unifying cloud
and carrier networks

Deliverable D3.2

Detailed functional specification and algorithm description

Dissemination level	PU
Version	1.0
Due date	30.04.2015
Version date	18.06.2015

This project is co-funded
by the European Union

Document information

Editors and Authors:

Editors: Pontus Sköldström (ACREO)

Contributing Partners and Authors:

ACREO	Pontus Sköldström, Manxing Du
BME	Balász Sonkoly, János Czentye, András Gulyás, Gábor Rétvári
DTAG	Mario Kind
EAB	Ahmad Rostami, Xuejun Cai, Wolfgang John
EHU	Jokin Garay, Jon Matias
ETH	David Jocha, Róbert Szabó
iMinds	Wouter Tavernier, Sahel Sahhaf, Steven van Rossem
OTE	George Agapiou
TI	Vinicio Vercellone
TUB	Matthias Rost

Project Coordinator

Dr. András Császár

Ericsson Magyarország Kommunikációs Rendszerek Kft. (ETH) AB

KONYVES KALMAN KORUT 11 B EP

1097 BUDAPEST, HUNGARY

Fax: +36 (1) 437-7467

Email: andras.csaszar@ericsson.com

Project funding

7th Framework Programme

FP7-ICT-2013-11

Collaborative project

Grant Agreement No. 619609

Legal Disclaimer

Important dates:

April 9: Draft versions of separate sections

April 16: Review of separate sections

April 27: All sections revised based on section reviews, all major updates completed

April 30: First complete draft of D3.2

May 15: External review completed

May 30: Revisions based on external review completed

June 14: Updates based on discussions during plenary meeting

June 22: Submission of document

Table of contents

1 Introduction **12**

2 Programmability interfaces and data models **15**

2.1 Efficient recursive NF-FG data queries 15

3 Service decomposition **17**

3.1 Introduction to service decomposition 17

3.1.1 Advantages of service decomposition 18

3.1.2 Limits and drawbacks of decomposition 19

3.1.3 Why and when to decompose 20

3.1.4 Which actors perform the decomposition? 20

3.2 Atomic blocks 21

3.2.1 Hardware-defined atomic blocks 21

3.2.2 Software-defined atomic blocks 22

3.2.3 Virtualization/isolation technologies 26

3.2.4 Atomic Block overview 32

3.3 Monolithic VNF's and Services constructed from atomic blocks 33

3.3.1 Elastic Router use case 33

3.3.2 Flow-based Network Access Control 35

3.3.3 Gateway decomposition 37

3.3.4 Service Decomposition trade-offs 39

4 Scalable orchestration algorithms **41**

4.1 Context of the resource orchestration problem 41

4.2 Background on the Service Chain Embedding Problem 44

4.3 Online Algorithms 45

4.3.1 End-to-end Embedding on Dense (Multi-)Graphs 46

4.3.2 Virtual Cluster Embedding Problem 48

4.3.3 Flow Algorithm for Optimally Embedding Virtual Cluster Embeddings 49

4.4 Hybrid Online-Offline Cooperation 51

4.5 Pure Offline Algorithm: DiVINE 52

4.5.1 Outline of the Initial Computational Evaluation 53

4.5.2 Understanding the Out-of-the-Box Performance of the Optimal Offline Mixed-Integer Program 53
4.5.3 Initial DiVINE Evaluation 56
4.6 Service chain embedding supporting service decomposition 57
4.6.1 Problem description 57
4.6.2 Optimal solution for service decomposition w.r.t. resources footprint 58
4.6.3 Heuristic solution for service decomposition w.r.t. resource footprint 59
4.6.4 Evaluation of heuristic solution versus optimal solution 60
4.6.5 Evaluation of embedding time in heuristic solution 65
4.6.6 Potential improvements 67
4.7 Distributed orchestration 68
4.7.1 Implemented algorithms 68

5 Scalable and resilient services 71

5.1 Requirements on scalability mechanisms on BNG use-case 72
5.1.1 Reasons for BNG scaling 73
5.1.2 State classification and performance requirements 74
5.2 VNF Scaling and management of internal state 77
5.2.1 OpenNF architecture 79
5.2.2 Distributed state transfer (DiST) 82
5.2.3 Evaluation of DiST vs. original OpenNF 84
5.2.4 Conclusions on of evaluation and potential extensions 88
5.3 Service resiliency 89
5.3.1 Protection vs. restoration 90
5.3.2 Recovery scope 90
5.3.3 Failure detection 91
5.3.4 Recovery controlling component 91
5.3.5 State synchronization or migration component 92
5.3.6 NF protection example 92
5.3.7 Conclusions 94

6 Traffic steering and forwarding state 96

6.1 Theoretical approach to traffic steering 97
6.1.1 A framework for Software Defined Routing (SDR) 97
6.1.2 Classification of routing & forwarding paradigms 100
6.1.3 Trading off space across tiers 102

6.1.4 Results 105

7 Scalable orchestration architectures 107

7.1 Design and implementation of ESCAPEv2 107
7.2 Orchestration of Radio and Transport Resources 111
7.2.1 Integrating Radio Resource Control into the Orchestration Architecture 111
7.2.2 Proof of Concept Elements 112
7.2.3 Elastic Mobile Broadband Services: The Orchestration Process 113

8 Conclusions 116

9 References 119

Annex 1 Main components of ESCAPEv2 128

A.1.1 Static model of Service module 128
A.1.2Dynamic model of Service module 131
A.1.3Static model of Orchestration module 134
A.1.4Dynamic model of Orchestration module 137
A.1.5Static model of Adaptation module 139
A.1.6 Dynamic model of Adaption module 141
A.1.7Integration issues 145

List of figures

Figure 2-1: Recursive Monitoring Query 16
Figure 3-1: IDS service decomposition 17
Figure 3-2: Decomposition of a GStreamer pipeline transcoding AVI to FLV and MKV formats. 18
Figure 3-3: Routing atomic blocks as used in eg. Quagga 23
Figure 3-4: Example VNF implemented as a Click script. 24
Figure 3-5: VM a) versus Container b) architecture 26
Figure 3-6: Two Docker containers A and B. B is related to A with both links and volumes. 30
Figure 3-7: WebCache decomposed into 4 processes 31
Figure 3-8: Elastic Router Decomposition. 35
Figure 3-9: FlowNAC architecture 37
Figure 3-10: Gateway decomposition example 39
Figure 3-11: Decomposition trade-offs 40
Figure 4-1: BT network structure 42
Figure 4-2: Network embedding concept 44
Figure 4-3: A simple NF-FG to be orchestrated. 45
Figure 4-4: Inter-IXP network with multiple potential transit ISPs. 46
Figure 4-5: Acceptance ratio and resource utilization of a 28 node IXP multi-graph in the online scenario. 48
Figure 4-6: Virtual Cluster abstraction: a set of servers is connected via a central switch. 49
Figure 4-7: Flow algorithm to optimally embed virtual clusters on any topologies 50
Figure 4-8: Heuristic offline Integer Program to maximize the number of request embedded. 51
Figure 4-9: Comparison of optimal solution value with the heuristic approaches 52
Figure 4-10: Overview of the topologies contained in the Topology Zoo. 53
Figure 4-11: Impact of the different parameters 55
Figure 4-12: ECDF of reached objective gap over time. 56
Figure 4-13: 3D representation of the difference of ECDFs of DiVINE vs. the pure MIP on the Geant topology. 57
Figure 4-14: Execution time of ILP and DSBM for SGs with 5 and 10 NFs 62
Figure 4-15: Service request acceptance ratio over time 62
Figure 4-16: Average cost of accepting requests over time 63
Figure 4-17: Average cost normalized by acceptance ratio 64
Figure 4-18: Service request acceptance ratio over time in DSBM 64
Figure 4-19: Average cost of accepting requests over time in DSBM 65
Figure 4-20: Average cost normalized by acceptance ratio in DSBM 65
Figure 4-21: Embedding execution time for SGs with one decomposition 66
Figure 4-22: Embedding execution time for SGs with 5 NFs 66

Figure 4-23: Example Service Graph, topology and potential paths, with shortest combined path in bold 69
Figure 4-24: Boxplot of initial results for distributed and local SFC calculations. 70
Figure 5-1: The BNG function at the edge of a Service Provider network 72
Figure 5-2: Possible scenarios for BNG stateful scaling/resiliency mechanisms 73
Figure 5-3: OpenNF architecture, taken from [Gember-Jacobson2014] 79
Figure 5-4: OpenNF NG Move operation 80
Figure 5-5: OpenNF LF with PZLLER 81
Figure 5-6: DiST LF Move with PZLLER 83
Figure 5-7: DiST OP Move with PZLLER 84
Figure 5-8: Testbed setup 84
Figure 5-9: MoveTime for different amount of state (left), ratio between DiST and OpenNF (right). 87
Figure 5-10: Traffic pattern at Sink during the move (indicated by the boxes) at 1000 pps 87
Figure 5-11: DiST OP Move, alternate order with stream synchronization and buffering before state transfer 89
Figure 5-12: NF protection example 93
Figure 5-13: Sequence diagram for NF protection process 94
Figure 6-1: Routing & forwarding paradigms categorized by forwarding state representation. 101
Figure 6-2: Possible forwarding decision paths in a physical node 103
Figure 6-3: The “underlay forwarding vs. escalation to higher tiers” trade-off 105
Figure 6-4: Number of forwarding abstractions in the optimal SDR configuration 105
Figure 6-5: Number of forwarding abstractions in the approximate SDR configuration 106
Figure 7-1: System architecture of ESCAPEv2 109
Figure 7-2: Radio-Transport Orchestration PoC 113
Figure 7-3: State diagram of the EMBS operation 114
Figure 7-4: Changes to the NF-FG during the lifetime of a EMBS. 115
Figure 9-1: Class diagram of the Service module 129
Figure 9-2: Interaction steps during a Service Request in the Service module 133
Figure 9-3: Class diagram of the Orchestration module 136
Figure 9-4: Interaction steps during a Service Request in the Orchestration module 138
Figure 9-5: Class diagram of the Adaptation module 140
Figure 9-6: Interaction steps during a Service Request in the Adaptation module 142
Figure 9-7: Interaction steps during a request of missing Virtualizers 144

List of Tables

Table 3-1: Click Network Function examples 25
Table 3-2: Atomic Block overview 32
Table 4-1: Service provider network size 43
Table 4-2: List of compared algorithms 60
Table 5-1: DiST vs OpenNF performance data gathered at 1000 pps, with 95% CI 86

Summary

This deliverable details the functional description and algorithms for the UNIFY orchestration framework, we systematically go through the UNIFY architecture presented in [D2.2] in a top-down manner. We start at the Service Layer where the focus is on service programming, here our main technical challenge is the data models to define a service and the programmability interfaces needed to transfer and maintain the service. From the Service layer we continue down to the Orchestration layer where we have two main technical challenges, orchestration algorithms and service decomposition. In orchestration algorithms we then investigate how we can allocate, in a scalable and optimized way, the resources required by a service. In service decomposition we narrow the scope from the service to the individual network functions that makes it up and investigate how abstract service components can be broken down into a set of hardware and implementation specific atomic blocks capable of delivering the requested functionality. From the Orchestration layer we finally arrive at the Infrastructure layer where there again are two main technical challenges, management of state in the network functions and management of forwarding state in the network. We first investigate how internal state in the network functions can be managed in order to provide elastic and resilient services, which can handle increases in demand and failures without disruption. We then follow with an investigation on how forwarding state from different forwarding paradigms can be combined in an efficient and scalable manner to steer traffic between the network functions that compose a service. Finally, we arrive at our evolution of a scalable orchestration framework, ESCAPEv2, intended to provide a framework where our ideas can be prototyped and tested.

UNIFY programmability is focused on the interface between the Service Layer and the Orchestration Layer, as well as recursively through Orchestration Layers. The heart of this programmability model is the NF-FG, for which an initial definition was made in [D3.1]. Here we document two approaches to modelling the NF-FG. One is a service-centric approach which is an updated version of the earlier NF-FG, based on feedback from prototyping and ongoing work in SDOs and other projects. The second approach focuses on modelling the Virtualizer component introduced in [D2.2], responsible for presenting virtual views to higher layers. In this approach the configuration of the virtualized resources is the NF-FG. We also show the relationship of the NF-FG models to ongoing external work.

In deliverable D3.1 we introduced service decomposition, a process of breaking down a high-level service into its *atomic blocks* in order to e.g. better utilize available resources and more quickly construct new services. Here we document the pros and cons and trade-offs of this approach in the context atomic blocks as hardware blocks, executable-based software blocks, or as Click Modular Router components. In addition, an analysis is provided of the relationship of decomposition to using virtualization technology relying either on containers (e.g., Docker) or virtual machines (e.g., Xen). A more in-depth application of decomposition is performed for of a flexible router, flow-based network access control, gateway, and IDS service.

Initially a Service Graph describes the service requested by a customer, and its requirements. The Service Graph is transformed into a NF-FG through multiple steps of decompositions and mappings until it is finally instantiated on

the available infrastructure resources. Embedding an NF-FG on the available resources is not a trivial problem, especially when taking the number of requests, the size of the underlying topology and heterogeneous environments into account. In this document we present a range of ILP-based algorithms such as DiVINE for off- and on-line scenarios, with or without combining the mapping with the decomposition, as well as heuristics, enabling to scale on larger problem sets. The main idea behind decomposition-aware embedding heuristics is to rank potential service decompositions on estimated cost in order to minimize resource consumption of the resulting embedding. To further improve the scalability of embedding components, a framework is proposed for distributing these types of calculations to multiple nodes.

Elastic, scalable, and resilient network services are one of the main selling points of NFV. Supporting elastic VNFs require careful handling of the internal state of the VNF in order not to disrupt services when expanding on contracting the resource allocations. Existing NFV experimentation platforms such as OpenNF support various state migration schemes only but relies on centralized control components. We introduce a distributed state transfer mechanism which have been implemented and evaluated, indicating substantial gains in migration time and required bandwidth. Compared to traditional hardware-based network functions that have been engineered for resiliency, the software-based network functions that NFV relies on is intended to run on hardware platforms with typically higher failure rates, requiring new resiliency mechanisms on the software layer. In the context of service resilience, the same set of state migration mechanisms might be re-used in support of protection or restoration mechanisms at the scope of a single NF, or subgraphs of the NF-FG relying either on service-specific control functionality (Control NFs) or on the existing orchestration and control functionality of the underlying UNIFY platform. We also provide an analysis of the state management requirements of the Broadband Network Gateway (BNG) which plays a crucial role in a Service Provider environment as an IP edge router that provides aggregation capabilities (e.g. IP, PPP) between the access network and the transport network which also includes functionality for subscriber management, advanced IP processing, including QoS, and enhanced traffic management capabilities.

In the UNIFY architecture, abstract forwarding information on how traffic should traverse the network functions in a service is stored in NF-FGs in our programmability framework. At the lower layers, this abstract information must be translated into different types of forwarding states that can inserted in the network elements as different types of flow rules, tunnel information or packet header fields. Managing that state and steering traffic efficiently in a flexible and controllable way is an indispensable part of an orchestration framework. The advent of software defined networking and virtualization pushes the challenge of routing to extremity as the flexibility of SDN architectures allows deploying many existing routing approaches at the same time on the same network. We take a first step towards the characterization of fast and scalable SDN routing strategies which can be foundations of service chaining architectures in the future. We argue that our approach, Software Defined Routing, is an approach that can effectively distribute forwarding state between nodes using different routing paradigms according to the resources and desired functions of the network.

While theoretical work is valuable, many good theoretical concepts fail or perform less than expected when actually implemented. To test and refine our processes and algorithms a proof-of-concept system has been designed, with the focus of providing a scalable and flexible way of integrating components developed by partners in the project. This proof-of-concept system has evolved from the earlier implementation ("ESCAPE") based on experience gained and the updated architecture presented in [D2.2]. ESCAPE relies on Mininet, Click, NETCONF and POX and implements the 3 layers of UNIFY architecture, namely, Infrastructure Layer (IL), Orchestration Layer (OL), Service Layer (SL). Increased scalability and improved integration functionality with additional components or other prototypes is enabled through the implementation of virtualizer functionality supporting BiS-BiS abstractions, and control adaptation functionality supporting OpenStack and Docker-based domains. Controlling Radio Access Networks in the UNIFY architecture is possible through considering the RAN and associated controller as a lower-layer orchestration domain. As the RAN controller only operates on physical resources, the BiS-BiS model will need to be support physical resources such as BBU unites, as indicated in an initial Proof-of-Concept.

1 Introduction

Starting from a multi-layer UNIFY architecture as detailed in [D2.1] and [D2.2], an initial programmability framework involving a range of processes, data models, interfaces and orchestration concepts was defined in [D3.1]. This deliverable will refine these concepts, interfaces, functions, as well as algorithms providing the core components of the UNIFY service programming and orchestration platform. The goal of this section is two-fold: i) providing a more quantitative context of the targeted orchestration environment and related requirements, and ii) introducing the core technical challenges of the service programming and orchestration platform which will be addressed later on in this document.

Orchestration is fundamentally enabled by the top-down and bottom-up service programming processes in which multiple layers and components interact with each other. As identified in earlier deliverables, the focus of **UNIFY programmability** is on the interface between the Service Layer and the Orchestration Layer, as well as on between (hierarchically connected) Orchestration Layers. The core information model impacting this interface is the Network Function-Forwarding Graph (NF-FG). The latter plays a dual role within the framework:

- The role of exposing (virtualized) network and cloud resources
- The role of mapping a Service request (Service Graph) to exposed (virtualized) resources.

An initial **NF-FG data model** has been specified in [D3.1] and was the starting point for a range of prototypes in this and other work packages. However, depending on the particular purpose and context of the prototype, the resulting implementations and structure of these models has changed and diverged. In addition, other research projects, standardization bodies and other initiatives have developed their own data models and interfaces fulfilling a purpose which might be related to the NF-FG. Section 2 documents the two NF-FG models we are currently evaluating. The first model has evolved from the initial NF-FG model with the focus on describing the requested service as it passes down through the layers of the architecture before being instantiated on the infrastructure. The second NF-FG model takes a different approach and focuses on the Virtualizer component responsible for presenting a virtual network view to higher layers and the configuration of resources in that view, which combines to an NF-FG. We relate, and consolidate these models with models developed in parallel outside of the project, such as those defined by OASIS TOSCA or OpenStack Heat. The focus of this analysis is on the differences and added value of data models themselves; however technology-specific aspects will be addressed briefly as well. Future consolidation will be encouraged through the use of a UNIFY NF-FG library module providing a common set of functionality for NF-FG parsing, interpretation and manipulation.

The ambitious concept of **service decomposition** was introduced in D3.1 as a step-wise translation of high-level (compound) NFs into more elementary or atomic NFs, which can eventually be mapped onto the infrastructure. Within the context of service programming processes, service decomposition might occur at the Service Adaptation Layer, or on any of the (potentially stacked) Orchestration Layers. The merit of service decomposition is that it enables lower layers to refine high-level components in order to optimize its internal resource consumption or performance, for example by decomposing functions into functions which can be decomposed on already used servers with spare capacity, rather than instantiating on network functions on additional servers. Decomposing network functions and services into smaller blocks also enables network function and service developers to reuse already available functionality. A more in-depth analysis of these tradeoffs and a path towards a more quantitative approach in characterizing service decomposition is handled in Section 3.

Resource orchestration refers to the process of handling service requests in the form of NF-FGs, mapping them to individual (infrastructure) resource elements of a resource model obtained by collecting information received by network- and server controllers. Subgraphs of the NF-FG might be recursively delegated to lower-layer orchestrators, which ultimately need to map their components to infrastructure resources. The challenges related to orchestration relate to two main aspects: i) scalability of the process and involved algorithms with respect to the number of requests, the size of the underlying network topology and involved components, and ii) the technology dependence of resource orchestration. The second challenge relates to the particular nature of network functions, the particular way they can be decomposed in, and their performance given a set of infrastructure-related characteristics such as hardware-acceleration, caching capabilities and others. Section 4 positions the resource orchestration challenge within the context of the virtual network embedding problem and investigates the impact of online vs. offline algorithms, considers the relationship and added value of service decomposition within the embedding algorithms, investigates a set of techniques to improve scalability of the orchestration infrastructure and investigates a number of ways to handle technology-related aspects to orchestration.

Support for providing **elastic network services** in one of the most important merits of NFV. Whereas traditional telecom services built on dedicated hardware devices usually require additional investments and manual re-provisioning processes in order to scale the service to support a larger number of customers, flows or just plain throughput, NFV relies on software-based implementations of Network Functions which can be deployed on demand on generic-purpose hardware (servers). However, traditional hardware-implemented network functions are engineered in order to provide high reliability with respect to individual component failures, in order to guarantee the integrity of involved network and session state up to certain degrees. Section 5 specifies and evaluates a range of mechanisms in order to enable effective and reliable state migration between VNFs. These have been evaluated in an actively maintained OpenNF platform. In addition, the section details how these mechanisms might be re-used in order to provide high availability in NFV-based services.

A **scalable forwarding plane** to support the Network Functions and their interconnections is another requirement of the UNIFY architecture. With many years of research onto different routing and forwarding paradigms, there is still no clear answer to what that forwarding plane should be. In Section 6 we propose a framework for Software Defined Routing (SDR) that leverages the power and flexibility given by SDN to combine multiple forwarding paradigms in an attempt to better manage the required forwarding state in the network. In this section we take the first steps towards defining fast and scalable SDN routing strategies that can be the foundation of future service chaining architectures. This is done through a 3-tiered architectural model which can incorporate many existing routing paradigms.

The true value of a service programming and orchestration framework only becomes apparent if its concepts, processes and algorithms are actually implemented in a proof-of-concept, which on its turn can provide feedback in order to refine the framework and its components. This supports an iterative process which avoids the traps of overly complex theoretic frameworks without addressing lower-level, technological or implementation-related issues. The above observation was the departing point of a set of component developments focusing on the implementation of: i) the network dataplane and control functionality building further on a mathematical foundation for traffic steering and forwarding state management, and ii) a global emulation platform in order to validate the main layers, components and algorithms of the UNIFY service programming and orchestration framework. Section 7 provides an overview on the implementation side of important **architectural components of the service programming and orchestration framework** in the evolved ESCAPE emulation environment, and gives an overview on the impact of and relationships to the framework with respect to the control of radio and transport networks.

2 Programmability interfaces and data models

The main information element of the UNIFY architecture is the Network Function Forwarding Graph, introduced in [D3.1]. Based on the input and lessons learned in the different prototypes, and the evolution of our understanding of the requirements for the main components and features covered in the final architecture described in [D2.2], we are evaluating two NF-FG model definitions, one service-centric based on the model introduced in [D3.1] and one approach based on modelling the Virtualizer component introduced in [D2.2].

As we expect these models to further evolve based on prototyping experience within the project these NF-FG models and the current state of the art models we relate them to are described in a separate, living, document that can be more easily updated as more experience is gathered, [D3.2a]. Other sections of this document (D3.2) are not as tightly bound to the exact specification of the NF-FG and therefore have not been placed in D3.2a.

Finally, section two introduces a recursive query language for gathering parameters on Service Graphs by recursing through the orchestration hierarchy.

2.1 Efficient recursive NF-FG data queries

In NFV environments, operators or developers sometimes need to query the performance of a Virtualized Network Function. In existing systems this is usually done by mapping the performance metrics of NFV to primitive physical network function or elements statically and manually when the virtualized service is deployed. However in UNIFY a multi-layer hierarchical architecture is adopted, and the VNF and associated resources may be composed recursively in different layers of the architecture. That will put greater challenge on the performance of query on VNFs as the mapping of VNF performance metrics from service layer to cloud infrastructure is more complex compared to cloud infrastructure with a single layer orchestration. It is important to have an automatic and dynamic way to decompose of the performance query in a recursive manner. We propose to use declarative language such as Datalog to perform recursive queries on NF-FGs.

As shown in Figure 2-1, the Query Engine is responsible to receive the query scripts written in variant of a declarative language (i.e., Datalog) from receivers like VNF developers and infrastructure operators, and then automatically translate these queries. The queries should be translated into corresponding database queries based on the resource graphs described in the NF-FGs, which may contain multiple nested NF-FG corresponding to different levels of orchestration. Based on the instruction of the query scripts, the Query Engine will parse the NF-FGs and decompose the performance metrics in a recursive way until some primitive resources (such as CPU of VM, delay between two physical network elements) are identified or terminated according to instruction. Then the query engine will query the distributed database containing monitoring results for these decomposed performance metrics and aggregate them according to the query scripts. The distributed monitoring database stores the measurement results collected by monitoring functions implemented in WP4, a monitoring function consists of a Monitoring Function Control Application, and one or more Observability Points. An Observability Point in turn consists of one or more pairs of node-local control and data plane components (LCP and LDP respectively) which are the ultimate source of monitoring information. The Query Library is used to store some pre-defined query template or library developed by the receivers or service provider. A more detailed definition of the language will be described in upcoming milestone M3.4 of WP3. And further detail on the design of the query engine and sample usage of the language will be documented in D4.2 of WP4.

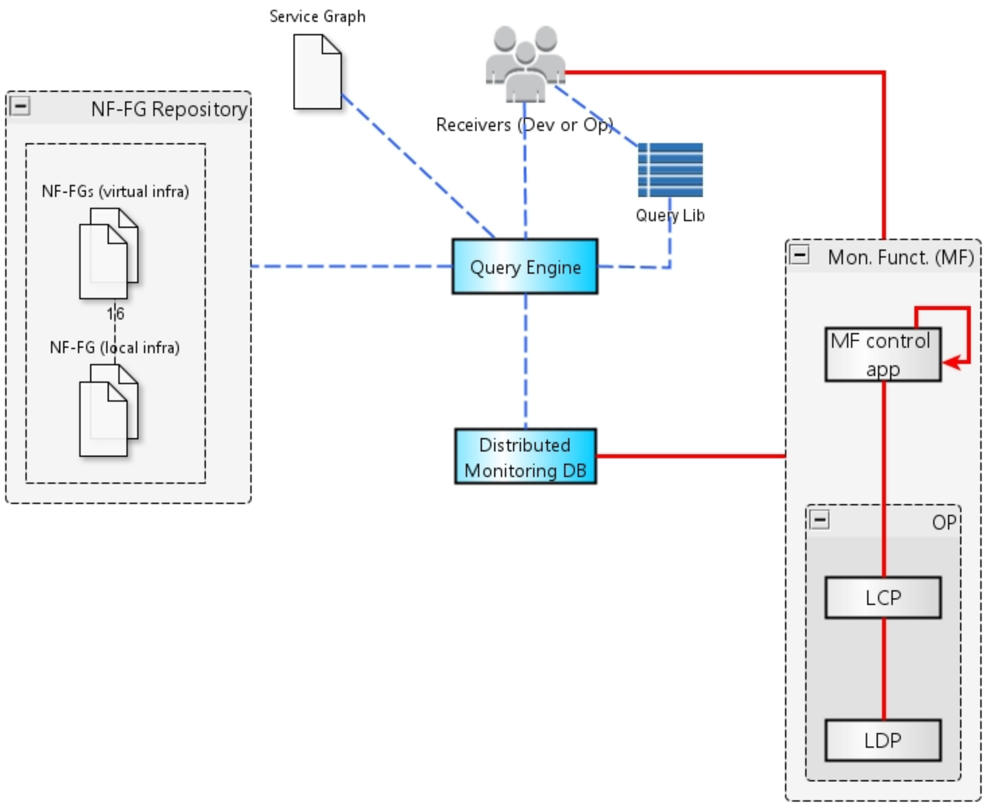


Figure 2-1: Recursive Monitoring Query

3 Service decomposition

Model-based service decomposition was introduced in [D2.2], which allows for a step-wise translation of high-level (compound) NFs into more elementary or atomic NFs, which can eventually be mapped onto the infrastructure. The decomposition processes and methods were described in [D3.1], including NF-IB based decomposition as well as ControlApp-driven decomposition. After a short introduction we address aspects related to the benefits of decomposition and when and to what extent decomposition should be used. We then investigate various potential atomic blocks that can be composed into larger functions or services. Finally we provide several examples of how services can be composed from atomic blocks.

3.1 Introduction to service decomposition

One of the main goals of the UNIFY framework is to allow a quick and reliable deployment of services across the whole network. It is clear that service decomposition is an important aspect in this context. The service decomposes into discrete NF's, which we will define as atomic blocks later on in this section. This modular approach offers a more flexible way of mapping, or orchestrating, the service onto a universal network infrastructure. So the service decomposition is a logical first step before the orchestration can begin. Also after the initial orchestration and deployment of a service, reiterations of the decomposition and orchestration can occur due to scalability needs of NF's. To illustrate this, we can shortly recapitulate some examples given in earlier deliverables. In the following subsections we elaborate on the different aspects of decomposing a service.

In [D2.2] Section 3.3.2 a service decomposition example was given to show the monolithic vs. decomposed network functions and the control and data plane split. This example was about an Intrusion Detection System (IDS), whose role is to identify and block malicious traffic. It was shown how an IDS can be decomposed as depicted in Figure 3-1.

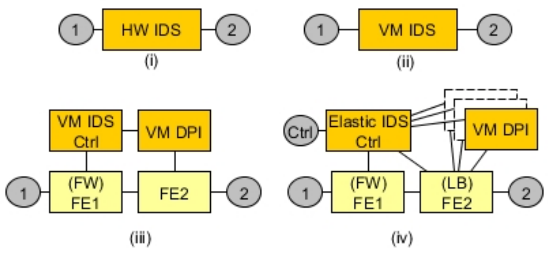


Figure 3-1: IDS service decomposition (i) a hardware based monolithic component; (ii) a monolithic VM; (iii) a data and control plane split design; (iv) and a scalable split realization.

In [D3.1] a video use case was described, which from a business point of view has a high potential of increasing the traffic in the network due to the social media advancement. A critical network function for transferring video is the transcoder, which is responsible for adapting multimedia streams for different platforms. Transcoding could be done in order to for example lowering the resolution before sending the video to a mobile device with a small screen or changing from one encoding format to another which is better supported by the end device. Interactive two-way communication services, peer to peer gaming, audio visual conferencing, etc. require that adaptation of the IP media packet streams is required between different formatted devices in both directions and in real time. Without the advantages of buffering the GStreamer framework supports constructing pipelines of "atomic" functions in order to perform various multimedia tasks such as transcoding [GStreamer]. With GStreamer the pipelines are executed on a single node, but one can imagine constructing similar pipelines in a distributed fashion utilizing multiple VNFs. In Figure 3-2 an example of transcoding AVI formatted files into FLV and MKV formats using GStreamer is shown.

The boxes represent different functions that can send data to the following function, with the different colours to represent how they could be grouped into different VNFs for decoding, scaling, encoding, and packaging. The atomic functions needed in the transcoding pipeline can be flexibly delivered even by third party operators for offering higher quality audio or video services with less bandwidth.

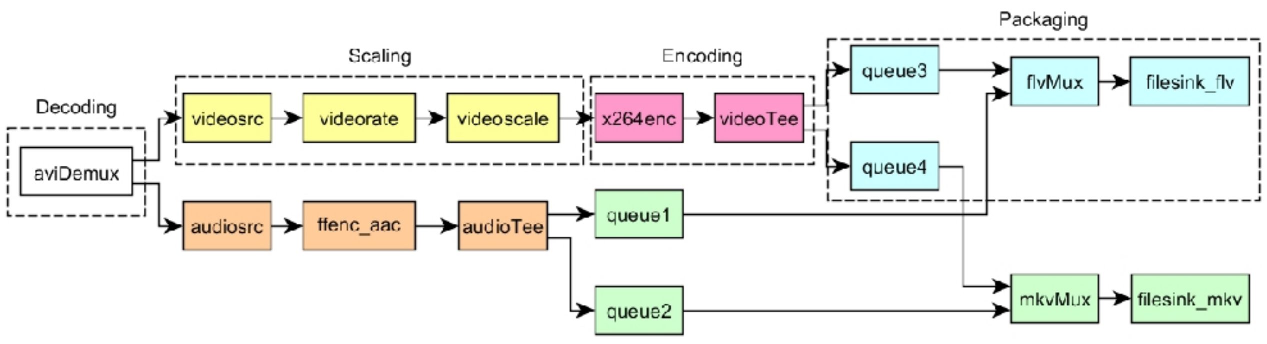


Figure 3-2: Decomposition of a GStreamer pipeline transcoding AVI to FLV and MKV formats. The different colours indicate groups of functions that could potentially be placed in different VNFs.

3.1.1 Advantages of service decomposition

There are many motivations for service decomposition. One of them is well-known from software development, namely the efficiency of *reusing existing components* to build new modules. It results in increased development speed and reduced development costs [Bazilchuk2005]. Relying on generic functionality provided by external components provides flexibility, as seen in software development in case of software libraries. The external libraries can be updated, bug fixed, optimized, as long as the interface remains the same the application making use of the library does not have to be changed.

Multiple implementations of the same functionality may also be possible. This allows a generic NF type to be mapped to different platforms (e.g. x86, ARM, OF switch ...). There can be multiple implementations *optimized for different* aspects (e.g. one using more CPU but less storage, while another using less CPU but more storage), where the most appropriate one can be chosen based on various factors. In all these cases the User/Operator does not need to be aware of what resources are available when requesting the generic service.

Decomposition can enable advanced *resource optimization* by jointly optimizing decomposition and resource allocation (also see Section 4.6). On one hand, taking available resources into account when deciding which of the potential decompositions to embed in the infrastructure will increase the likelihood that the embedding can be satisfied. So providing various decompositions for the same service makes the service available under more conditions, as well as enables further resource usage optimization.

From a user's perspective service decomposition is advantageous, because there are cases where with decomposition *higher performance* or better KPI/KQI can be achieved than without decomposing the service. For example, the delay sensitive part of the NF can be placed close to the user, while the computation intensive part can be placed in a datacenter or in a dedicated hardware appliance.

From operator perspective service decompositions can result in *better resource usage*, like load balancing to the appropriate infrastructure components, or sharing resources or even NFs between users based on e.g. statistical multiplexing. This can reduce the cost while providing the services to the users.

3.1.2 Limits and drawbacks of decomposition

Decomposition of functions to subcomponents has certain limits and overheads. There can be for example *metadata* associated to packets of service traffic, to be communicated between the (decomposed) components. Consider the Click Modular Router platform where a colour can be assigned to packets. This meta-information is to be transmitted with the original packet between Click components (potentially deployed on remote hosts). There can be configuration or other type of data not directly associated to packets, which must be shared among the components.

There is *communication overhead* between components, compared to the non-split alternative, or compared to the locally-split alternative. Once a function is decomposed, in the generic case it's up to the RO where to place them, so they should be prepared to handle this. There can be additional communication/networking aspects to be considered, like delay, packet loss, reduced throughput, costs.

Taking available resources into account when deciding which of the potential decompositions to embed in the infrastructure will increase the likelihood that the embedding can be satisfied. However, providing a description of available resources may pose other issues such as how to *effectively provide a summary of resources* or in the black-box service decomposition case, how to do it in a secure manner, assuming that the decompose functions may be provided by a 3rd party.

Handling the *interaction between automatic service scaling and decomposition* may also be an issue, as they should agree on how a certain NF-FG should be instantiated, in order to avoid cases where the scaling mechanism instantly modifies a newly instantiated NF-FG determined through decomposition. A potential solution to this problem is to treat scaling and decomposition as the same process while another is to keep the processes separated but mandate that the scaling logic uses the same NF models and logic as the decomposition logic. In order to perform scaling in an efficient way state internal to NF instances most likely is necessary to determine when to scale up or down. This type of internal state and state processing may be difficult to capture in a generic model.

3.1.3 Why and when to decompose

Considering the limits of decomposition, it can be investigated case by case whether/when decomposing a service is desirable. However, there are certain generic situations, when the advantages of decomposing are hard to question. Such a situation which allows decomposition is *when the interface between the components is very clear*, like just transmitting the pure user payload packets without any metadata between the components. An example can be a firewall after a NAT.

We can have a strong reason to decompose to a given NF, if there is certain *functionality realized by this NF in pure data plane*, or even more, by a dedicated appliance. In this case the speed and cost can motivate decomposition. There are cases, when certain *functionality can be realized close to the source*, thus less networking resources have be used. This can be seen as resource optimization or resource policy motivated decomposition.

From the NF development point of view, once there are proven, cheap, well-known, ready-to-use building blocks available, the developer aims to use them, by decomposing the more complex NF to make use of the existing components.

3.1.4 Which actors perform the decomposition?

Deliverable [D3.1] defines the actors relevant to programmability, which corresponds to the potential business roles as given in [D2.1]. Decomposition steps can be mapped to these actors. According to [D2.2], model-based service decomposition has a time aspect, as discussed below.

There is a decomposition model, made at design-time by the *VNF Developer*. When developing a new NF, it's up to the developer to use existing functions or to split the high level NF to more atomic ones. Of course, various alternative decompositions can be provided by the VNF Developer. These decompositions are foreseen to be stored in the NF-IB. Later the decompositions stored in the NF-IB can be extended or updated.

In the instantiation time the model can be taken from the NF-IB. As there can be multiple possible decompositions, the instantiation can dynamically take into account e.g. available resources to make the best decision according to its policies or optimization goals. This step is performed by the Orchestration Layer (developed by *the Orchestration SW Provider*, configured and operated by the *Service Provider*). According to the functional architecture described in [D2.2], the Resource Orchestrator is the functional entity performing the actual decomposition, in the sense of selecting which decomposition to use, if multiple ones are possible.

There is also a Service Graph Adaptation layer in the UNIFY architecture, which may also deal with decomposition in the RO sense, however, its task is to convert a Service Graph (SG) to a NF-FG, but not to decompose NFs in a NF-FG.

3.2 Atomic blocks

Decomposition follows a recursive, top-down approach, i.e. the *RO point of view*. The RO of a NF-FG might involve decomposition of (potentially abstract) NFs into other (potentially abstract) NFs. Resource orchestration of an NF or of a subgraph of the NF-FG might be delegated to lower-layer ROs, but ultimately, at the lowest orchestration layer, NFs must be decomposed into atomic NFs. In UNIFY, we define a NF to be *atomic*, if;

> **Definition:** An NF is atomic if *the NF type is directly*[1] *instantiable from the RO perspective on the available infrastructure nodes*.

Another approach could be V*NF developer's view*. Here the atomicity level may depend on the NF itself. It could be split according to logical/practical components or interfaces. The atomicity level may depend on the development environment, e.g. the available software library. Choosing the right level of atomic blocks has some kind of analogy to choosing the right programming language as well as associated libraries for the task (assembly vs. C vs. C++ vs. Java vs. Python, etc.).

During decomposition, the RO will have the following priorities:

1. *Fulfil requirements* of the request (NF-FG). Maybe only a decomposed NF can be placed close enough to meet delay requirements. Maybe a decomposed NF can relax on requirements: NF needs high BW, but as decomposed only one component needs this.
2. *Optimize cost*, once both compound and decomposed NF requirements are fulfilled. Cost = NF costs + Communication costs of decomposed NFs. This depends on resource availability and placement. It may be impossible to scan all options, therefore heuristics should be applied [Sahhaf2015a].

We can classify atomic NF's according to the platform they are running on or the framework they fit in. As explained above, the abstraction level for atomic blocks will differ also depending on the context (e.g. for a VNF developer, for the Controller Adaption layer, and for the Resource Orchestrator).

3.2.1 Hardware-defined atomic blocks

Hardware defined NFs refer to dedicated resources implemented in hardware which are only able to implement a particular functionality, for example a Firewall function. The NFs that may be deployed on such devices are atomic due to the constrained infrastructure/substrate. Required deployment information (i.e., software) in order to instantiate NFs in this case is minimal, as the functionality itself is already hardwired in the infrastructure.

In fact, any infrastructure hardware substrate constrains the NFs that may be deployed on them, due to the architecture and programmability of the platform. In the context of UNIFY, we might consider two ends of the spectrum. On one end, we can consider specialized hardware-implementations for one particular NF, which offer

[1] This implies that deployment information for the considered NF is directly available to instantiate it on suitable infrastructure.

very few programmability options for changing the considered NF (mainly configuration of the function), while on the other end, we can consider the Universal Nodes (UNs) developed in WP5 and COTS servers which offer significant programmability, although still constrained by their architecture (e.g., number CPU's, 32-bit vs. 64-bit, ARM vs. x86, etc.) [D5.2, D5.4]. In between, there are a range of hardware substrates such as FPGAs which do allow wider programmability (e.g., VERILOG specs) than specialized hardware-implementations of NFs. In fact, any hardware substrate defines or rather constrains the atomicity of the NFs that may be deployed on them. However, very programmable platforms, such as the Universal Node, may benefit from finer-grained software-defined atomic blocks, which will be discussed in the next section.

3.2.2 Software-defined atomic blocks

The infrastructure platforms considered in UNIFY, in particular COTS server hardware, offers tremendous programming opportunities for VNF developers. Software defined atomic NFs for service composition are characterized by their choice of programming language or specific networking software framework. They can be considered as blocks of code providing an easier and optimized way of programming networking functionality. This allows a more modular and higher-level abstraction of network functions, which can otherwise be complicated, error-prone and difficult to develop and maintain, among other advantages listed earlier in this section. We highlight some examples of these frameworks below.

3.2.2.1 Program- and daemon based NFs

The majority of existing software implementations of network functions are programs and daemons, for which either the source code or the binary files are available. The first requires compilation (potentially after installing required libraries), while the second may be directly executed if compatible with the architecture (incl. hardware and OS-architecture) for which it has been prepared. A related way of packaging executables as well as their dependencies based on isolation technology is Docker, documented in section 3.2.3.2.

A growing range of programs becomes available, which can be combined in different ways to construct services and/or NF's. These might be independent executables, or sets of executables which interact with each using predefined interfaces. Below, we give a brief snapshot of typical NFs in the context of routing (a similar exercise could be done for other NF categories).

Routing-related NFs: Routing-related NFs provide functionality to distribute and discover topological information and calculate paths within networks. Traditional monolithic routing software is made as a one process program which provides all of the routing protocol functionalities. Routing platforms such as Quagga, XORP or BIRD [Quagga, XORP, Bird] take a different approach. They are made from a collection of several daemons that work together to build the routing table and populate the OS-level forwarding table. There may be several protocol-specific routing daemons (for OSPF, RIP, BGP…) involved, as well as executables responsible for internal communication between daemons and or OS-level functionality (e.g., 'zebra' the kernel routing manager). A feature that illustrates the control/data plane layering is a 'FIB push' interface. This allows an external component to learn the forwarding information computed by the Quagga routing suite. This architecture creates new possibilities for the routing system as the multi-process architecture brings extensibility, modularity and maintainability. It can be run with Linux and can use the standard Linux kernel for forwarding (as software router), or it could be connected to a distributed forwarding platform using OpenFlow or any other open or proprietary interface (as a high-end distributed router), e.g., [Nascimento2011]. It could also be used just for the routing protocols to interface with off-the shelf routers to receive and announce routes. In Quagga, the Routing Information Base (RIB) resides inside zebra. Routing protocols communicate their best routes to zebra, and zebra computes the best route across protocols for each prefix. This latter information makes up the Forwarding Information Base (FIB). Zebra feeds the FIB to the kernel, which allows the IP stack in the kernel to forward packets according to the routes computed by Quagga. The FIB push interface aims to provide a cross-platform mechanism to support scenarios where the router has a forwarding path that is distinct from the kernel, commonly a hardware-based fast path. In these cases, the FIB needs to be maintained reliably in the fast path as well. This routing architecture is illustrated in Figure 3-3. Similar dependencies between routing executables/daemons exist in frameworks such as [XORP, BIRD].

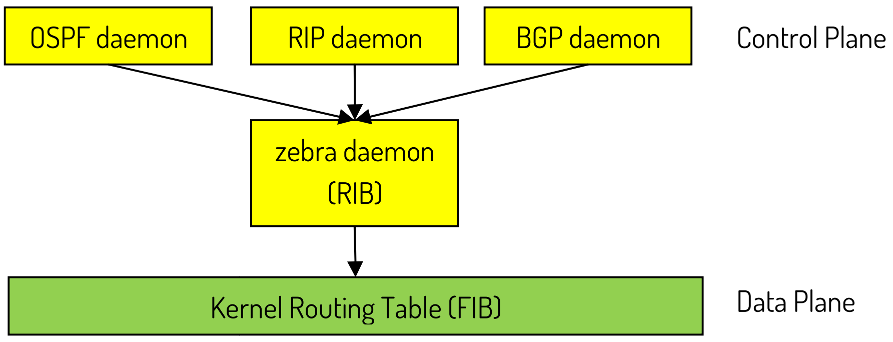


Figure 3-3: Routing atomic blocks as used in eg. Quagga

Forwarding-related NFs: Forwarding-related NFs provide optimized programs for processing and forwarding network packets (usually based on header content), typical examples are [OpenVSwitch, VALE, xDPd, LINCX]. These functions are often relying on libraries which are able to enhance packet processing handling such as [xDPd, NETMAP]. The performance impact of these is being investigated in WP5.

3.2.2.2 Click Modular Router NFs

Click is a software framework for building NF's in a modular way by combining elements, which perform simple operations on network packets, into a graph-like structure defining the packet's processing flows and thus implementing a more advanced NF. This modularity offers unique possibilities for (de)composing VNF's, which can be difficult to achieve using other frameworks for VNF programming. The very fine-grained modularity which can be achieved by implementing VNF's using these discrete elements, the easy re-use and development of new elements, are key aspects making the Click framework very suitable as a tool to deploy VNF's in a decomposed way.

In ESCAPE, the Unify prototyping framework, VNF's implemented using Click can be deployed. The ESCAPE architecture is further explained in section 7.1.

