unifying cloud
and carrier networks

Deliverable 3.3: Revised framework with functions and semantics

Dissemination level	PU
Version	0.1
Due date	31.10.2015
Version date	03.09.2015

This project is co-funded
by the European Union

Document information

Editors

Jokin Garay (EHU)

Contributors

Pontus Skoldstrom (ACREO), Balazs Sonkoly (BME), Balazs Nemeth (BME), Janos Czentye (BME), Jokin Garay (EHU), Jon Matias (EHU), David Jocha (ETH), Robert Szabo (ETH), Wouter Tavernier (IMINDS), Sahel Sahhaf (IMINDS), Steven Van Rossem (IMINDS), Matthias Rost (TUB)

Coordinator

Dr. András Császár
Ericsson Magyarország Kommunikációs Rendszerek Kft. (ETH)
Könyves Kálmán körút 11/B épület
1097 Budapest, Hungary
Fax: +36 (1) 437-7467
Email: andras.csaszar@ericsson.com

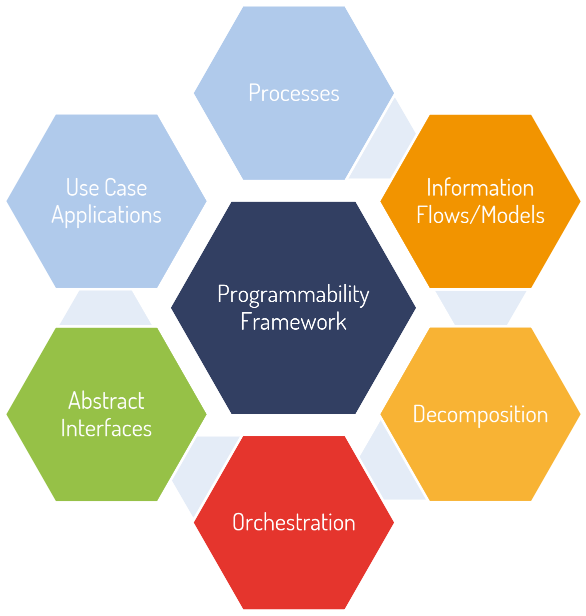
Project funding

7th Framework Programme
FP7-ICT-2013-11
Collaborative project
Grant Agreement No. 619609

Summary

This deliverable documents the final service programmability framework for the UNIFY architecture. Building on the initial framework described in [D3.1] and further updated in [D3.2; D3.2a], the final framework contains the relevant process flows, interfaces, information models and orchestration functionality in support of service programming in UNIFY. The work carried on WP3 has been continuously kept in line with the other technical work packages, WP4 for Service Provider DevOps and WP5 for the Universal Node, and has resulted in the development of the proof of concept prototype of WP3 as the outcome towards the project-wide Integrated Prototype.

The main goal of the programmability framework is to enable on-demand processing anywhere in the physically distributed network as well as in clouds, with dynamic and fine granular service (re-)provisioning, which can hide significant part of the resource management complexity from service providers and users, hence allowing them to focus on service and application innovation. The core parts of the framework are shown in the next figure:



Core parts of the programmability framework

The initial definition of the programmability framework covered in [D3.1] provided a common ground for the UNIFY partners to target different key challenges, which were identified and further pursued in [D3.2], while at the same time different focused prototypes evaluated the proposed alternatives. Building on the lessons learned, the programmability framework has been updated and is presented here on its final form:

- The flexible service provisioning needs to reconcile two sides of a spectrum: on one side there is the service definition, on the other side there is a heterogeneous landscape of infrastructure on which services need to be deployed. The first reflects the Service Layer, the latter is part of the Infrastructure Layer. It is the goal of the Orchestration Layer to bring both together. The Orchestration Layer receives the service information on its northbound interface from the Service Layer, and receives infrastructure resource models from network and cloud controllers on its southbound interface. The provisioning process flow is detailed both top-down and bottom-up. The first refers to the service invocation process, whereas the second refers to the service confirmation

process. Both programmability process flows have been further refined from the global definition in the UNIFY architecture covered in [D2.2].

- The information models to be used in the different reference points, both in the top-down and bottom-up information flows, have been formalized and basic primitives defined: endpoints, Network Functions, network elements and monitoring parameters. The research focuses on two missing and crucial information models: the Service Graph and the Network Function-Forwarding Graph (NF-FG). The former refers to the service request, whereas the latter defines the information model to enable the resource orchestration and enables the recursivity of the framework.
- The Network Function Information Base (NF-IB) supports both the Orchestration and Decomposition processes and is the entity responsible for storing the NF models/abstractions, NF relationships, NF implementation image(s) and NF resource requirements. The NF-IB is capable of storing the relationships between abstract NFs to deployable NFs (be it a 1:1 or 1:N relationship, effectively decomposing an abstract NF into multiplef NFs interconnected into an NF-FG) into a tree-like data structure in support of the decomposition process. A proof of concept of the NF-IB has been implemented in the Neo4j database, resulting in database read performance that scales nearly linearly with the number of nodes in the cluster. Also, the Neo4j High Availability support facilitates implementation of parallel/distributed embedding algorithms which leads to a more scalable orchestration process.
- UNIFY(ed) service decomposition is an important concept in UNIFY, sometimes referred to as Model-based decomposition, and is key for multi-stage service programming, enabling the decomposition at the appropriate stage on the orchestration process. This implies on one hand a time aspect of the decomposition: there is a decomposition Model, made (decided) in design time, while in execution time the Model is static, the instantiation is dynamically taking e.g. actual resources into account. Topics addressed include aspects related to the benefits of decomposition and when and to what extent decomposition should be used. Then various potential atomic blocks have been investigated that can be composed into larger functions or services. Finally several examples are provided, of how services can be composed from atomic blocks.
- The orchestration process has been extended to incorporate in a coherent process the service and associated monitoring provisioning, aligned with WP4. The research on the orchestration functionality focuses on virtual network embedding, which deals with the mapping of the NF-FG components on infrastructure resources. The abstraction and decomposition in multi-domain scenarios is also covered. Furthermore, the scale-in and -out mechanisms are proposed to improve the overall scalability of the framework. Also, the latest results considering the development of our efficient large-neighborhood search algorithm called Divine are presented. In essence, Divine builds upon a series of novel Integer Programming (IP) formulations that enable us to model the embedding of virtual networks (and thereby also service chains), with adaptations to allow for reconfigurations and elasticity. By reconfigurations we here specifically refer to migrating virtual nodes (or service functions) and adapting routing entries. Elasticity refers to the concept of being able to adapt the resource requirements of an already existing request. This is of particular interest, as requests may evolve over time and up or down scaling of services might be triggered to adapt to the customer's varying demand. As such, Divine offers the orchestration capabilities required for large-scale (re-)optimizations.
- The abstract interfaces in the framework have been implemented following the extended definition from the UNIFY architecture. The Cf-Or interface is an important proposal of UNIFY to enable service elasticity, the main

achievements on the mechanisms and components related to this reference point are summarized and demonstrated here for the Elastic Router use case. It is shown how the implementation of this Cf-Or interface creates the possibility for new service dynamics, different from current commercially available cloud platforms. Further on, the context of where this elastic router is usable will be also elaborated on and several aspects of how the Cf-Or interface is involved in the UNIFY architecture and programmability framework are handled.

- The impact of state migration is analyzed for two WP3 use cases: the Elastic Router and FlowNAC. To support VNF resiliency and VNF scaling mechanisms the internal state of the VNFs has to be managed properly (the state will typically consist of configuration state created by control components as well as transient state created by the traffic itself). To provide a VNF resiliency mechanism the VNF state management mechanism could e.g. periodically synchronize the state between a primary and a backup VNF instance, so when the primary instance fails, the backup instance is up to date and able to handle traffic flows redirected from the primary instance. In the case of scaling, appropriate transient and configuration state has to be migrated to a new or old instance of the VNF for scale-out and -in respectively. As different methods of scaling as suitable for different applications here we investigate a type of VNF that may require a different a state handling approach: "Split State VNFs", with a data/control plane division similar to the division between an OpenFlow controller and OpenFlow switches. In Split State VNFs state may be distributed among multiple sub-components and traffic processing may happen in multiple locations, controlled by a central node. Here we investigate state transfer, both for scalability and for resiliency, for two Split State VNFs; FlowNAC, a service authentication function, and the Elastic Router, a scalable router function.

All relevant components of the service programming and orchestration framework have been implemented in ESCAPE, the main proof of concept prototype of WP3 [D3.4]. Designed as a platform for intra and inter workpackage integration, ESCAPE supports efficient integration of different modules implemented by different partners and the easy re-use and integration of available tools. Oriented towards the Integrated Prototype, on the one hand, it supports the orchestration on top of Universal Node (UN) which is the main outcome of WP5. On the other hand, verification, monitoring and troubleshooting tools implemented by WP4 will also be integrated with this framework.

Contents

List of Figures

1 Introduction

This deliverable covers the final definition of the service programmability framework for the UNIFY architecture in section 2, updating the detail from deliverable D3.1 [D3.1]. Specifically, the updates refine core programmability flows and the information models to be used in each of the defined reference points for both the bottom-up and top-down information flows. The design of the Network Function Information Base is also updated, as well as the orchestration process, with details on service decomposition and embedding algorithms and incorporating in a coherent process the service and associated monitoring provisioning, aligned with WP4. The abstract interfaces are revised with additional detail on the possibilities opened up by the Cf-Or interface, described for the Elastic Router use case. Finally, possible mechanisms for state migration are exemplified in both the Elastic Router and FlowNAC use cases.

Next, section 3 cross checks the requirements imposed on the framework with the final definition. At last, future steps and conclusions are listed in section 4 with respect to prototyping.

Besides the main body of the deliverable, annex A provide additional detail on the support to the NF decomposition process in other approaches in the state of the art.

2 Description of final framework

This sections presents the final definition of the service programmability framework for the UNIFY architecture. The initial definition of the programmability framework covered in [D3.1] provided a common ground for the UNIFY partners to target different key challenges, which were identified and further pursued in [D3.2], while at the same time different focused prototypes evaluated the proposed alternatives.

Building on the lessons learned, we have updated the programmability framework which is defined next, following the structure presented in [D3.1] and represented in Figure 1. Those concepts which remain valid from the initial framework are only summarized here, for more detailed information, we refer to their original definition.

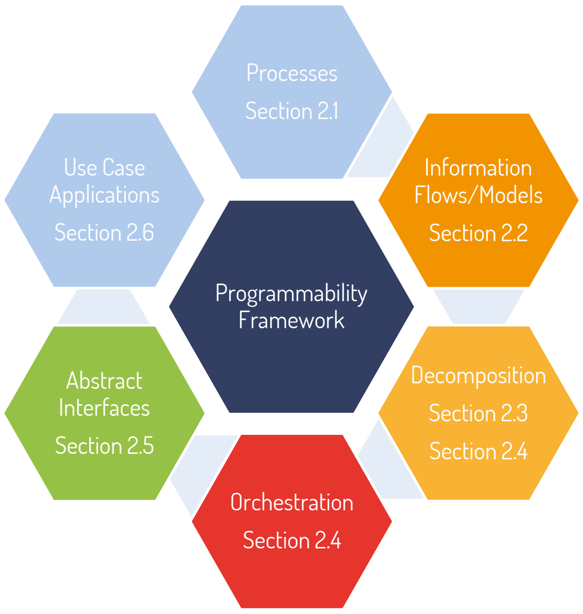


Figure 1: Core parts of the programmability framework

One aspect that demands a slightly more extended explanation, is the evolution on the understanding and approach to both the Service Graph (SG) and Network Function Forwarding Graph (NF-FG), as they are key concepts of the UNIFY programmability framework. Both were first introduced and defined in [D2.1]. From these seminal concepts in the initial architecture, the next release of deliverables in the UNIFY project [D2.2; D3.1] defined the final architecture and initial programmability framework. There, the NF-FG was established as central to the UNIFY framework for programming resource orchestration at compute, storage and network abstraction, in accordance with the virtualization, monitoring functions and quality indicators for rapid and flexible service creation. These deliverables provided updated definitions and a formalization of both the SG and NF-FG abstract information model to be used at the Sl-Or, Cf-Or and Or-Ca reference points.

Based on the initial model, several evaluation prototypes have further extended and adapted the definition of the NF-FG. As the prototypes identified conflicting requirements, tailored toward different use-cases, deployment scenarios or business models, this has led to two main NF-FG variants: a Service Centric (SC) NF-FG and a Virtualizer Based (VB) NF-FG. Both are thoroughly described and compared in [D3.2a] and potential updates resulting from the integration effort and final conclusions on the models will be given in the future Deliverable D3.5. Ultimately, the VB model, which is updated next, has been implemented as common model, as detailed in [D3.4]. During the development of ESCAPE the Virtualizer library became available, so it was decided to incorporate it. Since ESCAPE was elected as the kernel for integration toward the WP3 prototype, the natural decision was to follow through with the VB NF-FG as well for integration.

2.1 Programmability process flows

Our goal with the introduction of UNIFY's programmability framework is to enable on-demand processing anywhere in the physically distributed network as well as in clouds. Our objective is to create a programmability framework for dynamic and fine granular service (re-)provisioning, which can hide significant part of the resource management complexity from service providers and users, hence allowing them to focus on service and application innovation.

Before delving into the details of the framework, a short overview of the global mapping process is given below. While most important concepts will be described in this context, a more complete overview of the recurring terminology in UNIFY can be found in Section 2 of [D2.2].

The flexible service provisioning needs to reconcile two sides of a spectrum: on one side there is the service definition, on the other side there is a heterogeneous landscape of infrastructure on which services need to be deployed. The first reflects the Service Layer, the latter is part of the Infrastructure Layer. It is the goal of the Orchestration Layer to bring both together. The Orchestration Layer receives the service information on its north-bound interface from the Service Layer, and receives infrastructure resource models from network and cloud controllers on its southbound interface.

Service provisioning starts with the user defining a service request in the form of a Service Graph (SG). An SG describes a service requested by a user and defines how (which Network Functions) and where (which Service Access Points) the service is provided; and how successful delivery of the service is measured.

In order to enable mapping of the individual components of the SG to the infrastructure, the Service Layer performs translation of NF descriptions into resource requirements, as well as translating NF interconnections into concrete forwarding abstractions which can be mapped to network abstractions such as Big Switch with Big Software (BiS-BiS) connectivity between NFs (see Figure 2). The BiS-BiS abstraction is defined in [D2.2], and refers to the virtualization of a Forwarding Element with a Compute Node, enabling to instantiate and interconnect NFs. The result of this adaptation is a topology of BiS-BiS nodes, which is constructed and forwarded to the Orchestration Layer. Based on the resource model obtained via controllers interfacing with infrastructure, the Resource Orchestrator (RO) decomposes and maps NFs to server infrastructure, and network forwarding abstractions to infrastructure switching functionality. The resulting mapping is the UNIFY Resource service provided by the Orchestration Layer. In the particular example of Figure 2, the VNFs of the NF-FG on the left upper side are deployed on two separate Universal Nodes (UNs), and the Big Switch abstraction interconnecting them is decomposed into the combined switching functionality of two OpenFlow switches and the virtual switching capabilities of UN1 and UN2. The output of the orchestration is the mapping/embedding of instantiable Network Functions to physical or virtual resources defined as a Network Function-Forwarding Graph. Therefore, an NF-FG always shows a deployment view (configuration view) of a Network Function chain in a given infrastructure view defined over compute, storage and network abstraction.

The programmability process flow is described in [D3.1]. The service initiation process flow consists of

- Service Graph to Service Function Chain mapping, which is an NF-FG over the virtualization view presented to the Service Layer;
- NF and network overlay embedding process at the Orchestration Layer, which maps the NF-FG at the virtualiza-

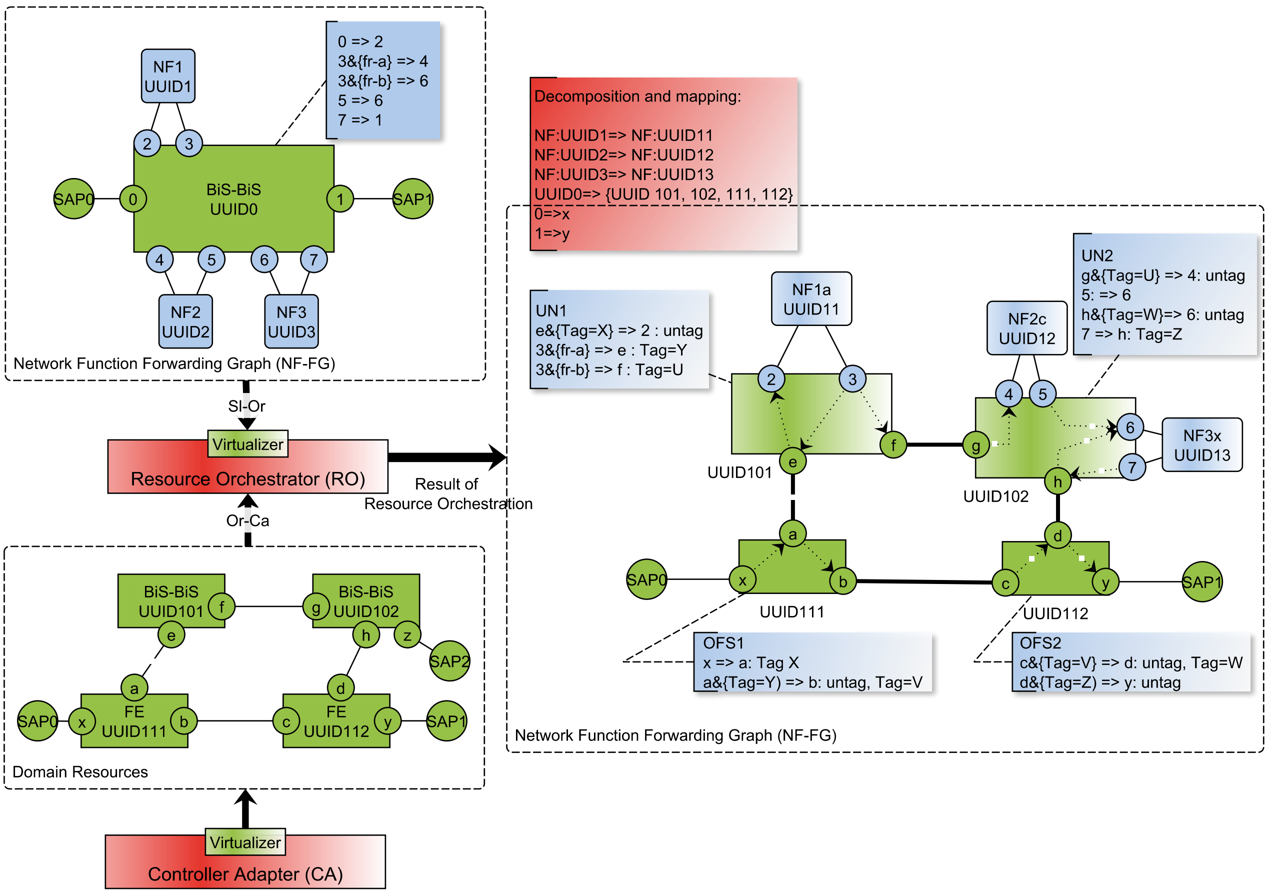


Figure 2: An illustrative example of the Resource Orchestrator's UNIFY Resource Service

tion view presented to the Service Layer to the underlying resources;

- Southbound adaptation to execute NF deployment and network overlay creation;
- Service configuration, which happens in the Service Layer or in the Tenant domain by the means of Element Managements and Operation Support System (OSS) once the Network Function chain is deployed.

Furthermore, Service Graph or Service Function Chain (re-)provision triggers may be provided by the user, the service management system, the resource management system or from within control plane components of the deployed Service Function chain. The steps include how this information will travel from the user down to the infrastructure nodes.

References to particular instances in the Infrastructure Layer can be assigned in a top-down manner. Once the Service Graph is instantiated at the Infrastructure Layer, the individual instantiation of components can be acknowledged and propagated back to the Service Layer in order to allow operation and management tasks to be performed, as described in [D3.1].

2.2 Information models according to the reference points

The information models form the essential information units transferred between different reference points in the programmability process. The role of the Orchestration Layer is to reconcile the bottom-up resource information flow driven by the infrastructure with the top-down service information requests. Information models are described in details in[D3.1]. In this section we will highlight the concept and give an example.

2.2.1 Bottom-up information flow

Information concerning networking, compute, storage resources or particular capabilities travels from the Infrastructure Layer up to the Orchestration Layer. Then, through a Virtualizer, it can travel one layer higher, to a next Orchestration Layer or to the Service Layer. This happens on various timescales and different level of details. Networking resources refer to available interfaces, bandwidth, delay characteristics. Compute resources are for example CPU characteristics, RAM memory, and storage refers to available disk space. The possibility for infrastructure elements to expose particular capabilities enables to expose specific execution environments (e.g., hardware-optimized implementations), particularly Network Functions (e.g., firewall of type x). Basic resource information, e.g., the existence of a switch or link, is seldom updated unless equipment is added, removed, or upon failure. Information flow upwards is a multi-level process: individual infrastructure resources announce themselves to their immediate controllers, and controllers consolidate information towards the Orchestration Layer. In addition, resource virtualization might be applied in order to shield lower layer details to higher layers. Resource virtualization might occur at the level of compute and/or network controllers, the Controller Adapter or the Resource Orchestrator.

Information at Co-Rm, Ca-Co, Or-Ca and Sl-Or reference points are described in [D3.1]. We prefer to use the virtualizer model at the Sl-Or and Or-Ca interfaces, as described in [D3.2a].

A short summary of the software and resource virtualization related to the up-ward information flow from [D3.2a] is presented in the following according to Listing 1, where:

node represents Big Switch with Big Software (BiS-BiS)

port represents an entry point for both BiS-BiS nodes (used by external links) and for Network Functions placed within a BiS-BiS.

sap Service Access Point (SAP) represents technology specific information (e.g. encapsulation, tagging) associated with a port; this is typically used at the edges of a domain.

link represents network resources like connection with bandwidth and delay characteristics between two ports; they describe both links interconnecting BiS-BiS nodes and internal logical connections within a single BiS-BiS node.

resources under the BiS-BiS node directly represent software execution related resources like CPU, memory and storage.

capabilities can contain a list of Network Functions the node supports.

Listing 1: Data model of domain resources for virtualization at Sl-Or, Cf-Or, Or-Ca: YANG tree

```
module: virtualizer3
  +--rw virtualizer
     +--rw id?        string
     +--rw name?      string
     +--rw nodes
     |  +--rw node* [id]
     |     +--rw id               string
     |     +--rw name?            string
     |     +--rw type             string
     |     +--rw ports
     |     |  +--rw port* [id]
     |     |     +--rw id             string
     |     |     +--rw name?          string
     |     |     +--rw port_type?     string
     |     |     +--rw capability?    string
     |     |     +--rw sap?           string
     |     +--rw links
     |     |  +--rw link* [src dst]
     |     |     +--rw id?          string
     |     |     +--rw name?        string
     |     |     +--rw src          ->
     |     |     +--rw dst          ->
     |     |     +--rw resources
     |     |        +--rw delay?        string
     |     |        +--rw bandwidth?    string
     |     +--rw resources
     |     |  +--rw cpu        string
     |     |  +--rw mem        string
     |     |  +--rw storage    string
     |     +--rw capabilities
     |     |  +--rw supported_NFs
     |     |     +--rw node* [id]
     |     |        +--rw id             string
     |     |        +--rw name?          string
     |     |        +--rw type?          string
     |     |        +--rw ports
     |     |        |  +--rw port* [id]
     |     |        |     +--rw id             string
     |     |        |     +--rw name?          string
     |     |        |     +--rw port_type?     string
     |     |        |     +--rw capability?    string
     |     |        |     +--rw sap?           string
     |     |        +--rw links
     |     |        |  +--rw link* [src dst]
     |     |        |     +--rw id?          string
     |       |           |       +——rw name?         string
     |       |           |       +——rw src           —>
     |       |           |       +——rw dst           —>
     |       |           |       +——rw resources
     |       |           |           +——rw delay?         string
     |       |           |           +——rw bandwidth?     string
     |       |           +——rw resources
     |       |               +——rw cpu          string
     |       |               +——rw mem          string
     |       |               +——rw storage      string
     +——rw links
         +——rw link* [src  dst]
             +——rw id?             string
             +——rw name?           string
             +——rw src             —>
             +——rw dst             —>
             +——rw resources
                 +——rw delay?           string
                 +——rw bandwidth?    string
```

A three BiS-BiS based virtualization is shown in Figure 3 and Listing 2.

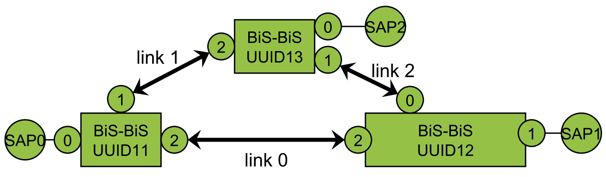


Figure 3: 3-Node domain resources virtualization example: topology view

Listing 2: 3-Node domain resources virtualization example: xml view

```
< virtualizer >
    <id>UUID002</id>
    <name>3—node simple  infrastructure    report </name>
    <nodes>
        <node>
            <id>UUID11</id>
            <name>West Bis—Bis node</name>
            <type> BisBis </type>
            <ports>
                <port>
                    <id>0</id>
                    <name>SAP0 port</name>
                    <port_type>port—sap</port_type>
                    <sap—type>vx—lan</sap—type>
                    <vxlan  >...</  vxlan>

                </port>
                <port>
                    <id>1</ id>
                    <name>North port</name>
                    <port_type>port—abstract</port_type>
                    < capability   >...</   capability >
                </port>
                <port>
                    <id>2</ id>
                    <name>East port</name>
                    <port_type>port—abstract</port_type>
                    < capability  >...</  capability >
                </port>
            </ports>
            <resources>
                <cpu>20</cpu>
                <mem>64 GB</mem>
                <storage>100 TB</storage>
            </resources>
        </node>
        <node>
            <id>UUID12</id>
            <name>East Bis—Bis node</name>
            <type> BisBis </type>
            <ports>
                <port>
                    <id>1</id>
                    <name>SAP1 port</name>
                    <port_type>port—sap</port_type>
                    <sap—type>vx—lan</sap—type>
                    <vxlan  >...</  vxlan>
                </port>
                <port>
                    <id>0</id>
                    <name>North port</name>
                    <port_type>port—abstract</port_type>
                    < capability  >...</  capability >
                </port>
                <port>
                    <id>2</id>
                    <name>West port</name>
                    <port_type>port—abstract</port_type>
                    < capability  >...</  capability >
                </port>
            </ports>
            <resources>
                <cpu>10</cpu>
                <mem>32 GB</mem>
                <storage>100 TB</storage>
            </resources>
        </node>
        <node>
            <id>UUID13</id>
            <name>North Bis—Bis node</name>
            <type> BisBis </type>
            <ports>
                <port>
                    <id>0</id>
                    <name>SAP2 port</name>
                    <port_type>port—sap</port_type>
                    <sap—type>vx—lan</sap—type>
                    <vxlan  >...</  vxlan>
                </port>
                <port>
                    <id>1</id>
                    <name>East port</name>
                    <port_type>port—abstract</port_type>
                    < capability  >...</  capability >
                </port>
                <port>
```

```
          <id>2</id>
          <name>West port</name>
          <port_type>port-abstract</port_type>
          < capability >...</ capability >
        </port>
      </ports>
      <resources>
        <cpu>20</cpu>
        <mem>64 GB</mem>
        <storage>1 TB</storage>
      </resources>
    </node>
  </nodes>
  < links >
    < link >
      <id>0</id>
      <name>Horizontal link </name>
      <src >../../ nodes/node[id=UUID11]/ports/port[id =2]</ src>
      <dst >../../ nodes/node[id=UUID12]/ports/port[id =2]</ dst>
      <resources>
        <delay>2 ms</delay>
        <bandwidth>10 Gb</bandwidth>
      </resources>
    </ link >
    < link >
      <id >1</ id>
      <name>West link</name>
      <src >../../ nodes/node[id=UUID11]/ports/port[id =1]</ src>
      <dst >../../ nodes/node[id=UUID13]/ports/port[id =2]</ dst>
      <resources>
        <delay>5 ms</delay>
        <bandwidth>10 Gb</bandwidth>
      </resources>
    </ link >
    < link >
      <id>2</id>
      <name>East link </name>
      <src >../../ nodes/node[id=UUID12]/ports/port[id =0]</ src>
      <dst >../../ nodes/node[id=UUID13]/ports/port[id =1]</ dst>
      <resources>
        <delay>2 ms</delay>
        <bandwidth>5 Gb</bandwidth>
      </resources>
    </ link >
  </ links >
</ virtualizer >
```

2.2.1.1 Infrastructure report updates/example

If different deployment options are supported by the virtualization provider beyond generic VNF placement, then those can be explicitly reported in the capabilities part of the resource virtualization.

