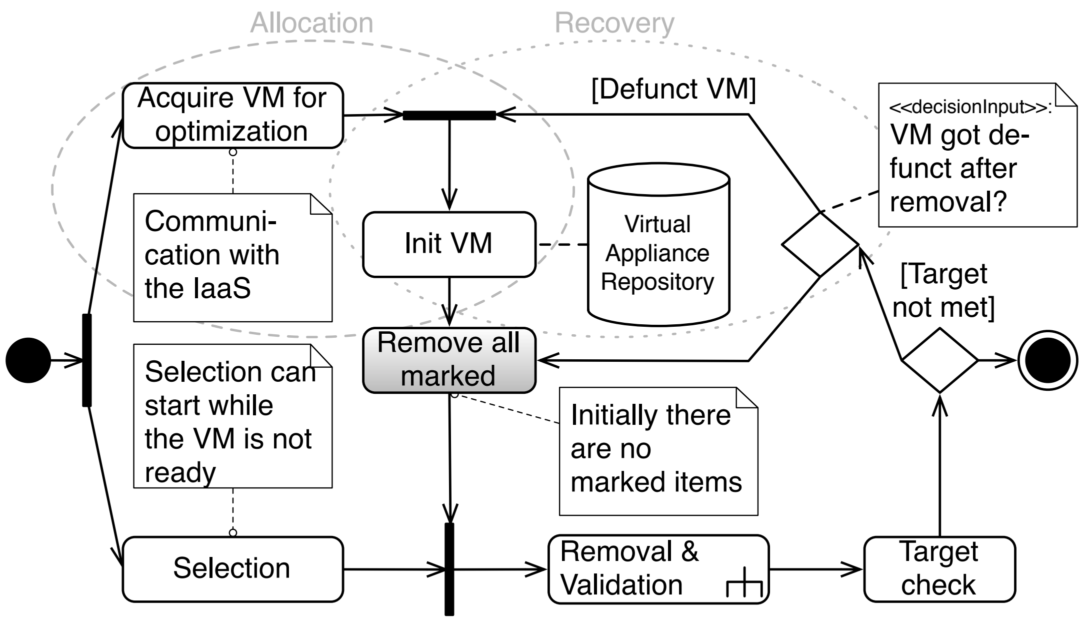

Allocation
Recovery
Acquire VM for optimization
[Defunct VM]
<<decisionInput>>: VM got defunct after removal?
Communication with the IaaS
Init VM
Virtual Appliance Repository
[Target not met]
Remove all marked
Selection can start while the VM is not ready
Initially there are no marked items
Selection
Removal & Validation
Target check

Figure 7.5: Handling virtual machine instances of MMVA enabled virtual appliances

hosting VMs (as discussed in Section 5.2.3). After the VMs are initialized, they are ready to be used for the removal and validation tasks. The initial allocation of the available virtual machines is executed in parallel with the itemization, item pooling, grouping and weighting process (see Figure 5.2).

If the optimization facility detects an unsuccessful validation, then the affected virtual machine becomes defunct. This leads towards the second task of the management. The recovery of the defunct VM could imply the addition of the previously removed content. However, this step would require the revalidation of the entire restored virtual machine. Therefore, instead of trying to heal the defunct VM, the manager uses the following *recovery* strategy (the actions of the strategy are circled with a dotted line in Figure 7.5). It simply terminates the current instance and initiates a new VM with the appliance in it. Then the manager synchronizes the previously successfully removed content in the initialized VM. During recovery, the parallel validation branches compete with each other for the available VMs.

Instead of the intermediate virtual appliance creation task of the previous management strategy, this strategy solves the content unification of the different virtual machines with the synchronization of the removed content. After each optimization iteration, the final task of the manager synchronizes the content of all the available VMs with the *remove all marked* operation (marked with a gray box in Figure 7.5). Each of these virtual machines has one of the previously selected items or groups removed from them. To allow the next iteration of the validation procedure the manager synchronizes all still func-

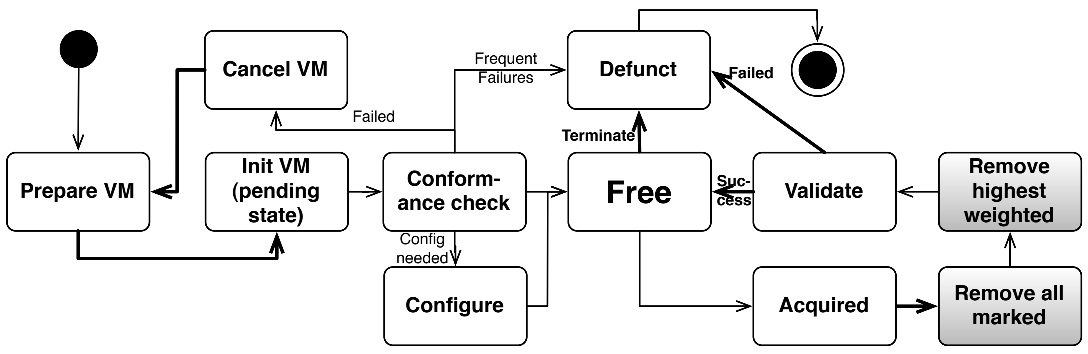


Figure 7.6: Virtual machine management states with virtual appliances embedding an MMVA

tional VMs by removing the successfully validated items or groups from them.

Because of these three new tasks, the strategy based on the use of management interfaces consumes less bandwidth and utilizes smaller amount of virtual machines during the entire optimization process.

INDIVIDUAL VIRTUAL MACHINE HANDLING. Because of minimal manageable virtual appliances, individual virtual machines are also handled differently compared to the original algorithm introduced in Section 5.2.3.1. Figure 7.6 defines the list of virtual machine management states a virtual machine passes through during its extended lifecycle. These states are highly similar to the ones previously introduced. Therefore, I have highlighted the differences with bold arrows and gray states. This section only discusses the differences; for the non-highlighted actions please have a look at Section 5.2.3.1.

Virtual machines in the pool are created in two cases. First, when optimization facility has just started the optimization and the initial pool of virtual machines are under construction. Second, one of the virtual machines in the pool become *defunct* and a replacement has to be created by the virtual machine handler.

If a virtual machine passes to the *initialization* state, then the IaaS was requested to create a virtual machine using the original virtual appliance (p_{new}). The use of the original appliance avoids the necessity to publish intermediate virtual appliances in the repository.

After the *free* state of the virtual machine, all states represent a phase in the validation process. First, the *acquired* state designates the VM's participation in the procedure. During this phase the virtual machine handler generates the list of removal requests for the highest weighted removable items or groups. Then the contents of the VM are synchronized with the other VMs in the pool, while it is in *remove all marked* state. As a result, the VM will represent the intermediate virtual appliance. Before the actual validation takes place, the VM handler arranges the execution of the removal requests that were generated during the acquired state. The VM stays in the *remove highest weighted* state while the actual virtual machine executes the generated removal requests.

Finally, the VM enters the actual *validation* state that starts by checking the success of the VM synchronization and the success of the requests for the removal of the highest weighted items. Then the virtual machine handler restarts the VM. Next, the handler evaluates the validation algorithms in the slightly modified service instance. If both the evaluation of the validation algorithms and the removal checks were successful then the virtual machine becomes free again. Otherwise, the VM gets *defunct* and the handler initiates the creation of a new VM replacing the defunct one.

CHAPTER SUMMARY. This chapter provided three possible options for defining management interfaces on appliances (basic, combined, advanced) and came to the conclusion that combined interfaces are the most suitable to support the AVS functionality. Then the chapter defined an algorithm for checking the management capability of an existing appliance. Based on this algorithm I have introduced a technique to create minimal manageable virtual appliances using the optimization facility (my second contribution). The chapter then offered two approaches for extending already existing or planned appliances with management interfaces using the previously defined MMVAs. Finally, the chapter offered an analysis on the effects of applying the MMVA concepts to the rebuilding, transformation and size optimization techniques introduced in the previous chapters of this work.

Part III

ANALYSIS

8 METHODOLOGY

CHAPTER OVERVIEW. This chapter offers the grounds for evaluating the findings and contributions of this thesis. It provides the evaluation scenarios and use cases that are used in later Chapters. These scenarios are defined so they can directly support my contributions. It also outlines the dependencies and requirements about the evaluation system and testbed that can support the execution of the evaluation scenarios.

8.1 INTRODUCTION

Throughout this chapter, I intend to introduce the evaluation methodology of the proposed architecture. This methodology investigates the following four main areas:

CORRECTNESS. Independently from the applied optimization and active repository functionalities, the AVS architecture should always create virtual appliance packages that after rebuilding behave as the original service of the appliance developer. Detailed in Section 8.3.

INFRASTRUCTURE INDEPENDENCE. The AVS architecture and its effects on deployment time should be independent from particular infrastructures. Discussed in Section 8.3.

DEPLOYMENT EFFICIENCY. The virtual appliance packages created by the architecture should result in faster deployment times for the encapsulated services. Introduced in Section 8.4.

COST. The cost of the application of the proposed techniques should be lower than the estimated gain of future faster deployments. For details, see Section 8.5.

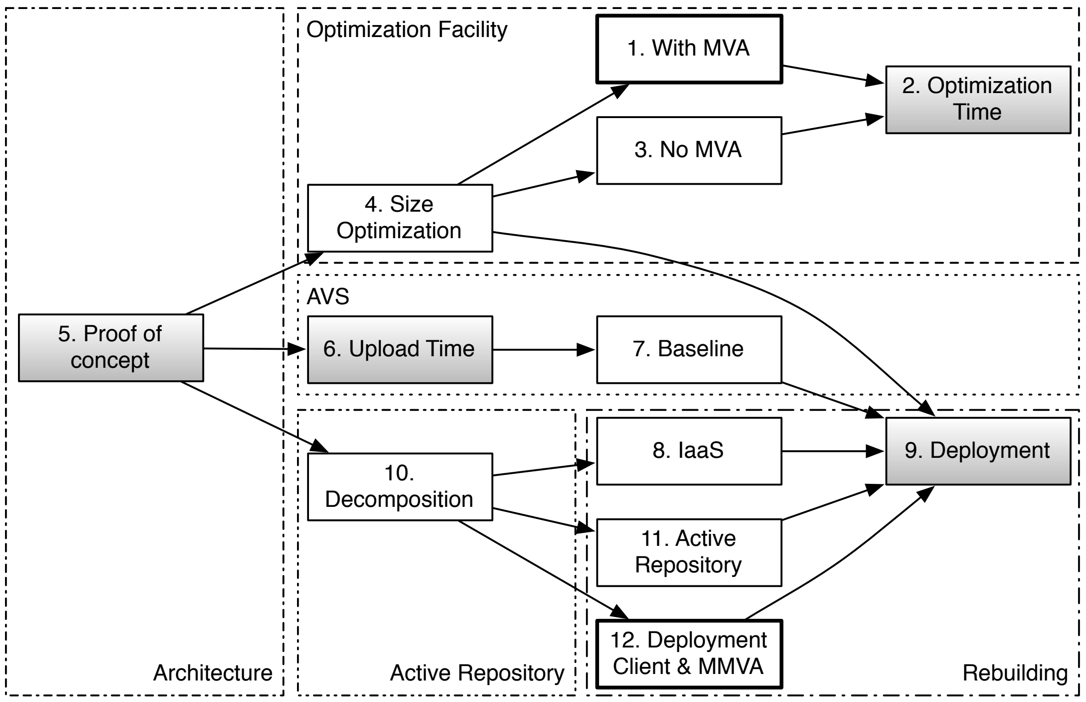


Figure 8.1: Classification and relationship of the evaluation scenarios

Figure 8.1 shows the relations of the evaluation scenarios with each other and with the proposed architecture. The figure depicts the four basic measurement groups with gray colored boxes: (*i*) the *proof of concept* checks the correctness of the proposed architecture; then (*ii*) *upload time* checks the efficiency of the initial upload algorithm discussed in Algorithm 4.1; next, (*iii*) *optimization time* measures the required time to reach the $optimalsize(p_\sigma)$ criterion on a service package – p_σ; finally, (*iv*) the effectiveness of *deployment* is measured with various rebuilding and optimization scenarios compared to the *baseline* deployment time measurement made right after measuring the initial upload time.

Figure 8.1 represents the various measurements to evaluate the architecture with the numbered boxes (later the measurements are referred by their numbers). The dependencies of the different measurements are depicted with arrows (the direction of the arrows reveal the dependent scenarios). By circling the various measurement options, the figure also presents those components of the architecture (the optimization facility, AVS, active repository and rebuilding algorithms) that are evaluated by a given measurement (see the dashed and dotted lines around the measurement boxes). The effects of the minimal manageable virtual appliance are targeted on the optimization facility and the rebuilding; therefore, the MMVA related measurements are depicted with bold boxes.

Throughout this chapter, measurements are represented with the function $measure : \mathcal{F} \to \mathbb{R}$. This function evaluates its arguments ($\mathcal{F}$ that stands for an arbitrary function and a specific parameter set) repeatedly and measures the execution time (t_{ev}) for each individual evaluation. Measurements are executed until the sample standard deviation (s_N) of the t_{ev} values becomes stable, thus the value of two subsequent standard deviation calculations are within 1%:

$$\frac{s_N(t_{ev}) - s_{N+1}(t_{ev})}{s_N(t_{ev})} < 0.01 \text{ where } N \geq 2 \tag{8.1}$$

After reaching a stable deviation value, the measure function returns with the median of all the measured execution time values.

In the next sections, I present the appliance classes considered during the evaluation scenarios. Then, I introduce the evaluation of correctness and efficiency respectively. The methodology introduced in this chapter supports the evaluation of the proposed architecture. First, in Chapter 9, I discuss the use of the correctness criteria (defined in Equation 8.6) to assess the infrastructure independence of the AVS services. Afterwards, in Chapter 10, I use deployment efficiency (see Section 8.4) and cost (see Section 8.5) estimation to present the positive effects of the proposed architecture on IaaS systems.

8.2 APPLIANCE CLASSIFICATION

The different evaluation scenarios require different virtual appliances. In order to allow the generic definition of the scenarios, I have investigated several virtual appliances and identified a classification that can be used to specify generic requirements. Based on the required network and service configuration I have classified virtual appliances into three complexity classes:

MINIMAL COMPLEXITY appliances cannot be decomposed into more than two packages ($|D(p)| = 2$): the MMVA and the service package. The service package holds a single executable that incorporates the target functionality for the users; this functionality is accessed through the *execution* interface of the MMVAs. Consequently, this service instance requires the same network configuration as the MMVA service. *Examples:* SSH (see Section A.1), TINKER [46] appliances.

MEDIUM COMPLEXITY appliances are built on multiple packages – $|D(p)| \geq 2$. These appliances do not need to incorporate an MMVA, because their service packages offer the activation of their target functionality after the instantiation of the appliance in a virtual machine. After the deployment of these virtual appliances, their target functionality is externally

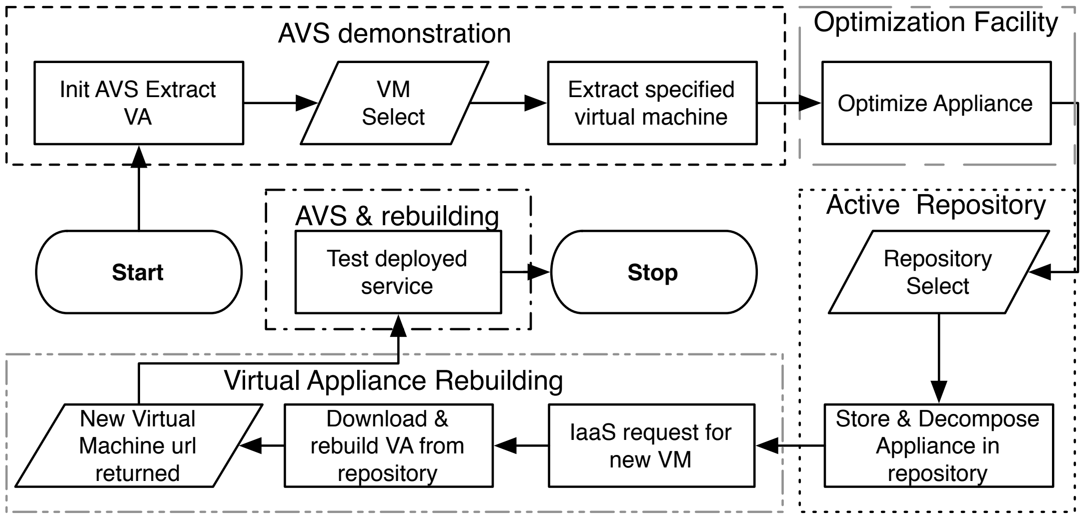


Figure 8.2: Correctness checking scenario of the architecture

available through appliance specific interfaces. As a result, the network configuration of these service instances needs to take into consideration of the appliance specific interfaces. *Examples:* Apache web server [67] (see Section A.2), LAMP (Linux, Apache, MySQL, PHP) projects [84].