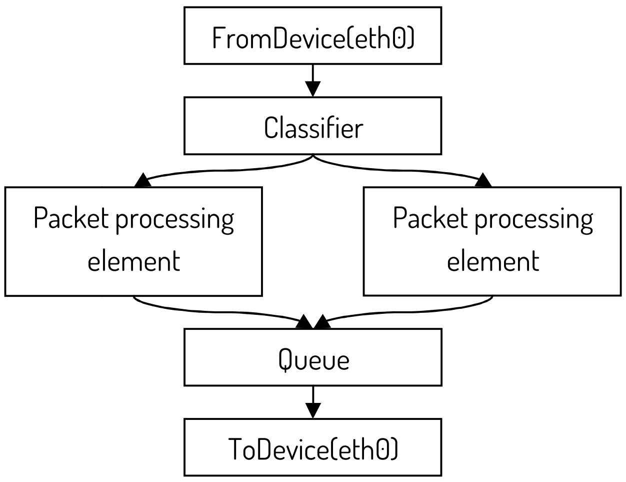


Figure 3-4: Example VNF implemented as a Click script. The VNF functionality is decomposed into atomic elements. Network packets follow the programmed flow according to the connections between the elements.

The elements are the lowest level atomic part of the Click framework. These contain basic (sub) network functionality. Individual elements implement simple packet-processing network functions like packet classification, queuing, scheduling, and interfacing with network devices. Inside a running Click configuration, each element is a C++ object and connections are pointers to elements.

A higher-level, Click-specific, language is used to connect elements into a graph-like structure, packets flow along the graph's edges. The overhead of passing a packet along a connection is a single virtual function call. Advanced network functions can be implemented through combination of multiple Click elements. To illustrate the versatility of this framework, some example network functions and basic Click elements to support them are reported in Table 3-1. Documentation and source code of existing Click elements can be obtained from this reference [Kohler2000].

Table 3-1: Click Network Function examples

Network function	**Click elements**
Firewall	FromDevice, ToDevice, Classifier, CheckIPHeader, IPFilter, Queue
Transparent HTTP Proxy	Classifier, CheckIPHeader, IPRewriter, EtherEncap
Parental control	Traffic classifier, HTTP proxy, Firewall
Forwarder	FromDevice, ToDevice, Queue, Forwarder (e.g. LookupIPRoute)
IP Router	Classifier, Queue, EtherEncap, ARPResponder, StaticIPLookup, Strip, CheckIPHeader, Paint, DropBroadcasts, IPFragmenter, DecIPTTL, IPGWOptions, ..
NAT	AddressInfo, FromDevice, Queue, ToDevice, ScheduleInfo, ToHostSniffers, EtherEncap, Classifier, ARPQuerier, ARPResponder, IPRewriter, TCPRewriter, GetIPAddress, CheckIPHeader, ...
DNS proxy	Classifier, KernelTun, IPRewriter, CheckIPHeader, Discard

The fine-grained decomposition possibilities imply also a trade-off regarding performance. Click's modularity imposes performance costs in two ways: the overhead of passing packets between elements, and the overhead of unnecessarily general element code [Kohler2000]:

- For best performance, extra patches are needed. (kernel patch for optimized driver performance ; netmap, kernel module for high speed I/O packet processing)
- Performance delta reported 10% slower for using a Click implemented router to using the Linux network stack on a standard PC.
- The cost of packet handoff by adding an element to the flow is measured to be 70ns on a Pentium III setup.
- Some tools are included in the Click distribution, which alter the configuration code. They optimize the passing from one element to another (by eliminating virtual function calls) or by code-optimizing certain elements for their use (e.g. by using compiled-in constants)

Regarding monitoring and migration, Click supports two techniques for reading/changing a configuration without losing information:

Handlers: Each element can easily install any number of handlers, which are access points for user interaction. This lightweight mechanism is most appropriate for modifications local to an element, such as changing a maximum queue length, adding/deleting routes in table, exporting statistics and other element information. An example of how handlers can be implemented to offer fast state migration can be found in [Dietz2015].

Hot swapping: Some configuration changes, such as adding new elements, are more complex than handlers can support. In these cases, the user can write a new configuration handle and install it with a hot-swapping option. Also, if the new configuration is correct, it will automatically take the old configuration's state before being placed on line; for example, any enqueued packets are moved into the new configuration.

As a next step to virtualizing network functions as Click scripts, we also mention ClickOS as a light-weight VM for Click NF's [Martins2014]. With ClickOS, the Click script is incorporated in a VM image, as explained further in 3.2.3.1.

3.2.3 Virtualization/isolation technologies

Virtualization technology enables NFs to be deployed on an isolated set of resources of an infrastructure node. This technology is of particular interest as it may provide: snapshotting functionality, easy and predictable reproducibility of installations, pre-configured dependencies between lower-layer software. We distinguish between two types of technologies: VM-based and container-based technologies.

Figure 3-5 illustrates the typical architecture of a VM or Container environment, we will elaborate some more on both types in below sections.

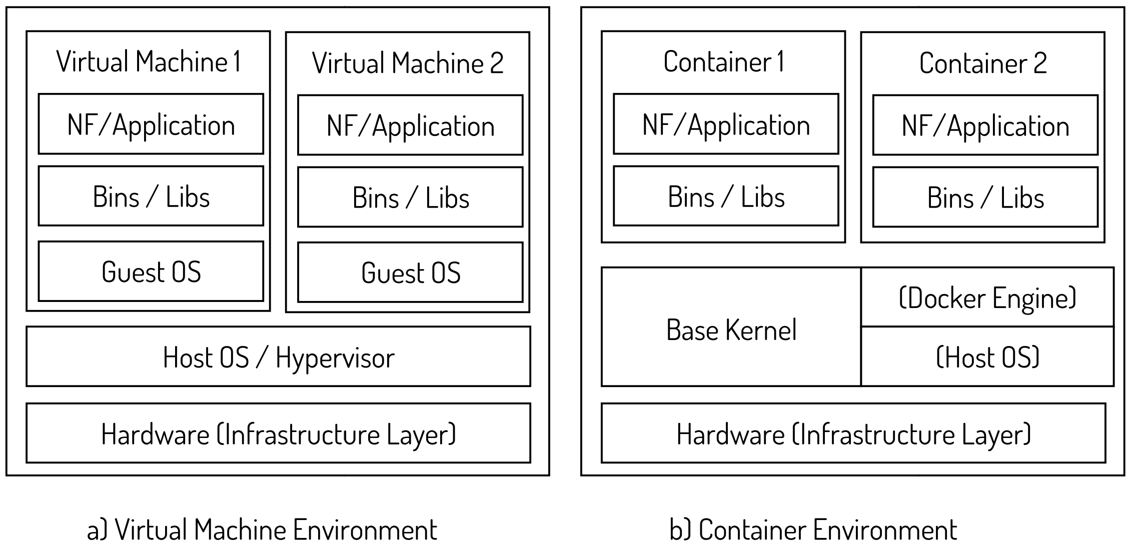


Figure 3-5: VM a) versus Container b) architecture

3.2.3.1 Virtual Machine based atomic blocks

Virtual Machine (VM) for hosting an NF can be seen as images to be mounted in a network node. The atomicity is at the level of an image file which contains the programs to execute the NF. The image typically combines several software blocks as described above in combination with an Operating System. The hardware platform of the network node is a server environment, where the VM manager (a hypervisor-like control layer) can instantiate and manage several images (e.g. Xen, KVM, VMWare, Virtual Box). Each VM image has its own limited pool of resources such as CPU cores or memory. Also traffic steering elements such as switches need to be implemented to enable communication between several VMs (e.g. OpenvSwitch). The server can be built with COTS hardware, possibly specialized HW blocks can be added. Performance and specifications of such an environment is the focus of work performed in WP5 (Universal Node). The VM is considered a very practical tool to deploy NF's as it offers high flexibility and modularity. By instantiating more or less VM's, scalability is easily achieved. By running NF's in VM's, they can also be more isolated and offer better possibilities for resiliency. Live migration of VM's is possible but needs to be supported by the VM manager.

The RO can easily deploy a VM of a NF on any appropriate infrastructure node. It can offer a good level of abstraction, as it can be described with high-level parameters (such as high-level functionality, needed resources) without any further details on the inner workings.

Below is a short description of some research examples, to illustrate how VM's can be used in this context.

ClickOS consists of a tiny, Xen-based virtual machine aimed at network processing. ClickOS VMs are made up of the Click modular software (see previous section) running on top of MiniOS, a minimalistic OS provided with the Xen sources [Martins2014]. It can be seen as a dedicated VM which is running only a Click config file. The lightweight nature of ClickOS images makes it comparable to a container (as explained in Section 3.2.3.2). It has the advantage of being small in size (around 5MB) and fast instantiable (order of 30ms). A more in-depth analysis of how ClickOS can be used in an NFV scenario is found in [Martins2014].

An implementation of a BRAS in ClickOS is discussed in [Dietz2015]. The main advantage is that easy and fast state transfer can be accomplished by implementing a new specialized Click element. By using the Click framework, it is inherently easy to visualize and analyse the different lower-level atomic blocks (Click elements) which make up this BRAS VNF.

Several case studies emerge that propose a VNF architecture based on VM's (implemented with proprietary VM images). As example we can mention:

NFV has been applied to a specific network function, the Session Border Controller [Monteleone2015]. Deployed at the border between different network domains, an SBC must concurrently process a high number of media flows transmitted by the users, with strict real-time constraints. For the SBC, the following atomic blocks are used (each unit is supposed to run on a dedicated VM):

1. BCF: operating on the control plane, implements the Border Control Function (BCF). It performs call and session control, establishing communication sessions between different users and/or network applications.

2. BGF: operating on data plane, implements the Border Gateway Function (BGF) performing control of media streams. It provides media processing functions under the control of the BCF, allowing establishing media streams.
3. LBF: operating on the control plane, implements the Front End and Load Balancing Function (LBF). The LBF unit has the role of distributing signaling messages towards different instances of the BCF unit (and, for optimization, also between different processes internal to a single BCF unit). Load balancing is not required for the BGF unit, because the choice of a specific instance is performed by the BCF, according to specific resource management algorithms.
4. OAM: dedicated to providing the SBC Operation, Administration & Management (OA&M) functions

The use of VM's can also be applied to a Gateway architecture, which can decompose into one VM dedicated to CP and another VM dedicated to DP. This can be beneficial for resource management, as described in [Hahn2015] and also further on in section 3.3.3. The usefulness of VM's in a Virtual Wireless Sensor Network use case is shown in [Mouradian2015], where VMs are used for their elastic and scalable deployment and migration characteristics. This makes it easy to deploy specific gateway images in function of the targeted type of sensors or communication protocol.

3.2.3.2 Container based atomic blocks

Container based atomic blocks are processes confined to a limited set resources on the platform they run on. Container virtualization is also known as Operating-System-level virtualization. There are however important differences between a container and the above described VM environment. Most processes running on a server can easily share a machine with others, if they could be isolated and secured. Further, in most situations, different operating systems are not required on the same server, merely multiple instances of a single operating system. OS-level virtualization systems have been designed to provide the required isolation and security to run multiple applications or copies of the same OS (but different distributions of the OS) on the same server. So containers tend to have less overhead since no complete OS has to be included in the NF image, but have therefore also less flexibility since all containers running on a host are all need to be on top of the same kernel. Multiple container technologies exist (e.g. Docker, OpenVZ, Virtuozzo, Linux-VServer, Solaris Zones, FreeBSD Jails) and further analysis and benchmarking is necessary to see how these compare to each other and the VM based deployment of NF's.

An interesting feature of certain container technologies is the ability to do checkpointing. This is a kernel feature that makes it possible to freeze a running container (i.e. pause all its processes) and dump its complete in-kernel state into a file on disk. Such a dump file contains everything about processes inside a container: their memory, opened files, network connections, states etc. Then, a running container can be restored from the dump file and continue to run normally. The concept is somewhat similar to suspend-to-disk, only for a single container and much faster. A container can be restored from a dump file on a different physical server, opening the door for live migration. Live migration is an ability to move a running container from one physical server to another without a shutdown or service interruption. Network connections are migrated as well, so from a user's point of view it looks like some delay in response. Systems that support this are [OpenVZ, Virtuozzo], while other solutions are trying to provide similar functionality even for processes that are not containerized [CRIU].

There is another difference to VM's: Updates in the kernel (required for e.g. driver or security updates), require an update of the complete VM images. Things are easier with containers: since the kernel is outside of the scope of the container image, you don't have to change all your container images when you upgrade the kernel. Also, in a container environment, the single point of failure points to the kernel, while in VM environments this is the VM manager (hypervisor), see Figure 3-5.

A traditional "atomic block" for computing is the *process*, which typically runs in an execution environment where it gets a certain share of CPU time and access to an isolated chunk of memory. Usually whole programs are divided into separate processes when different tasks can be isolated and act somewhat independent, with low requirements on shared state between them. When tasks can be identified but have high demands on state sharing usually threads are used instead, which share access to the same chunk of memory. Processes typically communicate with each other by exchanging messages using one of many IPC mechanisms provided by the operating system or libraries such as signals, pipes, message queues, and shared files when they are executing on the same host. When processes are not executing on the same host the typical method of IPC is TCP/UDP sockets, Remote Procedure Calls, or through middleware providing distributed shared memory or file systems.

Based on the atomic block requirement that an atomic block must be able to run on separate hosts, a large amount of software implemented as one or more processes fits. Some processes do not, for example ones that require IPC mechanism such as memory mapped files or directly shared memory in order to communicate large amounts of data with low latencies[2]. Docker is a system that allows us to package processes into containers and deploy them on substrate nodes, these containers be good candidates to be atomic blocks according to the above requirement [Docker].

A basic structure in Docker is the *image,* a file system containing data, runnable processes, and initial configuration. One image can be instantiated in an isolated environment called a *container,* which runs one or more processes. Processes inside the container are restricted to their own namespace of for example process IDs, file system, network interfaces, mounted file systems, and SysV IPC mechanisms, allowing multiple processes, which would normally interfere with each other, run independently [Menage2015]. The container acts similarly to a virtual machine but with significantly lower overhead.

Docker allows relationships between containers to be easily configured and modified if necessary when instantiated. Container relationships can be defined using *links, exposed ports, and volumes.* Links and ports refer to network

[2] Although many of these may be able to function using a distributed / networked file system or memory.

connectivity and by linking container B to container A, container B will have environment variables and hostnames automatically created containing the IP address of container A and it's exposed ports. Volumes can be used to share files and directories between containers, for example container A may generate data files in directory /output that should be processed by container B. When starting B we can specify that all data volumes from container A should be mapped to B, processes running in container B will then be able to access files mounted in /output on B. An example of this is illustrated in Figure 3-6 where one process running in a container is acting as a Producer and creating files into a volume. Container B has both types of relations to A, more specifically it is linked to A (see *Link* relationship) and additionally, it is importing A's volume(s) (see *Volume_from* relationship). B then processes the files generated by A and communicates with A using the linked information such as the IP address, port number and password.

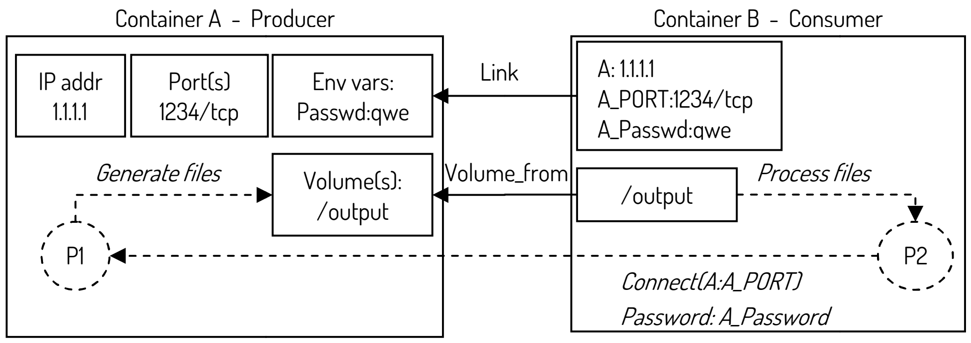


Figure 3-6: Two Docker containers A and B. B is related to A with both links and volumes.

Docker-compose [DockerComp] allow us to compose a *service* from multiple containers, define their relationships and scale in or out individual roles. To create the service *File* consisting of the producer and consumer(s) illustrated in Figure 3-6 we first create two images, file-generator and file-processor. These images contain any libraries that the processes need, configuration files, and the processes themselves. We then use the docker-compose configuration to define the service as:

```
producer:
  image: file-generator
  volumes:
    -/output
  environment:
    - passwd=qwe
  expose:
    - "1234"
```

```
consumer:
  image: file-processor
  links:
    - producer
  volumes_from:
    - producer
```

To start our File service composed of the two container types we run "`docker-compose up`" which would start one producer and one consumer, and automatically link them and mount the data volumes. If File service is not running fast enough, perhaps the consumer container is not fast enough to process generated files, we can add more consumers using the scale command "`docker-compose scale consumer=3`", which would create two additional consumer containers and link them with the producer. In this example the two containers cannot be seen as two atomic blocks as they have to be instantiated on the same node due to the shared file system volume, the atomic block in this case is the File service itself. If we remove the volume relationship leaving only the linking, the individual containers could be seen as atomic blocks. While the current Docker version (version 1.6) alone is not able to automatically link containers instantiated on different hosts there are many projects aimed at solving this issue [SocketPlane, Flocker, Swarm, Kubernetes], which likely will be integrated into Docker soon. With this issue solved we could analyze a docker-compose service definition and automatically detect what the atomic blocks are, i.e. those container types without volume dependencies. Another option is to introduce new configuration commands into the docker-compose configuration to manually define atomic blocks / deployment groups that have to be co-located on the same node, this approach is taken in Kubernetes which calls this concept *pods.*

A practical example of decomposing a WebCache VNF into atomic block as processes is presented in Figure 3-7, where load-balancing is implemented by an haproxy process, sending connections to squid process(es). The squid process in turn is assisted by a RADIUS process for user authentication and MongoDB process for storing cached objects. When the WebCache service is deployed all these processes may run on other servers and could be scaled independently from each other. This enables a single service definition to handle various load scenarios, e.g. if the total throughput is putting a load on the Squid process, it can be scaled out. If however it is the amount of objects that is causing performance issues, the MongoDB process can be scaled.

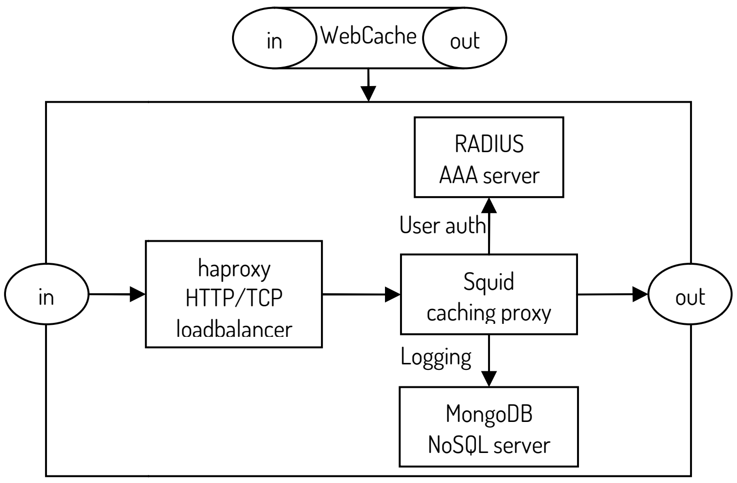


Figure 3-7: WebCache decomposed into 4 processes

3.2.4 Atomic Block overview

The above described atomic block categories each have their own advantages and limitations. We try to give a high-level comparison and overview in below table.

Table 3-2: Atomic Block overview

HW based	**SW based**	**VM based**	**Container based**	**Monolithic HW+SW**
Atomic HW block	Atomic SW code block	Atomic VM image file	Atomic container image file	Atomic physical device with dedicated firmware
Significant processing power gain Need optimized code for maximum performance Only useful for specific decomposed NF tasks (eg. encryption, transcoding) RO or Controller needs to support the HW specific functions	Higher abstraction offers better view on possible decompositions (eg. Click) Flexible for reuse by developers Can be a piece of code, library, executable, daemon, OS, or kernel Very fine granularity of NF's can be achieved Can be deployed in a container or VM context	Image contains NF code + OS Each image can have different OS Image file size might contain significant overhead Can run on COTS HW (possibly combined with HW optimized blocks) Checkpointing and (live) migration can be supported Easy scalability Isolation down to the hypervisor level	Image contains only the NF code Kernel is shared between all containers Image file size is compact More resource efficient Can run on COTS HW (possibly combined with HW optimized blocks) Checkpointing and (live) migration can be supported Easy scalability, manageability Isolation down to the kernel level	Optimized performance by combination of closed SW + proprietary HW Not open, expensive, vendor specific Not flexible for updating, scaling, migrating Can be preferred solution in case of very strict requirements regarding SLA, security, reliability, and performance
Example: FPGA, ASIC, Network Processors	*Example*: Click, Quagga, NOX controller code	*Example*: ClickOS image for Xen	*Example*: Container repository for Docker	*Example*: Vendor specific hardware

3.3 Monolithic VNF's and Services constructed from atomic blocks

The ability to remotely deploy and run decomposed VNF's in a VNF infrastructure opens up a new domain of possibilities. Trade-offs are created between latency, reliability, centralizing services, scalability, communication overhead, etc. A service provider might use its own VNF infrastructure domain or someone else's; this will also have security, policy enforcement implications.

SDN principles and above described atomic blocks bring new possibilities to legacy (monolithic) Network Functions. They can bring the appropriate granularity (fine- or coarse-grained) depending on the target scenario. It offers flexibility to dynamically identify the needed functions for the decomposition (e.g. atomic blocks and a set of flow configurations).

A high-level description of different VNF use cases is available in ETSI documentation [ETSI NFV Use case]. The Unify project is developing a framework in which these use cases are practically deployable. Focusing solely on the decomposition aspect (the orchestration part will be handled in a next chapter), below subsections highlight some examples to illustrate how existing services can be practically decomposed into the aforementioned atomic blocks.

3.3.1 Elastic Router use case

One category out of different VNF use cases is to offer a Virtual Network Function as a Service (VNFaaS): Functionality can be moved from purpose-built hardware to Universal Nodes. This enables e.g. a service provider to offer these services as scalable and decomposed VNF deployments, as is illustrated in this subsection with a basic network function such as an elastic router.

The objective of the Elastic Network Function use case is to demonstrate the different dynamicity and, more specifically, scalability approaches supported by the UNIFY architecture (use case proposed in D2.1 sec 3.4.1 [D2.1]). Later on, the elastic router use case in D2.1 was amended with an elastic stateful firewall use case to illustrate more the state migration possibilities in the UNIFY framework. The decomposition of the elastic firewall is discussed in [D2.1 a], however, to better illustrate the decomposition options into different atomic blocks we build further upon the elastic router. From a top-bottom analysis, we can decompose the elastic router like this:

Router functionality is either a monolithic HW+SW atomic block or decomposed into a SW part in combination with a HW forwarding part. The SW part will calculate the routes and store them in the RIB (Routing Information Base). The HW forwarding plane is part of the general Data Plane (DP), which holds the Forwarding Information Base (FIB) and the forwarding HW itself. CP and DP can now be decomposed further.

- *Control Plane (CP)*: As mentioned under SW defined atomic blocks in section 3.2.2, several SW solutions exist who implement legacy routing protocols (Quagga, XORP, BIRD). Additionally, routing CP functions can be implemented in an Openflow controller, or be supported by OS network stack.

- *Data Plane (DP):* OS Network stack, Click, or (Virtual) OpenFlow switch DP decomposition is relatively easy. A decomposed part of the NF can be split of and put in series, it can duplicated and run in parallel, whether or not combined with load balancing. This principle was explained in Deliverable 2.1 (section 3.4.1.1).
- *Semi-transparent scalable service chain:* Distribute incoming packets via load balancing over parallel CP and/or DP chunks.
- *Non-transparent scalable service chain:* NF's in the forward path can be put in series. Eg. Click implemented buffers/queues, HW acceleration by using NICs with faster line-rate.

The general decomposition of a router is visualized in Figure 3-8.

State migration can be implemented by exporting the RIB, FIB tables in a reliable way. Mechanisms such as OpenNF can do this and are further described in Section 5.2.1. Achieving fine-grained decomposition or scalability in the DP seems easier achievable, as CP decomposition seems to stop at the coarse-grained level of the different routing protocols. One could state that legacy routing protocols were not designed for scalability or decomposition. More research in this area can reveal if scalability or decomposition is a limiting factor and come up with new ways to do it. As stated in [ETSI NFV Use case] traditional IP routers based on custom hardware are among the most capital-intensive portions of service provider infrastructure. Provider Edge (PE) routers run out of control plane resources before they run out of data plane resources and virtualisation of control plane functions improves scalability. Substantial cost saving may be possible by moving routing functionality from purpose-built routers to equivalent VNF in COTS hardware environments. Such an SDN/NFV enabled network architecture, as developed in the UNIFY framework, offers a lot more possibilities to tackle this problem. E.g. a possible way of implementing an elastic control plane is handled in [Wang2014] by exploiting the high control plane throughput capacity and scalability of vSwitches.

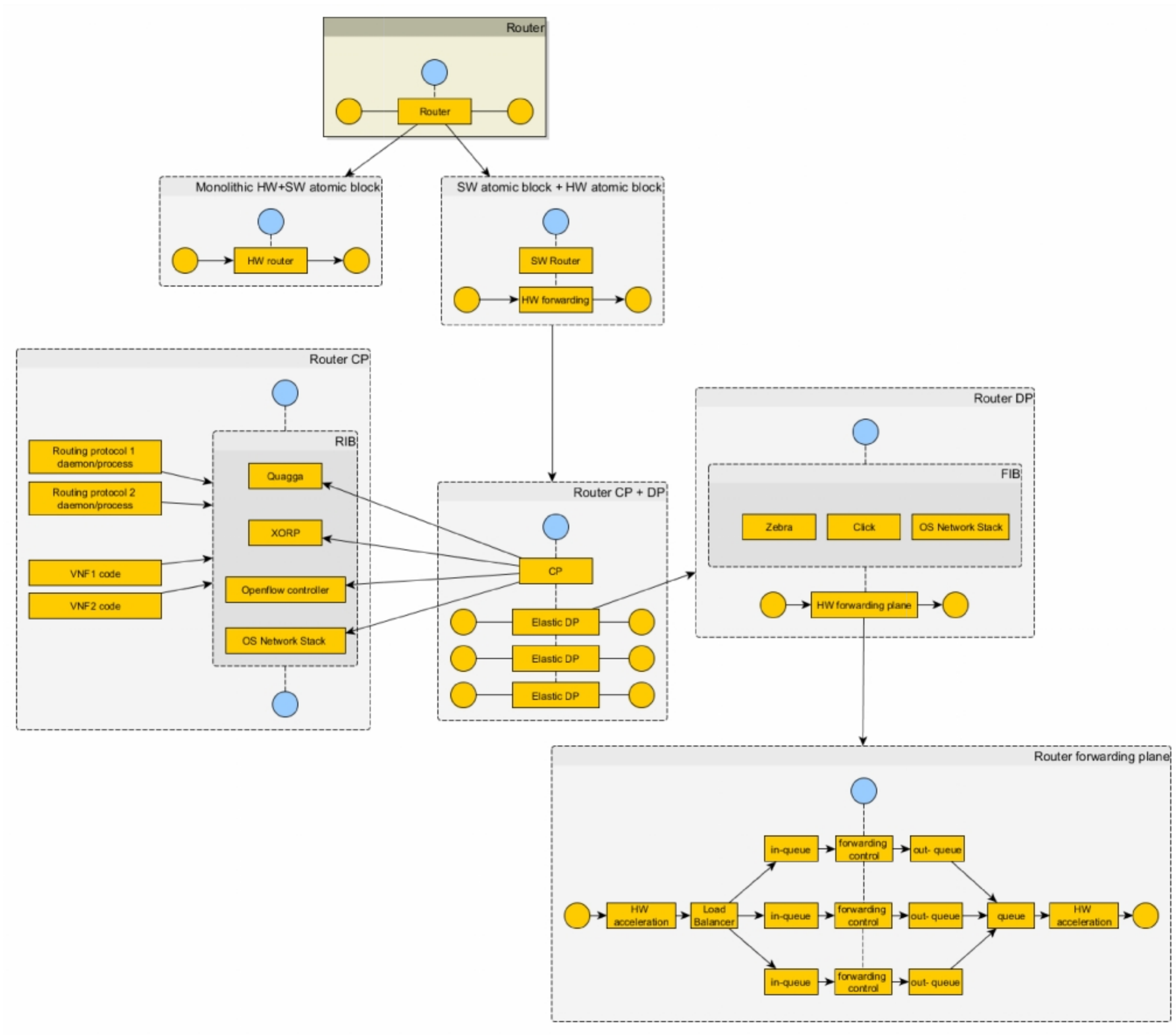


Figure 3-8: Elastic Router Decomposition. Starting at the top, it is decomposed further into finer-grained atomic blocks, grouped by the dashed boxes. (CP, DP, RIB, FIB, forwarding elements). Dark grey dashed boxes depict possible technology choices (which could also run in combined way).

3.3.2 Flow-based Network Access Control

In addition to the elastic router function described in previous subsection, we can also use VNF decomposition in more advanced network functions such as access control services. This is illustrated in FlowNAC [FlowNAC], a Flow-based Network Access Control solution that allows granting users the rights to access the network depending on the target service requested. Each service, defined univocally as a set of flows, can be independently requested and multiple services can be authorized simultaneously. Building this proposal over SDN principles has several benefits:

SDN adds the appropriate granularity (fine- or coarse-grained) depending on the target scenario and flexibility to dynamically identify the services at data plane as a set of flows to enforce the adequate policy.

Instead of a monolithic authenticator or Policy Enforcement Point (PEP), three different elements are separated performing the same functionality:

1. The *SDN datapath* is a flow-based forwarding element (i.e. OpenFlow datapath). The flow entries (matching fields and actions) that rule its behaviour are stateless, since they do not depend on previous matched frames. It basically distinguishes between two types of traffic. On the one hand, the authentication traffic is allowed and forwarded to the authenticator network function to be processed. This type of frames can be easily identified in the datapath by its Ethertype (i.e. 0x888E). On the other hand, the rest of the traffic or the service related traffic is enabled or dropped depending on the result of the authentication process. Both types of traffic (i.e. authentication frames and service traffic) can be controlled for each user (i.e. identified by its MAC address) by any standard SDN datapath. It basically enforces access control at port (flow)-level (i.e. the physical port of the network node)
2. The *Authenticator Network Function* (ANF) basically implements the access control process and operation. It receives and parses the authentication frames exchanged with the users (forwarded by the SDN datapath) and encapsulates them in the appropriate protocol (e.g. RADIUS) to communicate with the authentication server. The ANF also maintains the state associated with each AA (Authentication and Authorization) process coming from the same or different users. The entire authentication related traffic (i.e. the AA control traffic) remains in the data plane and is not processed by the SDN controller. This means that this traffic is not encapsulated by OpenFlow in Packet_In messages, avoiding the overhead and the consolidation of the AA processing in the controller.
3. The *SDN controller* is responsible of adding and removing the flow entries at the SDN datapath. Both the authentication frames and the service traffic depend on this component. Therefore, it rules the behaviour of the SDN datapath by enforcing the policy decision on the access device. In this case, the southbound protocol used to communicate is OpenFlow. On the other hand, the northbound interface exposes the programmability of the SDN datapath to forward the authentication traffic to the ANF and activate the authorized services. This means that the authorization of services (i.e. adding the authorized flow entries) is achieved by means of this northbound interface.

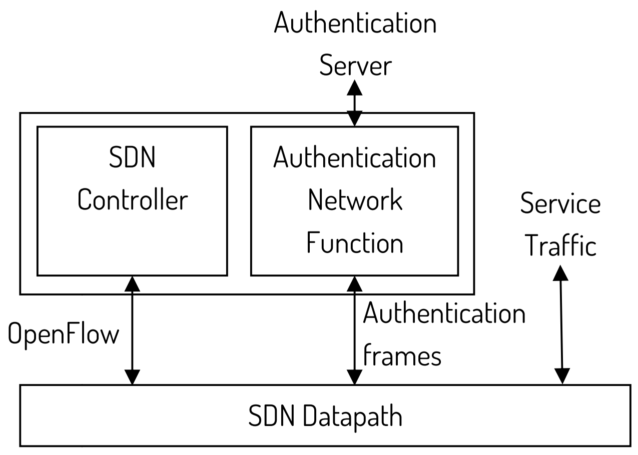


Figure 3-9: FlowNAC architecture

3.3.3 Gateway decomposition

This example fits in a typical use case described in [ETSI NFV Use case]: Virtualisation of the Mobile Core Network and/or virtualisation of the Home Environment: The gateway decomposition shows how gateway functions typically implemented in mobile network EPC architectures or Home-environment Set-top-boxes can be virtualized and decomposed in a more efficient way.

Gateways are present in many network topologies on different levels e.g. mobile networks, service provider access networks, and aggregation network. Service provider networks typically include a Provider Edge (PE) gateway at the edge of the core, facing the Customer Premises Equipment (CPE) gateway at the end-host. Both gateway types could be decomposed and virtualized which offers advantages in the context of: centralized management and data, modest tool footprint at the end-user, and efficient use of resources. In the rest of this section we give more details on use cases from the mobile network world.

Evolved Packet Core (EPC) is a flat architecture that provides a converged voice and data networking framework to connect users on a Long-Term Evolution (LTE) network. The key components of the EPC are:

- Mobility Management Entity (MME) - manages session states and authenticates and tracks a user across the network.
- Serving Gateway (SGW) - routes data packets through the access network, also specific signalling data between network nodes (e.g. during handover, authentication data).
- Packet Data Node Gateway (PGW) - acts as the interface between the LTE network and other packet data networks; manages quality of service (QoS) and provides deep packet inspection (DPI).
- Policy and Charging Rules Function (PCRF) - supports service data flow detection, policy enforcement and flow-based charging.

The modular architecture of the EPC makes it inherently fit for decomposition using some of the above described atomic blocks. Especially the SGW and PGW can benefit from an even further decomposition. In the EPC context, there is a specific classification of the traffic (That is different from the CP/DP layering in an SDN-context):

- User plane traffic (UP): the application creates or processes data packets that are processed by protocols such as TCP, UDP and IP.
- Control plane traffic (CP): the radio resource control protocol writes the signalling messages that are exchanged between the base station and the mobile.

Both types of traffic are passing through the S/PGW, but have a different nature and require a different routing strategy. In [Hahn2015] two main approaches are investigated: combined GW implementation within a single virtual machine handling both user and control plane traffic and, alternatively a GW implementation that relies on a decomposed processing of user and control plane data in dedicated virtual machines. Both virtualized GW models can benefit from dynamic adaptation of GW resources based on current traffic demands (eg. night/day, weekday/weekend traffic load). Different VM's are created, each with the specific characteristic of assigned cores.

- *Inelastic model:* Fixed physical gateway (HW+SW atomic block) with fixed CP/UP core allocation.
- *Elastic model 1:* The number of gateway VMs can be scaled in/out according to the current network load.
- *Elastic model 2:* previous model enhanced with an added specific day and night image of network elements. The images are characterized by different CP/UP cores allocation schemas. The number of network elements can be scaled in/out according to the traffic requirements.
- *Decomposed model:* assuming a CP and UP split and the possibility of independent scaling of CP and UP cores of a gateway according to the traffic characteristics.

In this use case, the resource saving achieved with the Decomposed model are only a few percent more, compared to Elastic Model 2. Most of the gain is already achieved when changing from an Inelastic model to an Elastic Model 2. The decomposition architecture needs to consider firstly that traffic profiles usually do not change so dramatically over the day as but instead in a longer time frame and secondly the simplicity of the Elastic model 2, its good performance and compatibility with the present GW design. Hence the scalability gains of a decomposed GW implementation might not be a sufficient reason to substitute combined GW solutions.

This is a good example of possible trade-offs we need to consider during decomposition. In addition to this we can highlight an investigation done in [Basta2014]. Similar to above, an operator has a network of multiple clusters of 1 PGW with several SGW's. Or more general, several CPE's connected to a PE node, or several access network nodes connected to an aggregation network node. With the help of NFV/SDN decompostion, some optimizations can be done:

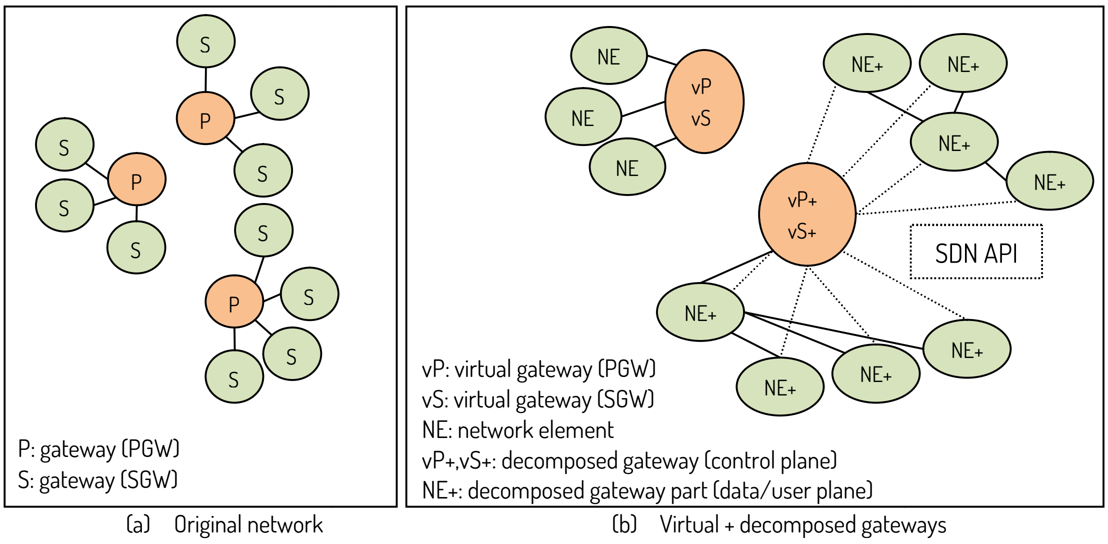

S
P
S
S
S
P
S
S
S
P
S
S
S
P: gateway (PGW)
S: gateway (SGW)
(a) Original network
NE
NE
NE
vP
vS
NE+
NE+
NE+
NE+
vP+
vS+
SDN API
NE+
NE+
NE+
NE+
vP: virtual gateway (PGW)
vS: virtual gateway (SGW)
NE: network element
vP+,vS+: decomposed gateway (control plane)
NE+: decomposed gateway part (data/user plane)
(b) Virtual + decomposed gateways

Figure 3-10: Gateway decomposition example (fully meshed network, only main connections are drawn)

- Use virtualized GW's: To reduce costs, all GW's could be centralized and virtualized in just a limited number of datacenters. The datacenter locations are strategically chosen to minimize latency. All traffic is routed to those central virtualized GW's in the datacenters. If all latencies are known in the network, an optimal topology can be calculated where the virtualized gateways must be located to ensure a certain delay budget. This is shown in Figure 3-10 where eg. all gateways in (a) can be virtualized in 2 central nodes (b). The orignial GW's in (a) are replaced by a virtual instance (vP,vS) + a simple switching network element (NE).

- Next, we can decompose the virtual GW into a control plane and a SDN data plane (As in a classic SDN context). The measurable advantage: not all traffic needs to be routed to the data centres, the SDN data plane can offer faster traffic delivery. So to meet the delay budget, less data centres are needed. This is illustrated in above figure, where multiple GW control planes (vP+,vS+) are centralized in 1 datacentre and the data plane is implemented via SDN enabled switching elements (NE+).
The disadvantage we introduce is that more SDN control data is generated which imposes more traffic overhead in total. Which induces again more delay, and compromises the delay budget. The volume of SDN control data is a specific to the network (a range of 10-50% of the non-SDN network load is proposed). So decomposing the virtual GW's further is only possible up to the point where the delay budget contraint is met.

This shows that decomposition will have its limits in function of the available data centres, delay-budgets and SDN control volume.

3.3.4 Service Decomposition trade-offs

Each decomposition should specify quantitative parameters for the derived VNF's including the required delay upper bound, function call frequency, exchanged data volume or deployment cost. This was illustrated in the examples in above section 3.3.3 where CPU core usage, end-to-end delay or traffic overhead is considered as a KPI for the decomposed service. As discussed also in [Basta2013], Figure 3-11 illustrates the expected behaviour of some chosen key criteria where decomposition results in cost savings (black line). It is generally accepted that decomposing into VNF's will mean a cost saving specific to each use case up to a certain decomposition degree. If we decompose too much, we realize that cost will rise again e.g. because of the many (un-optimized) interconnections between the blocks, rising development effort, maintenance, and overhead. We also argue that decomposing or virtualizing may impose some additional data and delay overhead (red and blue lines). Note that the order of the functions might be different for each cost line. Given the right order, the cost function will be a decreasing function, however not necessarily steady, but with steps and possible flat parts. Our future prospective target is to find the optimal deployment solution via a quantitative evaluation of the function deployment and decomposition in the context of specific use-cases.

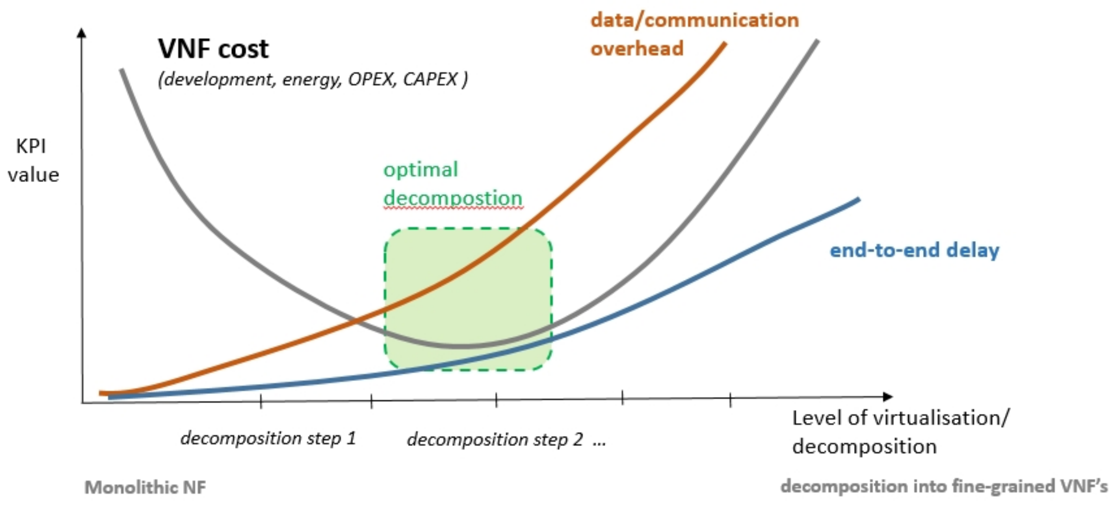


Figure 3-11: Decomposition trade-offs

4 Scalable orchestration algorithms

The programmability framework described in [D3.1] and in section 2 above describes the data models involved in orchestrating service graphs (SG) or network function forwarding graphs (NF-FG) respectively. Within the orchestration layer appropriate mappings of network functions (NFs) to physical devices as well as a mapping of the NFs' interconnections to routes have to be determined. Within this section we focus on the algorithmic challenges involved and how UNIFY tackles and plans to tackle these. We start with some context on the scale of the resource orchestration problem in Section 4.1 and provide an introduction to the Virtual Network Embedding (VNE) problem in Section 4.2. Within Sections 4.3 to 4.5 we focus on the so called pure embedding problem that frames the NF-FG embedding in rather general terms. While in Section 4.3 the online problem is considered, Sections 4.4 and Section 4.5 deal with how offline solutions can be incorporated. In Section 4.6 we highlight our work on tackling a novel problem arising in the service chain context, namely the joint decomposition of composed network functions and their embedding. Lastly, we outline in Section 4.7 how the technical challenge of distributed embedding processes can be tackled.

4.1 Context of the resource orchestration problem

The orchestration brings together two information flows: a top-down flow initiated by the users of the UNIFY platform initiating service requests, and a bottom-up information flow driven by available resources in the cloud and carrier network. As a consequence, the difficulty of the orchestration challenge is heavily related to the scale of these involved processes and their underlying (re-)sources.

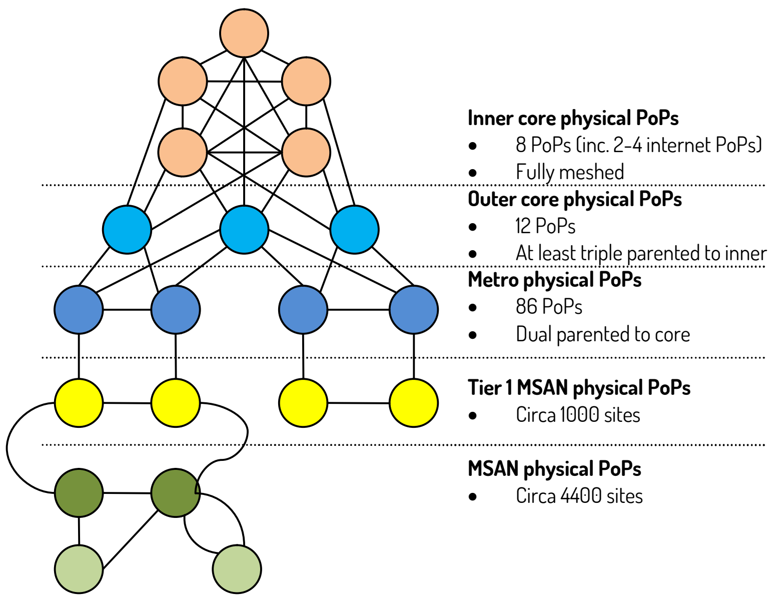
This section documents assumptions on these processes based on publicly available documentation, as well as input by the operator partners in the consortium. The considered network architecture and associated parameters is based on the UK BT network and information available on datacenters in UK.

A typical Telecom operator network has a hierarchical structure with a dense core router meshed network consisting of inner and outer core Points of Presence (PoPs). The end customers are interconnected to this core network via a hierarchy of tree-structured access- and metro aggregation networks. As an example, Figure 4-1 illustrates the network structure of BT operator network in UK.

Based on the information available in [BT21CN, DCMap, ISPreview, SPARCD2.1], an (maximum[3]) estimation of the number of devices in a service provider network (BT) is made which is reported in Table 4-1.

