Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of telecommunications and media informatics

Graph separation algorithm for incremental resource (de)allocation in hierarchical orchestration

Scientific Students' Associations Paper

Creator
Recse Ákos

Supervisor
Dr. Szabó Róbert

October 27, 2017

Contents

List of Figures

Abstract

In order to deliver end-to-end services, virtualization providers must cooperate to overcome their physical and virtual boundaries. Federation or hierarchies of autonomous systems will emerge where customers will receive single stop shop global service offerings. The underlying orchestration system (virtualization management), however, must support custom operational policies, request aggregation and de-aggregation, constraints on resource or service level performance indicators, etc. Since requests in these systems do not travel unaltered vertically through the layers, but rather are aggregated together, subordinate systems must employ mechanisms to separate (likely) independent sub-requests in order to maximize their freedom in resource allocation. Additionally, requests are usually formulated as updates (change set) over an existing configuration (see netconf[RFCxxxx]).

The resource allocation problem can be formulated as a multi-constrained graph allocation: a service graph must be embedded into a given topology of resources and capabilities, where the service graph consists of capability typed vertices and edges with local, segment or end-to-end constraints (e.g., end-to-end latency).

Under such conditions, the construction and the continuous reconstruction of independent allocation graphs based on incremental aggregated inputs is highly not trivial. Such separation provides base for the constraint based embedding but also to debugging and visualization tools, when an overview of end-to-end services across autonomous systems and orchestration layers are needed. For example, it is highly non trivial how to follow embedded encapsulations in order to recover individual services over aggregated network segments.

Chapter 1

Introduction

Today 5G slowly comes into play so as Network Fuction Virtuailzation[5] which owns an important role in 5G's approach. In this chapter we would like to give a high level picture of Network Function Virtualization to be able to drill down into the specific subtopic this paper is about.

1.1 Network Fuction Virtualization

The main idea behind Network Function Virtualization follows cloud approaches but they are transplanted into network solutions. This topic is very common today however there are several challenges appearing when talking about such a system. It assumes we can access all the benefits derived from cloud considerations in networking if we replace current hardware based network functions to software based ones. In case we do so a whole spectrum of adventages come into play which can be beneficial both for service providers and the customers.

Today sarting a new network function can be a complex task and entails several chellenges in the system. Thanks to the current workflow of launching new network services it needs widespread knowledge and may requires high amount of cost to complete beacuse the concrete infrastructure is involved in the process. Also a pressing factor can be the contantly growing complexity of network functions.

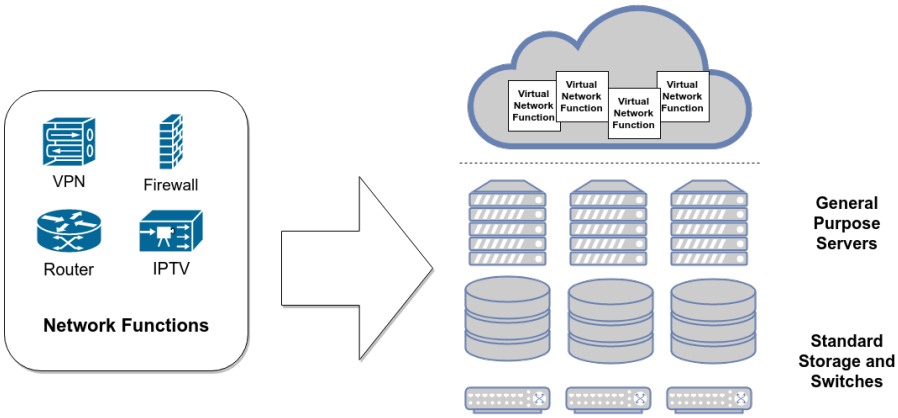

Figure 1.1: *Difference between present-day and NFV approach*

The difference between the current and the Network Fucntion Virtualization approach can be seen on figure 1.1. in details. On the left side you can see the present-day satus which shows individual and hardware based newtork functions. While the right side depicts scheme behind Network Fuction Virtualization: there are only general purpose computers in the system and all functionality realized in software. In term of Network Function Virtualization we can apply a cut between physical infrastructure and the Network Functions based on the separation described before. The first one in often called Network Function Virtualization Infrastructure or NFVI and contains the underlying network and devices whilte the latter is Virtual Network Function or VNF refering to their software based manner.