For example, if a "Parental Control B.4" type Network Function can only be instantiated in two flavors at a BiS-BiS node, then the corresponding resource virtualization and capability report is shown in Figure 4 and Listing 3.

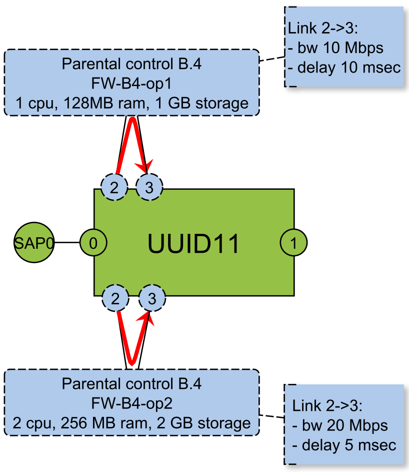


Figure 4: Example for two flavors of "Parental Control B.4" NF: topology view

Listing 3: Example for two flavors of "Parental Control B.4" NF: xml view

```
<?xml  version =" 1.0 "  ?>
< virtualizer >
    <id >UUID001</id>
    <name>Single  node with  link  internal  delays  infrastructure  report </name>
    <nodes>
        <node>
            <id >UUID11</id >
            <name>single  Bis —Bis node</name>
            <type> BisBis </type>
            <ports>
<!—— ...  ——>
            </ports>
            < capabilities >
                <supported_NFs>
                    <node>
                        <name >... NF1 deployment  option  1</name>
                        <id >FW—B4—op1</id> <!—— different for  all  deployment  options  of  the  same NF ——>
                        <type> Parental  control  B.4</type>  <!—— same for  all  deployment  options  of  the  same NF ——>
                        <ports>
                            <port>
                                <id >2</id >
                                <name>in</name>
                                <port_type >port—abstract</port_type >
                            </port>
                            <port>
                                <id >3</id >
                                <name>out</name>
                                <port_type >port—abstract</port_type >
                            </port>
                        </ports>
                        < links >
                            < link >
                                <id > int0 </id >
                                <name> internal  horizontal </name>
                                <src  >../../../  ports/port[id =2]</src>
                                <dst  >../../../  ports/port[id =3]</dst>
                                <resources>
                                    <delay>10 ms</delay>
                                    <bandwidth>10 Mb</bandwidth>
                                </resources>
                            </link>
                        </links>
                        <resources>
                            <cpu>1</cpu>
                            <mem>128 MB</mem>
                            <storage>1 GB</storage>
                        </resources>
                    </node>
                    <node>
                        <name>... NF1 deployment option 2</name>
                        <id>FW-B4-op2</id>
                        <type> Parental control B.4</type>
                        <ports>
                            <port>
                                <id>2</id>
                                <name>in</name>
                                <port_type>port-abstract</port_type>
                            </port>
                            <port>
                                <id>3</id>
                                <name>out</name>
                                <port_type>port-abstract</port_type>
                            </port>
                        </ports>
                        <links>
                            <link>
                                <id>int0</id>
                                <name>internal horizontal </name>
                                <src >../../../ ports/port[id=2]</src>
                                <dst >../../../ ports/port[id=3]</dst>
                                <resources>
                                    <delay>5 ms</delay>
                                    <bandwidth>20 Mb</bandwidth>
                                </resources>
                            </link>
                        </links>
                        <resources>
                            <cpu>2</cpu>
                            <mem>256 MB</mem>
                            <storage>2 GB</storage>
                        </resources>
                    </node>
                </supported_NFs>
            </ capabilities >
            <resources>
                <cpu>20</cpu>
                <mem>64 GB</mem>
                <storage>100 TB</storage>
            </resources>
        </node>
    </nodes>
</ virtualizer >
```

2.2.2 Top-down information flow

The top-down information flow was introduced in Sec.2.1. The information itself was described in details [D3.1] for the U-Sl, Sl-Or, Or-Ca, Ca-Co and Co-Rm reference points. Below we will give an example.

We prefer to use the virtualizer model at the Sl-Or and Or-Ca interfaces, as described in [D3.2a].

A brief summary of the virtualizer based NF-FG allocation is presented here according to the extension defined in Listing 4, where:

node under NF_instances represent a Network Function, which accommodates software resources (CPU, memory, storage);

flowentry represents forwarding behavior configuration over the Big Switch, with associated network resource need (bandwidth, delay);

link within an NF describing VNF dimensioning targets, i.e., expected networking characteristics if defined.

Listing 4: Data model for the resource allocation view at Sl-Or, Cf-Or, Or-Ca: YANG tree

```
module:   virtualizer3
   +——rw  virtualizer
      +——rw id?        string
      +——rw name?      string
      +——rw nodes
      |  +——rw node* [id]
 (...)
      |     +——rw NF_instances
      |     |  +——rw node* [id]
      |     |     +——rw id               string
      |     |     +——rw name?            string
      |     |     +——rw type?            string
      |     |     +——rw ports
      |     |     |  +——rw port* [id ]
      |     |     |     +——rw id               string
      |     |     |     +——rw name?            string
      |     |     |     +——rw port_type?       string
      |     |     |     +——rw  capability ?    string
      |     |     |     +——rw sap?             string
      |     |     +——rw links
      |     |     |  +——rw link* [ src   dst ]
      |     |     |     +——rw id?           string
      |     |     |     +——rw name?         string
      |     |     |     +——rw src           —>
      |     |     |     +——rw dst           —>
      |     |     |     +——rw resources
      |     |     |        +——rw delay?         string
      |     |     |        +——rw bandwidth?     string
      |     |     +——rw resources
      |     |        +——rw cpu          string
      |     |        +——rw mem          string
      |     |        +——rw storage      string
      |     +——rw flowtable
      |        +——rw flowentry* [ id ]
      |           +——rw id               string
      |           +——rw name?            string
      |           +——rw  priority ?      string
      |           +——rw port             —>
      |           +——rw match            string
      |           +——rw action           string
      |           +——rw out?             —>
      |           +——rw resources
      |              +——rw delay?         string
      |              +——rw bandwidth?     string
```

```
        +——rw links
(...)
```

An example allocation of a request is shown in a single BiS-BiS resource view in Figure 5 and Listing 5.

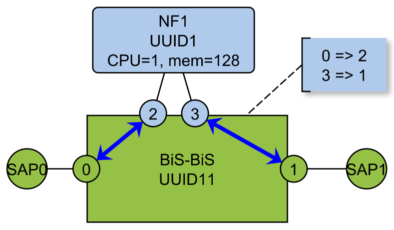


Figure 5: Resource allocation request (NF-FG) over a single BiS-BiS: topology view

Listing 5: Resource allocation request (NF-FG) over a single BiS-BiS: xml view

```
< virtualizer >
    <id>UUID001</id>
    <name>Single node simple  request</name>
    <nodes>
        <node>
            <id>UUID11</id>
            <NF_instances>
                <node>
                    <id>NF1</id>
                    <name>first  NF</name>
                    <type>Parental  control  B.4</type>
                    <ports>
                        <port>
                            <id>2</id>
                            <name>in</name>
                            <port_type>port—abstract</port_type>
                            < capability  >...</  capability >
                        </port>
                        <port>
                            <id>3</id>
                            <name>out</name>
                            <port_type>port—abstract</port_type>
                            < capability  >...</  capability >
                        </port>
                    </ports>
                    <resources>
                        <cpu>1</cpu>
                        <mem>128 MB</mem>
                    </resources>
                </node>
            </NF_instances>
            <flowtable>
                <flowentry>
                    <id>f1</id>
                    <port  >../../../  ports/port[id =0]</port>
                    <match></match>
                    <out  >../../../  NF_instances/node[id=NF1]/ports/port[id =2]</out>
                    <action></action>
                </flowentry>
                <flowentry>
                    <id>f2</id>
                    <port   >../../../   NF_instances/node[id=NF1]/ports/port[ id =3]</ port>
                    <match></match>
                    <out   >../../../   ports/port[ id =1]</ out>
                    <action ></ action >
                </flowentry>
            </ flowtable >
        </node>
    </nodes>
</ virtualizer >
```

2.3 Network Function Information Base

The Network Function Information Base (NF-IB) is the entity responsible for storing the NF models/abstractions, NF relationships, NF implementation image(s) and NF resource requirements. The NF-IB supports the definition of abstract NFs such as a FireWall, referring to a type, a potential number of ports/interfaces, as well as dependencies to other NFs. Abstract NFs might be implemented through more refined NFs or might be decomposed themselves into multiple NFs interconnected into an NF-FG with the same external interfaces as the higher level NF. The NF-IB is capable of storing these relationships into a tree-like data structure in support of the decomposition process. The leaves of the decomposition tree are NFs for which low-level implementation and deployment information is available such as images, provisioning scripts, resource requirements in terms of CPU, memory and storage.

We have implemented the NF-IB in Neo4j database. As this database is capable of storing key-value pairs for nodes and edges, for each NF we have stored the explained tree-like structure with all the corresponding information of nodes and edges. Several modules have been implemented to enable i) updating of the database and ii) retrieval of all possible decompositions of a given NF.

Additionally, Neo4j supports High-Availability (HA) by distributing the full database onto multiple nodes. This results in database read performance that scales nearly linearly with the number of nodes in the cluster. The Neo4j HA facilitates implementation of parallel/distributed embedding algorithms which leads to a more scalable orchestration process.

In order to jointly decompose services and allocate resources to the VNFs, the embedding algorithms should retrieve the NF decompositions from the NF-IB and select a suitable decomposition and perform a mapping. An ILP-based algorithm and a heuristic solution for such a joint optimization were detailed in [D3.2].

2.3.1 Service management/deployment support in NF-IB

The main functionalities of an orchestrator include optimal mapping (embedding) of VNFs across infrastructure and instantiating VNFs at reasonable locations. In order to find an optimal mapping, the orchestrator uses the NF information and NF decompositions stored in the NF-IB. In order to instantiate the VNFs at the locations determined by the embedding process, the orchestrator requires additional information on the VNFs dependencies and the order in which VNFs should be instantiated. Assuming a service comprising a web application that depends on a database, once the orchestrator determines the location where these components should be located (using the embedding algorithms) it should first initiate/deploy the database and then the web application is deployed.

The designed NF-IB can be extended to support such dependencies in addition to NF information and NF decompositions. These dependencies can be indicated as relationships between the VNFs which enables deriving the order of VNFs instantiation. The relationship is defined in terms of source and target VNFs which means that the target VNF 'depends on' the source VNF. In other words, the source VNF should be initiated/deployed first and then the target VNF

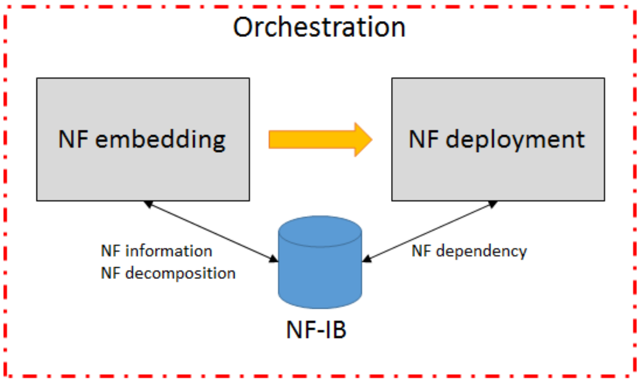


Figure 6: NF embedding and deployment using NF information, decomposition and dependency stored in the NF-IB

can be deployed. This can simply be implemented in the Neo4j-based NF-IB, as different types of relationship (edges) between nodes (VNFs) in the database can be defined. An additional relationship type, 'depends on', can be defined in support of VNFs dependencies. The two main functionalities of the orchestrator together with the required information stored in the NF-IB are illustrated in Figure 6.

2.4 Orchestration process

The overall orchestration process was introduced in Deliverable D3.1, section 6.7. The main steps presented here remain the same as is D3.1; Decomposition, Virtual Network Embedding (VNE), Verification, and finally Scoping, but as the SP-DevOps workflow has been defined in Deliverable D4.2, the required steps for both NF-FG and associated monitoring provisioning can now be combined into a coherent process [D4.2]. Figure 7 shows the updated Orchestration process, starting from an NF-FG with its associated MEASURE description arriving at the Virtualizer at the left. The NF-FG and MEASURE combination then passes through the Decomposition, VNE, Verification, and finally Scoping steps before reaching the Controller adapter. Updates to the orchestration process coming from SP-DevOps primarily affects the Decomposition and Scoping steps as the MEASURE monitoring description has to be handled there.

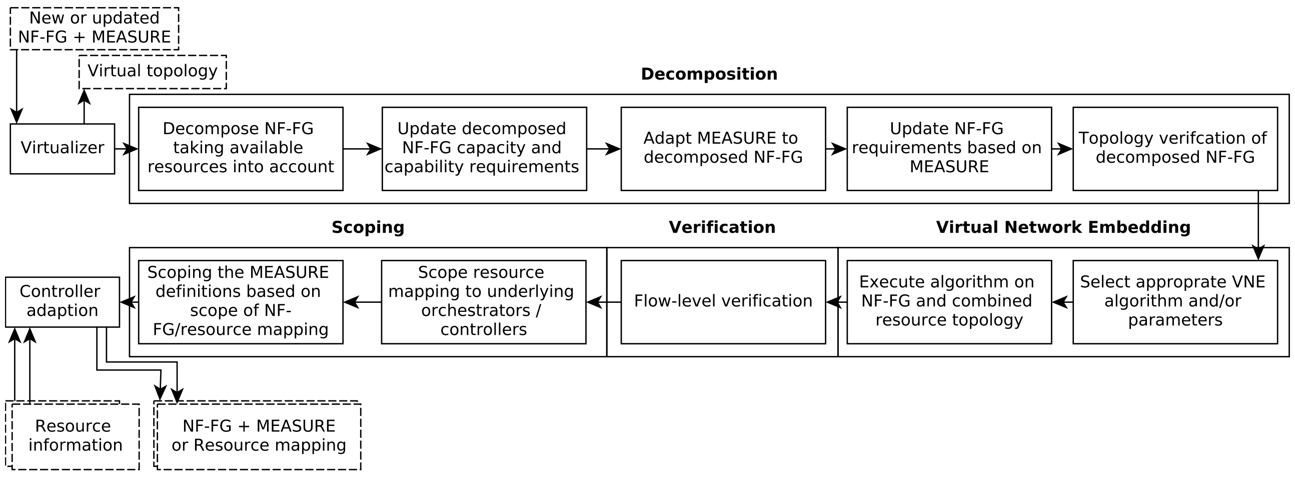

New or updated NF-FG + MEASURE
Virtual topology
Decomposition
Virtualizer
Decompose NF-FG taking available resources into account
Update decomposed NF-FG capacity and capability requirements
Adapt MEASURE to decomposed NF-FG
Update NF-FG requirements based on MEASURE
Topology verifcation of decomposed NF-FG
Scoping
Verification
Virtual Network Embedding
Controller adaption
Scoping the MEASURE definitions based on scope of NF-FG/resource mapping
Scope resource mapping to underlying orchestrators / controllers
Flow-level verification
Execute algorithm on NF-FG and combined resource topology
Select approprate VNE algorithm and/or parameters
Resource information
NF-FG + MEASURE or Resource mapping

Figure 7: Extended view of the orchestration process

During the **decomposition** step the MEASURE monitoring description has to be adapted to the decomposed NF-FG and additional resource and capability requirements coming from the need to support the monitoring tools defined by WP4 has to be taken into account. Updates on the Decomposition step are presented in section 2.4.1, while a detailed description of the SP-DevOps processes in the orchestration layer can be found in section 3.3 of D4.2, with a discussion of MEASURE in section 5.2.

The incoming NF-FG and MEASURE annotations may represent either a new service or an update on an already deployed service. Updates may originate either in a higher layer in which case they arrive over the Sl-Or interface, or from a lower layer when arriving over the Cf-Or interface. In both cases, an update is likely related to a scalability trigger and performed in order to allocate additional resources or to lower the already allocated amount. In a running service updates may also be only on either the NF-FG or MEASURE in case of new or additional resource allocations that does not affect monitoring or vice versa in case of additional monitoring tools or thresholds are required without updating the resource allocations.

Updating the current **virtual network embedding** without disrupting the service is the focus of the two following chapters. Section 2.4.2 presents the latest results on the DiVINE VNE algorithm, together with the implication of NF-FG updates on the algorithm. Section 2.6 follows with the latest results on VNF state management which is required to perform NF-FG updates without disrupting services. The focus of this chapter is on what we call "Split state VNFs", i.e. VNFs that are not monolithic but rather consists of a collection of distributed components. These present a slightly different challenge than VNFs handled by the state management methods described in deliverable D3.2 as their state may be distributed in several components [D3.2].

Section 5.10 of D4.2 describes the pre-deployment **verification** of the decomposed NF-FG. Verification consists of two steps, the first is to verify the correctness of decomposed NF-FG topology. This step verifies that the decompositioning has been performed correctly, e.g. without unreachable nodes or black holes in the graph. The second step is performed after the embedding/mapping phase and verifies that VNF configuration (if available), verification policies, and the mapped NF-FG are consistent. A verification policy may express a goal such as "traffic from SAP1 should be able to reach VNF2". Verifying if that policy is fulfilled may require information not available in the NF-FG itself. If traffic from SAP1 to VNF2 passes through VNF1, information about VNF1 is needed, such as its type or configuration. If VNF1 is a type of NF that allows any traffic to pass through it, only the type information is necessary. On the other hand, if VNF1 is a Firewall details about its configuration is necessary.

In the **scoping** step the MEASURE monitoring description needs to be scoped following the NF-FG so that underlying orchestrators and/or controllers receive an appropriate MEASURE script adapted to the partitioned NF-FG(s). Scoping in the context of the Cf-Or interface is discussed in section 2.5.1, whereas scoping for MEASURE is discussed in D4.2, section 5.2.

2.4.1 Service decomposition

The model-based service decomposition was identified in [D2.2] as one of the main features of the UNIFY Architecture.

The service decomposition concept is described in [D3.1], it is part of the orchestration. Service decomposition is the process of transforming a NF-FG containing abstract NF(s) to NF-FG(s) containing less abstract, more implementation-close NF(s). This can also include dividing the functionality of a complex NF to more, less complex NFs. In UNIFY, we have a generic concept of UNIFY(ed) service decomposition, and two realization options, the NF-IB-based and the CtrlApp based decomposition.

UNIFY(ed) service decomposition is an important concept in UNIFY, sometimes referred to as Model-based decomposition. This implies on one hand a time aspect of the decomposition: there is a decomposition Model, made (decided) in design time, while in execution time the Model is static, the instantiation is dynamically taking e.g. actual resources into account. The model is basically the decomposition model set. A decomposition rule in generic form is a $NFFG \rightarrow \{NFFG\}$ mapping (which means that a NF-FG will be transformed to another NF-FG or a set of NF-

FGs). On the other hand, the model based decomposition implies abstraction. The model describes abstract type to type mappings, not instances: the model stored in the NF-IB can exist without ever instantiating any NF. The model is also a dynamic entity, it can change over time: e.g. new possible realization options can be developed for an abstract NF. The model based service decomposition also refers to the way the Orchestrator decomposes an NF-FG's NFs. It will decompose them as long as there are no possible further decompositions at the given layer. This means the last (lowest layer) Orchestrator (e.g. the Universal Node local Orchestrator) has to perform the decomposition step as long as there is no more decompositions possible, meaning getting to atomic NF(s).

In this sense the Orchestrator thinks that it has received a high level NF-FG with abstract NFs; while it decomposes to low-level, "atomic" NFs, a.k.a., instances, according to his view. Taking into account the recursion possibility in the UNIFY architecture, an NF "instance" of a higher layer can be an abstract NF to be further decomposed by a lower layer.

[D3.1] addresses both the NF-IB-based and the CtrlApp based decomposition, as well as the types of decomposition rules.

Service decomposition is described in more details in [D3.2]. Topics addressed include aspects related to the benefits of decomposition and when and to what extent decomposition should be used. Then various potential atomic blocks have been investigated that can be composed into larger functions or services. Finally several examples are provided, of how services can be composed from atomic blocks.

Decomposing or generalizing unknown or underspecified NFs is out of our scope. In the UNIFY concept the NFs and the decomposition rules or functions are provided by the NF developer.

The high level concept how to formalize the decomposition rules with the help of our virtualizer model is presented in [D3.2a]. A detailed description is given below with examples.

In the next examples we will show some cases, how a decomposition rule can be expressed with the Virtualizer model. Important to note here, that these are decomposition rules (stored e.g. in the NFIB), which can be applied according to the decision of the RO.

The left part of Fig.8 shows an example rule that "NF a" can be decomposed to a set of NFs ("NF1", "NF2", "NF3") and forwarding rules between them. The right part of Fig.8 shows how this rule can be formulated with a single Virtualizer-based description. The Virtualizer contains a single node, which corresponds to the NF to be decomposed, while its NF instances are the compound NFs. The flow rules represent the decomposed forwarding rules.

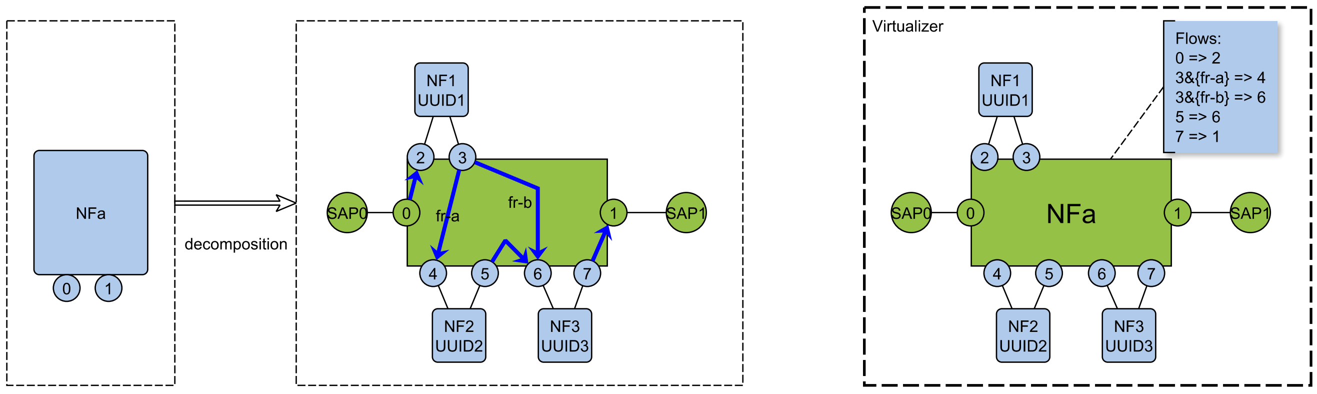


Figure 8: Simple decomposition: topology view

The next example in Fig.9 shows how resources of the original NF are mapped to decomposed NF instance resources.

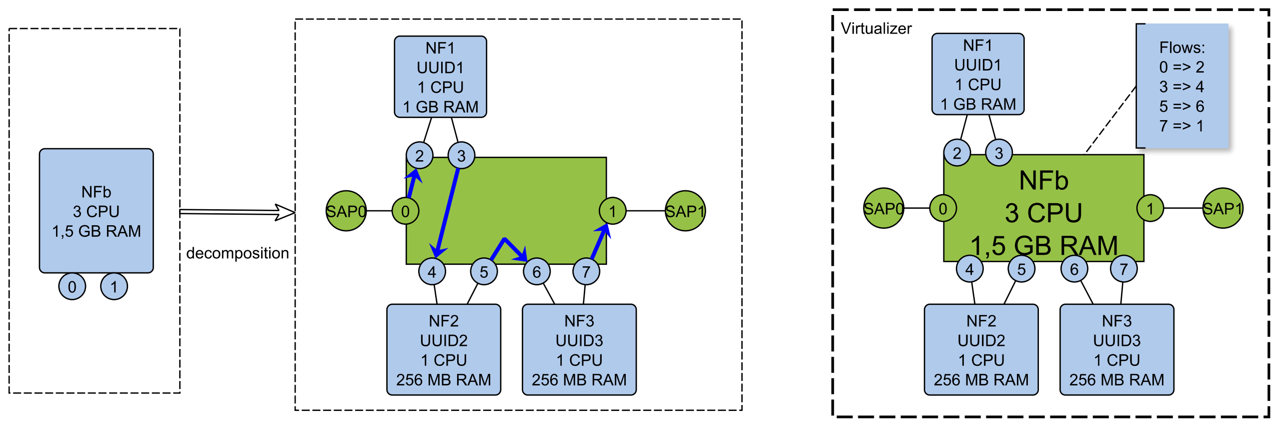


Figure 9: Resource decomposition: topology view

As depicted in the example in Fig.10, link resources of the original NF are mapped to a combination of:

- Node resources (see NF1 in the example)
- BiS-BiS flow resources (see delay attached to flows in the BiS-BiS of Fig.10)
- Link resources of decomposed NFs, which may then later be further decomposed (see NF3 in the example of Fig.10)

According to the information model introduced in sec. 2.2, the XML representation of Fig.10 is given in Listing 6.