According to this estimation, a typical network operator infrastructure consists of almost 50K devices at different parts in the network and data centres excluding CPEs. In case of including CPEs, almost 10M devices are needed to be orchestrated.

[3] When room for interpretation (or detailed information is lacking), estimations were rounded upward, in order have quantitative orchestration targets which are sufficiently future proof.



Inner core physical PoPs
• 8 PoPs (inc. 2-4 internet PoPs)
• Fully meshed
Outer core physical PoPs
• 12 PoPs
• At least triple parented to inner
Metro physical PoPs
• 86 PoPs
• Dual parented to core
Tier 1 MSAN physical PoPs
• Circa 1000 sites
MSAN physical PoPs
• Circa 4400 sites

Figure 4-1: BT network structure

Based on a set of studies in UK [ISPreview, UBM, Tonse, BIS], the number of new customers per day is about 2.5K and the number of service requests that may be expected to a network operator such as BT is estimated around 5-10K per day. From an NF-FG orchestration point of view, we expect that the number of "IP Edge" PoPs / "Service PoPs" is the most indicative number[4], as the access node only plays a role in the subscriber physical access connection to the "Service PoP". But when the dealing with the provisioning of services, or dynamically asking for new services, or changing profile, etc. it is the IP Edge/service edge that the actual configuration has to be performed. The targeted timescale of actions in the orchestrator should be as low as possible with significant difference from traditional/manual provisioning. The resulting timescale is thus expected to be in the range of 100 ms to 1 s.

[4] When a new customer needs to be initially provisioned a broadband line, some operation involves the access network segment (including the MSAN) too.

Table 4-1: Service provider network size

	Total number
Telecom network	
Access network	
End-user CPEs (for market share of 30%)	10,000,000
MSANs (DSLAMs)	30,000+
IP Edge/Service PoPs	1,000
Core network	
Core PoPs	10
Datacenter network	
BT data centres (assuming 70% of 45 global data centres in UK)	32
BT data centre servers (assuming 70% of 25000 global BT data centre servers in UK)	17,500
Top of the rack switches (1 switch per 40 servers)	438
Data centre routers (2 per site)	64
Total # of devices to be orchestrated	49,011
Total # of devices incl. CPE for market share of 30%	10,049,011

4.2 Background on the Service Chain Embedding Problem

In the early 2000's the so called "Testbed Problem" arose when researchers were trying to embed overlay topologies into a given testbed. Back then, the task was to place the overlay nodes in such a fashion that the testbed nodes as well as the testbed links were not over-provisioned [Ricci2003]. In the light of the virtualisation trend, the Virtual Network Embedding Problem (VNEP) arose, to attend to the general problem of mapping or embedding a (virtual) graph onto another (substrate) graph.

Figure 4-2 outlines the general idea: Given multiple Virtual Networks (VNets) and a common physical infrastructure, an embedding which maps virtual nodes onto substrate nodes and virtual links onto paths in the substrate is searched for. In the literature many different versions of the VNEP are considered (see [Belbekkouche2012; Fischer2013] for surveys). The VNEP's very general problem formulation allows for modelling arbitrary resources, as e.g. CPU, RAM and hard drive space and arbitrary type information (see e.g. [Schaffrath2012] for an hierarchical description language of resources). Especially the ability to attribute network nodes with specific type information is important in the area of service chaining, as embedding e.g. a Firewall on a universal node will impose different resource requirements than embedding a Firewall on a hardware-based middlebox. For further information on the different application of the VNEP and the models we also refer the reader to the Appendix A.2.6 of D.3.1 providing a brief taxonomy.

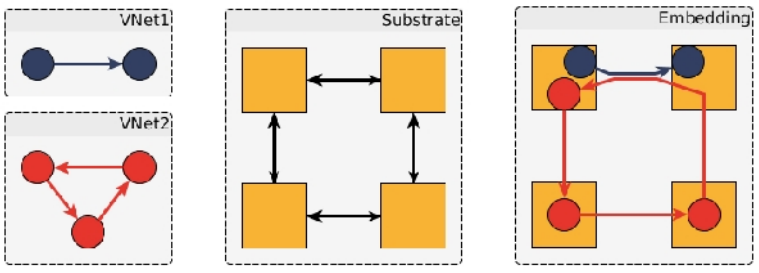


Figure 4-2: Network embedding concept

The service chain embedding problem that pertains to UNIFY is closely related to the classic VNEP, as the VNEP generally allows for location and type requirements on network nodes and generally considers strict QoS guarantees in terms of the used resources and e.g. the end-to-end latency. However, the service chain embedding problem differs from the VNEP problem by representing network **functions** rather than specific node locations with no semantic information about the function. In Figure 4-3 the firewall functionality may e.g. be implemented using a load-balanced series of firewalls. For embedding service chains, functional decompositions therefore must be taken into account (see Section 4.6). This is of particular importance as NFV capable COTS hardware may be easily extended by additional specialized hardware such as network processors, ASICs, FPGAs to improve performance. Therefore some nodes may e.g. do load-balancing on the fly while others may require more computing power. Optimizing the hardware choice during decomposition can thus greatly improve the total performance and reduce the resources necessary to establish services. The special capabilities of HW optimized atomic blocks must be advertised appropriately and understood by the controllers [D3.1, D2.2] and taken into account while orchestrating.

However, once a concrete decomposition is found, the **orchestration problem** reduces to the VNEP with decomposition specific constraints pertaining e.g. to the type of implementation variants and concrete SAP endpoints.

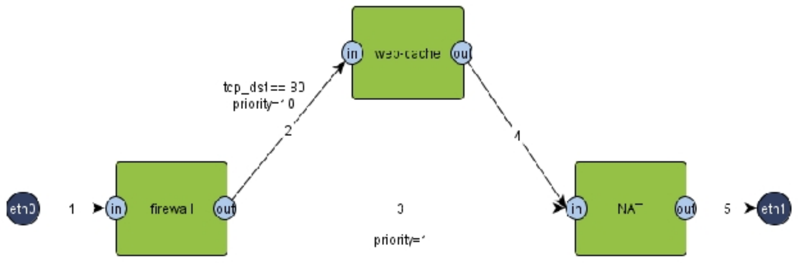


Figure 4-3: A simple NF-FG to be orchestrated.

With respect to our VNEP based work, we mainly orient the following sections along the division of online vs. offline algorithms. The VNEP is classically treated as an online problem, i.e. requests arrive over time without any knowledge about future requests and the orchestrator needs to decide which of the requests to accept or reject one by one. In contrast, offline algorithms are given a set of multiple requests to be orchestrated *at the same time*.

4.3 Online Algorithms

Online algorithms have the objective of finding the best embedding of a single request with respect to the currently available resources. Within the VNEP literature most proposed algorithms target the online use case in different settings with random network sizes (uniformly drawn number of nodes and uniform edge connections) and resource requirements (see [Chowdhury2009] for the work introducing the specific settings and [Fischer2013] for an overview). It must be noted that all works except [Even2012] and [Bienkowski2014] consider heuristics.

Within UNIFY the following contributions to the state of the art have been made:

- On very dense graphs, where nodes are connected via multiple edges, we have considered randomized graph search algorithms to find suitable end-to-end paths (see Section 4.3.1)
- For a very specific virtual topology, namely the virtual cluster which is often used in data centers, it was shown that the online embedding problem can be solved in polynomial time. While the virtual cluster topology was specifically designed for the data center use case, it shows that not all embedding problems are actually hard to solve and may be a step towards approximation algorithms or competitive online algorithms (see Section 4.3.2)
- For the joint decomposition and embedding of service chains a simple graph search was devised to find a suitable embedding quickly (see Section 4.6.3 for details)

4.3.1 End-to-end Embedding on Dense (Multi-)Graphs

In the VNEP literature graphs the substrate and the request networks are generally considered to be simple directed graphs, i.e. two nodes are connected by at most one edge. While this model generally is quite suitable, it fails to recognize that especially when considering an aggregated network view, two routers may be connected by multiple connections with varying available bandwidth and varying latencies. While a multi-graph, i.e. a graph with multiple edges connecting two nodes, can always be represented as a simple graph by introducing one additional node per edge, algorithms on simple graphs may fail to identify the underlying graph structure, hence increasing the algorithm's runtime manifold. Multi-graphs are generally of interest as e.g. ISPs use multiple connections to connect PoPs for the sake of resiliency. The work presented in this section can help in designing faster algorithms for end-to-end computations.

A problem field where multi-graphs naturally arise are so called IXP multi-graphs, where nodes are IXP locations and are connected by multiple transit ISPs (see Figure 4-4). On this graph, we have considered the problem of installing end-to-end paths from a source to a destination via a centralized CXP controller / orchestrator.

1. The path may carry some specific amount of bandwidth, and
2. The path has a total latency less than specified by the customer.

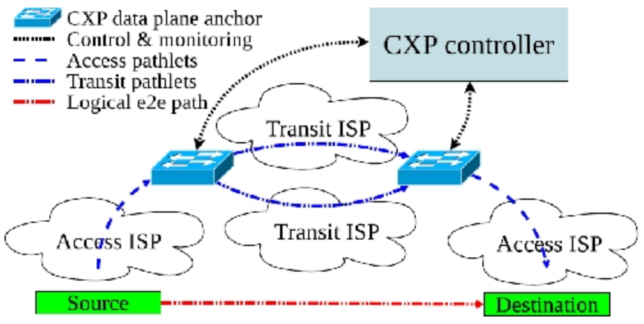


Figure 4-4: Inter-IXP network with multiple potential transit ISPs.

Three different graph search algorithms have been considered in a sample-and-select framework. First a set of feasible paths is sampled from the exponential number of potential paths. According to an arbitrary complex selection criterion, the best of the found paths is chosen to embed the path.

The above approach is especially useful in the light of the fact that choosing one path optimizing multi-objectives is generally NP-hard and opting for an optimal path may be prohibitive based on the runtime [Garroppo2010]. We have devised three graph search algorithms presented in the following sections.

4.3.1.1 Perturbed Dijkstra Path Sampling

The first algorithm is a simple adaption of the well-known Dijkstra algorithm for finding shortest paths. The algorithm proceeds as follows:

1. First all edges with an available bandwidth less than the specified bandwidth are pruned from the graph.
2. The resulting multi-graph is projected onto a simple graph by only considering the lowest latency edges and the classic Dijkstra algorithm is used to determine the lowest latency path in this network.
3. The simple graph is perturbed by removing all utilized edges and replacing them with the next best (latency wise) edge.

We note that the above algorithm can naturally be used to yield multiple edge-disjoint paths to support resiliency.

4.3.1.2 Guided Random Walk Path Sampling

This algorithm performs a guided random walk to connect one endpoint to another. Beginning in the start of the path, any neighbouring node is selected with the same probability. However, to guarantee termination only neighbouring nodes are considered that still allow reaching the target node within the required latency. To this end, using a reverse Dijkstra iteration (as already proposed in [Korkmaz2001]), first the minimum distances from each node towards the target is computed. If a suitable neighbour was found, then one of the feasible edges is chosen.

4.3.1.3 Guided Randomized Dijkstra Sampling

This algorithmic variant of Dijkstra combines the classical algorithm with a random edge selection: instead of projecting the multi-graph onto a simple graph the set of edges is projected onto a single suitable edge in the exploration of neighbours. Using the information on the minimal distance from each node towards the target, the algorithm is guaranteed to find suitable paths.

4.3.1.4 Online Evaluation of the Path Sampling Algorithms

The above algorithms were compared on an IXP multi-graph with 28 nodes and approximately 6,500 edges with an average node degree of 230 and an average edge multiplicity of 12.

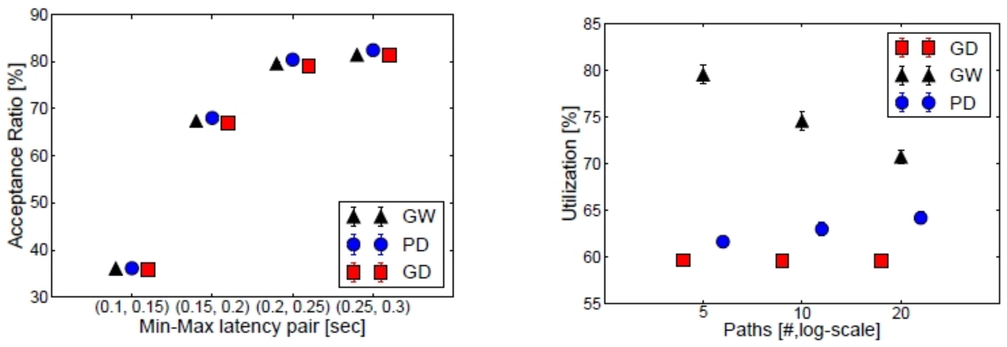


Figure 4-5: Acceptance ratio and resource utilization of a 28 node IXP multi-graph in the online scenario. On the left side, the acceptance ratio is plotted over the required latency. On the right side, resource utilization is plotted against the number of paths generated in the sampling stage. (GW = guided walk, GD = guided Dijkstra, PD = perturbed Dijkstra)

As can be seen in Figure 4-5 the three different sampling algorithms achieve around the same acceptance ratio (i.e. the number of requests that can be accepted while respecting the request's resource requirements) while varying heavily in the bandwidth utilization. While the perturbed Dijkstra algorithm achieves the best acceptance ratio, the guided Dijkstra variant achieves the best utilization while the guided random walks algorithm requires up to 20% more resources as the competing algorithms.

4.3.1.5 Application to Service Chain Embeddings

While the above scenario of embedding end-to-end paths can be considered the simplest service chain embedding, namely without any network functions, the above discussed algorithms pertain also to the service chain embedding problem by using a specific graph construction introduced in Merlin [Soulé2013]: given a network function forwarding graph consisting only of a line, the substrate network can be extended such that any path connecting the source to the end must traverse all network functions.

4.3.2 Virtual Cluster Embedding Problem

Based on the general NP-hardness of the VNEP (see e.g. [Fischer2013]), in the above setting only heuristics were used to find solutions within polynomial time. However, the found solutions may be far from optimal. While Integer Programming approaches can be used (see e.g. Sections 4.4 and 4.5) to compute (near-)optimal solutions in *non-polynomial* time, approaches yielding strict performance guarantees were only considered in the works of [Even2012] and[Bienkowski2014]. Within UNIFY we show that a quite specific topology can actually be embedded optimally while previous works always considered this to be NP-hard [Rost2015].

Concretely, the virtual cluster network topology is considered (see Figure 4-6). The virtual cluster topology (VC) is a simple star topology with servers being connected in a star-like topology to a central switch. The virtual cluster topology is especially used in data centres as the central switch allows for any server-to-server (VM) communication pattern that does not exceed the specified amount of bandwidth between the servers and the central switch. In

[Rost2015] we derive a flow algorithm that can be used to solve the embedding of virtual clusters onto arbitrary substrate topologies in polynomial time, showing that this specific embedding problem is actually in P, i.e. solvable in polynomial-time (see Section 4.3.3). While the Virtual Cluster topology can be applied to embed services inside data centres, the direct usage in service chains seems to be rather limited. Nevertheless, our result makes room for the potential existence of algorithms with strict quality guarantees for other specific VNEP cases, as e.g. embedding a simple line or embedding trees. Indeed, the ability to optimally solve these subproblems would give rise to novel algorithms in the service chain context that would first decompose the request and then embed the subgraphs independently.

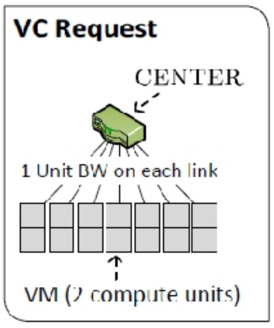


Figure 4-6: Virtual Cluster abstraction: a set of servers is connected via a central switch.

4.3.3 Flow Algorithm for Optimally Embedding Virtual Cluster Embeddings

At the heart of the algorithm VC-ACE (see Figure 4-7) lies in the observation that the virtual embedding problem can be reduced to a series of flow problems on an extended substrate graph. We exploit the following facts:

1. The required bandwidth B and the respective compute resources C of each VM in a virtual cluster is the same. As connections between the VMs and center are embedded as unsplittable paths, the substrate's edge capacities (and costs) can be normalized for the unit request case.
2. Assuming that the VM mappings as well as the location of the center are fixed, the cost-optimal link mapping can be computed in polynomial-time: Concretely, the minimum-cost unsplittable multi-commodity flow problem can be transformed into an integral minimum-cost single-commodity flow problem in the following way. We introduce a super source and ask for an integral minimum-cost flow of value equal to the number of VMs from the super source to the location of the center (which is fixed). The equivalence of these problems follows from construction, since the bandwidth demands are uniform and edge capacities are integral.
3. Assume that the mapping of center is fixed. The mapping decision for the VMs can be incorporated into the above described integral minimum-cost flow problem in the following way. The super source is connected to all substrate nodes via edges whose capacities equal the number of VMs the node may host and appropriate unit costs for hosting a VM. In the above construction, when considering an integral minimum-cost flow from

the super source to the fixed center location, the flow may only originate at the super source and leave the center's location. Hence, if a feasible flow exists, the flow from the super source to the substrate nodes can be identified as the number of virtual machines that shall be hosted on the respective substrate node. By construction, the node capacities cannot be violated and the costs are correctly accounted for. Furthermore, since all flows must be forwarded to the center, all the VMs are actually connected to it.

The above insights are instrumental for designing VC-ACE and for understanding its correctness. Based on **3.**, if the virtual switch's mapping is fixed, then the optimal embedding can be computed by solving a single integral minimum-cost flow problem (see Line 6).

On a specifically constructed graph (see Lines 3-5). For each possible location of center the optimal flow together with the mapping of center is stored (see Lines 7,8). Lastly, if a feasible flow was found, it is decomposed into a mapping as outlined in above by constructing arbitrary paths from the super source to the location of the center. VC-ACE has a polynomial runtime, which is dominated by solving one flow problem per potential center location. By employing the Successive Shortest Paths Algorithm, a runtime of $O(N(n^2 \log n + n \times m))$ can be obtained, where N is the number of virtual nodes, n the number of nodes, and m the number of edges in the substrate respectively.

```
   Input:  Substrate $S = (V_S, E_S)$, request $\mathrm{VC}(\mathcal{N}, \mathcal{B}, \mathcal{C})$
   Output: Optimal VC mapping $\mathrm{MAP}_V, \mathrm{MAP}_E$ if feasible
 1 $(\hat{f}, \hat{v}) \leftarrow$ (null, null)
 2 for $v \in V_S$ do
 3     $V_{S'} = V_S \cup \{s^+\}$ and $E_{S'} = E_S \cup \{(s^+, u) | u \in V_S\}$
 4     $\mathrm{CAP}_{S'}(e) = \begin{cases} \lfloor \mathrm{CAP}(e)/\mathcal{B} \rfloor & \text{, if } e \in E_S \\ \lfloor \mathrm{CAP}(u)/\mathcal{C} \rfloor & \text{, if } e = (s^+, u) \in E_S \end{cases}$
 5     $\mathrm{COST}_{S'}(e) = \begin{cases} \mathrm{COST}(e) \cdot \mathcal{B} & \text{, if } e \in E_S \\ \mathrm{COST}(u) \cdot \mathcal{C} & \text{, if } e = (s^+, u) \in E_S \end{cases}$
 6     $f \leftarrow$ MinCostFlow($s^+, v, \mathcal{N}, V_{S'}, E_{S'}, \mathrm{CAP}_{S'}, \mathrm{COST}_{S'}$)
 7     if $f$ is feasible and cost($f$) < cost($\hat{f}$) then
 8         $(\hat{f}, \hat{v}) \leftarrow (f, v)$
 9 if $\hat{f}$ = null then
10     return null
11 return DecomposeFlowIntoMapping($\hat{f}, \hat{v}$)
```

Figure 4-7: Flow algorithm to optimally embed virtual clusters on any topologies.

4.4 Hybrid Online-Offline Cooperation

While the above presented online algorithms can be used to embed requests upon their arrival, changes in the substrate (e.g. based on requests being terminated) can render the overall embedding sub-optimal. We therefore propose to couple the online algorithms with appropriate offline algorithms. While for the VNEP many different optimal offline algorithms on the basis of Integer Programs (IPs) have been proposed, these algorithms generally have a large runtime, rendering them hard to use for quick reconfigurations.

We have therefore proposed to use the Integer Program (IP) in $Y_{P_R} \in \{0,1\}$ $\square R \in \mathcal{R}, P_R \in \mathcal{P}_R$ $(\mathrm{HP}-4)$

Figure 4-8 to trade-off computational complexity and runtime by first generating a limited set of paths for each request first and then choosing one of these paths by the heuristic online methods discussed in Section 4.3.1. In this IP only two types of variables exist. The binary variable x_R decide which whether request R is to be embedded while y_{P_R} determines which path shall be used to embed R. As the paths' latencies are valid by construction, only the bandwidth resource requirement has to be enforced in Constraint HP-2. The Constraint HP-1 states that if a request is embedded then exactly one of the pre-computed paths must be chosen to embed the request.

$$\max \sum_{R \in \mathcal{R}} \mathcal{X}_{\mathrm{R}} \qquad \text{(OBJ)}$$

$$\mathcal{X}_R = \sum_{P_R \in \mathcal{P}_R} y_{P_R} \qquad \square R \in \mathcal{R} \qquad (\mathrm{HP}-1)$$

$$bw_e \geq \sum_{R \in \mathcal{R}, e \in P_R} bw_R \cdot y_{P_R} \qquad \square e \in E_G \qquad (\mathrm{HP}-2)$$

$$\mathcal{X}_R \in \{0,1\} \qquad \square R \in \mathcal{R} \qquad (\mathrm{HP}-3)$$

$$Y_{P_R} \in \{0,1\} \qquad \square R \in \mathcal{R}, P_R \in \mathcal{P}_R \qquad (\mathrm{HP}-4)$$

Figure 4-8: Heuristic offline Integer Program to maximize the number of request embedded.

In the realm of the IXP graph, we have performed a large scale computational evaluation on a graph with 14 nodes and 3.9k edges and 10,000 requests. Using the path sampling algorithms we have computed 20 potential embeddings for each request and then employed the above Integer Program to obtain a solution. Our results are shown in Figure 4-9: The coupling of the heuristic Integer Program with the guided walk scheme achieves near perfect acceptance ratio with respect to the optimum, which is computed using the optimal Integer Program (OPT). Importantly, obtaining an optimal solution to the Integer Program is much faster than computing the optimal Integer Program which allows for any embeddings: while the average runtime lies below 5 seconds, the optimal integer program comes close to the prohibitive runtime of 10,000 seconds.

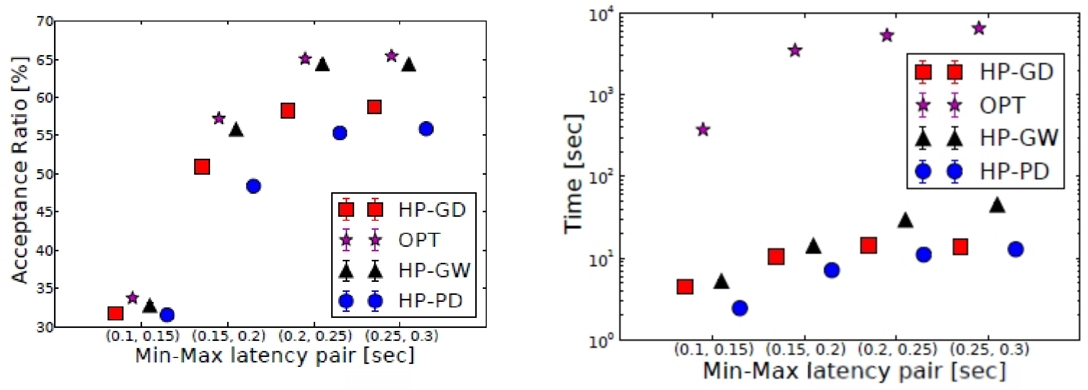


Figure 4-9: Comparison of optimal solution value with the heuristic approaches which compute first a set of paths and then applying the heuristic MIP (see $Y_{P_R} \in \{0,1\}$ $\quad$ □$R \in \mathcal{R}, P_R \in \mathcal{P}_R \quad (\mathrm{HP}-4)$

Figure 4-8).

4.5 Pure Offline Algorithm: DiVINE

The Hybrid Offline-Online approach presented in Section 4.4 is appropriate for very quick recomputations, if for each request already a set of potential embeddings have been (pre-)computed. If service chains with many different network functions are considered, much more embeddings need to be sampled as the consideration of only a few combinations of network function instances may yield to bottlenecks at these specific instances. We are therefore also targeting the offline embedding problem, where more than one service chain is to be orchestrated, in its pure form, i.e. without the restriction on a specific set of potential embeddings.

As offline computations will be most likely used to reconfigure existing embeddings, we are working on the DiVINE Algorithm which is characterized by the following properties:

1. DiVINE uses the rather standard Mixed-Integer Programming approach to the general VNEP as a basis, but extends it with the following two features:

    a. DiVINE allows for reconfigurations, i.e. given a previous embedding of a request, the tool may reconfigure single node or link mappings at a certain reconfiguration cost.

    b. DiVINE allows for elasticity, i.e. given a previous request specification, the customer may ask to e.g. increase the bandwidth on a certain links.

2. DiVINE tries to significantly speed up finding good solutions by sacrificing aiming at optimality: it employs a large-neighbourhood search procedure similarly to the one proposed in [Fischetti2004] to improve solutions locally.

As we are still evaluating the impact of different DiVINE parameters on the performance, we only outline our preliminary results, mainly stressing our methodological approach.

4.5.1 Outline of the Initial Computational Evaluation

While most of the works in the general VNEP area use purely synthetic substrate graphs and (random) requests, we are using real-world topologies collected in the Topology Zoo [Topology].

Figure 4-10 gives an overview on the size of most of the Topology Zoo instances. The Topology Zoo contains approximately 260 different instances that mainly have been reverse-engineered. The types of topologies vary from small-scale ISPs of a single country, to pan-European research networks and international backbones. As we are targeting mainly ISPs, the Topology Zoo instances are very well suited for our task.

For our initial evaluation we have chosen four major topologies:

- Bell Canada, Bell's ISP and corporate network in Canada before 2005.
- China Telecom's ISP network of China as of 2010.
- UUNET's ISP topology spanning the United Stated of America in 2011.
- GÉANT's pan-European research network as of 2012.

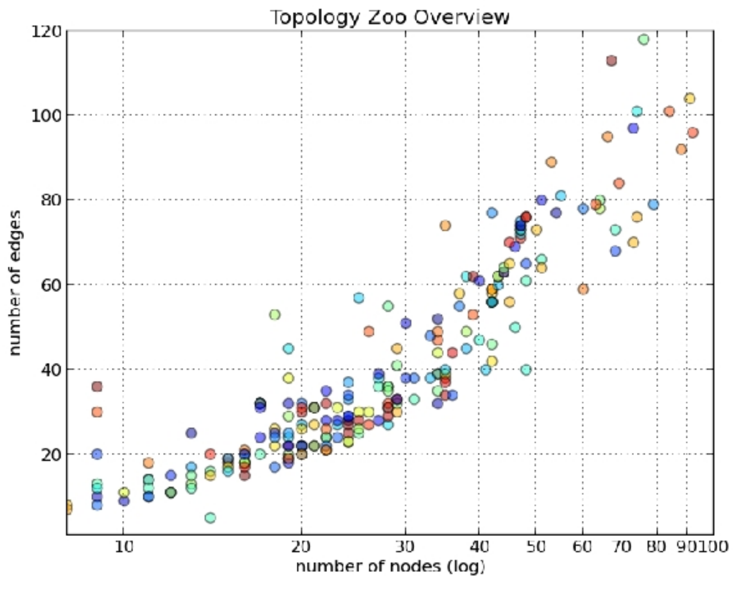


Figure 4-10: Overview of the topologies contained in the Topology Zoo.

4.5.2 Understanding the Out-of-the-Box Performance of the Optimal Offline Mixed-Integer Program

Before evaluating the performance of DiVINE, we have conducted a large scale study to understand which of the VNEP problem instances are hard to solve for today's optimization frameworks like Gurobi [Gurobi2015].

Our initial evaluation therefore considered the pure Integer Programming formulation using the following parameter combinations, i.e. the cross-product:

- Substrate
    - One of the four above discussed topologies.
    - Resources of nodes and edges are set statically to 100.
- Requests
    - The number of requests was chosen to be 100 over all experiments.
    - For each request the number of nodes was chosen uniformly from the interval [5,10].
- Varying factors for simulation:
    - Connection probability: Any two nodes of the virtual networks were connected with a probability of either 0.2 or 0.3.
    - Potential node factor: For each virtual node either 10%, 30%, or 50% of the substrate nodes are chosen as candidates for embedding. This restriction is very similar to the ones in the service graph embedding, where only specific nodes in the network may be able to execute a specific function.
    - Node resource factor: Requested node resources are chosen uniformly at random, such that – if all nodes were embedded – the load on all substrate nodes was 75%, 100% or 125%.
    - Edge resource factor: Requested edge resources are chosen uniformly at random such that on average the hop distance between embedding links may be less than 0.5, 1.0, 4.0 or 8.0 to embed all links.

Together with three different settings for the objective that determines the cost to benefit ratio of embeddings and three different general variability options, we have generated one experiment for all combinations of the above parameters (2,592 experiments overall). Using our implementation of the offline VNEP, we terminated the experiments after one hour of single-threaded execution time. While we consider also other metrics as e.g. the resource utilization, for now the main metric of interest is the so called objective gap. Since Integer Linear Programs are solved using branch-and-bound techniques which rely on linear relaxations to bound the objective value, such formulations always allow to estimate the quality of found solutions with regard to this automatically computed bound (as we consider the maximization of the profit here, this bound is an upper bound). The objective gap is then defined as the relative quotient $\frac{|P-D|}{|D|}$, where P denotes the (primal) objective value of the best found solution and D denotes the (dual) bound based on the linear relaxations which upper bounds the best possible solution. The relative objective gap therefore measures how far away from the optimum a solution may lie.

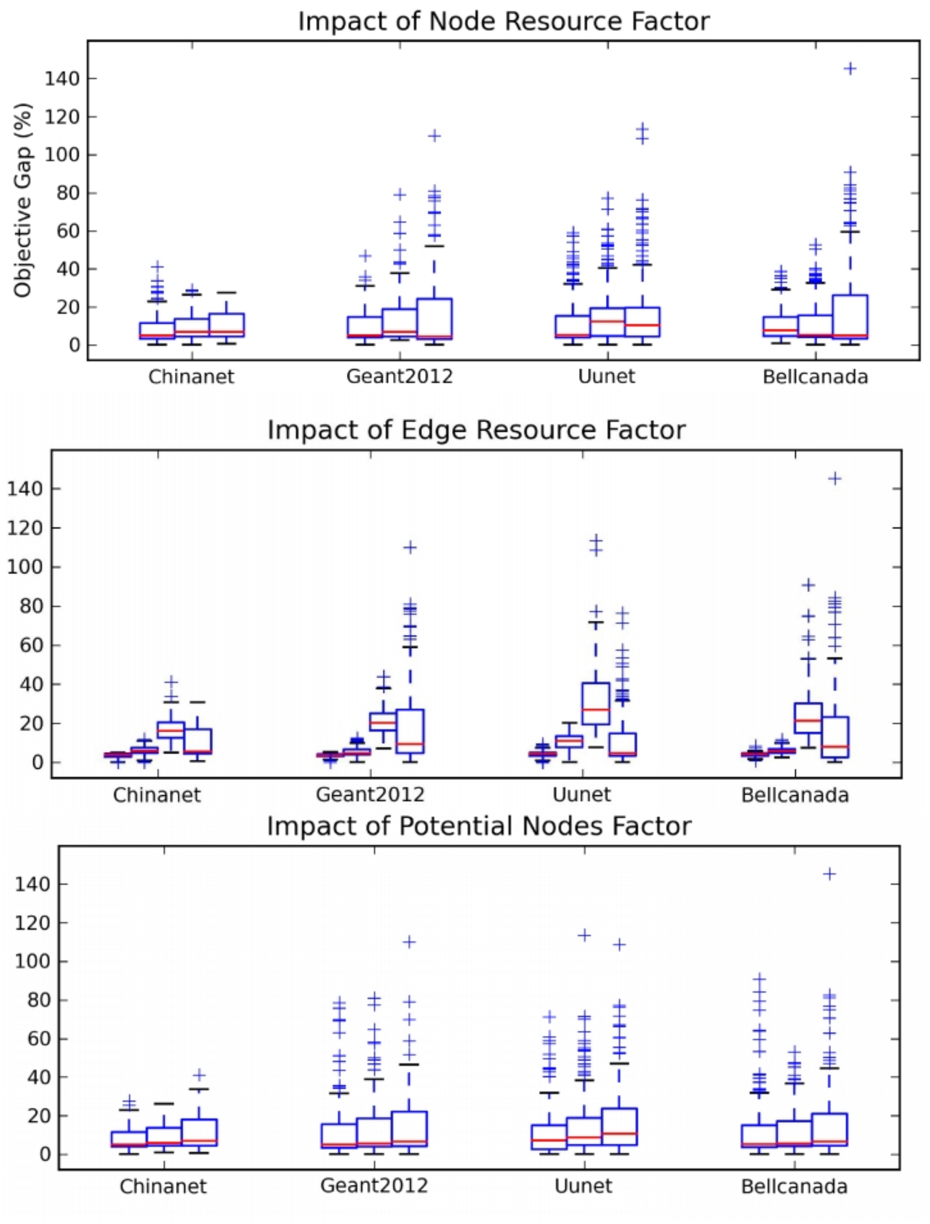


Figure 4-11: Impact of the different parameters (implicitly enumerated on the x-axis for each topology) on the objective gap on the different topologies.

Figure 4-11 summaries our initial results with standard R box plots, i.e. 25% – 1.5IQR and 75% + 1.5 IQR quantiles (IQR here denotes the interquartile spread):

- The objective gap of the Integer Program mainly depends on the node / edge resource factors, i.e. how over-subscribed the substrate would be if all requests could be embedded.

- The edge-oversubscription factor especially drives the objective gap.
- The number of substrate nodes onto which a virtual node may be embedded however does not clearly influence the objective gap.

4.5.3 Initial DiVINE Evaluation

Based on our initial assessment of the performance of the standard solution approach, we have selected a set of 27 scenarios per topology for our initial evaluation of DiVINE. We have chosen the three highest edge resource factors as these scenarios show the largest variance in the objective gap (see Figure 4-11).

Together with a preliminary selection of 24 parameter combinations for DiVINE we have obtained the following initial results depicted in Figure 4-12 and Figure 4-13. Figure 4-12 shows the ECDF of reaching a objective gap (33%, 66%, 100%) over time. As can be seen, DiVINE finds good solutions much quicker. Especially in the range beneath 33% of optimality, the pure MIP approach struggles in the first 30 minutes, with DiVINE clearly outperforming the pure MIP approach from approx. 400 to 2800 seconds of execution time.

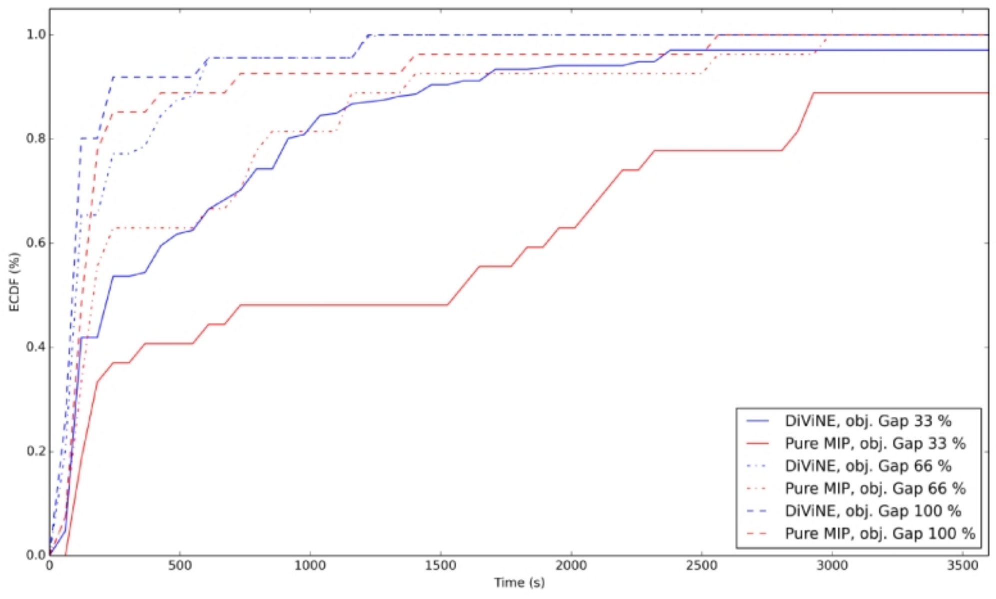


Figure 4-12: ECDF of reached objective gap over time.

This behaviour is not singular to the choice of the 33% barrier, as Figure 4-13 illustrates: the figure shows the difference of the ECDFs to reach a certain objective gap over time. Beginning at around 200 seconds, a hill between 20% and 50% objective gap with a maximal height of 40% indicates that DiVINE clearly outperforms the pure MIP.

It must be noted that within this first initial evaluation the ECDF of DiVINE is computed over the 24 different parameter settings used in our evaluation. The next step will therefore be to analyse the impact of the different parameters and determine the parameter set yielding reliably the best results.

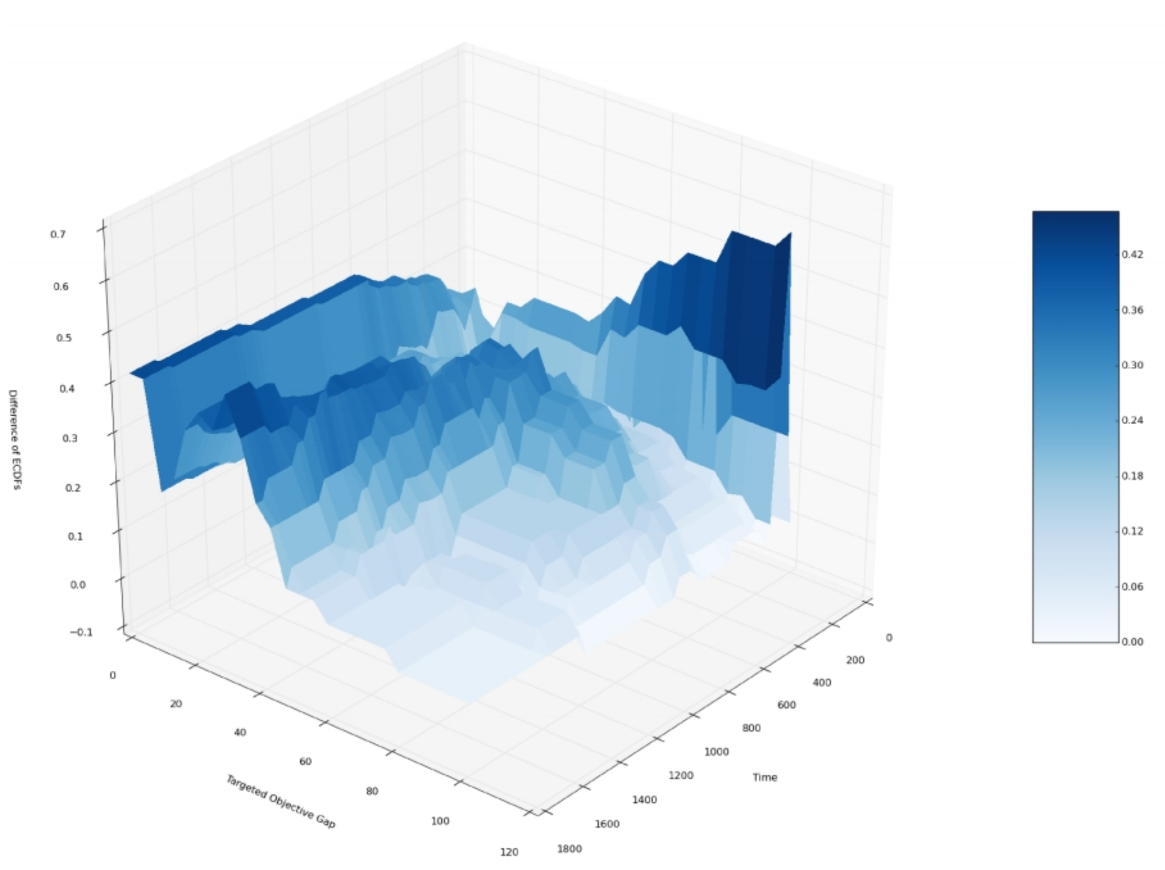


Figure 4-13: 3D representation of the difference of ECDFs of DiVINE vs. the pure MIP on the Geant topology. The standard MIP approach is outperformed by DiVINE, especially in the area of 20-40% of optimality.

4.6 Service chain embedding supporting service decomposition

UNIFY allows for decomposing compound network functions into several compound / atomic network functions. Service decomposition and related challenges were already explained in detail in Section 3. The decomposition chosen may / should highly depend on the actual resources available in the substrate (i.e. which hardware / VNFs are used, what are the different compute, storage, network requirements etc.).

4.6.1 Problem description

Service requests arrive over time and the embedding algorithm should decide whether the NFs within the requested SG and their corresponding connections can be mapped to the components of the physical network or not. Once requests are accepted, the required resources (physical links and nodes) are assigned and they are released once the requests expire.

Atomic NFs within an SG can be of different types which mean that they can be implemented in different ways using different techniques such as Hardware-defined NFs, Software-defined NFs, Virtual Machine-based NFs, and Container-based NFs. Having several types of NFs imposes additional constraints on the embedding problem because not all physical components of the network support all types. In this work, we prioritize an SG decomposition in which the number of same-type NFs which are directly interconnected is larger. The reason is twofold, first to reduce the amount of required compute and network resources, and second to improve NF performance by reduce communication overhead and latency. Prioritizing NFs of the same type likely lead to less compute resource consumption as more NFs can be mapped to the same physical node; this also leads to less network resource consumption as no physical links are used for the mapping. Interconnecting NFs over physical links implies additional communication and/or computational overhead due to e.g. additional tunnelling requirements, which is not as high if NFs are located in the same physical node.

In order to enable this prioritization, we define a parameter referred to as Cluster-Factor (CF) which is calculated as follows for each decomposition: the NFs with similar types which are connected directly (without intermediate nodes with other types) are grouped in the same cluster. The number of clusters in the decomposition determines the CF of that decomposition.

Our objective is to minimize the embedding cost which is achieved by minimizing the resources consumed in the infrastructure to map a request. This allows accepting more requests over time and increases the acceptance ratio. As service decompositions are known from the design time, we can make a resource-aware decomposition selection taking decompositions CFs into account at the time of the embedding.

4.6.2 Optimal solution for service decomposition w.r.t. resources footprint

We developed a Mixed-Integer Program which finds the best decomposition and the respective best embedding.

In the model, the physical infrastructure is represented as an undirected graph, $G_p = (N_p, L_p)$. The infrastructure consists of nodes (N_p) connected via links (L_p). Each node has certain capacity in terms of computation, memory and disk/storage, and links have delay and capacity in terms of bandwidth. These resources are actually the residual capacities which are calculated based on the resources assigned to the NFs included in previous mappings. Each physical node can support different types of NFs (Hardware-defined NFs, Software-defined NFs, Virtual Machine-based NFs, and Container-based NFs).

An SG can be realized through multiple decompositions. Therefore, for each SG there exists a decomposition set, $Decomp_{SG}$. Note that this set contains all possible decompositions such that hierarchical decompositions are already "fully" resolved.

Each decomposition is represented as a directed graph, $G_{dc} = (N_{dc}, L_{dc})$, to support the dependency between different NFs. Therefore, the NFs in the decomposition are represented as nodes (N_{dc}) connected via the directed links L_{dc} in the graph. Each NF has some requirements in terms of computation, memory and storage and links connecting different NFs have requirements in terms of delay and bandwidth. Each NF can be implemented differently and thus can be of different type. Finally, each decomposition is assigned a Cluster Factor, CF, which is the number of clusters in the decomposition.

The objective is to minimize the total cost of the mapping while prioritizing the decomposition with lower CF. Minimize:

$$\sum_{dc \in Decomp_{SG}} \sum_{u \in N_p} \sum_{i \in N_{dc}} cost\ (i, u) + \sum_{dc \in Decomp} \sum_{e_{uv} \in L_p} \sum_{e_{ij} \in L_{dc}} cost\ (e_{ij}, e_{uv})$$

In the above formula, $Decomp_{SG}$ refers to the decomposition set for a given SG. N_P and L_p are the sets of physical nodes and links in the infrastructure. N_{dc} and L_{dc} represent the sets of NFs and links within decomposition dc.

Note that in this objective function, the cost is considered as the cost of the mapping multiplied by the CF. This way the decompositions with lower CF are prioritized. The complete ILP formulation is detailed in [Sahhaf2015b].

4.6.3 Heuristic solution for service decomposition w.r.t. resource footprint

We propose a heuristic-based approach to overcome the scalability limitation of the optimal solution. Similar to any heuristic approach, the proposed scheme compromises optimality for shorter execution time. We use the proposed optimal solution as a benchmarking reference. The proposed scheme is referred to as DSBM (Decomposition Selection-Backtracking Mapping algorithm), this algorithm comprises two phases: i) decomposition selection and ii) mapping.

4.6.3.1 Decomposition selection

Given the physical network, the Service Graph and its decompositions, the CF of each service decomposition dc is calculated. Additionally, for each NF in each dc, the number of candidate physical nodes with sufficient capacities which can potentially host that NF is counted. We define parameter p to be the minimum number of candidate physical nodes for NFs of a dc. This parameter is defined to enable a resource-aware selection. We select a decomposition which is less restricting (has more mapping options). Finally, we define a cost function which combines CF, p and n (n is the number of NFs in a decomposition).

$$C(dc) = \frac{a}{p_{dc}} + b \times CF_{dc} + g \times n_{dc}$$

In the decomposition selection phase, we select a decomposition with minimum cost $C(dc)$. The a, b and g coefficients are introduced to allow the impact of the different factors to be tuned.