This new stance carries numerous benefits which can save time or even money for the infrastructure owners. One example for that relies on the financial costs of the infrastructure itself. There is a high gap between the price of general purpose servers compared to some task specific devices. So as at maintain cost of the mentioned hardwares what can be reduced also. In case of maintainability there is not only cost reduction can be achieved but the time and complexity of upgrading or fixing can be lower because we can think of these tasks as software updates now. There is the possibility for testing the Virtual Network Functions in the real environment which is not that trivial in case of the old approach. An important adventage is the versatility of the system which makes possible to act upon to the almost real-time utility.

However these adventages makes Network Function Virtualization attractive there are some challenges which has to be solved. These are described in the following bullet points:

Compatibility Through the process of switching between concepts the legacy approach must be supported without trade-off.

Performance A software running on general server can never be as fast as a dedicated hardware for that task. Altough there have to be a way to minimize this performance loss.

Simplicity We aim to create an easy to use system in general so clarity should be on top of the agenda.

Security Refering back to cloud computing there are several security issues have to be considered.

Management Important job is to manage virtual functions in an optimal way.

1.2 Motivation

In this section we would like to describe the incentive behind this new approach and palce it in the network nunction virtualization workflow. As it was mentioned before in NFV we would like to create software based network services and depoly them on general purpose servers. These servers must be conneced together especially because we work with them in a 5G related project. In this case the underlying infrastructure can be imagined as a

network of servers with the servers themself and the network links among them. We would like to provide a simple way for starting new services in the system but this infrastructure can be quite complex. It can contain several subparts built upon several technologies that results a hard job to deploy these service requests to proper places what is all against the goal to keep deploying simple. To avoid dealing with these wide range of technologies there is an option for creating an abstraction layer over the physical infrastructure. As a result the actor who sees the whole topology can show a simplified picture of the infrastructure where there is enough knowledge for someone to be able to send a service request. Because this request is defined on the compressed picture of the topology the actor who simplified it have to make a mapping back to the extended physical infrastructure which procedure is called orchestration. Also it is possible to apply abstraction over the already compressed infrastructure view in many levels until only one infrastructure element is shown.