HIGH COMPLEXITY appliances are composed of multiple delta packages after decomposition – $|D(p)| > 2$. After deployment, these appliances require configuration because they depend on externally available services. On top of the network requirements of the previous classes, the network configuration of these service instances also expects the connectivity towards their external dependencies. *Examples:* GEMLCA service [22] (see Section A.3), Gridway broker [60].

8.3 CORRECTNESS OF THE ARCHITECTURE

Figure 8.2 presents the *proof of concept* (see *measurement 5* in Figure 8.1) evaluation scenario that utilizes all major functionalities of the proposed architecture. With the requirement set below, this scenario ensures that there is no such package constellation that could result an invalid virtual appliance after applying the various techniques introduced with the architecture.

Before discussing the scenario itself, the following paragraphs list the requirements against a particular testbed infrastructure (φ). First, the scenario requires the following minimal set of repositories in the infrastructure that should store the below detailed contents:

$$\begin{aligned}
R_\varphi &:= \{r_1, r_2, r_3\} \\
P_\varphi &:= \{p_\mu, p_1, p_3\} \\
contents(r_1) &:= \{p_3\} \\
contents(r_2) &:= \{p_1\} \\
contents(r_3) &:= \{p_\mu\} \\
\Theta(p_1) &:= \{\{p_1, p_\mu\}\} \\
mmva(p_\mu) &:= true
\end{aligned} \tag{8.2}$$

For the demonstration of the architecture, a *highly complex web service* has to be selected. This web service will be extracted (to the initial self-contained package p_2, where $selfcontained(p_2) = true \wedge va(p_2) = true \wedge servicepkg(p_2) = true$). The creation of the initial self-contained package is depicted with the operation "*Extract specified virtual machine*" in Figure 8.2. Then, p_2 will be minimized with the size optimization facility until it reaches its optimal size (in the package that I later refer to as p_2', where $selfcontained(p_2') = true \wedge va(p_2') \wedge servicepkg(p_2') = true \wedge optimalsize(p_2') = true$). This phase of the demonstration is depicted as "*Optimize Appliance*" in Figure 8.2. In order to enable the demonstration of the initial upload, decomposition and rebuilding algorithms of the architecture, the p_2 has to be prepared so that its optimized appliance (p_2') has common items ($I_{2,1}, I_{2,\mu} \in I_\varphi$) with the previously defined contents of the repositories as follows:

$$\begin{aligned}
I_{2,1} &:= items(p_2') \cap items(p_1) \neq \emptyset \\
I_{2,\mu} &:= items(p_2') \cap items(p_\mu) \neq \emptyset \\
\text{and} \quad & \textstyle\sum_{i_x \in I_{2,1}} size(i_x) > \sum_{i_y \in I_{2,\mu}} size(i_y)
\end{aligned} \tag{8.3}$$

Because of these requirements, the initial upload operation should find repository r_2 more desirable (see step "*Repository select*" in Figure 8.2), and publish the extracted and optimized virtual appliance of the service in there. When the repository receives the optimized package (p_2') it should automatically apply the decomposition algorithm on it resulting the common (p_4) and delta parts (p'_1, p''_2) of the packages as follows:

$$\begin{aligned} contents(r_2) &= \{p_1, p'_1, p''_2, p_4,\} \\ items(p''_2) &= items(p'_2) \backslash items(p_1) \\ items(p_4) &= I_{2,1} \\ items(p'_1) &= items(p_1) \backslash items(p_4) \\ dep(p''_2) &= \{p_4\} \\ dep(p'_1) &= \{p_4\} \\ dep(p_4) &= dep(p_1) = \{p_\mu\} \\ \Theta(p''_2) &= \{\{p''_2, p_4, p_\mu\}\} \end{aligned} \tag{8.4}$$

This step of the proof of concept scenario is revealed as *"Store & Decompose Appliance in repository"* in Figure 8.2.

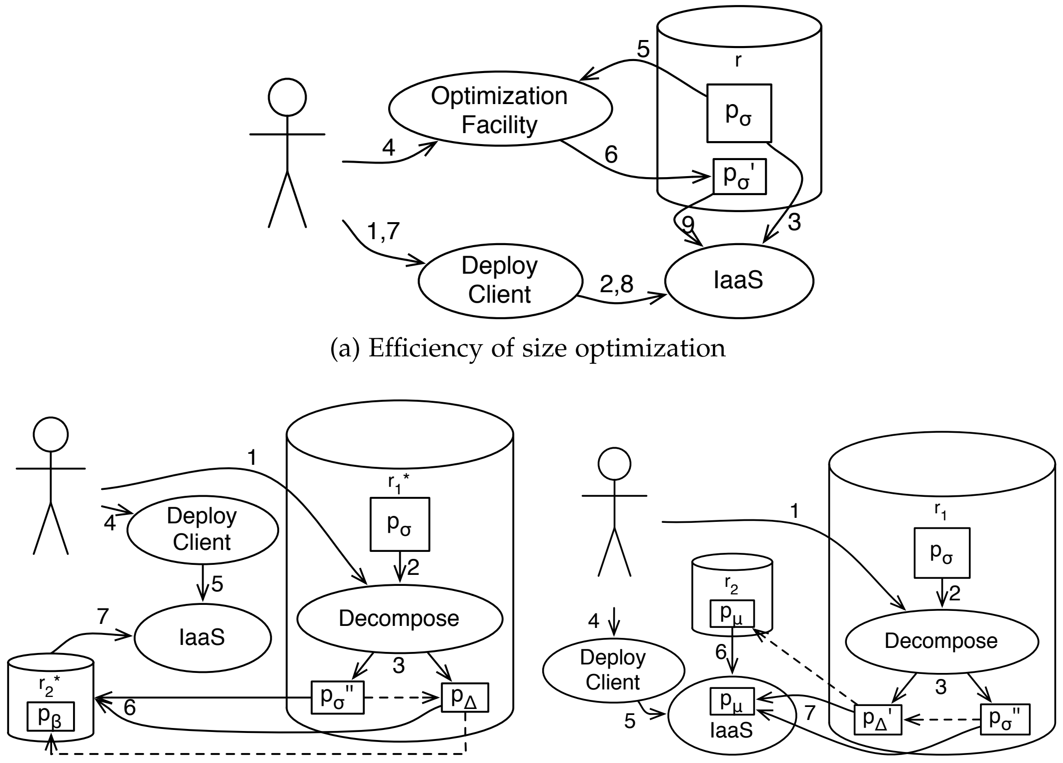
Consequently, the AVS should return the URL of p''_2 to the appliance developer. Using this URL, the validation scenario advances to the deployment phase. At this stage, the deployment client is manually requested to deploy the service package (p''_2) in an IaaS system (see step *"IaaS request for new VM"* in Figure 8.2). As a result, the IaaS system initiates a virtual machine using the minimal manageable virtual appliance package (p_μ) referred by the service package. Then the *online reconstruction strategy* (see Section 7.3.1.2) downloads and configures the dependency set ($D(p''_2) \in \Theta(p''_2)$) inside the newly created virtual machine to install and configure the entire p'''_2 appliance supposedly with the original *highly complex web service*:

$$\begin{aligned} & p_2 \xrightarrow{\text{optimize}} p'_2 \xrightarrow{\text{decompose}} p''_2 \xrightarrow{\text{rebuild}} p'''_2 \\ & p_2 \overset{?}{\equiv} p'''_2 \end{aligned} \tag{8.5}$$

At the end of this procedure the deployment client should return the IP address of the newly created virtual machine with the activated web service running inside. Finally, the correctness ($correct : (P_\varphi, \Phi) \rightarrow true, false$) of the evaluated testbed infrastructure is proven when the validation of the initial virtual appliance (p_2) is successful on the new virtual machine:

$$\begin{aligned} p'''_2 &:= \sum_{p \in D(p''_2, x)} p \\ & \quad \text{where } 0 < x < PC(p''_2) \text{ and } x \in \mathbb{N} \\ correct(p_2, \varphi) &:= valid(p_2, initVM(p'''_2, \varphi)) \end{aligned} \tag{8.6}$$

The successful execution of the entire *proof of concept* evaluation scenario ensures the correctness of the proposed architecture by using all the AVS



(a) Efficiency of size optimization

(b) Efficiency of delivery optimization

(c) MMVA's effect on the efficiency

Figure 8.3: Basic deployment measurements

related functionality: (*i*) the AVS can extract functional virtual appliances – p_2 –, (*ii*) the optimization facility still returns valid optimized appliances – p'_2 –, (*iii*) the decomposition finds the common parts of the different appliances – p_4 – and (*iv*) the decomposed virtual appliances are still functional after rebuilding (p'''_2).

The evaluation of the proof of concept scenario on multiple implementations of the proposed architecture provides the evidence on reaching *infrastructure independence* – for the objectives of the methodology see page 122.

8.4 basic deployment efficiency

This section discusses the measurement group associated with appliance deployments (depicted as *measurement 9* in Figure 8.1). During the evaluation section, I calculate the efficiency of the various optimization algorithms with the speedup function – $S : (P, \Phi) \to \mathbb{R}$:

$$S(p_\sigma, \varphi) = \frac{\overbrace{measure(initVM(p_\sigma, \varphi))}^{\text{baseline}}}{\underbrace{measure(initVM(p'_\sigma, \varphi))}_{\text{optimized}}} \tag{8.7}$$

Where the system calculates the ratio (speedup – $S(p_\sigma, \varphi)$) of the measured deployment times of a service package before (p_σ) and after (p'_σ) the optimization was applied. This speedup value can be used to express the effects of the optimization facility, the decomposition and replication algorithms (distributed delivery optimization) and the MMVA based rebuilding. The currently tested optimization algorithm is successful and the system reached its target for efficient deployments – for the objectives of the methodology see page 122 – if it meets the following requirement:

$$S(p_\sigma, \varphi) > 1 \tag{8.8}$$

8.4.1 *Baseline and post-optimization deployments*

The *baseline* measurement ($measure(initVM(p_\sigma, \varphi))$) for the speedup values is presented in *step 1-3* in Figure 8.3a and in *measurement 7* of Figure 8.1. During this measurement, I used the deployment client to request the IaaS system for the initialization of the p_σ virtual appliance in a virtual machine.

To evaluate speedup values the testbed system has to be prepared specifically for the optimization algorithm. Therefore, Figure 8.3 reveals the testbed setup and evaluation scenario for the three basic efficiency measurements – $measure(initVM(p'_\sigma, \varphi))$.

Figure 8.3a displays that after the baseline measurement has completed, the optimization facility is ordered to optimize the original virtual appliance (see *step 4*). Then the optimization facility creates a virtual appliance playground (see Section 4.2.2) for p_σ in *step 5*. Next, the resulting package (p'_σ) of the optimization algorithm (see Section 5.2) is stored in the repository (see *step 6*). Finally, the testbed is ready to evaluate the speedup of the optimization by measuring the deployment time of p'_σ as in *steps 7-9*. This measurement is also depicted in the Figure 8.1 as *measurement 4* and it is independent from the complexity of the initial appliance (p_σ).

8.4.2 *Deployments with rebuilding*

Figure 8.3b highlights the two-phased evaluation of the decomposition and rebuilding algorithms. This figure represents *measurements 8, 10* and *11* in Figure 8.1. In order to measure the efficiency of the rebuilding algorithm this evaluation scenario requires a virtual appliance (p_σ) that can be decomposed and rebuilt throughout the scenario. Therefore, the encapsulated service in p_σ has to form a *medium complexity* appliance and should meet the following minimal requirements: $servicepkg(p_\sigma) = true \wedge selfcontained(p_\sigma) = true \wedge va(p_\sigma) = true$.

In order to allow the decomposition of p_σ the measurement requires at least two extra packages ($p_{E,1}, p_{E,2}$) to be present in the repository where the p_σ is going to be uploaded. These packages are needed to allow the decomposition of the p_σ into three pieces. The contents of the various packages are not relevant for the current measurements as long as they are related to each other as specified by these requirements:

$$\begin{aligned} items(p_\beta) &:= items(p_{E,1}) \cap items(p_{E,2}) \neq \emptyset \\ items(p_\beta) &\subset items(p_\sigma) \\ baseva(p_\beta) &= true \\ items(p_\Delta) &= (items(p_{E,1}) \cap items(p_\sigma)) \setminus items(p_\beta) \neq \emptyset \end{aligned} \tag{8.9}$$

Next, before starting the measurements, the extra packages have to be published in the available repositories (see r_1^* and r_2^* in Figure 8.3b). Their publication will result their decomposition in the repositories as follows:

$$\begin{aligned} contents(r_1^*) &:= \{p_{E,1}, p_{E,2}, p_\beta, p'_{E,1}, p'_{E,2}\} \\ contents(r_2^*) &:= \{p_{E,1}, p_{E,2}, p_\beta, p'_{E,1}, p'_{E,2}\} \\ &\text{where} \\ items(p'_{E,1}) &:= items(p_{E,1}) \setminus items(p_\beta) \\ dep(p'_{E,1}) &:= \{p_\beta\} \\ items(p'_{E,2}) &:= items(p_{E,2}) \setminus items(p_\beta) \\ dep(p'_{E,2}) &:= \{p_\beta\} \end{aligned} \tag{8.10}$$

Before taking the actual measurements, the last step (shown as *step 1* in Figure 8.3b) of the preparations requires the publication of p_σ in repository r_1. The figure shows the packages created as a result of the decomposition (*steps 2-3*) focusing only on the packages relevant for the rebuilding of

p_σ. In the following equation, I list all the packages that are present at the final stage of the measurement:

$$\begin{aligned}
contents(r_1^*) &:= \{p_\sigma, p'_\sigma, p_\Delta, p''_{E,1}, p_{E,1}, p_{E,2}, p_\beta, p'_{E,1}, p'_{E,2}\}\\
contents(r_2^*) &:= \{p_{E,1}, p_{E,2}, p_\beta, p'_{E,1}, p'_{E,2}\}\\
items(p''_{E,1}) &:= items(p'_{E,1}) \backslash items(p_\Delta)\\
items(p'_\sigma) &:= items(p_\sigma) \backslash (items(p_\Delta) \cap items(p_\beta))\\
dep(p''_{E,1}) &:= \{p_\Delta\}\\
dep(p'_\sigma) &:= \{p_\Delta\}\\
dep(p_\Delta) &:= \{p_\beta\}
\end{aligned} \tag{8.11}$$

Finally, the rebuilding measurements can take place after initiating the deployment request for p'_σ in *steps 4-5*. As a result, the system selects the rebuilding location (see *step 6*) according to Figure 6.7 and rebuilds the original medium complexity virtual appliance. Then, the measurement completes after the requested service package gets instantiated in a virtual machine in *step 7*.

8.4.3 *Deployments utilizing the MMVA*

Figure 8.3c represents the evaluation scenario when the deployment client manages the rebuilding of the decomposed virtual appliances with the help of an MMVA. Figure 8.1 depicts this scenario with the *measurement*s *10* and *12*. This scenario requires that the base virtual appliance of the original virtual appliance should be a minimal manageable virtual appliance (this special appliance is presented in the Figure as p_μ). The decomposition requirements of this scenario are not different from the previously discussed Equations 8.9 and 8.11. However, the rebuilding (shown as step 4-7 in Figure 8.3c) is handled by the deployment client through an initially deployed p_μ according to the online reconstruction strategy introduced in Section 7.3.1.

8.5 ESTIMATING THE COST OF THE ARCHITECTURE

The active repository functionality is executed during the idle time of the repositories. As a result, its cost is negligible for the appliance creators. Therefore, I only define the cost of the proposed architecture while the appliance developer initially uploads or optimizes the size of a virtual appliance. These algorithms are evaluated with *measurements 1, 2, 3 and 6* in Figure 8.1 and targeted at the cost of applying the architecture – for the objectives of the methodology see page 122.

8.5.1 *Evaluating the optimization time*

Optimization time (see *measurement* group *3* in Figure 8.1) is measured with *steps 4-6* in Figure 8.3a. These measurements are all independent from the complexity of the virtual appliance they are operating with. Depending on the management capability of the original virtual appliance (p_σ), the evaluation results in *measurement 1* or *2*:

$$\text{if } manageable(p_\sigma) = \begin{cases} true & \text{then measurement 1} \\ false & \text{then measurement 2} \end{cases} \tag{8.12}$$

This evaluation scenario aims at estimating the cost of the architecture. Therefore, I calculate the number of future deployments (N_{dep}) required to overcome the cost of the virtual appliance size optimization:

$$\begin{aligned} N_{dep}(p_\sigma, \varphi) &:= \frac{measure(optimize(p_\sigma))}{measure(initVM(p_\sigma,\varphi)) - measure(initVM(p'_\sigma,\varphi))} \\ \text{where} \quad & servicepkg(p_\sigma) = true \\ \text{and} \quad & p'_\sigma \leftarrow optimize(p_\sigma) \end{aligned} \tag{8.13}$$

The *optimize* operation is defined in Figure 5.2. By default, the optimization – and therefore the measurement – is executed until p'_σ reaches its optimal size:

$$optimalsize(p'_\sigma) = true \tag{8.14}$$

8.5.2 *The cost of initial upload*

Finally, the last evaluation scenario covers *measurement 6* in Figure 8.1. This evaluation scenario uses a medium complexity virtual appliance (p_σ) and starts right after the extraction of the appliance from its original hosting system.