4.6.3.2 Mapping

We cluster the NFs of the selected decomposition based on their types and connections (see explanation for CF calculation) and sort the clusters and their NFs based on their requirements in descending order. We start mapping the NFs of the cluster with maximum requirement.

For each of the unmapped NFs in the sorted list, we sort its corresponding candidate physical nodes based on their distance (hop count) to the used physical nodes in ascending order. We select the physical node from this sorted list by which the links connected to the NF can also be mapped in the physical network. If such a physical node does not exist, the algorithm backtracks to the previous mapped node and checks the next candidate and repeats the same procedure. The details of the heuristic algorithm with corresponding pseudo codes are available in [Sahhaf2015a].

4.6.4 Evaluation of heuristic solution versus optimal solution

The focus of the experiments is on quantifying the added value of considering service decompositions at the time of the embedding in terms of cost and acceptance ratio. In the simulations, we compare the heuristic-based approach with the ILP-based algorithm in network scenarios where ILP can be executed in reasonable time scale[5]. We evaluate the effect of employing service decompositions in both schemes. As there is no other approach which considers service decompositions in the embedding problem, we compared both ILP-based and DSBM algorithms with approaches in which decomposition is selected randomly. Table 4-2 presents the notations used for the compared approaches. In DSBM, we also evaluate the effect of different factors in $C(dc)$ on the performance of the embedding by tuning the a, b and g coefficients.

Table 4-2: List of compared algorithms

Notation	**Algorithm description**
ILP	Proposed optimal solution
DSBM	Proposed heuristic solution
ILP-5, ILP-10	ILP on SGs with 5 and 10 NFs respectively
DSBM-5, DSBM-10	DSBM on SGs with 5 and 10 NFs respectively
ILP-random	Optimal solution with random decomposition selection, mapped based on the ILP model
DSBM-random	DSBM with random decomposition selection in the selection phase

[5]This ruled out network scenarios with over 50 nodes.

4.6.4.1 Simulation setup

For the physical network we considered two scenarios: i) small network scenario and ii) large network scenario. We used topologies from the Internet Topology Zoo[Topology]. For the small network, we considered the "BT Europe" topology with 24 nodes and 37 edges. In the large network scenario, the "Interoute" topology is used. This network is composed of 110 nodes and 148 edges. For both scenarios, it is assumed that some of the nodes have general purpose servers supporting different virtualization technologies, and some of them have specific hardware appliances such as Firewall, this is set randomly for the nodes. The CPU, memory and storage capacity of the nodes and bandwidth of the links are numbers uniformly distributed between 100 and 150 in both network scenarios. The cost of each unit of capacity is 1. Finally, the delay of each physical link is randomly set to a number with uniform distribution between 10 and 50 time units.

The service requests arrive over time in a Poisson process with average rate of 4 requests per 100 time units. Each of which has a lifetime, exponentially distributed with an average of $\mu = 1000$ time units. Each request can be realized with a few decompositions which is a number between 2 and 5 with uniform distribution. The number of NFs within each of the decompositions is a number uniformly distributed between 2 and 10. The CPU, memory and storage demands of each NF are a number with uniform distribution between 1 and 20. The NF types are assigned randomly. The bandwidth requirement of each link is a number between 1 and 50, uniformly distributed. The maximum allowed delay of each link is set to 1000 time units. Each pair of NFs within a decomposition is connected with probability 0.5. The hardware used to run the simulation is Intel Xeon quad-core CPU at 2.40 GHz with 12 GB RAM. Each simulation scenario is iterated 10 times and the average over all the iterations is reported.

4.6.4.2 Performance metrics

We measure the following performance metrics to evaluate and compare the proposed schemes:

- Execution time: This metric measures the time used by a scheme to find an embedding for a service request.
- Acceptance ratio: It measures the ratio of the accepted service requests which refer to services that are successfully mapped to the physical network.
- Embedding cost: The embedding cost is equivalent to the amount of physical resources used for mapping a service request. In our evaluations, this cost is equal to the total CPU, memory, storage capacity of nodes and bandwidth capacity of the links which are reserved for a service request.

4.6.4.3 Evaluation results

Before detailing the evaluation results on the two network scenarios explained earlier, we report the results related to the execution time of the proposed algorithms on different size physical networks ranging from 10 to 50 nodes to observe the scalability behaviour of the schemes. The topologies were randomly generated and the service requests of two sizes, 5 and 10 were considered.

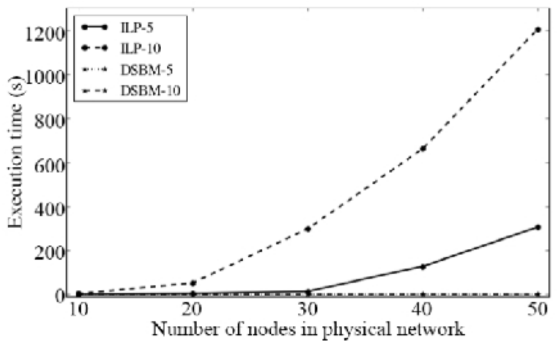
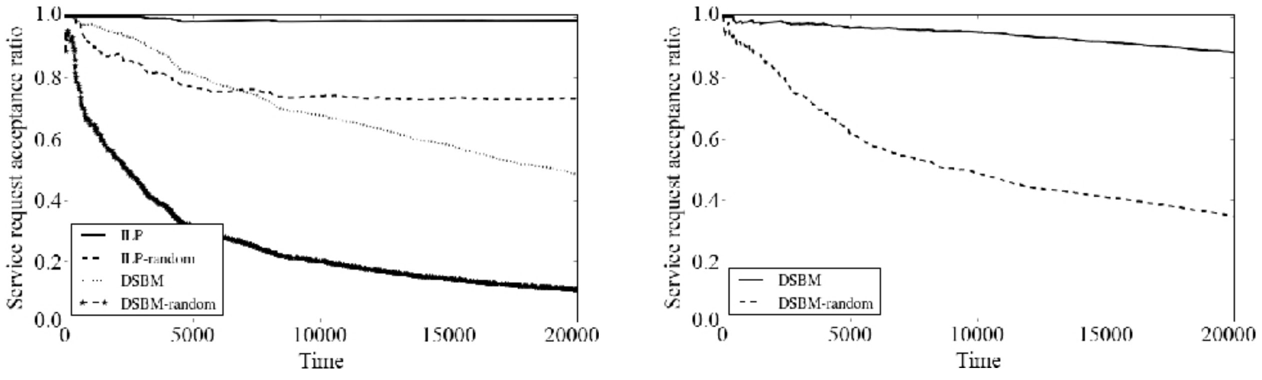


Figure 4-14: Execution time of ILP and DSBM for SGs with 5 and 10 NFs

Figure 4-14 depicts the execution time of different schemes. As expected, execution time of the ILP-based algorithm increases almost exponentially with the increase in the network size. This increase is more if the number of NFs in the service requests increases as well (see ILP-5 compared to ILP-10 in Figure 4-14). The heuristic-based approach scales significantly better and the execution time in DSBM-10 does not exceed a few hundreds of milliseconds.

We measured the average acceptance ratio and the corresponding embedding cost for service requests over time. We report these performance metrics against time to indicate how different schemes perform in the long run.



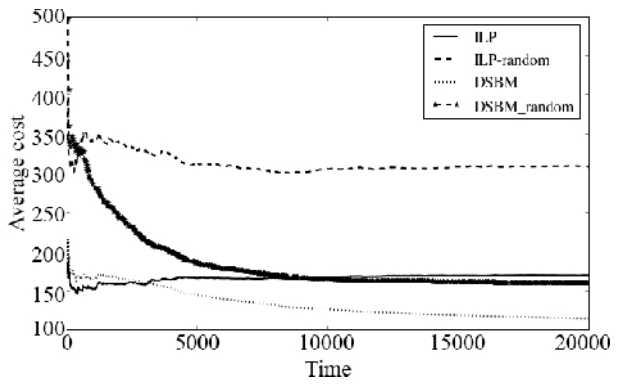
(a) BT Europe network (24 nodes)

(b) Interoute network (110 nodes)

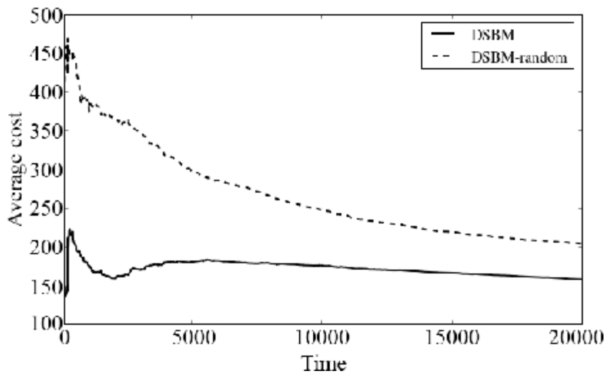
Figure 4-15: Service request acceptance ratio over time

Figure 4-15a depicts the service acceptance ratio in the two network scenarios for four different approaches: i) ILP, ii) DSBM, iii) ILP-random and iv) DSBM-random. In both network scenarios, the results indicate significant improvements in terms of acceptance ratio in the proposed ILP-based and heuristic-based algorithms compared to approaches in which a random decomposition is selected. The acceptance ratio of DSBM is higher in the Interoute network compared to the results in BT Europe which is the result of having more resources in the network. In DSBM the parameters in the $C(dc)$ function are set as follows: a = 0.25, b = 0.25 and g = 0.5. These parameters are tuned experimentally to achieve a reasonable performance. The effect of each factor in $C(dc)$ function on the embedding performance is evaluated and reported later in this section.

The average embedding cost in the four explained approaches are presented in Figure 4-16. Comparing the proposed schemes and the ones with random decomposition selection, we observe a significant difference in the average cost of the embedding. Both ILP-random and DSBM-random consume more resources compared to ILP and DSBM respectively. Additionally, the results indicate that the ILP-based solutions lead to almost constant average costs while the heuristic solutions result in decrease of the embedding cost in the long run. The reason for this behaviour is explained by the optimality of the embedding solution. In DSBM, due to sub-optimal placement of the service requests, less requests can be accepted in the long run (presented in Figure 4-15). Furthermore, as there are less available resources in the network compared to when an optimal placement is found, the service requests with less NFs can be accepted. This leads to a decrease in the average embedding cost over time. This behaviour is also visible in the Interoute network for DSBM-random approach which is less efficient than DSBM.



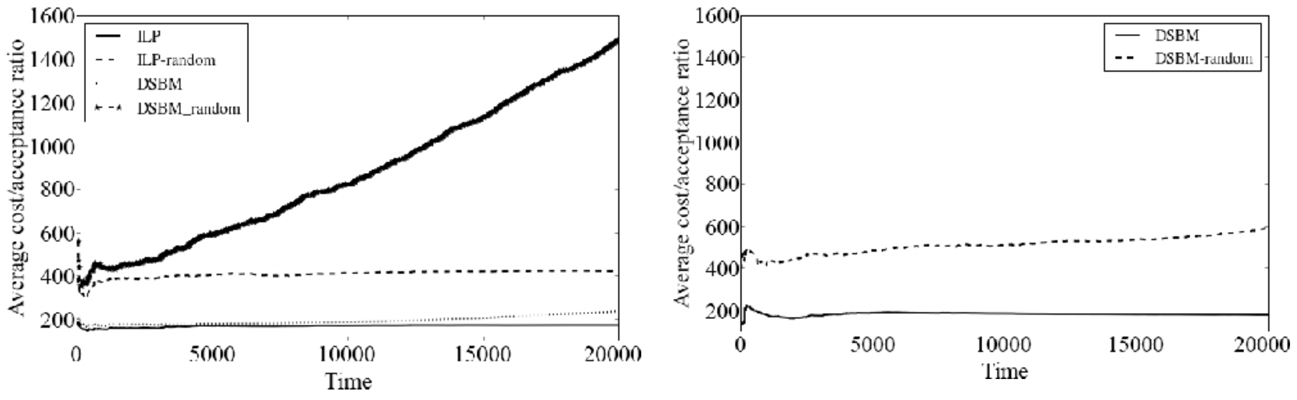
(a) BT Europe network (24 nodes)



(b) Interoute network (110 nodes)

Figure 4-16: Average cost of accepting requests over time

Considering the average cost alone without taking the acceptance ratio of the schemes into account might lead to incorrect conclusions regarding the performance of the different schemes (as mentioned above, the sub-optimality of the schemes might also lead to decreased cost). Therefore, in order to have a better view on the performance of different schemes, we report the value of the cost normalized by the acceptance ratio of the schemes. With these values we can have a better view on the performance/success of the algorithms. The results in Figure 4-17 indicate that both ILP and DSBM, in both networks, have very low normalized cost while the schemes with random decompositions lead to large increase in this value.

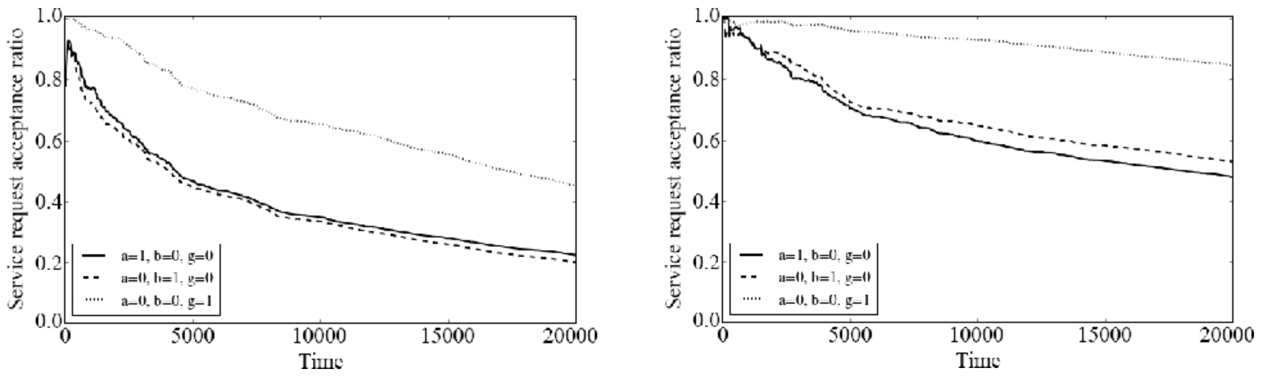


(a) BT Europe network (24 nodes)

(b) Interoute network (110 nodes)

Figure 4-17: Average cost normalized by acceptance ratio

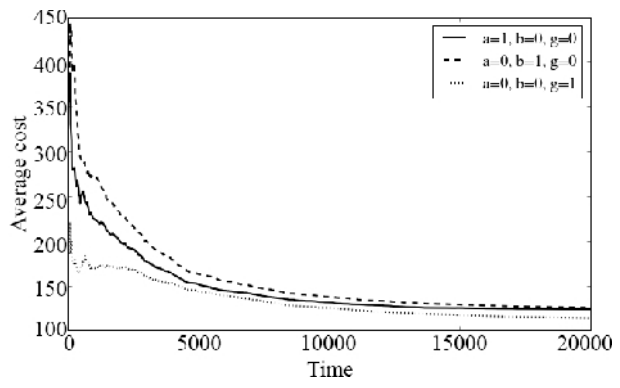
Next, we report the effect of different coefficients in the $C(dc)$ function of DSBM on the embedding performance. Figure 4-18 illustrates the service request acceptance ratio when only one of the three coefficients in $C(dc)$ is considered for decomposition selection, this evaluation identifies the main factor in $C(dc)$. Based on the results, considering the number of NFs in a request (g=1) leads to higher acceptance ratio compared to cases when only CF (b = 1) or p (a = 1) are considered. The last two cases result in very similar performance. The related average embedding cost is presented in Figure 4-19 and as expected, selection of decompositions with less number of NFs leads to lower cost compared to the other two cases. Similar to previous evaluation, in order to have a better view on the performance of the scheme, we report the normalized cost with acceptance ratio to have a fair comparison of the algorithm success in Figure 4-20.



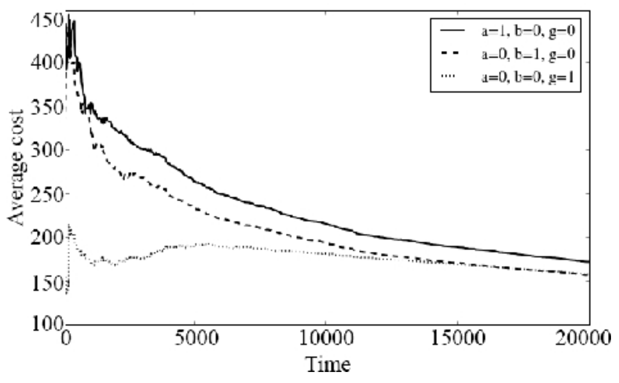
(a) BT Europe network (24 nodes)

(b) Interoute network (110 nodes)

Figure 4-18: Service request acceptance ratio over time in DSBM

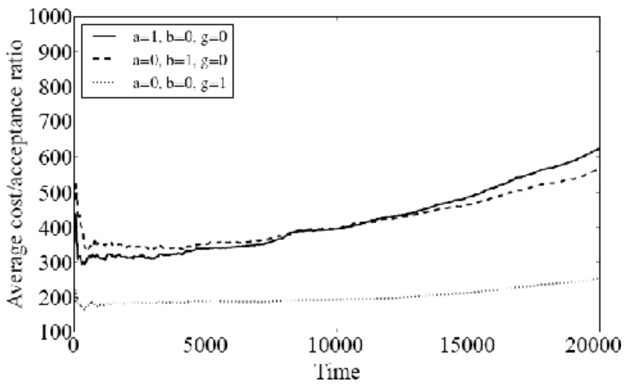


(a) BT Europe network (24 nodes)

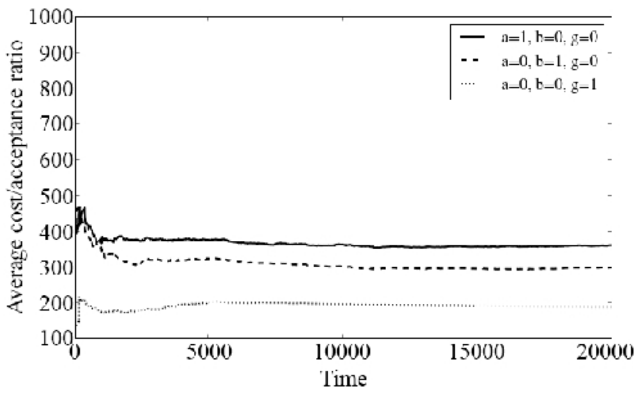


(b) Interoute network (110 nodes)

Figure 4-19: Average cost of accepting requests over time in DSBM



(a) BT Europe network (24 nodes)



(b) Interoute network (110 nodes)

Figure 4-20: Average cost normalized by acceptance ratio in DSBM

4.6.5 Evaluation of embedding time in heuristic solution

The goal of experiments in this section is to identify the major blocks in embedding time in an implemented proof of concept prototype. These blocks include: i) retrieving/reading of all decompositions from a Network Function Information Base (read dcmp), ii) decomposition selection (select dcmp) which includes the cost calculation for all the decompositions and selecting the minimum cost decomposition and iii) mapping of the selected decomposition (map). We have implemented the heuristic-based solution to use a Network Function Information Base (NF-IB) based on Neo4j graph database containing information on NFs and decomposition rules [Neo4j]. We evaluate the effect of increase in the topology size, SG size and number of service decompositions on the performance of the embedding.

As our intention is to evaluate the embedding execution time on physical topologies with different sizes, we have generated random regular networks with 100-1000 nodes with degree 3. Other parameters of these topologies (e.g., capacity) are similar to simulation setup explained in Section 4.6.4.1.

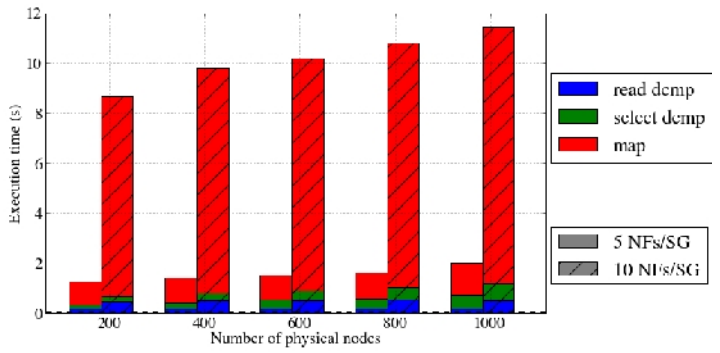


Figure 4-21: Embedding execution time for SGs with one decomposition

In the first experiment, we have evaluated the execution time of the embedding of an SG into physical networks of different sizes for two scenarios: i) SGs with 5 NFs and ii) SGs with 10 NFs. Each SG has only one decomposition. Figure 4-21 reports the execution time of different blocks (i.e. read dcmp, select dcmp and map) in the embedding. . Based on the results, the mapping is the dominant block and it increases significantly with the increase in the number of NFs, when only one decomposition exists for an SG.

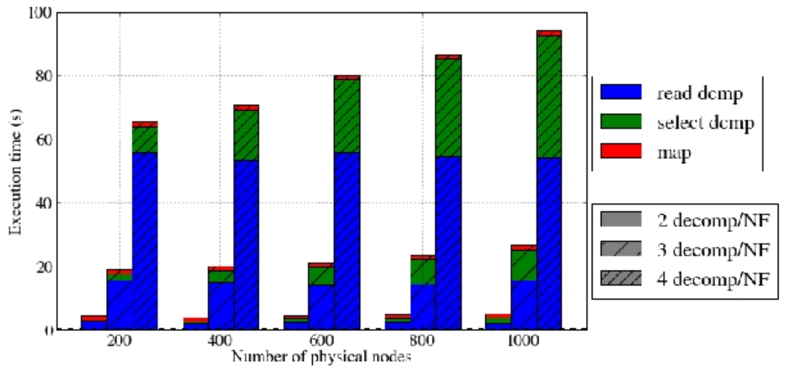


Figure 4-22: Embedding execution time for SGs with 5 NFs

The next experiment evaluates the effect of increase in the number of decompositions for an SG. Figure 4-22 reports the execution times for 3 scenarios in which the number of decompositions per NF in an SG changes from 2 to 4. As there are 5 NFs in each SG, the number of decompositions in each scenario is: 5^2, 5^3 and 5^4. As we see "map" and "read dcmp" blocks seem to scales quite well, whereas "select dcmp" is the block which scales poorly with increasing number of nodes in the network. For small topologies with few nodes, reading the decompositions is the dominant block while for larger topologies (1000 nodes and more) "select dcmp" seems to be the main concern. It is worth mentioning that the time spent for reading the decompositions from the NF-IB includes the time needed for calculating the service decompositions as well. This is because only NFs decompositions are stored in the NF-IB and possible service decompositions should be calculated based on them upon request.

Compared to the orchestration requirement mentioned in section 4.1 the performance of the proposed scheme might seem quite low (inefficient for orchestration of thousands of devices). The main goal of the experiments in this section was to identify the most time consuming blocks in the proposed embedding approach so that we can find solutions towards a more scalable orchestrator. Therefore, based on these identified blocks, in the next section, we propose several architectural enhancements to improve the scalability of the orchestrator.

4.6.6 Potential improvements

In this section, we propose several architectural enhancements and explain existing challenges in order to implement a highly scalable embedding.

4.6.6.1 Parallel/distributed embedding

Based on the results, we identified the "read dcmp" block to be the most time consuming block in the embedding in smaller topologies while “select dcmp” block seems to be a major issue in larger topologies. Changing the embedding algorithm to a distributed approach in which costly calculations are done in parallel can improve the performance of the embedding significantly. A possibility to parallelize the "read dcmp" block of the algorithm is to use Neo4j which supports High-Availability (HA) which is explained in more detail in Section 4.7. Using the Neo4j HA the "select dcmp " block can simply be computed in parallel as the cost calculation of each decomposition is independent of others. However, parallelizing the "mapping" block is a challenging task. This phase is equivalent to the typical VNEP as service graphs composed of atomic NFs are similar to virtual networks which should be mapped to a physical infrastructure and thus similar solutions to VNEP can be considered for the mapping phase.

The options for mapping parallelization are: i) considering all possible combination of NFs mapping to the physical nodes and selecting the minimum cost mapping. The feasibility/cost of each mapping can be checked in parallel. This approach is feasible only in small topologies (in the order of hundreds of nodes) as the number of combination increases drastically with a small increase in the topology size. ii) selecting the first-fit physical node for mapping of NFs and finding the shortest path between nodes. NFs mapping/path calculation can be done in parallel. If the first-fit mapping is not successful, the next one is selected. A challenge is to avoid different threads reserving the same resource. Batch scheduling is a solution in which each job gets dedicated access to the resources.

Section 4.7 explains a basic proof of concept framework for performing distributed embedding calculations.

4.6.6.2 Hierarchical embedding

The other alternative to achieve a scalable embedding is to have a hierarchical embedding process. In this process, SG can be divided into different subgraphs using service decompositions available in the NF-IB and each subgraph can be given to a different domain to be orchestrated locally.

The main challenge in such distributed embedding relates to the amount of resource and infrastructure information that needs to be advertised to the upper layer orchestrators to facilitate an efficient embedding process. Each domain may expose to upper layers only high-level and aggregated information such as total available capacities and capabilities or aggregated PoP-level information instead of detailed router-level topologies. Such incomplete information in the higher layer orchestrator might lead to inefficient embeddings with performance far from the optimal solution. It is a challenging task to identify the trade-off between the efficiency of the embedding and the amount of infrastructure information exposed by each domain.

4.6.6.3 Pre-defined service chains

Another enhancement option, independent of the embedding approach, is to have pre-defined service chains with pre-defined decomposition templates. With such templates different parts of the embedding can be done proactively, before the request arrives.

4.7 Distributed orchestration

To see if a distributed orchestration framework could be implemented using open-source components, and what its performance would be, we investigated several methods of distributing both the substrate topology and calculation requests. For the topology there are many candidates among graph databases that can be deployed in a distributed fashion, for example Neo4j, OrientDB, and Titan [Neo4j, OrientDB,Titan]. These have different characteristics in implementation and performance with Neo4j and OrientDB both providing low read/write latency and good throughput compared to Titan, which in turn can store very large graphs (billions of nodes and edges). As we favour speed over extremely large graphs we decided to use Neo4j, although OrientDB and others appear to have similar performance [Kolomicenko2013,Salim2013, McColl2014]. To distribute orchestration tasks we decided to use Hazelcast [Hazelcast], which in addition to task distribution also provides distributed data structures that can be used by the VNE algorithms to for example temporary store shared data such as intermediate results. There are many alternatives to parts of Hazelcast, for example we could use Java RMI to invoke commands on other nodes, but the simplicity of using Hazelcast with e.g. automatically node discovery make it a good initial candidate.

Each node in the framework runs Neo4j in embedded, high-availability mode (Master-slave replication) coupled with Hazelcast. In embedded mode the database is started and accessed through its Java API, providing a low-latency interface to the data. High-availability mode automatically distributes the database to all nodes, allowing database reads to scale linearly, however writing to the database scales less well as writes has to pass through the current Master node. There appears to be mechanisms for improving write throughput if necessary, through upgraded hardware or by reducing the consistency requirements. To distribute commands we use Hazelcast's distributed drop-in replacement of Java's *ExecutorService* which allows tasks to execute asynchronously tasks in available threads on nodes in the cluster, rather than in threads on the local node.

4.7.1 Implemented algorithms

To evaluate if the framework worked as expected (and if it could be used for the more complex algorithms described above), we implemented two simple algorithms using it. The first algorithm performs distributed embedding of a Service Function Chain defined in a Service graph on already running, sharable, NFs. The second algorithm is a previously existing heuristic VNE algorithm called VF2x, this algorithm was chosen to test the framework as the algorithms presented above were still being developed [Yin2012].

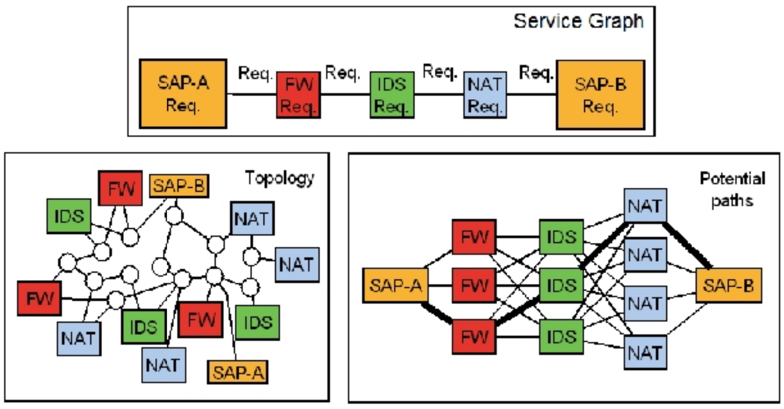


Figure 4-23: Example Service Graph, topology and potential paths, with shortest combined path in bold

The SFC algorithm takes as input an undirected graph with NFs connected as pearls on a string between two SAPs forming a basic Service Function Chain, and a topology with node and link capacity. From this input we calculate all potential paths between SAP-A and FWs fulfilling the requirements, we then continue with all paths from FWs to IDSs, from IDSs to NATs, and so on. This results in a graph of potential paths with aggregate costs on links, we use this graph to calculate the best end-to-end path from SAP-A to SAP-B that fulfils the requirements on the NF-FG. This algorithm is easily parallelized as all the path calculations are independent. We tried two ways of distributing these calculations; first by constructing a task as all pair-wise path calculations between NF types e.g. all paths between FWs and IDSs (called "chunk"), and secondly seeing each individual path as a task.

We tested these strategies using a topology of 19200 CPEs distributed on 300 DSLAMS connected to 20 MSANs that finally are linked by 4 Metro nodes in a tree with some meshing links between MSANs and Metro nodes. In this topology 10 FWs, 8 IDSs, 7 NATs, and 5 vCPEs are connected, mostly at MSAN and Metro levels. All nodes and links are randomly assigned costs. The NF-FG we test with connects SAP-vCPE-FW-NAT-IDS-SAP with link and node requirements, leading to a maximum of 239 + 1 path calculations necessary (SAP-vCPE is 8 paths; vCPE-FW is 50 paths, and so on). We tested this scenario in a distributed way with two orchestration nodes and locally on one node for the two strategies, results can be seen in Figure 4-24. Clearly the distributed individual calculation is the fastest with a median calculation time of 50 ms compared to over 100 ms for the other strategies. The reason why distributed chunk is not faster than the local chunk is that there are only five chunks in the tested NF-FG (as it has only six NF nodes), these chunks are in both the distributed and the local case assigned to local threads and thus not utilizing the second node in the cluster.

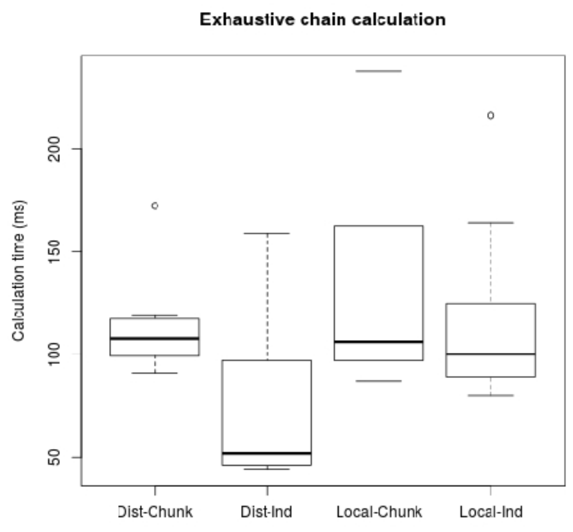


Figure 4-24: Boxplot of initial results for distributed and local SFC calculations. Each algorithm was executed 20 times.

The implemented VNE algorithm, VF2x, is currently not utilizing the distributed capabilities but is able to map more complex Service Graphs onto the topology described above. As this implementation is only operating on a single orchestration node as of writing, no elaborate evaluation has been made. However initial results for Service Graphs with tens of node components gives a calculation time in the order of tens of milliseconds using the above described topology. As this topology was developed to test the SFC algorithm it only has a few nodes that can be mapped, so these results should not be taken as indicative of performance in more realistic scenarios but rather to indicate that the algorithm is working and is able to find mappings.

Testing of the simple SFC algorithm shows that the distribution framework is working and provides expected benefits when enough tasks are executed in parallel. However more work is needed to try the setup with more nodes, investigate ways of utilizing the distributed capabilities for VNE algorithms such as VF2x and the heuristic algorithms discussed in section 4.6. It would also be interesting to see how performance is affected by a write load, for example by adding and modifying resources during calculations.

5 Scalable and resilient services

One of the merits of NFV is the ability to enable scalable telecom services without losing the reliability associated with hardware implementations of NFs. Service scalability shall be interpreted as the ability of a service to scale up or down, while service reliability refers to the ability of a service to remain operable over the considered time period. This section documents a set of techniques in order to enable service elasticity and increase service reliability. The core of both aspects is the ability to maintain service- or NF-related state under changing conditions. Changing conditions might involve an increase of demand in terms of throughput or number of requests, or might refer to failures of components of the deployed service such as failures in network connectivity, VNFs or the infrastructure which is hosting VNFs. Enabling elasticity or providing service reliability therefore involves the support of state migration between VNFs hosted on different infrastructure components.

This section follows a bottom-up approach by approaching the problem of scalability in a Border Network Gateway use case. This use case is central to UNIFY and has been documented in use-case UC5 (see D2.1), and as well it is in focus for the UN assessment in WP5. Additional reason for selecting this platform for the stateful scaling analysis is the fundamental role it plays in any service provider network scenario, along with the challenge that its state management poses on the scaling and resiliency requirements. The BNG performs the functions of an IP edge router that provides aggregation capabilities (e.g. IP, PPP) between the access network and the transport network which also includes functionality for subscriber management, advanced IP processing, including QoS, and enhanced traffic management capabilities. As such it is a complex and expensive node which often forms a bottleneck in provider networks which makes it a good target for a VNF implementation, both to reduce cost and improve scalability. Our study of the BNG internal state concludes that while state management is indeed needed for the BNG, the requirements are fairly lax.

While the BNG requirements on state management likely do not require complex mechanisms to meet, this is not the case for all VNFs. Several mechanisms for state-management in VNFs with higher consistency and quality requirements have been proposed, however they all fall short of fulfilling all requirements. One of the more complete systems proposed is OpenNF which is able to perform state migration synchronized with the traffic migration, with guarantees on packet loss and packet reordering [Gember-Jacobson2014]. In section 5.2 we investigate the OpenNF approach to state management and identify a number of bottlenecks that would make it unlikely to properly operate in a scalable fashion in full scale production environments. We then attempt to solve improve the performance of OpenNF in order to move the approach closer to a production environment. We do this by removing one of the bottlenecks, the reliance on a central controller on the critical path for both state and user traffic during a migration or scaling event.

Section 5.3 then discusses how protection and recovery mechanisms apply to VNFs and how for VNFs these differ from traditional server applications. Section 5.3.5 discusses and provides an example of how VNF management mechanisms like the one discussed in section 5.2 can be used to provide service resiliency.

5.1 Requirements on scalability mechanisms on BNG use-case

The Broadband Network Gateway (BNG) plays a crucial role in a Service Provider environment. BNG is an IP edge router that provides aggregation capabilities (e.g. IP, PPP) between the access network and the transport network. Beyond aggregation, it also supports policy management and IP QoS, being the edge device where service features, bandwidth, and QoS policies may be applied to subscribers. Therefore, its functionality is enhanced to include subscriber management, advanced IP processing, including QoS, and enhanced traffic management capabilities (e.g. hierarchical queuing/shaping).

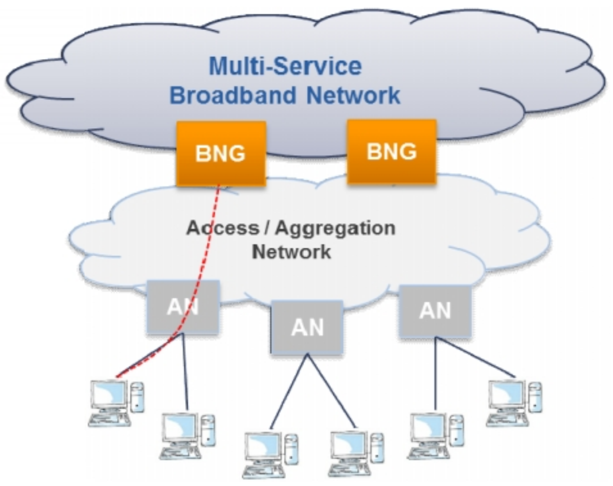


Figure 5-1: The BNG function at the edge of a Service Provider network

The BNG is hosting IP subscriber sessions, logical constructs intended to represent a network connectivity service instance at a node. The session is basically an abstraction of connections and resources associated with a subscriber IP end-point; it represents a subscriber's traffic, or a portion thereof, and is associated with an IP address or prefix. The notion of a session is essential, as data and control plane policies are associated with subscriber sessions. These may be initiated and configured dynamically or statically and do have associated state. Therefore scaling and migration mechanism need to properly deal with this stateful properties.

A subscriber session can typically be either a PPP Session or an IP Session (e.g. DHCP based), although in many environments the PPP is still the most commonly adopted. Subscriber sessions are used to represent all traffic that is associated with a particular subscriber by a given Service Provider in order to provide a context for policy enforcement.

5.1.1 Reasons for BNG scaling

The ability to support graceful and efficient stateful BNG scaling and session live migration mechanism can find useful application in various scenarios. Some of the possible reasons for supporting such scaling features are depicted in the following Figure 5-2.

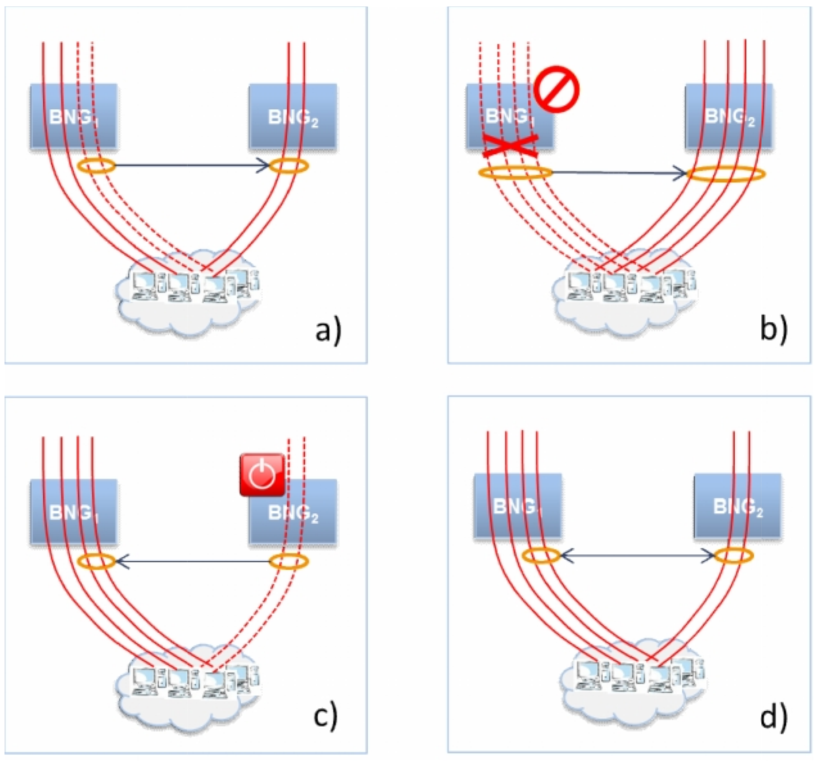


Figure 5-2: Possible scenarios for BNG stateful scaling/resiliency mechanisms

The scenarios identified so far have to do with both possible dynamic scaling and resiliency requirements of the BNG network function.

A first rather straightforward use-case is shown in Figure 5-2 a); this scenario corresponds to a classical scale-out use-case, i.e. when a high load on BNG1 causes performance issues, compromising the ability to continue meet the SLAs for subscribers, a number of active session needs to be migrated to a different BNG instance (BNG2), thus offloading the original network element.

A different and exactly opposite scaling situation, corresponds to the scenario depicted in Figure 5-2 c); here the focus is on consolidating the workload in order to enable cost savings. This scale-in capability may be triggered on a low load condition on the/some BNGs, for instance during off-peak conditions, when active sessions can be migrated and consolidated on a fewer devices in order to be able to switch-off some of the BNGs in order to reduce energy consumption and increase the resource utilization level.

A couple of use-cases are illustrated in Figure 5-2 b), these are related to live subscriber session migration for resiliency or network element maintenance reasons. The need for supporting such a switchover scenario may occur in different situations, e.g. a BNG needs to be taken out-of-service for a while in order to perform scheduled maintenance; on the other hand, a BNG may become partly unavailable or unreachable due to a fault, requiring a redundancy failover of the subscriber sessions. Both use-cases involve a stateful migration of the affected sessions to a different network element instance.

Finally, Figure 5-2 d) deals with the case of providing on-demand high-availability mode for active sessions. In this scenario, some sessions require dynamically to switch to high-availability redundant (dual-homing) configuration during their lifetime. Should this scenario be addressed, it would require the ability to duplicate the concerned subscriber sessions to another BNG, thus sharing the related state between the dual-homing BNGs.

5.1.2 State classification and performance requirements

The typical subscriber session life cycle comprises the stages of: a) session creation, b) subscriber authentication with determination and execution of applicable subscriber policies/profiles (triggered by the control plane on the BNG), and c) session termination.

Once a subscriber session is created, it is also necessary to monitor the subscriber session state via control or data plane events to be able to detect a number of possible conditions such as: explicit disconnect by the subscriber, the expiration of an inactivity timeout, or the excess of a volume quota. In addition, while the session is active, it may be necessary to change its state or some associated parameters.

The actions taken following session creation constitute the subscriber session policy. The policy is applied to a subscriber session either at session activation or dynamically following a subscriber or operator request. A traffic policy is typically built using the concept of IP Flow and an associated set of actions. IP Flow is a classification mechanism that allows specifying the portion of subscriber traffic to which policies need to be applied. Therefore, the BNG has to support configurable IP Flow classifiers. The IP Flow definitions are usually predefined or can be downloaded from a central repository at start up or periodically, and subscriber sessions are associated to them depending on their profile. IP Flow traffic classifiers, in their basic form, typically use the 5-tuple definition (i.e., destination IP address, source IP address, transport protocol, destination port, and source port).

The BNG function also has to deal with accounting records, representing a summary of information collected for the subscriber session. The BNG then needs to create and maintain an accounting record, according to its accounting policy, this is exported to the AAA (e.g. Radius) Server at subscriber session start, stop, and optionally at interim intervals during the session. The exported record at session start is mainly required to update the Radius server about the relevant information about the session, for instance by providing the IP address assigned to the subscriber, providing a relationship between Session ID and the IP address.

Session termination requires monitoring the possible failure of a “keep-alive” protocol, this is for instance the case for PPP based connections, or alternatively the session state derived from a control protocol (e.g. DHCP for IP sessions).

Most of the state associated with subscriber sessions consists of policy configuration state. These policy configurations templates are defined for a class of subscribers or profile, rather than for each individual subscriber, and usually they are already resident on the BNG. After the authentication the new session created is simply set to use the proper configuration template, depending on the profile information. However, sometime, when recalling the policy in the subscriber configuration, the BNG may also allow specifying a set of parameters that are specific to that subscriber. Therefore, once the session needs to be migrated to a different BNG, only a reference to the proper policy configuration template, already predefined and available on the new BNG, along with possible specific parameters associated with the session have to be moved.

Transient state, on the other hand, can be represented by state information dynamically created or updated during session lifetime. This dynamic state may typically consist of:

- State associated to keepalive timers (e.g. PPP, BFD) used to monitor session continuity; the PPP Echo Request messages, for instance, are typically sent every 10 s as a default value.
- Counters used by the policer(s) used to enforce traffic contract (e.g. bandwidth) parameters for the session; these counters are typically updated on a packet-by-packet basis for each session.
- Accounting records, possibly storing counters to track the session activity, these may be used for collection of statistical information or to enforce quota based profiles; the accounting record is typically updated on packet basis, to monitor traffic generated by the subscriber.

When trying to identify the actual state information that needs to be migrated when moving a session across different instances, we can consider separately the configuration and the transient/dynamic portions of state.

We can start observing that much of the actual basic policy definition/configuration information already pre-exist on the target BNG when subscriber sessions are moved.

In the discussion of the management of the transient portion of the state, and for the subsequent evaluation of performance requirements, we will focus mostly on residential customers, as they represent the vast majority of customers on the BNG and typically may tolerate less stringent requirements than business customers. Moreover, for many of the use-cases introduced before, some temporary limited impact on the session may be in any case a better compromise than a session disconnect. The latter would in fact imply a worse quality of experience from the customer point of view, especially if we consider that many concurrent attempts to re-establish a new session will create a bottleneck on the system, particularly on the authentication, resulting in significant delays to reconnect.

For the dynamic component of the state, the following considerations may apply to policer's counters, keepalives, and accounting records. The current values of the counters used by the policer's should in principle be maintained and therefore moved along with the session packets. However, should this transfer of policer's state not be performed but the bandwidth configuration parameters maintained the impact would be limited to a transient inaccuracy in the bandwidth enforcement, which could be acceptable, especially if the inaccuracy period is limited.

Similarly, the state associated with the management of keepalives and related counters could also tolerate some transient inaccuracy, without disrupting the session continuity. For instance PPP keepalives (i.e. Echo Request/Reply messages) are sent periodically every 10 s, and the missing of 5 consecutive reply packets determines the closing of the session, assuming the remote peer is down. Therefore, provided that the time consumed by migration of a subscriber session does not exceed the above condition, the session should be able to continue on the destination BNG, without the need to manage the keepalive events during the stateful switchover phase.

The accounting records stored in the BNG also needs to be moved to the destination BNG. This is of course more critical information to preserve in the case of time based billing profiles (albeit less common in favour of flat-rate profiles nowadays); in case of volume quota based profiles, the loss of some accounting information would cause some inaccuracy in determining the crossing of the threshold. For all other profiles, typically the accounting records have mostly only statistical relevance.