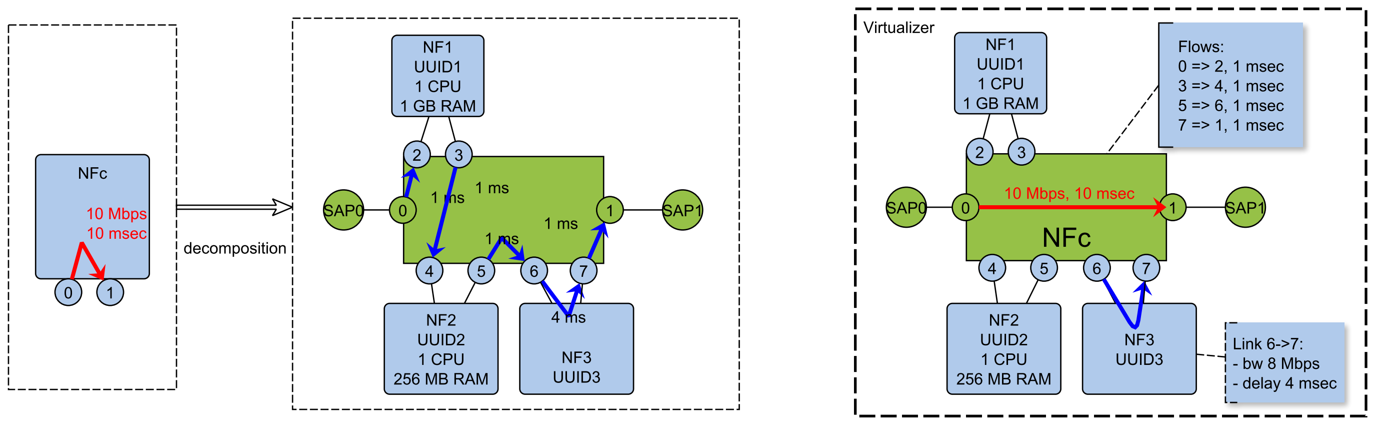


Figure 10: Advanced decomposition: topology

Listing 6: Advanced decomposition: xml example

```
<?xml  version ="1.0"  ?>
< virtualizer >
    <id> rule_xy0123 </id>
    <name >... decomposition  rule  example</name>
    <nodes>
        <node>
            <id> rule_xy0123 </id> <!-- different  for  all  decomposition  options  of  the  same NF -->
            <name >... decomposition  rule  example</name>
            <type>NFc</type> <!-- same for  all  decomposition  options  of  the  same NF -->
            <ports> <!-- ports  of  the  original  NF -->
            <port>
                <id>0</id>
                <name>input port</name>
                <port_type>port—abstract</port_type>
            </port>
            <port>
                <id>1</id>
                <name>output port</name>
                <port_type>port—abstract</port_type>
            </port>
        </ports>
        <links> <!—— link parameters of the original NF ——>
            <link>
                <id>int0</id>
                <name>internal link</name>
                <src>../../../ports/port[id=0]</src>
                <dst>../../../ports/port[id=1]</dst>
                <resources>
                    <delay>10 ms</delay>
                    <bandwidth>10 Mbps</bandwidth>
                </resources>
            </link>
        </links>
        <resources>
            <!—— resource parameters of the original NF ——>
        </resources>
        <NF_instances> <!—— decomposed NFs: ——>
            <node>
                <id>UUID1</id>
                <name>first NF</name>
                <type>NF1</type>
                <ports>
                    <port>
                        <id>2</id>
                        <name>in</name>
                        <port_type>port—abstract</port_type>
                    </port>
                    <port>
                        <id>3</id>
                        <name>out</name>
                        <port_type>port—abstract</port_type>
                    </port>
                </ports>
                <resources> <!—— resource requirement on the decomposed NF ——>
                    <cpu>1</cpu>
                    <mem>1 GB</mem>
                </resources>
            </node>
            <node>
                <id>UUID2</id>
                <name>cache</name>
                <type>NF2</type>
                <ports>
                    <port>
                        <id>4</id>
                        <name>in</name>
                        <port_type>port—abstract</port_type>
                    </port>
                    <port>
                        <id>5</id>
                        <name>out</name>
                        <port_type>port-abstract</port_type>
                    </port>
                </ports>
                <resources>
                    <cpu>1</cpu>
                    <mem>256 MB</mem>
                </resources>
            </node>
            <node>
                <id>UUID3</id>
                <name> firewall </name>
                <type>NF3</type>
                <ports>
                    <port>
                        <id>6</id>
                        <name>in</name>
                        <port_type>port-abstract</port_type>
                    </port>
                    <port>
                        <id>7</id>
                        <name>out</name>
                        <port_type>port-abstract</port_type>
                    </port>
                </ports>
                <links> <!-- forwarding requirement on the decomposed NF -->
                    <link>
                        <id>012345</id>
                        <name>requirement on NF delay</name>
                        <src>../../../ports/port[id=6]</src>
                        <dst>../../../ports/port[id=7]</dst>
                        <resources>
                            <delay>4 ms</delay>
                            <bandwidth>8 Mbps</bandwidth>
                        </resources>
                    </link>
                </links>
            </node>
        </NF_instances>
        <flowtable> <!-- decomposed forwarding rules : -->
            <flowentry>
                <id>f1</id>
                <port>../../../ports/port[id=0]</port>
                <match></match>
                <out>../../../NF_instances/node[id=NF1]/ports/port[id=2]</out>
                <action></action>
                <resources> <!-- resource requirement on the decomposed forwarding : -->
                    <delay>1 ms</delay>
                </resources>
            </flowentry>
            <flowentry>
                <id>f2</id>
                <port>../../../NF_instances/node[id=NF1]/ports/port[id=3]</port>
                <match></match>
                <out>../../../NF_instances/node[id=NF2]/ports/port[id=4]</out>
                <action></action>
                <resources>
                    <delay>1 ms</delay>
                </resources>
            </flowentry>
```

```
                <flowentry>
                    <id>f3</id>
                    <port   >../../../   NF_instances/node[id=NF2]/ports/port[ id =5]</ port>
                    <match></match>
                    <out   >../../../   NF_instances/node[id=NF3]/ports/port[ id =6]</ out>
                    <action ></ action >
                    <resources>
                        <delay>1  ms</delay>
                    </resources>
                </flowentry>
                <flowentry>
                    <id>f4</id>
                    <port   >../../../   NF_instances/node[id=NF3]/ports/port[ id =7]</ port>
                    <match></match>
                    <out   >../../../   ports/port[ id =1]</ out>
                    <action ></ action >
                    <resources>
                        <delay>1  ms</delay>
                    </resources>
                </flowentry>
            </ flowtable >
        </node>
    </nodes>
</ virtualizer >
```

With this approach, all decomposition rule types from [D3.1] are covered. Furthermore, allowing variables to be used in decomposition rules instead of constant values, e.g.

<delay>x\</delay>

instead of

<delay>5</delay>

enables formalization of requirements described in [D3.2a].

2.4.2 Results of Divine embedding algorithms

In the following we present our latest results considering the development of our efficient large-neighborhood search algorithm called Divine. In essence, Divine builds upon a series of novel Integer Programming (IP) formulations that enable us to model the embedding of virtual networks (and thereby also service chains). In particular, based on the classical IP for the Virtual Network Embedding Problem, we devise adaptions to allow for reconfigurations and elasticity. By reconfigurations we here specifically refer to migrating virtual nodes (or service functions) and adapting routing entries. Elasticity refers to the concept of being able to adapt the resource requirements of an already existing request. This is of particular interest, as requests may evolve over time and up or down scaling of services might be triggered to adapt to the customer's varying demand. As such, Divine offers the orchestration capabilities required for large-scale (re-)optimizations.

Our Divine algorithm employs the novel Integer Programming formulations as building blocks to construct a parallelized large-neighborhood search scheme (see Section 2.4.2.6). As shown in our extensive computational evaluation in Section 2.4.2.9, Divine might improve the quality of solutions dramatically, when compared with just solving the formulation using commercial solvers. Indeed, our computational evaluation shows an improvement of up to 40% (see our evaluation below) when resources are scarce.

2.4.2.1 Integer Programming Models: Enabling Reconfigurations and Elasticity

In this section we show how the classical integer programming (IP) formulation can be adapted to cater for reconfigurations and elasticity. In particular, after having introduced some notation, we will first present the classical IP formulation and then add reconfigurability of embedded requests and then incorporate elasticity, i.e. the ability to change the specification of requests.

2.4.2.2 Notation

We consider the problem of embedding (simultaneously) a set K of VNet requests, each represented by a graph (V^k, E^k) with $k \in K$, in a substrate network $(V^{\mathcal{S}}, E^{\mathcal{S}})$. We denote by G^k and $G^{\mathcal{S}}$ the union of the network elements $V^k \cup E^k \, \forall k \in K$ and $V^{\mathcal{S}} \cup E^{\mathcal{S}}$, respectively.

We consider arbitrary resource types for both links and nodes, denoting by $\mathcal{T}^V$ the set of node resource types (e.g., CPU, RAM, storage) and by $\mathcal{T}^E$ (mainly bandwidth) the set of edge resource types. For each type of resource $\tau \in \mathcal{T} = \mathcal{T}^V \cup \mathcal{T}^E$, each substrate network element $j \in G^{\mathcal{S}}$ has a capacity $c_{j\tau} \geq 0$. Accordingly, each virtual network request $k \in K$ demands an amount of resources $d^k_{i\tau} \geq 0$ for each of its network elements $i \in G^k$ and each resource type τ. Note that to keep notation short, we do not explicitly state that a node cannot specify link resources and vice versa. Using these types, arbitrary node and edge specific resource types can be modeled.

In the following, we will denote the set of ingoing and outgoing substrate edges of a substrate node $v \in V^{\mathcal{S}}$ by δ^-_v and δ^+_v, respectively.

2.4.2.3 The VNEP Model

We use the following constructs to solve the VNEP problem. We introduce the following decision variables:

1. Binary variables $x^k \in \{0, 1\}$ to indicate whether request $k \in K$ is embedded or not;
2. Binary variables y^k_{iv} to indicate whether a virtual node $i \in V^k$ of request k is embedded in a substrate node $v \in V^{\mathcal{S}}$. Each virtual node i of a satisfied request must be embedded in exactly one substrate node $v \in V^{\mathcal{S}}$. Furthermore, the customer may restrict the locations where i may be embedded in by whitelisting a subset of allowed substrate nodes $V^{k,\mathcal{S}}_v \subseteq V^{\mathcal{S}}$.
3. Binary variables $z^k_{le} \in \{0, 1\}$ to indicate whether a virtual link $l = (i, j) \in V^k$ of request k is embedded in a substrate edge $e \in E^{\mathcal{S}}$. We stress that the embedding of l corresponds with identifying a single path p in the substrate graph, which connects the substrate nodes where i and j are embedded in. Therefore, l may be mapped to a multiplicity of substrate edges, namely those that constitute the path p.

Using these decision variables, we can model the static virtual network embedding problem as the Integer Program 1. Constraints (1) impose that, if a request is embedded, then each virtual node of the request must be embedded in exactly one substrate node. Constraints (2) model the embedding of a virtual link $l = (i, j)$ under the form of flow conservation constraints that represent the flow sent through the substrate network between the two substrate nodes where i, j are embedded in. Constraints (3) and (4) enforce that the capacities of substrate links and nodes are not exceeded. Finally, (5) are logical constraints forcing the link embedding variables to 0 when the corresponding request embedding variable is 0.

Integer Program 1 can be complemented with different objective functions. A natural and often studied objective is profit maximization: the substrate provider aims to maximize the total revenue (given by the set of accepted requests)

Integer Program 1: Static VNEP

$$\forall\, k \in K, i \in V^k \qquad x^k = \sum_{v \in V_v^{k,\mathcal{S}}} y_{iv}^k \tag{1}$$

$$\forall\, k \in K, v \in V^{\mathcal{S}}, l=(i,j) \in E^k \qquad y_{iv}^k - y_{jv}^k = \sum_{e \in \delta_v^+} z_{le}^k - \sum_{e \in \delta_v^-} z_{le}^k \tag{2}$$

$$\forall\, v \in V^{\mathcal{S}}, \tau \in \mathcal{T}^V \qquad c_{v\tau} \geq \sum_{k \in K, i \in V^k} y_{iv}^k\, d_{i\tau}^k \tag{3}$$

$$\forall\, e \in E^{\mathcal{S}}, \tau \in \mathcal{T}^E \qquad c_{e\tau} \geq \sum_{k \in K, l \in E^k} z_{le}^k\, d_{l\tau}^k \tag{4}$$

$$\forall\, k \in K, e \in E^{\mathcal{S}}\;\; l \in E^k \qquad x^k \geq \sum z_{le}^k \tag{5}$$

minus the total cost (given by the resource cost):

$$\max\ F = \sum_{k \in K} \pi^k x^k - \sum_{j \in G^{\mathcal{S}}, \tau \in \mathcal{T}} \gamma_{j\tau} R_{j\tau} \tag{Obj-Profit}$$

where π^k is the revenue derived from satisfying request k, and where $\gamma_{j\tau}$ is the cost of using one unit of resource of type τ in network element j, multiplied by the total use of resources $R_{j\tau}$ of τ in j.

To be comparable to other orchestration algorithms in the VNEP context, we have decided to explicitly model latency requirements. However, additive constraints on links or nodes can be easily added (see e.g. [MKK14]).

2.4.2.4 Supporting Reconfigurations

Given the basic program for the static VNEP, let us now introduce mechanisms to support reconfigurations. We extend the above stated problem by identifying the subset $K^{\texttt{OLD}} \subseteq K$ of requests that are already embedded in the substrate, and hence reduce the available capacity for new requests $K \backslash K^{\texttt{OLD}}$. We will call such requests old. For each old request $k \in K^{\texttt{OLD}}$, we denote: 1) by $v_i^k \in V^{\mathcal{S}}$ the substrate node in which the virtual node $i \in V^k$ is embedded; 2) by $E_l^k \subseteq E^{\mathcal{S}}$ the subset of substrate edges in which the virtual link $l \in E^k$ is embedded.

We say that an old request is reconfigured or migrated when the embedding of at least one of its elements (virtual nodes or edges) is modified. To indicate whether a node or a link have been reconfigured, we introduce the decision variables $\mu_i^k \in \{0,1\}$ for virtual elements $i \in G^k$ and $k \in K^{\texttt{OLD}}$: $\mu_i^k = 1$ if i is reconfigured and is 0 otherwise. For each $k \in K^{\texttt{OLD}}$, we also introduce a global reconfiguration variable μ^k, which is 1 if $k \in K^{\texttt{OLD}}$ is reconfigured and 0 otherwise.

We extend the VNEP formulation of Integer Program 1 by introducing additional decision variables for reconfigurations, obtaining Integer Program 2): here, we first need to introduce constraints (6) to impose that old requests must be embedded and cannot be preempted. Constraints (7) and (8) express the logical connections between the global reconfiguration variable and the single-element reconfiguration variables for every old request: if any single-element reconfiguration variable is activated, then the global variable must be activated as well; if the global variable is activated, then at least one single-element reconfiguration variable must be activated.

Constraints (9- 11) are logical constraints that enforce old node and link mappings, when no respective reconfigurations take place: (9) impose that the virtual node $i \in V^k$ is mapped to its old substrate node $v_i^k \in V^{\mathcal{S}}$, if no

Integer Program 2: VNEP with Reconfigurations

$$
\begin{array}{rll}
 & (1) - (5) & \\
\forall\, k \in K^{\mathtt{OLD}} & x^k = 1 & (6)\\
\forall\, k \in K^{\mathtt{OLD}}, i \in G^k & \mu^k \leq \sum_{i \in G^k} \mu_i^k & (7)\\
\forall\, k \in K^{\mathtt{OLD}} & \mu^k \geq \mu_i^k & (8)\\
\forall\, k \in K^{\mathtt{OLD}}, i \in V^k & y^k_{iv_i^k} \geq 1 - \mu_i^k & (9)\\
\forall\, k \in K^{\mathtt{OLD}}, l \in E^k, e \in E_l^k & z_{le}^k \geq 1 - \mu_l^k & (10)\\
\forall\, k \in K^{\mathtt{OLD}}, l \in E^k, e \in E^{\mathcal{S}} \setminus E_l^k & \mu_l^k \geq z_{le}^k & (11)\\
 & M^V \geq \sum_{k \in K, i \in V^k} \mu_i^k & (12)\\
 & M^E \geq \sum_{k \in K, l \in E^k} \mu_l^k & (13)
\end{array}
$$

reconfiguration takes place; (9) are similar constraints defined for enforcing the use of old substrate links when a link reconfiguration does not take place (note that these constraints are complemented by (11) imposing that if a non-old substrate link is used for some embedded virtual link, then the corresponding reconfiguration variable must be activated). Lastly, constraints (12) and (13) impose upper bounds on the number of node and link reconfigurations that may take place.

The possibility to reconfigure a virtual network introduces a tradeoff: among a set of feasible embeddings, some embeddings may be more desirable as they yield, e.g., a low load or a low resource utilization, however, migrating to the corresponding configurations can be costly. Thus, a reconfiguration algorithm should find a good tradeoff between embedding and reconfiguration costs. While the best tradeoff may depend on the scenario, a wide spectrum of objectives can be supported: e.g., minimizing migration costs may be prioritized over the embedding costs, or embedding cost may be prioritized over migration costs, or any tradeoff between. For example, the objective function (Obj-Profit) can simply include a new cost summation reflecting the costs γ_i^k of reconfiguring elements $j \in G^{\mathcal{S}}$ of request $k \in K^{\mathtt{OLD}}$:

$$
\max\ F - \sum_{k \in K^{\mathtt{OLD}}, i \in G^k} \gamma_i^k\, \mu_i^k \qquad \text{(Obj-Min. Reconfig.)}
$$

2.4.2.5 Enabling Elasticity

We can now complete the model by adding support for the following additional operations:

1. Add new and/or remove existing virtual nodes and links.

2. Increase and/or decrease the resource demand of virtual links and nodes.

Note that while downgrade requests are obviously always satisfiable, increasing the resource demands or adding new virtual elements may not be feasible.

We model reconfigurations as follows. For an existing request $k \in K^{\mathtt{OLD}}$, a new specification $(V^k_{\mathtt{UP}}, E^k_{\mathtt{UP}})$ with corresponding resource demands $\tilde{r}^k_{i\tau}$ for $i \in G^k_{\mathtt{UP}}$ and node placement restrictions $V_v^{k,\mathcal{S}}$ for $v \in V^k_{\mathtt{UP}}$ is specified.

We denote the subset of old requests demanding an updated by $K^{\text{UP}} \subseteq K^{\text{OLD}}$. Given an updated request $k \in K^{\text{UP}}$, the subset $V^k_{\text{UP}} \cap V^k$ includes nodes of k that are not updated, the subset $V^k_{\text{UP}} \setminus V^k$ includes nodes that demand an update, whereas the subset $V^k \setminus V^k_{\text{UP}}$ includes nodes that are removed in k with respect to the corresponding old request. Similar considerations can be made for the two sets of updated and old links E^k_{UP}, E^k.

As multiple update requests may arrive simultaneously and not all of them can be satisfied, we present Integer Program 3 which extends Integer Program 2 to also support the identification of request subsets K^{UP} which can actually be satisfied.

For modeling purposes, to take into account the presence of the updated requests K^{UP}, we define a modified overall set of requests K' that replaces K and is defined as follows:

$$K' = \underbrace{(K \setminus K^{\text{OLD}})}_{\text{new}} \cup \underbrace{(K^{\text{OLD}} \setminus K^{\text{UP}})}_{\text{old non-updated}} \cup \underbrace{(\Gamma^{\text{UP}} \cup K^{\text{UP}})}_{\text{old requesting update}}$$

K' is thus the union of the set $K \setminus K^{\text{OLD}}$ of new requests, the set $K^{\text{OLD}} \setminus K^{\text{UP}}$ of old requests that do not demand updates and the set $\Gamma^{\text{UP}} \cup K^{\text{UP}}$ of old requests demanding an update. The last set of the union determining K' needs a specific explanation: it involves the set Γ^{UP}, namely a modeling artifact that we adopt in the optimization model to represent that an old request demanding an update can be either satisfied (thus we must embed the corresponding updated request) or cannot be satisfied (thus we must embed the corresponding old request).

Given an updated request $k \in K^{\text{UP}}$, we denote by k^{UP} the corresponding "artificial" request that we include in Γ^{UP} and whose features correspond with the updated specifications, namely $V^{k^{\text{UP}}} = V^k_{\text{UP}}$, $E^{k^{\text{UP}}} = E^k_{\text{UP}}$, $d^{k^{\text{UP}}}_{i\tau} = \tilde{r}^k_{i\tau}$ and $V^{k^{\text{UP}},\mathcal{S}}_v = V^{k,\mathcal{S}}_v$.

Using the new set K', the optimization model that also includes updates is depicted as Integer Program 3. Constraints (14) impose that all old requests that do not ask for an update must be embedded. Constraints (15) state that for each old request demanding an update, either the old request (when the update is not accepted) or the updated request (when the update is accepted) must be embedded and thus exactly one of the two binary request variables must be activated. Constraints (16) are the canonical embedding constraints that every request must satisfy. Constraints (17) and (18) link the embedding and the reconfiguration variables as in Integer Program 2. The three families of constraints (19-21) establish the relation between the node and link embedding variables and the reconfiguration variables, taking into account the old embedding. We remark that these constraints are similar to constraints (9-11): the difference lies in the right-hand-sides of (19), (20), where the embedding variable x^k substitutes value 1 and is needed to activate the constraints for old non-updated requests $K^{\text{OLD}} \setminus K^{\text{UP}}$ and for requests K^{UP} that do not receive the update (i.e., when $x^{k^{\text{UP}}} = 0$). The three following families of constraints (22-24) have a similar role: they establish the relation between old and new configuration for node and links that are maintained in an updated requests in K^{UP}. Also here, we remark the presence of a binary variable $x^{k^{\text{UP}}}$ in the right-hand-sides of the constraint that has the task of activating/deactivating the constraints. Finally, (25), (26) impose the upper bound on the number of reconfigurations that can be operated.

To take into account updates in the objective function, we extend Obj-Min. Reconfig. by including a new summation representing the additional profits π^k_{UP} coming from satisfying updated requests:

$$\max\ F - \sum_{k \in K^{\text{OLD}}, i \in G^k} \gamma^k_i \mu^k_i + \sum_{k \in K^{\text{UP}}} \pi^k_{\text{UP}} x^{k^{\text{UP}}} \qquad \text{(Obj-Min. Updates)}$$

Integer Program 3: Elastic VNEP with Reconfigurations

$$\forall\, k \in K^{\mathsf{OLD}} \setminus K^{\mathsf{UP}} \qquad x^k = 1 \tag{14}$$

$$\forall\, k \in K^{\mathsf{UP}} x^k + x^{k^{\mathsf{UP}}} = 1 \tag{15}$$

$$(1) - (5) \text{ on } K' \tag{16}$$

$$\forall\, k \in K^{\mathsf{OLD}}, i \in G^k \qquad \mu^k \leq \sum_{i \in G^k} \mu_i^k \tag{17}$$

$$\forall\, k \in K^{\mathsf{OLD}} \qquad \mu^k \geq \mu_i^k \tag{18}$$

$$\forall\, k \in K^{\mathsf{OLD}}, i \in V^k \qquad y_{iv_i^k}^k \geq x^k - \mu_i^k \tag{19}$$

$$\forall\, k \in K^{\mathsf{OLD}}, l \in E^k,\ \ e \in E_l^k \qquad z_{le}^k \geq x^k - \mu_l^k \tag{20}$$

$$\forall\, k \in K^{\mathsf{OLD}}, l \in E^k, e \in E^{\mathcal{S}} \setminus E_l^k \qquad \mu_l^k \geq z_{le}^k \tag{21}$$

$$\forall\, k \in K^{\mathsf{UP}}, i \in V^k \cap V^{k^{\mathsf{UP}}} \qquad y_{iv_i^k}^{k^{\mathsf{UP}}} \geq x^{k^{\mathsf{UP}}} - \mu_i^k \tag{22}$$

$$\forall\, k \in K^{\mathsf{UP}}, l \in E^k \cap E^{k^{\mathsf{UP}}}, e \in E_l^k \qquad z_{le}^{k^{\mathsf{UP}}} \geq x^{k^{\mathsf{UP}}} - \mu_l^k \tag{23}$$

$$\forall\, k \in K^{\mathsf{UP}}, l \in E^k \cap E^{k^{\mathsf{UP}}}, e \in E^{\mathcal{S}} \setminus E_l^k \qquad \mu_l^k \geq z_{le}^{k^{\mathsf{UP}}} \tag{24}$$

$$M^V \geq \sum_{k \in K, i \in V^k} \mu_i^k \tag{25}$$

$$M^E \geq \sum_{k \in K, l \in E^k} \mu_l^k \tag{26}$$

2.4.2.6 Outline of Divine Algorithm

Since already the classical virtual network embedding problem is NP-hard to solve, solving the above stated Integer Programs is also at least NP-hard [Fis+13]. Nonetheless, as commercial solvers for Integer Programs have been developed over the course of decades, these solvers can generally compute high quality solutions within a reasonable time frame and for limited problem sizes.

In our initial evaluation reported in UNIFY deliverable D3.2 [D3.2], we have surveyed the empirical complexity of solving such virtual network embedding instances depending on several generation parameters. We have showed that already for 100 requests consisting of 5 to 10 nodes, the solvers often only find solutions in the range of 10% to 30% of optimality (also denoted as relative objective gap[1]). The complexity of the respective embedding problem becomes apparent when considering that under an uniform node distribution and a probability of 0.3 that a pair of nodes is connected by an edge, the solver needs to find mappings for around 750 nodes and 1500 edges overall.

As the constraint matrix of the above Integer Programs grow linearly with the number of requests, even solving the initial linear relaxation for 100 requests may take up to 180 seconds. Other preliminary experiments have shown that with 250 requests consisting of 5 to 15 nodes, the time compute the linear relaxations already exceeds 600 seconds. Hence, there are natural limits with respect to the problem size for the pure Integer Programming approach to work within reasonable time constraints.

One of the most promising options to obtain high-quality solutions to many NP-hard problems is the local search

[1]Solving integer programs generally relies on branch-and-bound methods, where bounds are computed in the form of linear relaxations. Hence, given any solution it is possible to upper bound the distance to the optimal solution

approach. Given an initial solution, a local neighborhood of the current solution is searched for potential improvements. If the local changes are limited to e.g. changing a single node location or a single edge embedding, a single local search step may always be executed in polynomial time. Based on the sheer number of (virtual) nodes and (virtual) edges to be mapped, applying small-scale steps will be computationally expensive and will most probably only yield local optima. Furthermore, the proposed problem formulations contain different levels of decisions which are also represented in our Integer Programming formulations. Concretely, the high-level decisions of whether to embed a new request, reconfigure an existing request or upgrade an elastic request are decisions that require solving the NP-hard embedding task itself. Based on the above observations, we have devised the following large-neighborhood local search procedure.

2.4.2.7 Large-Neighborhood Local Search Procedure

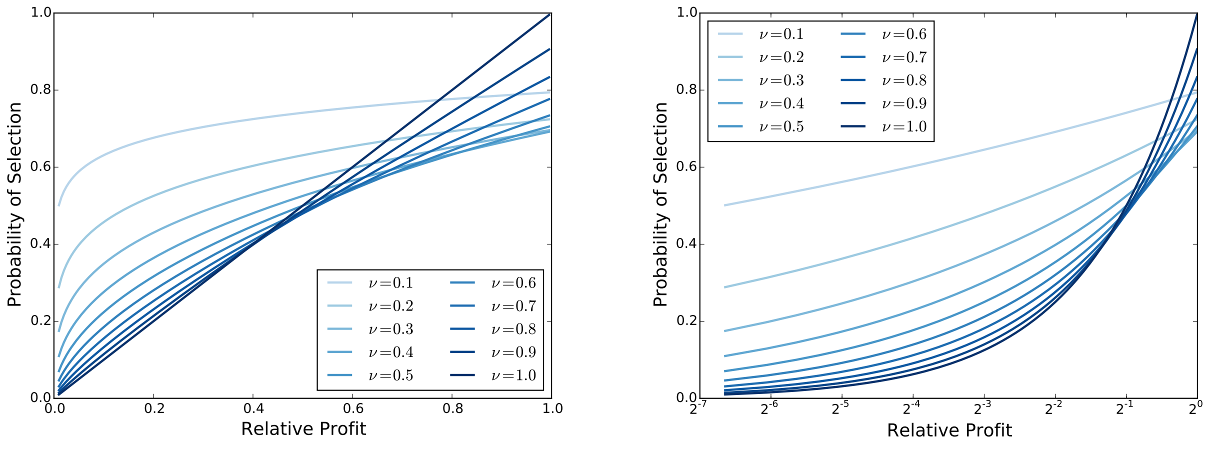


Figure 11: Visualization of the selection probability (see Equation 27) for different normalization values ν. The relative profit denotes the quotient $\Delta_p(k)/\max_{k' \in K} \Delta_p(k')$. The left shows the normalization effect over a linear x-axis, while the right one depicts a logarithmic scale.

1. Given an existing solution, the linear relaxation of the respective problem is computed under the additional constraint that only some percentage of requests' embeddings may be changed. This relaxation thereby yields the information on the set of requests whose adaption yields the maximal benefit.

2. Given these fractional solutions, a probabilistic scheme is employed to first select a subset of requests that yielded an improvement in the objective. Concretely, each of the requests $k \in K$ for which the profit has improved is chosen with the probability

$$\mathbb{P}(k \text{ is selected}) = \left(\nu \cdot \Delta_p(k) / \max_{k' \in K} \Delta_p(k') \right)^{\nu} \tag{27}$$

   where $\Delta_p(k)$ denotes the improvement of the profit and $\nu \in [0,1]$ is a normalization factor. The effect of the normalization factor is visualized in Figure 11: a small value equalizes the probability of selecting a request while a higher values increasingly favor the requests with the highest profit. After having selected a certain number of requests according to this probabilistic scheme, this set is then extended by a certain number of requests correlating resource-wise. Concretely, for each pair of (selected) profit request and any other nonselected request the metric

$$\Delta(k_p, k_o) = \alpha \cdot \left(\sum_{\substack{u \in V^{\mathcal{S}} \\ \Delta_{k_p}(u) > 0}} \max(0, -\Delta_{k_p}(u) \cdot \Delta_{k_o}(u)) \right) + \beta \cdot \left(\sum_{\substack{e \in E^{\mathcal{S}} \\ \Delta_{k_p}(e) > 0}} \max(0, -\Delta_{k_p}(e) \cdot \Delta_{k_o}(e)) \right) \tag{28}$$

is used, where k_p is a profitable request and k_o is any other (non-profitable) request and $\Delta_k(u)$, $\Delta_k(e)$ denotes the changes in the substrate allocations on node $u \in V^{\mathcal{S}}$ or edge $e \in E^{\mathcal{S}}$ of request $k \in K$, and $\alpha, \beta \in \mathbb{R}$ are weights for node and edge changes respectively. In the above formula only nodes and links are considered for which the allocation with respect to the profit request k_p has increased. If for the other request the resource allocation has decreased, then the products $-\Delta_{k_p}(u) \cdot \Delta_{k_o}(u)$ or $-\Delta_{k_p}(e) \cdot \Delta_{k_o}(e)$ yield positive values larger 0.