In order to estimate the cost of the initial upload algorithm (introduced in Section 4.2.3) I first take a baseline measurement (t_{IUBL}) by uploading the extracted virtual appliance to one of the following empty repositories:

$$\begin{aligned} R_\varphi &:= \{r_1, r_2, r_3\} \\ P_\varphi &:= \varnothing \\ contents(r_1) &:= contents(r_2) := contents(r_3) := \varnothing \\ t_{IUBL} &:= measure(uploadVA(p_\sigma)) \end{aligned} \tag{8.15}$$

During the measurement the repositories are always cleared after a single upload operation in order to maintain the initial requirements throughout the entire baseline evaluation.

Next, in order to determine the effects of the initial upload algorithm, I upload several packages to the repositories according to the requirements defined by Equations 8.2 and 8.3. These requirements allow the initial upload algorithm to optimize the upload process by analyzing the original virtual appliance (p_σ) in order to find the most suitable repository that can host the appliance with minimal upload. Therefore, I measure the time spent during the initial upload operation with the new repository layout to take the optimized timings (t_{IUOpt}). As a result, the initial upload algorithm is economically applicable when $t_{IUOpt} < t_{IUBL}$.

CHAPTER SUMMARY. This chapter revealed the ways future Chapters are evaluating my contributions. First, the chapter systematically lists the evaluation scenarios and links them to the appropriate subsystem of the AVS architecture. Next, based on their complexity, I have identified the kinds of appliances (*minimal-*, *medium-* and *high complexity*) needed for the complete evaluation of my research results. Afterwards, the chapter presents an elaborate scenario that not only involves all parts of the architecture, but it checks the *correctness* of the various components through extracting, size optimizing, decomposing and rebuilding a high complexity appliance. Finally, I have also included the use cases that are directly testing the *performance* of the various AVS subsystems.

9

TESTBED

CHAPTER OVERVIEW. Building on the methodology defined in the previous chapter, this chapter evaluates the proof of concept scenario and proves the infrastructure independence of the architecture. To do so the chapter defines several testbed infrastructures, details their implementations and provides the evaluation of the proof of concept scenario on them.

9.1 INTRODUCTION

This chapter has two main objectives: (*i*) introduce the testbed infrastructures (Nimbus, Eucalyptus, Proprietary) that can be used to execute the experimental measurements defined in Chapter 8, and (*ii*) demonstrate the *infrastructure independence* of the architecture by evaluating the proof of concept scenario (see Section 8.3) on the various infrastructures. These infrastructures all provide the required functionality for the proposed architecture, however they approach it from different perspectives: (*i*) *adapting* existing open source projects to support the AVS; (*ii*) using an *existing* infrastructure without modifications; finally, (*iii*) providing a *new minimal implementation* of an IaaS and a repository system that are fully aware of the AVS services.

First, to demonstrate how an existing open source IaaS system can be adapted for the use of the architecture, I have evaluated several available open source IaaS systems (Nimbus, OpenNebula, Eucalyptus) that could use the appliance rebuilding functionality with the smallest modifications (see Figure 6.6b and Section 6.3.3). I have chosen Virtual Workspace Service (VWS – [40]) offered by the Nimbus project [54], because it was not bound to a specific repository implementation. Therefore, I have extended the VWS service to support accessing and rebuilding appliances from an active repository implementation. For the active repository functionality, I supervised an MSc project that has extended and implemented the application content service (ACS – [33, 77]) – a proposed recommendation of the Open Grid Forum. This recommendation offers extensible metadata definition (requirements defined in Section 4.4), various metadata search operations (required for the appliance rebuilding and decomposition algorithms detailed in Chapter 6) and seamless integration to web-service based systems.

Second, I am discussing an implementation based on Eucalyptus, because of its three advantages: (*i*) Eucalyptus supports the basic requirements of the proposed architecture without extensions or modifications, (*ii*) Eucalyptus could be easily replaced with Amazon EC2 to present the viability of the proposed architecture on a commercial IaaS cloud system, finally, (*iii*) the implementation can avoid Grid Security Infrastructure and Web Services Resource Framework [6] that was a requirement while I was using the Virtual Workspace Service based testbed.

Finally, I have implemented a prototype IaaS system with an active repository to enable the evaluation of the proposed architecture to its full extent. This implementation only focuses on those IaaS and repository functionalities that are used by the various components of the architecture, but it still maintains the main characteristics of traditional IaaS systems and repositories.

9.2 THE GENERIC TESTBED

Before detailing the different implementations, I introduce the generic testbed aimed at supporting the different evaluation scenarios discussed in the applied methodology (see Chapter 8). Then, I detail the mapping of the proof of concept scenario (see Section 8.3) to this generic testbed. Afterwards, the specific testbed implementations are discussed through highlighting the behavioral differences and challenges compared to the generic testbed presented in this section.

Figure 9.1 reveals the generic testbed infrastructure required to accomplish the proof of concept evaluation scenario. This scenario involves all components of the proposed architecture; therefore, the system requirements of the components have to be met by the infrastructure. In the figure, the hosts of the infrastructure are denoted with ellipses (e.g. *n33* or *portal014*), the different administrative domains are circled with dashed and dotted lines (e.g. *University of Westminster*), the different software components are represented with boxes (e.g. *Xen*, *AVS*), arrows symbolize the direction of direct control between components, finally, repositories are represented with cylinders (e.g. r_1, r_2).

For the demonstration of the *extraction* functionality, I have implemented the central extraction service (see Section 4.2.1.1) of the AVS. Thus, the AVS must be deployed with direct access to the VMMs (namely *Xen* – [7]) of the infrastructure (e.g. see the host named *source* in Figure 9.1). Therefore, in or-

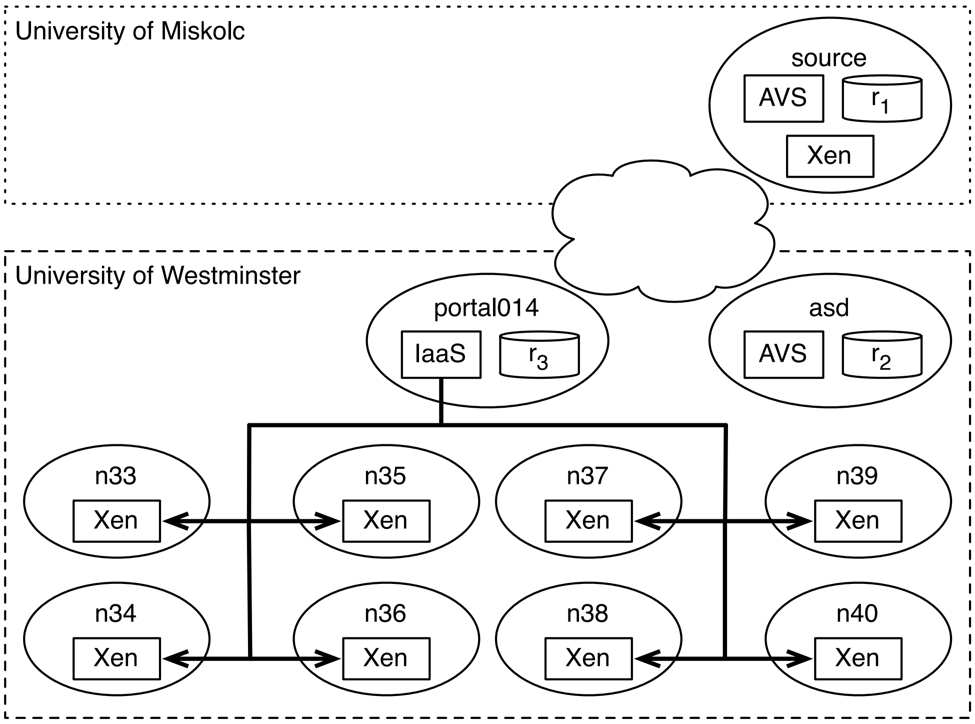


Figure 9.1: The generic testbed used for evaluating the AVS service and its components

der to allow extraction a highly complex web service has to be prepared in a virtual machine prior the first step of the proof of concept scenario (see Section 8.3). The validator algorithm has to be available for this high complexity service because the appliance of the service will undergo the optimization procedure. After analyzing various services, I have identified GEMLCA as the best candidate to represent the high complexity services. I have selected the GEMLCA service [22] for the following advantages: (*i*) it supports web service notifications, therefore, it requires extra firewall rules during its execution; (*ii*) it supports connectivity with various grid and web services as its external dependencies (e.g. grid sites supporting legacy code execution); (*iii*) its appliance has high amount of internal dependencies – e.g. a Globus Toolkit 4 (GT4 – [32]) service container; finally, (*iv*) I have personal development experience with the service as it is a product of the University of Westminster, therefore, I can create a complete validator for the service.

Consequently, I have installed GEMLCA in a virtual machine that is hosted by one of the VMMs accessible for the AVS service. Throughout the evaluation of the proof of concept scenario, I will use the GEMLCA virtual appliance (see Section A.3) as the service package:

$$p_2 := GEMLCA \tag{9.1}$$

PROOF OF CONCEPT SCENARIO. According to the proof of concept scenario, the evaluation of the architecture starts with the extraction of the virtual appliance of the highly complex service from its initial virtual machine.

The size of the new virtual appliance is optimized ($p_\sigma \rightarrow p'_\sigma$) after the extraction step in the scenario. To accomplish the evaluation of this task the optimization facility requires an IaaS service ($c \in \varphi$ – executed on host *portal014* in Figure 9.1) with a dedicated cluster ($\Xi_\varphi(c)$ – *n33-n40* in the figure) allowing the execution of the size optimization algorithm in parallel. University of Westminster provided 8 dedicated compute nodes of its cluster as the backbone of my testbed. All the nodes have the same hardware configuration (4 CPUs, 4GB of RAM and 80GB of HDD), software configuration (Debian Lenny with Xen virtual machine monitor [7]) and they are interconnected with Infiniband connections towards the host of the IaaS service. Only the service's host (*portal014*) has external connections towards the Internet. In this testbed, the 8 cluster nodes and the host of the IaaS service represent an entire IaaS system.

Next, the optimized GEMLCA appliance is uploaded to a repository. According to the requirements of the evaluation scenario, the testbed infrastructure should consist of three repositories (r_1, r_2, r_3). To represent the diversity of the repositories they should belong to three different connection classes according to their connection properties with the host of the IaaS service (c):

$$\begin{array}{ccccc} l(c,r_1) & > & l(c,r_2) & > & l(c,r_3) \\ BW(c,r_1) & \leq & BW(c,r_2) & \leq & BW(c,r_3) \end{array} \tag{9.2}$$

As a result, the generic testbed consists of the host called "*source*" that runs r_1 the local repository of the AVS service at the University of Miskolc ($BW(c,r_1) = 16Mbps$). This host runs the AVS service to demonstrate the scenario when the appliance developer uploads the initial virtual appliance from an external location of the IaaS system. Similarly to the host of r_1, r_2 is hosted on "*asd*" and it runs the local repository of the AVS service at the University of Westminster. However, this host is not part of the dedicated cluster, therefore, its sole connection towards the IaaS system is through a gigabit Ethernet port. The AVS service at Westminster can be used to apply the decomposition and optimization functions on existing appliances.

Finally, to allow the decomposition of the uploaded GEMLCA appliance the r_2 and r_3 should store packages according to Equations 8.2 and 8.3. These requirements not just define the desired locations of these packages but they also require to have common parts with the GEMLCA service. As the GEMLCA service is built on Globus services, I have used the "Globus 002" virtual appliance from the *Science Clouds marketplace* [20]. The Globus virtual appliance (referred as p_1 in Equation 8.2) is stored in r_2 before the evaluation of the proof of concept scenario is started.

An *SSH* appliance (p_μ, see Section A.1) is also stored in repository r_3 to serve as a dependency of the Globus appliance: $\exists n < PC(p_1) : \left(p_\mu \in D(p'_1, n)\right)$. In most of the measurements an optimized SSH appliance is used as the minimal manageable virtual appliance because its basic functionality can be used to present the management interfaces required by the various algorithms of the architecture. The SSH appliance can be used as a replacement for an MMVA because of its minimal complexity.

9.3 TESTBED WITH NIMBUS

Nimbus [54] is one of the earliest academic IaaS solutions [43]. During the early stages of my research the developers of Nimbus offered the only open source IaaS solution, therefore, I have used their solution as a base of my initial experiments on virtual appliance creation for services. These experiments were first published in [44].

Nimbus provides the Virtual Workspace Service (VWS) as a Web Services Resource Framework (WSRF – [6]) compliant service. This service is the central control point of the underlying infrastructure. Users contact the VWS to create, manage and destroy virtual machines on the VWS controlled cluster nodes. The service uses the *Workspace Controller* component on each node to enact the management tasks requested on the VWS interfaces. The service also handles repositories on a generalized way, therefore, support for new repositories can be added by third party developers.

When I conducted my initial experiments, Nimbus did not provide any particular repository implementations. Therefore, I have also investigated repositories (e.g. [13, 33, 69]) to present the advanced features of the architecture. The four main requirements against the repositories was (i) their WSRF behavior (to allow seamless integration with the virtual workspace service), (ii) their extensibility and the availability of their open source implementation, (iii) their capability to store and handle occasionally large sized virtual appliances and finally, (iv) their standardized behavior. After the analysis of various repository systems, I have selected the application content service (ACS – [33]) to serve as the repository of my Nimbus installation, because, the ACS is an Open Grid Forum proposal with multiple [61, 77] open source implementations designed with the WSRF concepts and extensibility in mind.

Figure 9.2 presents the deployment of the Nimbus services on the Westminster cluster. *eVWS* represents the service of the virtual workspace service and *WC* represents its workspace controller component. The workspace service is

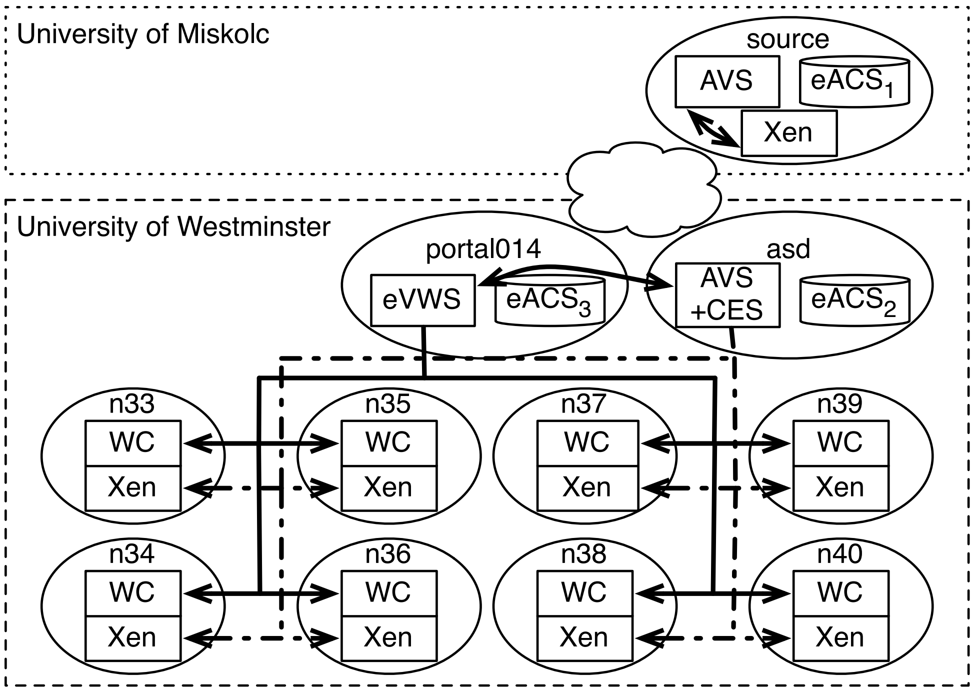


Figure 9.2: Nimbus based testbed

accessible from other administrative domains in contrary to the workspace controllers that are only accessible from the host of the *eVWS* service. Alongside with Nimbus services ACS repositories are also deployed and they are denoted as *eACS* in the figure. The AVS service runs the centralized extractor service (*CES* – see Section 4.2.1.1) component that can reach the Xen VMMs on all the cluster nodes (displayed with a dashed and dotted line). The following paragraphs describe the extensions that allow Nimbus and ACS services to meet the requirements of the generic testbed and the AVS service.