Of course, it would be also possible, in principle, to support a complete storing and migration of the above mentioned state information, e.g. PPP keepalives and counters could be continuously tracked and stored/replicated for the session migration purposes, but the actual cost of such a full state management would be high in front of the lack of any additional benefit from the end customer point of view. That option would therefore represent a rather academic speculation and would not constitute a realistic choice from an actual service provider perspective.

Looking at different possible strategies for migrating sessions to a different BNG instance, one could consider in principle the option of starting to create only the new sessions in the target BNG, while waiting for active sessions on the origin BNG to spontaneously terminate. To be able to evaluate the actual applicability and effectiveness of such a strategy, it is worthwhile to have a look at how long sessions stay alive in a typical production network environment. Looking at the statistical measured data, it comes out that the typical duration of subscriber sessions is about 300 min, when averaged on the overall subscriber base. This value is also the average lifetime of typical broadband subscribers (ADSL, flat rate), while it almost doubles (500-600 min) for fibre subscribers (FTTC, FTTH) and reaches about 1500 min for business customers. The above data clearly show that a natural strategy for migration, based on redirection of the new session setups to a new machine, is not viable as it cannot satisfy the requirements or be efficient enough for any of the use-cases addressed (see Figure 5-2). Just considering the average session duration, it would take hours to migrate the workload, even more so considering the possible amount of long-living sessions.

When trying to identify performance requirements for the BNG stateful session migration, one can say that a limited transient packet loss can be mostly tolerated, especially when residential customers are concerned, to the extent this can be managed by the transport/application protocols and has not impact on the overall quality of experience. Therefore, loss-free migration seems not to be a strong requirement in general. Similar considerations can also be applied for the packet out-of-sequence performance; some limited impairment on packet ordering can therefore be tolerated, as far as it can be managed by the transport/application protocols without disrupting the service.

To give some example of more quantitative values for possible performance objectives, we can for instance refer to the study published by the MEF (Metro Ethernet Forum) for a carrier Ethernet service [MEF23.1]. The performance requirements reported in the above document are application specific and referred to the end-to-end network configuration, therefore allocation or budgeting of the overall objective is generally required.

Moreover, it is important to notice that the above performance objectives, for the different classes of service, are mostly referred to the measurement over a long time interval, typically in the order of a month. It is therefore quite difficult to actually specify a target for a short period of time. However, a perceivable albeit limited impact on some services, seems anyway more tolerable than a total session disconnect.

As an example, according to the above MEF specification, that in turn sometime refers to other sources like e.g. the ITU-T Recommendations G.1010 and Y.1541, possible objective values for VoIP and Interactive Video services are: < 400 ms for the one-way delay, 1-3 10^{-2} for the packet loss ratio, and < 40 ms for the packet jitter (IFDV); for Standard/High Definition Video the target values are: < 200 ms for the one-way delay, 1 10^{-3} for the packet loss ratio, and < 40 ms for the packet jitter (IFDV); for the Internet Streaming Audio/Video the target values are: < 10 s for the one-way delay, < 1% for the packet loss ratio, and < 1,5 s for the packet jitter (IFDV); for Interactive Transaction data the target values are: < 250 ms for the one-way delay, 1 10^{-3} for the packet loss ratio (assuming TCP loss recovery); finally, Best Effort services targets looks to be only specified in terms of Web browsing response time, that ranges from < 2 s (preferred) to < 4 s (acceptable).

Finally, subscriber sessions that are in the creation or termination phase during the migration can be quite easily managed. Session setup may typically fail during a BNG switchover, however in this case the subscriber will simply reattempt to setup the session on the new BNG. Moreover a proactive management strategy could, when possible, start redirecting all new session setup to the new BNG beforehand, when a session migration is planned. Session termination also does not seem to pose any particular additional requirement or criticality on the session migration mechanisms.

5.2 VNF Scaling and management of internal state

Various options for scaling individual NFs and whole NF-FGs were discussed in [D3.1] together with which events might trigger the scaling actions. Different scaling alternatives discussed in D3.1 were for example:

- Scaling up an individual NF by dynamically giving it more resources, or migrating to an execution environment with more resources.
- Scaling out by adding new resources and dividing traffic load depending on traffic type, where some traffic can be divided based on e.g. Layer 2 or 3 addresses while other cases requires dedicated load balancers able to interpret higher layer protocols

Triggers for scaling and migration that were considered in D3.1 may come from higher layers, lower layers or from parts of the VNF control systems:

- From higher layers a trigger can be e.g. a client or higher layer orchestrator requesting more resources.
- From lower layers migration may be triggered by topology and resource changes, such as new resources being introduced, resources taken down for administrative purposes, or failures.
- Scaling and migration may also be triggered by auto-scaling mechanisms in e.g. the VNF Control application due to increased traffic load.

Here we go deeper into how to manage the internal state within the VNFs during these events, in order to ensure appropriate synchronization between the VNFs and the traffic they are processing. Without proper state management services may stop functioning completely during scale-out/in events or in other ways degrade due to packet losses, increased latency and jitter, or simply being misconfigured. The internal VNF state can be managed by different components of the UNIFY architecture with different models elaborated in the amendment to D2.1, [D2.1a].

Here we focus on the challenge of coordinated network and VNF state management during a scale-in or -out event. As others have pointed out these two types of states have to be synchronized to avoid service degradation. For certain VNFs, e.g. stateless NATs, only the configuration state has to be taken into account, e.g. by configuring appropriate address mapping rules before directing user traffic to the new instance. In a stateful NAT the transient state created by the traffic itself has to be transferred before user traffic reaches the VNF to avoid that e.g. existing connections over the NAT are disconnected. As we saw in the previous section analysing the state and state requirements of a BNG, requirements on the consistency of state or accurate transfer of state is not always high. However, other VNFs may be sensitive to packet loss or reordering during the transfer process. As shown in [Gember-Jacobson2014], packet reordering and loss in an IDS VNF may trigger false positives or fail to trigger on real positives.

While there are several methods for dealing with network state or VNF state separately [Gember-Jacobson2013, Dilip2008, Qazi2013], there are only a few that handle them in a coordinated manner [Gember-Jacobson2014, Rajagopalan2013a, Rajagopalan2013b]. Here our focus was on improving OpenNF as it is the most feature rich of the methods and is able to provide loss-free and order preserving state transfers. Our improvements reduce the amount of messages over the control plane by distributing the state transfer mechanism. The next section gives a brief introduction to OpenNF works, for more details see [Gember-Jacobson2014].

5.2.1 OpenNF architecture

The OpenNF architecture consists of two main parts, 1) a shared library that is linked with the VNF application on the data plane and 2) a control application running on a controller. The shared library provides an API with methods for exporting and importing different types of state from a VNF instance and to enable generation of various events (Southbound API in Figure 5-3). The control application runs on an SDN controller and is responsible for coordinating the transfer of both network and VNF state using the OpenFlow protocol and OpenNF protocol respectively, in Figure 5-3 the Control Application can be seen interfacing with the NF State Manager and Flow Manager in order to do this. In OpenNF all VNF state is associated with the flow(s) that updates a particular chunk of state, either as a one-to-one mapping for an individual flow, a group of flows or for all flows, these could e.g. be a packet counter for an individual IP address, for an IP subnet, and for all IP packets respectively. Grouping the state into these categories and associating the state with the flow(s) allows OpenNF to easily export the appropriate state when a certain set of traffic flows are to be moved to another instance.

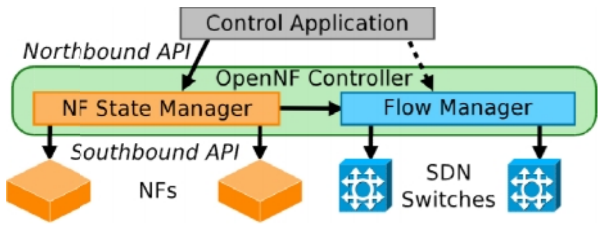


Figure 5-3: OpenNF architecture, taken from [Gember-Jacobson2014]

5.2.1.1 Southbound API to data plane VNFs

The southbound API is implemented using JSON over TCP, relevant commands are shortly summarized below. Here SrcVNF refers to the source of the state being transferred and DstVNF where it should be placed.

State export/import To export state the command *getPerFlow(filter)* is sent to SrcVNF, the *filter* defines which flow(s) the command is referring to, and by extension which state should be exported. State chunks are returned to the controller in *statePerFlow* messages. To import state the message *putPerFlow(map<flowid, chunk>)* is sent to DstVNF, containing a map of flow identifiers and their state chunks. Similar commands exist for handling state associated with multiple or all flows, and to delete state.

Events are enabled and disabled by the controller to handle data traffic during the VNF state transfer. At the SrcVNF *enableEvents(filter, drop)* is used to encapsulate and redirect packets to the controller without further processing. At DstVNF *enableEvents(filter, buffer)* is used to redirect and buffer packets.

5.2.1.2 Controller functionality and API

The controller in turn uses the data plane API to provide three different operations that control applications can utilize to manage VNF and network state, *Move(src, dst, filter, scope, prop)*, *Copy(src, dst, filter, scope)*, and *Share(instances, filter, scope, consistency)*.

The *filter* parameter defines which network flows (and which state) the operation refers to, *prop* is used to select guarantees. The *Move* operation transfers both VNF state and traffic to a new instance whereas the *Copy* operation only transfers VNF state (e.g. for periodic backups or loose synchronization). The *Share* operation provides strict consistency update for VNF state that has to be strictly synchronized between instances. While the use-cases for both *Copy* and *Share* are interesting here we focus on the *Move* operation. The three guarantees provided by *Move* are:

No guarantees (NG) All packets arriving during the state transfer are processed by the SrcVNF, state may be unsynchronized after the operation. The message sequence for this operation is depicted in Figure 5-4.

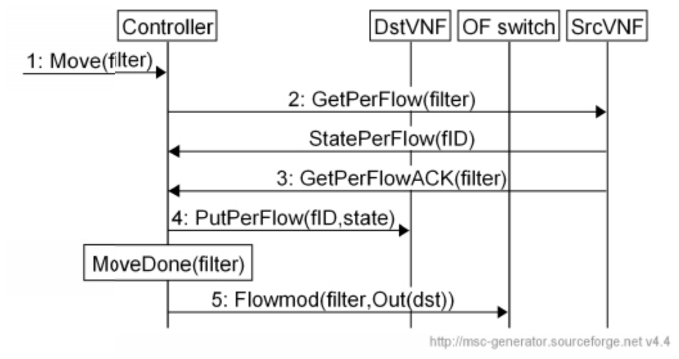


Figure 5-4: OpenNF NG Move operation

Loss-free (LF) Packets belonging to flow(s) currently being moved are redirected using *events(filter,drop)* to the controller where they buffered until the VNF state transfer finishes. When completed buffered packets are sent to DstVNF via the switch using Packet-Out. Finally the switch is updated to send traffic to DstVNF.

Order preserving (OP) Extends LF with packet stream synchronization and a two phase forwarding update that ensures that all redirected packets are processed at DstVNF before packets arriving from the switch. When VNF state transfer is completed the *event(filter,buffer)* is enabled on the DstVNF, causing it to buffer packets coming directly from the switch and also redirect them to the controller. Redirected packets (from SrcVNF) buffered at the controller are sent via switch but this time with a special flag, causing them to bypass the buffering at DstVNF. Phase one update duplicates traffic to both controller and DstVNF. When the first packet from the switch arrives at the controller, phase two update is triggered to only send packets to DstVNF. Packets synchronization is achieved at the controller by trying to match the packets arriving directly from the switch with the ones arriving from the DstVNF, When the packet stream synchronization has been observed the controller disables *event(filter,buffer)* and triggers the release of buffered packets at the DstVNF.

The LF and OP versions of the *Move* operation achieves state transfer without packet loss or re-ordered packets which e.g. Split/Merge [Rajagopalan2013a] could not. However, these guarantees come at a cost in packet latency and control plane overhead as data plane packets are redirected and buffered at the controller.

5.2.1.3 Optimizations

OpenNF implements three optimizations of the basic *Move* operation, these optimizations are:

Parallelize (PZ) Immediately send received state to DstVNF without waiting for GetPerFlow to finish.

Late Locking (LL) Packets arriving at SrcVNF are redirected to the controller on a per-state basis; redirection only happens if the associated state has been sent to the controller

Early release (ER) Packet redirection from the controller to DstVNF is performed on a per-connection basis. Instead of waiting for the full state transfer to complete, packets are sent to the DstVNF if corresponding state has been sent and acknowledged by DstVNF[6]. LF *Move* with all optimizations can be seen in Figure 5-5.

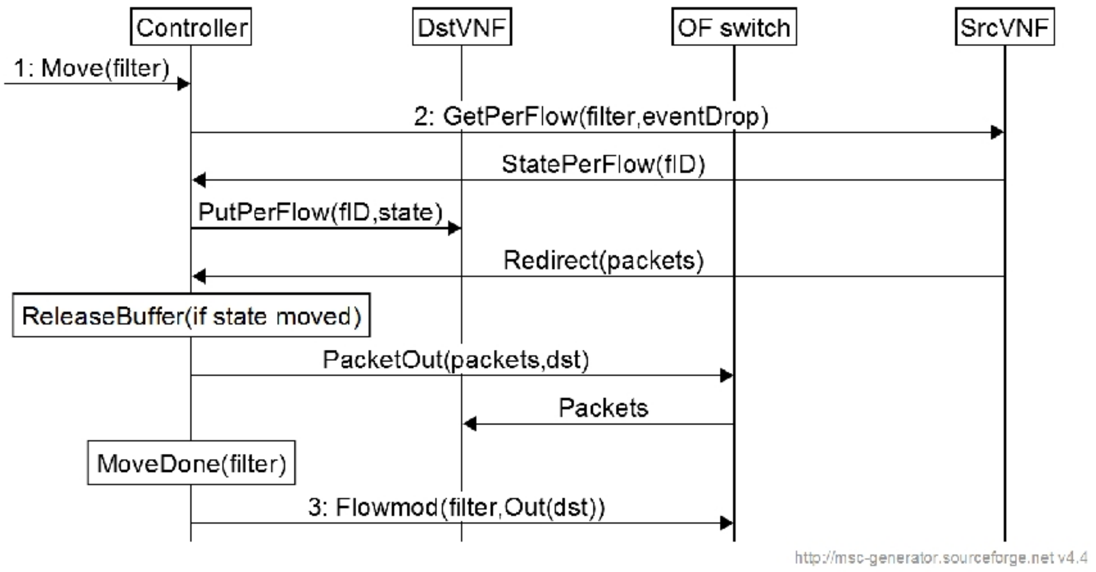


Figure 5-5: OpenNF LF with PZLLER

[6] Both LL and ER optimizations will change the order of packets *between* flows, while maintaining the order within flows. This reduces the order preserving guarantee to be order preserving only within flows, which may be significant for some VNFs.

5.2.1.4 Bottlenecks in OpenNF

Our main concern with the OpenNF protocol is using the controller for VNF state transfer and in particular for redirection of data packets during both LF and OP *Move*. One of the goals we have when transferring state is to affect traffic passing through the VNFs as little as possible e.g. the LF *Move* reduces the impact on traffic by not causing packet loss. However, when buffering data plane packets in the LF and OP cases we will induce additional latency on the flows that are being moved. While we cannot avoid buffering, we can reduce the amount of traffic buffered by reducing the total time it takes to transfer state by not sending messages via the controller.

Transporting data plane traffic over the control network (as events and as OpenFlow Packet-In/Out) risks overloading the control network which typically has a lower bandwidth and higher latency than the data plane links. Data plane flows can have a higher bandwidth than what is supported by the control network, making it impossible to redirect these packets to the controller without loss.

5.2.2 Distributed state transfer (DiST)

To address the problems discussed in the previous section we extended OpenNF with Distributed State Transfer (DiST) to avoid passing VNF state and data packets through the controller, this also removes the need for packet buffering in the controller. DiST required the addition of new commands in the data plane API as well as changes to the messages exchange between the controller and the VNFs. The API has been extended to include a destination address at appropriate places, e.g. for *enableEvents*, and a new command was added to transfer VNF state directly between VNFs*, MovePerFlow(filter,dstIP).* How the *Move* operation message exchanges have been extended in DiST for the various guarantees is described below:

DiST No guarantees The initial *GetPerFlow* is replaced with a *MovePerFlow* message to SrcVNF containing the IP of DstVNF. SrcVNF establishes a connection to DstVNF and uses *PutPerFlow* to transfer state. When all state for the filter has been transferred, SrcVNF acknowledges the *MovePerFlow* to the controller which updates forwarding in the switch.

DiST Loss-free with PZ The loss-free operation starts by sending an *enableEvent(filter,drop,DstVNF)* message to SrcVNF, which initiates a TCP connection with DstVNF and creates a filter for redirecting packets to DstVNF where they are buffered. Once *enableEvent* is acknowledged the controller sends a *MovePerFlow* message to SrcVNF and state transfer starts. Once all affected state has been transferred SrcVNF sends a command to start processing the buffered packets. SrcVNF finally acknowledges the *MovePerFlow* command to the controller, which updates forwarding in the switch.

DiST Loss-free with PZLLER Each state for which transfer has started is marked and arriving packets associated with marked state are redirected to DstVNF. Packets associated with state for which transfer has not yet started are processed directly by SrcVNF. DstVNF in turn maintains a hash table that buffers packets per-flow. When all state for a flow has been transferred DstVNF starts processing packets from that flows bucket in the hash table. If state for a flow has been transferred and there are no packets in the buffer, redirected packets are not buffered but processed immediately. A message sequence chart for this flavour of *Move* is depicted in Figure 5-6.

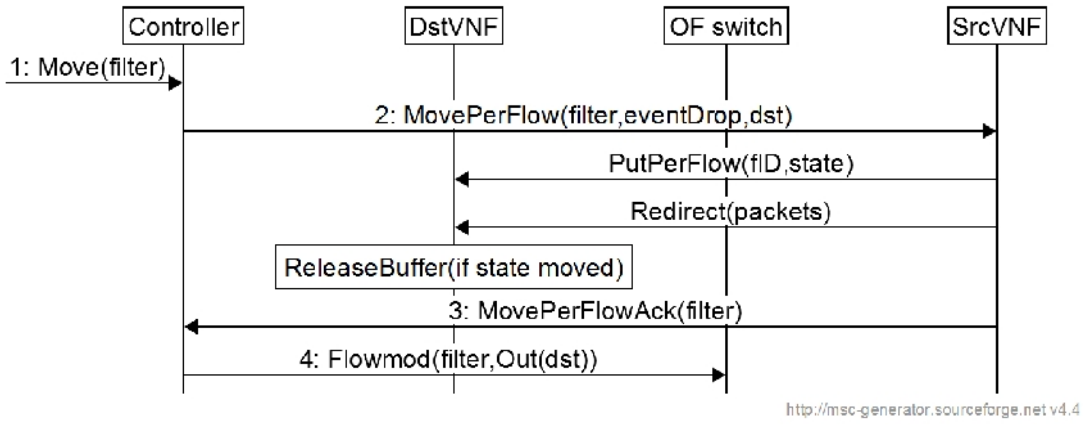

Controller
DstVNF
OF switch
SrcVNF
1: Move(filter)
2: MovePerFlow(filter,eventDrop,dst)
PutPerFlow(fID,state)
Redirect(packets)
ReleaseBuffer(if state moved)
3: MovePerFlowAck(filter)
4: Flowmod(filter,Out(dst))
http://msc-generator.sourceforge.net v4.4

Figure 5-6: DiST LF Move with PZLLER

DiST Order-preserving with PZLLER Extends the LF procedure with a two-phase forwarding update to ensure that redirected packets from SrcVNF are processed before packets forwarded from the switch. Synchronizing the two packet streams (one redirected from SrcVNF and the other forwarded from switch) is done using an ``InBand Control packet'' (IBCPkt). The IBCPkt must be crafted differently for each *filter* and also depends on how the VNF processes packets; the IBCPkt must match the filter definition in both VNFs but also be distinguishable from normal data packets[7]. The two-phase update starts by updating the switch forwarding rule for *filter* to duplicate traffic to both Src- and DstVNF, followed by a Packet-Out message sending the IBCPkt to both VNFs at once[8]. Finally the controller updated the switch forwarding rule to send traffic belonging to *filter* to DstVNF only.

DstVNF will now receive two IBCPkts, one directly from the switch and the other redirected by SrcVNF. If the IBCPkt arrives first from the switch (Event1) DstVNF starts buffering packets from the switch[9] while still processing redirected packets from SrcVNF, until the second IBCPkt arrives. At that point processing of redirected packets stops and processing of buffered packets from the switch starts. If an IBCPkt arrives from SrcVNF before the IBCPkt from the switch (Event2) processing of redirected packets from SrcVNF is stopped. Once the second IBCPkt arrives processing of packets from the switch is started. A message sequence chart for order-preserving *Move* is depicted in Figure 5-7.

[7] A similar problem exists in many OAM protocols in which probe packets must be fate-sharing with the traffic flow they measure but still be distinguishable from user traffic. One solution is to inform the VNFs of the content of the IBCPkt ahead of the Move operation so that they know what to look for.
[8] Assumes that the IBCPkt is inserted at the same position in both of the packet streams.
[9] Packets matching *filter* from the switch before IBCPkt are dropped, these packets will also be redirected from SrcVNF.

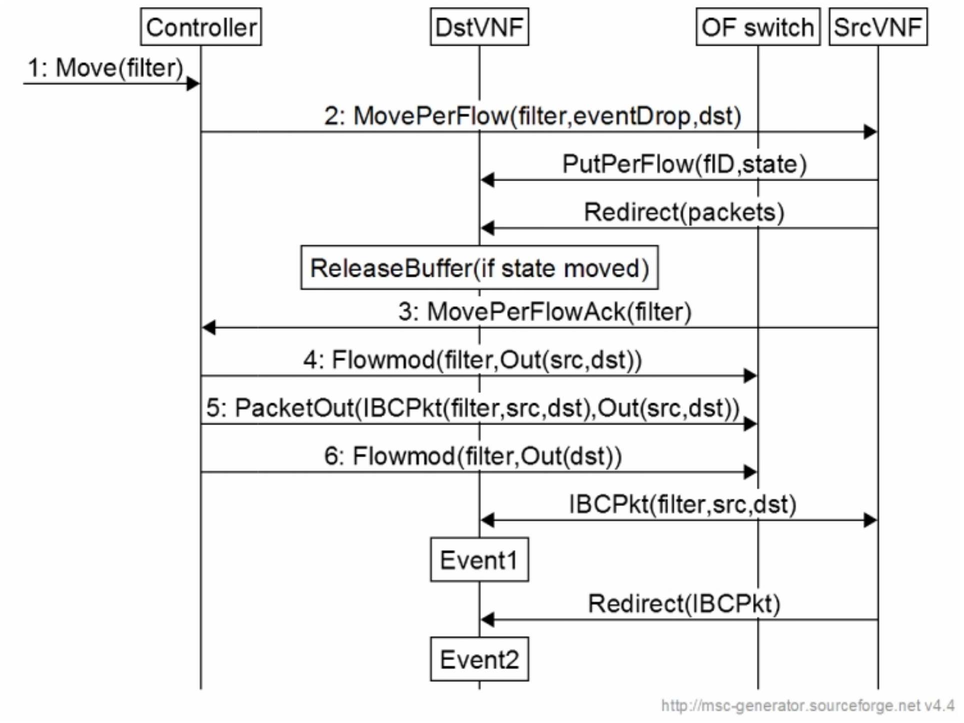

Controller
DstVNF
OF switch
SrcVNF
1: Move(filter)
2: MovePerFlow(filter,eventDrop,dst)
PutPerFlow(fID,state)
Redirect(packets)
ReleaseBuffer(if state moved)
3: MovePerFlowAck(filter)
4: Flowmod(filter,Out(src,dst))
5: PacketOut(IBCPkt(filter,src,dst),Out(src,dst))
6: Flowmod(filter,Out(dst))
IBCPkt(filter,src,dst)
Event1
Redirect(IBCPkt)
Event2
http://msc-generator.sourceforge.net v4.4

Figure 5-7: DiST OP Move with PZLLER

5.2.3 Evaluation of DiST vs. original OpenNF

We evaluate DiST in the testbed configuration shown in Figure 5-8. Based on an assumption of roughly one order of magnitude performance difference between control and data plane networks, the data plane links are limited to 100 MBit/s bandwidth with a 1 ms one-way delay while control plane links are limited to 10 MBit/s with 10 ms one-way delay.

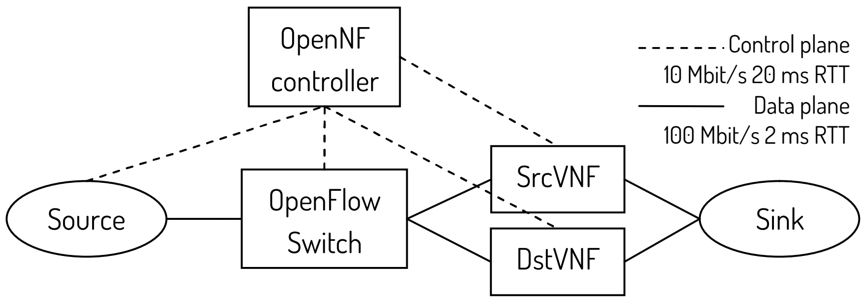


Figure 5-8: Testbed setup

Looking closer at the assumption of one order of magnitude difference for these two performance metrics, for bandwidth is arguably too low as measurements reported in for example [DevoFlow] put the difference as high as four orders of magnitude. In those measurements their switch line-card is capable of 300 GBit/s whereas the control plane is limited to a mere 17 MBit/s. As for the latency difference, one order of magnitude may appear to be too high instead. However, as reported in the same paper the line-card is capable of forwarding a packet within 5 μs while the round-trip-time between line-card and local control plane CPU is 500 μs. How that difference translates into a choice of latency is our case is difficult to say, as many other things also affect the difference in latency between control and data plane, including e.g. the topology of both control and data plane networks, the distance to the controller from the VNF as well as the distance between VNFs. While it is likely that the latency on the direct path between two VNFs is lower than going via a third party, the assumption of a 10 times more latency is somewhat arbitrary. For this reason we also evaluated the mechanisms with no performance difference between control and data plane which is not realistic but provides a baseline for comparison.

As VNF we use the PRADS implementation from the authors of [Gember-Jacobson2014]. PRADS passively monitors network traffic to gather information about the hosts and services it can see. The state transferred when moving flows for PRADS is thus the information found about a host with an address in the flow definition, e.g. what services it is using and whom it is communicating with.

As only the NG and LF DiST extensions to *Move* are currently implemented, we compare these with the original OpenNF versions by replaying a live network traffic trace using tcpreplay[TCPr] and after 20 seconds we initiate a *Move*. We replayed the traffic at 100, 500, 1000, 2500, and finally 5000 pps. During normal operation, i.e. no state transfer is ongoing, the VNF implementation is able to handle around 15000pps. However during a *Move;* at 2500 pps the OpenNF SrcVNF starts dropping incoming packets, the same happens in DiST at 5000 pps[10]. For a fair comparison we therefore focus on the results obtained below 2500 pps, i.e. the ones obtained at 1000 pps.

5.2.3.1 Controller load

In OpenNF *Move* the total amount of messages *is the same as the amount of messages passing the controller*, approximately $3n + 2r + c$ where n is the number of state chunks, r the number of redirected packets, and c the number of control messages (which is different for depending on the optimizations, between 3 and 5). With the DiST extension the total amount of messages is approximately $2n + r + c$; the number is reduced since state and packet do not need to go via the controller. More importantly for the controller load, in DiST the only messages affecting the controller is c as all other messages are sent directly between the VNFs. While the cost of the DiST *Move* still scales with the amount of state to transfer and the number of redirected packets, that cost is placed at the VNFs, keeping the controller load constant for each *Move* operation.

5.2.3.2 Operation Time

Some of the data we collected is shown in Table 5-1, these are results at 1000 pps.

[10] Both are likely due to locking of shared data structures between the packet processing and state transferring threads.

Table 5-1: DiST vs OpenNF performance data gathered at 1000 pps, with 95% CI

Config	# States	# Redirected	MoveTime(s)	RedirTime(s)	Ser%	Deser%
OpenNF LF PZ	2186±1	6678±179	7.44±0.21	8.11±0.17	2.63	1.95
DiST LF PZ	2184±2	489±39	0.51±0.06	0.82±0.04	40.22	33.01
OpenNF LF PZLLER	2186±1	17±5	2.25±0.04	1.51±0.05	7.99	6.67
DiST LF PZLLER	2204±2	6±1	0.41±0.04	0.41±0.21	48.63	40.81
(OpenNF* LF PZLLER)	2187±1	6±2	0.78±0.02	0.96±0.17	26.83	22.07

MoveTime is the time from start of the *Move* operation on the controller until the switch forwarding update. MoveTime for the original OpenNF solution is about 3 to 6 times higher than DiST for the PZLLER case, this is mostly explained by the latency differences on the control vs. data plane, but the reduced number of messages in DiST contributes as well. To separate the contribution from latency from the number of messages we ran OpenNF in an unrealistic case where control and data plane have the same performance, shown as OpenNF*. Comparing the MoveTime for DiST and OpenNF* it is clear that the reduced number of message makes DiST about twice as fast even in this case. The importance of the LL and ER optimizations is also obvious here; they are very effective at reducing the number of redirected packets. In DiST these optimizations reduce the MoveTime by about 20% at 1000 pps, whereas for OpenNF the reduction is about 70%. However, the effectiveness of these optimizations depends on the composition of the incoming traffic.

RedirTime is the time from initiating state transfer at the SrcVNF until the last packet is redirected. We measured RedirTime in order to see how long time packets are still being redirected after the VNF state has been moved. This can be significant if we e.g. shut down SrcVNF once we believe *Move* is completed. As can be seen, in some cases RedirTime is longer than MoveTime indicating that RedirTime should be considered when determining if the *Move* has completed or not. We also measured *Ser* and *Deser* which is the percentage of MoveTime spent (de-)serializing data in the VNF during state export and import respectively. As can be seen a large part of the MoveTime in DiST is spent on serializing, showing another bottleneck to be removed.

MoveTime in LF mode with PZLLER versus number of state chunks transferred for the different pps values is shown in Figure 5-9 on the left y-axis, illustrated with bars. As can be seen, DiST is always faster than OpenNF. On the right y-axis is the ratio between OpenNF, OpenNF* and DiST MoveTime, plotted with dotted lines. While it appears as OpenNF* is getting closer to DiST in performance as the number of states and pps increases, it should be noted that OpenNF* is dropping packets at these speeds.

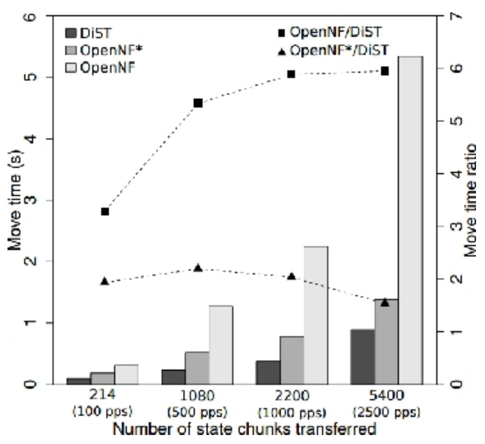


Figure 5-9: MoveTime for different amount of state (left), ratio between DiST and OpenNF (right). Each state chunk averages around 750 bytes.

5.2.3.3 Traffic Pattern at Sink

Figure 5-10 depicts the traffic pattern of the processed packets at Sink for LF *Move* with PZLLER at 1000 pps. The dotted boxes in the figure indicate when the *Move* started and finished. With PZLLER the SrcVNF should continue to process packets during the operation and hence the packet rate should not be affected, which is observed for DiST. In original OpenNF the traffic rate drops to zero in the beginning of *Move*, likely due to shared resources being locked by the state transfer thread when the *Move* is initiated. This in turn causes incoming packets to be queued; the release of the lock explains the spike in the traffic pattern as incoming packets are being processed. Moreover, these packets updated the state in SrcVNF instead of DstVNF, leading to unsynchronized state. In DiST this works correctly as can be seen from the number of states transferred. OpenNF transfers the same number of states regardless of optimization while DiST PZLLER also transfers states created at SrcVNF during the operation.

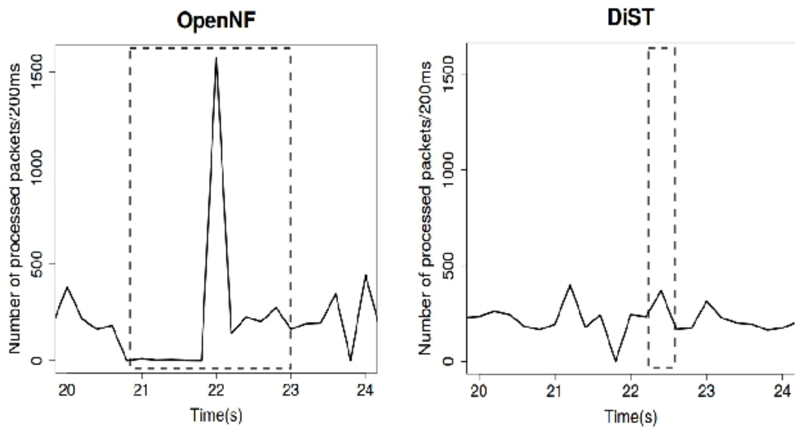


Figure 5-10: Traffic pattern at Sink during the move (indicated by the boxes) at 1000 pps. Identical traffic traces used in both cases, with the same timing.

5.2.4 Conclusions on of evaluation and potential extensions

Distributing the Move operation heavily reduces the amount of messages exchanged during the operation, halving the amount of messages per redirected packet and removing a third of the messages per state chunk transferred. Additionally, only a few control messages, instead of all messages traverse the control network to load the controller, increasing the scalability of the system as a whole. In a more realistic scenario where performance in the control plane is lower than that of the data plane these changes show a substantial performance gain, being roughly 3 times faster at 100 pps, 5 times at 1000 pps, and 6 times at 2500 pps. These performance values are however depending on the assumption of 10 times higher latency on the control plane, how accurate this assumption is depends on many factors. There are however good reasons to be suspicious of control plane performance, e.g. in some switches Barrier messages can cause up to 400 ms control plane latency [Kuzinar2014].

Another issue that is clear is the time required for (de-)serialization, which consuming a large part of the total MoveTime. We believe the largest contribution to this time comes from the generic C structure serialization library used to extract the state. It allows C structures with pointers to be converted to strings with hardware independent references, avoiding the need to manually write encodings for each state structure. However, the performance of this approach seems to be a limiting factor. Additionally the use of JSON also contributes, as shown in [JSONComp] other techniques are much faster and produce less data to transfer. Moving to e.g. Google Protocol Buffers for both the protocol and state serialization could be a large improvement [Protobuf].

The observation of packet losses at 5000 pps (about 20 MBit/s) during a Move with DiST surprised us as the PRADS VNF is capable of at least 15000 pps (about 60 MBit/s) during normal execution. After trying to debug the code we believe the reason for these losses is lock contention between the packet processing thread and the state transferring thread, combined with the extra cost of redirecting packets. One solution to this issue could be to change the order of the steps in *Move* and start with forwarding redirection at the switch and buffering at DstVNF, and then transfer VNF state. This would reduce contention as the packets associated with the state we are transferring would arrive at DstVNF rather than at SrcVNF. Another benefit is that SrcVNF does not have to redirect packets, further reducing contention. An order preserving version of this solution is illustrated in Figure 5-11.

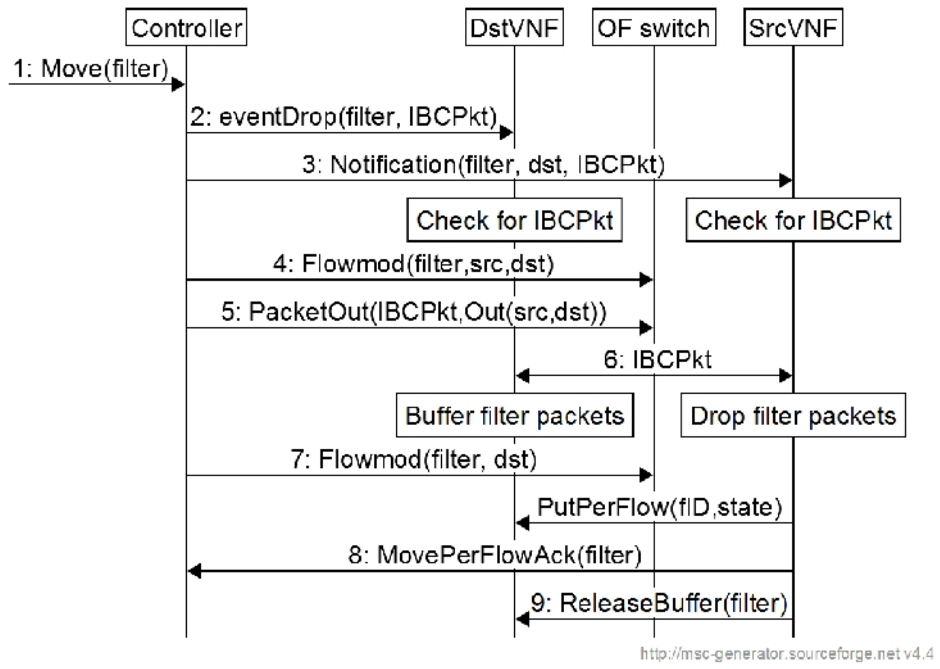


Figure 5-11: DiST OP Move, alternate order with stream synchronization and buffering before state transfer

One negative effect of redirecting traffic before moving state is that we cannot implement the Late Locking optimization; packets that with LL would be processed at SrcVNF will instead be buffered at DstVNF. The Early Release optimization can however be implemented even in this scenario. Without Late Locking in this alternative solution we risk buffering flows at DstVNF for a long time while waiting for their VNF state to be transferred, inducing long latencies for those flows. Individually transferring smaller flows instead of grouping them into a single large *Move* could reduce the impact of this (e.g. performing 255 *Move* operations on /16 IP subnets instead one *Move* operation on a /8 IP subnet). However, many *Move* operations would consume more flow rule entries in the switch and likely increase the total time, if this is a viable strategy remains to be seen.

5.3 Service resiliency

Traditional telecom services largely rely on specialized hardware devices which have been specially engineered to deliver high performance and high reliability. However, when virtualizing telecom services into software-based components (NFVs), additional mechanisms might be needed in order to cope with failures caused by for example software bugs, server overload, memory leaks, etc. These phenomena might cause failing NFs, links, or even entire NF-FG instantiations. Service resiliency refers to the ability of a service to withstand these changes/failures in its environment in order to remain operable over the considered time period (reliability). Often this involves a range of corrective actions which are usually referred as recovery mechanisms. Maintaining state of involved NFs is crucial in providing service resiliency. Depending on involved resiliency mechanisms, state may need to be migrated using techniques which are very similar to the ones detailed in previous sections. Most resiliency mechanisms somehow rely on the adequate use of backup components. Network resiliency [Vasseur2004] is an established research and engineering domain involving a wide range of protection and restoration mechanisms at node, link, path segment or path-level on any network layer up to layer 3. Resiliency/Reliability for server-based component also has been researched and documented extensively. For example, (application-independent) server pooling mechanisms, e.g. [Lei2008] involving some form of duplication on the server side enabling one instance of a server pool to stand in for the other when a failure has been detected. However, virtualized Network Functions and Service Graphs of NFs have particular characteristics and requirements which cannot be easily accommodated by simply applying existing network or server reliability mechanisms:

- NFs are different from usual application servers, in the fact that they might focus on low-level and high-speed packet operations (e.g., header-based lookups, header-rewriting, deep packet inspection) and related packet-level (session) state
- NFs are different from usual application servers, in the fact that they might not assume reachability/addressability using traditional Layer 2 or Layer 3 addressing techniques, for example Ethernet MAC- or IP address, because one NF might depend on raw packet information from another NF

In summary, because network packets, rather than application-level state are the focus point in NFV-based services, recovery mechanisms must be very efficient in order to avoid significant packet loss, retransmission or buffering when components fail. The following subsections present an overview of resiliency mechanisms for (parts of) NFV-based services. The section is concluded with a concrete example of applying NF protection.

5.3.1 Protection vs. restoration

Protection mechanisms are mechanisms which pro-actively reserve backup resources which might be activated upon failure detection, while restoration mechanisms only trigger the search for backup resources upon the moment of failure detection. Many NFs are stateful, referring to either: i) state which is a direct result from control/configuration instructions (e.g., firewalling rules requested by an operator), or ii) state which is a result of the NF processing data packets (transient session state). Protection mechanisms involve mechanisms to also replicate one or both forms of state pro-actively, before the failure actually occurs and is detected. Restoration mechanisms risk that none or only part of the state might be recovered in case a failure occurs.

5.3.2 Recovery scope

The recovery scope refers to the set of Service Graph components or instances which might require recovery. We envision the following possible scopes:

- <u>NF (instance) recovery</u>: recovery of an individual NF instance

- Sub-NF-FG (instance) recovery: recovery of a part of an (or the entire) instantiated NF-FG, usually a combination of interconnected NF instances. This is usually needed if a failure has impact on closely located parts of the network/graph.
- NF-FG (instance) slice recovery: recovery of a set of correlated NFs, but not necessarily (directly) interconnected NFs.

5.3.3 Failure detection

Failures can occur in nodes and links of the instantiated NF-FG. Failures are detected through observation points, polling for example link, segment, path, NF instance activity. Observation points communicate events to monitoring points (as described in WP4 deliverable and milestone documents), and observation points interact with (Control) NFs, control or orchestration systems in order to handle the detected failure.

5.3.4 Recovery controlling component

Depending on protection or recovery setups, the notified component may trigger a restoration scheme, or activate (the signalling and/or provisioning of) the backup network or node instances. Depending on the control locality of the recovery process[11], any of these options might be preferable:

- Local recovery: when the recovery controlling components are close to the NF instance might be advantageous for minimizing the required time for handling the failover process. This might involve for example the activation of a backup forwarding entry (towards a backup NF instance) in a network element close to the failing NF instance
- Global recovery: when the recovery controlling components must interact with components which are not necessarily close to the failing instance in order to activate recovery. In case of global recovery, the full NF-FG might be required to be re-established and therefore, may involve reconfiguring multiple Network Functions and Network Elements. This might for example involve:
    - Interaction with the global or local orchestrator(s) to orchestrate additional/other resources
    - Interaction with network or compute controllers to re-provision and signal network or cloud resources. In an SDN setup this follows the (logically) centralized provisioning process, in other setups, this might involve the signalling of paths through distributed protocols such as RSVP(-TE).
    - Interaction with traffic steering mechanisms and components, for example enabling global-decision-based source routing encoded in the traffic steering packet headers.

[11] The control locality of the recovery process refers to the distance of the controlling/management entity to provisioned instances. Given the hierarchical control infrastructure, we consider the service layer OSS/BSS as the control entity with the highest possible distance to provisioned instance, succeeded by the global resource orchestrator, the local orchestrator(s) in case of recursive domains, the network/cloud control system, control NFs, network elements (switches), up to the level of the NF instance itself.

Note that the recovery control mechanism itself also must be provisioned. Especially in case of local recovery, backup instances, as well as backup entries in corresponding network elements must be pre-provisioned by the default top-down service programming process. When no recovery control mechanism is provisioned, failures might be unaddressed, or might be addressed through global restoration mechanisms.

5.3.5 State synchronization or migration component

Full service recovery ensures that not only backup (network, compute or storage) instances provide the same functionality, but also involve the same state of their primary counterparts.

State migration mechanisms described in the previous section might be re-used for NFV-service and NF protection mechanisms. Protection mechanisms may for example use periodic or triggered state migrations between primary and backup resources. Periodic state migration can occur very often, but potentially heavily impacting network (bandwidth) and processing requirements in order to keep state in sync as much as possible. On-demand/triggered state migrations might only be triggered when significant state changes are detected, reducing both migration resources as well as reducing the probability of outdated state on backup resources. These mechanisms might for example rely on the ASAP_COOKIE mechanism[12] of the Reliable Server Pooling protocols [RFC5351]. Although these mechanisms cover both control/management and data plane traffic-induced state replication, they might have a significant impact on required migration (network and computing) resources for regularly synchronizing state. An alternative mechanism might only replicate management/control-relate state by sniffing the corresponding channels and replicating actions on both primary and backup instances.

NFV-based service and NF restoration mechanisms face significant challenges for restoring state on backup resources because primary resources are not fully accessible anymore for migrating state. Potential future research might focus on mechanisms trying to recover parts of state of failing instances.

5.3.6 NF protection example

Let's consider a Service Graph with three NFs: NF1, NF2 and NF3, in which the SG description specifies high availability requirements for NF2. In order to fulfil the latter availability requirement, the Service Layer Adaption component might translate this into a protection setup for which NF2 has a primary instance NF2a and a backup instance NF2b. These are orchestrated by the underlying orchestrator, in addition to required monitoring and control NF functionality. Aliveness of NF2a and NF2b is monitored by MPCTRLNF (which both fulfils observation and monitoring functionality)[13]. This monitoring process might for example rely on active polling and acknowledging keep-alive messages from MPCTRLNF towards NF2a and NF2b. The default setup of the Network Element responsible for interconnecting NF2 with NF1 and NF3 is indicated in the mapped NF-FG of Figure 5-12. From the

[12] The ASAP_COOKIE mechanism is a RserPool mechanism enabling the periodic transmission of snapshots of a (server) state towards a user/client of the server pool. This enables the client to transfer state to a newly activated server once the primary server fails.

[13] Note that the ports on NF2a, NF2b and MPCTRLNF responsible for monitoring the aliveness of NF2a and NF2b are not depicted in the figure to improve readability.

moment MPCTRLNF is detecting a loss of signal of NF2a, it can trigger NE1 to reconfigure its switching tables such as to adapt the forwarding rules fr-a and fr-b to port 8 and 9 instead of to port 4 and 6 correspondingly. Note that MPCTRLNF is in this case both functioning as Control NF in addition to its role of monitoring point. The resulting steps are represented in the sequence diagram of Figure 5-13.