3. For the set of requests selected (profit plus correlation requests) the Integer Program 3 is constructed and solved, while enforcing that the objective (with respect to the existing solution) must improve. Hence, if a solution for the subset of requests is found, then this yields an improved solution for the overall problem.

To improve scalability with larger problem sizes and improve the chances of finding an improved solution, we have parallelized the execution of the large-neighborhood local search procedure as follows:

1. The LP-relaxation as outlined in the first point above is computed centrally.
2. For each physical core, the steps 2. and 3. are executed in parallel while any found solution is sent to the coordinating process.
3. The coordinator receives the solutions and reacts by iteratively diminishing the time limits for the spawned processes as follows. Initially, the subprocesses are given no time limit (except the global one). Once any of the processes has found a solution a timer is started. If the timer runs out without any new solution being found, then the spawned processes are terminated and the coordinator merges the results as described in the next step. Otherwise, i.e. if another solution was found, the timer is reset after having run out while decreasing the allowed time by a 'timeout decrease' factor.
4. If any of the processes has found a solution the existing solutions are merged using a novel multi-dimensional knapsack Integer Program 4 outlined below.
5. The above process is iterated as long as the global time limit is not reached.

2.4.2.8 Multi-Dimensional Knapsack

As outlined above multiple processes are started executing the large-neighborhood local search. Each of this processes reports any found solution to the coordinating process. The coordinating process therefore obtains for each of the requests $k \in K$ (independent of the type) a set of solutions $\mathcal{S}_k = \{S_{k,1}, \cdot, S_{k,n}\}$ is given. Each such solution represents a possible mapping of the respective request and is attributed with the following values:

$p_{k,i} \in \mathbb{R}$ These values denote the net profit that is achieved by choosing the solution $S_{k,i}$, i.e. the profit of embedding the request (or adapting it in the case of elasticity) minus the reconfiguration costs.

$l_{k,i} : \left(\begin{array}{c} V^{\mathcal{S}} \times \mathcal{T}^V \\ \cup E^{\mathcal{S}} \times \mathcal{T}^E \end{array} \right) \to \mathbb{R}$ These functions define the load on the substrate nodes and edges induced by choosing the i-th solution $R_{k,i}$.

$M^V_{k,i}, M^E_{k,i} \in \mathbb{N}$ These single numbers per solution denote the number of reconfigurations necessary for selecting the i-th solution $S_{k,i}$ with respect to the nodes and edges.

The multi-dimensional knapsack formulation is presented as Integer Program 4 and uses a single variable $x_{k,i}$ for each solution $S_{k,i} \in \mathcal{S}_k$ that indicates whether the i-th solution was chosen for the embedding. Note again that the set $\mathcal{S}_k$ is defined on the original set of requests K, such that it may contain solutions of different types: an elastic request $k \in K^{\mathsf{UP}}$ whose novel specification was accepted may be included in $\mathcal{S}_k$ as well as a reconfigured embedding of the old request.

The Constraint 30 states that at most one of the solutions may be chosen for new requests that were not embedded before. Similarly, Constraint 31 states that for each old (including the elastic ones) request exactly one solution must be chosen. Lastly, Constraints 32-35 safeguard that neither node or edge capacities are violated and that the maximal number of node or edge reconfigurations is not exceeded.

Integer Program 4: Multi-Dimensional Knapsack for Combining the found results

$$\max \sum_{k \in K} \sum_{S_{k,i} \in \mathcal{S}_k} p_{k,i} \cdot x_{k,i} \tag{29}$$

$$\forall\, k \in K \setminus K^{\mathsf{OLD}} \qquad \sum_{\forall S_{k,i} \in \mathcal{S}_k} x_{k,i} \leq 1 \tag{30}$$

$$\forall\, k \in K^{\mathsf{OLD}} \qquad \sum_{\forall S_{k,i} \in \mathcal{S}_k} x_{k,i} = 1 \tag{31}$$

$$\forall u \in V^{\mathcal{S}}, \tau \in \mathcal{T}^V \qquad \sum_{k \in K} \sum_{\forall S_{k,i} \in \mathcal{S}_k} l_{k,i}(u,\tau) \cdot x_{k,i} \leq c_{u\tau} \tag{32}$$

$$\forall e \in E^{\mathcal{S}}, \tau \in \mathcal{T}^E \qquad \sum_{k \in K} \sum_{\forall S_{k,i} \in \mathcal{S}_k} l_{k,i}(e,\tau) \cdot x_{k,i} \leq c_{e\tau} \tag{33}$$

$$\sum_{k \in K} \sum_{\forall S_{k,i} \in \mathcal{S}_k} M^V_{k,i} \cdot x_{k,i} \leq M^V \tag{34}$$

$$\sum_{k \in K} \sum_{\forall S_{k,i} \in \mathcal{S}_k} M^E_{k,i} \cdot x_{k,i} \leq M^E \tag{35}$$

2.4.2.9 Evaluation

In Section 2.4.2.1 the Integer Programs were introduced which can be used to solve the respective embedding problems optimally. Based on the scalability issues, we have devised the Divine algorithm and outlined its working in Section 2.4.2.6. In this section we now present our computational evaluation, showing that the Divine algorithm is effective in finding high quality solutions much quicker than the standard IP approach.

2.4.2.10 Scenario and Topologies

To minimize the number of different parameters in the scenario generation we only consider the instances already introduced in UNIFY deliverable D3.2 [D3.2]. In particular, we consider the subset of scenarios according to the cross product of the parameters defined in the following table.

Parameter	Values	Explanation
Topologies	{ Bellcanada, Geant, Uunet }	As discussed in deliverable D3.2 we use real-world ISP topologies (Bellcanada, Uunet) and the research-network Geant exported from the topology zoo. The chosen topologies are visualized in Figure 12
Number of requests	$\{100\}$	For each request, the number of nodes is uniformly drawn from the set $\{5, 6, \ldots, 10\}$
Connection Probability	$\{0.3\}$	Virtual edges between any two virtual nodes are created uniformly at random with a probability of 0.3.
Node Resource Factor	$\{1.0\}$	This (single) factor states that in expectation, all node demands can be satisfied. As this is only true in expectation and the mapping of virtual nodes is restricted to model functions, this is generally not true.
Edge Resource Factor	$\{1.0, 4.0, 8.0\}$	To embed all requests in expectation, the hop distance between virtual nodes being connected by an edge may on average not be more than the inverse of this factor. Concretely, a value of 1.0 would dictate that the average hop distance over any virtual edge may not be more than one as otherwise resources were not sufficient. The value of 8.0 implies that if the average embedding of any virtual link has an hop count less than $1/8$, the available edge resources should be sufficient (in expectation) to embed all requests. This is justified as in the VNEP resources may be collocated on a single location.
Objective Skew	$\{0.3, 0.8, 1.5\}$	The expected profit of any request is set to the cumulative demanded link resources times the average substrate distance times the objective skew value. As costs for using links are proportional to the metric (i.e. geographical) distance, this implies that embedding a request does not pay off when the average (metric) distance of neighboring nodes is more than the objective skew value. Hence, a value of 0.3 forces embeddings to be rather local, while 1.5 allows for geographically more diverse deployments.
Variability	$\{0.1, 0.2, 0.5\}$	For the generation of the requests, the above explanations only give the expected values. The variability factor defines the interval around the expected value from which the actual values are drawn. A variability of 0.5 may therefore lead to way more diverse 'types' of requests than a factor of 0.1 would allow.

According to the above table, for each of the three topologies (visualized in Figure 12) we consider 27 scenarios overall.

2.4.2.11 Computational and Algorithmic Setup

All experiments were conducted on servers equipped with two 4-core Intel Xeon L5420 processors with a clock speed of 2.5 GHz and 16 GB of RAM running Ubuntu 12.04.

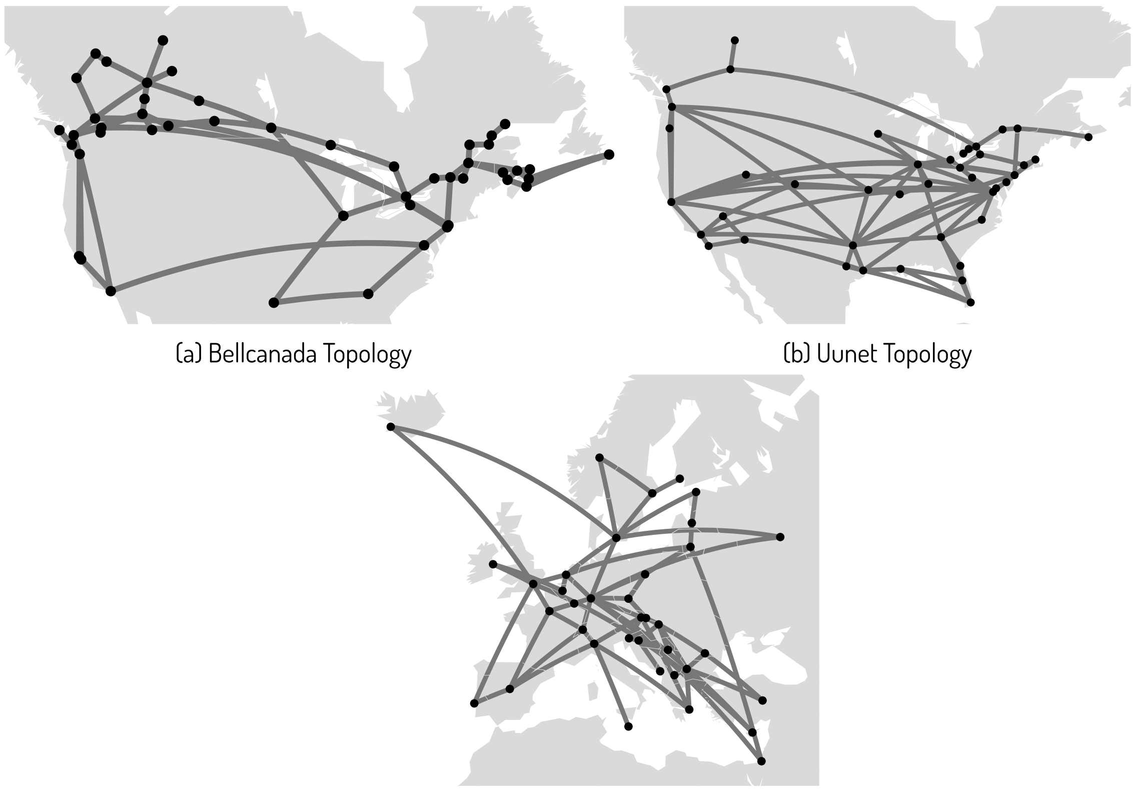

(a) Bellcanada Topology
(b) Uunet Topology
(c) Geant Topology

Figure 12: Visualization of selected topologies from the topology zoo [Kni+11].

We report on the performance of the Integer Program 3 implemented using the commercial solver Gurobi 5.6.2. In contrast to our initial experiments reported in deliverable D3.2 [D3.2], we have setup Gurobi to use all cores (i.e. 8 threads). Similarly, we allow Divine to spawn up to 8 subprocesses for executing the large-neighborhood search.

Divine is set up to select at most 12 of the 100 requests in any of the subprocesses. Subprocesses select normalization values ν in a round robin fashion from one of following equally spaced intervals $\{[0.0625, 0.5], [0.125, 1.0], [0.3125, 0.75], [0.5625, 1.0]\}$. For the resource correlation coefficients we set $\alpha = \beta = 0.5$ (see Formula 28). We set the general timeout value to 20 seconds and timeout decrease factors from the set $\{0.4, 0.6, 0.8\}$. Hence, after the first solution was found by any of the processes, the timeout after which the execution is aborted (unless another solution was found) amounts to $20 \cdot d^i$, where d is the decrease factor and i denotes the times the time limit has been extended.

2.4.2.12 Evaluation of Divine Parameters

As the first part of our evaluation, we consider the impact of the two different Divine parameters, namely the timeout decrease factor and the normalization interval, on the performance. The performance of the Divine algorithm is generally measured using the improvement in the relative objective gap and is depicted over time as the solution process of both Divine and Gurobi solving the standard Integer Program over time progresses. Hence, in the figures shown in this section, an improvement of 1% means that the solution found by Divine is at least 1% closer to the optimum profit than the solution found by Gurobi at the same point in time.

As an artifact of the Divine solution process, the performance of the Divine algorithm generally exhibits a sharp drop until approximately 200 seconds into the solution process. This is based on the fact that we only consider progress with respect to the main optimization problem at hand: while the subprocesses are indeed finding improved local solutions, these intermediate solutions are not reflected in the plots as solutions to the original problem are only found once the multi-dimensional knapsack has been executed.

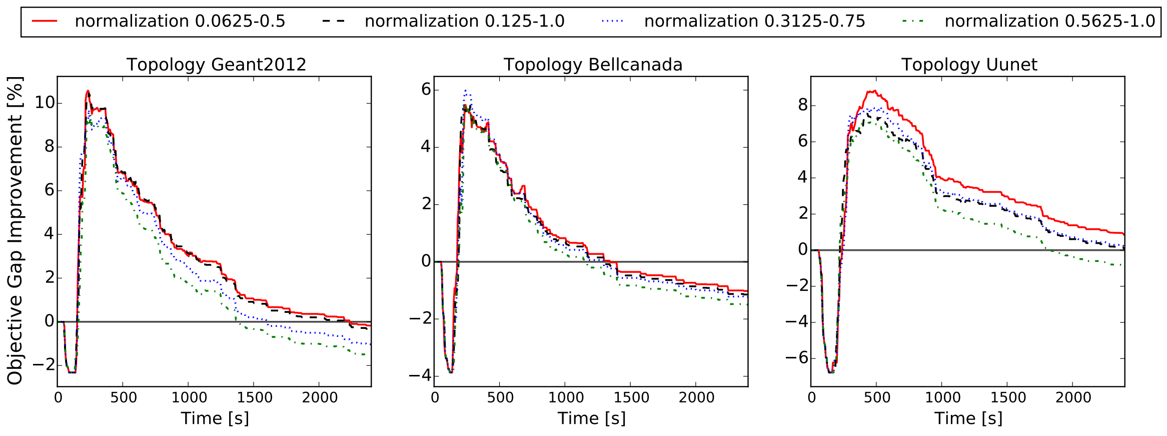


Figure 13: Impact of the different normalization intervals on the improvement in the relative objective gap over time on the different topologies.

In Figure 13 the impact of the different normalization ranges on the performance are depicted as averages over all scenarios and timeout decrease factors on the different topologies. The overall impact of this factor on the performance of Divine are rather slim. Nonetheless, the lowest normalization interval $[0.0625, 0.5]$ yields the best average performance, especially outperforming the other intervals on the Uunet topology by approximately one percent.

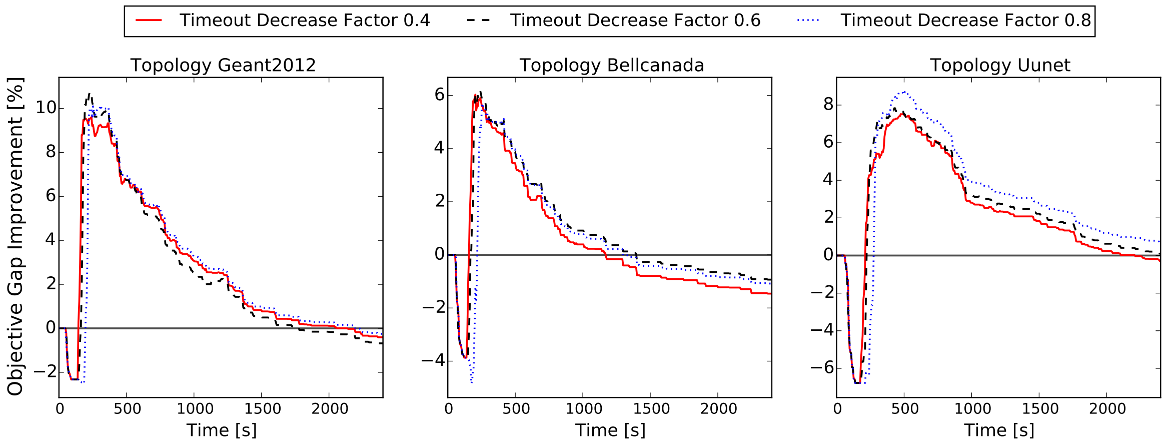


Figure 14: Impact of the timeout decrease factors on the improvement in the relative objective gap over time on the different topologies. While achieving a similar performance, the factor 0.8 especially on the Uunet topology yields the biggest improvement.

Figure 14 depicts the impact of the timeout decrease factor on the performance of the Divine algorithm. First note that the timeout decrease factor to some extent prolongs the "drop" in the initial phase, as the multi-dimensional-knapsack formulation is executed at a later point in time. However, giving the subproceses a little more computation time also reflects in better solutions, increasing the achieved improvement especially on the Uunet topology by up to 1%.

Based on the above observations, the timeout decrease factor of 0.8 and the normalization interval $[0.0625, 0.5]$ are determined to be the best parameters. The rest of our evaluation only focuses on this parameter combination. Accordingly, in Figure 15 the average overall performance on the different topologies is depicted. We note that once

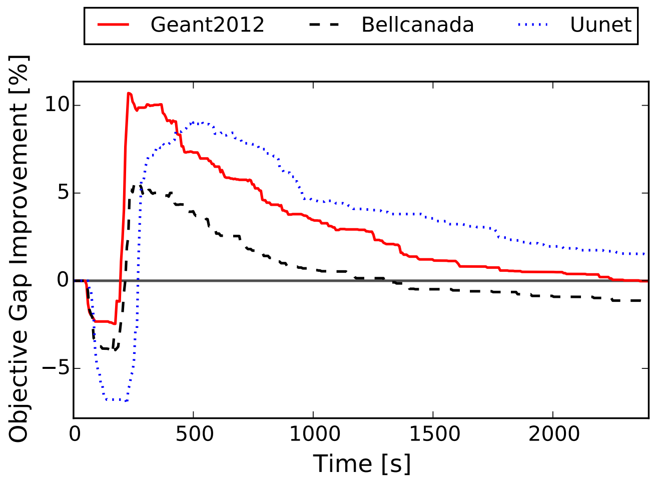


Figure 15: Overview on the improvement in the relative objective gap over time when using the Divine algorithm compared to the standard Integer Program with a timeout decrease factor of 0.8 and normalization interval $[0.0625, 0.5]$.

Divine determines the first solution between 200 and 300 seconds, an improvement in the objective gap of around 5% to 10% can be observed. On average, the commercial solver first outperforms Divine after 1200 seconds on the Bellcanada topology, while this point is achieved at 2400 seconds for Geant2012. To put these results into perspective, we will continue to analyze the performance as a function of the different generation parameters.

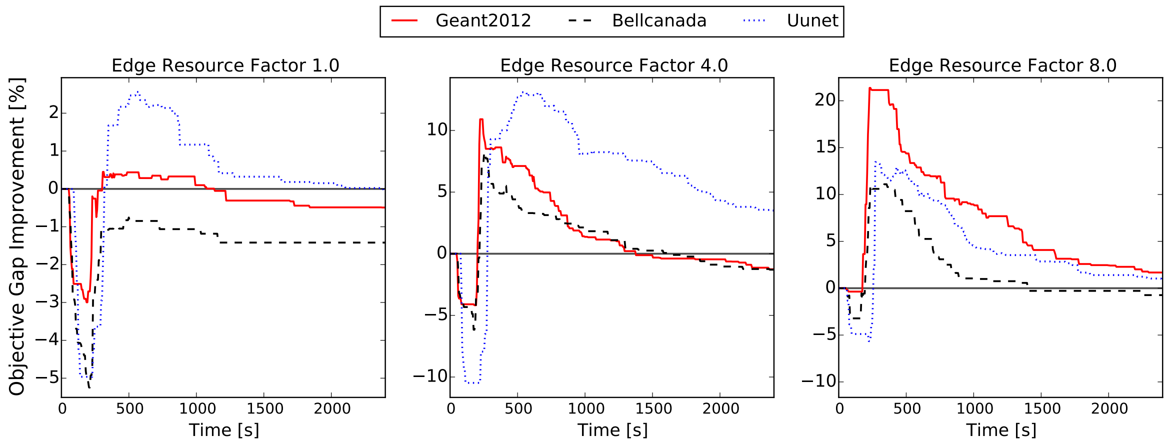


Figure 16: Overview on the improvement in the relative objective gap over time when using the Divine algorithm compared to the standard Integer Program with a timeout decrease factor of 0.8 and normalization interval $[0.0625, 0.5]$ depending on the different edge resource factor values.

Figure 16 depicts the average improvement in the objective gap depending on the respective edge resource factors. While Divine yields a diminishing performance improvement for the edge resource factor 1.0, the improvement for the factor 8.0 is staggering, reaching 20% on the Geant2012 topology. Using the edge factor 1.0, nearly all requests can be easily embedded by standard integer program, yielding very low objective gaps in the first place. On the other hand, once resources are scarce (edge factor of 8.0), the local search procedure produces much better solutions.

Figure 17 depicts the progress of Divine with respect to the 27 individual scenarios which are averaged in Figure 16. Importantly, the performance of the individual runs shows that Divine may yield substantial improvements on any of the three topologies while the worst performance of Divine may yield a decline in the objective gap of less than 5%.

Lastly, we also depict the improvement of the objective gap over time as a function of the other two generation parameters variability (see Figure 18) and objective skew (see Figure 19). While these parameters clearly have an

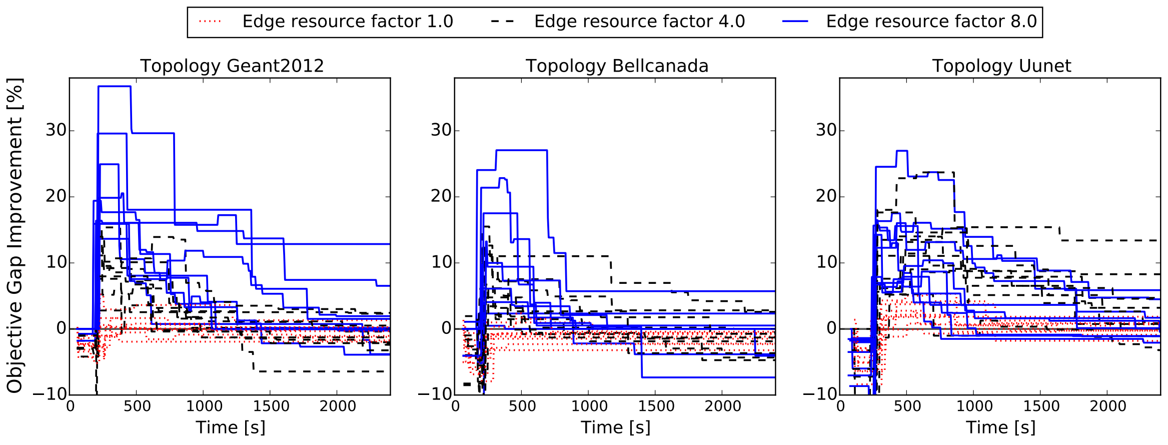


Figure 17: Individual performance of the different scenarios with a timeout decrease factor of 0.8 and normalization interval $[0.0625, 0.5]$. We highlight the edge resource factor generation parameter using different colors.

impact on the performance across the different topologies, the general order in the performance improvement does not change and lies again on average between 5% to 10%.

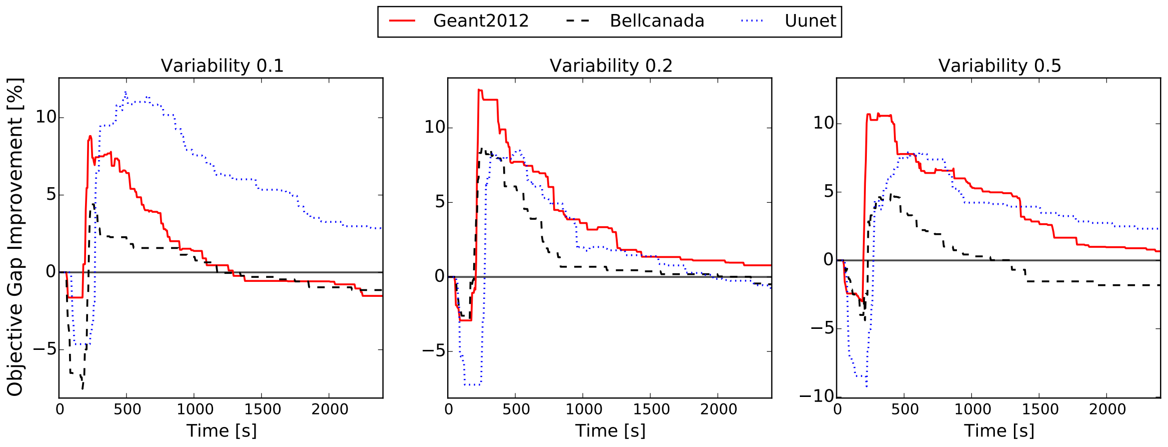


Figure 18: Overview on the improvement in the relative objective gap over time when using the Divine algorithm compared to the standard Integer Program with a timeout decrease factor of 0.8 and normalization interval $[0.0625, 0.5]$ depending on the different variability values.

2.5 Abstract interfaces

In this section, we highlight the abstract interfaces which have been defined in [D2.2] with the corresponding implementation status. [D3.4] provides further details on the implementation of different primitives. As $\mathrm{Cf\text{-}Or}$ interface is an important proposal of UNIFY to enable service elasticity, we devote a dedicated subsection to summarizing our main achievements on the mechanisms, use-cases and components related to this reference point.

In Table 2, we summarize the current state of the implementation regarding the relevant UNIFY reference points, namely $\mathrm{U\text{-}Sl}$, $\mathrm{Sl\text{-}Or}$, $\mathrm{Or\text{-}Ca}$ and $\mathrm{Cf\text{-}Or}$. In case of $\mathrm{Cf\text{-}Or}$, we have designed the concept and needed components, however, the implementation is in an initial phase. As we see, the most important functions related to WP3 and the service programming and orchestration framework have been covered (see service deployment column). Certain parts of the management system and Service Provider DevOps (SP-DevOps) support are missing here (see service operation column). These components (or selected ones) will be integrated by WP2 in IntPro (the project level integrated

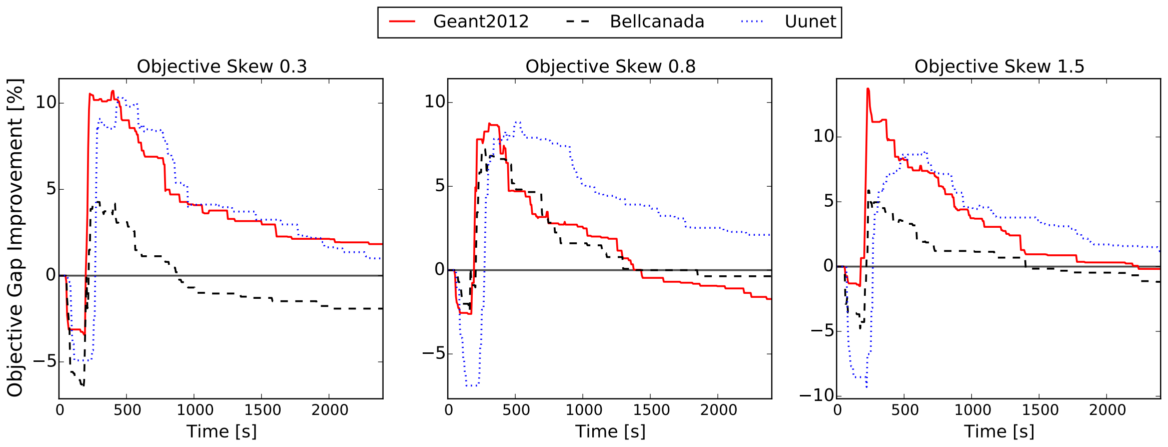


Figure 19: Overview on the improvement in the relative objective gap over time when using the Divine algorithm compared to the standard Integer Program with a timeout decrease factor of 0.8 and normalization interval $[0.0625, 0.5]$ depending on the different objective skew values.

prototype).