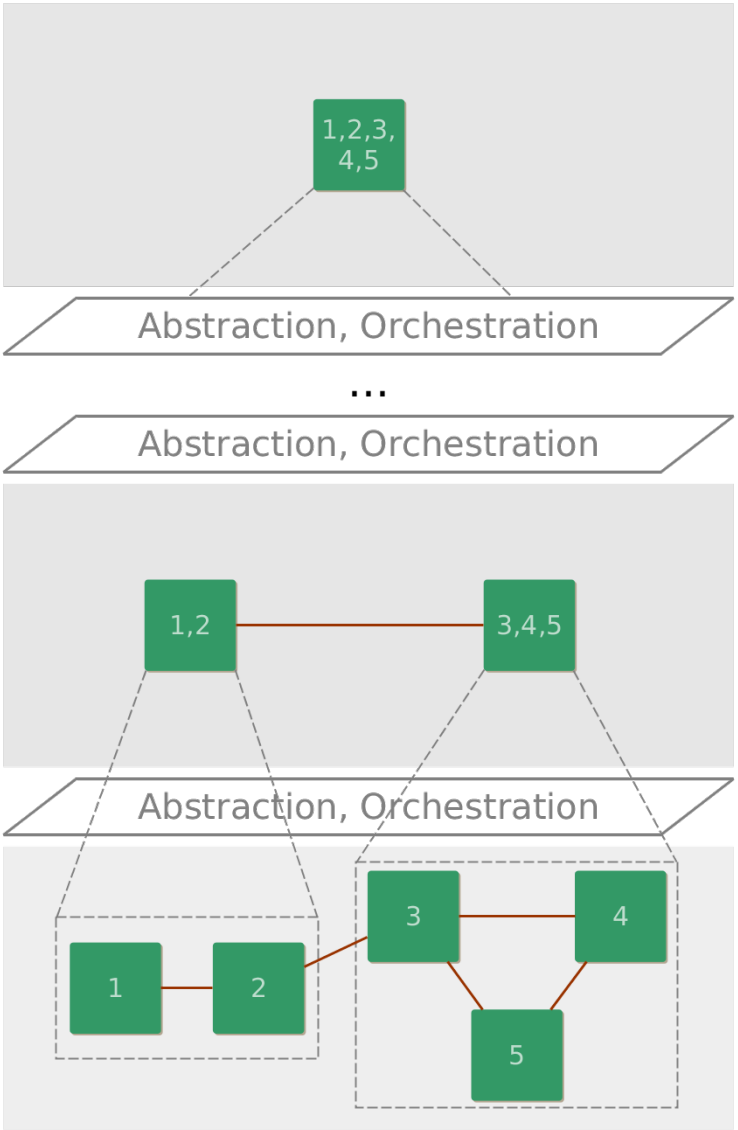

Figure 1.2: *Abstracting infrastructure in several levels*

Figure 1.2. depicts this abstraction through multiple levels. On the very bottom of the

figure there is a detailed view of some topology. Not neccesarily the physical infrastructure, altought it can be. The dashed box encapsules the elements abstracted in the next level. While the lower level contains 5 elements in this basic example on the next level there are only 2 elements shown thanks to the abstaction. This can countinue over numerous steps and at the end maybe one homogeneous domain will show up.

As service requests starts to come in they can build up a chain of individual services defined between two fix points of the infrastructure. These fix points also serve as the base of the orchestration where the upper layers compressed view can be connected to the lower level detailed view. These fix points appear on both views as the services can be accessed through these points and this is why we call them Service Access Points or SAP for short. These are assigned to a general purpose server on the bottommost level or to be exact it is a port of a server where services can be accessed. As we start to add abstraction over the topology these ports start to transform into a logical element in the view of the actual level.

An other aspect of the system to consider is its distributed manner. Each actor in the network can show the abstracted view to several other actors who may want to send service requests. These actors can stand for their own domain which can contain several elements. We can think of domains like technically different groups supervised by an actor. Because of this decentralized operation service requests can come in from several places at the same time. This is important to notice these individual requests based on the same view of an infrastructure. When we try to define service chains we can step over the boundaries of domains what results a chain which consist of several partial requests from different actors. However these requests arrive in multiple parts we would like to follow them and find out they are coherent.