DEPLOYMENT CHALLENGES OF THE NIMBUS TESTBED. In 2008, the available workspace service (downloadable from the website of the Nimbus project [54]) supported only basic virtual appliance staging. Thus, it was able to download the virtual appliance by either using HTTP or by collecting the appliance from a shared file system between the Virtual Workspace Service and the virtualization-enabled machines (*n33-n40* in Figure 9.2). The workspace service allows its extension towards extra virtual appliance sources with the help of *StagingAdapters*. Therefore, as my extension to the local service, I have implemented a new staging adapter that allows downloading virtual appliances from ACS repositories. As a means to mark the difference between the publicly available and the extended workspace service, I have depicted the service as "*eVWS*" in Figure 9.2. The extension expects that the host of the VWS service also include a repository client for the ACS service.

This testbed infrastructure uses repositories compliant with the *Application Content Service* standard proposal of the Open Grid Forum. As the minimal implementation of the standard is not capable to serve as an active repository, I have specified the required extensions that reach the boundaries of the standard proposal and meet the requirements of my proposed architecture. Then, I supervised a successful MSc project (at the University of Miskolc) that implemented the specified extensions resulting [77]. This implementation adds several extra functionalities so that the standard ACS clients are still capable of using the extended features of the implementation. Because of these extended features, Figure 9.2 depicts the repository instances based on this implementation as *eACS*. This implementation adds the following functionalities to the proposal:

- To enable the transfer of larger virtual appliances, this ACS implementation offers *new transfer methods* above the minimal required by the standard (namely: gridftp and http).
- To enable decomposition and replication, it allows the creation of external package dependencies among repositories:

$$p \in contents(r_i) : (\exists p_x \in dep(p) : p_x \notin contents(r_i))$$

- To remove the threat for destroying an external dependency for a package, the implementation incorporates *new two phased package deletion and update* procedures ensuring the consistency of the system.
- Finally, the implementation allows extended *content management* so repositories can handle individual items in stored virtual appliance packages if necessary for the decomposition algorithm.

9.4 TESTBED WITH EUCALYPTUS

During my research, the influence of the commercial infrastructure as a service providers [39, 50] continuously increased on cloud computing technologies. Therefore, the proposed virtual appliance creation techniques should be applicable in these IaaS solutions. However, the evaluation of the architecture cannot be executed on commercial systems because of financial reasons. Consequently, a local testbed has to be constructed – based on the cluster of the University of Westminster – with equivalent properties (*closed* infrastructure and fixed *service interfaces*) to the commercial systems. To emulate these commercial systems, I have investigated the available non-commercial implementations of their interfaces [21, 40, 54, 58] and selected the open source version of Eucalyptus based on (i) its widespread use in the academia and

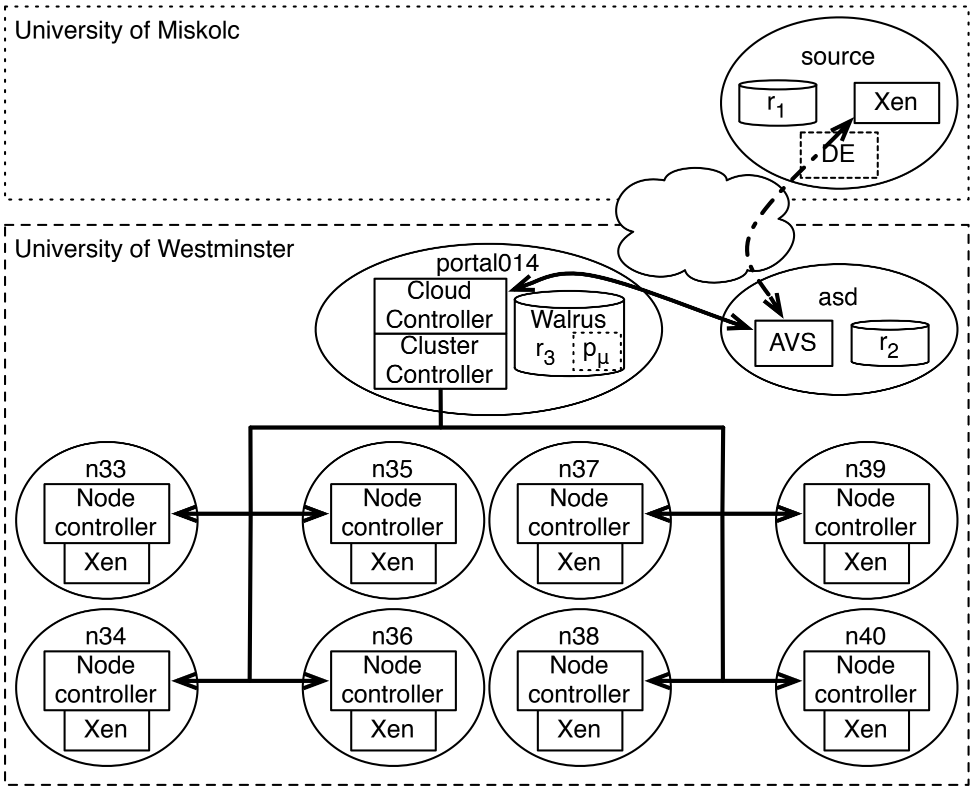


Figure 9.3: Eucalyptus based testbed

on (*ii*) its most complete implementation of the Amazon EC2 interface family.

Eucalyptus offers four basic services: (*i*) the *cloud controller* offers the entry point to a Eucalyptus IaaS system by implementing the EC2 interfaces of Amazon; (*ii*) a Eucalyptus system could consist of multiple clusters that can be reached through their *cluster controller* services; (*iii*) within a specific cluster the virtualization layer of each computing node is made accessible by the local *Node Controller* service; finally, (*iv*) *Walrus* offers the permanent storage – repository – facilities for the entire Eucalyptus system and it also implements the Simple Storage Service interface of Amazon.

Figure 9.3 reveals the specifics of a Eucalyptus based testbed system on the Westminster cluster. This installation only contains a single cluster thus both the *cloud* and the *cluster controller* services are deployed on the same host (*portal014*). This host runs as the controller of the local cluster by managing both its IaaS behavior and also its networking – e.g. DHCP, DNS and routing. The walrus repository is also located on *portal014*. Finally, in this testbed the Xen virtual machine monitors of the cluster nodes (*n33-n40*) are only accessible to the *Node controller* services.

CHALLENGES OF THE EUCALYPTUS TESTBED. I have identified three challenges while designing the Eucalyptus testbed: (*i*) support of the *closed behavior* of commercial IaaS systems, (*ii*) support for remote repositories and the *active repository* functionality, (*iii*) support the rebuilding of decomposed virtual appliances.

First, taking into consideration the requirement of *closed behavior*, the AVS (hosted on *asd*) only uses the *cloud controller* and *walrus* services throughout its entire operations (depicted with a bold bidirectional arrow in Figure 9.3). Consequently, the AVS service on this testbed cannot extract appliances deployed within the local cluster. Instead, the AVS supports the decoupled extraction component (shown as a dashed arrow in Figure 9.3, for details see Section 4.2.1.2) that allows the extraction operation to be executed remotely.

Second, the active repository functionality is supported by the decomposition functionality of an AVS service (see Section 6.2.1 on page 6.2.1). However, when the AVS provides the decomposition functionality the AVS has to be connected with the repository on a low latency high bandwidth link. Consequently, for each passive repository there should be a local AVS installation. Therefore, the AVS should be hosted within the same administrative domain as the repository by either deploying it within the IaaS system on a virtual machine or deploying it next to the repository. As I was the maintainer of the testbed infrastructure, I have chosen the second option by installing the AVS on the host of "*asd*".

Next, an MMVA (p_μ) stored in the central Walrus repository (r_3) enables the online rebuilding capabilities of the testbed. As a result, the system can rebuild decomposed virtual appliances by downloading and installing packages from repositories (either local or remote). Installing a package needs the MMVAs support for interfacing with the repository where the package is stored. As a result, in order to show that the system can handle mixed repository types, Figure 9.3 does not specify the type of the repositories r_1 and r_2.

Finally, the MMVA in the repository also enables the virtual machine pooling methods of the optimization facility (in the AVS service hosted at *asd*). VM pooling (detailed in Section 7.3.3.2) increases the efficiency of the optimization facility in closed IaaS systems, like Eucalyptus, where deployable virtual appliances have to be stored in their central repository (r_3).

9.5 MY PROPRIETARY TESTBED

The proprietary testbed system (presented in Figure 9.4) is built on the basic virtualization-enabled cluster nodes, and adds my custom built IaaS system and repositories that meet all the requirements of the AVS service. These cus-

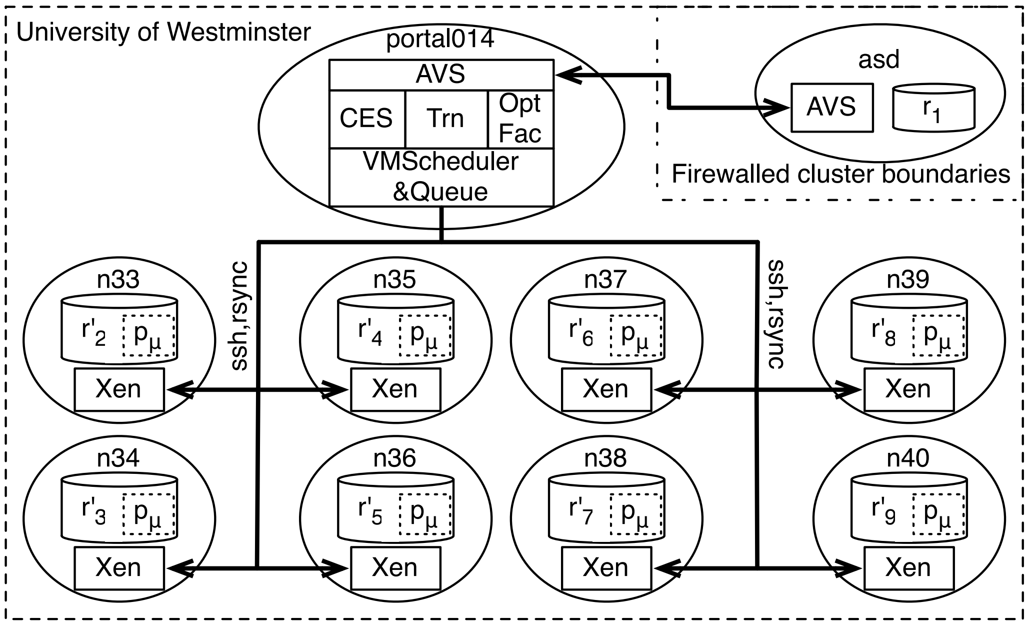


Figure 9.4: The proprietary testbed

tom built components allow the flexible reconfiguration of the testbed system for the subsequent evaluation of all scenarios introduced in Chapter 8.

The proprietary IaaS system is based on direct super user access (e.g. root login on Unix machines) to all cluster nodes and it manages the virtual machines with the command line toolset of Xen. The *VMScheduler & Queue* component is restricted to three basic functionality: (*i*) it keeps a record about virtual machine allocations on the cluster nodes, (*ii*) it also schedules the creation and destruction of new or unused virtual machines, finally, (*iii*) it allows the remote execution of virtual machine management operations like restart, suspend. The AVS components dependent on this IaaS system are drawn on its top: (*i*) the *central extraction service* (marked as "*CES*" in Figure 9.4) is dependent on the system's virtual machine management operations, (*ii*) the IaaS initiated MMVA based appliance transformation (depicted as "*Trn*" in Figure 9.4) uses the virtual machine creation capabilities of the system and (*iii*) the optimization facility (shown as "*Opt Fac*" in Figure 9.4) queues the virtual machine requests for the parallel validation tasks with the system.

The proprietary repositories ($r_1, r_2, \ldots, r_9$) keep their contents (P_φ) in a simple file structure. These repositories offer queries on their contents over a simple web service interface and transfer their content with the help of rsync (between repositories – [83]) and sftp (between a repository and the host of a VMM). They can be requested to deliver a package without rebuilding, however their default behavior is to rebuild virtual appliances before delivery.

Figure 9.4 reveals a major difference between the generic testbed and its proprietary implementation. As this testbed is controlled entirely by me, new administrative domains can be established within the cluster itself (no need for the site at University of Miskolc). In order to maintain the repository classes introduced in Equation 9.2 I have installed repositories $(r'_2, r'_3, \ldots, r'_9)$ on all the cluster nodes to compensate the removal of the host "*source*" from the testbed. Consequently, each repository in the cluster can behave as a medium or a high bandwidth one depending on the source of the package request. If the host of the repository requests the package, then the repository behaves as a high bandwidth one (see Equation 3.2) because only disk transfers are required to deliver the requested contents. In contrast, the repository behaves as a medium bandwidth one when it has to transfer the requested content to a third party.

9.6 SUMMARY

Table 9.1a presents the comparison of the applied technologies in the three testbed infrastructures. The table offers a cross-reference between the various testbed infrastructures and the discussion of their applied technologies (with section references in the last row).

The various testbed infrastructures introduced in the previous sections were evaluated with the proof of concept scenario (as described in Sections 8.3 and 9.2). The initial virtual appliances (GEMLCA – p_2 –, globus 002 and SSH) were identical during all three evaluations. Therefore, after the successful execution of the proof of concept scenario ($correct(p_2, \varphi) = true$) the three repositories (r_1, r_2 and r_3) should store identical packages independently from the used testbed infrastructure ($\Phi := \{\varphi_{\text{Nimbus}}, \varphi_{\text{Eucalyptus}}, \varphi_{\text{Proprietary}}\}$). Consequently, if all implementations result the same package layout and content ($\forall p \in P_{\varphi_x} : (\exists p_1 \in P_{\varphi_y} : (p = p_1))$), then I consider the target of *infrastructure independence* reached – see page 122. As a result, I have defined the following conditions for infrastructure independence:

$$\begin{aligned} 1^{\text{st}}: &\quad \nexists \varphi_x \in \Phi : (correct(p_2, \varphi_x) = false) \\ 2^{\text{nd}}: &\quad \forall \varphi_x, \varphi_y \in \Phi : (P_{\varphi_x} = P_{\varphi_y}) \end{aligned} \tag{9.3}$$

In the first condition, I specify that all infrastructures should pass the correctness criterion specified in Equation 8.6. The second condition for infrastructure independence can only be met between service-based systems ($\varphi \in \Phi$) that did not change after the evaluation of the proof of concept scenario. Consequently, this evaluation requires the execution of the proof of concept scenario on isolated infrastructures that nobody else uses during the evaluation.

Table 9.1b presents the evaluation of the proof of concept scenarios and the infrastructure independence criteria defined in the previous equation. This table presents the entire package set after the execution of the proof of concept scenario (this package set was initially discussed in Equation 8.4), in order to allow the comparison between the various package sets I have calculated the SHA1 hash values [28] for the images of every package. These hash values can be seen in the second column of the table (representing the Nimbus testbed – φ_{Nimbus}). Afterwards, I only present values if they are different from the original hash values of the Nimbus testbed. I have used a "==" sign, in case the packages in the later evaluated testbeds represent the same content as the original. Consequently, I consider the architecture universally applicable, when only the second column contains hash values.

---

chapter summary. In this chapter, I have defined the way a generic testbed system should be constructed for the evaluation of the proof of concept scenario (see Section 8.3). Then, the chapter details the implementation details of three testbed systems (namely Nimbus, Eucalyptus and a proprietary one) I have used to demonstrate the infrastructure independence of my research. Finally, the chapter presents the evaluation results of the three testbeds and compares them to each other to present their correctness and the IaaS independence of the AVS architecture.