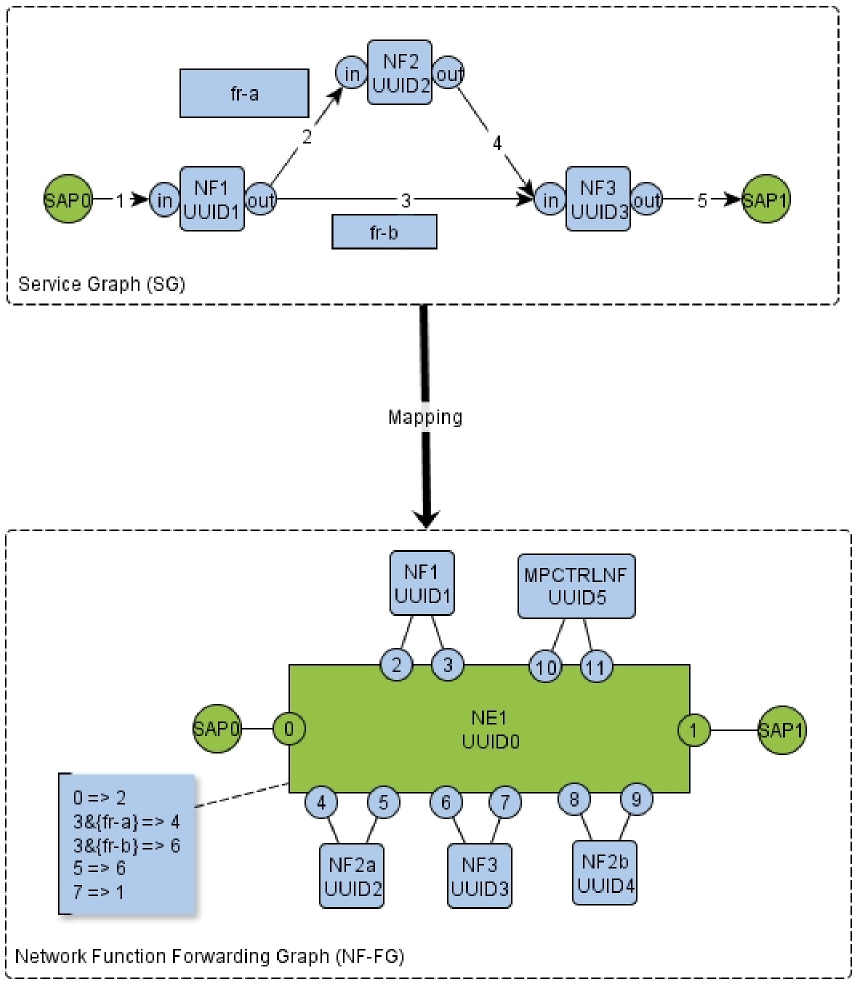


Figure 5-12: NF protection example

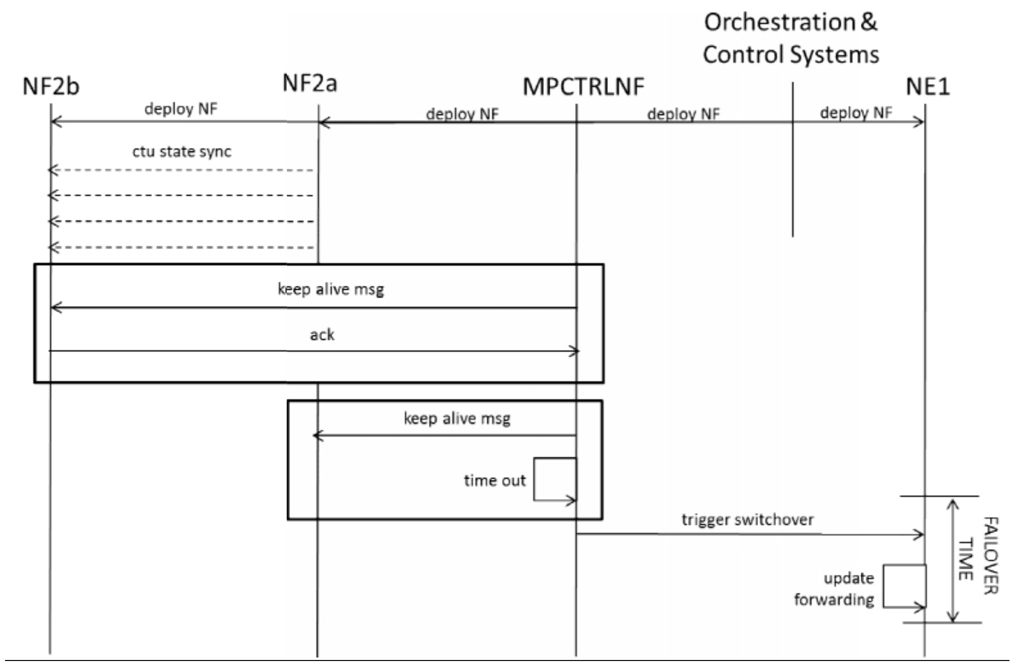


Figure 5-13: Sequence diagram for NF protection process

5.3.7 Conclusions

Service resiliency mechanisms target state maintenance under conditions when components of an NF-FG fail or degrade, for example caused by bugs in the software implementation of a VNF or due to hardware failures in the network or server infrastructure. Synchronization and transfer of state for resiliency purposes requires similar mechanisms and primitives as proposed for service scaling in section 5.2, allowing these systems to share functionality. However, state maintenance targeting service resiliency adds a particular context to *when* and for *which scope* state is migrated, what *triggers* the migration, and *which* component is in control of the process.

For the question of *when* state migration should be applied, it can be done either in a pro-active fashion by periodically or continuously synchronize updated state (for protection) or in a reactive fashion for restoration of failed components. As failed components may be unreachable due to the failure, the reactive approach may be mostly applicable in cases when a service is degrading but the internal state is still recoverable. This could be for example due to infrastructure failures such as failing hard drive or network interface on compute nodes, or a faulty forwarding element in the network. On the other hand, protection mechanisms provide a higher level of resiliency but come with a high cost in terms of both compute and network resources.

There is also a need to determine for which scope resiliency mechanisms should be activated e.g. for a single NF, a whole service, or a group of services. Here monitoring and troubleshooting tools developed in WP4 could provide assistance to detect the extent of the failure, together with aliveness mechanisms described in the protection example above. This largely determines *which component* is *controlling* the recovery process, e.g. a CtrlApp or the

Resource Controllers of the UNIFY platform. For example an infrastructure failure may cause alarms both in the infrastructure and service layer, but be solvable by a simple traffic redirection from a failing switch rather than migrating or triggering failover for all affected VNFs or services.

6 Traffic steering and forwarding state

The UNIFY project aims at defining a layered architectural framework for automated service chain orchestration and deployment. In this deliverable we have systematically gone through the UNIFY architecture in a top-down manner and starting from service programmability, the definition of Service Graphs and NF-FG in the Service Layer (section 2), through resource orchestration, service decomposition, elasticity and scalability in the Orchestration Layer (sections 3 and 4) and finally we have arrived to the Infrastructure Layer where we considered individual network functions including NF state (section 5). In this section we close our top-down journey through the whole architecture with a short study on scalable traffic steering or **data plane orchestration**.

Steering traffic efficiently in a flexible and controllable way between the NFs of a service is an indispensable part of an orchestration framework. While it is indispensable, the design of fast and scalable routing architectures is still a great challenge after decades of routing research. The advent of software defined networking and virtualization pushes the routing challenge even to extremity as the flexibility of SDN architectures allows deploying many existing routing approaches at the same time on the same network, moreover there is no obstacle to introduce hybrid routing technologies which can result in tailor-made solutions for specific network circumstances. The throughout analysis and development of traffic steering strategies which can fully exploit the features of SDN is way beyond the scope of this deliverable. Nevertheless the UNIFY project members agree that the importance of this topic deserves taking a first step at least, towards the characterization of fast and scalable SDN routing strategies which can be foundations of service chaining architectures in the future. In what follows we present the results of our first attempt in directions. Our contribution will be twofold. First, we present a 3-tiered architectural model which can incorporate many existing routing technologies and projects research directions towards finding the ultimate routing for SDN which we call Software Defined Routing (SDR). We argue that SDR in something that can effectively distribute routing state between the tiers of our model according to the resources and desired functions of the network. To our opinion, the distribution of state between tiers can be feasible way towards scalable routings. Secondly, we illustrate the benefits of state distribution through a simple numerical evaluation over well-known network topologies. Besides the power of our model to unify the existing routings under a single framework it may generate more questions than it answers. This is made somewhat intentionally as the goal of our study is to call the research community and the industry to invest time in investigating the algorithmic aspects of data plane orchestration, in order to supply a solid mathematical foundation and an optimization toolset to practitioners for building truly dependable software-defined networks and service chaining architectures like UNIFY.

In the UNIFY architecture, abstract forwarding information is stored in NF-FGs in our programmability framework. At the lower layers, this abstract information must be translated into different types of routing states that can be pushed directly to the network elements as different types of flow rules, tunnel information or packet header fields. This is the responsibility of Network controllers in the Infrastructure Layer, which has to manage the forwarding states at different elements in the network. From the operators' point of view, it is crucial to manage this behaviour as efficiently as possible and in an optimal way. Unlike UNIFY, the abstract forwarding information can be originated from different high-level policies of the operator other than service chaining. Thus in what follows we present an architectural model which gets the abstract forwarding information as input no matter from where is actually comes from and the only purpose is to map these abstract rules to forwarding state.

In the rest of this section we explicitly define the problem of data plane orchestration by using our 3-tiered model. After that the classification of existing and possible future routing strategies is presented based on the state distribution among the three tiers. Finally we formalize the problem of trading-off state between tiers and present some early results which illustrate that there are sweet spots in the trade-off space and promote further studies of hybrid routing strategies.

6.1 Theoretical approach to traffic steering

The grand vision of Software-Defined Networks (SDN) is to consolidate SDN-capable devices and legacy network infrastructure, stripped from the "ossified" distributed control plane, into a common virtualized data plane resource, operated under the supervision of a central software controller. Control plane programmability (via a standardized API) is expected to offer a greater extent of automation, dependability, flexibility, and better profitability to operators, to open up the control plane for rapid innovation, and to eliminate vendor-lock in. This great deal of generality, however, creates a new algorithmic challenge [Raghavan2012, Koponen2011, Kang2013, Ashwood2014, Ashwood2013, Kolias2014], broadly posed as follows: *If SDN can accommodate such a large variety of packet routing and forwarding paradigms, or essentially any combination thereof, then exactly which one is to choose for a particular set of operator goals and how to design the corresponding routing itself?* With regard to UNIFY the above general question can be translated as: *which is the best forwarding paradigm for hosting service chaining architectures?*

6.1.1 A framework for Software Defined Routing (SDR)

To cast this algorithmic challenge in a clear framework, we borrow some basic ideas from the SDN literature [Raghavan2012, Kang2013].

In SDN, the **high-level policies** implemented as applications running on the central controller, execute direct control over the way traffic flows through the data plane, which is in turn abstracted away towards these high-level policies as a "big switch" with the usage of forwarding abstractions. This offers an easy way to specify global operational priorities, like access control rules, admission control preferences, traffic steering criteria for service chaining, etc., with the bulk of the underlying complexity hidden. High-level policies are then mapped to the data plane by the **routing policy**, producing a forwarding path for each individual flow according to some fixed traffic engineering goals, like multipath load-balancing, resilience, or minimization of latency and congestion. Note that, from an algorithmic standpoint, there is nothing particularly fancy about these policies, as a clean, well-tested, and widely implemented collection of algorithmic and numerical recipes is available today to convert any set of abstract traffic control requirements (essentially, a traffic matrix) into adequate forwarding paths [Mehdi2010].

The lion's share of the complexity is buried deep within the third, final component, responsible for compiling the traffic flow specifications into actual forwarding rules that can then be readily deployed at the data plane. Such forwarding rules involve MPLS forwarding tables, IP routing tables, Ethernet MAC tables, optical lightpath configurations, OpenFlow flow tables, etc. This component, called the rule-placement algorithm [Kang2013], or the data plane **orchestration algorithm** [Kolias2014], needs to deal with the specifics of the underlying topology, the type, software and hardware version, and rule space limits of the inventory of individual switching devices, and the characteristics of the interconnection links. Implicit in most of the related literature [Kang2013, Alex2010, Gupta1999, Rottenstreich2013, Kanizo2013, Yu2010, Abley2007] is that, apart from correctness, the main optimization objective is to **minimize and balance the forwarding state** across the data plane. Here, the term "forwarding state" refers to any hardware and software resource necessary to realize some intended forwarding behaviour; "minimization" thereof implies cutting down on the amount of resources involved in programming that behaviour as these resources are subject to stringent, and often rather restrictive [Kang2013, Kanizo2013], technological limits; and finally "balance" means to dispense this state as much as possible to relieve individual data plane components from forwarding state overflow [Kanizo2013, Retvari2013]. To make this idea explicit though, we need to dig a bit deeper into how the data plane is modelled.

6.1.1.1 The data plane abstraction

First, we leverage the idea from [Raghavan2012] to separate the **edge**, that uses software forwarding over generic purpose CPUs and GPUs to implement a broad selection of traffic control, conversion, and steering functionalities, from the **core**, deployed over a mix of legacy and SDN-capable switches, responsible solely for providing transport services between edge ports, that is, to realize the "big switch" abstraction. Note that both the edge and the core work under the supervision of the central controller. Moreover, we assume that all traffic control, logging, shaping, and all access control measures are applied at the edge in software, where resource constraints are not that compelling, which leaves only packet transport services to be realized in the core. In what follows we consider only the core of the network.

The second constituent is a tiered approach; decoupling core components based on the way forwarding state is represented across these tiers.

Underlay – The underlay consists of "dumb" devices, and links between them, providing nothing more than bare point-to-point packet delivery service between configurable pairs of endpoints. In what follows, we shall call these point-to-point connections **forwarding abstractions** and we view these as the sole bearers of forwarding state within the underlay. As the underlay can be of very limited functionality, we like to think about it as something very fast, quasi-static (reconfigured infrequently or not reconfigurable at all), dumb (cannot combine forwarding abstractions, does not understand higher level network semantics), and deployed on specialized, costly, hard-to-upgrade devices with limited resources. This tier mostly corresponds to the "infrastructure" as of SDIA [Raghavan2012].

Overlay – The overlay contains "smarter" devices connected by forwarding abstractions, providing end-to-end packet delivery service between the edge ports. Overlay nodes arbitrate packets between forwarding abstractions, relying on higher level or even global protocol semantics (e.g. addressing, header types, configuration of forwarding abstractions) encoded as forwarding state (e.g. in the form of forwarding or flow tables). Due to the increased complexity of the forwarding operations in this tier, we think about an overlay node as something slower, more dynamic, programmable and running on cheaper forwarding fabric. This tier roughly coincides with the "architecture" in SDIA [Raghavan2012].

Note that any single physical equipment can find itself hosting both underlay and overlay functionality, similarly to how an OpenFlow switch can be seen as an MPLS switch for one flowspace and an IP router for another. Also note that complex SDN architectures may contain additional layers using the connections provided by the overlay as forwarding abstractions (for an analogy consider an IP over MPLS over optical network). For tractability reasons and space constraints we limit ourselves to the two-layer case and leave the analysis of more complex architectures for future work.

Packets – Somewhat remarkably, we consider the collection of data packets circulating around the data plane as a separate tier, on the basis that any data packet can contain forwarding state. Most legacy network protocols offer some means to communicate loose or strict explicit routes within the packet headers and SDN makes this even simpler. Correspondingly, we use the following abstraction for representing packets:

Flow identifier	Forwarding information	Payload

The flow identifier (like MPLS labels or the IP 5-tuple) can be used in combination with underlay and overlay state to make the appropriate forwarding decisions in these tiers. In turn, forwarding information can be interpreted in the context of a given node and contains explicit information regarding the treatment of the packet at that node (e.g. source routing). We assume that forwarding info is filled by routers/switches at the edge.

6.1.1.2 The cross-tier orchestration algorithm

We can now formulate the problem of orchestration within this framework:

The task of the orchestration algorithm is to, given a set of flows and corresponding forwarding paths, generate an optimal configuration of forwarding state to be programmed into the data plane in order to realize those paths, with the objective to minimize and balance said forwarding state across the data plane devices and tiers, subject to the capacity of each individual device.

Within this context, capacity constraints involve any technological barriers in the data plane, like for instance any limitation on the number of forwarding abstractions an underlay node can originate or terminate (e.g., the number of wavelength converters at an optical switch), any technological restriction on the forwarding table size at an overlay node (MPLS forwarding table size, max IP FIB or flow table entries, etc.) or on the size of the packet headers (e.g., max MPLS label stack depth, max IP header size, etc.).

Research is already under way to design such orchestration algorithms. There are methods to remove state from the underlay [Wang2001], the overlay [Alex2010, Rottenstreich2013, Retvari2013], or from the packet headers [Krishna2004], and several proposals exist to distribute state across the topology [Kanizo2013, Yu2010]. Unfortunately, these approaches do not allow trading off forwarding space in one tier for that in another, cutting down a sizable and valuable portion of the problem space.

The main goal of our research was, accordingly, to investigate the algorithmic aspects of data plane orchestration. We argue that the key is to minimize forwarding state across the tiers, and for mental model we advocate the above three-tier data plane abstraction. Only making the tedious job of uncovering all options in this problem space will we be able to supply a solid mathematical foundation and an optimization toolset to practitioners for building truly dependable software-defined networks.

In the next section, we demonstrate the usefulness of the three-tier data plane abstraction: we show that it suggests a representative classification of routing and forwarding paradigms based on the way each one represents forwarding state. It also suggests a new model that we call Software-Defined Routing, encompassing essentially all these routing paradigms.

6.1.2 Classification of routing & forwarding paradigms

Next, we turn to exercise the three-tier data plane abstraction to classify routing and forwarding paradigms, starting from simple ones like hop-by-hop routing and full-mesh overlays, all the way to complex ones like pathlet routing (see Figure 6-1). The classification will be intentionally made incomprehensive: the goal is not to cover all known routing paradigms but rather to come up with a broad but workable subset. In the below classification, the numerical code *X/Y/Z* marks the way forwarding state is represented: *X=1* if forwarding state appears in some form encoded as forwarding abstractions at the underlay and 0 otherwise, and similarly *Y (Z)* indicates the appearance of state at the overlay (packets, respectively). We start from the three basic paradigms that encode forwarding state only at a single tier and we move gradually towards more complex paradigms.

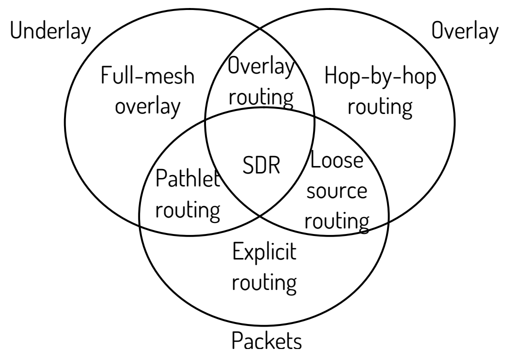


Figure 6-1: Routing & forwarding paradigms categorized by forwarding state representation.

Full-mesh overlays[14] (*1/0/0*) – In a full-mesh overlay every single flow is mapped to a separate forwarding abstraction, thereby encoding all forwarding state within the underlay and stripping all state from the overlay and the packet headers (apart from the flow identification which is always there anyways). Full-mesh overlays used to enjoy wide popularity to build IP/MPLS overlays [Wang2001], but later got dismissed due to quadratic scaling (in terms of the number of source-destination pairs, if this happens to be our traffic unit of choice).

Hop-by-hop routing (*0/1/0*) – This is the good old IP routing paradigm (MPLS with LSP counts here too, along with basic Ethernet etc.), whereas each individual network link and LAN is perceived as a separate forwarding abstraction. Data forwarding occurs by each node passing packets on to the next-hop by inspecting (some part of the) the flow identifier in the headers, and the related forwarding state is fully encoded into the forwarding tables maintained at network nodes. Hop-by-hop routing is known to suffer scalability issues, manifested in the rapid overflow of these per-node flow tables, a resource heavily restricted in today's hardware [Kang2013, Kanizo2013, Retvari2013].

Explicit routing (*0/0/1*) – Explicit routing confines all forwarding state into the packet headers, eliminating state from the underlay (so again, each physical link is a separate forwarding abstraction) and from the overlay (hence this model is sometimes also called "stateless routing" [Tapolcai2012]). This paradigm is often criticized due to notorious MTU-overflow issues, bad security and inefficiency (with excess header info overtaking useful packet payload), not to mention the maximum header length limitations found in many forwarding plane technologies.

Pure paradigms, which encode state at only a single tier, suffer from grave scalability issues. A plausible way out is to disseminate state at multiple tiers, as of the following three paradigms.

Overlay routing (*1/1/0*) – Overlay routing uses tunnels as forwarding abstractions, whereby the underlay exposes a logical network topology to the overlay which can then readily route over this logical topology. Hence, state appears at the underlay and the overlay, but not in packet headers. A typical example is IP/MPLS overlays or, more recently,

[14] The naming of this paradigm slightly confusing as it is called "full-mesh overlay" and we don't keep state in the overlay at all, however we keep this term to be compliant with the literature which uses this term covering the same functionality.

SDN traffic steering mechanisms, whereas tunnels are configured through OpenFlow and overlay nodes are hosted at middleboxes [Anwer2013].

Loose source routing (*0/1/1*) – Loose source routing (LSR) uses a combination of state encoded in forwarding tables at the overlay to route packets between subsequent stages of the forwarding path and the succession of these stages is encoded within the packet header as loose source routes. In a pure form of LSR no forwarding state appears within the underlay. LSR seems to enjoy renewed interest, with segment routing advocated at the IETF for better routing flexibility [Previdi2014b].

Pathlet routing (*1/0/1*) – Pathlet routing uses short path slices (called "pathlets") as forwarding abstractions and nodes arbitrate packets between these abstractions based on the stack of pathlet IDs in the packet headers. Pathlet routing is mostly intended for general inter-domain routing [Koponen2011, Godfrey2009] and SDN traffic steering [Chiesa2014], but there is no reason whatsoever why we couldn't leverage this concept as a general routing and forwarding concept.

One might wonder now what is exactly at the coordinates *0/0/0* and *1/1/1*. The former corresponds to the interesting but completely out-of-scope case whereas no state appears anywhere, neither in the underlay, nor in the overlay or in the headers (apart from the mandatory flow identification field), yet packets somehow still arrive to the destination. One might argue that random walks, network flooding, ant-colony optimization, or quantum communication could be classified into this category, but the model just breaks at this point. More interesting is the case of *1/1/1*, the all-encompassing and ultimately flexible routing paradigm that allows to represent forwarding state in any assortment of tunnels, routing tables, or packets. Due to this versatility and universality, we call this combination the Software-Defined Routing (SDR) paradigm.

Software Defined Routing (*1/1/1*) – Within SDR, we are free to establish whatever forwarding abstractions at the underlay, exposing these to the overlay, and get packets through this virtualized data plane using both forwarding state placed by the controller at the overlay nodes and explicit routing information encoded by the edge into the packets.

The algorithmic challenge is now to come up with suitable orchestration algorithms that can map any set of input paths to such a broad combination of data plane tiers, meanwhile minimizing the amount of state encoded at any single tier. Finding a mathematical model for the problem in this general form seems daunting though. In what follows, therefore, we define and analyze a slightly simpler model.

6.1.3 Trading off space across tiers

The complexity and the memory requirement of making a forwarding decision can exhibit high variation across our three tiers. In case of forwarding in the underlay, the process is simply determining the forwarding abstraction the packet belongs to and output the packet on the corresponding port. If the packet cannot be forwarded in the underlay (e.g. the node is the last node of the forwarding abstraction) it is escalated to the overlay or the packet tier (see Figure 6-2). Forwarding in the overlay implements the logic of combining the forwarding abstractions exposed by the underlay based on the flow identification part of the header. We argue that overlay forwarding may be more complex (e.g. involve more complex matching processes and actions). For an analogy, label-based switching is much faster than the enforcement of IP level policies prescribing which MPLS tunnels to combine to realize some end-to-end connection in IP. However if we want to implement the same connections with MPLS the memory requirement would be much higher as we would need MPLS tunnels between every pair of IP endpoints. Finally, forwarding using the forwarding information in the packet needs to *(i)* interpret the forwarding info e.g. read the appropriate section and translates what it means at the given node and *(ii)* updates this field by e.g. adding/removing new/obsolete parts before outputting the packet.

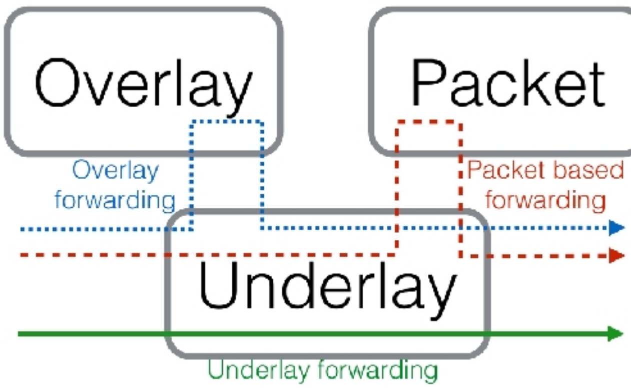


Figure 6-2: Possible forwarding decision paths in a physical node.

In what follows, we formulate a model that can capture non-trivial aspects of the above general architecture yet permitting analytic tractability. To achieve this goal we will lose a little of the generality of the above framework and will not differentiate between the packet and overlay tiers. However, we do differentiate between the underlay and the rest of the tiers, as underlay forwarding is clearly the cheapest, fastest and greenest option, and we try to push forwarding decisions as much as possible down into the underlay and escalate the forwarding process to other tiers when absolutely unavoidable. For the sake of simplicity from now on we consider our underlay tier to be the bare bone physical infrastructure.

6.1.3.1 Model

We associate an undirected graph $G(V,E)$ with the underlay network, where V $(|V|=n)$ corresponds to physical switching devices (e.g., routers or switches) and E $(|E|=m)$ represents physical links (e.g., cables). Let the traffic unit be source-destination pairs: $S = \{(s, d)\}, s, d \in V$, with a desired forwarding path specified for each (s,d)-pair by the routing policy.

Forwarding abstractions – A forwarding abstraction t_{ij} of length $|t_{ij}|$ from node i to j is an ordered set of $|t_{ij}|$ edges that form a path from i to j in G

$$t_{ij} = \{(i = v_0 , v_1), (v_1 , v_2), ..., (v_{|tij|}-1, v_{|tij|} = j)\}, \text{ where}$$

$$\forall k \in 0, ...,|t_{ij}|-1, (v_k, v_{k+1}) \in E .$$

Denote the set of all forwarding abstractions by T and let $l_T = |T|$. With a slight abuse of notation, we let t_{ij} to also denote a unique id for the corresponding forwarding abstraction.

Forwarding – Forwarding in the underlay is done by an exact matching on the forwarding abstraction id encoded in the packet header from a static table stored at the node and running a simple output action. The forwarding table F_v at node $v \in V$ consists of a set of rules $F_v = \{t_{ij} \rightarrow (o_{ij})\}$, where t_{ij} is a forwarding abstraction id and o_{ij} is an output port forward the packet on. The number of entries in F_v essentially determines the amount of forwarding state stored at the underlay at node v. For simplicity, we assume that l_T is a good representative for the size of F_v. In fact, this is only an upper bound. Yet, this assumption will prove indispensable for the below analysis. Note that whenever there is no appropriate id t_{ij} in the header or a matching entry is not found in F_v, the decision needs to be escalated to the overlay or the packet tiers.

6.1.3.2 Problem formulation and complexity

The problem of orchestration in this model is to determine the optimal configuration for T, so that the forwarding state at the underlay nodes (represented by l_T) and the number of escalations to other tiers ε are minimal.

Definition 1 (The SDR problem) – Given a graph $G(V,E)$, flows S with paths P_{sd}, and integers E and L, find a configuration T so that there is a correct route for each $(s,d) \in S$ (i.e. each $P \in P_{sd}$ can be expressed as a concatenation of some $tij \in S$), while $l_T < L$ and $\varepsilon \leq E$.

The difficulty lies at provisioning just the optimal number and configuration of forwarding abstractions. On the one hand, if we let all source-destination paths be assigned a separate forwarding abstraction then we end up with $l_T = O(n^2)$ and $\varepsilon = 0$ (full-mesh overlay). The other way around, i.e., letting each physical link be a forwarding abstraction yields $l_T = O(m)$ and $\varepsilon = O(n)$ (explicit routing or hop-by-hop routing). Easily, the truth must be somewhere in between. It turns out, however, that finding the sweet spot at which state is optimally distributed between the two extremes is very difficult.

Theorem 1 – The SDR problem is NP-complete.

The proof is by reduction to the hitting set problem. Note also that the problem is not even approximable within any constant $c>0$, unless $P=NP$.

6.1.4 Results

It seems that exercising the underlay forwarding vs. escalation trade-off analytically is very difficult in general, due to the intractability of the underlying optimization problem. Instead of diving deep into analytical bounds and exact and approximate algorithms, we rather highlight here some of the results we have in order to demonstrate this trade-off. For the sake of simplicity, from here onwards we shall assume that $S=V\times V$ (all-pairs version) and the routing policy is shortest path over unit cost graphs. Under this assumption, $E\in T$.

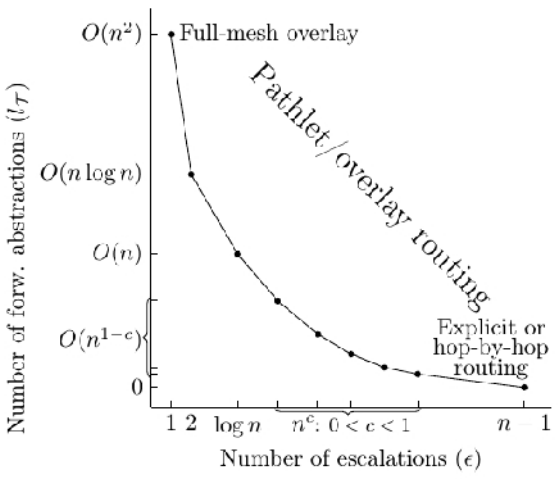


Figure 6-3: The "underlay forwarding vs. escalation to higher tiers" trade-off.

If we restrict the underlying graph to be a tree then we can give some useful upper bounds. Figure 6-3 illustrates the above trade-off for this case. For the sake of cleaner presentation, the notation l_T in this case denotes only the number of forwarding abstractions in excess to the trivial one-hop ones (i.e., $l_T = |T| - m = |T| - n + 1$). The most interesting finding seems to be that we can do with at most *2* escalations just by adding $O(\log n)$ forwarding abstractions to each node on average or $\log n$ escalations with constant number of forwarding abstractions per node. Note that the optimum might be even better, as these results are upper bounds. Note also that the claims hold for any tree as the bounds are analytic.

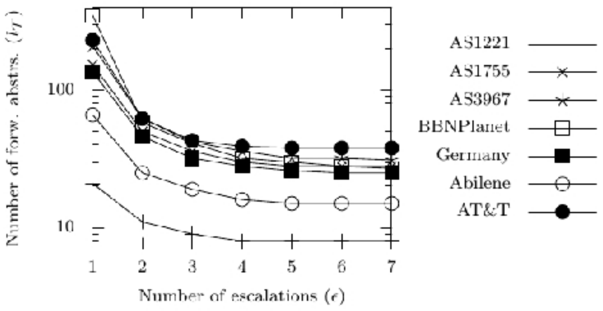


Figure 6-4: Number of forwarding abstractions in the optimal SDR configuration as the function of the escalations, for various real-world topologies. Note the logarithmic scale on the y-axis.

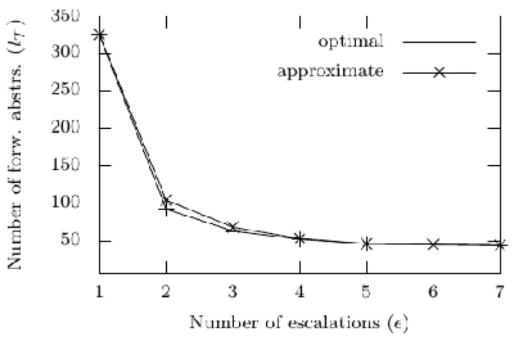


Figure 6-5: Number of forwarding abstractions in the approximate SDR configuration as the function of escalations for the NSF topology.

Stepping beyond trees, we defined an integer linear program (ILP) suitable to obtain the optimal solution for the SDR problem over general graphs. The ILP contains a couple of thousand variables and constraints even for mid-size problems, so for larger graphs we went with a greedy heuristic. The idea here is to, in each iteration, instantiate the forwarding abstraction that reduces the number of escalations the most. The results with the ILP over various ISP topologies are given in Figure 6-4 and a comparison of the optimal and the approximate solutions is given in Figure 6-5 for the representative NSF (AS 102) topology.

It seems that the transition from "all state in the underlay" (full-mesh) to "all state above" (explicit/hop-by-hop routing) is reassuringly fast: adding only a moderate number of forwarding abstractions, such that the average forwarding state per router is limited to only a few entries, say, *1–5*, the number of escalations drops quickly from the prohibitive regime of dozens to the fairly reasonable range of *2–4 constituting a sweet spot mixing the advantages of both forwarding technologies..*

This showcases the power of SDR: while it is very well imaginable that an operator could not allow (due to intrinsic limitations of the switches) or would not want (for efficiency reasons) the header size/overlay state to grow beyond a certain limit, or could not allow establishing hundreds of forwarding abstractions for supporting full-mesh overlays, and so said operator could not realize the intended routing that he or she sees best fit for the traffic engineering goals, our SDR model brings the solution within reach by distributing forwarding state across the tiers. Meanwhile, it significantly reduces the state in all tiers from what was available with "pure" routing paradigms. We believe that this short case study sufficiently demonstrates the power of an SDN orchestration algorithm based on our model of SDR. Moreover, these results can provide useful inputs for designing future traffic steering algorithms hosting the programmability and orchestration framework of UNIFY.

7 Scalable orchestration architectures

This section is devoted to summarize our main achievements regarding a service programming and orchestration framework to implementation the UNIFY concepts, where they can be validated, tested, and in turn provide feedback to further refine concepts and developed algorithms. Our main goal is to design a scalable orchestration architecture with all relevant components and to implement several elements of that in a common framework. These building blocks are the key components of our UNIFY programmability framework which enables the joint programming and virtualization of cloud and networking resources. An essential requirement from our framework is the support of efficient integration of different modules implemented by different partners and the easy re-use and integration of available tools.

Second, we give a high-level description of the next version of our proof of concept prototyping environment called ESCAPEv2. This framework realizes all the relevant components of our scalable orchestration architecture. More specifically, it supports (i) operations with almost arbitrary sized SGs, RGs and NF-FGs; (ii) different virtual network embedding/mapping algorithms; (iii) NF decomposition; (iv) different technological domains via adapter modules; (v) easy integration and extension. The current state of the prototype and further implementation details can be found in Annex 1.

On the one hand, the orchestration framework has to inherently support different legacy domains (network and cloud as well) controlled by state-of-the-art software and/or hardware tools. On the other hand, the relevant fraction of today's internet traffic is originated or terminated at wireless end-points; therefore, orchestrating radio resources has significant importance. Hence, a separate section is devoted to present our approach on joint orchestration of radio and optical transport resources including the concept and a proof of concept demonstrator.

7.1 Design and implementation of ESCAPEv2

We have established a prototyping framework called ESCAPE (Extensible Service ChAin Prototyping Environment using Mininet, Click, NETCONF and POX) including the 3 layers of UNIFY architecture, namely, Infrastructure Layer (IL), Orchestration Layer (OL), Service Layer (SL) and demonstrated the first version in [Csoma2014]. ESCAPE has been significantly redesigned in order to:

- Get a scalable orchestration architecture with all relevant components
- Follow more closely the current functional architecture of UNIFY
- Make integration with other prototypes easier.

The main goal of ESCAPE is to support the development of several parts of the service chaining architecture including VNF implementation, traffic steering, virtual network embedding, etc. However, here we focus on the orchestration part. ESCAPE is (mainly) implemented in Python on top of POX platform and Mininet [POX, Mininet].

The modular approach and loosely coupled components make it easy to change several parts and evaluate own algorithms. In this section, we give a high level overview of the framework. Further details on the implementation are given in Annex 1. The system architecture of the latest version of ESCAPE is shown in Figure 7-1.

The Service Layer contains an API and a GUI at the top level where users can request and manage services and VNFs. The API is capable of formulating SG from the request and passes that to a dedicated service orchestrator which is responsible for gathering resource information (RG) from Virtual resource manager. This is the virtual view provided by the Virtualizer of the lower layer (e.g, BiS-BiS view). Mapping of SG to RG is delegated to the SG mapper module which constructs an NF-FG storing the request, the virtual resources and the mapping between NFs and infrastructure nodes in a common data structure. In case of a single BiS-BiS view, the mapping is trivial and might be omitted by constructing an NF-FG carrying information only on the request and the resources (unmapped NF-FG).

OL encompasses the most important components of the resource orchestration process which replaces the ETSI's VIM. An API is set up on the top centralizing the interaction with the upper layer and realizing the Sl-Or interface. On the one hand, the request coming as an NF-FG is forwarded to the RO via the corresponding Virtualizer (which is responsible for policy enforcement as well). On the other hand, the virtual view created and managed by the Virtualizer is provided as an RG to the upper layer. RO is the key entity managing the components involved in the orchestration. The input is an NF-FG which should be mapped to the detailed domain view. Albeit, this view is also an abstraction provided by the Domain Virtualizer. RO collects and forwards all required data to RO mapper. More specifically, the NF-FG, the domain view (as an RG) and the NF-IB are passed to the RO mapper which invokes the configured mapping strategy and interacts with the Neo4j-based graph database containing information on NFs and decomposition rules. NF-IB corresponds to "VNF Catalogue" in NFV MANO with the difference of supporting service decomposition. The outcome is a new NF-FG which is sent to the Controller Adaptation part. The role of CA is twofold. First, it gathers technology specific information on resources of different domains then builds an abstract domain view. The interaction with different types of technology domains are handled by adapters, such as POX adapter for OpenFlow networks controlled by POX, Mininet adapter for Mininet domains. Second, the incoming NF-FG request is decomposed according to the low-level domains and delegated to the corresponding adapters.

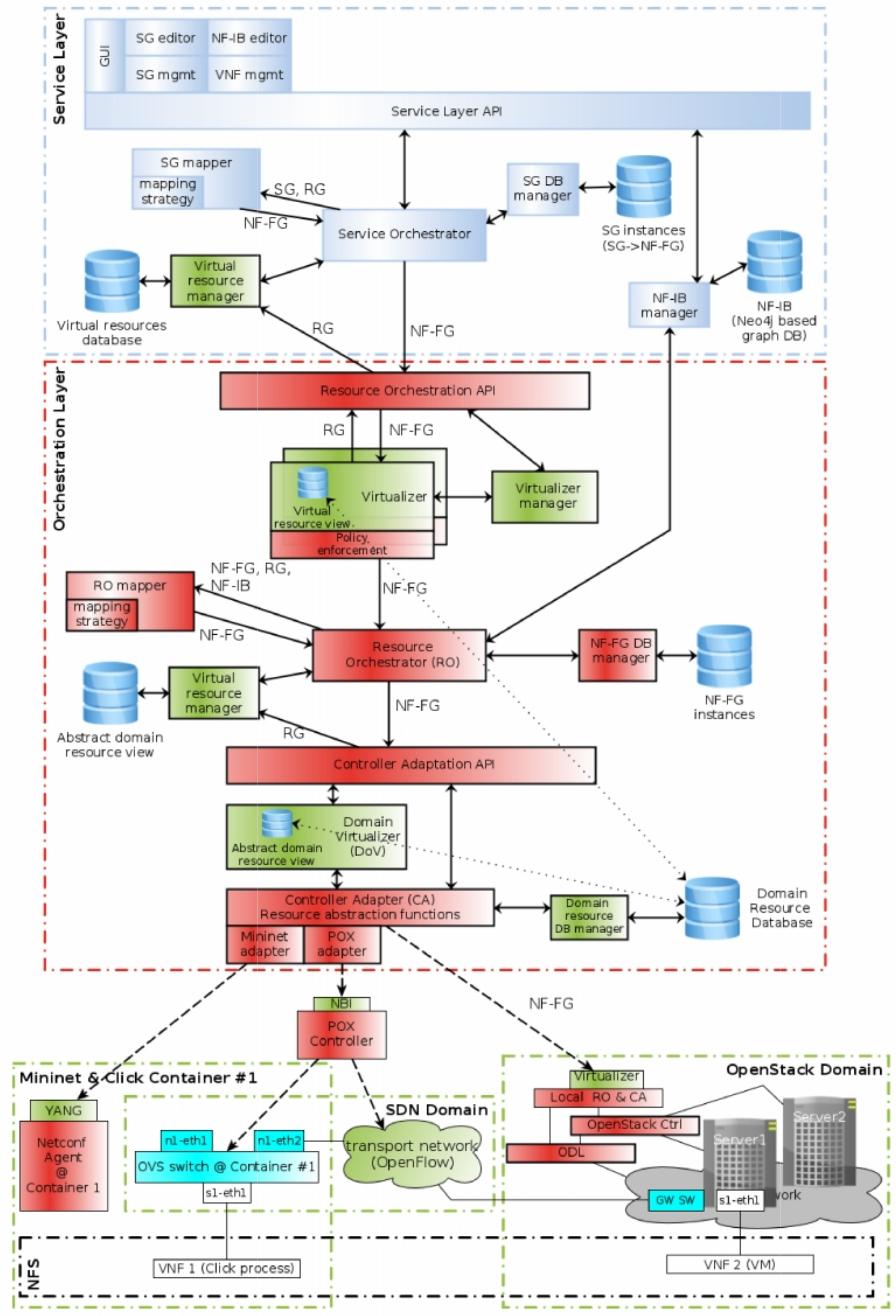


Figure 7-1: System architecture of ESCAPEv2

We have implemented a simple, Mininet-based infrastructure environment in order to support rapid prototyping (see lower left part of Figure 7-1). Here, VNFs are implemented in Click modular router [Kohler2000] and run as distinct processes with configurable isolation models (based on Linux cgroups), while the network infrastructure consists of OpenFlow switches (realized with Open vSwitch software switches). The orchestration over this domain is realized by two distinct control interfaces. We have a dedicated adapter for compute resources of Mininet domain (Mininet adapter) communicating the NETCONF protocol, and we have another one for network resources. The latter is implemented by the POX OpenFlow controller and a corresponding adapter. A dedicated controller application is responsible for steering traffic between VNFs, implemented as a POX application which sends flow entries to OpenFlow switches. We extended Mininet with NETCONF capability in order to support managed nodes (VNF containers) hosting VNFs. Via the Mininet adapter module, the orchestrator is able to start/stop VNFs on demand. The interface with the remote procedure calls and data structures is described by our special YANG model. Based on the YANG model, low-level instrumentation codes were implemented as NETCONF modules to hide the infrastructure level details. This includes dynamic port creation in the switch of the containers, running Click processes with given parameters, and connecting them via virtual Ethernet pairs, etc. It is worth noting that this approach supports migration to other environments, such as Docker domains (only the instrumentation codes have to be replaced).

ESCAPE supports recursive orchestration by implementing the Sl-Or interface at CA towards lower level domains. This shows the benefits of our joint programmatic interface based on the NF-FG model. In this way, we need only a single control interface. The Sl-Or interface supports integration of Universal Node domains as well as other Sl-Or capable domains. We have for example combined our framework with an OpenStack (OS) data center with the OpenDaylight (ODL) controller to form a multi-domain setup where multi-level orchestration could be performed. The corresponding components of this complex framework are shown in the lower right part of Figure 7-1. Here, ESCAPE is responsible for global orchestration on top of Mininet and OS/ODL domains while the latter has its own local orchestrator connected by an Sl-Or interface. In OS/ODL domain, VNFs are deployed as virtual machines (which run Click processes in our framework). An abstract view of the whole OS/ODL domain is exposed to ESCAPE orchestrator by a virtualizer. In the simplest case, this virtualizer provides a BiS-BiS view, i.e., a VNF container with a lot of resources and with local orchestrating capability. The global orchestrator is capable of partitioning an NF-FG into multiple subgraphs which can be given to the OS/ODL domain for further decomposition and instantiation. The local orchestrator of OS/ODL domain is based on OS components managing VMs (e.g., Nova) and ODL[15] controlling OpenFlow switches of the data center. These elements can be considered as traditional infrastructure controllers with corresponding controller adapter modules responsible for low level technology dependent details, such as creating ports dynamically, starting/stopping VMs, configuring network to steer traffic properly, etc. After successful instantiation of the received NF-FG, the local orchestrator notifies the global orchestrator and provides necessary access information.

[15]OS delegates network configuration requests to ODL via the Neutron REST API and ODL configures the internal switches via OpenFlow.

7.2 Orchestration of Radio and Transport Resources

Today a big fraction of Internet traffic is originated and/or terminated at wireless devices and end-points, and it is predicted that in the future the wireless traffic will become the majority. Therefore, it is potentially important to factor in the wireless networks' resources in the orchestration process In particular, we need to understand how the resource management in a wireless access network can be supported by the UNIFY architecture, and if including new types of resources, i.e. radio resources, in the orchestration would have an impact in the designed architecture and processes. For this purpose, we have designed and implemented a proof of concept (PoC) that includes radio connectivity in an LTE networks and optical transport networks. The proof of concept has been developed in a joint effort with the COMBO project [COMBO]. The PoC includes the infrastructure layer and the orchestration layer of the UNIFY architecture.