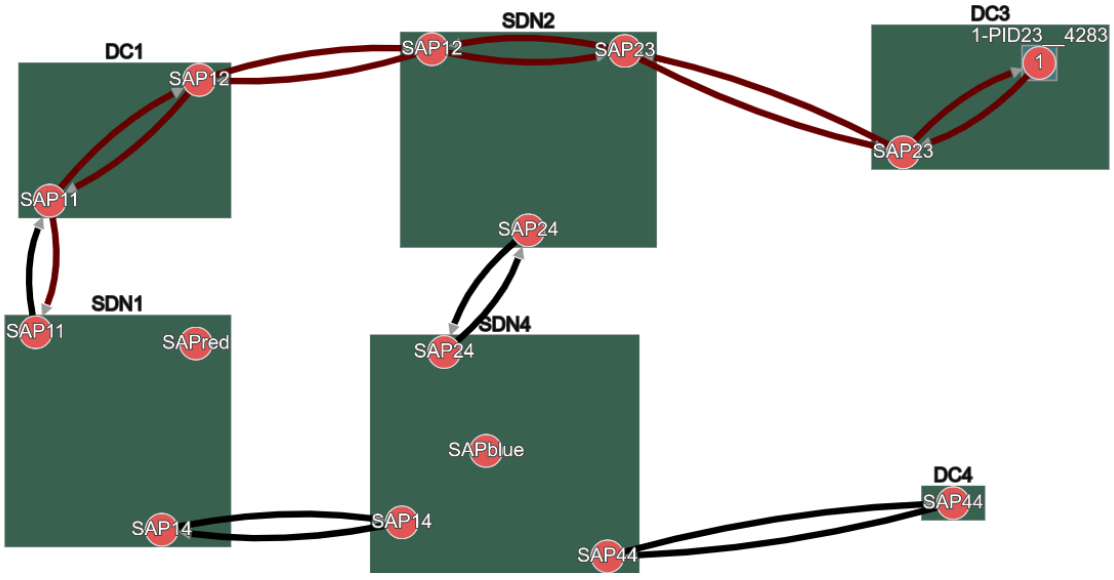

Figure 1.3: *Example for a service chain between two SAPs*

As it was mentioned before individual services can be connected together to form a service chain. These service chains can be accessed through SAPs, however they may not

use the shortest path between the start and destination point because of other preferences orchestration have to deal with. These introducing factors can be for example the provided capability of a server which determines if it can run a Virtual Network Fuction or not. There also can be requirements for resources like memory storage or computational performance. As a result these service chains may pass through several servers and links until they reach the destination SAP.

To provide human-processable representation of the infrastructure, applied abstractions and service requests a visualization tool was created which also generated the Figure 1.3.. In this tool the visualization of these service chains take great part in distinctness of images. It is easy to see each layer can be represented as a graph what visualization is also based on. The arriving different requests for services most of the time does not contain complete information only some partial data about where to put elements exactly. This is because of the system works in a distributed way. On the one hand these individual requests must be aggregated into one graph that represents the infrastructure. We must also be able to show individual service chains based on the aggregated view of requests and on the ground of local decisions. The algorithm detailed in this paper provides a real-time solution for that problem.

1.3 Precedent Use-Case

In this section we would like to present an example use-case when this algorithm is applied. This demo setup also used by Ericsson to showcase the 5G Exchange project because of that the requests shown here saved from a real run of the algoithm and the underlying network. To simulate this old run for presentation purposes requests sent by simulated source senders but over this difference the process is the same from this systems perspective.