Testbed	AVS		Size Optimization		Delivery Optimization	
	Extraction	Transformation	Intermediate VA	MMVA	Active repository	Rebuilding
Nimbus	CES	eACS	–	√	eACS	eACS
Eucalyptus	DE	MMVA	√	√	AVS	MMVA
Proprietary	CES/DE	IaaS/MMVA	√	√	Proprietary	IaaS/MMVA
Sections	4.2.1	4.3, 7.3.2	5.2.3	7.3.3.2	6.2.1, 9.3, 9.5	6.3.1, 7.3.1

(a) Applied techniques in the testbeds

	Testbed			Universality Criteria
	φ_{Nimbus}	$\varphi_{\text{Eucalyptus}}$	$\varphi_{\text{Proprietary}}$	
$correct(p_2, \varphi_x)$	true	true	true	1st
Packages	Hashes			–
SSH: p_μ	6fd45f863b45e3c83d97abb0dbd88a4095266cfb	==	==	2nd, see Equation 9.3
globus: p_1	fdc94bbfc78fdae904b76716faf7d65a7ec23d73	==	==	
globus: p'_1	3253fffd7bc905262d24e2732320eff60b7da137	==	==	
gemlca: p_2	1df9043a384905af89c6a903334d537f6295cd16	==	==	
gemlca: p'_2	7bce9a16d1a21398df31afa54d748491382c7096	==	==	
gemlca: p''_2	b3ca6595cd584bc5529dc53ef89dcc5052fea6b2	==	==	
independent: p_3	a0d7e21f6511039d9e395d9d584f3d98eda906ef	==	==	
common: p_4	307b3637cdc7825dc1e186ef1e40f140849e4a5c	==	==	

(b) Evaluation of the different testbed infrastructures

Table 9.1: Comparison of the testbed infrastructures

10 EVALUATION

CHAPTER OVERVIEW. This chapter continues the discussion of the experiments and measurements conducted to verify the four major objectives identified in Chapter 8. After Chapter 9 confirmed the *correctness* and *infrastructure independent* behavior of the architecture, this chapter aims at the evaluations and measurements for *deployment efficiency and cost*. To support these objectives the following experiments and measurements are all executed in the *proprietary testbed* (discussed in Section 9.5) that was designed to allow automatic reconfiguration between the different experiments so the infrastructure is always fit to suit the prerequisites of the evaluation scenarios.

10.1 USED VIRTUAL APPLIANCES

The evaluation of the various measurements defined in Sections 8.4 and 8.5 requires at least medium complexity virtual appliances. High complexity virtual appliances are dependent on external services, therefore, they would require these external services to be either deployed within the cluster or accessible through the network. However, the application of external services in different administrative domains could cause the following problems during the evaluation: (*i*) because of the increased network latencies and the unpredictable network changes they might reduce the performance of the proposed architecture and therefore increase its apparent operating costs; (*ii*) several architectural measurements (e.g. the full optimization of the appliance) result in the frequent execution of the virtual appliance validators that could involve extensive use of the external services. Consequently, the preparation for these evaluation scenarios should include the deployment of those external services that the selected high complexity virtual appliances are dependent on.

The outputs of the various evaluation scenarios are not dependent on high complexity appliances. Therefore, medium complexity appliances are superior to high complexity ones during the evaluation, because they can still be

Table 10.1: Basic virtual appliance properties of the experimental appliances

Appliance	Size	Compressed Size	Number of Files
SSH	476 MiB	120MiB	10647
Apache	667MiB	165MiB	14050

used to present the findings but with reduced prerequisites on the testbed infrastructure. Consequently, I have searched for typical medium complexity virtual appliances and made a selection based on the following criteria: (*i*) the popularity and wide use of the service and (*ii*) the effort required to create a validator for the service. Measurement results with widely used appliances allow the comparison with related works (e.g. the rPath Apache Appliance – [67]). The ideal candidate appliances should either have an already existing validator or the development of a new validator should require minimum effort. In order to ensure that my findings are not specific to a particular virtual appliance all evaluation scenarios should be executed on at least two independent virtual appliances.

Based on these criteria, I have defined appliances for two basic Internet services, namely: the SSH and the Apache web server appliances. The *SSH* appliance (defined in Section A.1) enables a user to transfer executables to its hosting machine and allows their execution. In contrast, the *apache* appliance (detailed in Section A.2) does not allow arbitrary code execution, however it enables the user to upload static html content to the machine. As a result, the internal apache web server could offer the uploaded content later on. Neither of these virtual appliances require configuration during deployment, because they are standalone services and they do not depend on external network connections (consequently they can be classified as medium complexity appliances). The basic properties of the two original virtual appliances are listed in Table 10.1.

During the experiments, these appliances were configured with 100 MiB of free space to allow custom content for their users. The free space is big enough to allow the initial use of the appliance, however for advanced usage it either has to be extended with one of the available tools (e.g. `resize2fs`, `xfs_growfs`) before publishing the appliance or alternatively a new file system can be attached for the custom content (e.g. by attaching an EBS volume on Amazon EC2).

10.2 THE COST OF APPLYING THE ARCHITECTURE

This section focuses on the *cost* of the application of the architecture. This is the last remaining evaluation target identified during the introduction of

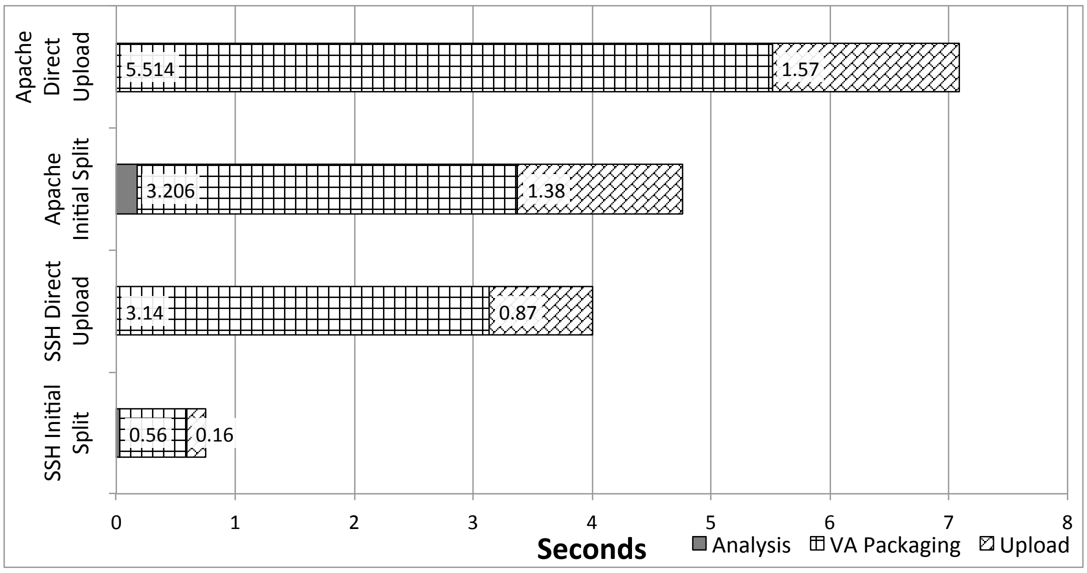


Figure 10.1: Comparison of initial upload phases with and without using Algorithm 4.1
As the length of the *Analysis* phase is negligible this figure does not provide its exact timing. In case of direct upload the analysis phase is not applicable. The number in the checked bars represent the time required for *VA packaging*. The number in the bricked bars represent the time required for *Upload*

the methodology. The measurements and their requirements, supporting this objective, were introduced in Section 8.5. Throughout this section, I discuss the cost of initial upload then the cost of the optimization. The discussion involves the basic measurements and the factors that can influence the estimated cost.

10.2.1 *The cost of initial upload*

Among the first tasks of the appliance developer is the publication of its newly created virtual appliance. This task is supported by the initial upload algorithm introduced in Section 4.2.3. This algorithm decomposes the virtual appliance and only uploads its unique parts for publication. Consequently, for the evaluation of the algorithm, I erased all the previous contents of the repositories in the proprietary testbed and uploaded my MMVA to r'_3. Then I have published both experimental virtual appliances in repositories r'_2 and r'_3. I have measured the duration of the publication operation and plotted it in Figure 10.1. The initial upload timings towards r'_2 are used as the baseline (t_{IUBL}) for these measurement and presented as *"DirectUpload"* in the Figure. The upload timings towards r'_3 have identified the common parts between the repository contents and used Algorithm 4.1 for publication. These timings (t_{IUOpt}) are presented as *"InitialSplit"* in the Figure.

Each upload procedure is composed of three phases. First, the system *analyzes* the contents of the current appliance with the contents of the available repositories and identifies the unique parts of the appliance. This analysis phase is missing if the proposed algorithm is not used, and the system proceeds as the entire virtual appliance would be unique. Next, the unique parts of the virtual appliance are enclosed in a repository *package*. Finally, the just created repository package is *uploaded* to the most suitable repository. If applied, the proposed algorithm selects this repository; otherwise, the appliance developer has to pick one based on his/her preferences and experiences with the infrastructure.

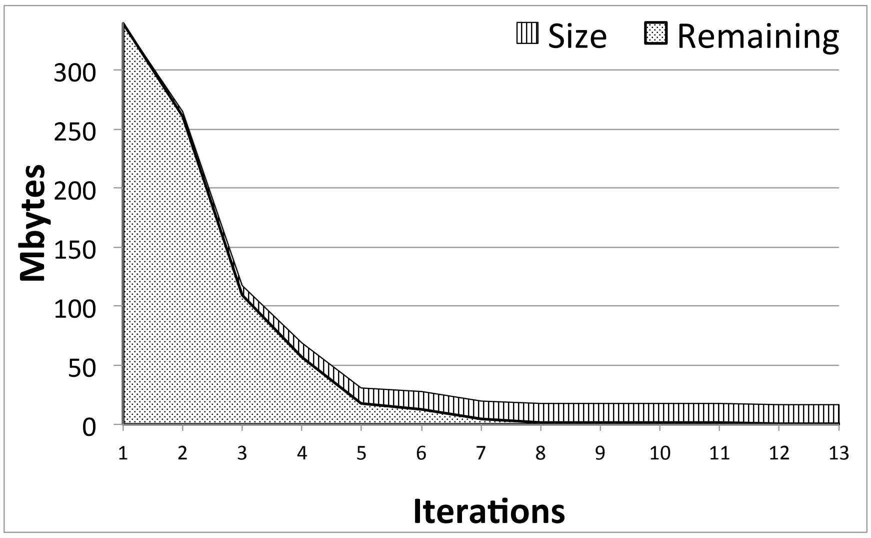
As it can be seen in Figure 10.1, using the initial upload algorithm the system can spare considerable amount of bandwidth for the appliance developer. Because of the algorithm, the experimental appliances became available more than 1.5 faster than the baseline measurement.

However, the initial upload algorithm is heavily dependent on the actual repository contents. The algorithm could even fail to serve its purpose on small but unique virtual appliances where the amount of data transferred to identify the candidate repositories is comparable to the size of the virtual appliance under publication. The algorithm is most useful for uploading appliances with small variance (e.g. when the appliance developer creates virtual appliances based on the same service with different software environments and configurations). This case is beneficial for both the appliance developer who needs to upload just a fraction of the appliance and for the repository because of the reduced storage needs of these appliances.

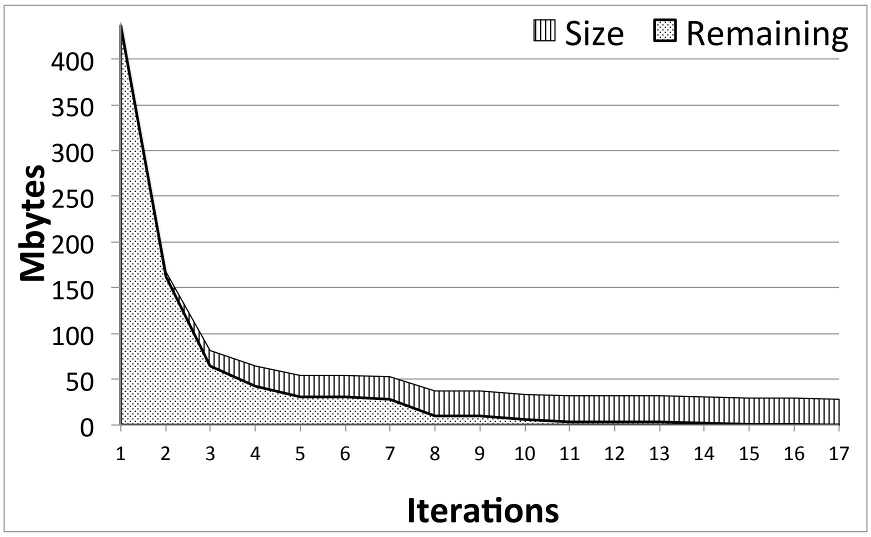
10.2.2 *Evaluating the cost of size optimization*

This section further details the measurements initially introduced in Section 8.5.1. These measurements are focused on the efficiency and the benefits of the proposed virtual appliance size optimization technique (see Chapter 5). This section analyzes the optimization facility from multiple viewpoints including (*i*) the gains on the deployment times of the optimized appliances, (*ii*) the impacts of the various optionally applicable techniques (snapshotting, using the optimization target criterion and application of MMVAs) of the facility on the optimization time.

During the evaluation of the optimization facility the repositories are not playing any role because the optimization takes place entirely in the cluster behind the AVS service, thus there is no need to publish the virtual appliance before the optimization completes. In order to evaluate the optimization facility, I have executed an unlimited optimization of both experimental virtual appliances (I have executed the optimization procedure without an



(a) SSH appliance



(b) Apache appliance

Figure 10.2: The process of optimization demonstrated on the size of the appliance

end criterion). Figure 10.2 presents these executions through the progression of appliance size. The figure depicts the ratio between the size of the intermediate virtual appliance after a specific optimization iteration and the size of the non-validated (and thus still remaining) items of the virtual appliance.

Instead of evaluating the optimization facility with a regular execution, I have initiated the optimization process that monitored the state of the optimization system at the end of the optimization iterations (see Section 5.2.1). The state includes the items removed, the current value of all five completion metrics (see Section 5.2.1.4), the number of virtual machines used, the number of validations passed, the average time to initiate a virtual machine, etc.

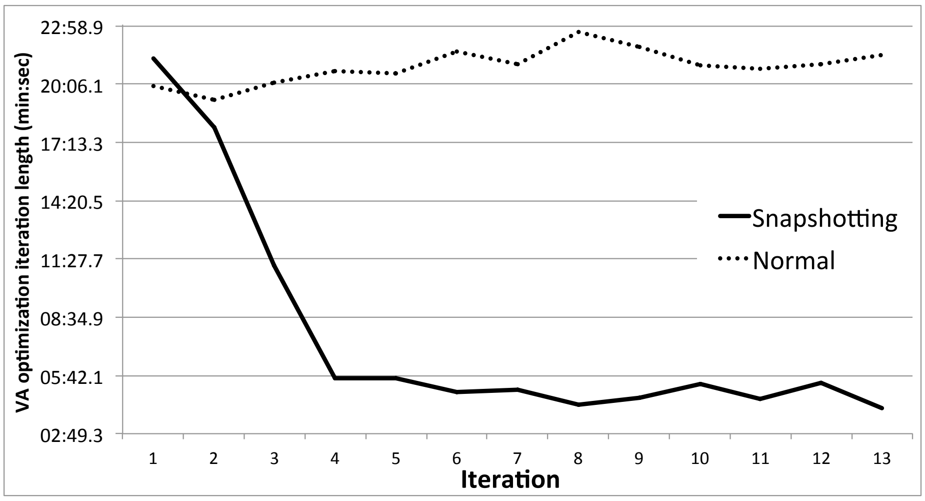
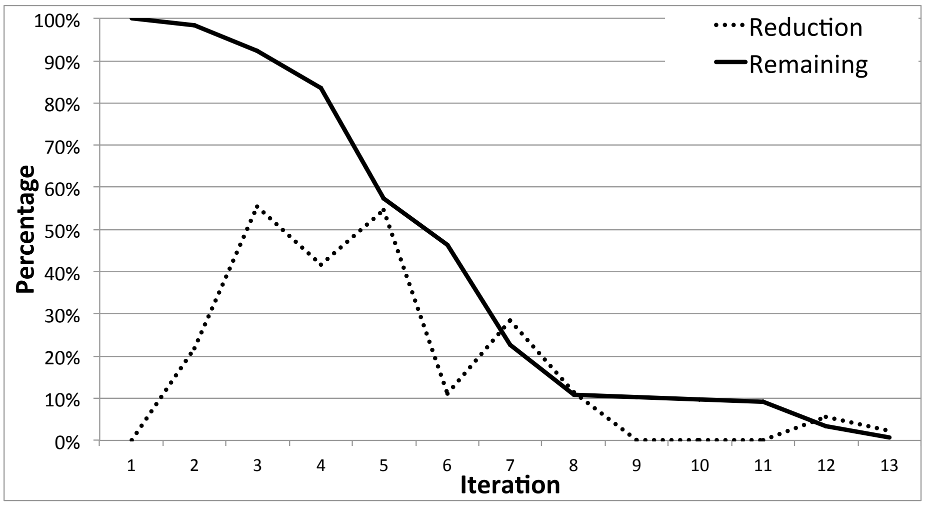


Figure 10.3: Effect of intermediate virtual appliance creation in the optimization facility

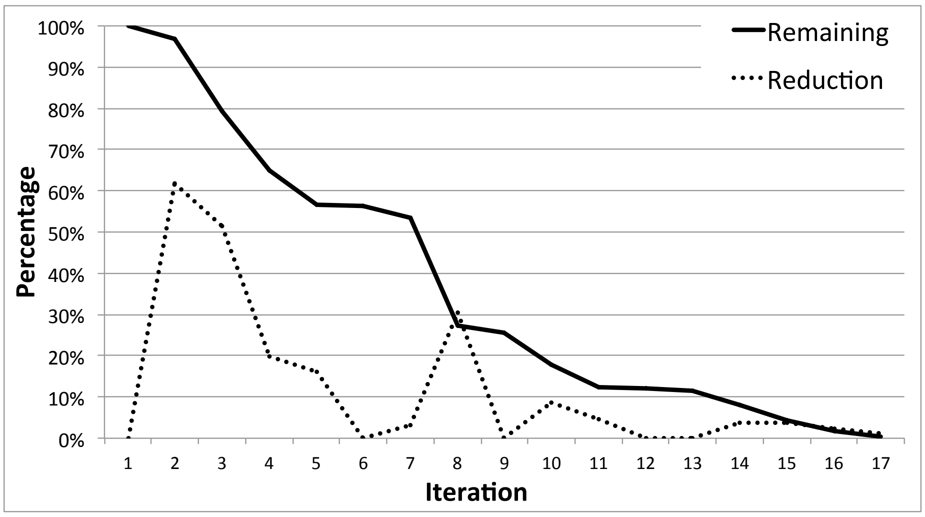
Based on these values I can evaluate the system's behavior without executing the optimization process in too many configurations.