Table 2: Abstract interfaces

intf.	function	service deployment	service operation
U-Sl	request / release / update service	×	
	get / send service report	×	
	notification/alarm		×
	list SG	✓	
	request / release / update SG	✓	
	get / send SG info	×	
	list NFs from NFIB	✓	
	add / remove / update NF in NFIB	✓	
	add / remove OP to/from SG		×
	request / release UNIFY Resource Service		×
Sl-Or	instantiate / tear down / change NF-FG	✓	
	get / send virtual resource info	✓	
	notification/alarm		✓
	get / send observability info		✓
Or-Ca	instantiate / tear down / change NF-FG	✓	
	get / send virtual resource info	✓	
	notification/alarm		✓
	get / send observability info		✓
Cf-Or	instantiate / tear down / change NF-FG	(✓)	
	get / send virtual resource info	(✓)	
	notification/alarm		✓
	get / send observability info		✓

2.5.1 Cf-Or, enabling service elasticity: the Elastic Router use case

The goal of this part is to describe the mechanisms of the UNIFY Cf-Or interface when applied to a use case involving an elastic router. Here, it will be shown how the implementation of this Cf-Or interface creates the possibility for new service dynamics, different from current commercially available cloud platforms. Further on, the context of where this elastic router is usable will be also elaborated on. Next, several aspects of how the Cf-Or interface is involved in the UNIFY architecture and programmability framework are handled, this includes:

- Decomposition of (a part of) an NF into multiple components or NFs.
- The relation of decomposition to existing NF-FG model, i.e. the impact to the virtualizer data model.
- Service-driven orchestration by automated functionality (e.g., algorithms, etc.).
- Delegating control of individual parts of an NF-FG to the Control NF behind the Cf-Or.

2.5.1.1 UNIFY service scaling

Service scaling can be seen as a form of service decomposition, where the decomposition is specifically done to meet a changing resource request. This includes both horizontal and vertical scaling. In UNIFY, we have a generic concept of service decomposition as described in [D3.1; D3.2] and also recapitulated in section 2.4.1. The UNIFY service decomposition framework enables decomposition at the appropriate stages of the orchestration process. We consider white-box decompositions guided by exposed rules (e.g., an IDS might be decomposed using a Firewall and a Deep Packet Inspection component, a Firewall might be implemented by an Open vSwitch FW, an elastic router by Open vSwitch data paths with an OpenFlow controller etc.). These rules might be given by the Service Layer and stored in the NF-IB. It is a white-box approach because these decomposition rules can be visible and editable via the Service Layer, e.g. they can be extended by adding extra decompositions models at a later stage to the NF-IB. A second type of decomposition might be steered by a particular Control NF (black-box). The latter enables dynamic decomposition according to application-specific logic (e.g., dynamic decomposition into multiple NFs based on an internal learning algorithm). This can be seen as a black-box approach, since the logic performing the decomposition is internal to the Control NF and is not visible from outside via the NF-IB or Service Layer. This is illustrated in Figure 20, showing the 2 possibilities to store the decomposition models of a generic network service.

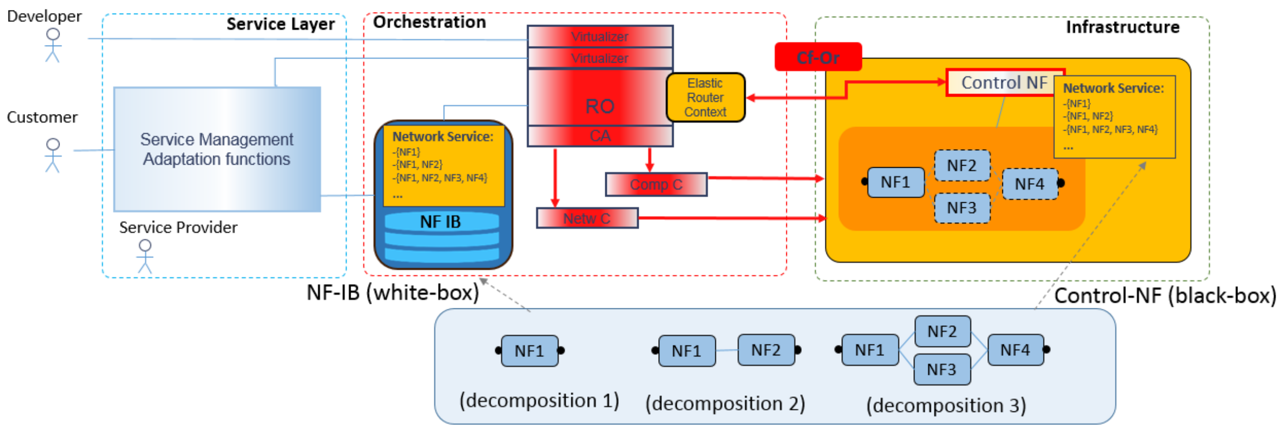


Figure 20: Two approaches to store service decomposition models in the UNIFY architecture: white-box, where decomposition rules are stored in the NF-IB (addressable through the Service Layer) vs. black-box, when decompositions are stored in the Control NF.

In this section, the focus will be mostly the Control NF driven decomposition. Service dynamics and elastic scaling actions are implemented using the Cf-Or interface. A Control NF (CtrlApp) is part of the deployed service and is in charge of triggering an elastic scaling action. The Control NF knows the decomposition rules and triggers their deployment. It does this by sending a modified NF-FG to the Orchestrator. This updated NF-FG is deployed and replaces the old one. This can be seen as a black-box service decomposition, since the logic performing the decomposition is internal to the Control NF and is not visible from outside. This process is illustrated by an elastic router use case. In section 2.5.1.8, we come back on how this differs from a NF-IB-based or white-box perspective, where all decomposed topologies are stored in the NF-IB. Furthermore, the UNIFY framework supports generic scaling and decomposition options as discussed in section 2.5.2. Any service specific logic related to scaling or decomposition, according to a specific topology, is programmable by the service developer through modification of the Control NF.

2.5.1.2 Elastic Router use case scenario

A router can for example be offered as a service itself, Routing-as-a-Service (RaaS), or be deployed as a functional part of a more advanced service. The elastic router might be a building block of a more complicated service, such as a VPN service, where a business is connecting its offices over the public internet. This is shown in Figure 21. It would need a L3VPN connecting N private LAN networks over the Internet, each requiring a full-duplex throughput R. In this context, the provider would need an elastic routing function in his offered VPN service such that:

1. The number of ports N scales to the number of requested VPN islands by the enterprise user.
2. It can dynamically adapt its traffic processing capacity to the required service needs (throughput R).
3. Hardware resources are assigned dynamically, optimized in function of the current service requirements (N, R).

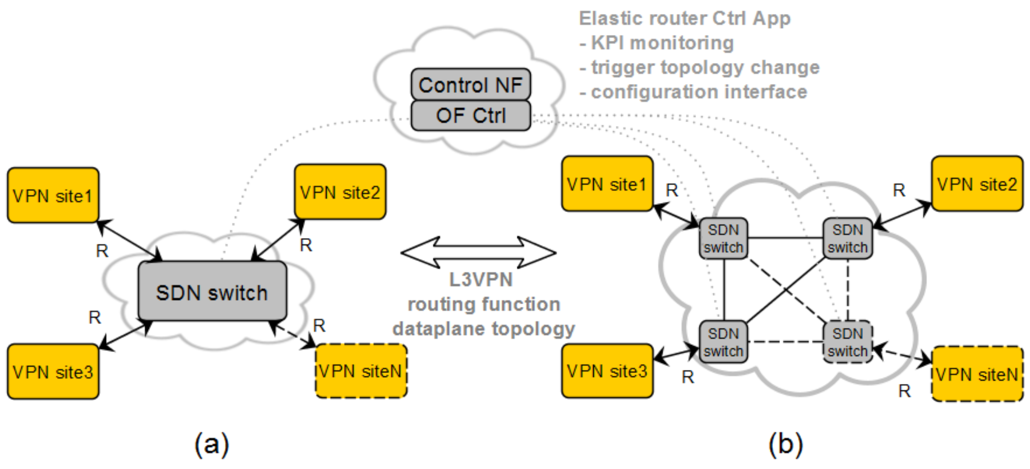


Figure 21: VPN service with an elastic router providing N ports. The dynamic creation of extra ports is illustrated by the dashed links/blocks. The Ctrl App triggers the change between (a) single switch instance and (b) a new switch topology (VLB) with distributed processing.

It would be interesting for a service provider to use a dedicated (logical) NF for this. It allows easier measurement of parameters like datavolume, needed for billing or Service Level Agreement (SLA) provisioning and isolate the routing table configurations from the switch fabric of the infrastructure provider. It enables an easier implementation of a service user interface to configure service parameters like the IP routing table, without interfering with the internal configurations of the underlying network managed by the infrastructure provider. By migrating dataplane instances to PoPs in the access network, link delay could be shortened. In the core network, the elastic router offers possibilities to enable high-rate throughput which would be impossible using a single dedicated server.

It can be noted that, using NFV, the functionality of the forwarding nodes in this topology can also be augmented from simple packet switching to more advanced packet processing (like firewalling), which will further increase the load when deployed on generic/multi-purpose servers in the datacenters or PoPs. This further justifies the need for elasticity in this service.

2.5.1.3 Elastic Router process flow

During the lifetime of a service, two different stages related to service decomposition can be distinguished. This is shown in Figure 22. At different moments, NF-IB or Control NF define the decomposed topologies that can be used:

1. At **initial service deployment**, the Service Graph is translated to an NF-FG by the Service Layer. This layer discovers the elastic router has a decomposed topology, as found in the NF-IB. As explained in 2.3, the NF-IB supports the definition of abstract NFs, which might be decomposed themselves, at a later stage, into multiple NFs interconnected into an NF-FG with the same external interfaces as the higher level NF. Next to this, the Service Layer also receives a virtualized infrastructure view from the Orchestrator via the Sl-Or interface. Via this same interface, an NF-FG is sent which contains the implementation template, where all VNFs present in the decomposed topology are connected to the virtualized infrastructure. It is further left to the Orchestrator to determine the best choice of infrastructure mapping in its own right. The initial service deployment process are the blue arrows in Figure 22. It can be seen that during the orchestration actions of the deployment, an initial decomposition is fetched from the NF-IB (Figure 22(a)).

2. At the **service runtime**, elastic scalability is possible. This is the process of deploying more or less resources in function of the workload or required performance of the service. This requires a control loop from the running service to the Orchestrator, to trigger and deploy the required hardware. The Cf-Or interface, exposed by the Orchestrator, full-fills this function. In UNIFY, it is chosen to connect to this interface from a Control NF, which is an actual part of the deployed service. The elastic control loop active during service runtime is depicted by the red arrows in Figure 22. Here, the decompositions (as seen in Figure 22(b) or (c)) are delivered to the Orchestrator by the Control NF. Complementary work done in WP4 [M4.3] describes an Orchestration dynamicity support process, using a UNIFY specific monitoring mechanism, called MEASURE. This process provides the Control NF the necessary information to trigger a scaling action or not.

In the UNIFY framework, it is argued that the Control NF is the most fit entity to implement service elasticity:

- It knows the service logic inherently; it is designed and developed as part of it. It knows its instantiated VNFs, the corresponding forwarding overlay, and an abstract view of the infrastructure resources through the RO's dedicated virtualizer view, learned via the Cf-Or interface.
- As the Control NF is deployed as an actual part of the service, it runs on the infrastructure close to the other NFs in the service and close to the Orchestrator controlling the infrastructure.
- Its position offers the best opportunities to gather custom monitoring data, which can be analyzed inside the Control NF and therefore does not need to propagate outside of the infrastructure layer. Which can otherwise pose privacy or security issues or increased network load.

The monitoring mechanism which triggers the Control NF is not specifically shown in Figure 23. Different observation points are deployed along with the different DPs in the elastic router, which provide the necessary monitoring

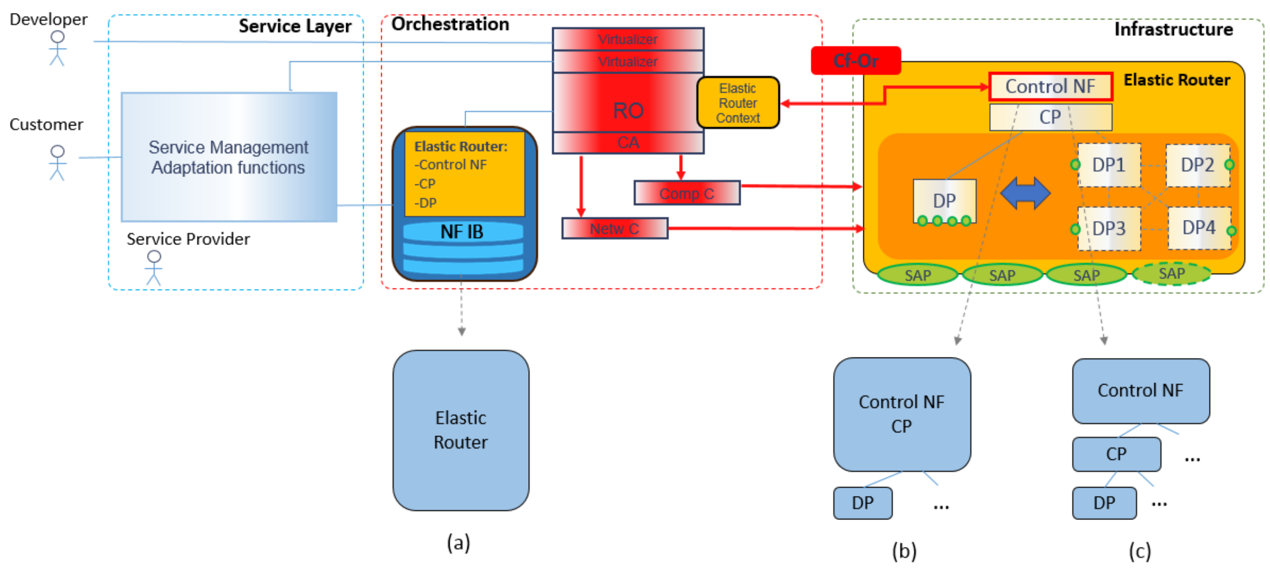


Figure 22: NF-IB (a) and Control NF decomposition (b),(c) during the elastic router's lifetime. (c) shows the decomposition where Control NF, CP and DP are all running in a separate VM/container.

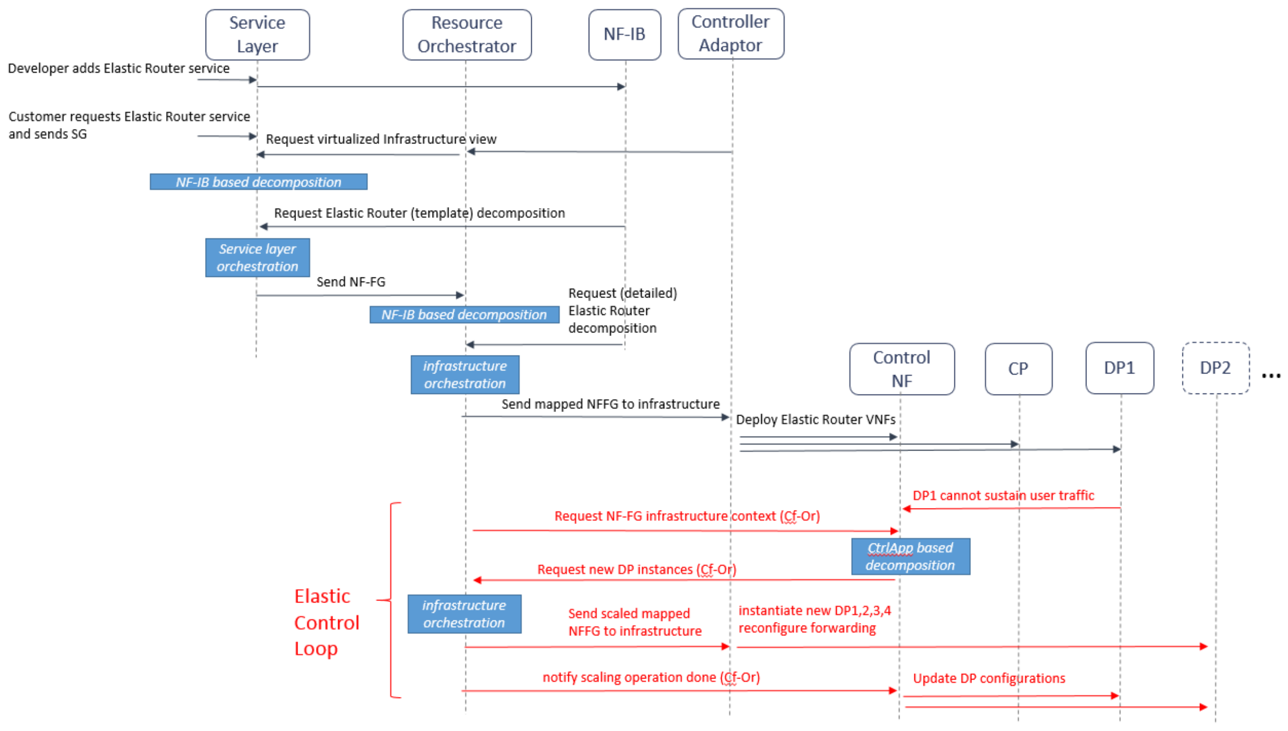

Service Layer
Resource Orchestrator
NF-IB
Controller Adaptor
Developer adds Elastic Router service
Customer requests Elastic Router service and sends SG
Request virtualized Infrastructure view
NF-IB based decomposition
Request Elastic Router (template) decomposition
Service layer orchestration
Send NF-FG
Request (detailed) Elastic Router decomposition
NF-IB based decomposition
infrastructure orchestration
Send mapped NFFG to infrastructure
Deploy Elastic Router VNFs
Control NF
CP
DP1
DP2
...
DP1 cannot sustain user traffic
Request NF-FG infrastructure context (Cf-Or)
CtrlApp based decomposition
Request new DP instances (Cf-Or)
Elastic Control Loop
infrastructure orchestration
Send scaled mapped NFFG to infrastructure
instantiate new DP1,2,3,4 reconfigure forwarding
notify scaling operation done (Cf-Or)
Update DP configurations

Figure 23: Process flow of the elastic router deployment in the UNIFY framework.

data to the Control NF. This is further detailed in [M4.3] (and applied to the elastic router use case in M43 section 4.2.2). Additionally discussed in M43 is a 'Verification Workflow', which would introduce additional steps in this process flow. At different stages in the process flow, a verification step can be included (by analysis of the SG, NF-FG or functional verification during run-time by using tags or specific test packets).

As seen in the process flow graph (Figure 23), the UNIFY control loop is kept short, enabling a fast and quasi real-time adaptation to changing conditions. This in contrast to ETSI-alike architectures where the scaling logic would reside in the VNFM and NFVO entities, requiring a longer control loop, up to the Service Layer (as elaborately described in

[Sza+15]). This is also shown in Figure 24, where the Cf-Or interface and the Control NF allow a fast control loop. The customer has a configuration interface available through the EM (Element Manager) of the elastic router, but scaling actions are completely transparent to the user, they do not propagate up to the Service Layer.

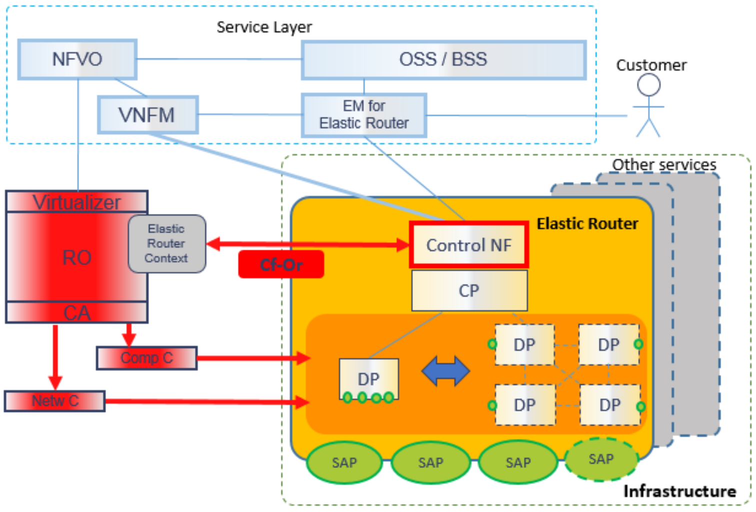


Figure 24: Elastic router fitting inside the UNIFY architecture. The Cf-Or interface allows fast scaling actions, avoiding scaling messages to go up to the Service Layer.

2.5.1.4 Elastic Router Service Graph

Let's consider the SG as depicted in Figure 25. It involves two main NFs: an elastic router having four interfaces, and a FireWall NF. The involved elasticity in this case refers to ability of the router function to handle higher bandwidth on it's incoming ports. A typical implementation of a software router involves for example a control component and a data plane component as illustrated in Figure 26. The control component might be implemented by an OpenFlow controller block such as the FloodLight or Ryu control frameworks in combination with a particular control plugin/application, and the data plane component might be implemented by a OpenFlow software switch such as Open VSwitch (OVS). Both might be contained in a monolithic virtual machine (VM), or might be split into multiple VMs.

A known mechanism in order to enable scaling or elasticity of software switch implementations is Valiant Load Balancing (VLB) [Dob+09]. This is an approach where a monolithic software switch is actually built from multiple switches, each responsible for handling the incoming traffic of one single interface of the original router. Such mechanism typically works in case network traffic passing the interfaces of the router component is relatively homogeneous. This concept is illustrated in the SG depicted in Figure 27. More advanced mechanisms involve multi-stage switching archi-

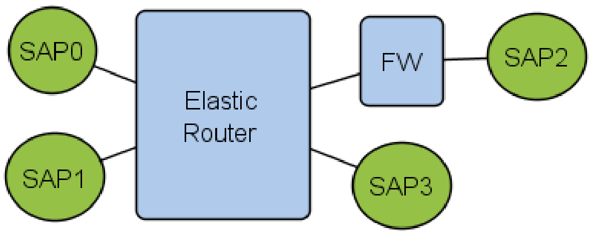


Figure 25: Service Graph involving elastic router

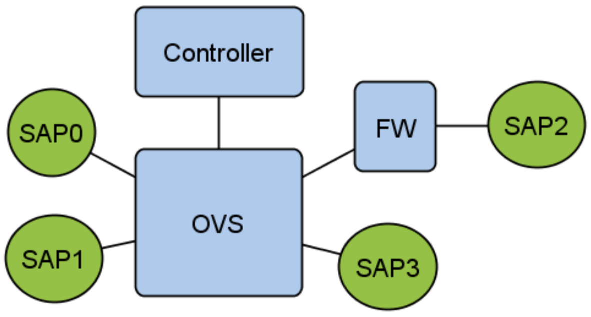


Figure 26: Service Graph involving decomposed elastic router

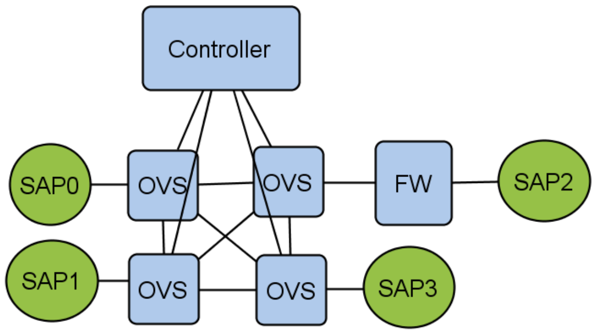


Figure 27: Service Graph involving further decomposed elastic router

tectures, usually referred as Clos switching architectures. These mechanisms are left for future work. Although these mechanisms are known in literature and research, currently there is no practical common platform available in order to actually use such schemes without over-specializing on the particular protocol or underlying switch implementations, or virtualization platforms. This is where the UNIFY control and orchestration platform might fill the gap, by enabling:

- Reservation of both network and computing resources on demand.
- Provisioning of a flexible programmability orchestration framework in order to let services decide on how to apply the scaling logic.
- Abstraction of the underlying infrastructure specifics.

2.5.1.5 Infrastructure virtualizer

UNIFY Service Orchestration relies on two concepts (as indicated in Deliverable D3.2 [D3.2] and its amendment documents):

1. An infrastructure model exposed by the Orchestration Layer towards the Service Layer.
2. The translation of a service (or Service Graph) into a directed graph of NFs.

Both concepts come together into the NF-FG as a mapping of the individual NFs and their interconnection logic onto the exposed infrastructure model (virtualizer).

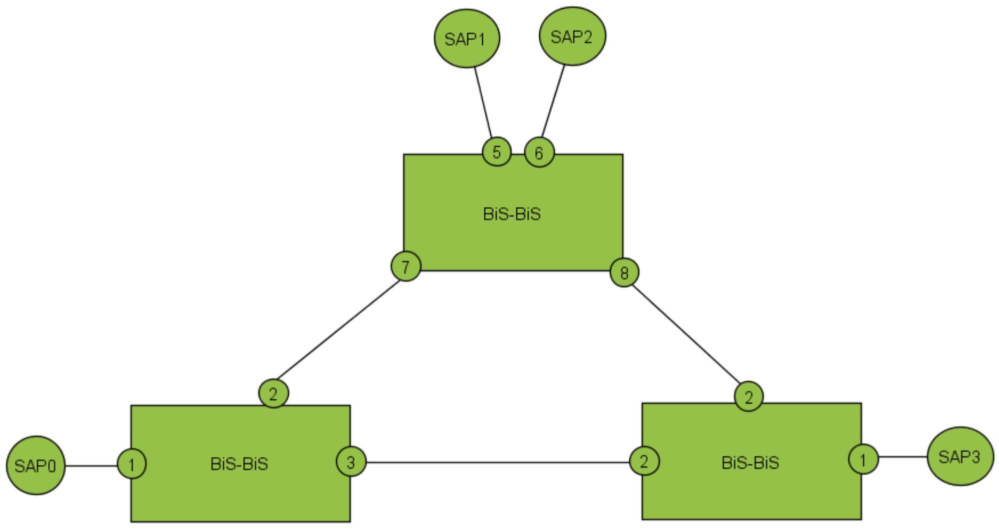


Figure 28: Infrastructure virtualization for the service

For the considered case consisting of one monolithic VM implementing both control and data, let's consider the infrastructure as depicted in Figure 28, consisting of three BiS-BiS domains of which a given Service Access Point (i.e., SAP0) is connected to the lower left one (e.g., an OpenStack-based domain), SAP1 and SAP2 are connected to the upper BiS-BiS domain, and SAP3 is connected to the lower-right BiS-BiS.

The components of the considered service (in the case of a monolithic VM implementing both control- and data plane functions, as well as implementing the Cf-Or interface depicted by the red interface) might for example be placed/mapped all on the upper BiS-BiS domain as shown in Figure 29. This mapping requires the considered BiS-BiS to have forwarding rules configured in order to force the interconnection of the ports on the router to the corresponding SAPs and firewall NF.

2.5.1.6 Scoping of the NF-FG through the Cf-Or

The Cf-Or interface now enables particular NFs (Control NFs) to directly interface with the UNIFY Orchestration Layer using exactly the same data model as the Sl-Or. Similar to the virtualized infrastructure model provided via the Sl-Or interface, the Cf-Or interface also exposes limited view of the available infrastructure. However, the scope of the considered interface might be limited to the view of a considered abstract NF (e.g., the scope of the elastic router NF) in order to limit the risk of breaking the semantics of the original Service Graph.

In the considered example, the Cf-Or interface of the Elastic Router NF (red port in Figure 29) is only capable of changing the implementation (and thus potential decomposition) of the Elastic Router NF itself, not the interconnection of the Elastic Router NF towards the SAPs or the FW NF. In other words, every configuration behind the interfaces 1 to 4 is out of the reach of the Elastic Router Control Function.