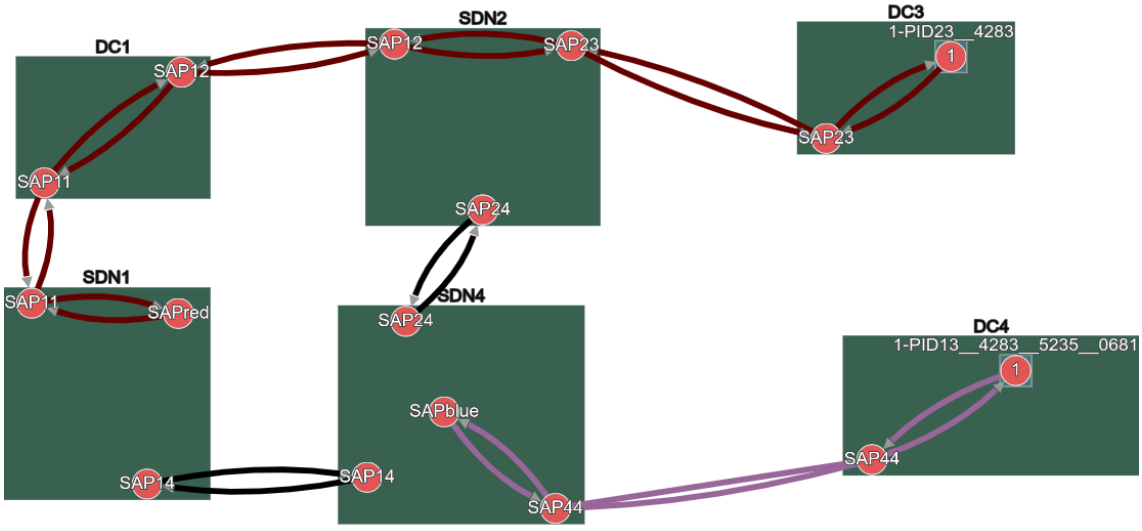

Figure 1.4: *Basic setup of robotics use-case*

In this setup there are 4 domain orchestrator actors take place. These can orchestrate their own domain then send requests to the system. These domains are interconnected

through SAPs in a defined way which is shown on figure 1.4. As you can see the domain orchestrators can own two kind of infrastructure nodes: one is a Docker Container and the other is an SDN[1] network. There are two additional actors appear called Robot1 and Robot2. These were balancing LEGO robots with two wheels and tried to stand on those. To perform this balancing they had basically a balancer service or a proportional-integral-derivative (PID) controller. The robots can access this network through the SAPblue and SAPred ports in SDN4 and SDN1 nodes. As you can see on the figure there are the two virtual network functions (1-PID23 and 1-PID13) and the robots can access them with their own service chains. Later this will be extended with most complex requests.

[1]Software Defined Network

Chapter 2

Design

This chapter aims to describe the components involved in the algorithm and the method itself. As described before the algorithm runs on a graph representation of the unerlying network or its abstracted view. To create the graph there should be a clear mapping of network elements to the constructed graph. First of all we have to identify what elements we will include in the graph. Through this chapter all required elements will be described in details.

2.1 Requirements

The goal of the algorithm is clear however there are some additional requirement towards the implemented software and also important to define the existing ones.

Creation of graph

The algorithm have to create a graph from the incoming requests and should be able to maintain it based on other incoming requests. These requests may associated with the underlying network or with virtualized network functions.

Service chain

Within the algorithm independent service chains must be identified however they are may not defined explicitly. This can happen by following defined paths between SAPs while we become able to visualize the graph and get adventage for some graph management steps.

Distributed environment

Requests may come in from different senders who only knows their own part of the system. Because of that in the recieving server these parts must be combined together to be able visualize the intresting interdomain service chains.

Local decision

One of the most important feature it has to fulfill is the handling of requests with only partial information. The individual requests must be aggregated but at the same time the service chains must be discovered based on only local informations. Through the next sections we will introduce all the components involved during the execution of the algorithm.

2.2 Infrastructure Node

The main units that can serve Virtual Functions are called Infrastructure Nodes in this environment. We can think of servers as infrastructure nodes so as their abstracted logical identities. However these elements carry many essential information they do not appear in the graph directly. Althought are stored and at the end of the algorithm a visualizable serialization of this graph is created where these element take place. This is neccessary because without infrastructure nodes the visualization would be meaningless. On the visualization they are marked with green squares.

2.3 Ports

Port appear on two levels: once Infrastructure Nodes are connected through their ports together. Second virtual network functions also own ports to be able to connect them to their infrastructure node. After an abstraction these ports are rather logically called ports than they really are. There is no limit for the number of ports an infrastructure node or a virtual network function can have.

SAPs are special ports as described previously. These ports can mean a start or a destination port of service chains while also play important role in abstraction. SAPs can appear both at an infrastructure node and a virtual network function however in the latter case they only have meaning inside the system. An other task SAPs have is to connect infrastructure nodes and virtual network functions together without concrete links between them. The need for this option derives from a higher level in this project.