7.2.1 Integrating Radio Resource Control into the Orchestration Architecture

Controlling wireless networks is a complicated task, partly due to the fluctuating nature of the wireless links, and its large set of functions. These range from pure radio resources' control to the radio network connectivity functions. The control functions might also vary depending on the employed wireless technology. In our PoC, we consider an LTE-based mobile broadband network with the centralized radio access network architecture. In this architecture the baseband processing functions are decoupled from access points and pooled in one or more baseband processing units (BBU) hotel. The control of such a network can be performed through a centralized domain-specific controller, managing different aspects of the radio access network (RAN Controller). The main functionality of the RAN Controller in the architecture is to configure and allocate BBU resources, i.e. physical network functions, from the existing pools to the radio remote units (RRUs). This process is required for the activation of cells in a RAN area. Also, the RAN Controller can be used to trigger users connected to an access point (RRU) to perform a handover to a new access point, for example in case the former one needs to be deactivated. In addition, the RAN Controller has access to monitoring functions within BBUs, which can provide higher layer controllers and applications with up-to-date information about the performance of the RAN. This, for example, can be used to monitor the throughput of an active cell in the RAN.

From the control architecture's perspective there are several alternatives for integrating the RAN controller into the UNIFY orchestration architecture. One possibility is to consider the RAN Controller as a client of the transport and cloud resources' orchestration. In that case, the RAN Controller receives an abstract view of the transport and cloud resources via the RO, e.g. in form of a BiS-BiS abstraction, and performs resource allocation according to the received abstract view. The benefit of this architecture will be simplicity of the orchestration layer, as it does not need to integrate RAN resources. On the other hand, the resource allocations decisions of the RAN Controller might not be globally optimized.

Another alternative, which we adopt, is to consider the RAN Controller as a domain-specific controller in the Infrastructure Layer and at the same level as the network and compute controllers. A difference between the RAN

Controller and the network and compute controllers would be that the former one operates only on the physical resources, like BBU units.

The introduction of the RAN Controller at this level of the control hierarchy will also impact the orchestration layer. First and foremost, the radio resources should be included in the abstract view of the resources presented towards the service layer by the orchestration layer. More specifically, the Bis-Bis abstract view of the resources should be extended with pools of physical resources, e.g., BBU units, connected to a virtualized view of networking and processing resources. Also, the stringent requirements of the BBU-RRU association in terms of bandwidth and latency need to be considered when mapping a service request to an abstract view of the resources containing centralized RAN.

7.2.2 Proof of Concept Elements

Figure 7-2 depicts the high-level architecture of the proof of concept (PoC) for the radio and transport orchestration. The PoC contains two domains: a Dense Wavelength Division Multiplexing (DWDM) centric transport network and an LTE-based mobile broadband network.

The transport domain is a dynamic wavelength routed network and provides transport service at the wavelength level. The domain is composed of optical DWDM switches, optical add/drop multiplexers (OADMs) and tuneable optical transponders (TPs) at the edge of the network. The offered wavelength services are programmable through an SDN controller that manages resource allocation in the domain. To realize the transport controller we utilized the open-source OpenDaylight (ODL) as the basis, and extended it with several functions to optimize it for control of large-scale DWDM networks. The first extension includes the design and implementation of southbound plugins for control of existing DWDM network products (i.e., DWDM switches and TPs). Also, the ODL is extended to support the circuit-switched types of services. Additionally, we have integrated an optical path computation element (PCE) into the ODL. Finally, ODL is extended with an additional layer of transport abstraction/virtualization on its top, so that all details of the DWDM layer are abstracted and hidden from higher-layers controllers (i.e., orchestration layer). Specifically, the Orchestrator only sees a “big switch” representation of the whole transport domain, which makes the service creation process simpler and more scalable.

The LTE-based mobile broadband domain provides broadband services to mobile users. The domain is composed of LTE access points, deployed according to the centralized RAN architecture. In this architecture the common public radio interface (CPRI) is used to interface between RRUs and BBUs. DWDM is the technology of choice for transporting the CPRI, due to its high bandwidth as well as stringent latency/jitter requirements. Accordingly, this domain employs wavelength services of the transport domain for CPRI transport (i.e., DWDM domain is used as fronthaul). In our PoC, two wavelength channels are used to connect a pair of (BBU, RRU); one wavelength per direction. For controlling the mobile domain we use an existing domain-specific controller that centrally controls the RAN resources.

Two general-purpose machines are used in our platform for running the control plane elements. Specifically, the first machine is used for transport controller and the orchestration, and the second one is used for running the RAN Controller and a Radio-Transport Optimization Application. The Optimization App. is developed and managed by a mobile network operator (MNO), which utilizes the infrastructure for providing the mobile broadband service.

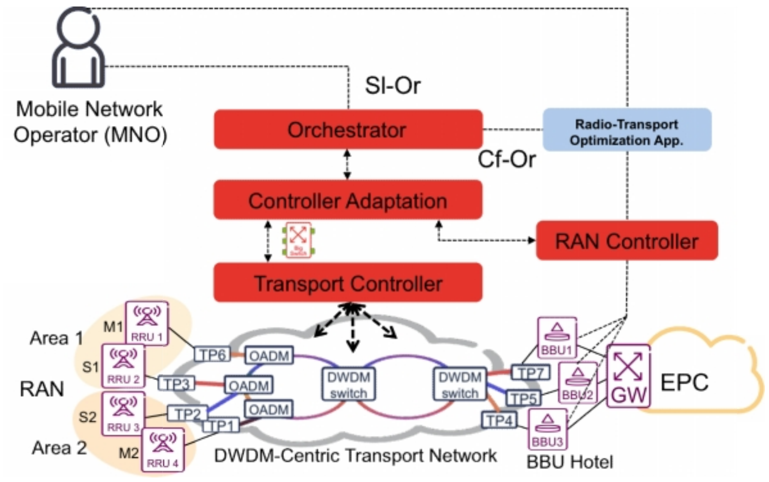


Figure 7-2: Radio-Transport Orchestration PoC

7.2.3 Elastic Mobile Broadband Services: The Orchestration Process

To demonstrate the benefits of the radio-transport orchestration, we have implemented a use-case called elastic mobile broadband services (EMBS), where the service capacity is dynamically and automatically scaled up and down—when and where needed. The EMBS service can be particularly beneficial for environments with dynamic service demands like business/residential areas of cities at different times of the day. The EMBS operation requires that the DWDM transport and radio resources are dynamically utilized in a coordinated manner. We have implemented the EMBS in our PoC by creating a RAN with two areas: Area 1 and 2 (see Figure 7-2). Each RAN area is equipped with two RRUs: a macro cell (M) and a small cell (S). While macro cells provide the coverage across the areas, small cells are used for providing additional capacities in areas if needed. The total of 4 RRUs (across two areas) are served by a BBU hotel with 3 BBUs. The service management logic is integrated into the Radio-Transport Optimization App (Figure 7-2). The application receives an abstract view of the available radio and transport resources (from the orchestrator), and continuously monitors the service demand in the RAN, through monitoring the throughput of active cells (via the RAN Controller).

The state diagram of the EMBS operation is depicted in Figure 7-4. In the default operation mode (State 1), only the two macro-cells are active for providing coverage in both areas. When extra demand is identified in an area by the optimization application, the corresponding small cell is activated (State 2 or State 3). In our PoC, only one small cell is activated at a time, to serve the area with a higher demand. This demonstrates dynamic (time-sharing) reuse of resources where a 4-cells RAN requires only 3 pairs of WDM connections and 3 BBU resources, leading to a saving in terms of transport and radio resources. In a real deployment, obviously, there would be much more RRUs and BBUs leading to a much larger saving.

The interactions among different control plane elements of the demo will be as follows. At the first step the control plane connectivity is established and the MNO instantiates the optimization App. Then, the Optimization App can request, from the Orchestrator, an abstract view of the network ("get resource info"), which is composed of a big switch, representing the optical transport network, and 3 physical NFs (i.e., BBUs) connected to the ports of the big switch.

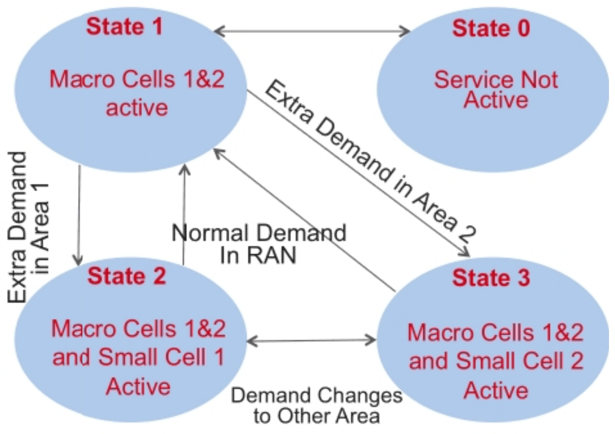


Figure 7-3: State diagram of the EMBS operation

Then, the Optimization App. triggers the default operation of the MBBS, i.e., State 1 in Figure 7-3. This request is sent to the RO, in form of an "instantiate NF-FG" over the Cf-Or interface. The requested NF-FG is composed of transport resources (abstracted as point to point links) and BBUs, as depicted in Figure 7-4a. The RO translates the requested NF-FG to the connectivity in the data-plane. In this case, the orchestration layer requests the transport controller to provide two wavelength connections: TP7-TP6 for activation of macro cell in RAN Area 1, and TP4-TP1 for macro cell in RAN Area 2 (see Figure 7-2). The Transport Controller performs the routing and wavelength assignment for the two requested wavelengths and accordingly configures the optical network.

Over the lifetime of the EMBS service the Optimization App can trigger switching among states 1-3 depending on the demands for service and the policy. For example, at some point in time the Optimization App might decide to switch from state 1 to state 3. This will involve sending a "change NF-FG" request to the RO, again over the Cf-Or interface, to add a new NF-FG, as shown in Figure 7-4b. The NF-FG change request will be translated by the RO into the addition of new wavelength connectivity in the transport: TP5-TP2. Similarly, the Optimization App. might trigger deactivation of the small cell in Area 2 and activation of small cell in Area 1 (i.e., switching from state 3 to state 2).

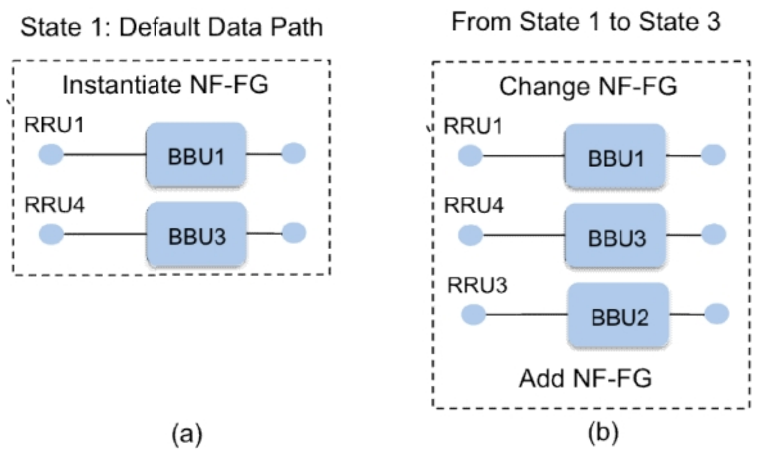


Figure 7-4: Changes to the NF-FG during the lifetime of a EMBS. a) Default data path NF-FG is requested by the Optimization App. (b) NF-FG request for switching from State 1 to State 3.

The Radio-Transport orchestration presented above highlights how the wireless networks' and optical transport networks' resources can be integrated into the orchestration architecture of UNIFY. In summary, we observe that wireless networks extend the UNIFY orchestration process with another dimension, namely the radio resources. Specifically, the wireless network in the above example has particular network functions, which require a domain-specific controller (i.e., RAN Controller) in the infrastructure layer, at the same level as the networking and processing controllers. Consequently, this will influence the domain-wide resource abstraction and virtualization performed at the Orchestration layer. For example, in our PoC the Centralized RAN network includes physical network functions like BBU that cannot be supported with the BiS-BiS abstraction model. One possibility here would be to extend the BiS-BiS model of resource abstraction to further include connected pools of physical resources, like BBU pools. That is, a MNO on top of the UNIFY Orchestrator can get, over the Sl-Or, a view of the resources composed of Big Switch, Big Software and one or more pools of physical resources.

8 Conclusions

This deliverable has refined the service programmability framework documented in D3.1. The goal of service programmability and orchestration functionality is to enable fast, on-demand, flexible and elastic deployment of telecom services based on the interconnection of virtualized network functions. In this document we took a top-down approach from the Service layer, through the Orchestration layer, and ended in the Infrastructure layer. Through this we documented our findings in the key technical areas of service programming and orchestration in each of the layers. In the Service layer the identified key technical issues are on providing service programming interfaces and their associated data models. In the Orchestration layer we focused on the technical challenges of performing scalable resource orchestration and dynamic decomposition of abstract network functions into specific implementations. In the Infrastructure layer the main challenge is management of state, both inside the network functions and the network elements providing connectivity between the functions. This deliverables summarizes the advancements made corresponding to state of the art, by proposing concrete data models, algorithms, implementations and/or evaluations of resulting performance.

Service programmability requires sound interfaces, languages, and most importantly underlying data models. The NF-FG data model is at the heart of the UNIFY programming framework, specifying how services should be deployed, what their requirements are and how they should be monitored. Due to the recursive nature of the UNIFY orchestration framework, it must fully support the abstractions and virtualization steps in the architecture, adding additional level of complexity when specifying it. We presented the two NF-FG modelling approaches that we are evaluating, one service-centric and one based on the Virtualizer component. The first, service-centric, model has its roots in the model initially specified in D3.1 which has since evolved based on feedback from prototyping efforts, an updated architecture from WP2, developments in WP4 and the universal node prototype in WP5. The second NF-FG model takes a different approach with the focus on modelling the Virtualizer component responsible for presenting a virtual network view to higher layers. Here the configuration of the exposed virtualized resources defines the NF-FG. At the same time parallel developments outside the project, such as OASIS TOSCA and OpenStack Heat have been taken into account in the proposal of the two NF-FG YANG data models. These include modelling capabilities of the exposed topology and resource model of the underlying (virtualized) infrastructure. Due to the recursive nature of the UNIFY architecture, finding information regarding various NF-FGs, resources, and monitoring results become non-trivial at higher layers as several virtualization and aggregation steps separate the higher layers from the bottom layer infrastructure. To aid the higher layers in obtaining fresh information from lower layers a recursive query language based on Datalog is proposed and under development. SLA assurance and associated monitoring functionality is covered in the NF-FG model through the MEASURE framework detailed in WP4.

Service decomposition is the programmability concept which enables stepwise refinement of NF specifications within the programmability process flow. The model-based service decomposition introduced in D3.1 and D2.2 naturally suits the recursive UNIFY architecture. This concept has been further clarified, highlighting the advantages it has, its limitations, and when it could and should be applied. Furthermore, the concept of atomic blocks, the entity on which decomposition terminates, has been refined. Several types of atomic blocks have been identified, from dedicated hardware NFs, to different granularities of software blocks that can be composed into a VNF function. For the software based atomic blocks not only the software components themselves are important, but also the tools necessary for packaging the components into a format that can easily be deployed. Here several tools have been identified, such as virtual machines, containers, and the specialized ClickOS VMs for deploying Click scripts. Finally, several examples of how these different tools and technologies can be combined to create new services have been shown.

Resource orchestration, the process of mapping decomposed NF-FGs and their requirements to the lower layer resources, has been discussed starting from a VNEP problem formulation. Here several advancements have been made, both for "pure" VNEP mapping algorithms and algorithms that support decomposition of the NF-FG together with the mapping of it. These are further split into on-line algorithms and off-line algorithms. Three algorithms for "pure" VNEP mappings have been presented and evaluated. The first algorithm is an on-line algorithm to handle service chain embedding in dense multi-edged topologies, in which forwarding elements can have multiple links to each other. The second algorithm handles virtual cluster topologies, a star topology for server-to-server communication over a central switch, this shows that the VNEP for virtual cluster topology is not NP-hard but can in fact be solved in polynomic time. Finally an off-line Mixed-Integer-Programming algorithm that maps multiple NF-FGs at the same time has been presented. It additionally allows for reconfigurations of existing embeddings and elasticity. Two algorithms, one off-line and one on-line for service decomposition and mapping has been developed. These show large benefits to taking the decomposition options into account when performing the resource mapping. Finally, a framework for distributing on-line algorithm calculations over multiple nodes has been prototyped and tested, using open source components.

To support the service elasticity and scalability promised by the NFV approach, the ability to dynamically add and remove VNF components from an NF-FG is required. To be able to do this without interrupting or degrading the service provided by the VNFs involved their internal state must be handled in unison with the network state. Here we have analyzed a network function common in provider networks in terms of its internal state and how it should be handled. We also improved on previous work on a VNF state management framework in order to lessen the impact of state management both on the user traffic and the orchestration infrastructure, showing significant improvements in both time and messaging overhead. The proposed schemes have been validated in an extension of the state-of-the-art OpenNF emulation framework. Additionally, mechanisms for providing high availability to NFV services through protection and recovery mechanisms have been characterized.

Supporting scalable VNFs alone is not enough without a scalable data plane to connect the VNFs and steer the user traffic through them in an efficient and flexible manner. To provide this we described our initial findings on the theoretical aspects of traffic steering and introduce the novel concept of Software-Defined Routing (SDR), which allows placement of forwarding states at arbitrary levels in the network architecture. This study shows that the SDR approach can significantly reduce the state in all forwarding tiers compared to the amount of state required with "pure" forwarding paradigms.

Concepts, algorithms and characterized components of the service programming and orchestration framework are only able to show their true value if once they are integrated within proof-of-concept prototype(s), confronting them with low-level technical aspects. The latter has been the goal of different prototypes, and particularly of the ESCAPE emulator. The result of this continuous confrontation has led to a high-level functional description of the next version of the prototyping environment ESCAPEv2. This environment supports operations with large SGs, RGs and NF-FGs; pluggable VNEP algorithms; service decomposition; interactions with different technological domains, and is easily integrated and extended. Finally, to support the large fraction of today's internet traffic that terminates in wireless environments, the radio resources has to be taken into account. We presented our approach to joint orchestration of radio and optical transport resources, including a proof of concept demonstrator.

In summary, this deliverable provided refined view and trade-off analysis of service decomposition, concrete proposals and evaluations of orchestration algorithms and state-migration functionality in realistic scenario's, as well as a detailed functional description of an proof-of-concept prototype of the complete programmability and orchestration framework in ESCAPEv2. The companion document, D3.2a, provides a refined description of the NF-FG data models related to UNIFY's key programming interfaces.

9 References

[Abley2007] J. Abley et al. "*Deprecation of Type 0 Routing Headers in IPv6*". RFC5095, December 2007.

[Alex2010] Alex X. Liu, Chad R. Meiners, and Eric Torng. "*TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs*". IEEE/ACM Trans. Netw., 18(2):490–500, April 2010.

[Anwer2013] Bilal Anwer, Theophilus Benson, Nick Feamster, Dave Levin, and Jennifer Rexford. "*A slick control plane for network middleboxes*". In Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking, pages 147–148. ACM, 2013.

[Ashwood2013] P. Ashwood-Smith, M. Soliman, and T. Wan. "*SDN state reduction*". draft-ashwood-sdnrg-state-reduction-00.txt, February 2013.

[Ashwood2014] Ashwood-Smith. "*Research challenges in Software-Defined Networking*". Infocom Panel, April 2014.

[Basta2014] Basta, Arsany, et al. "*Applying NFV and SDN to LTE mobile core gateways, the functions placement problem.*" Proceedings of the 4th workshop on All things cellular: operations, applications, & challenges. ACM. 2014.

[Basta 2013] Basta, Arsany, et al. "*A Virtual SDN-enabled LTE EPC Architecture: a case study for S-/P-Gateways functions.*" Future Networks and Services (SDN4FNS), 2013 IEEE SDN for. IEEE, 2013.

[Bazilchuk2005] Bazilchuk, N., Mohagheghi, P.: "*The Advantages of Reused Software Components*" in ERCIM News No. 60, January 2005, available: <http://www.ercim.eu/publication/Ercim_News/enw60/mohagheghi.html>

[Belbekkouche2012] Belbekkouche, A., Hasan, M. M., & Karmouch, A. (2012). "*Resource Discovery and Allocation in Network Virtualization.*" IEEE Communications Surveys & Tutorials, 14(4), 1114–1128. doi:10.1109/SURV.2011.122811.00060

[Bienkowsi2014] Bienkowski, M., Feldmann, A., Grassler, J., Schaffrath, G., and Schmid, S. "*The Wide-Area Virtual Service Migration Problem: A Competitive Analysis Approach*". IEEE/ACM Transactions on Networking, 22(1), 165–178, 2014.

[BIS] Department for Business Innovation & Skills, "*Business population estimates for the UK and regions 2012*", Accessed: 2015-04-27, <https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/80247/bpe-2012-stats-release-4.pdf>

[Bondan2014] Bondan, Lucas, Carlos Raniery Paula dos Santos, and Lisandro Zambenedetti Granville. "*Management Requirements for ClickOS-based Network Function Virtualization.*", Network and Service Management (CNSM), 2014 10th International Conference on. IEEE, 2014.

[BT21CN] Kitz, *"BT 21CN - Network Topology & Technology"*, Accessed: 2015-04-27, <http://www.kitz.co.uk/adsl/21cn_network.htm>

[Chiesa2014] Marco Chiesa, Gabriele Lospoto, Massimo Rimondini, and Giuseppe Di Battista. "*Intra-domain routing with pathlets*". Computer Communications, 46:76–86, 2014.

[Chowdhury2009] Chowdhury M, Rahman, & Boutaba, R. (2009). "*Virtual network embedding with coordinated node and link mapping*". In INFOCOM 2009, IEEE (pp.783-791).

[Clicky] "*Clicky*". Accessed: 2015-04-27, <http://www.read.cs.ucla.edu/click/clicky>.

[COMBO] FP7 COMBO, "*COMBO (COnvergence of fixed and Mobile BrOadband access/aggregation networks)*", Accessed: 2015-05-26, <http://www.ict-combo.eu>

[Csoma2014] Attila Csoma, Balázs Sonkoly, Levente Csikor, Felicián Németh, András Gulyás, Wouter Tavernier, Sahel Sahhaf, ESCAPE: "*Extensible Service ChAin Prototyping Environment using Mininet, Click, NETCONF and POX*". In Proceedings of ACM SIGCOMM 2014, pp. 125-126, Chicago, Illinois, USA, August 2014.

[D2.1] "*Deliverable 2.1:.Use Cases and initial architecture*" Tech. rep. UNIFY Project, 2014.

[D2.1 a] *"D2.1 Amendment: Stateful, Elastic Firewall use case."* Tech. rep. UNIFY Project, 2014.

[D2.2] "*Deliverable 2.2: Functional Architecture*" Tech. rep. UNIFY Project, 2014.

[D3.1] "*Deliverable 3.1: Programmability framework*". Tech. rep. UNIFY Project, 2014.

[D3.2a] "*Deliverable 3.2a: Network Function Forwarding Graph specification*", Tech. Rep. UNIFY Project 2015

[D5.2] "*Deliverable 5.2: API and Universal Node software architecture*". Tech. rep. UNIFY Project, 2014.

[D5.4] "*Deliverable 5.4: Initial Benchmarking documentation*". Tech. rep. UNIFY Project, 2014.

[DCMap], Data Center Map, "*Colocation United Kingdom – Data Centers*", Accessed: 2015-04-27, <http://www.datacentermap.com/united-kingdom/>

[DevoFlow] Curtis, Andrew R., et al. "*DevoFlow: Scaling flow management for high-performance networks.*" ACM SIGCOMM Computer Communication Review. Vol. 41. No. 4. ACM, 2011.

[Dietz2015] T. Dietz, R. Bifulco, F. Manco, J. Martins, HJ. Kolbe and F. Huici. "*Enhancing the BRAS through Virtualization*" 1st IEEE Conference on Network Softwarization (NETSOFT 2015). IEEE, 2015

[Dilip2008] Joseph, Dilip A., Arsalan Tavakoli, and Ion Stoica. "*A policy-aware switching layer for data centers.*" ACM SIGCOMM Computer Communication Review 38.4 (2008): 51-62.

[Docker] Docker, Inc. "*Docker - Build, Ship and Run Any App, Anywhere.*"Accessed: 2015-04-27. Docker homepage. <http://www.docker.com>.

[DockerComp] Docker Inc. "*Docker Compose* – Docker documentation". Accessed: 2015-04-27. Docker homepage. <http://docs.docker.com/compose/>.

[Esposito2014] Esposito, F., & Matta, I. (2014). "*A decomposition-based architecture for distributed virtual network embedding*". In Proceedings of the 2014 ACM SIGCOMM workshop on Distributed cloud computing - DCC '14 (pp. 53–58). New York, New York, USA: ACM Press. doi:10.1145/2627566.2627569

[ETSI NFV Use case] ETSI GS NFV 001, *"Network Functions Virtualisation (NFV); Use Cases",* Accessed 2015-05-07, <http://www.etsi.org/deliver/etsi_gs/nfv/001_099/001/01.01.01_60/gs_nfv001v010101p.pdf>

[Even2012] Even, G., Medina, M., Schaffrath, G., & Schmid, S. (2012). "*Competitive and deterministic embeddings of virtual networks*". Distributed Computing and Networking, pp. 106–121.

[Fischer2013] Fischer, A., Botero, J. F., Beck, M. T., de Meer, H., & Hesselbach, X. (2013)."*Virtual Network Embedding: A Survey*". IEEE Communications Surveys & Tutorials, 15(4), 1888–1906. doi:10.1109/SURV.2013.013013.00155

[Fischetti2004] Fischetti, Matteo, Carlo Polo, & Massimo Scantamburlo. "*A local branching heuristic for mixed-integer programs with 2-level variables, with an application to a telecommunication network design problem.*" Networks 44.2 (2004): 61-72.

[Flocker] ClusterHQ Inc. "*Docker container data & volume management*", Accessed 2015-04-27, <http://clusterhq.com>

[FlowNAC] Matias, Jon, et al. "*FlowNAC: Flow-based Network Access Control.*" Third European Workshop on Software Defined Networks. 2014.

[Foster2011] Foster, Nate, et al. "*Frenetic: A network programming language*", ACM SIGPLAN Notices. Vol. 46. No. 9. ACM, 2011. <http://www.frenetic-lang.org/overview.php>

[Garroppo2010] Garroppo , R., Iordano , S., & Tavanti , L. A "*Survey on Multi-constrained Optimal Path Computation: Exact and Approximate Algorithms*". Computation Networks. 54, 17 (2010), 3081–3107.

[Gember-Jacobson2013]Gember-Jacobson, Aaron, et al. "*Stratos: A network-aware orchestration layer for virtual middleboxes in clouds.*" arXiv preprint arXiv:1305.0209 (2013).

[Gember-Jacobson2014] Gember-Jacobson, Aaron, et al. "*OpenNF: Enabling innovation in network function control.*" Proceedings of the 2014 ACM conference on SIGCOMM. ACM, 2014.

[Godfrey2009] P. Brighten Godfrey, Igor Ganichev, Scott Shenker, and Ion Stoica. "*Pathlet routing*". In ACM SIGCOMM 2009, pages 111–122, 2009.

[GStreamer] "*GStreamer: open source multimedia framework", Accessed 2015-05-07,* <http://gstreamer.freedesktop.org/>

[Gupta1999] Pankaj Gupta and Nick McKeown."*Packet classification on multiple fields*". In ACM SIGCOMM 1999, pages 147–160, 1999.,

[Gurobi2015] Gurobi Optimization, Inc., "*Gurobi Optimizer Reference Manual*", retrieved 2015 from <http://www.gurobi.com>

[Hahn2015] Hahn, Wolfgang, and Borislava Gajic. "*GW elasticity in data centers: Options to adapt to changing traffic profiles in control and user plane.*" Intelligence in Next Generation Networks (ICIN), 2015 18th International Conference on. IEEE, 2015.

[Handley2005] Handley, Mark, et al. "*Designing extensible IP router software.*" Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2. USENIX Association, 2005.

[Hazelcast] "*Hazelcast.org – The Leading Open Source In-Memory Data Grid*", Accessed: 2015-04-27, <http://hazelcast.org>

[Hemminger2005] Hemminger, Stephen. "*Network emulation with NetEm.*" Linux conf au. 2005

[Houidi2011] Houidi, I., Louati, W., Ben Ameur, W., & Zeghlache, D. (2011). "*Virtual network provisioning across multiple substrate networks*". Computer Networks, 55(4), 1011–1023. doi:10.1016/j.comnet.2010.12.011

[ISPreview] ISPreview.co.uk, "*ISP BT Top 6.28M Broadband Customers as FTTC Passes 10M UK Homes",* Accessed: 2015-04-27, <http://www.ispreview.co.uk/index.php/2012/05/isp-bt-top-6-28m-broadband-customers-as-fttc-reaches-10m-uk-homes.html>

[JSONComp] "*Comparing various aspects of Serialization libraries*", Accessed: 2015-04-27, <https://code.google.com/p/thrift-protobuf-compare/wiki/BenchmarkingV2>

[Kanada2015] Kanada, Y. "*High-Level Portable Programming Language for Optimized Memory Use of Network Processors*", Communications and Network, 7, 55-69. (2015) doi: 10.4236/cn.2015.71006.

[Kang2013] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. "*Optimizing the "One Big Switch" abstraction in Software-defined Networks*". In Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT '13, pages 13–24, 2013.

[Kanizo2013] Yossi Kanizo, David Hay, and Isaac Keslassy. "*Palette: Distributing tables in software-defined networks*". In INFOCOM, pages 545–549, 2013.

[Kao2015]CN. Kao, S.SI, NF.Huang, I. Liao, RT. Liu and HW.Hung . "*Fast Proxyless Stream-Based Anti-Virus for Network Function Virtualization*". 1st IEEE Conference on Network Softwarization (NETSOFT 2015). IEEE, 2015

[Kohler2000] E. Kohler et al., "*The Click Modular Router*", ACM Trans. Computer Systems, vol. 18, no. 3, Aug. 2000, pp. 263-297. <http://www.read.cs.ucla.edu/click/click>

[Kolias2014] Christos Kolias. "*Bundling NFV and SDN for Open Networking and a call for NFV research*". Stanford University Networking Seminar, May 2014.

[Kolomicenko2013] Kolomicenko, V. "*Analysis and Experimental Comparison of Graph Databases*", Master thesis, <http://www.ksi.mff.cuni.cz/~holubova/dp/Kolomicenko.pdf>

[Koponen2011] Teemu Koponen, Scott Shenker, Hari Balakrishnan, Nick Feamster, Igor Ganichev, Ali Ghodsi, P Godfrey, Nick McKeown, Guru Parulkar, Barath Raghavan, et al. "*Architecting for innovation*". ACM SIGCOMM Computer Communication Review, 41(3):24–36, 2011.

[Korkmaz2001] Korkmaz, T., and Krunz, M. "*A randomized algorithm for finding a path subject to multiple QoS requirements*" Computer Networks, 36(2-3), 251–268, 2001.

[Krishna2004] Krishna P. Gummadi, Harsha V. Madhyastha, Steven D. Gribble, Henry M. Levy, and David Wetherall. "*Improving the reliability of Internet paths with one-hop source routing*". In Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation, OSDI'04, pages 13–13, 2004.

[Kubernetes] Google Inc. "Kubernetes by Google". Accessed 2015-04-27, <http://kubernetes.io>

[Kuzinar2014] Kuzniar, Maciej, Peter Peresini, and Dejan Kostic. "*What you need to know about SDN control and data planes*". No. EPFL-REPORT-199497. 2014.

[Lei2008] Lei, P., Ong, L., Tüxen, M., & Dreibholz, T. (2008). "*An overview of reliable server pooling protocols.*" IETF, Informational RFC, 5351, 2070-1721.

[Leonard2015b] Nobach, Leonhard och David Hausheer. "*Open, elastic provisioning of hardware acceleration in NFV environments.*" 2015 International Conference and Workshops on Networked Systems (NetSys). IEEE, 2015. 1-5.

[LINCX] FlowForwarding community, "*LINCX – OpenFlow software switch*", Accessed: 2015-05-22, <http://flowforwarding.github.io/lincx/>

[Martins2014] Martins, Joao, et al. "*ClickOS and the art of network function virtualization.*" Proc. USENIX NSDI. 2014. <http://cnp.neclab.eu/clickos/>

[McColl2014] McColl, Robert Campbell, et al. "*A performance evaluation of open source graph databases.*"Proceedings of the first workshop on Parallel programming for analytics applications. ACM, 2014. <http://www.stingergraph.com/data/uploads/papers/ppaa2014.pdf>

[MEF23.1] Metro Ethernet Forum, (2012). "*Implementation Agreement MEF 23.1. Carrier Ethernet Class of Service – Phase 2.*"

[Mehdi2010] D. Medhi. "*Network Routing: Algorithms, Protocols, and Architectures*". The Morgan Kaufmann Series in Networking. Elsevier Science, 2010.

[Melo2013] Melo, M., Sargento, S., Killat, U., Timm-Giel, A., & Carapinha, J. (2013). "*Optimal Virtual Network Embedding: Node-Link Formulation*", 10(4), 356–368. Retrieved from <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6616685>

[Menage2015], Paul. "*CGROUPS.*" Accessed: 2015-04-27. Linux Kernel Documentation. <http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt>.

[Mininet] Mininet Team, "*Mininet An Instant Virtual Network on your Laptop (or other PC)*", Accessed: 2015-04-27, <http://mininet.org>

[Monteleone2013] Monteleone, Giuseppe, and Pietro Paglierani. "*Session Border Controller Virtualization Towards" Service-Defined" Networks Based on NFV and SDN.*"Future Networks and Services (SDN4FNS), 2013 IEEE SDN for. IEEE, 2013.

[Mouradian2015] Mouradian, Carla, et al. "*NFV Based Gateways for Virtualized Wireless Sensors Networks: A Case Study.*" arXiv preprint arXiv:1503.05280 (2015).

[Nascimento2011] Nascimento, Marcelo R., et al. "*Virtual routers as a service: the routeflow approach leveraging software-defined networks.*", Proceedings of the 6th International Conference on Future Internet Technologies. ACM, 2011. <https://sites.google.com/site/routeflow/>

[Neo4j] "*Neo4j, The world's leading graph database*", Accessed: 2015-04-27, <http://www.neo4j.com>

[NETMAP] The netmap project, "*netmap – the fast packet I/O framework*", Accessed: 2015-05-22, <http://info.iet.unipi.it/~luigi/netmap/>

[Nobach2015] Nobach, Leonhard, Hausheer, David. "*Open, elastic provisioning of hardware acceleration in NFV environments.*" Networked Systems (NetSys), 2015 International Conference and Workshops on. IEEE, 2015.

[OrientDB] "*OrientDB Multi-Model NoSQL Database*", Accessed: 2015-04-27,<http://orientdb.com>

[POX] NOXRepo, "*About POX | NOXrepo*", Accessed: 2015-04-27, < http://www.noxrepo.org/site/about/>

[Previdi2014a] S. Previdi et al. "*IPv6 Segment Routing Header*". draft-previdi-6man-segment-routing-header-01, December 2014.

[Previdi2014b] S. Previdi et al. "*SPRING Problem Statement and Requirements*". draft-previdi-spring-problem-statement-00.txt, August 2014.

[Protobuf] Google Inc, “Protocol Buffers – Google’s data interchange format”, Accessed: 2015-05-23, <https://github.com/google/protobuf>

[Qazi2013] Qazi, Zafar Ayyub, et al. "*SIMPLE-fying middlebox policy enforcement using SDN.*" ACM SIGCOMM Computer Communication Review. Vol. 43. No. 4. ACM, 2013.

[Quagga] “*Quagga Routing Suite*” Accessed: 2015-04-27,<http://www.quagga.net/>.

[Raghavan2012] Barath Raghavan, Martín Casado, Teemu Koponen, Sylvia Ratnasamy, Ali Ghodsi, and Scott Shenker. “*Software-defined internet architecture: decoupling architecture from infrastructure*”. In Proceedings of the 11th ACM Workshop on Hot Topics in Networks, pages 43–48. ACM, 2012.

[Rajagopalan2013a] Rajagopalan, Shriram, et al. "*Split/Merge: System Support for Elastic Execution in Virtual Middleboxes.*" NSDI. 2013

[Rajagopalan2013b] Rajagopalan, Shriram, Dan Williams, and Hani Jamjoom. "*Pico Replication: A high availability framework for middleboxes.*" Proceedings of the 4th annual Symposium on Cloud Computing. ACM, 2013

[Retvari2013] Gábor Rétvári, János Tapolcai, Attila Kőrösi, András Majdán, and Zalán Heszberger. “*Compressing IP forwarding tables: towards entropy bounds and beyond*”. In ACM SIGCOMM 2013, pages 111–122, 2013.

[Ricci2003] Ricci, R., Alfeld, C., & Lepreau, J. (2003). “*A solver for the network testbed mapping problem*”. ACM SIGCOMM Computer Communication Review, 33(2), 65–81.

[Rost2014] Rost, M., Schmid, S., & Feldmann, A. (2014). “*It’s About Time: On Optimal Virtual Network Embeddings under Temporal Flexibilities*”. In 2014 IEEE 28th International Parallel and Distributed Processing Symposium (pp. 17–26). IEEE. doi:10.1109/IPDPS.2014.14

[Rottenstreich2013] Ori Rottenstreich, Marat Radan, Yuval Cassuto, Isaac Keslassy, Carmi Arad, Tal Mizrahi, Yoram Revah, and Avinatan Hassidim. “*Compressing forwarding tables*”. In INFOCOM, pages 1231–1239, 2013.

[Sahhaf2015a] Sahel Sahhaf, Wouter Tavernier, Didier Colle, Mario Pickavet, “*Network Service chaining with efficient network function mapping based on service decompositions*”, NetSoft 2015.

[Sahhaf2015b] Sahel Sahhaf, Wouter Tavernier, Matthias Rost, Stefan Schmid, Didier Colle, Mario Pickavet, Piet Demeester, “*Network service chaining with optimized network function embedding supporting service decompositions*”, submitted to Computer Networks.

[Salim2013] Jouili, Salim, and Valentin Vansteenberghe. "*An empirical comparison of graph databases.*" Social Computing (SocialCom), 2013 International Conference on. IEEE, 2013. <http://euranova.eu/upl_docs/publications/an-empirical-comparison-of-graph-databases.pdf**>**

[Schaffrath2012] Schaffrath, G., Schmid, S., Vaishnavi, I., Khan, A., & Feldmann, A, "*A Resource Description Language with Vagueness Support for Multi-Provider Cloud Networks*". In Computer Communications and Networks (ICCCN), 2012.

[Shah2004] N., Plishker, W., Ravindran, K. and Keutzer, K. (2004) "*NP-Click: A Productive Software Development Approach for Network Processors*". IEEE Micro, 24, 45-54 <http://dx.doi.org/10.1109/mm.2004.53>

[SPARCD2.1] SPARC D2.1, "*Initial Definition of use cases and carrier requirements*". Accessed 2015-05-20, <http://www.fp7-sparc.eu/assets/deliverables/ SPARC_D2.1_Initial_Defintion_of_use_cases_and_carrier_requirements_v3.0.pdf>

[SocketPlane] SocketPlane Inc. "*socketplane*", Accessed: 2015-04-27, SocketPlane homepage, <http://socketplane.io/>

[Soulé2013] Soulé, R., Basu, S., & Kleinberg, R. (2013). "*Managing the network with Merlin*". Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks, pp. 1–7.

[Swarm] Docker Inc. "*Docker Swarm – Docker documentation*". Accessed 2015-04-27, <https://docs.docker.com/swarm/>

[Tapolcai2012] János Tapolcai, András Gulyás, Zalán Heszberger, József Bíró, Péter Babarczi, and Dirk Trossen. "*Stateless multi-stage dissemination of information: Source routing revisited*". In GLOBECOM, pages 2797–2802, 2012.

[Tcpr] "*Tcpreplay: Pcap editing and replay tools for *nix*", Accessed: 2015-04-27.< http://tcpreplay.synfin.net>

[Titan] "*Titan: Distributed graph database*", Accessed: 2015-04-27, <http://thinkaurelius.github.io/titan/>

[Tonse] Tonse Telecom Pvt. Ltd., "*Service Provider Profile – British Telecom*", Accessed: 2015-04-27, <http://www.tonsetelecom.com/Downloads/Tonse_British%20Telecom_Operator%20Profile.pdf>

[Topology] "*The Internet Topology Zoo*", Accessed: 2015-04-27, <http://www.topology-zoo.org>

[UBM] UBM Tech, "*BT Increases Profits, Broadband Market Share*", Accessed: 2015-04-27, <http://www.informationweek.com/business/bt-increases-profits-broadband-market-share/d/d-id/1109901>

[Vasseur2004] Vasseur, J. P., Pickavet, M., & Demeester, P. (2004). "*Network recovery: Protection and Restoration of Optical, SONET-SDH, IP, and MPLS.*", Elsevier.

[Wang2001] Yufei Wang, Zheng Wang, and Leah Zhang. "*Internet Traffic Engineering without full mesh overlaying*". In INFOCOM, pages 565–571, 2001.

[Wang2014] Wang, An, et al. *"Scotch: Elastically Scaling up SDN Control-Plane using vSwitch based Overlay.",* Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies. ACM, 2014.

[Yin2012] Yin, Qin, and Timothy Roscoe. "*VF2x: fast, efficient virtual network mapping for real testbed workloads.*" Testbeds and Research Infrastructure. Development of Networks and Communities. Springer Berlin Heidelberg, 2012. 271-286.

[Yu2010] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. "*Scalable flow-based networking with DIFANE*". In ACM SIGCOMM 2010, pages 351–362, 2010.

[Zawawy2014a] El-Zawawy, Mohamed A., and Adel I. AlSalem. *"ImNet: An Imperative Network Programming Language.*", arXiv preprint arXiv:1403.8028 (2014).

[Zawawy2014b] El-Zawawy, Mohamed A. "*ConNet: A Network Programming Language with Concurrency.*", Computational Science and Its Applications (ICCSA), 2014 14th International Conference on. IEEE, 2014.

Annex 1 Main components of ESCAPEv2

This annex introduces a more-detailed description of the latest version of ESCAPE, called ESCAPEv2. The architecture of the former framework was reconsidered and redesigned to match the distinct system components to the layers of the final UNIFY architecture more accurately and ensure a generalized and easily replaceable way for communication between the loosely coupled components. Due to the modular design, loosely coupled communication approach and well-defined interfaces based on Virtualizer elements, ESCAPEv2 allows for adapting custom architecture parts and algorithms seamlessly to maximally benefit from the scalable, recursively layered UNIFY architecture. The functionalities of ESCAPEv2 framework was separated into three different components following the sublayers of UNIFY's Service and Orchestration layers. More exactly, we have defined Service, Orchestration and Adaptation modules, respectively. The following subsections give details on these components (or modules as called in the context of POX). First, the static architectural model of a given module will be presented with the class structure of the internal building elements. Second, the cooperation of the previously introduced architectural parts and the interaction steps between them will be described through a specific case study. This example will show the top-down process of a UNIFY Service Request and the main steps. The process is divided in three large parts following the logical structure of ESCAPEv2 sublayers.

A.1.1 Static model of Service module

The Service component represents the Service (Graph) Adaptation Sublayer (SAS) of UNIFY's Service layer described in [D2.2]. In order to focus on the core Service Chaining Mapping process, the ESCAPEv2 framework is without the supplementary management part of the Service layer called Service Management Sublayer. However, the SAS additionally contains a REST API on top of the layer for unified communication with upper components such as UNIFY actors via a GUI or other standalone applications. The static class structure of this main module is shown in Figure 9-1.

One of the main logical parts of Service module is the REST API. The REST API is responsible for offer a unified interface based on the HTTP protocol and the REST design approach. The *RESTServer* class encompasses this functionality by run a self-implemented HTTP server in a different thread and initialize the *ServiceRequestHandler* class which defines the interface of the relevant UNIFY reference point (namely the U– Sl interface). The class consists of entirely the abstract UNIFY interface functions therefore it can be replaced without the need to replace or modify other components.

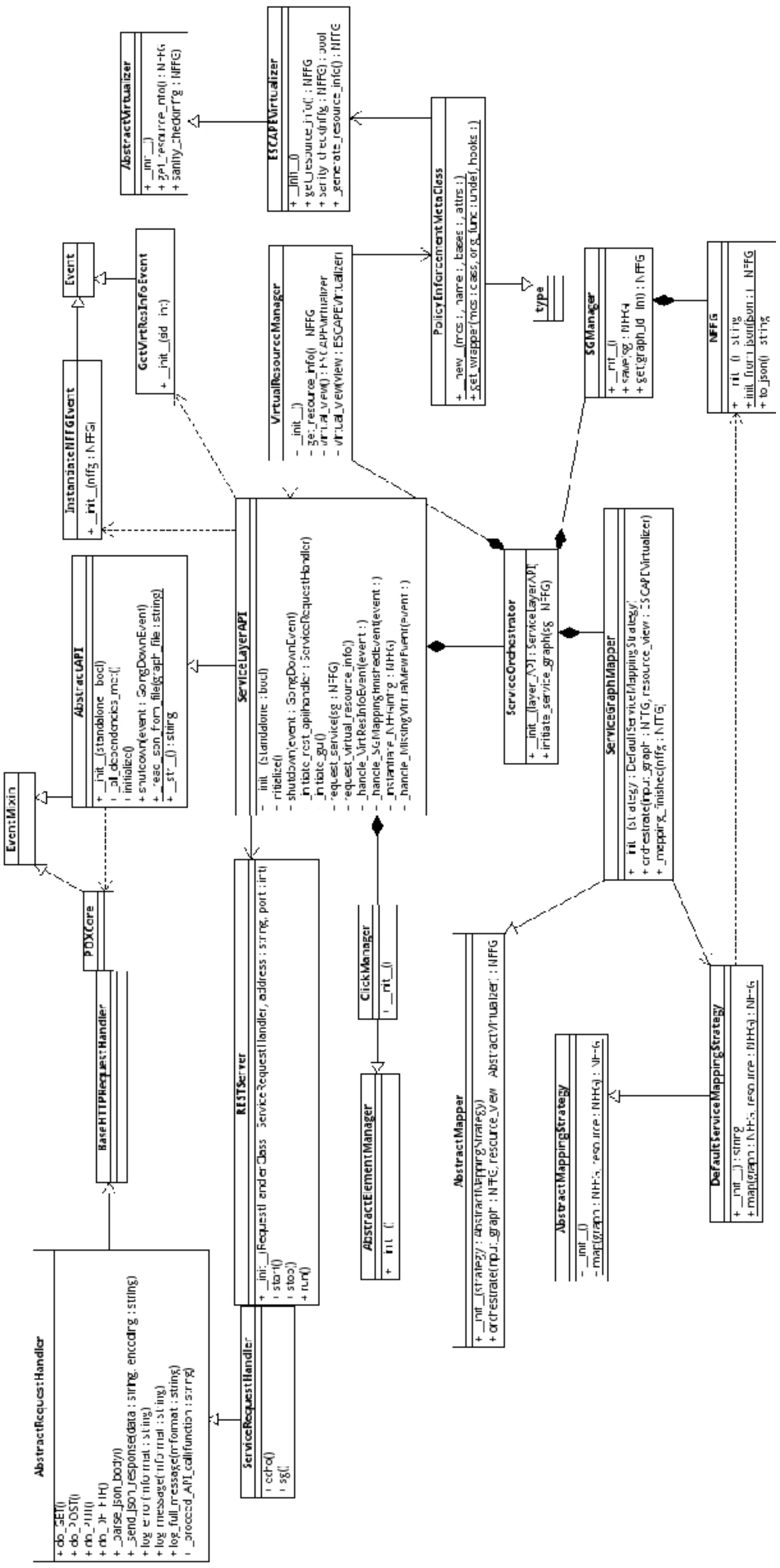


Figure 9-1: Class diagram of the Service module

In order to separate the UNIFY's API from the REST behaviour the general functionality of HTTP request handling is defined in an abstract class called *AbstractRequestHandler*. This class contains the basic common functions which

- parse the HTTP requests,
- split and interpret the URLs according to the REST approach to determine the UNIFY API function need to be called,
- parse the optional HTTP body as the parameter with respect to security requirements,
- and delegate the request process to the actual module-level API function with the processed parameters in a common form (as an NF-FG).