Ideally the interfaces 1 to 4 could be reused in any desired way, however this is not possible because their are part of the existing Elastic Router NF, which after decomposition might not exist any longer. The solution within the NF-FG model is to re-define these ports when referred over the Cf-Or interface as external interfaces of the considered BiS-BiS (similar to the case of using interfaces for SAPs). The existing Elastic Router NF could then be connected to these interfaces via forwarding rules as illustrated in Figure 30. The advantages of this approach are as follows:

- Control logic of the Control NF behind the Cf-Or has full freedom in tearing down existing NFs (even itself).

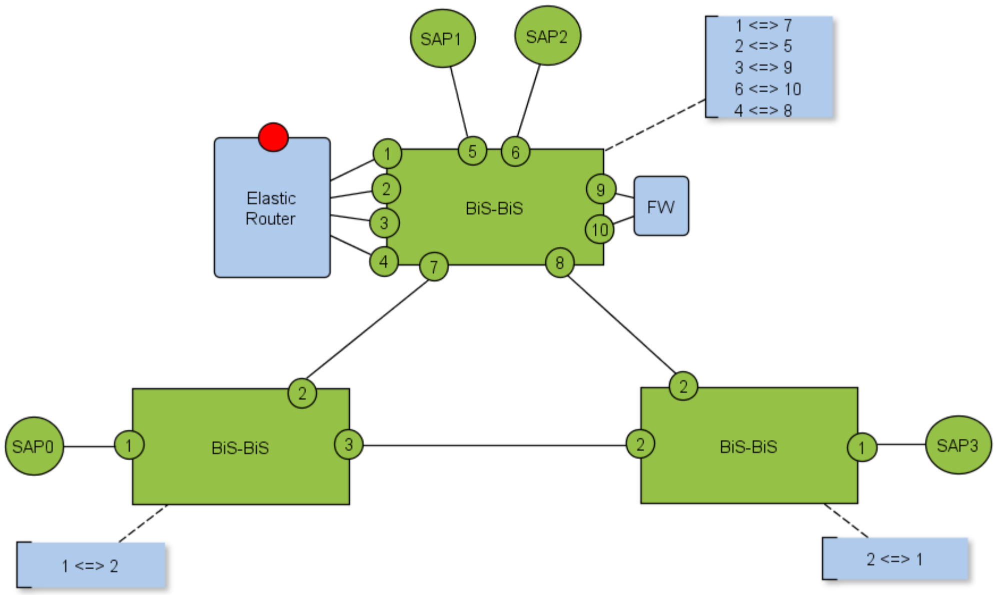


Figure 29: Initial NF-FG mapping at the Service Layer

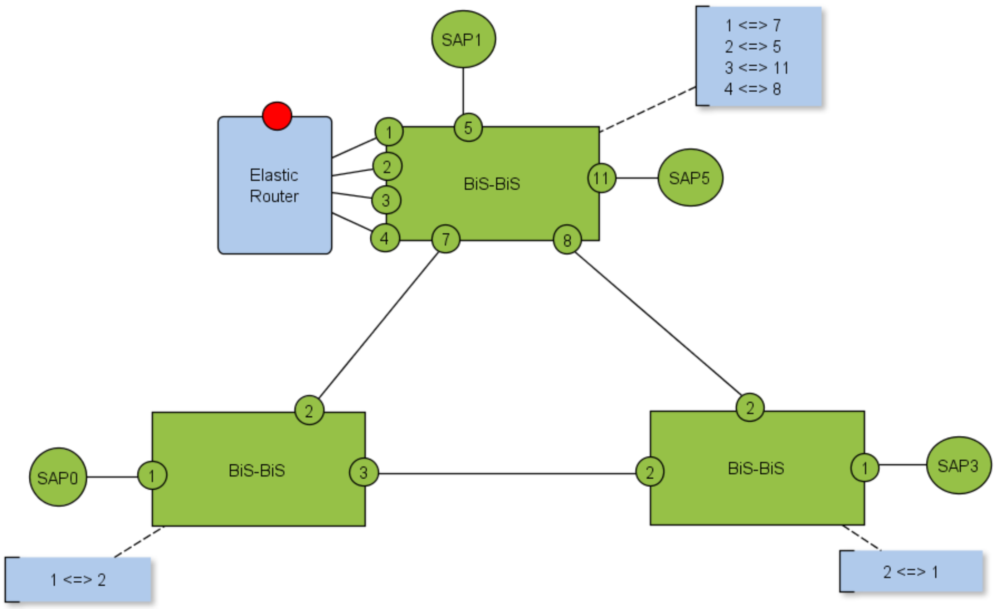


Figure 30: Scoping of the NF-FG for the Cf-Or. SAP5 is the connection point of the FW in previous NF-FG. This FW is not given in this scoped NF-FG, but its access point must remain as input port for the elastic router.

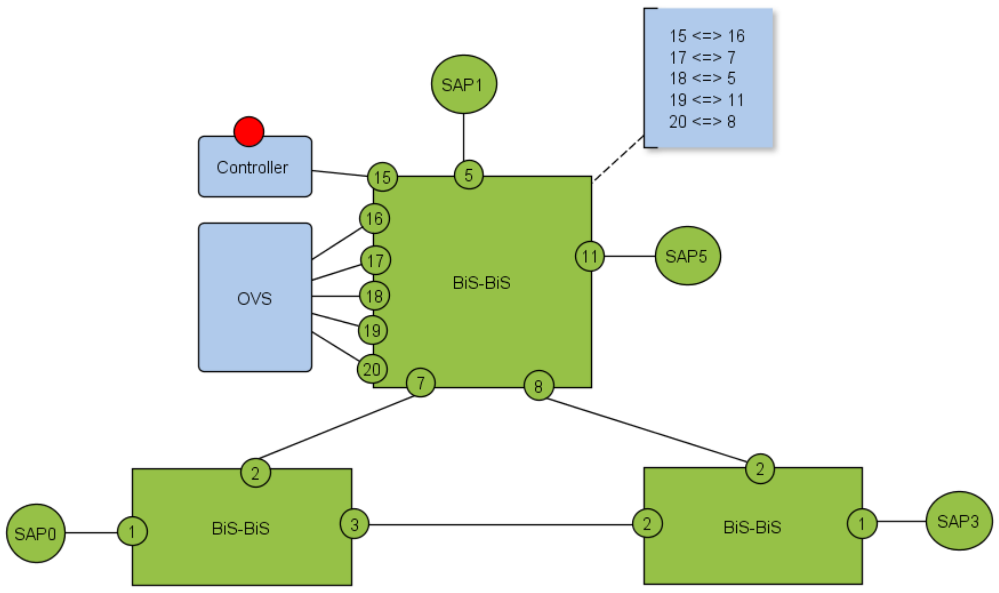


Figure 31: Cf-Or-driven decomposition and orchestration into control- and data plane NFs

- The semantics of the existing interconnection points (interface 1 to 4) remains and can be re-used in the new Cf-Or context within the existing data model.
- Different infrastructure domains can be made selectable in the NF-FG, instead of combining them into one big BiS-BiS. This allows extra fine-grained placement options for the Control NF, taking this task over from the Orchestrator.

Using the described Cf-Or interface, the Control NF might for example further decompose the monolithic VM into a Control NF and a dataplane NF as illustrated in Figure 31. This is a rather particular case, because it involves tearing down the NF to which the Cf-Or itself is connected, and deploying a new Control NF connecting towards the Cf-Or. Further research and agreement must clarify if this is an acceptable scenario. It might make more sense to have an initial decomposition into a control and data plane NF at the Service Layer, such that the NF implementing the Cf-Or does not need to be torn down.

Control functionality at the Controller NF of Figure 32 might now instruct the Resource Orchestrator to deploy a VLB configuration where 2 Open vSwitch instances are deployed on the upper BiS-BiS domain, while the lower BiS-BiS domains each deploy another Open vSwitch instance. As illustrated in the Service Graph (Figure 27), the Open vSwitch instances are interconnected by a meshed topology by the BiS-BiS in Figure 32.

Elasticity support in NF-FG and NF-IB In order to support scaling/elasticity functionality in UNIFY in a well-integrated way, the following aspects are required:

- an attribute in the NF and NF-FG data model for characterizing scaling/elasticity requirements/logic within the NF-FG
- support within the decomposition model for NFFG to decompose an NF in NF-FG with particular scaling/elasticity attributes into appropriate NF deployments

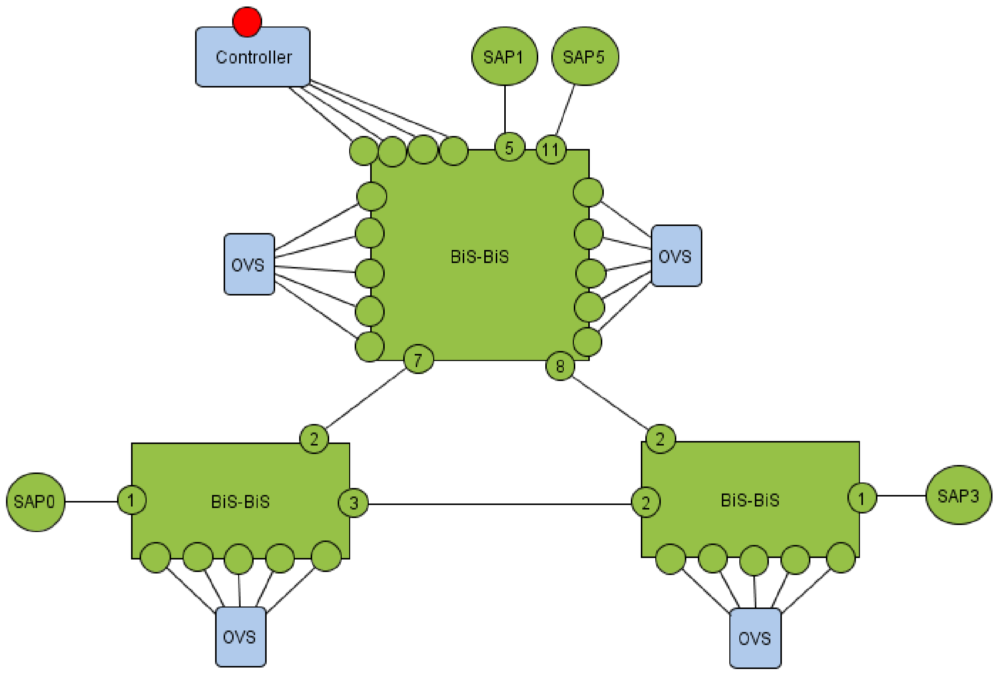


Figure 32: Cf-Or-driven decomposition and orchestration into VLB setup. The OVS instances are interconnected by a meshed topology as illustrated in the Service Graph of Figure 27. The forwarding rules in the BiS-BiS are set up to achieve this (the forwarding table is not shown for readability)

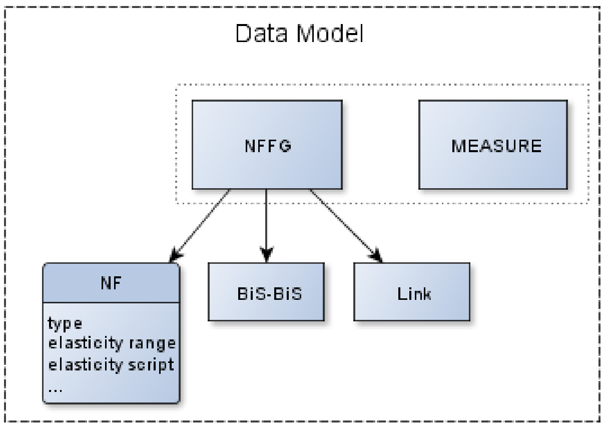


Figure 33: Data model: elasticity support in NFFG

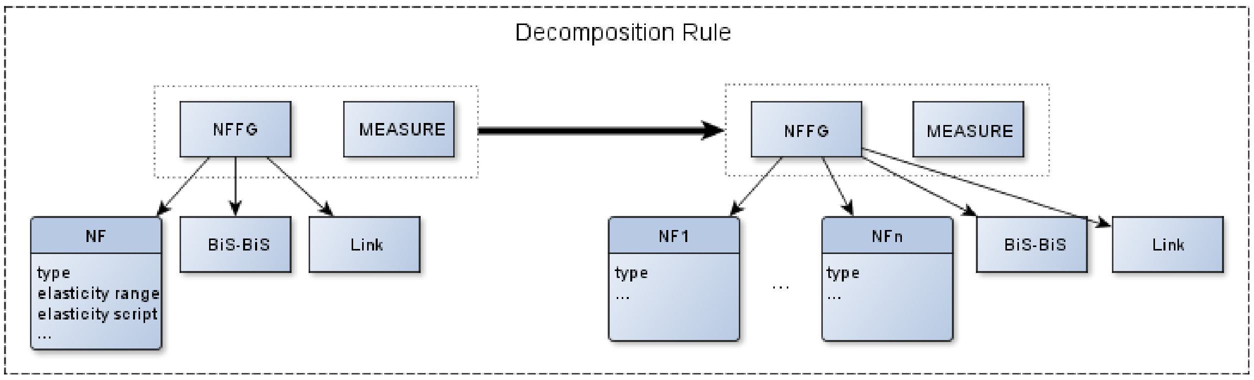


Figure 34: Decomposition rule: elasticity support in NF-IB

Figure 33 illustrates the proposed data model which supports the NF elasticity in NFFG. In this model, the NFs within the NF-FG can have different attributes including NF types, elasticity requirements and elasticity behavior. Regarding elasticity attributes, we account for two possibilities:

- a declarative option: e.g., an elasticity range can be determined indicating the minimum and maximum required throughput of the NF
- a procedural option: e.g., indicating upon what condition additional NFs need to be deployed (e.g., if packet loss $>$ 1 percent then deploy more DP NFs), through the use of, e.g., Domain-Specific Language (DSL)

In addition to the elasticity support within NF-FG, decomposition rules should be provided to translate the NF-FG s and their corresponding NFs' attributes to NF-FG s with more refined NFs and their attributes. These rules are stored in the NF-IB which are then used by the RO for service deployment. The first option (declarative option) is addressed by adding a decomposition rule which translates the elastic NF and its requirements (e.g. an Elastic Router (ER) with minimum 1 Gbps to 100 Gbps throughput) to atomic NFs such as Control Plane (CP) and Data plane (DP) NFs. In this case, all the scaling logic is part of the CP/Control NF which was provided by the NF developer. In the second option (procedural option), the Control NF is programmable, and the decomposition process actually involves a compilation step of translating the elasticity instructions from the NF attribute into adequate Control NF configurations. Figure 34 illustrates the decomposition rules which address the two possibilities in support of elasticity functionality. In this figure, the two elasticity attributes refer to the two explained options.

Elastic Router Demonstration A first implementation of the elastic router, with Cf-Or triggered decomposition, is presented in [Ros+15]. It can be seen in Figure 35 that the number of switches and dataplane topology is adapted dynamically, in function of the packet rate. Figure 36 further illustrates a possible use case of the elastic router, where a user adds more ports to the service. Because of this, the ingress packet rate rises and the router scales, invisible to the user.

2.5.1.7 VNF Profiling

Managing for predictable performance in virtualized environments is far from straightforward, taking into account the variable processing overheads of virtualization and the underlying available infrastructure resources. A VNF might have equivalent implementations but optimized for different execution environments. When a Service Provider wants to use this VNF, it must know how much resources to allocate to meet the QoS requirements in the SLA. Next to this, the

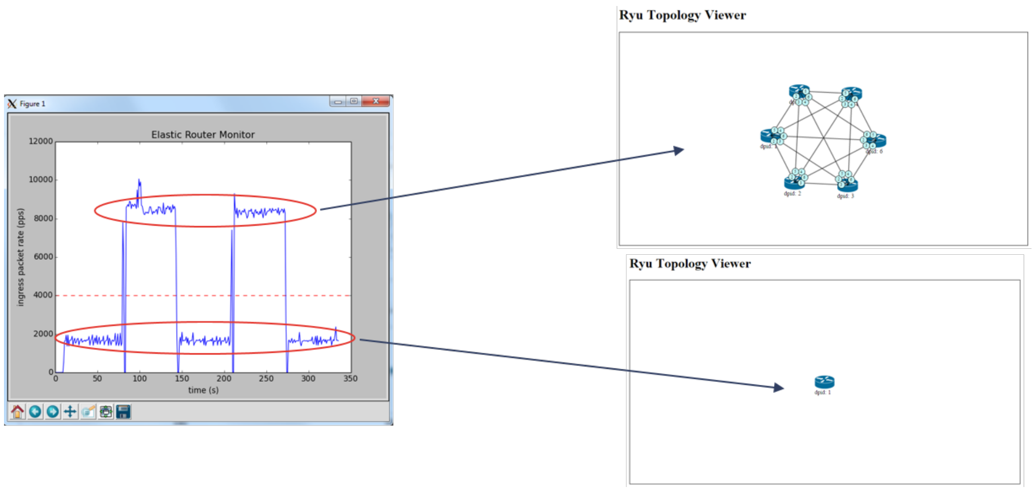


Figure 35: Left side: Info provided by the Control NF: plot showing the ingress packet rate and the threshold. On the right side is the visual representation of the switch topology deployed, in function of the ingress packet rate shown on the left.

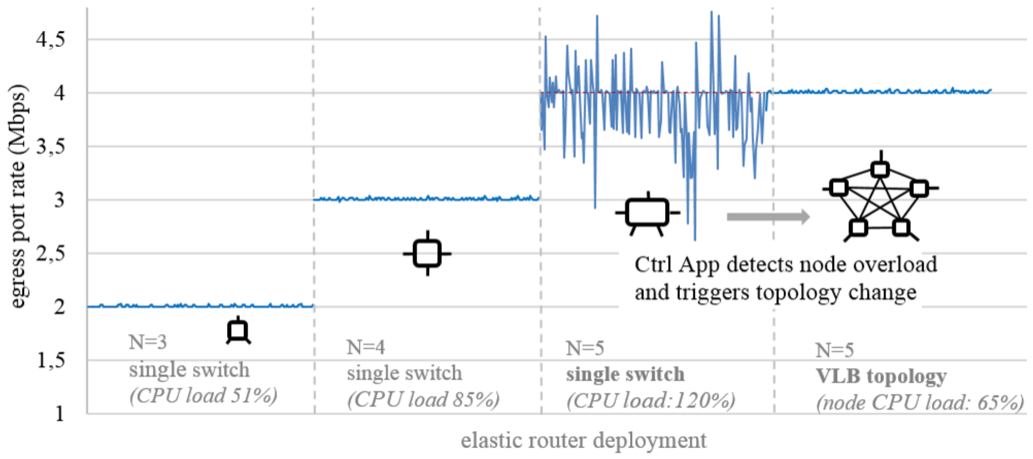


Figure 36: Elastic router demo showing the ability to set the number of ports N via the Service Layer and scale resources adaptively to the required throughout R via the Cf-Or interface. (The used topology and CPU load is displayed as info).

threshold which triggers a scaling action is a figure that requires a deeper analysis of the NF or service to be scaled. In the Cf-Or scaling process discussed up to know, the Control NF knows when to trigger the scaling action by monitoring some specific Key Performance Indicator (KPI)s (eg. packet loss, throughput or cpu load). To determine the threshold, and do the scaling action in an efficient and pro-active way, VNF profiling is a possible method.

We can define a VNF profile as: a mapping between virtualized resources (e.g., vCPU, memory) and VNF performance (e.g., bandwidth, delay between in/out or ports) at a given infrastructure. An orchestration function can use the VNF Profile to select a host for a VNF and to allocate the desired resources to deliver a given (predictable) service quality.

An important aspect of a deployed service is the SLA. This agreement or contract typically specifies the high-level service requirements which affect the performance, expressed as workload parameters (eg. maximum number of users, requests per second...). This is combined with Quality-of-Service (QoS) parameters or the associated performance metrics, also known as KPIs (eg. response time, available bandwidth ...). VNF profiling contributes to service guarantees for the deployed NF-FG. If the VNF profile is very reliable and its performance becomes very predictable,

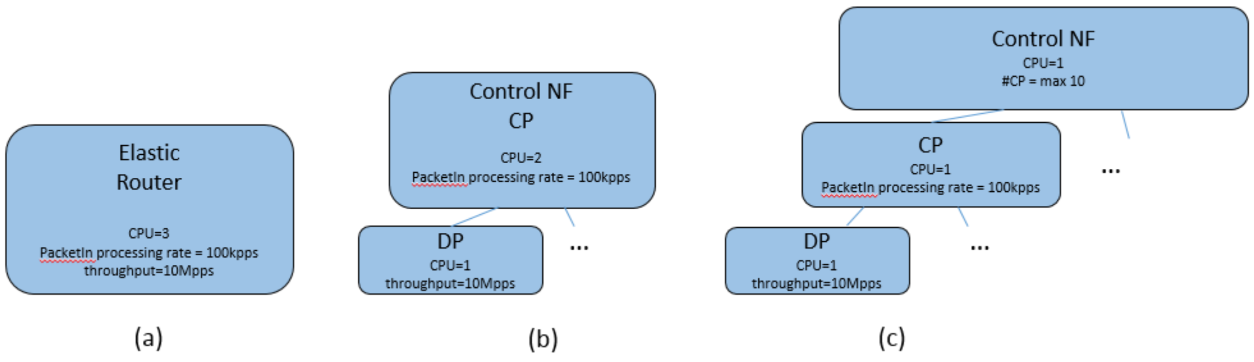


Figure 37: Elastic Router decompositions with performance profile attributes

then this might reduce the amount of parameters to continuously monitor during the service lifetime. The QoS performance is guaranteed as long as the profiled hardware resources are available.

A necessary functionality to derive any performance data is the ability to monitor several KPIs (eg. CPU load, link/processing delay, throughput,...). The KPI parameters are very service-specific (how, when, where to measure?), so ideally they are defined as part of the service decomposition. In UNIFY, they can be added inside the NF-FG. This offers a generic way of adding monitoring agents. To this purpose, the MEASURE monitoring language is introduced in the UNIFY NF-FG. This is further subject of WP4 ([D4.2] and further detailed in [M4.3]).

In Figure 37, an example is shown, how several decompositions stored in the NF-IB or Control NF can contain profiling data. In the example only one single performance metric is specified, depicting the maximum workload the elastic router decomposition can handle with that specified CPU allocation. More elaborate benchmarking or profiling can result in wider set of performance metrics, allowing more options to scale (eg. indicating the effects of vertical scaling, by adding more CPU).

When applied to the Elastic Router profiling, we need to monitor following metrics:

- Workload = ingress packet rate (pps), generated over a number of ports. Ideally this should be co-specified by the arrival rate of new flows, since this is a known bottleneck of Openflow switches and controllers [Wan+14].
- KPIs = the delta between ingress/egress packet rate (pps) can be used to determine packet loss and throughput in the data plane. The PacketIn processing rate provides a way of monitoring the control plane performance.
- HW requirements = CPU load

There are some benchmark tools available, specifically designed for the profiling of OpenFlow switches and controllers:

- OFlops [Rot+12] (OpenFLow Operations Per Second) is a standalone controller that benchmarks various aspects of an OpenFlow switch. Oflops implements a modular framework for adding and running implementation-agnostic tests to quantify an switch's performance.
- Cbench [Too+12] (controller benchmarker) is a program for testing OpenFlow controllers by generating packet-in events for new flows. Cbench emulates a bunch of switches which connect to a controller, send packet-in messages, and watch for flow-mods to get pushed down.

Specific tools for profiling and monitoring data analysis could be integrated into the service platform to enable the performance benchmarking of VNFs. However, other methods are also applicable. In [BS15] and [SBP15], the authors introduce a methodology for the modelization of network functions focused on the identification of recurring execution patterns and aimed at providing a platform independent representation. By mapping the model on specific hardware, the performance of the network function can be estimated in terms of maximum throughput that the network function can achieve on the specific execution platform. By code analysis, the elementary operations (such as packet copy between I/O and memory, hash table lookup ...) function are identified and used to model a network function (an MPLS switch in the referenced papers). Each elementary operation can be mapped to a sequence of elementary, atomic steps like CPU instructions, of which the CPU or memory load can be easily calculated in function of the used hardware platform. This method needs further refinement by modelling also the effects of concurrency and packet interaction. Also for complex or closed software, this approach is infeasible. This analytical modelling is a white-box approach which needs a significant amount of information regarding the internal structure of the applications and a sufficient amount of time spent by an expert to design and fine-tune it. In case of a single critical and non-complex application for example, this type of modelling can efficient. However it cannot be realistically applied in a service provider scenario that has to model numerous services.

Another approach to determine VNF profile is VNF Benchmarking. As proposed in [RRS15], the benchmark process could be offered as a service to the service developer or provider. Ideally it could be provided by the infrastructure provider to a higher-level entity like the Orchestrator or the Service Layer. By testing a VNF on representative hardware, with one or more representative workloads and before it is deployed in production, the performance profile can be extracted and a resource estimation is known. The challenges in this approach lie in the correct definition of the benchmarking process: the QoS metrics to monitor, a representative input load during the test and a representative test environment (simulation, emulation or physical infrastructure).

2.5.1.8 NF-IB based decomposition

As explained in the introductory text of this chapter, the Cf-Or related decomposition as described until now in this section, is based on internal logic from the CtrlApp or Control NF. With a short control loop, invisible from the outside, upper Service Layer and out of control from the service user (black-box). Another option to implement elasticity, at service runtime, is via NF-IB decomposition. This approach would have the advantage that all possible decomposition topologies are stored in one single place, the NF-IB. However, as can be seen in the process flow of Figure 23, a downside is that the elastic control loop will take longer if possible decompositions need to be fetched from the NF-IB first. This in contrast to the methodology followed so far, where decomposition updates at runtime are implemented into the Control NF. There are some advantages and disadvantages to a more NF-IB based scaling approach, as discussed further.

The inner structure of the NF-IB was described earlier in section 2.3. The NF-IB is capable of holding the relationships between a abstract, high-level template-like NF and more refined interconnected NFs with the same external interfaces as the higher level NF. This is stored in a tree-like data structure in support of the decomposition process. It can be beneficial to store many possible decomposition templates in the NF-IB. The research done in [KRK14] and [Sah+15] shows that the mapping or embedding algorithm in the Orchestrator can benefit substantially if the possible decompositions are taken into account up front. It helps the NP-hard embedding problem to find a solution faster, as it offers more options to map the service unto the available infrastructure. These options are not available to the Orchestrator, if only stored and accessible by the Control NF logic.

During Service Layer orchestration, at initial deployment, the service is not yet scaled, because no workload information is known. Moreover, the service does not know how to scale, since this logic is only implemented in the Control NF. As mentioned in [D3.1], a solution might be that the scaling functionality inside the Control NF is callable from the Orchestrator or somehow exported into it. New or updated decompositions can be added to the NF-IB along with their performance profile, instead of updating the complete Control NF. The elastic control loop would then look like this: when the Control NF detects a scaling action is necessary, it does not send an updated NF-FG, but instead sends updated performance or workload requirements. These are processed by the embedding algorithm in the Orchestrator, taking into account all possible decompositions found in the NF-IB and the available infrastructure. This puts more complexity in the orchestration algorithm, as it must be able to translate updated workload requirements to the appropriate HW specifications (using the VNF profiles).

2.5.2 Notes on service dynamics

We have illustrated in above sections how the Cf-Or interface and the NF-IB database in the UNIFY architecture enable certain dynamics in the deployed service. Traditional telecom services are highly dependant on physical topologies and vendor-specific hardware. Next to this, rigid network control limits the flexibility of service creation. To overcome this, SDN and NFV principles are getting commonly adopted to design new platforms for creating, managing and scaling network services on-demand, in a faster and more resource-optimized way. Continuing progress in the field of NFV allows flexible deployment and scalability of NFs as virtual machines running on common servers. Complementary efforts in SDN are bringing new possibilities to dynamically manage network traffic between these NFs. In this context, elasticity denotes the ability of assigning hardware resources dynamically, as response to fluctuations in eg. service demand or resource availability. This is certainly a requirement for services running on commercial cloud providers using pay-per-use billing models. By monitoring specific KPIs, the service can deploy more or less NFs, only consuming the resources needed for its real-time average performance. In worst-case conditions, or as the service configuration changes, the number/type of resources can be altered. In previous sections, we discussed the two important components that mainly enable the elastic scalability in UNIFY, which are the Cf-Or interface and the NF-IB database. To conclude this chapter, we briefly highlight some additional service dynamics which were not discussed before but are also possible in the UNIFY architecture.