Ports are marked with red circles on the visualization and they belong to the closest containing node in the hierarchy. So if a port is contained by a virtual network function and an infrastructure node also it is in term of virtual network function.

2.4 Links

Links are connecting infrastructure nodes together to present a network of them. If infrastructure nodes appear inside the same domain they can be connected via links directly. In this case these links defined inside the request that responsibe for creating the proper part in the graph. Althought links can be defined through SAPs as mentioned in section 2.3. The service chains later can only follow the paths defined by these links as prohibited to go directly between not connected infrastructure nodes. Most of the time these links are multidirectional but in the graph we assume them to represent only one direction. As a

result usually there are two edges between connected infrastructure nodes showing the two way permeable manner. Links are displayed with black arrows between the proper ports inside the connected infrastructure nodes.

2.5 Virtual Network Function

Virtual network functions are virtualized services that actors can deploy on infrastructure nodes and then they can be accessed through SAPs later. These can be anything from a firewall to a Content Delivery Network or a routing service. However there are numerous variants available from virtual network functions the type of a virtualized function is irrelevant from the perspective of this task and the work of the algorithm. In the workflow we have to take care of the localization of virtual service and the accessibility of them.

A service chain denotes this two approaches together: the deployed virtual network function on the defined infrastructure node and a path between the given SAPs where the virtualized service can be accessed on. Additionally there is an option to define not only individual virtual network functions but connect multiple of them on a service chain. In this case we also have to construct a path which goes through not one but many virtual services before it reaches the destination SAP.

When someone plans to start a virtualized service he has to deploy it on one of the available infrastructure nodes. If the requests defined on an abstracted view this task leaks down on the abstraction topology and affects the orchestration method. One way or an other virtual network function have to be deployed on a proper infrastructure node with the other virtual functions in the service chain. They can be deployed on any combination of infra nodes - together or completly separated - until the path can follow the defined orders. However this concept is important to understand the algorithm only comes in after this procedure when all virtual network functions have the proper place where they are deployed and the path is planned to and from SAPs. Virtual services are marked with blue squares on the generated images with red ports in it if there is any.

2.6 Flowentry

There were less word about what is inside the infrastructure nodes or how VNFs deployed in this theoretical level. When we place a virtual service on an infrastructure node it has to be able to connect to the SAPs or to some other VNFs. Within infrastructure nodes special rules can define connections between ports. These rules are called flowentries. By the help of them we can connect VNFs' port with the port of the containing infrastructure node. These rules are carrying an important concept for the algorithm. When constructing a rule to connect two ports we have 4 parameters to define:

- **Start Port** The port to connect with an other. As the connection is unidirectional the orientation must be set here.
- **Destination Port** The destination point of the connection.

- **Match** In this parameter we can filter the data by an attribute which is called Tag.
- **Action** Here some tag modification actions can be performed. These actions will be detailed later.

By filling these parameters with exact values we can define a flowentry which is mapped into the graph as an edge. This edge will go from the Start Port to the Destination Port making available to walk from one to an other and will be essencial while chaining virtual services and identifying them.

Identification of services is also based upon the tags used in the system. As VNFs deployed and connected together somehow the data have to follow service chains and can't be mixed with other service chains permanently altought they have to be aggregated to commonly use links between infrastructure nodes. There are complex rules for tag handling but these will identify the service chains.

2.7 Algorithm

At this point all of the concerned components are described and the requirements of the result are defined. In the remaining part of this chapter we would like to present the design procedure of the algorithm that is able to fullfill all described before. Until the end of chapter all important steps will be detailed about the method however in 4.1 it is illustrated on a concrate example which is in association with the use-case presented in section 1.3

2.7.1 Borders of Algorithm

To create an algorithm for a proper task it is neccessary to determine the boundares, inputs and outputs of it. This subsection would like to describe the responsibilities of the system components and tries to introduce the input and output materials the algorithm will work with.