As it was depicted in Table 10.3, the unlimited optimization process requires hours to complete. I have implemented a *snapshotting* approach that creates an *intermediate virtual appliance* after the optimization iterations as discussed in Sections 5.1.1 and 5.1.2.2. This appliance is stored in a virtual appliance playground (see Section 4.2.2) with the validated items already removed; therefore, later on the algorithm uses this reduced virtual appliance in the parallel validation algorithm. Figure 10.3 presents the optimization of the SSH virtual appliance with and without the creation of the intermediate virtual appliances. The figure reveals that intermediate virtual appliances can dramatically decrease (e.g. from 20 minutes to less than five) the time and cost of later iterations of the size optimization process, therefore, this technique immediately results in the reduction of the total optimization time.

However, this approach still does not exploit the completion criterion evaluation in the *target check* action. Figure 10.2 reveals another problem: in both executions, the optimization process could not reach significant size reduction in its late stages (the size of the remaining non-validated items follow a Pareto distribution). This is also reflected in Figure 10.5 where we can see that more than 40% of optimization time is spent on less than 10% of further size gain. Therefore, to avoid the tail problem, I have investigated the various completion conditions (introduced in Section 5.2.1.4) whether they can predict the inefficiencies of the late optimization process. However, most of the completion conditions cannot predict the inefficiencies, because they are either not stable enough throughout the entire optimization process (see



(a) SSH



(b) Apache

Figure 10.4: Comparing the stability of the *remaining* and *reduction* completion conditions – see Section 5.2.1.4

reduction in Figure 10.4a), or their inefficiency threshold cannot be generalized for multiple appliances (e.g. optimization time). A stable completion condition has to be monotonic, in order to allow the definition of a threshold value that the completion condition variable only crosses once during the optimization process. I have defined *inefficiency indicators* as special completion condition variables that reveal the inefficiency of the optimization process after passing the previously defined threshold value.

From the list of Section 5.2.1.4, I have identified two completion condition variables as candidates for the role of inefficiency indicator. The first one is the *size reduction* percentage achieved during a single iteration. The

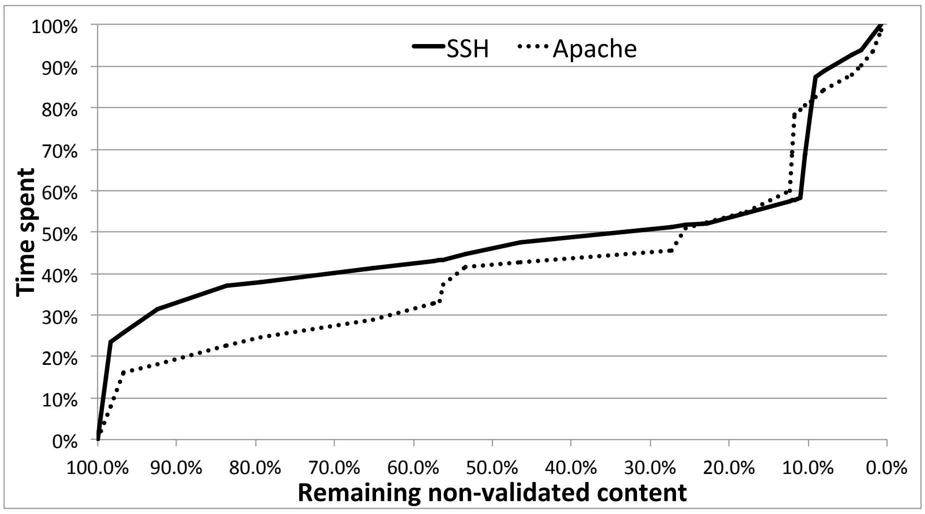


Figure 10.5: Effects of *remaining* size exit condition on execution time

second one is the percentage of the *remaining* (non-validated) appliance contents compared to the current size of the intermediate virtual appliance. Figures 10.4a and 10.4b presents the changes of these two completion conditions throughout an entire optimization. I have chosen the *remaining* completion condition for further investigation because its monotonic nature.

Figure 10.5 demonstrates the effects of applying different *remaining* completion condition values in the *target check* action. In this figure, I have normalized the optimization time for better comparison (for the concrete optimization time values see Table 10.3 or Figure 10.2). In order to decrease optimization time but still maintain adequate appliance size I have identified that the *remaining* completion condition variable can be used as an inefficiency indicator with the threshold of *10*%. I have identified the 10% as the indicator value because – according to Figure 10.5 – it significantly reduces (around 60%) the optimization time (the effect of Pareto distribution on the late stages of the optimization) and still maintains the size of the virtual appliance close to optimal.

The last option to increase the effectiveness of the optimization process is the introduction of embedded *minimal manageable virtual appliances* (MMVA). This option enables a new item removal algorithm and allows the pooling and reuse of virtual machines according to Section 7.3.3. As discussed before, I have designed the two experimental virtual appliances with embedded MMVAs. Therefore, the optimization facility can reuse the virtual machines utilized during the optimization process if it ensures that they are still functional after some of its items were removed. Figure 10.6 compares the execution time of the optimization procedure depending on its use of the management capability in the appliance under optimization.

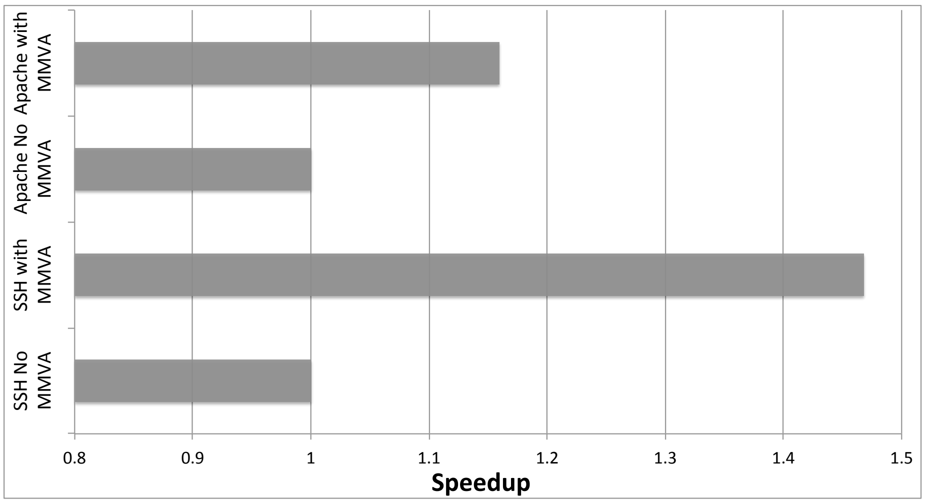
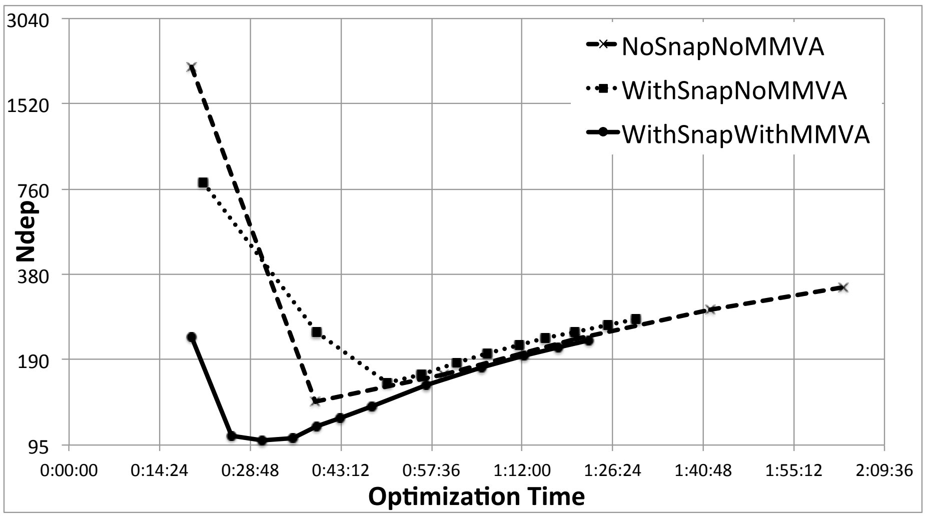
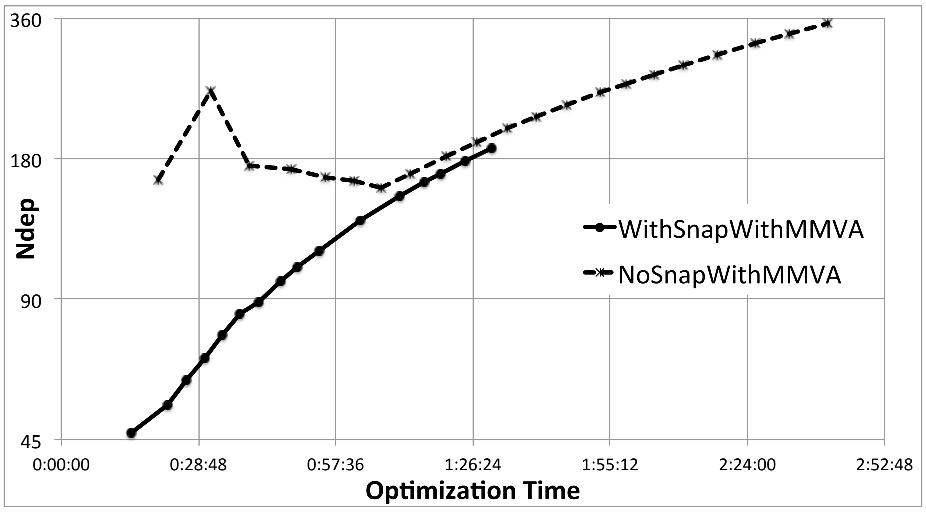


Figure 10.6: Effects of MMVA usage during optimization

Finally, using the previously introduced options (snapshotting and MMVA) for increasing effectiveness I have calculated the N_{dep} values (see Equation 8.13 in Section 8.5.1) based on the statistical information collected during the execution of the optimization processes. Figure 10.7 presents the calculated values and presents the minimum amount of the future deployments required before the optimization becomes profitable. The data points in the figure represent the optimization iterations: (i) x-values represent the exact time spent upon reaching a specific iteration and (ii) y-values depict the actual N_{dep} value calculated based on the time spent on the optimization and on the deployment time of the reduced appliance (p_{red}) after the completion of the iteration. According to the figure, the cost of the optimization procedure is high in the early stages reflecting the high grouping failure rates (see Figure 5.3) and the long iteration lengths (see Figure 10.3). Later, the cost increases linearly with the executed optimization iterations because the last iterations of the process are not reducing the deployment time considerably (see Figure 10.2).



(a) SSH appliance



(b) Apache appliance

Figure 10.7: The number of future deployments required before size optimization becomes profitable

Table 10.2: Deployment times [in seconds] before and after the optimization process

			Optimized & Rebuilt by		
	Baseline	Optimized	IaaS	Repository	MMVA
SSH	26.87	4.65	4.63	6.48	4.64
Apache	36.4	7.78	6.73	11.98	6.83

10.3 THE INFLUENCE OF THE AVS ARCHITECTURE ON DEPLOYMENT

This section discusses the preparations and measurements taken to present the *efficiency of deployment*. The experimental measurements are based on the requirements and discussion in Section 8.4. In order to present the viability of the proposed architecture I have measured the deployment time of the experimental virtual appliances (see Section 10.1) in various situations and testbed configurations.

First, I have prepared for the baseline measurement introduced in Section 8.4.1. During this measurement r'_2 (see host *n33* Figure 9.4) stored the original version of both experimental virtual appliances. All other repositories were empty and r'_2 did not store any other packages. Then I have measured the deployment time of both virtual appliances while they were sequentially instantiated in their virtual machines on the host of r'_2 – namely *n33*. This baseline measurement is shown in the second column of Table 10.2. The table presents the measured timings formed from both the *installation* and the *activation* deployment tasks (see Section 1.1). In the proprietary testbed the installation task of virtual appliance based services are composed of *downloading* and *unpacking* the virtual appliance on the target host. The activation of the service instance is composed of two steps: (i) the virtual machine is started (*VM Startup*), and then (ii) the service is started up within (*Activation*). The individual timings of all four phases of the basic deployment procedure can be compared in Figure 10.8.

Next, I have executed the size optimization procedure on the virtual appliances stored in r'_2 without any completion criteria specified for the *target check* action (see Section 5.2.1.4 for details). Therefore, the optimization facility progressed with the optimization task until it has not found any further removable items in the virtual appliances. Table 10.3 presents the basic properties of the virtual appliances received as the results of the unlimited optimization task (optimized appliances are listed with an apostrophe entailed to their names). The eight machines optimized the appliances to less then tenth of their original size within no more than two hours. In addition, the file counts of the appliances have dropped to less than $\frac{1}{50}th$ of their ori-

Table 10.3: Basic virtual appliance properties of the optimized experimental appliances

Appliance	Opt. size	Compr. Opt. Size	Files	Opt. time
SSH′	147MiB	6.6MiB	197	4958 secs
SSH″	N/A	1.2MiB	28	N/A
Apache′	192MiB	13MiB	236	7468 secs
Apache″	N/A	7.6 MiB	67	N/A
rPath Apache	1938MiB	152MiB	28923	N/A

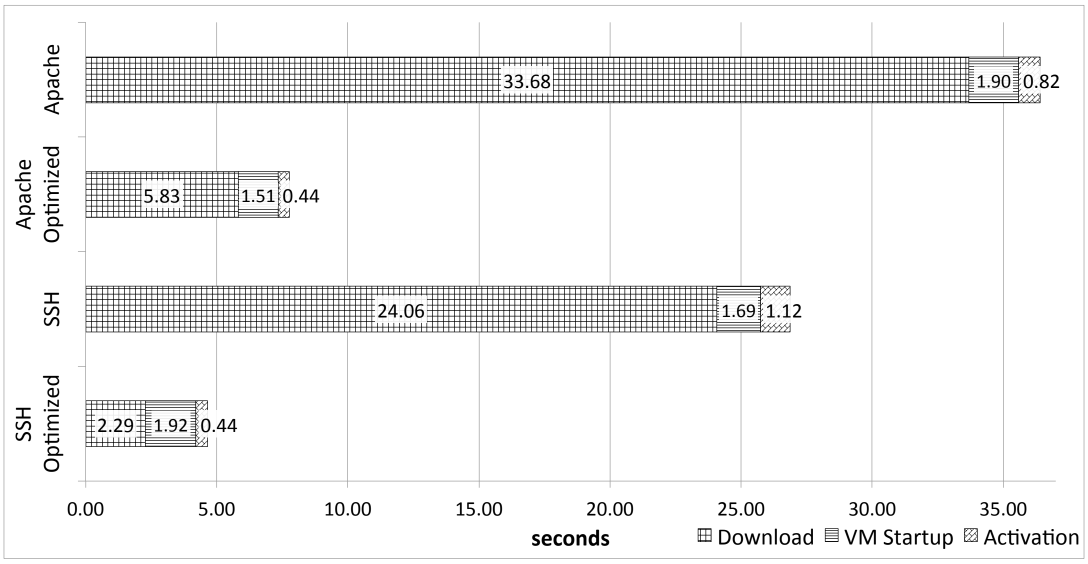


Figure 10.8: Deployment phases

ginals. The optimized appliances were also stored in repository r_2'. To allow comparison with the currently available appliances, Table 10.3 also lists the properties of the rPath Apache appliance [67] created and optimized with rPath's *pre-optimizing* approach (see Section 2.3). This appliance is a typical developer created appliance that does not take into consideration the target functionality of the appliance therefore the appliance contains non-necessary content.