2.5.2.1 Control plane elasticity

Elasticity is not only applicable to horizontal/vertical scaling of NFs in the data plane. As proposed in [SSS13], also the control plane itself should be elastic. The functions running in the control plane typically involve monitoring analysis, messaging, allocation of compute and storage resources and VM image management. As the cluster size of the service grows, it is clear that also the control plane functions need to scale along with the number of client/data plane nodes they have to support. To scale the control plane in such scenarios, its functions will need to be replicated on multiple nodes and the incoming workload to the service will need to be distributed across the replicas of the control plane functions. Typically replication can be done in one of two ways: (i) In the clustering approach, control plane replicas collectively serve all the requests from each data plane as a single logical entity – as shown in Figure 38a. (ii) In federation, each control plane instance services a subset of data planes and forwards only the necessary requests to another control plane – as shown in Figure 38b. Both clustering and federation approaches partition the workload but the clustering based approach also allows high availability while federation does not. It is also clear from the figure that the UNIFY architecture supports both clustered and federated topologies and that any approach can be used to implement workload balancing between the control and data plane.

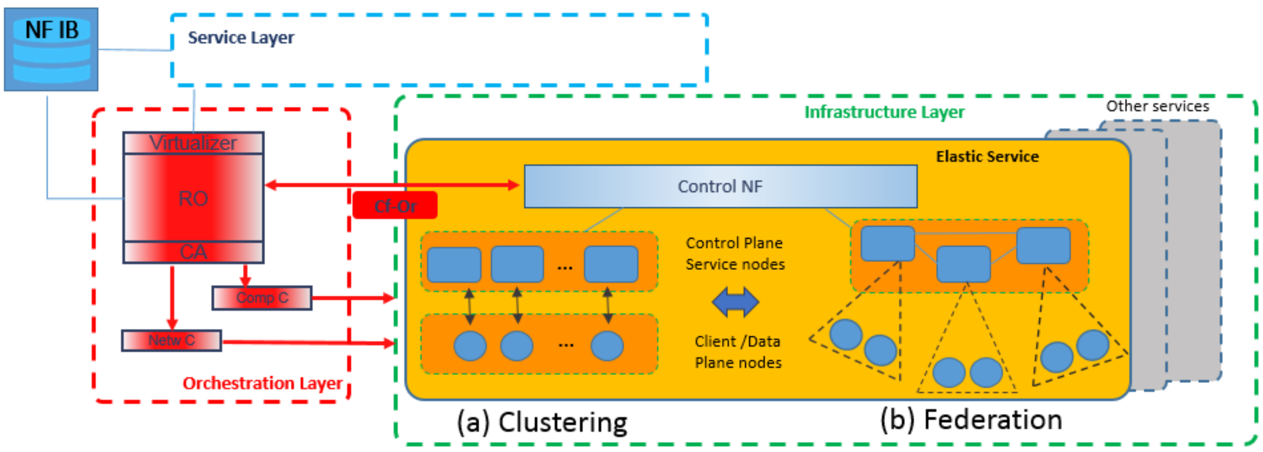


Figure 38: Clustering and Federated approaches for elastic control planes

2.5.2.2 Cloud service scaling vs. NFV service scaling

It needs to be noted that services deployed in current available cloud environments have limited options to implement automated scaling actions or elasticity. Commercial cloud providers like Amazon or Microsoft are heavily focused on web services handling requests and following an N-tier architecture (eg. webserver + databases). These services follow a relatively straightforward decomposition strategy where auto-scaling approach consists of cloning servers and putting a load balancer in front of them (similar to Figure 39(a)). The time required for the load balancing is neglible compared to the request handling time. Openstack has no real support for defining elastic actions during service runtime but can be modified accordingly as described in [Bee+12] or [SSS13]. Latest release also offers elastic load balancing options. These features still only apply to automating scaling actions defined by simple server cloning and load-balancing, following the commercial cloud features. Horizontal scaling is limited to the case where stateless NFs are cloned and connected to a load balancer or moved to another datacenter (eg. closer to the users). Also vertical scaling is possible, where the number of NF instances is not altered, but a single NF is granted more/less CPU or memory.

This is in contrast to the NFV services we intend to deploy in the UNIFY framework. There, part of the service might be about rapid packet handling or generation. Stateful VNFs need to be supported, so cloning VNFs and simple load balancing might not be sufficient, because a simple load-balancing action as in typical cloud platforms might have a cost of the same order as the original packet-handling itself, thus not leading to any improvement in performance. The scaling of NFV services requires in-depth knowledge of the service, and needs to be controllable/programmable by the service developer. To illustrate that scaling strategies might differ between services, possible decomposition topologies are shown in Figure 39. UNIFY tackles these limitations by allowing a Control NF as part of the deployed service, which holds the scaling logic as implemented by the service developer. This still leaves a lot of flexibility to developer to program different topologies to scale to, as long as the Control NF can communicate through the Cf-Or interface.

2.6 State migration

To support VNF resiliency and VNF scaling mechanisms the internal state of the VNFs has to be managed properly. Depending on the particular VNF functionality the state will typically consist of configuration state created by control components, such as CtrlApps and/or users of the VNF, as well as transient state that is created by the traffic itself,

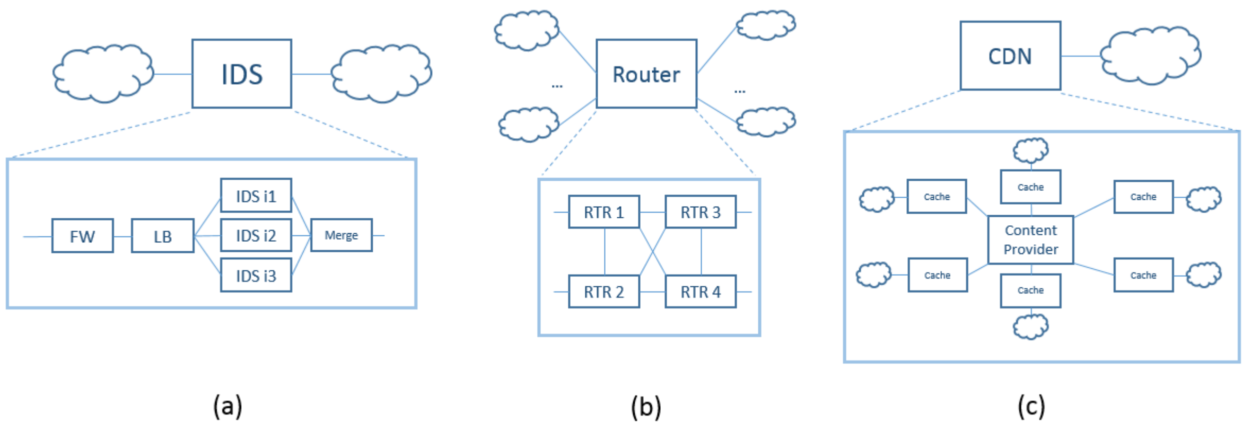


Figure 39: Scaling or decomposition is service-dependant. Topologies differ between (a) Intrusion Detection System (b) Elastic Router and (c) Content Delivery Network

such as packet counters. For a VNF to function properly both these types of state has to be managed and synchronized. To provide a VNF resiliency mechanism the VNF state management mechanism could e.g. periodically synchronize the state between a primary and a backup VNF instance, so when the primary instance fails, the backup instance is up to date and able to handle traffic flows redirected from the primary instance. In the case of scaling, appropriate transient and configuration state has to be migrated to a new or old instance of the VNF for scale-out and -in respectively.

D3.1, section 6.7.2.2 [D3.1], introduced various approaches for scaling VNFs for example by scaling-out by adding new VNF instances or scaling up by allocating additional resources to a VNF. The problem of managing the state of the VNFs during these events was introduced there as well. The main issues in state management is to 1) locate and transfer the approriate state, and 2) synchronizing the transfer of state with the redirection of traffic to avoid invalidation of the transferred state and packet losses. In [KDS15] and D3.2 section 5, an extended mechanism for handling state migration for typical VNFs, OpenNF with Distributed State Transfer (DiST) extensions. The OpenNF mechanism differ from similar mechanisms such as traditional Virtual Machine migration [KVM15] or Checkpoint/Restore In Userspace (CRIU) [CRI15] in several ways, primarily in that it discriminates which state(s) should be transferred by associating state blocks with the traffic flow(s) responsible for creating or updating a particular state block. This association allows the state for a certain flow definition (address, subnet, VLAN, etc.) to be transferred without interfering with other flows passing through the VNF. The association between flow and state makes it possible to modify the forwarding tables for a particular flow while at the same time transferring the state, in a synchronized manner. It also reduces the amount of state that needs to be transferred as the whole VM memory space (VM migration) or process memory space (as in CRIU) is not transferred, but just the necessary state blocks.

For most VNFs the association of the incoming traffic flows with their state makes sense from a scaling/migration point of view as incoming traffic flows that would be migrated to other places. For many other cloud applications this association would not be useful as their input is not traffic flows but something else. For example a database application distributed over multiple VMs has as its input a set of reads and writes to/from certain tables and columns, perhaps over a single TCP connection. Here it does not make sense to try to divide the database state based on the source of the state (i.e. the TCP connection) but rather based on some other characteristics of the state such as the column or a group of rows. For example a database server may start on a single node with a table containing two columns, 'User name' and 'Address'. When the database needs to scale-out one approach would be to divide the state based on the 'User name' field, moving all the customers with names starting from J to Z to the second server (called Sharding in the context of databases). This simple sharding example is depicted in Figure 40. The approach taken by OpenNF/DiST

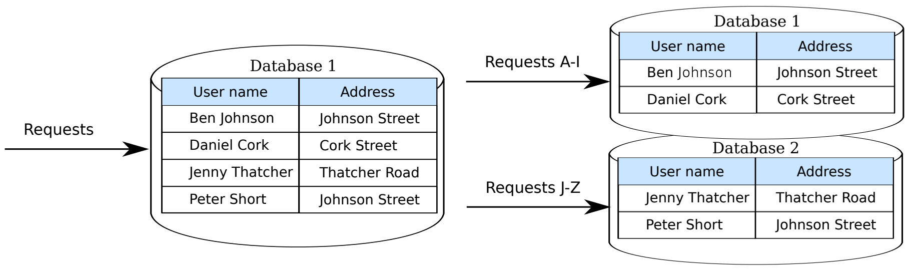


Figure 40: Database sharding on 'User name' column

could be seen as sharding using the traffic flows as key (rather than 'User name' as in the figure).

D3.2, section 5.3, discusses service resiliency in the context of NFs, the ability of a service to withstand failures due to software and hardware failures while remaining operational. As described there, the mechanisms for transferring and/or synchronizing state may be the same for resiliency and scaling, however, the mechanism is triggered by other events, the scope of state transfer is different, and the process is likely controlled by a different component.

As different methods of scaling as suitable for different applications here we investigate a type of VNF that may require a different a state handling approach than the one investigated in D3.2, "Split State VNFs". The OpenNF/DiST approach is based on monolithic VNFs where a single entity (e.g. on running on a single VM) receives, processes, transmits traffic, and stores state internally. In our definition a "Split State VNF" is slightly different, with a data/control plane division similar to the division between an OpenFlow controller and OpenFlow switches. In Split State VNFs state may be distributed among multiple sub-components and traffic processing may happen in multiple locations, controlled by a central node. Here we investigate state transfer, both for scalability and for resiliency, for two Split State VNFs; FlowNAC, a service authentication function, and the Elastic Router, a scalable router function.

2.6.1 FlowNAC state migration

The FlowNAC VNF has been previously introduced in D3.2 section 3.3.2 [D3.2]. The aim of this VNF is to implement a network access control which is able to independently authorize a service identified by a specific flow (or a set of flows). It can be decomposed in two basic components as shown in figure 41: FlowNAC Enforcing (FNE) and FlowNAC Control App (FNC). The former is a stateless component responsible for enforcing the policies at data plane level, whereas the latter is a stateful component which stores the state associated to the authentication (AuthN) and authorization (AuthZ) processes.

2.6.1.1 Load Balancing scenario

In this context, we analyze a scenario in which the FlowNAC load (and its associated state) is balanced between different Control Apps. At this point, the FNE component is excluded and remains invariant located in the first access node, which means that each and every access node has a FNE deployed on it. The FNC component is deployed on the core network (e.g. on the Cloud). Each FNE is associated with one FNC component and several FNEs can be associated to the same FNC (i.e. N:1 relation). Since the FNC component stores the state of the AAA process, if one FNE is associated to a different FNC, all the state associated to the FNE must be migrated to the new FNC. This is when the migration of the state comes into the scene.

The proposed state migration in the context of a load balancing scenario is the following. There are four access nodes with a FNE (FNE1 to FNE4) already deployed on them and several end users behind requesting for access to several services. Initially all FNEs are associated to the same FNC (FNC1) already deployed on the core, as seen in figure

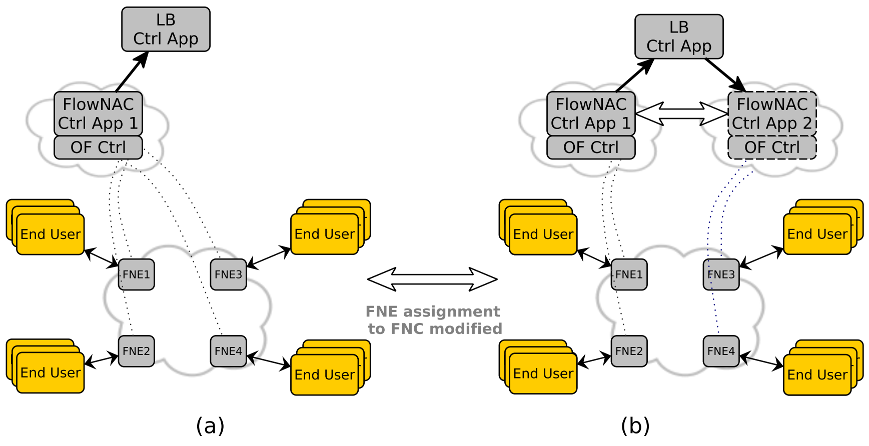


Figure 41: FlowNAC VNF in the load balancing scenario

41a. At a certain point, there is a trigger (e.g. depending on the number of AAA sessions from end users managed by the FNC1) to instantiate a new FNC (FNC2) process on the core and migrate all the state associated to half of the FNEs (i.e. from the end users behind FNE3 and FNE4) to the new FNC2, as seen in figure 41b. How to manage the state migration from FNC1 to FNC2 and its impact on the FlowNAC solution is the topic to investigate. One important aspect to consider is the association between the FNE and the FNC, which means that this association must be coordinated with the state migration from FNC1 to FNC2.

The process behind the load balancing scenario is described below as a set of steps:

1. LB trigger. The Load Balancer (LB) functionality detects the LB trigger, which initiates the state migration process.

2. Select FNE to migrate. The LB trigger is analyzed and the appropriate (or set of) access node to migrate is selected. In this stage, the overall system is analyzed to determine the best manner to redistribute the FlowNAC load, e.g. how many new FNC to instantiate, which FCE to associate to the new instance and how to minimize the impact of those migrations.

3. Instantiate FNC2. The new FNC instance is requested and deployed. The Cf-Or interface is used to request the new instance and the orchestrator decides the best location for the new deployment.

4. Hold FNE3/4-FNC1 association/control traffic. The association between FNE3/4 and FNC1, and the related control traffic between them, must be stopped during the state transfer process from FNC1 to FNC2. Otherwise, the control traffic is lost once the state is transferred to the FNC2. [2]

5. Get FNE3/4-state from FNC1. The state associated to the users (and services) behind FNE3/4 must be extracted from FNC1.

6. Push FNE3/4-state to FNC2. The previously extracted state in step 5 is pushed to the new FNC2 instance.

7. Establish FNE3/4-FNC2 association. The association between FNE3/4 and FNC1 is released and FNE3/4 are associated to the FNC2.

[2]In OpenNF this traffic is temporarily buffered in the OpenFlow controller, this buffer is moved to the target VNF (in this case FNC2) when DiST extensions are applied. However, there may be cases where neither the OpenFlow controller nor the target VNF is an appropriate location for buffering, for example due to memory constraints. A solution in these cases could be to have another buffering device, for example be a temporarily instantiated VNF whose only function is to act as a buffer, or provide this function as part of a Universal Node that could be shared by all the running VNFs on the UN (similar to a memory swap file).

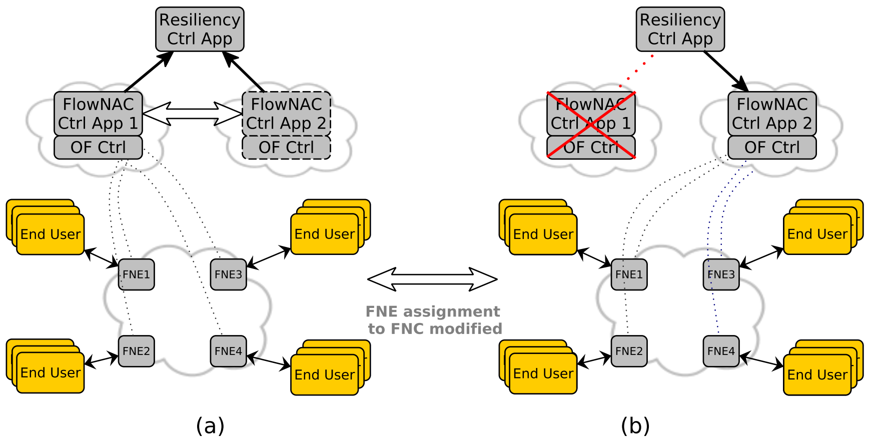


Figure 42: FlowNAC VNF in the resiliency scenario

8. Reactivation of FNE3/4. The FlowNAC functionality is reactivated after the state migration process with the FNC2 as the stateful component associated to the FNE3/4.

One of the studies to carry out is the actual impact of AAA process (such as new end user requests) held during the state transfer which involves steps 4 to 8.

2.6.1.2 Resiliency scenario

The second scenario focuses on the resiliency of the FlowNAC VNF. Since the FNC is the stateful component, the state migration in the resiliency scenario is analyzed with regard to the FNC, whereas the FNE component is excluded from this study. However, the association between each FNE and the corresponding FNC must be considered when migrating from one FNC to another.

The proposed scenario to analyze is the following. There are four access nodes with a FNE (FNE1 to FNE4) already deployed on them and two FNCs (FNC1 and FNC2), as seen in figure 42a. All FNEs are initially associated to one of the FNCs (FNC1), which is considered the active element. The FNC2 is the passive element in standby as backup component, which is not used during normal operation, and it only becomes active in case of failure in the FNC1. This scenario, 1+1 redundancy, could be easily extended to N+1 redundancy by adding extra active FNCs components which rely on FNC2 in case of failure. The proper behavior of FNC1 component is continuously monitored to detect possible failures of the system. This means that FNC2 would be activated triggered by the monitoring functionality provided by WP4. To provide a seamless migration from FNC1 to FNC2, the state in FNC1 needs to be synchronized periodically. In case of failure of FNC1, the monitoring system triggers the migration to FNC2, as seen in figure 42b. The mechanism for detecting that a failure has occurred could be based on tools developed within WP4, for example using the RateMon tool to trigger an alarm if no control traffic is transmitted by the FNC during a certain interval. Another approach could be to make the FNC act as a monitoring tool itself that connects to the monitoring system. Once connected the FNC could send a watchdog message e.g. every second if it still working; if no message is received over an interval the FNC is concluded to have failed and the fail-over to the backup FNC is triggered. The watchdog mechanism could be made more sophisticated by performing self-tests and only sending the watchdog message if the tests are passed. The self-tests could check that e.g. enough memory is still available, the internal data structures are consistent, etc. The mechanism could be further extended by adding a “negative” watchdog message which immediately triggers the fail-over, this message could be sent e.g. if any of the processes in the FNC crashes.

The process related to the resiliency scenario is described below as a set of steps:

1. Instantiate FNC2. To implement the 1+1 resiliency a new FNC component is deployed.
2. State sync - Get and Put state from FNC1 to FNC2. Periodically the state in FNC1 and FNC2 is synchronized.
3. Monitoring trigger. A failure in FNC1 is detected by the monitoring functionality, which initiates the migration to FNC2.
4. Buffer FNE1/4-FNC1 association/control traffic. Before the association to FNC2, all the control traffic from FNE1, FNE2, FNE3 and FNE4 to FNC1 is temporarily buffered.
5. Establish FNE1/4-FNC2 association. FNE1 to FNE4 are associated to the FNC2 instance.
6. Release the buffered control traffic from FNE1, FNE2, FNE3 and FNE4.
7. Reactivation of FNE1, FNE2, FNE3 and FNE4. The FlowNAC functionality is reactivated to recover from the failure in FNC1.

The synchronization between FNC1 and FNC2 is performed periodically. However, as the FNC1 failure could happen at any time state updates that occurred after the last update will be lost. One solution to this problem could be to synchronize after each state update, this approach may however be too costly or not feasible if the state update frequency is very high (this approach is taken by [RWJ13]).

The minimum recovery time expected with the approach outline above depends on primarily two variables, how fast the failure can be detected and how fast the temporary buffering and redirection of traffic to the backup FNC can be performed. The watchdog approach to failure detection could potentially be fairly quick, in the millisecond range if necessary. Since the monitoring framework developed in WP4 allows measurements to be processed locally a watchdog interval of one millisecond could be possible (although maybe too strict). The buffering and redirection of traffic to the backup FNC depends primarily on the amount of packets that has to be buffered and the time to establish the new path. The measurement results for OpenNF with DiST shown in Table 5-7 in D3.2 can provide a hint on the latency added by buffering, however, these measurements includes the state transfer but not the new path establishment so they are not directly comparable. The difference in the measurements of "DiST LF PZ" and "DiST LF PZLLER" of 0.1 seconds is the contribution of redirecting about 500 messages, if this scales linearly a fail-over time of 0.2s for each Kpps of control traffic may be a reasonable expectation.

2.6.2 Resiliency of the Elastic Router

As any network service, the elastic router is prone to network failures such as failing network links, switches or server hardware or software. A service developer therefore might implement several mechanisms in order to detect and act upon these failures. In this section we focus on two categories of failures which a service developer might anticipate: i) data plane failures, i.e. failures in the OVS components of the elastic router, or ii) control component failures, i.e. failures in the OpenFlow controller component.

Figure 43 illustrates the use of two redundant switching instances, i.e. a primary and a backup OVS instance in order to handle a failing data plane component. The Controller is in charge of triggering the re-wiring of the NFs in case a failure is detected, and is also in charge of handling state migration in between both OVS components. State migration might occur proactively in the form of a regular sync process driven by the Controller NF in between the state of OVS1 and OVS2, or might happen re-actively when the failure is detected. Failure detection might happen

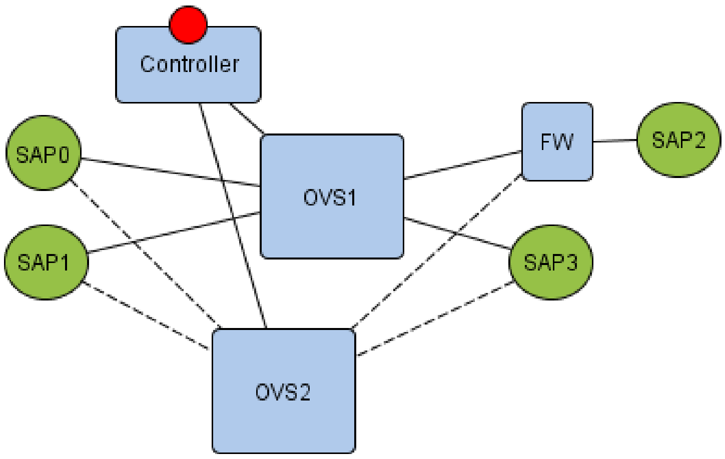


Figure 43: Resiliency in the data plane through redundant use of switching (i.e., OVS) components

via the control channel between the Controller and the OVSes (detecting loss of signal), or might occur through the integration of observation and/or monitoring points. When the failure is detected, the controller shall make sure that the state is correctly transferred between primary and backup OVS (pro-actively or re-actively) and shall connect the SAPs to which OVS1 NF was connected to OVS. This might happen through either the Cf-Or interface, steering the RO to reconfigure BiS-BiS component to which both OVS1 and OVS2 are connected, as indicated in Figure 44, or might involve the configuration of additional switching components connecting between the SAPs and the OVSes and thus not requiring interaction with the RO. The proposed approach might be extended to pools of OVS NFs which are kept in sync, and which might be selected upon failure detection drawing from techniques known in the Reliable Server Pooling Protocols (cfr. [Lei+08]).

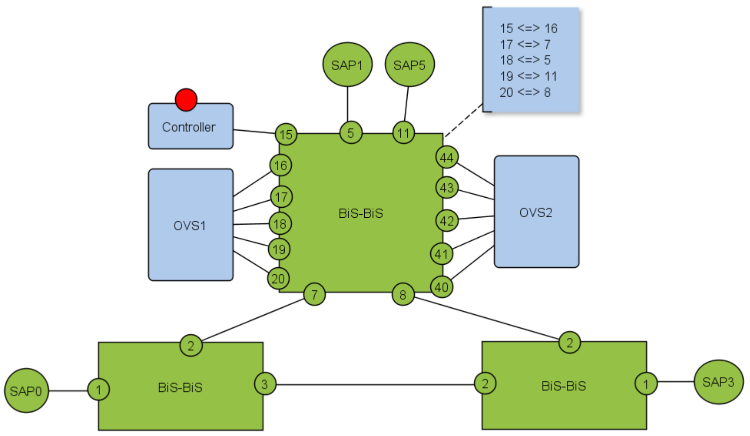


Figure 44: Resiliency in the data plane through redundant use of switching (i.e., OVS) components at the NF-FG level

Resiliency at the level of the Control NFs of the Elastic Router also might apply similar redundancy techniques, by creating a pool of one or more Controller NFs between which state is synchronized or migrated. However, this requires a component which is in charge of the detection, state migration and failover process. The UNIFY architecture again enables service developers to bundle this functionality into a separate, higher-level Control NF. We refer to this NF as the CP Controller in Figure 45. This setup is an illustration of how multiple Cf-Or interfaces might be combined into a single service deployment, each with a different scope. The original Controller NF has a scope between SAP1 to SAP3 and the newly created SAP4 hiding the connection to the FW NF. This scope remains active in this case, however, the role of the active Controller NF might be taken by either the primary Controller1 or the backup Controller2. The scope of higher-level CP Controller Cf-Or however is limited to the connection between it and both Controller1 and Controller2 as indicated in Figure 45. Both Cf-Or interfaces are therefore independent of each other and might operate in a parallel fashion, one to update the connectivity between the OVSes or to instantiate multiple OVSes (as documented in prior sections), and the Controller NF Cf-Or to instruct the RO to rewire a Controller to existing dataplane OVS instances. A Controller NF failover process thus involves the following steps:

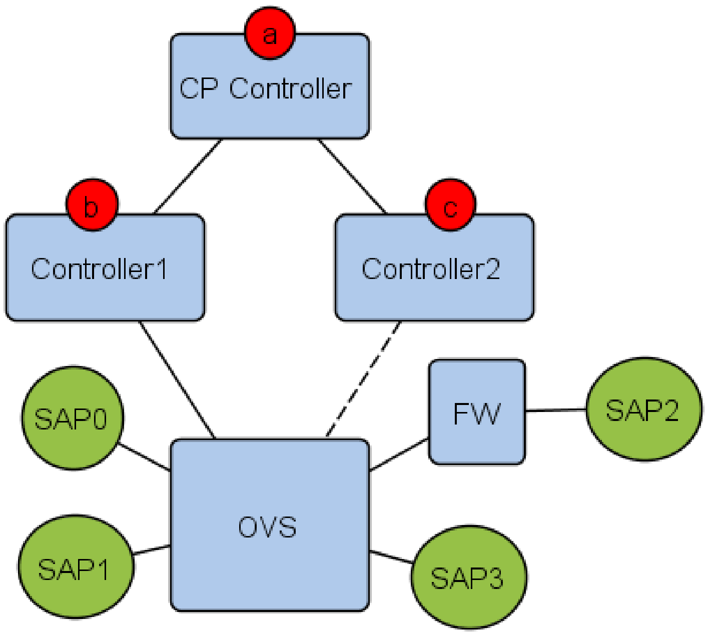


Figure 45: Resiliency in the control plane through redundant use of controller components at the NF-FG level

1. Continuous state migration between Controller1 and Controller2 steered by the CP Controller or on-demand state migration (as indicated in section 2.6 this might either happen in a direct fashion in beetween the Controller NFs or via the CP Controller).
2. Failure detection of a Controller NF by the CP Controller (either through direct polling or through the use of observation or monitoring points)
3. Re-wiring the connection between the Controller NF and the OVS instance(s) via the Cf-Or interface in order to update the forwarding rules of involved BiS-BiS components (note that this also might be activated by involving an additional switching component between the CP Controller and Controller1 and Controller2 which can be configured without Cf-Or/RO involvement). This is illustrated in Figure 46.