The other main part of Service module represents the Service Adaptation Sublayer. The main entry and exit point is the *ServiceLayerAPI* class. This element realizes the actual interface of the SAS sublayer and proceeds the calls comes from external source (e.g. REST API, file, other modules) to the appropriate subcomponents.

The general behaviour for each ESCAPEv2 top-layer module is defined in the *AbstractAPI* class. This class contains the

- basic and general steps related to the control of module's life cycle,
- definition of dependencies on other components,
- initiation and tear down of internal elements,
- general interface for interaction with other modules and external actors,
- and the management of communication between internal elements.

According to these functions the role of the actual API classes derived from *AbstractAPI* is threefold. First, it hides the implementation and behaviour of POX to make the modules' implementation more portable and easily changeable. Second, it handles the module dependencies to grant a consistent initialization process.

Third, it handles the event-driven communication between modules so internal elements only have to know and use the common functions of the derived *AbstractAPI* class defined in each top-level module. Furthermore, with these functionalities provided by the *AbstractAPI* base class the main layer modules of ESCAPEv2 can achieve a loosely coupled, transparent communication and easily adjustable module structure.

The central point of the Service layer is the *ServiceOrchestrator* class. It initializes, contains and handles the internal elements which are involved in the top part of the UNIFY Service Chaining process. This class also supervises the supplementary functions related to the Service orchestration such as managing, storing and handling Service Graphs, handling virtual resource information and choosing the algorithm for service-level mapping.

The Service Graph managing functionality is realized by the *SGManager* wrapper element which offers a common interface for handling and storing Service Graphs in a platform and technology independent way. The format in which the Service Graphs are stored is a multipurpose NF-FG container class, called *NFFG*. Moreover, every graph representation, e.g. NF-FGs, Service Graphs and resource information is stored and used for algorithms with the help of this wrapper class.

The *VirtualResourceManager* class handles the virtual resource information assigned to the Service module in the same way as the *SGManager* for Service Graphs. In the background the resource information is not stored in *NFFG*. Instead of the Manager class have a reference to a dedicated Virtualizer element, which can generate the resource information on the fly. Due to the wrapper classes the storing format can be modified easily to use only *NFFG* representation and a fully separated module design can be achieved. This manager class as all *Manager* classes in ESCAPEv2 hides the actual format of the stored resources and provides the opportunity to change its implementation transparently.

The orchestration steps are encompassed by the *ServiceGraphMapper* class, which pre-processes and verifies the given information and provide it in the appropriate format for the mapping algorithm.

The mapping algorithm is defined in a separate element for simplicity and clarity. The trivial service-level mapping algorithm, which use a simple BiS-BiS view as the abstract resource information is contained by the *DefaultServiceMappingStrategy* class. The general interfaces for the mapper and strategy classes are defined in the *AbstractMapper* and *AbstractStrategy* classes.

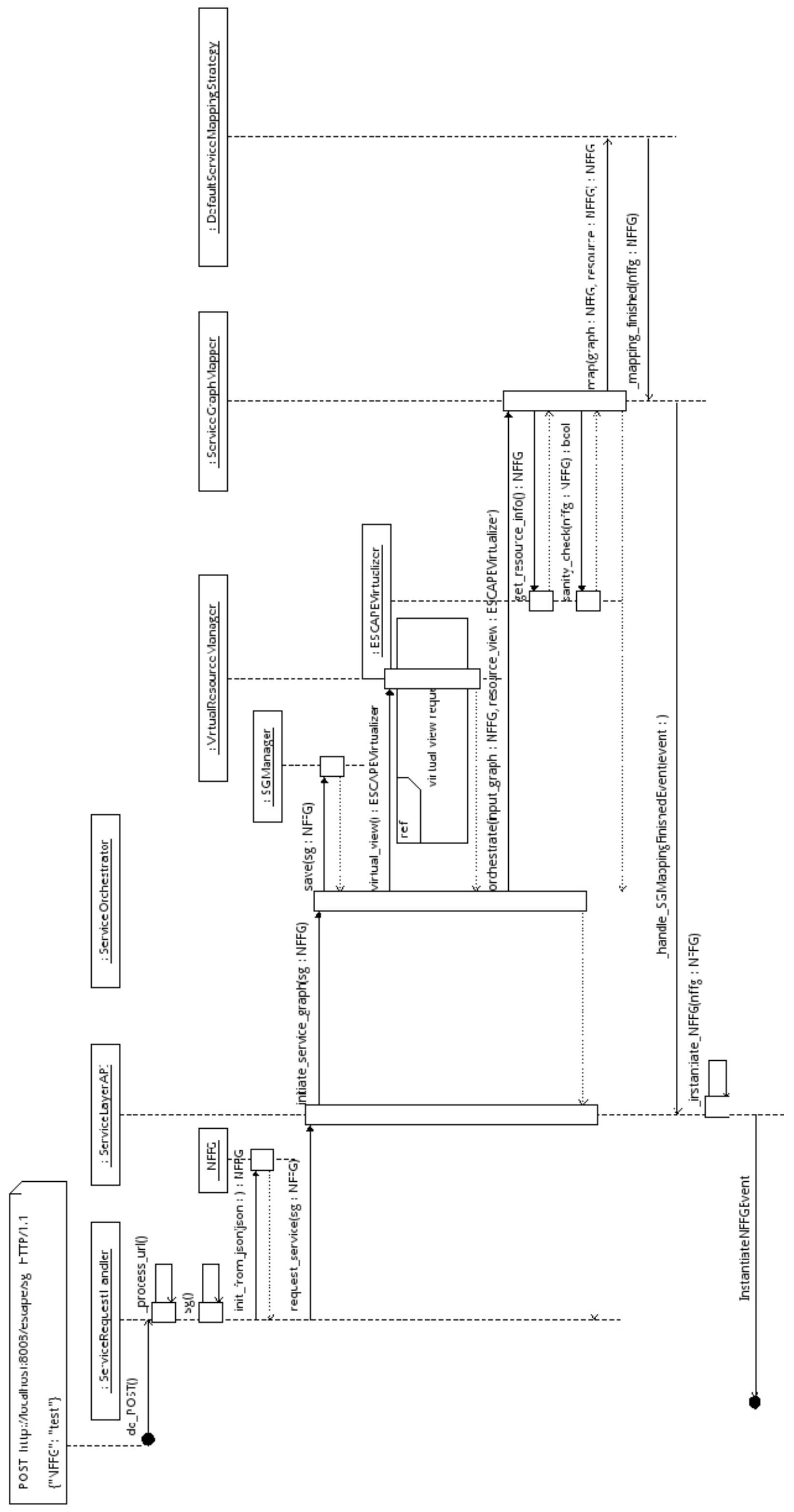
The communication between the elements inside the modules is based on events. The *Event* classes in the layer components represent the different stages during the ESCAPEv2 processes. The event-driven communications relies on POX's own event handling mechanism, but every communication primitive is attached to well-defined functions for the purpose of supporting other asynchronous communication forms, e.g. different implementations of event-driven communication based on Observer design pattern, Asynchronous Queuing or Message Bus architecture based on ZeroMQ.

A.1.2 Dynamic model of Service module

The first part of the Service Requesting example is shown in Figure 9-2.

1. The Service Request is provided via the REST API in a HTTP message. The function is defined in the URL (the general *sg* function along with POST HTTP verb) with a formatted body as an NF-FG in JSON format.
2. The message is processed; the optional parameters are parsed and converted concerning the HTTP verb and delegated to the *sg()* function which is part of the UNIFY U – SI API representation in the *ServiceRequestHandler* class.

3. The main *ServiceLayerAPI* class receives the UNIFY API call and forwards to the central *ServiceOrchestrator* element.
4. The orchestrator saves the generated Service Graph in the *SGManager* with internal *NFFG* format, obtains the resource information via the V*irtualResourceManager* and invokes the *SGMapper* in order to start the Service graph mapping process.
5. The *SGMapper* requests the resource information from the given *ESCAPEVirtualizer* in the *NFFG* format, validates the Service Graph against the resource info in respect of sanity and syntax and invokes the actual mapping algorithm of the *DefaultServiceMappingStrategy*.
6. After the mapping process is finished, the actual Strategy element returns the outcome in a *SGMappingFinished* event, which is processed by the module API class and proceeds the given NF-FG to the lower layer for instantiation via a general function. The instantiation notification is performed via the *InstantiateNFFGEvent* class.



POST http://localhost:8008/escape/sg HTTP/1.1
{"NFFG": "test"}
: ServiceRequestHandler
: ServiceLayerAPI
: ServiceOrchestrator
: VirtualResourceManager
: ServiceGraphMapper
: DefaultServiceMappingStrategy
: SGManager
: ESCAPEVirtualizer
: NFFG
do_POST()
_process_url()
sg()
init_from_json(json :) : NFFG
request_service(sg : NFFG)
initiate_service_graph(sg : NFFG)
save(sg : NFFG)
virtual_view() : ESCAPEVirtualizer
ref
virtual view request
orchestrate(input_graph : NFFG, resource_view : ESCAPEVirtualizer)
get_resource_info() : NFFG
sanity_check(nffg : NFFG) : bool
map(graph : NFFG, resource : NFFG) : NFFG
_mapping_finished(nffg : NFFG)
_handle_SGMappingFinishedEvent(event :)
_instantiate_NFFG(nffg : NFFG)
InstantiateNFFGEvent

Figure 9-2: Interaction steps during a Service Request in the Service module

A.1.3 Static model of Orchestration module

The Orchestration component represents the Resource Orchestration Sublayer (ROS) of UNIFY's Orchestration layer. The communication with the upper and lower layer is managed by the POX event mechanism as in the case of the Service module component. The static class structure of this main module is shown in Figure 9-3.The structure of this module, the separation of internal component and its connections to each other are designed in compliance with the Service module as precisely as possible in order to support the transparency and consistency of the ESCAPEv2 architecture.

The main interface of the Orchestration module is realized by the *ResourceOrchestrationAPI* class. Its function and responsibility are identical to the *ServiceLayerAPI* in the previous section, namely managing the module's life cycle, handling internal and external communications and translate calls from events to class functions, etc. Based on the external event-driven communication the *ResourceOrchestrationAPI* realizes the relevant Sl – Or interface of UNIFY functional architecture.

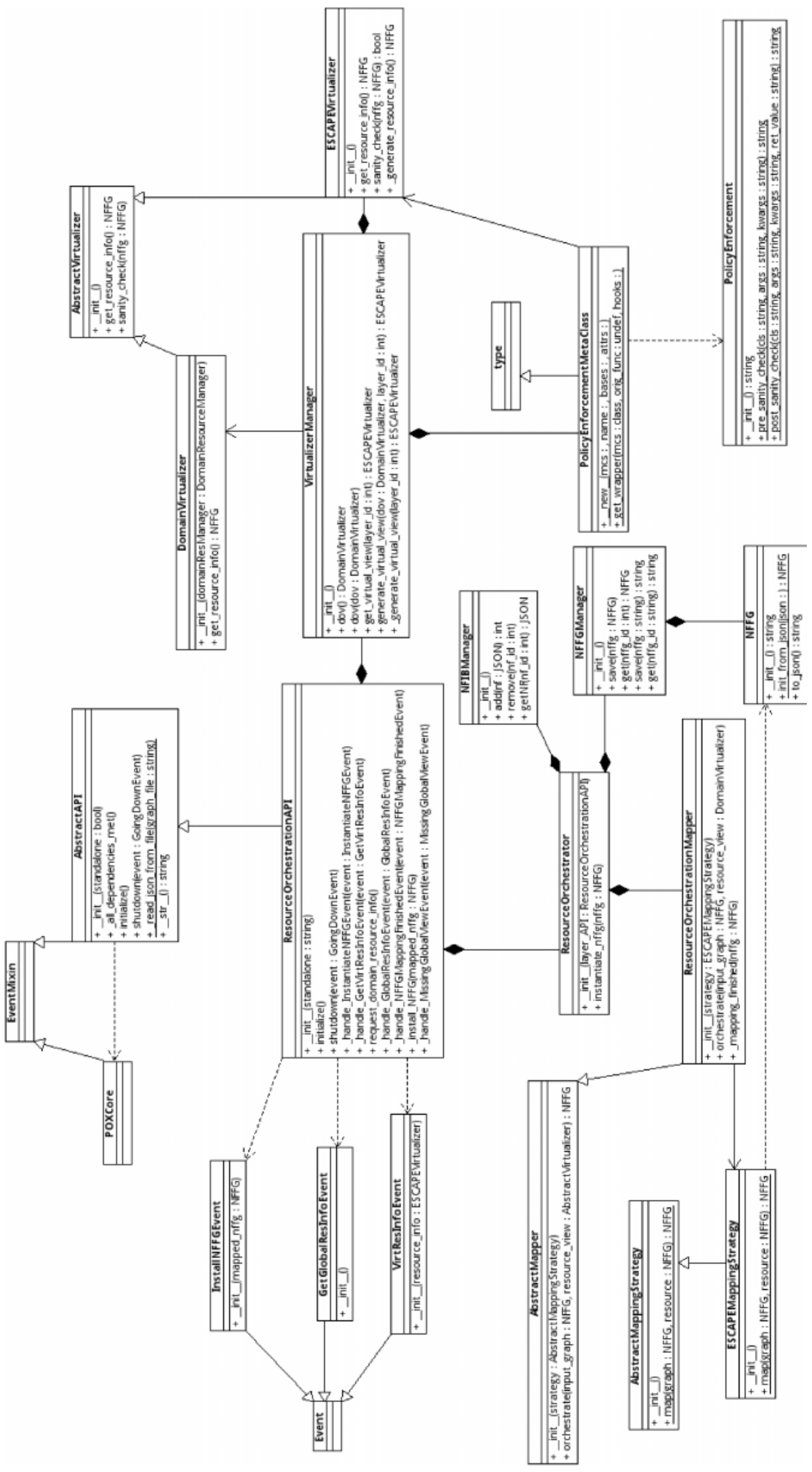
The central component of this module is the *ResourceOrchestrator* responsible for orchestration at the level of global domain view. It initializes, contains and controls internal module elements and gathers needed information similarly to the *ServiceOrchestrator* class.

The management of requested and already installed NF-FG instances is performed by the *NFFGManager* class. The manager class uses the *NFFG* wrapper as the storage format.

The orchestration steps are encompassed by the *ResourceOrchestrationMapper* class which have the same responsibilities as the Mapper class inherited from the *AbstractMapper* base class in the Service module. The layer-level mapping algorithm which do the actual mapping using the generated NF-FG and actual resource information is defined in a derived *AbstractStrategy* class, namely in the *ESCAPEMappingStrategy* similarly as before. The network function descriptions can be requested via a wrapper class, i.e., the *NFIBManager*, which hides the implementation characteristics and offers a platform-independent interface. This manager class is provided for the orchestration-level mapping algorithm by default.

The only major difference from the Service module appears on the handling of the virtual resources views. The orchestration module is responsible for the creation, assignment and storing of virtual resource views. The functionality of these virtual views is encompassed by the *AbstractVirtualizer* base class. This class offers a general interface for the interaction with the actual Virtualizers and contains the common functions such as generating the virtual resource information into the internal *NFFG* representation. The derived classes of the *AbstractVirtualizer* represents the different kind of Virtualizers defined in the UNIFY architecture and contains the metadata for the resource information filtering. The *ESCAPEVirtualizer* class represents the Virtualizer component assigned to the upper layer(s). The *DomainVirtualizer* (DoV) class represents the abstract global resource view, which is created by and requested from the lower layer. The Virtualizer instances are managed and created by the *VirtualizerManager* class. This manager class also stores the *DomainVirtualizer* instance which is used for the creation of the virtual views.

The policy enforcement functions which are closely related to the Virtualizers are defined in the *PolicyEnforcement* class. This class consists of entirely the enforcement and checking functions. In every case when a derived *AbstractVirtualizer* instance is created the *PolicyEnforcement* class is attached to that Virtualizer in order to set up the policy related functionality automatically. The attachment is performed by the *PolicyEnforcementMetaClass*. The policy enforcement functionality which realized by the previous classes follows the Filter Chain approach associated with the functions of the Virtualizers. That design allows defining and attaching a checking or enforcing function before and/or after the involved function of a Virtualizer is invoked by other internal module components.



ESCAPEVirtualizer
+ __init__()
+ get_resource_info() : NFFG
+ sanity_check(nffg : NFFG) : bool
+ _generate_resource_info() : NFFG
AbstractVirtualizer
+ __init__()
+ get_resource_info() : NFFG
+ sanity_check(nffg : NFFG)
DomainVirtualizer
+ __init__(domainResManager : DomainResourceManager)
+ get_resource_info() : NFFG
VirtualizerManager
+ __init__()
+ dov() : DomainVirtualizer
+ dov(dov : DomainVirtualizer)
+ get_virtual_view(layer_id : int) : ESCAPEVirtualizer
+ generate_virtual_view(dov : DomainVirtualizer, layer_id : int) : ESCAPEVirtualizer
+ _generate_virtual_view(layer_id : int) : ESCAPEVirtualizer
type
PolicyEnforcementMetaClass
+ __new__(mcs : , name : , bases : , attrs :)
+ get_wrapper(mcs : class, orig_func : undef, hooks :)
PolicyEnforcement
+ __init__() : string
+ pre_sanity_check(cls : string, args : string, kwargs : string) : string
+ post_sanity_check(cls : string, args : string, kwargs : string, ret_value : string) : string
NFIBManager
+ __init__()
+ add(nf : JSON) : int
+ remove(nf_id : int)
+ getNF(nf_id : int) : JSON
NFFGManager
+ __init__()
+ save(nffg : NFFG)
+ get(nffg_id : int) : NFFG
+ save(nffg : string) : string
+ get(nffg_id : string) : string
NFFG
+ __init__() : string
+ init_from_json(json :) : NFFG
+ to_json() : string
AbstractAPI
+ __init__(standalone : bool)
+ _all_dependencies_met()
+ initialize()
+ shutdown(event : GoingDownEvent)
+ _read_json_from_file(graph_file : string)
+ __str__() : string
EventMixin
POXCore
ResourceOrchestrationAPI
+ __init__(standalone : string)
+ initialize()
+ shutdown(event : GoingDownEvent)
+ _handle_InstantiateNFFGEvent(event : InstantiateNFFGEvent)
+ _handle_GetVirtResInfoEvent(event : GetVirtResInfoEvent)
+ request_domain_resource_info()
+ _handle_GlobalResInfoEvent(event : GlobalResInfoEvent)
+ _handle_NFFGMappingFinishedEvent(event : NFFGMappingFinishedEvent)
+ install_NFFG(mapped_nffg : NFFG)
+ _handle_MissingGlobalViewEvent(event : MissingGlobalViewEvent)
ResourceOrchestrator
+ __init__(layer_API : ResourceOrchestrationAPI)
+ instantiate_nffg(nffg : NFFG)
ResourceOrchestrationMapper
+ __init__(strategy : ESCAPEMappingStrategy)
+ orchestrate(input_graph : NFFG, resource_view : DomainVirtualizer)
+ _mapping_finished(nffg : NFFG)
InstallNFFGEvent
+ __init__(mapped_nffg : NFFG)
GetGlobalResInfoEvent
+ __init__()
VirtResInfoEvent
+ __init__(resource_info : ESCAPEVirtualizer)
Event
AbstractMapper
+ __init__(strategy : AbstractMappingStrategy)
+ orchestrate(input_graph : NFFG, resource_view : AbstractVirtualizer) : NFFG
AbstractMappingStrategy
+ __init__()
+ map(graph : NFFG, resource : NFFG) : NFFG
ESCAPEMappingStrategy
+ __init__()
+ map(graph : NFFG, resource : NFFG) : NFFG

Figure 9-3: Class diagram of the Orchestration module

A.1.4 Dynamic model of Orchestration module

The second part of the Service Requesting process is shown in Figure 9-4.The input parameter is the event which is raised by the *ServiceLayerAPI* in the end of the previous dynamic model.

1. The triggering event called *InstantiateNFFGEvent* is handled by the *ResourceOrchestrationAPI* class which is the communication point between the internal components and other top modules. The event contains the NF-FG generated by the Service module. Based on the type of the event a dedicated event handler is invoked. These handlers in the actual top module class represent the UNIFY SI-Or API.

2. The request is delegated to the central *ResourceOrchestrator* via the corresponding API function. In case of this Service Request example the invoked function is *instantiate_nffg()*.

3. The orchestrator saves the generated NF-FG using the NFFGManager wrapper (in the trivial format of the internal NFFG); obtains the global resource view as a DomainVirtualizer by invoking the VirtualizerManager class.

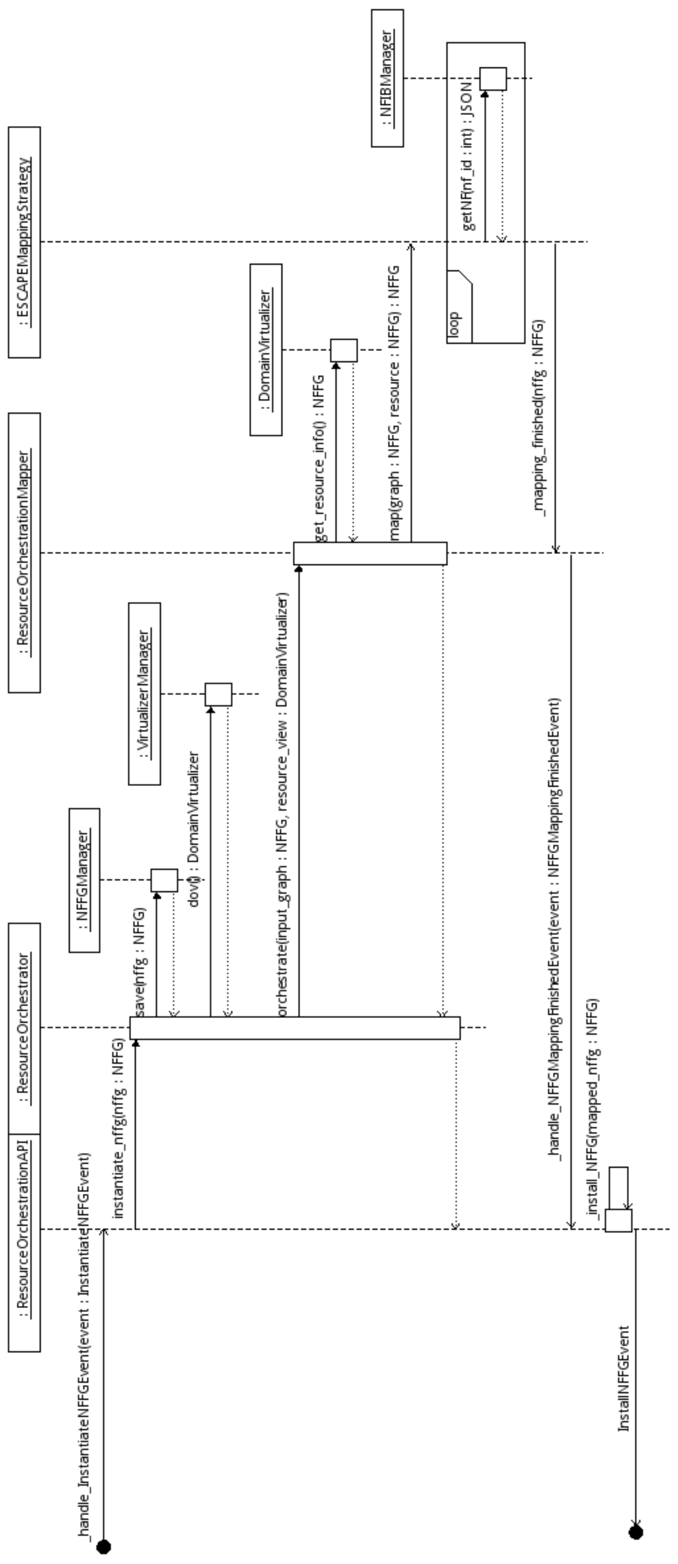
4. After the *ResourceOrchestrator* did the preparations, it invokes the *orchestration()* function of the *ResourceOrchestrationMapper* class in order to initiate the NF-FG mapping process.

5. The orchestration process requests the global resource information via the *DomainVirtualizer* and invokes the actual mapping algorithm of the *ESCAPEMappingStrategy*. The validation of the inputs of the mapping algorithm can be performed by the *ResourceOrchestrator* also.

6. The *ESCAPEMappingStrategy* uses the *NFIBManager* to run the algorithm and returns with the mapped NF-FG in the common *NFFG* format in an asynchronous way with the help of the *NFFGMappingFinishedEvent*.

7. The event is handled by the *ResourceOrchestrationAPI* class and it proceeds the on-going Service Request by invoking a general communication function.

8. The mapped NF-FG is forwarded to the lower layer via the *InstallNFFGEvent*.



: ResourceOrchestrationAPI
: ResourceOrchestrator
: NFFGManager
: VirtualizerManager
: ResourceOrchestrationMapper
: DomainVirtualizer
: ESCAPEMappingStrategy
: NFIBManager
_handle_InstantiateNFFGEvent(event : InstantiateNFFGEvent)
instantiate_nffg(nffg : NFFG)
save(nffg : NFFG)
dov() : DomainVirtualizer
orchestrate(input_graph : NFFG, resource_view : DomainVirtualizer)
get_resource_info() : NFFG
map(graph : NFFG, resource : NFFG) : NFFG
loop
getNF(nf_id : int) : JSON
_mapping_finished(nffg : NFFG)
_handle_NFFGMappingFinishedEvent(event : NFFGMappingFinishedEvent)
_install_NFFG(mapped_nffg : NFFG)
InstallNFFGEvent

Figure 9-4: Interaction steps during a Service Request in the Orchestration module

A.1.5 Static model of Adaptation module

The Adaptation component represents the Controller Adaptation Sublayer (CAS) of UNIFY's Orchestration layer. This module contains no other dedicated auxiliary component. The communication with upper layer is managed by the POX event mechanism as in the case of the Service module and Orchestration module, respectively. The static class structure of the Adaptation module is shown in Figure 9-5. The structure of this module is similar in the main lines with the previously mentioned top API modules.

The main interface of the Adaptation module is realized by the *ControllerAdaptationAPI*. Its functions and responsibilities are identical to the other top API classes derived from *AbstractAPI*. As the *ResourceOrchestrationAPI* class (in the context of the Orchestration module) this top API class also realizes the corresponding UNIFY reference point functions (namely the Or-Ca interface).

The central component of this module is the *ControllerAdapter*. It initializes, contains and handles the internal module elements. The purpose of the ControllerAdapter can be split into two major parts:

First, it handles the incoming NF-FG instances which are mapped and send down by the upper Orchestration module. For this task, the *ControllerAdapter* contains the functions for processing the mapped NF-FG instances, splitting into subsets of NF-FG descriptions based on the initiated domain adapters and controlling the installation of the subparts per domain. The graph representation in this module is also managed in the internal *NFFG* format.

Second, it handles the domain changes originated from lower layer which represents the UNIFY Infrastructure layer. For this task, the *ControllerAdapter* initiates and manages arbitrary domain adapters derived from the *AbstractDomainAdapter* base class. These adapters implement their controller managing function in an implementation-agnostic way, e.g. using a REST API, NETCONF protocol, etc. This base class defines the common management functionality and also offers a general interface for the *ControllerAdapter*. ESCAPEv2 providently defines three adapter skeletons:

- *POXAdapter* which is a simple adapter to the POX core functionality and to the OpenFlow controller implemented inside the POX. In this context the *POXAdapter* realizes the UNIFY Ca – Co interface in a built-in way.
- *MininetAdapter* for interacting with an SDN network (as the infrastructure layer) emulated by the Mininet tool.
- *OpenStackAdapter* for interacting with the cloud infrastructure managed by OpenStack.

The domain / topology information from the lower layers is stored via the *DomainResourceManager* wrapper class which managed by the *ControllerAdapter* too. The *ControllerAdapter* means the connection between the domain adapters and the domain resource database.

The *DomainResourceManager* class offers an abstract global view of the provisioned elements hiding the physical characteristics via the *DomainVirtualizer* class. The *DomainVirtualizer* is inherited from the *AbstractVirtualizer* therefore the common Virtualizer interface can be used to interact with that element. The abstract global resource information is generated by *DomainVirtualizer* on the fly.

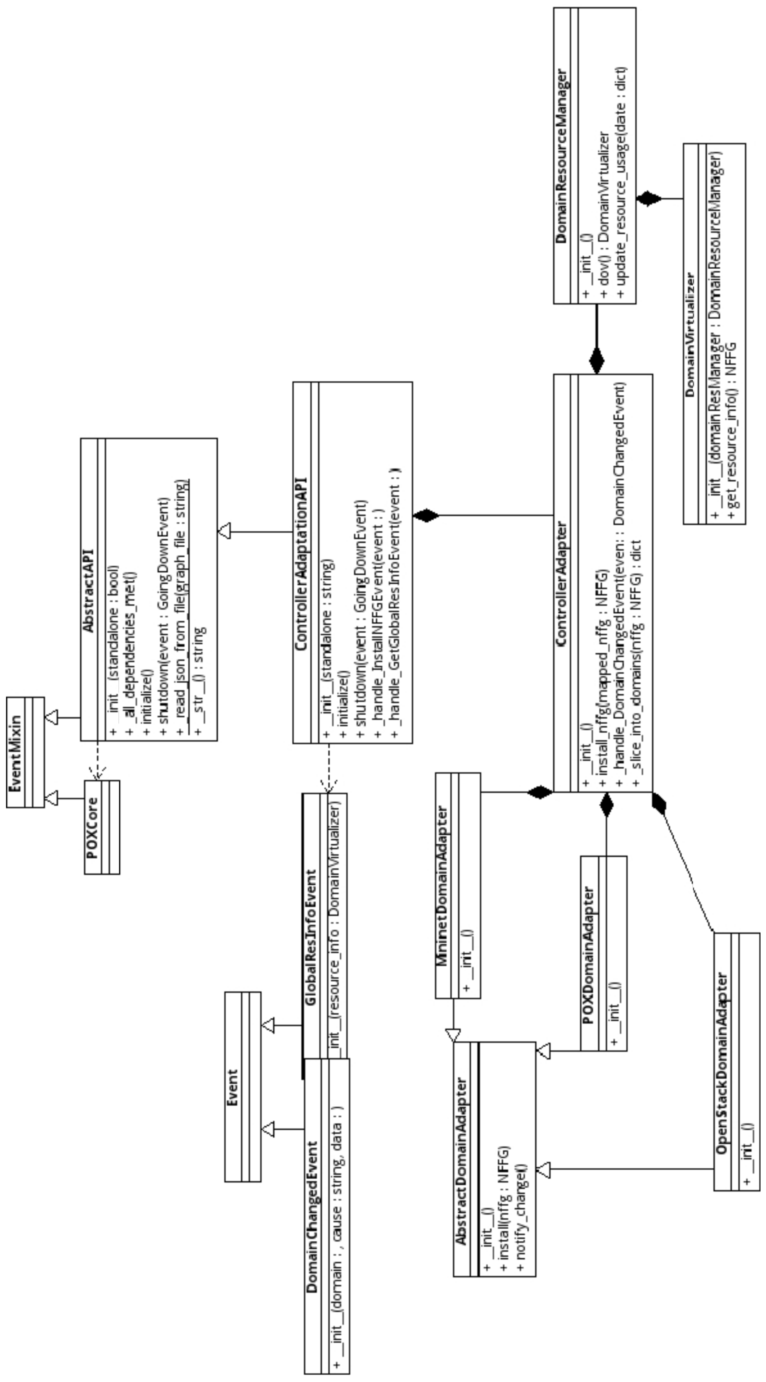

DomainResourceManager
+ __init__()
+ dov() : DomainVirtualizer
+ update_resource_usage(date : dict)
DomainVirtualizer
+ __init__(domainResManager : DomainResourceManager)
+ get_resource_info() : NFFG
ControllerAdapter
+ __init__()
+ install_nffg(mapped_nffg : NFFG)
+ _handle_DomainChangedEvent(even : DomainChangedEvent)
+ _slice_into_domains(nffg : NFFG) : dict
ControllerAdaptationAPI
+ __init__(standalone : string)
+ initialize()
+ shutdown(event : GoingDownEvent)
+ _handle_InstallNFFGEvent(event :)
+ _handle_GetGlobalResInfoEvent(event :)
AbstractAPI
+ init__(standalone : bool)
+ _all_dependencies_met()
+ initialize()
+ shutdown(event : GoingDownEvent)
+ _read_json_from_file(graph_file : string)
+ __str__() : string
EventMixin
POXCore
GlobalResInfoEvent
+ __init__(resource_info : DomainVirtualizer)
Event
DomainChangedEvent
+ __init__(domain : , cause : string, data :)
MininetDomainAdapter
+ __init__()
POXDomainAdapter
+ __init__()
OpenStackDomainAdapter
+ __init__()
AbstractDomainAdapter
+ __init__()
+ install(nffg : NFFG)
+ notify_change()

Figure 9-5: Class diagram of the Adaptation module

A.1.6 Dynamic model of Adaption module

The final part of the Service Requesting process is shown in Figure 9-6. The input parameter is the event which is raised by the *ResourceOrchestrationAPI* in the end of the previous dynamic model.

1. The triggering event called *InstallNFFGEvent* is handled by the *ControllerAdaptationAPI* class. The event contains the mapped NF-FG generated by the Orchestration module. Based on the type of the event, a dedicated event handler is invoked (i.e. UNIFY Or-Ca interface).
2. The request is delegated to the corresponding function of the central *ControllerAdapter* class.
3. The *ControllerAdapter* performs the parsing, slicing and distributing process and invokes the dedicated domain adapter classes.
4. After the adapters perform their tasks the global resource view is updated by the *DomainResourceManager* and an *InstallationFinishedEvent* is raised by the top API class to notify the upper layers about the successful Service Request process.

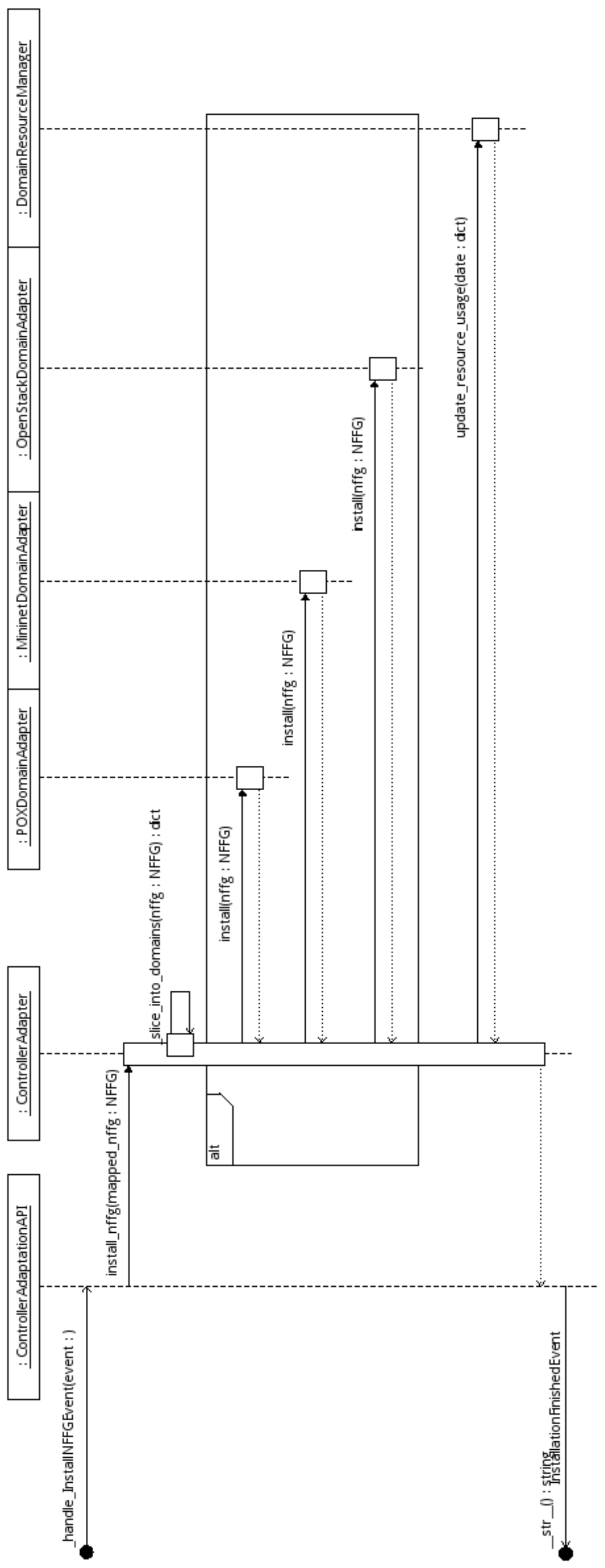
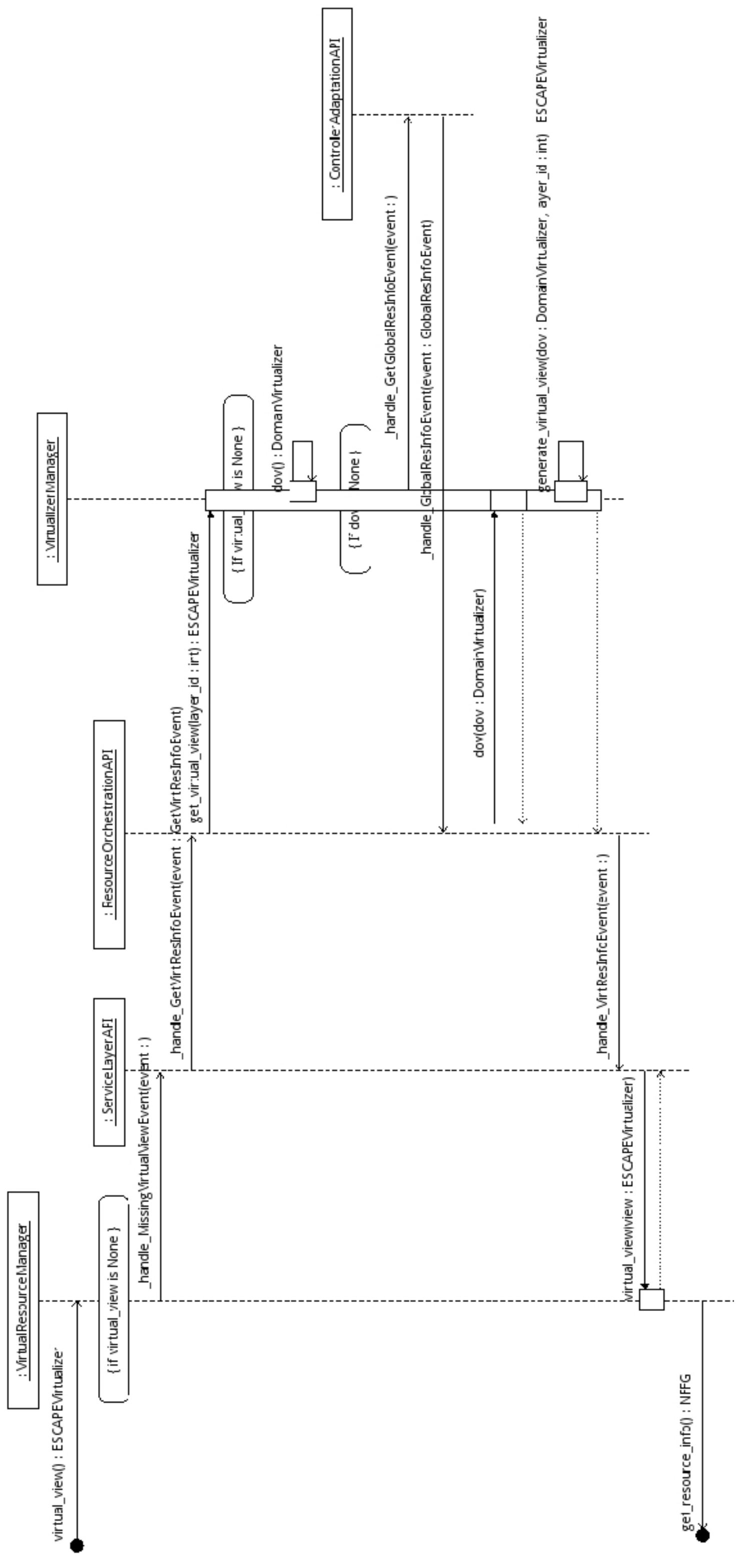

: ControllerAdaptationAPI
: ControllerAdapter
: POXDomainAdapter
: MininetDomainAdapter
: OpenStackDomainAdapter
: DomainResourceManager
_handle_InstallNFFGEvent(event :)
install_nffg(mapped_nffg : NFFG)
_slice_into_domains(nffg : NFFG) : dict
install(nffg : NFFG)
install(nffg : NFFG)
install(nffg : NFFG)
update_resource_usage(date : dict)
alt
__str__() : string
InstallationFinishedEvent

Figure 9-6: Interaction steps during a Service Request in the Adaptation module

The changes of the distinct domains are propagated upwards via the Virtualizer instances with the help of the DomainChangedEvent which is raised by the actual domain adapter classes. If a top module doesn't possess a dedicated Virtualizer a specific event is raised to request the missing Virtualizer. These steps are shown in Figure 9-7.

1. If an *ESCAPEVirtualizer* is missing in the *VirtualResourceManager* in the Service module a *MissingVirtualViewEvent* is raised which is forwarded to the Resource module via the *ServiceLayerAPI.*
2. The *ResourceOrchestrationAPI* receives the event and applies for an ESCAPEVirtualizer in contribution of the *VirtualizerManager*.
3. If the needed Virtualizer doesn't exist yet, the *ESCAPEVirtualizer* is generated by the *VirtualizerManager* using the *DomainVirtualizer*.
4. If the *DomainVirtualizer* is not available for the *VirtualizerManager* a *GetGlobalResInfoEvent* is raised to request the missing DoV.
5. The event is forwarded to the *ControllerAdaptationAPI* which responds the requested *DomainVirtualizer* in a *GlobalResInfoEvent.*
6. The event is handled by the *ResourceOrchestrationAPI*, the *DomainVirtualizer* is extracted from the event and set into the *VirtualizerManager*.
7. The requested *ESCAPEVirtualizer* is generated by the *VirtualizerManager* using the *DomainVirtualizer* and returned via a *VirtResInfoEvent* to the *ServiceLayerAPI* which set the Virtualizer into the *VirtualResourceManager*.



: VirtualResourceManager
: ServiceLayerAPI
: ResourceOrchestrationAPI
: VirtualizerManager
: ControllerAdaptationAPI
virtual_view() : ESCAPEVirtualizer
{ if virtual_view is None }
_handle_MissingVirtualViewEvent(event :)
_handle_GetVirtResInfoEvent(event : GetVirtResInfoEvent)
get_virtual_view(layer_id : int) : ESCAPEVirtualizer
{ If virtual_view is None }
dov() : DomainVirtualizer
{ If dov is None }
_handle_GetGlobalResInfoEvent(event :)
_handle_GlobalResInfoEvent(event : GlobalResInfoEvent)
dov(dov : DomainVirtualizer)
generate_virtual_view(dov : DomainVirtualizer, layer_id : int) : ESCAPEVirtualizer
_handle_VirtResInfoEvent(event :)
virtual_view(view : ESCAPEVirtualizer)
get_resource_info() : NFFG

Figure 9-7: Interaction steps during a request of missing Virtualizers

A.1.7 Integration issues

One of the main design goals of ESCAPEv2 is the support of easy integration with different components. Here, we highlight the main modules which can easily be changed or extended as we go further with the implementation of the prototypes.

- Communication interfaces
    - We use an event based communication between the main layers which is based on POX internal events. This approach supports the incorporation of other implementations of event-driven communication such as the Double Decker architecture based on ZeroMQ. This enables the integration with the monitoring framework established by WP4.
    - NETCONF based communication between the main elements can easily be introduced.
- Embedding/mapping algorithms
    - Strategy design pattern is used to enable different mapping algorithms (which follow the high level interfaces)
- NF-IB, NF-FG and resource databases
    - Abstract database interfaces and wrappers enable different types of catalogues and databases.
    - Neo4j based graph databases can easily be integrated.
- NF-FG class/library
    - This library providing helper functions can be used in different parts of the framework.
- Virtualizers, policy enforcement
    - These modules can be implemented in parallel.