Then, I used both optimized virtual appliances and deployed them to host *n33* (see Figure 9.4) while measuring their deployment times. These measurements are presented in the third column of Table 10.2 and in Figure 10.8. The various download time measurements clearly reveal the main cause of the deployment time speedup after the application of the optimization facility. After the comparison of the service activation times of the original and the optimized appliances, I conclude that the optimization facility successfully removed components causing unnecessary delays in activation (e.g. some unused system level services that were not important for the target functionality of the appliance). This feature increases the security of the optimized virtual appliances compared to the original ones, because the size optimized appliances cannot be attacked through unused system level services.

Afterwards, I have added an MMVA to the system (p_μ) that was prepared to serve as the base virtual appliance for both experimental virtual appliances. I have added this package to both r'_2 and r'_3. As a result, the system initiated the decomposition of the optimized appliances in r'_2 leading to the decomposed virtual appliances denoted in Table 10.3 with a tailing double apostrophe. The table presents the decomposed virtual appliances only with their service package. The dependencies of the service packages are not listed. Consequently, the table shows the files and their cumulative sizes that directly represent the target functionality in each appliance. The table reveals that these decomposed virtual appliances are significantly smaller than the originals, however their real uncompressed sizes cannot be determined until they are rebuilt before deployment.

Later, I deployed the decomposed virtual appliances on the host of repository r'_3 (see host $n34$ in Figure 9.4). During these deployments, I have configured the proprietary testbed to support one of the various rebuilding options introduced in the dissertation: (i) Active Repository based rebuilding (see Section 6.3.2), (ii) IaaS based rebuilding (see Section 6.3.3), finally, (iii) MMVA based rebuilding (see Section 7.3.1) using the *pre-transfer measurement threshold* value of $\tau = 0.1$. These rebuilding algorithms were individually used during the deployment of the decomposed virtual appliances and their deployment timings are presented in the last three columns of Table 10.2.

After all the measurements were executed successfully, the speedup values ($S(p,\varphi)$) for all the techniques are calculated and presented in Figure 10.9. As the calculated speedup values are all significantly higher than one (see Equation 8.8), I can conclude that the architecture is capable of decreasing the deployment time of the various virtual appliances.

If the deployment times of the size optimized virtual appliances (denoted as "*Opti*" in Figure 10.9) are used as the baseline value for calculating the speedup, then the figure also allows the comparison of the various rebuilding algorithms. As a result, the figure highlights that the measurements for active repository based rebuilding (denoted as "*RebuAR*" in Figure 10.9) confirm the extra transfer costs identified in Table 7.1. As it was revealed in Section 6.3.1 the active repository based rebuilding is only feasible when the IaaS system has low bandwidth connections towards those repositories that

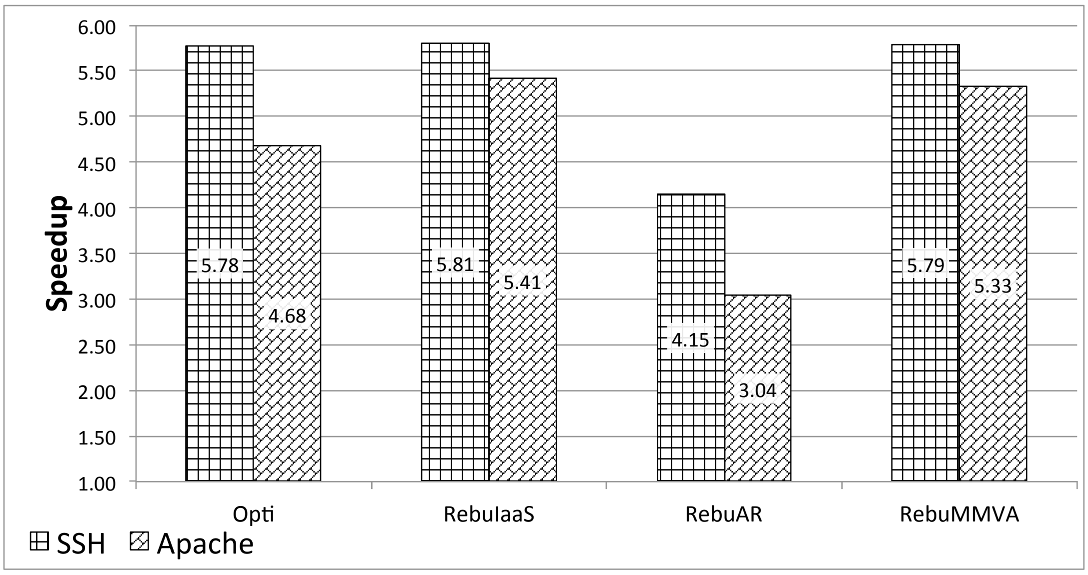


Figure 10.9: Effects of rebuilding

store the delta packages. However, this situation cannot be simulated with the available infrastructure at the University of Westminster.

In the proprietary testbed infrastructure, the last two rebuilding solutions could further improve the deployment times of size optimized virtual appliances by maximum 15% at the cost of storing the MMVA in all the repositories. The MMVA based (denoted as "*RebuMMVA*" in Figure 10.9) rebuilding results and their closeness to the IaaS based ones – denoted as "*RebuIaaS*" in Figure 10.9 – reveal the applicability of the architecture in a closed IaaS environment like the Amazon EC2.

---

chapter summary. This chapter has first discussed the virtual appliances chosen for evaluating the potential of the system. Next, I have demonstrated the experiments to evaluate the *cost* of applying the proposed architecture. Finally, the last part of the chapter presented the measurement results that verify the accomplishment of the increased *deployment efficiency*.

11 CONCLUSIONS

Highly dynamic service environments introduce new demands on service deployment systems because they might schedule service calls on yet-to-be deployed service instances. As a result, service users will face prolonged service calls until the deployment system handles the creation of the new service instance. Therefore, in highly dynamic service environments appliance based service deployment systems are only usable if the deployment time of the various appliances can be reduced. This dissertation has presented an architecture for automatically creating virtual appliances with reduced deployment time (see Chapter 3). The architecture is compared to other deployment systems in Table 11.1.

This novel architecture is aimed at supporting the virtual appliance developers to encapsulate services in appliances usable in highly dynamic service environments. In the current practice the virtual appliance creation task is usually performed manually that hinders dynamic service deployment and makes impossible to create dynamic adaptive systems. The primary contribution in my work is a mechanism that automates this task. Consequently, it enables faster adaptation of new services in the various Infrastructure as a Service cloud systems. As more services become deployable in cloud systems, they can increase the dynamism of the service-based system.

My first contribution to the knowledge was an "*approach for initial virtual appliance creation*" (see Section 1.3.1). This contribution allows appliance developers to extract an already existing service from a running system and encapsulate it within a virtual appliance. The proposed mechanism incorporates the automated metadata collection and initial upload algorithms that simplify the first publication of the newly created virtual appliance and prepares its participation in virtual appliance based deployments. The collected metadata supports the new size and delivery optimization algorithms and the on-demand deployment process. The presented initial upload algorithm minimizes the upload time of the appliance. As a result of this contribution, the appliance developer can reduce the required resources for publishing the

Deployment Systems	Isolation	Repository support	Universality	Non Invasiveness	State transfer support
IaaS systems	Virtualised	✓	✓	✓	—
Hot Deployment Service	Container	—	✓	✓	—
HAND	Container	—	✓	✓	—
WSPeers	Container	—	✓	✓	—
Dynagrid	Container	—	✓	—	✓
CDDLM implementations	N/A	—	—	—	—
This work	Virtualised	✓	✓	✓	✓

(a) Requirement based classification

Deployment Systems	Selection	Installation	Configure	Activate	Adapt	Deactivate	Update	Decommission
IaaS systems [40, 48, 58]	—	✓	—	✓	—	✓	—	✓
HDS [75]	—	✓	Partial	✓	—	✓	—	✓
HAND [62]	—	✓	Partial	✓	—	✓	—	✓
WSPeers [35]	✓	✓	Partial	✓	—	✓	—	✓
Dynagrid [15]	✓	✓	Partial	✓	—	✓	—	✓
CDDLM [81]	—	✓	✓	✓	✓	✓	—	✓
This work	—	✓	✓	✓	✓	✓	✓	✓

(b) Deployment task based classification

Table 11.1: Comparison of the proposed architecture to related works

virtual appliance, and concentrate on ensuring the target functionality of the service instead of its internal representation in IaaS cloud systems.

Next, my second contribution provided a "*parallel algorithm for virtual appliance size optimization*" (see Section 1.3.2). This contribution not only reduces the deployment time by optimizing the size of the virtual appliances but it also provides a technique that minimizes the optimization time and allows the early release of the optimal appliances. The proposed approach uses active fault injection to remove irrelevant parts of the virtual appliances. In order to maintain the target functionality of the appliance, the reduced virtual appliances are validated with algorithms provided by appliance developers. The proposed size optimization approach offers several advantages over the existing solutions. First, the appliance developer (who initiates the optimization process) does not need to know the dependencies of the service before its encapsulation in a virtual appliance. Second, this solution could also be used with existing virtual appliances. As a result, frequently deployed and decommissioned virtual appliances can be activated cost efficiently and rapidly.

Furthermore, my third contribution introduced the concept of active repositories for "*distributed virtual appliance storage and delivery*" (see Section 1.3.3). This contribution aims at optimizing the storage costs and delivery time of virtual appliances in repositories. The proposed solutions are based on the partial replication of virtual appliance contents where the replicated parts are defined automatically by a decomposition algorithm. The presented decomposition algorithm determines which parts of the virtual appliance have to be distributed and on which sites. I also provide a mechanism to rebuild the decomposed virtual appliances on the target site. My virtual appliance rebuilding algorithm reduces deployment time by employing the decomposed and replicated parts of the appliances. Therefore, this solution reduces the apparent service execution time of service calls preceded by deployments.

Later, my fourth contribution defined the "*minimal manageable virtual appliance*" (see Section 1.3.4). This contribution provides a virtual appliance that appliance developers can build on. The embedded minimal manageable virtual appliance offers more efficient deployment, transformation and size optimization procedures independently from the IaaS system in use. As a result, I have identified requirements for minimal manageable virtual appliances so that they can improve the possible adaptations of the architecture: (*i*) they should offer the manageability interfaces that can be utilized by the various components of the architecture (e.g. the optimization facility), (*ii*) they should allow their embedment in other virtual appliances and (*iii*) they should have minimal impact on the deployment time of the virtual appliances encapsulating them. Based on these requirements I have defined an approach to create and embed minimal manageable virtual appliances in future appliances.

Finally, I have designed three testbed infrastructures to present the applicability of the proposed architecture in different IaaS systems and environments. Afterwards, I have presented my implementation by experimenting on two well-known services (the SSH and the Apache Web servers) represented as virtual appliances. The results revealed that regular virtual appliances could be significantly optimized in size and their deployment time can be reduced with the help of the rebuilding algorithms. Based on these measurements I have also identified that the time and cost efficiency of the optimization algorithm can be improved by (*i*) creating intermediate virtual appliances and by (*ii*) terminating the optimization process using the ratio of the remaining non-validated content in the suboptimal VA as the completion criterion.

11.1 FUTURE RESEARCH DIRECTIONS

While this dissertation answered several questions regarding the minimization of the deployment time, the architecture has left open the following questions:

- Future research should be focused on the further optimization of the item selection and grouping algorithms of the proposed size optimization approach.

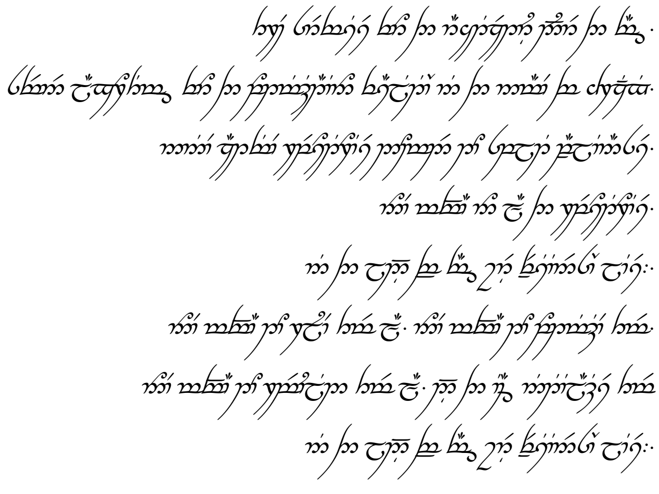
- I am also considering the development of more efficient scheduling algorithms for the parallel validation phase in order to reduce the resource usage caused by the revalidation of previously successful validation results (see Figure 5.4 and Section 5.2.2.1).

- In order to speed up the appliance rebuilding process the current implementation of the decomposition algorithm stores the same parts of a virtual appliance multiple times. I plan to investigate and extend the decomposition algorithm with simultaneous optimization of the repository contents for effective disk usage and achievable rebuilding speedup.

- I also intend to extend the rebuilding algorithm to give feedback on possible merging of previously decomposed repository content based on the evaluation of different appliance rebuilding strategies.

- The current extraction algorithm assumes that the virtual appliances are extracted from virtual machines. Later, this algorithm should be extended to support extracting virtual appliances from the physical machine of the virtual appliance developer.

- Even though the research evaluation is promising, its small scale did not allow the complete analysis of the proposed active repository functionality. Future research should identify the infrastructural requirements to enable the experimentation with the active repository and rebuilding functionalities of the architecture.



12

PUBLICATIONS DURING RESEARCH

JOURNAL

- **Gabor Kecskemeti**, Gabor Terstyanszky, Peter Kacsuk and Zsolt Nemeth. An Approach for Virtual Appliance Distribution for Service Deployment. *Future Generation Computer Systems*, 2011, volume 27, issue 3, pp. 280–289.

CONFERENCES

- Attila Kertesz, **Gabor Kecskemeti** and Ivona Brandic. Autonomic SLA-Aware Service Virtualization for Distributed Systems. *In: Proceedings of the 19th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2011)*, Ayia Napa, Cyprus, February 9-11, 2011. IEEE, Los Alamitos, USA.

- Attila Kertesz, **Gabor Kecskemeti**, and Ivona Brandic. An SLA-based resource virtualization approach for on-demand service provision. *In Proceedings of the International Conference on Autonomic Computing, 3rd International Workshop on Virtualization Technologies in Distributed Computing*, pp. 27–34, 2009.

- **Gabor Kecskemeti**, Peter Kacsuk, Thierry Delaitre, and Gabor Terstyanszki: Virtual Appliances: a Way to Provide Automatic Service Deployment. *In Proceedings of 3rd International Workshop on Distributed Cooperative Laboratories (INGRID 2008)*, April 2008

- **Gabor Kecskemeti**, Peter Kacsuk, Gabor Terstyanszky, Tamas Kiss and Thierry Delaitre: Automatic service deployment using virtualisation. *In: Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, Toulouse, France, February 13-15, 2008. IEEE, pp. 628–635. ISBN 9780769530895

- **Gabor Kecskemeti**, Gabor Terstyanszky, Tamas Kiss and Peter Kacsuk: Legacy code repository with broker-based job execution. *In: CoreGRID Workshop on Grid Systems, Tools and Environments in Conjunction with GRIDS@work: CoreGRID Conference*, Grid Plugtests and Contest, 01 Dec 2006, Sophia-Antipolis, France.
- **Gabor Kecskemeti**, Yonatan Zetuny, Gabor Terstyanszky, Stephen Winter, Tamas Kiss and Peter Kacsuk: Automatic deployment and interoperability of grid services. *In: Cox, Simon J. and Walker, D.W., (eds.) Proceedings UK e-Science All Hands Meeting 2005*, Steering via the Image in Local, Distributed and Collaborative Settings. EPSRC, Swindon, UK, pp. 729–736. ISBN 1904425534

BIBLIOGRAPHY

[1] Vmware. http://www.vmware.com.

[2] Bob Amstadt and Michael K. Johnson. Wine. *Linux Journal*, http://www.linuxjournal.com/article/2788, August 1994.

[3] Jonathan Appavoo, Volkmar Uhlig, and Amos Waterland. Project kittyhawk: building a global-scale computer: Blue gene/p as a generic computing platform. *SIGOPS Oper. Syst. Rev.*, 42(1):77–84, 2008.

[4] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, Jean-Charles Fabre, Jean-Claude Laprie, Eliane Martins, and David Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Trans. Softw. Eng.*, 16(2):166–182, 1990.

[5] M. Armbrust, A. Fox, R. Griffith, A. 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 UCB/EECS-2009-28, University of California at Berkley, February 2009.

[6] Tim Banks. Web services resource framework (wsrf) – primer v1.2. http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-os.pdf, 2006.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebar, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, October 2003. ACM.

[8] Hamid Abdul Basit and Stan Jarzabek. Detecting higher-level similarity patterns in programs. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 156 – 165, Lisbon, Portugal, Sept 2005.