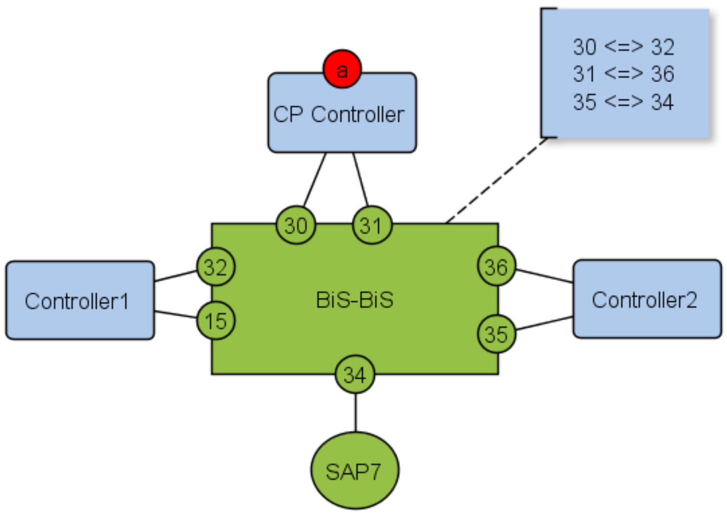


Figure 46: Resiliency in the control plane through redundant use of a CP Controller able to re-wire Controllers to OVSes NF-FG level

3 Requirements for Service Programming and Orchestration

Next, we check the coverage of the requirements on the programmability framework, gathered in deliverables D2.1 and D3.1 and summarized in table 3 below [D2.1; D3.1].

Table 3: Requirements for Service Programming and Orchestration.

Source	ID	Requirement	Coverage
D2.1	Req 2-1	Descriptions of virtual and physical network functions as well as forwarding information must be available for programming service graph.	Covered by the NF-FG model.
D2.1	Req 2-2	Configuration data must be conveyable and semi-transparent.	The UNIFY NF-FG focus on resource orchestration, this is transparent to configuration processes as available in state-of-the-art practice. The ESCAPE prototype implements configuration processes through NETCONF. This could also be covered by the inclusion of a control channel connectivity in the NF-FG for the included NFs.
D2.1	Req 2-3	OpenFlow compatible traffic steering definitions should be supported.	This is covered through the possibility of using (abstract) flow rules in the NF-FG.
D2.1	Req 2-4	Performance indicators of service graphs and network functions should be supported as part of the descriptions.	This is covered by options available in the MEASURE framework.

Continued on next page

Table 3 – continued from previous page

Source	ID	Requirement	Coverage
D2.1	Req 2-5	Preferences, restrictions and pinning of service graphs and (virtual) network functions should be supported as part of the descriptions.	This is covered by the ability of mapping NFs to virtualizers within the NF-FG model (based on information that is shared by potential additional attributes of virtualizers or BiS-BiS). E.g., one BiS-BiS node shown per geographical location, coordinates included in BiS-BiS name. If geographical location doesn't matter, then a single BiS-BiS node can be used, representing the whole area.
D2.1	Req 2-6	Optimisation triggers and goals of service graphs and (virtual) network functions should be supported as part of the descriptions.	This is covered by options available in the MEASURE framework.
D2.1	Req 2-7	Integration/instantiation of new virtual or physical network functions must be possible.	Covered by the NF-FG model.
D2.1	Req 2-8	Steering traffic to, from and between (virtual) network functions must be supported.	Covered by the NF-FG model.
D2.1	Req 2-9	(Virtual) network function and traffic steering demands and configurations must be modifiable.	NF-FGs might be updated through Sl-Or or Cf-Or interface.
D2.1	Req 2-10	Triggers, subjects and constraints should be taken in consideration in the programmability framework.	This is covered by options available in the MEASURE framework.
D2.1	Req 2-11	Resource availability exposure should be provided at different granularity levels (service performance parameters, location, capacity, etc.).	Covered by ability to update available resources in virtualizers of the NF-FG model.
D3.1	U-Sl/ Sl-Or 1	MUST include which SAPs are involved, and which NFs (both virtual and physical NFs MUST be supported) are required in the service (given that these NFs are listed in the NF catalogue)	Covered by the NF-FG model, the NF-IB, and decomposition process.
D3.1	U-Sl/ Sl-Or 2	MUST include a specification of connectivity types and connectivity levels in between NFs and/or SAPs. This SHOULD support flow space definitions.	The BiS-BiS and SAPs of the NF-FG model support (abstract) flow rules.
D3.1	U-Sl/ Sl-Or 3	SHOULD be able to provide SLA parameters on traffic requirements and its scope	This is covered by the MEASURE framework and the Recursive Query Language (Section 5.3 of D4.2)as well as by the possibility to have external and internal link characteristics of NFs in the NF-FG model.
D3.1	U-Sl/ Sl-Or 4	SHOULD support the attachment of performance indicators or Key Quality Indicators to NFs, the connectivity between NFs or on combinations of both	This is covered by possibility to have external and internal link characteristics of NFs in the NF-FG model.

Continued on next page

Table 3 – continued from previous page

Source	ID	Requirement	Coverage
D3.1	U-Sl/ Sl-Or 5	SHOULD support constraining the mapping of service components to the physical infrastructure (including pinning down NFs to particular resources)	This is covered by the ability of mapping NFs to virtualizers within the NF-FG model.
D3.1	U-Sl/ Sl-Or 6	SHOULD be able to specify resiliency required of NFs, connectivity between NFs or combinations of both	This may be tackled by appropriate ues of CTRL NFs and the Cf-Or interface.
D3.1	U-Sl/ Sl-Or 7	MAY support the characterization of optimization triggers related to the mapping of service components to the physical infrastructure (e.g., related to traffic characteristics)	This is covered by options available in the MEASURE framework and by the ability of mapping NFs to virtualizers within the NF-FG model (based on information that is shared by potential additional attributes of virtualizers or BiS-BiS, e.g., geographical information or coordinates).
D3.1	U-Sl/ Sl-Or 8	SHOULD be able to specify scaling requirements of service components	This may be tackled by appropriate use of CTRL NFs and the Cf-Or interface.
D3.1	U-Sl/ Sl-Or 9	SHOULD specify restrictions on what traffic is allowed in the Service Graph	This is covered through the possibility of using (abstract) flow rules in the NF-FG.
D3.1	U-Sl/ Sl-Or 10	SHOULD be able to specify service-specific policies defined by users	This is covered through the possibility of using flow rules in the NF-FG or MEASURE framework constructs.
D3.1	U-Sl/ Sl-Or 11	MAY specify how billing should be performed	Framework allows to be extended to support this.
D3.1	U-Sl/ Sl-Or 12	The reconfiguration of a service MUST support the addition or removal of NFs, links or SAPs, and the modification of any of the characteristics mentioned in the above requirements.	NF-FGs might be updated through Sl-Or or Cf-Or interface.
D3.1	Sl-Or 1	The NF description MUST include resource requirements in terms of computation, storage and memory requirements in order to enable mapping to infrastructure	This is covered by the resource requirements encoded in the NF-FG and NF-IB architecture.
D3.1	Sl-Or 2	Key Performance Indicators (KPI) related to ENFs or interconnected groups of NFs MUST be measurable	This is supported by the MEASURE framework as well as the SP-DevOps monitoring framework developed in WP4.
D3.1	Cf-Or 1	When programming the VNF as a component of the Service Graph its description MUST be able to contain compute and store resource demands.	This might be communicated through the virtualizer datamodel and might be stored in the NF-IB.

Continued on next page

Table 3 – continued from previous page

Source	ID	Requirement	Coverage
D3.1	Cf-Or 2	SHOULD be able to create and upgrade or remove NF images in an operational environment	NF images can be created, upgraded and removed to the NF-IB in a continuous manner. Service can dynamically use them through the Cf-Or interface.
D3.1	Cf-Or 3	SHOULD be able to modify (add/remove) links between NFs	This is covered by the Cf-Or interface.
D3.1	Cf-Or 4	SHOULD be able to change the NF description-related requirements:	This is covered by the Cf-Or interface.
D3.1	Cf-Or 5	SHOULD be able to modify the link requirements	This is covered by the Cf-Or interface.
D3.1	Cf-Or 6	SHOULD enable scaling of NFs (e.g. resize NF resources)	This is covered by the Cf-Or interface.
D3.1	Or-Ca 1	MUST support the Sl-Or interface requirements, as described there.	Covered by integrated prototype, but not primary scope of UNIFY.
D3.1	Or-Ca 2	MUST support the resource mapped NF-FG description	Covered by integrated prototype, but not primary scope of UNIFY.
D3.1	Or-Ca 3	MUST not be specific to any controller	Covered by integrated prototype, but not primary scope of UNIFY.
D3.1	Or-Ca 4	SHOULD support the merged NF-FG view (because it will be scoped to domains/controllers by the CA)	Covered by integrated prototype, but not primary scope of UNIFY.
D3.1	Or-Ca 5	In case of the Orchestrator and the Controller Adaptation are not separated, this interface MAY not exist or MAY be internal/proprietary in the given implementation	Covered by integrated prototype, but not primary scope of UNIFY.
D3.1	Ca-Co 1	MUST support a subset of the north bound interface (NBI) of the controller	Covered by integrated prototype, but not primary scope of UNIFY.
D3.1	Ca-Co 2	SHOULD support at least the minimal subset needed to initiate a NF and interconnect the initiated NF with the domain boundary (if applicable)	Covered by integrated prototype, but not primary scope of UNIFY.
D3.1	Ca-Co 3	MUST NOT contain information which is not related to the domain/controller scope (except reference to domain edges to other domains)	Covered by integrated prototype, but not primary scope of UNIFY.
D3.1	Ca-Co 4	SHOULD be specific to the given Controller. In case of networking, it MUST be able to describe the connectivity between the NFs. In case of computations, it MUST be able to manage NFs (including initiating, configuring, ...)	Covered by integrated prototype, but not primary scope of UNIFY.
D3.1	Ca-Co 7	MAY be skipped, in case of a domain which is able to directly receive NF-FGs.	Covered by integrated prototype, but not primary scope of UNIFY.

Continued on next page

Table 3 – continued from previous page

Source	ID	Requirement	Coverage
D3.1	Co-Rm 1	MUST support at least one north-bound interface of network switching equipment in order to start/stop, configure, model and discover switching functionality	Covered by integrated prototype, but not primary scope of UNIFY.
D3.1	Co-Rm 2	MUST support at least one north-bound interface of a server platform in order to start/stop, configure, model and discover NF and server functionality	Covered by integrated prototype, but not primary scope of UNIFY.

4 Conclusions

The main contribution of this deliverable is the finalization of the programmability framework for controlling carrier and cloud networks. Based on the initial framework defined in [D3.1], different components were refined and improved, addressing several research challenges as documented in [D3.2; D3.2a], leading to the final framework documented here which contains:

- The **core programmability flows** and the **information models** to be used in each of the defined reference points.
- The **orchestration process**, with additional detail on the functionality of the **Network Function Information Base**, the **decomposition process** and the **results of the Divine embedding algorithm**.
- The **abstract interfaces** of the architecture, with additional detail on the possibilities opened up by the Cf-Or **interface**, exemplified in the Elastic Router use case.
- The mechanisms for **state migration**, required for framework-supported service (or VNF) scalability and resiliency, exemplified in both the Elastic Router and FlowNAC use cases.
- To conclude the work on the framework, the **requirements** established by both WP2 and WP3 are revised and crosschecked against the final framework to ensure they are satisfactorily fulfilled.

In parallel, [D3.4] describes the prototype provided by WP3 for the integrated prototype, which is built around ESCAPE, a modular framework designed to support efficient integration of different modules implemented by different partners and the easy re-use and integration of available tools. Subsequent updates, if required by the Integrated Prototype, will be collected in D3.5 Programmability framework prototype report. The main focus of WP3 from now onwards, will be the integration of additional functionalities over ESCAPE, as well as the use cases mentioned in the sections about the Cf-Or interface (2.5.1) and State migration (2.6), namely the Elastic Router and FlowNAC. Aiming towards the Integrated Prototype, both use cases can take advantage of SP-DevOps observability features for the management of the triggers of the state migration.

References

[Bee+12] Leander Beernaert, Miguel Matos, Ricardo Vilaça, and Rui Oliveira. "Automatic elasticity in openstack". In: Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management. ACM. 2012, p. 2.

[BS15] Mario Baldi and Amedeo Sapio. "A network function modeling approach for performance estimation". In: 2015 IEEE 1st International Forum on Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI 2015). IEEE. 2015.

[CRI15] CRIU homepage. CRIU - Checkpoint/Restore In Userspace. 2015. URL: http://criu.org (visited on 10/16/2015).

[D2.1] Mario Kind et al. Deliverable 2.1: Use Cases and Initial Architecture. Tech. rep. UNIFY Project, 2014.

[D2.2] Robert Szabo, Balazs Sonkoly, Mario Kind et al. Deliverable 2.2: Final Architecture. Tech. rep. UNIFY Project, 2014.

[D3.1] Wouter Tavernier et al. D3.1 Programmability framework. Tech. rep. D3.1. UNIFY Project, Oct. 2014.

[D3.2] Pontus Skoldstrom et al. D3.2 Detailed functional specification and algorithm description. Tech. rep. D3.2. UNIFY Project, Apr. 2015.

[D3.2a] Pontus Skoldstrom et al. D3.2a Network Function Forwarding Graph specification (Supplement to D3.2). Tech. rep. D3.2sup. UNIFY Project, July 2015.

[D3.4] Balazs Sonkoly et al. D3.4 Prototype Deliverable. Tech. rep. D3.4. UNIFY Project, Nov. 2015.

[D4.2] Rebecca Steinert, Wolfgang John et al. Deliverable 4.2: Proposal for SP-DevOps network capabilities and tools. Tech. rep. D4.2. UNIFY Project, Sept. 2015.

[Dob+09] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. "RouteBricks: exploiting parallelism to scale software routers". In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM. 2009, p. 15.

[ETS13] ETSI. White Paper: Network Functions Virtualisation (NFV). 2013. URL: http://portal.etsi.org/NFV/NFV%5C_White%5C_Paper2.pdf.

[Fis+13] Andreas Fischer, Juan Felipe Botero, Michael Till Beck, Hermann de Meer, and Xavier Hesselbach. "Virtual Network Embedding: A Survey". In: IEEE Communications Surveys & Tutorials 15.4 (2013), pp. 1888–1906. ISSN: 1553-877X. DOI: 10.1109/SURV.2013.013013.00155. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6463372%20http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6463372.

[KDS15] Babu Kothandaraman, Manxing Du, and Pontus Sköldström. "Centrally Controlled Distributed VNF State Management". In: Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization. ACM. 2015, pp. 37–42.

[Kni+11] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. "The Internet Topology Zoo". In: Selected Areas in Communications, IEEE Journal on 29.9 (Oct. 2011), pp. 1765–1775. ISSN: 0733-8716. DOI: 10.1109/JSAC.2011.111002.

[KRK14] Matthias Keller, Christoph Robbert, and Holger Karl. “Template Embedding: Using Application Architecture to Allocate Resources in Distributed Clouds”. In: Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing. IEEE Computer Society. 2014, pp. 387–395.

[KVM15] KVM homepage. Migration - KVM. 2015. URL: http://www.linux-kvm.org/page/Migration#Algorithm_.28the_short_version.29 (visited on 10/16/2015).

[Lei+08] Peter Lei, Lyndon Ong, Michael Tüxen, and Thomas Dreibholz. An Overview of Reliable Server Pooling Protocols. Informational RFC 5351. ISSN 2070-1721. IETF, Sept. 2008. DOI: 10.17487/RFC5351. URL: https://www.ietf.org/rfc/rfc5351.txt.

[M4.3] Pontus Sköldström et al. M4.3 Update on SP DevOps with focus on automated tools. Tech. rep. M4.3. UNIFY Project, Oct. 2015.

[MKK14] Sevil Mehraghdam, Matthias Keller, and Holger Karl. “Specifying and placing chains of virtual network functions”. In: 2014 IEEE 3rd International Conference on Cloud Networking (CloudNet) (Oct. 2014), pp. 7–13. DOI: 10.1109/CloudNet.2014.6968961. arXiv: arXiv:1406.1058v1. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6968961.

[OAS15] OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) Technical Committee. TOSCA Simple Profile for Network Functions Virtualization (NFV) Version 1.0. Tech. rep. OASIS, May 2015. URL: http://docs.oasis-open.org/tosca/tosca-nfv/v1.0/tosca-nfv-v1.0.pdf.

[Ros+15] Steven Van Rossem, Wouter Tavernier, Balazs Sonkoly, Didier Colle, Janos Czentye, Mario Pickavet, and Piet Demeester. “Deploying elastic routing capability in an SDN/NFV-enabled environment”. In: Proceedings of the 2015 IEEE-NFV-SDN. IEEE. 2015.

[Rot+12] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W Moore. “OFLOPS: An open framework for OpenFlow switch evaluation”. In: Passive and Active Measurement. Springer. 2012, pp. 85–95.

[RRS15] Raphael Vicente Rosa, Christian Esteve Rothenberg, and Robert Szabo. “VBaaS: VNF Benchmark-as-a-Service”. In: Proceedings of the 4th European Workshop Software Defined Networks (EWSDN). EWSDN. 2015.

[RWJ13] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. “Pico Replication: A high availability framework for middleboxes”. In: Proceedings of the 4th annual Symposium on Cloud Computing. ACM. 2013, p. 1.

[Sah+15] Sahel Sahhaf, Wouter Tavernier, Didier Colle, and Mario Pickavet. “Network service chaining with efficient network function mapping based on service decompositions”. In: Network Softwarization (NetSoft), 2015 1st IEEE Conference on. IEEE. 2015, pp. 1–5.

[SBP15] Amedeo Sapio, Mario Baldi, and Gergely Pongrácz. “Cross-Platform Estimation of Network Function Performance”. In: European Workshop on Software Defined Networks (EWSDN 2015). EWSDN. 2015.

[SSS13] Upendra Sharma, Prashant Shenoy, and Sambit Sahu. “A flexible elastic control plane for private clouds”. In: Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference. ACM. 2013, p. 4.

[Sza+15] Robert Szabo, Mario Kind, Fritz-Joachim Westphal, Hagen Woesner, David Jocha, and Andras Csaszar. “Elastic network functions: opportunities and challenges”. In: Network, IEEE 29.3 (2015), pp. 15–21.

[Too+12] Amin Tootoonchian, Sergey Gorbunov, Yashar Ganjali, Martin Casado, and Rob Sherwood. "On controller performance in software-defined networks". In: USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE). Vol. 54. 2012.

[Wan+14] An Wang, Yang Guo, Fang Hao, TV Lakshman, and Songqing Chen. "Scotch: Elastically scaling up SDN control-plane using vswitch based overlay". In: Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies. ACM. 2014, pp. 403–414.

A NF decomposition and dependency support: comparison with ETSI-NFV and TOSCA

We have explained how NF decompositions and dependencies are supported in the designed Neo4j-based NF-IB in Section 2.3. As an add-on to [D3.1] and [D3.2], we provide an overview on how these concepts are supported in ETSI-NFV [ETS13] and TOSCA [OAS15].

TOSCA uses the concept of service templates for cloud workload description. These are graph of node templates modeling the components of the workload and relationship templates are modeling the relations between the components. In order to describe the building blocks of a service template and different kinds of relationship between them, TOSCA provides a type system including node types and relationship type. These types define lifecycle operations which can be used by the orchestrator to instantiate components at runtime and using the relationship between the components, it derives the order of component instantiation.

ETSI-NFV models network services using Network Service Descriptors (NSDs) which is a deployment template for a network service and describes the relationship between VNFs and possibly Physical Network Functions (PNFs) and the required links for connecting VNFs.

Other deployment templates include: i) VNF Descriptor (VNFD) which is used to describe a VNF in terms of its deployment and operational behavior requirements, ii) VNF Forwarding Graph Descriptor (VNFFGD) which is used to describe the topology of the Network Service or part of it, iii) Virtual Link Descriptor (VLD) which describes the resource requirement on the links between VNFs, PNFs and NS endpoints, and iv) Physical Network Function Descriptor (PNFD) which is used to describe the connectivity, interface and KPI requirements of Virtual Links to a Physical Network Function. These descriptors are stored in catalogues used by the orchestrator (NFVO) to instantiate VNFs according to the instantiation input parameters.

Figure 47 illustrates the general mapping of TOSCA and NFV deployment template [OAS15]. As we see in this figure, in TOSCA data model there is a service template at the top level including several node templates with different types. In NFV at the right of the figure, NSD is at the top level which includes the 4 explained descriptors. The NSD can be described as a service template in TOSCA. Other descriptors in NFV are considered as node templates with different node types. VNFD can be further described by another service template.

In support of service decomposition, in ETSI-NFV, a VNF can include: i) Virtualization Deployment Unit (VDU), which is a subset of a VNF which can be mapped to a single VM, ii) connection point and iii) internal virtual link which is used within VNF to have connectivity between VDUs. Therefore a VNF can be decomposed to several VDUs which are connected using internal virtual links.

In TOSCA, a VDU can be described as a node template with an appropriate node type. As a VNFD can be described by a service template with substitutable node types, the service decomposition is supported by using a nested service template. This is illustrated in Figure 48.

In ETSI-NFV, in support of VNF dependencies (used for indicating the order of VNFs instantiation by the orchestrator), the VNFD includes 'dependency' information element which describes dependencies between VDUs. This dependency is defined in terms of source and target VDU which means that the target VDU 'depends on' the source VDU. In other words first the source VDU must exist and then the target VDU is initiated. This can be described by the relationship-types in service templates in TOSCA.

The substitution of node types by service templates in TOSCA corresponds to the high level service decomposition approach in UNIFY. To have a general mapping of TOSCA and a UNIFY domain, one can match the whole UNIFY framework as a 'cloud provider' for a TOSCA client.

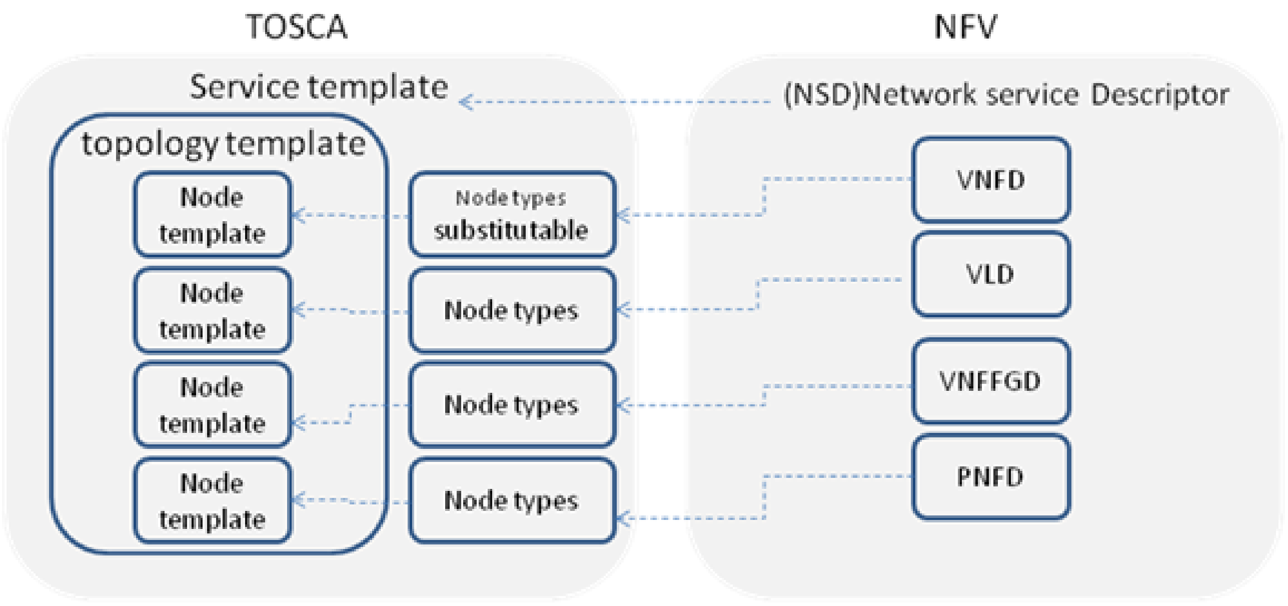


Figure 47: General mapping of TOSCA and NFV deployment template [OAS15]

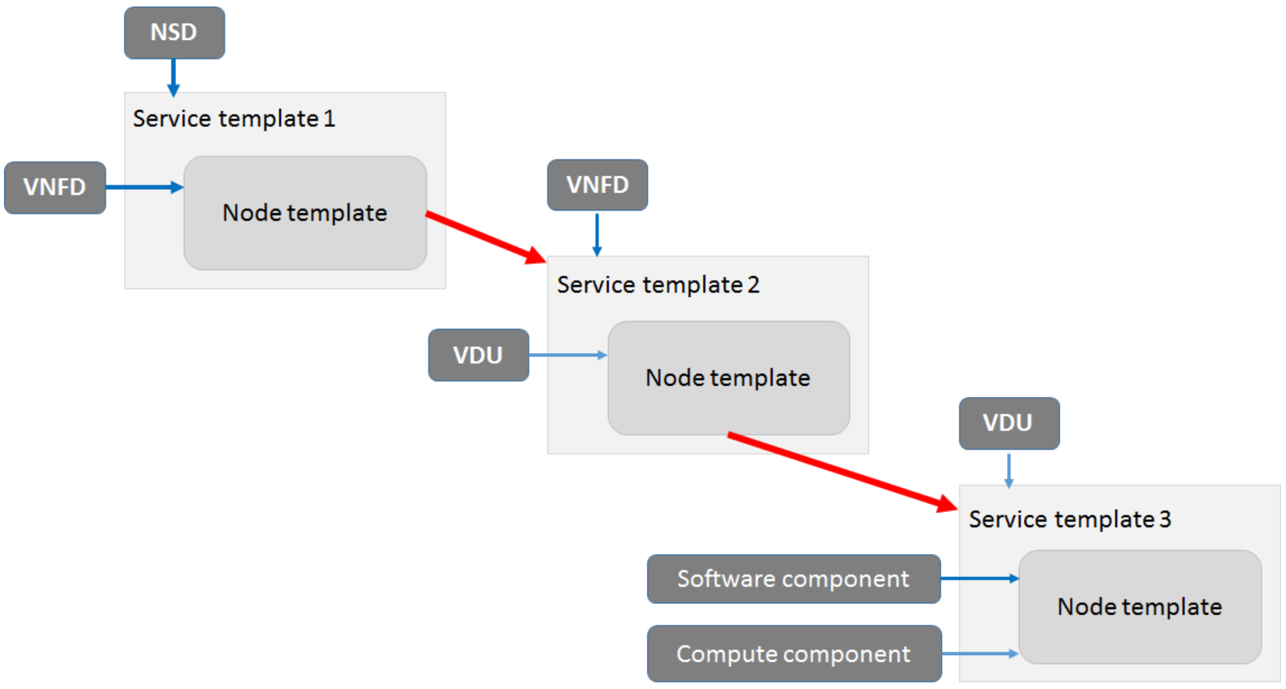


Figure 48: Service decomposition support in TOSCA

Regarding ETSI-NFV and its relation to our model in UNIFY, ETSI defines the Network Service (NS) element, which is the closest element related to the SG defined at UNIFY. Although the SG definition is based on ETSI's NS, the main model used in UNIFY (NF-FG) is different from the NS model in the sense that it represents deployment decisions instead of service description (see [D3.2] for more detailed comparison).