[9] Meriem Belguidoum and Fabien Dagnat. Dependency management in software component deployment. *Electr. Notes Theor. Comput. Sci.*, 182:17–32, 2007.

[10] D. Bell, T. Kojo, P. Goldsack, S. Loughran, D. Milojicic, S. Schaefer, J. Tatemura, and P. Toft. Configuration description, deployment, and lifecycle management (cddlm) foundation document, August 2005.

[11] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[12] David Blackman. Debian package management, part 1: A user's guide. *Linux Journal*, http://www.linuxjournal.com/article/4352?page=0,0, December 2000.

[13] Kathryn Breininger, Joseph M. Chiusano, Suresh Damodaran, Mike DeNicola, Anne Fischer, Sally Fuger, Jong Kim, Kyu-Chul Lee, Joel Munter, Farrukh Najmi, Joel Neu, Sanjay Patil, Neal Smith, Nikola Stojanovic, Prasad Yendluri, and Yutaka Yoshida. Oasis/ebxml registry services specification v2.0, April 2002.

[14] 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, June 2009.

[15] Eun-Kyu Byun and Jin-Soo Kim. Dynagrid: An adaptive, scalable, and reliable resource provisioning framework for wsrf-compliant applications. *Journal of Grid Computing*, 7(1):73–89, March 2009.

[16] Bin Chen, Nong Xiao, Zhiping Cai, Fuyong Chu, and Zhiying Wang. Virtual disk image reclamation for software updates in virtual machine environments. *Networking, Architecture, and Storage, International Conference on*, 0:43–50, 2009.

[17] A.L. Chervenak, R. Schuler, M. Ripeanu, M.A. Amer, S. Bharathi, I. Foster, A. Iamnitchi, and C. Kesselman. The globus replica location service: Design and experience. *IEEE Transactions on Parallel and Distributed Systems*, 20(9):1260–1272, Sept 2008.

[18] Vidyanand Choudhary. Software as a service: Implications for investment in software development. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 209a –209a, jan. 2007.

[19] Jeffrey A. Clark and Dhiraj K. Pradhan. Fault injection. *Computer*, 28(6):47–56, 1995.

[20] Science Clouds. http://scienceclouds.org/marketplace/, January 2011.

[21] OpenStack community. Openstack open source cloud computing software. http://www.openstack.org/, January 2011.

[22] T. Delaittre, T. Kiss, A. Goyeneche, G. Terstyanszky, S.Winter, and P. Kacsuk. Gemlca: Running legacy code applications as grid services. *Journal of Grid Computing*, 3(1-2):75–90, June 2005.

[23] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15(3):313–341, 2008.

[24] A. Epstein, D.H. Lorenz, E. Silvera, and I. Shapira. Virtual appliance content distribution for a global infrastructure cloud service. In *INFOCOM, 2010 Proceedings IEEE*, pages 1 –9, 2010.

[25] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[26] Raimar Falke, Raimund Klein, Rainer Koschke, and Jochen Quante. The dominance tree in visualizing software dependencies. In *Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2005.

[27] M. Feller, I. Foster, and S. Martin. Gt4 gram: A functionality and performance study. In *TeraGrid Conference*, 2007.

[28] PUB FIPS. Secure hash standard. Technical report 180-3, NIST, October 2008.

[29] J. Fontán, T. Vázquez, L. Gonzalez, RS Montero, and IM Llorente. Opennebula: The open source virtual machine manager for cluster computing. In *Open Source Grid and Cluster Software Conference*, San Francisco, CA, USA, May 2008.

[30] I. Foster, H. Kishimoto, A. Savva, et al. The open grid services architecture, version 1.5. http://ogf.org/documents/GFD.80.pdf, October 2006.

[31] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10, DOI: 10.1109/GCE.2008.4738445, 2009. IEEE.

[32] Ian T. Foster. Globus toolkit version 4: Software for service-oriented systems. *Journal of Computer Science and Technology*, 21(4):513–520, 2006.

[33] Keisuke Fukui. Application contents service specification 1.0, Sept 2006.

[34] David Geer. The os faces a brave new world. *Computer*, 42:15–17, October 2009.

[35] Andrew Harrison and Ian J. Taylor. Dynamic web service deployment using wspeer. In *Proceedings of 13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies*, pages 11–16. Louisiana State University, February 2005.

[36] T. Ho and D. Abramson. A unified data grid replication framework. In *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, 2006.

[37] Google Inc. App engine. https://appengine.google.com/, February 2011.

[38] Microsoft Inc. Windows azure services platform. http://www.microsoft.com/windowsazure/, February 2011.

[39] Rackspace US Inc. Rackspace cloud hosting solutions. http://www.rackspacecloud.com/, January 2011.

[40] K. Keahey, I. Foster, T. Freeman, X. Zhang, and D. Galron. Virtual workspaces in the grid. ANL/MCS-P1231-0205, 2005.

[41] Katarzyna Keahey and Tim Freeman. Contextualization: Providing one-click virtual clusters. In *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 301–308, Washington, DC, USA, 2008. IEEE Computer Society.

[42] Katarzyna Keahey, Mauricio Tsugawa, Andrea Matsunaga, and Jose Fortes. Sky computing. *IEEE Internet Computing*, 13(5):43–51, 2009.

[43] Kate Keahey, Matei Ripeanu, and Karl Doering. Dynamic creation and management of runtime environments in the grid. In *Workshop on Designing and Building Web Services*, 2003.

[44] Gabor Kecskemeti, Peter Kacsuk, Gabor Terstyanszky, Tamas Kiss, and Thierry Delaitre. Automatic service deployment using virtualisation. In *Proceedings of 16th Euromicro International Conference on Parallel, Distributed and network-based Processing (PDP 2008)*, pages 628–635, Tolouse, France, February 2008. IEEE Computer Society.

[45] Attila Kertész, Gábor Kecskeméti, and Ivona Brandic. An sla-based resource virtualization approach for on-demand service provision. In *Proceedings of the International Conference on Autonomic Computing, 3rd International Workshop on Virtualization Technologies in Distributed Computing*, pages 27–34, 2009.

[46] Attila Kertesz, Gabor Kecskemeti, and Ivona Brandic. Autonomic sla-aware service virtualization for distributed systems. In *Proceedings of the 19th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2011)*, Ayia Napa, Cyprus, February 2011. IEEE.

[47] Heather Kreger, Kirk Wilson, and Igor Sedukhin. Web services distributed management: Management of web services (wsdm-mows) 1.1. web, August 2006.

[48] Ivan Krsul, Arijit Ganguly, Jian Zhang, José A. B. Fortes, and Renato J. Figueiredo. Vmplants: Providing and managing virtual machine execution environments for grid computing. In *Proceedings of the ACM/IEEE SC2004 Conference on High Performance Networking and Computing*, pages 1–7. IEEE Computer Society, November 2004.

[49] Xiao Ling, Hai Jin, Song Wu, and Xuanhua Shi. Vmcol: A collector of garbage for virtual machine image files. In Tai-Hoon Kim, Stephen S. Yau, Osvaldo Gervasi, Byeong Ho Kang, Adrian Stoica, and Dominik Slezak, editors, *FGIT-GDC/CA*, volume 121 of *Communications in Computer and Information Science*, pages 74–83. Springer, 2010.

[50] Amazon Web Services LLC. Amazon elastic compute cloud. http://aws.amazon.com/ec2/, 2009.

[51] Stephane Manuel. Classification and generation of disturbance vectors for collision attacks against sha-1. *Designs, Codes and Cryptography*, 59(1-3):247–263, April 2011.

[52] Distributed Management Task Force. Open virtualization format specification, version 1.1. http://dmtf.org/sites/default/files/standards/documents/DSP0243_1.1.0.pdf, January 2010.

[53] Public EC2 Amazon Machine Images. http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryID=171, 2010.

[54] The Nimbus Project. http://www.nimbusproject.org/, 2010.

[55] Vizioncore Inc. voptimizer, optimization of virtual machine size and performance, 2008.

[56] VMWare public virtual appliances. http://www.vmware.com/appliances/, 2010.

[57] Hideo Nishimura, Naoya Maruyama, and Satoshi Matsuoka. Virtual clusters on the fly - fast, scalable, and flexible installation. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '07, pages 549–556, Washington, DC, USA, 2007. IEEE Computer Society.

[58] Daniel Nurmi, Richard Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In Franck Cappello, Cho-Li Wang, and Rajkumar Buyya, editors, *CCGRID*, pages 124–131. IEEE Computer Society, 2009.

[59] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. "big"' web services: making the right architectural decision. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 805–814, New York, NY, USA, 2008. ACM.

[60] GridWay Project. Gridway amazon ec2 ami. http://www.gridway.org/doku.php?id=documentation:howto:virtual_appliance_amazon.

[61] Naregi project. The naregi acs implementation. http://forge.gridforum.org/sf/go/projects.acs-wg/frs, January 2007.

[62] Li Qi, Hai Jin, Ian T. Foster, and Jarek Gawor. Hand: Highly available dynamic deployment infrastructure for globus toolkit 4. In *Proceedings of 15th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2007)*, pages 155–162, Los Alamitos, CA, USA, February 2007. IEEE Computer Society.

[63] Abhishek Singh Rana, Frank Würthwein, Kate Keahey, Timothy Freeman, et al. An edge services framework (esf) for egee, lcg, and osg. In *Proceedings of Computing in High Energy Physics (CHEP06)*, T.I.F.R. Mumbai, India, February 2006.

[64] K. Ranganathan and I. Foster. Computation scheduling and data replication algorithms for data grids. In *INTERNATIONAL SERIES IN OPERATIONS RESEARCH AND MANAGEMENT SCIENCE*, pages 359–376. Kluwer Academic Publishers, 2003.

[65] Felix Rauch, Christian Kurmann, and Thomas M. Stricker. Partition cast — modelling and optimizing the distribution of large data sets in

pc clusters. In *Proceedings European Conference on Parallel Computing Euro-Par 2000*, August 2000.

[66] Felix Rauch, Christian Kurmann, and Thomas M. Stricker. Partition repositories for partition cloning - os independent software maintenance in large clusters of pcs. In *Proceedings of the IEEE International Conference on Cluster Computing 2000*, Chemnitz, Germany, November 2000.

[67] rPath. Apache appliance. http://www.rpath.org/project/aa/releases, 07 2010.

[68] rPath - rBuilder. http://www.rpath.com/rbuilder/.

[69] Marta Sabou and Jeff Pan. Towards semantically enhanced web service repositories, 2007.

[70] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 167–176, New York, NY, USA, October 2005. ACM.

[71] Constantine Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Virtual appliances for deploying and maintaining software. In *LISA '03: Proceedings of the 17th USENIX conference on System administration*, pages 181–194, Berkeley, CA, USA, 2003. USENIX Association.

[72] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 377–390, 2002.

[73] Sunita Sarawagi and Anuradha Bhamidipaty. Interactive deduplication using active learning. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '02, pages 269–278, New York, NY, USA, 2002. ACM.

[74] M. Schmidt, N. Fallenbeck, M. Smith, and B. Freisleben. Efficient distribution of virtual machines for cloud computing. In *Proceedings of 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2010)*, pages 567 –574, Pisa, Italy, February 2010. IEEE Computer Society.

[75] Matthew Smith, Thomas Friese, and Bernd Freisleben. Hot service deployment in an ad hoc grid environment. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 75–83, New York, NY, USA, 2004. ACM.

[76] Nanda Susanta and Chiueh Tzi-Cker. A survey on virtualization technologies. Technical report, Stony Brook University, February 2005.

[77] Gabor Szmetanko. Virtual appliance acs. http://sourceforge.net/projects/vaacs/, June 2008.

[78] Vanish Talwar, Dejan Milojicic, Qinyi Wu, Calton Pu, Wenchang Yan, and Gueyoung Jung. Approaches for service deployment. *Internet Computing*, 9:70–80, March-April 2005.

[79] J. Tatemura. Cddlm configuration description language specification, version 1.0, January 2007.

[80] Sebastien Tixeuil, William Hoarau, and Luis Silva. An overview of existing tools for fault-injection and dependability benchmarking in grids, October 2006.

[81] Peter Toft and Steve Loughran. Configuration description, deployment and lifecycle management working group (cddlm-wg) final report. http://www.ogf.org/documents/GFD.127.pdf, 2008.

[82] SUSE Appliance tookit. Suse galery. http://susegallery.com/, January 2011.

[83] Andrew Tridgell and Paul Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, June 1996.

[84] tuxdistro. Vmware lamp virtual appliance. http://www.vmware.com/appliances/directory/54966, 01 2011.

[85] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Computer Communication Review*, 39:50–55, December 2008.

[86] L. Youseff, M. Butrico, and D. Da Silva. Toward a unified ontology of cloud computing. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10, DOI: 10.1109/GCE.2008.4738443, 2009. IEEE.

[87] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 18:1–18:14, Berkeley, CA, USA, 2008. USENIX Association.

Part IV

APPENDIX

A

VIRTUAL APPLIANCE DEFINITIONS

The purpose of this chapter is the introduction of the virtual appliances used during the validation of the proposed AVS service, the active repositories and the optimization facility. The chapter defines the utilized appliances in the following sections in a tabular format.

Each virtual appliance definition is presented as a table with four rows. First, the "*target functionality*" row provides a textual description of the given virtual appliance. It discusses the intended target functionality and use of the specified appliance. For details, see service packages in Section 3.3.1.

Second, "*complexity*" classifies these virtual appliances according to the complexity definitions of Section 8.2. Appliances could participate in the various validation scenarios of the proposed architecture according to their complexity.

Next, the row of "*construction*" offers a step-by-step algorithm for creating the initial virtual appliance. The description assumes the use of the xen-tools appliance creation system (e.g. by running `xen-create-image`) available on the initial host of the machines. I have used the default xen-tools package of the Debian GNU/Linux 5.0. The different virtual appliances are created with the same method and they are all built on the same Debian base system, therefore, the resulting virtual appliances will share common parts. These parts are required by the Equation 8.3 in order to allow their later identification by the decomposition algorithm.

Finally, the "*validator algorithm*" row shortly describes the validation steps for each appliance. This algorithm is used during the execution of the size optimization algorithm (see Section 5.1.2.2) and the proof of concept scenario (detailed in Section 8.3). The validator ensures that modified virtual appliances still maintain their intended target functionality.

A.1 THE SSH VIRTUAL APPLIANCE

Target functionality	The SSH virtual appliance should offer a virtual machine image that provides *remote execution and transfer* capabilities. Therefore, this is a general-purpose virtual appliance that enables the execution of arbitrary code. As a prerequisite, the user has to transfer the executable and its dependencies before execution.
Complexity	Minimal
Construction	1. Create a virtual machine image for Xen with xen-tools of debian. 2. Start up the created VM image. 3. Add the ssh daemon and the rsync transfer utility. 4. Enable remote ssh based root logins in the VM.
Validator algorithm	1. Create a shell script that prints out "hello world". 2. Transfer the previously created shell script to the target virtual machine with rsync. 3. Remotely execute the transferred script. 4. Check whether the execution returns with "hello world". If not then the virtual machine is not valid.

A.2 THE APACHE VIRTUAL APPLIANCE

Target functionality	The aim of the Apache appliance is to provide an HTTP server that is capable of *serving static html pages* to its users. As a prerequisite, the user has to transfer the static html pages that should be offered by the server.
Complexity	Medium
Construction	1. Create a virtual machine image for Xen with xen-tools of debian. 2. Start up the created VM image. 3. Add the ssh-, apache daemons and the rsync transfer utility.
Validator algorithm	1. Create an HTML document that has an html header with the text of "hello world". 2. Transfer the previously created HTML document to the apache web server's folder on the target virtual machine. 3. Download the pre transferred HTML file with an HTTP request. 4. Check for the html header with the text "hello world". If the header is not present then the target virtual machine is not valid.

A.3 THE GEMLCA VIRTUAL APPLIANCE

Target functionality	The GEMLCA appliance provides a GEMLCA grid service that is composed of a *GRAM4 submitter and a legacy code repository*. A deployed service should be able to execute applications on an arbitrary Globus Toolkit 4 GRAM [27], and it should be able to register, manage and destroy legacy codes in its repository. This virtual appliance is included among the test appliances to present the behavior of my algorithms with more complex services.
Complexity	High
Construction	1. Create a virtual machine image for Xen with xen-tools of debian. 2. Start up the created VM image. 3. Install the gridftp utility of globus toolkit 4 with all its dependencies. 4. Install GEMLCA service. 5. Configure GEMLCA service with GRAM 4 submission capabilities.
Validator algorithm	1. Create a single shell script to print out the contents of a file that by default contains the text "hello world". 2. Register the previously created script and default file with the target GEMCLA service. 3. Run the registered application on a predefined GRAM4 without any input files, and then transfer its results to the local machine. 4. Execute the application again with a custom input file containing the text "hello world", and transfer its results back. 5. Check whether both files were received with the correct content, if not then the target GEMLCA service is not valid.