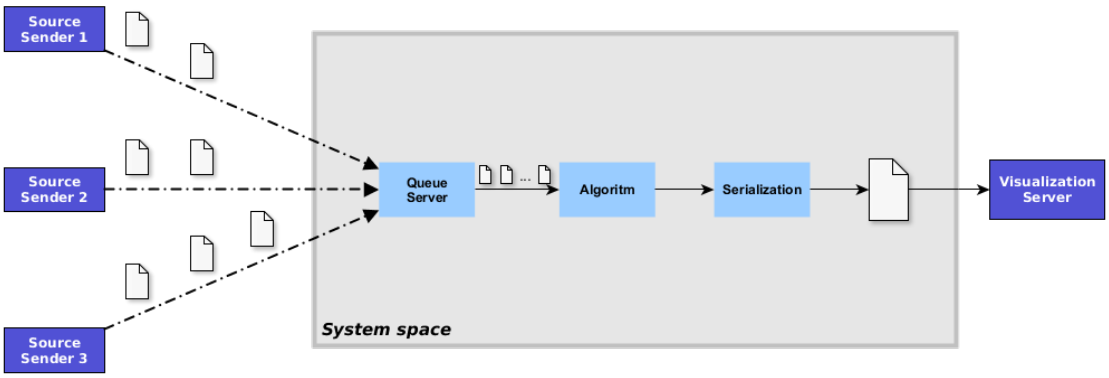

Figure 2.1: *Boundaries of the system and involved actors*

The main workflow of the system depicted in figure 2.1. Our algoritm works wrapped inside a server which waits for requests on a defined address. Request data arrives from

several senders who can only know its own domain so the information only covers the related infrastructure nodes, VNFs and other network elements. However these domain specific infrastructure nodes can be interconnected through their SAPs as detailed in section 2.3. Also these incoming requests contains additional information over the payload where they can specify an attribute called layer. As not all requests cohere together this attribute provides a way to separate the different layers. Sometimes these layers can be mapped to the abstarction layers but this is not neccessary. If different requests come in with the same layer they have to be merged and check for available SAP connections which is the algorithms responsibilitiy. One Source Sender can send requests into several layers if it has the knowledge for that.

As these requests arrive the Queue Server component orders them by income time and feeds the algoritm as a linear stream provider. This happens separated by layers because the server maintains individual graph objects for each of them. The algoritm always runs per layer and per every request altought not every input data contains change. The request processing step is followed by a serialization task where the chagned graph stored in a file which acts as input data for the visualization. Important to notice that Queue server during the preprocessing step creates a difference graph of the actual status and the request where determines what to create and what to delete within the graph instance.

Requests arrive in XML[1] format which is parsed and preprocessed in the Queue Server. This parsed XML object passed to the algorithm to work with. At the end of one processing cycle the serialized data generated in JSON[2] which is easily readable for the visualization tool. The serialization process is responsible for providing the exact format the visualization server can work with.

2.7.2 Patch the graph

Now all the input data is described and we are familiar with the expected result also. After that we can walk through the steps required to achieve the goal we marked out. We know that the input data for the algoritm is a reduced and preprocessed XML representation which is parsed into an object. We also know what operations we would like to execute on the individual elements. At this state the algoritm supports two of these operations: one for creating and one for deleting elements from the topology.

According to this we process a request operation by operation. But not this is the smallest element we work with. It is easy to notice the elements of the network can be considered as a tree also that is why XML works well to store this data. On the top level there is a layer where we interpret this. Within a layer there are infrastructure nodes also they can be connected together with links. We can think of the SAP defined connections as specially assigned links in this case. Inside the infrastructure nodes on the one hand there are the ports of the node. On the other there can be VNFs deployed on them which can also contains ports as described before in this chapter. The connection inside an infrastructure

[1] Extensible Markup Language
[2] JavaScript Object Notation

node is defined with flowentries which can be considered as a part of the node this way. These relations illustrated on figure 2.2.

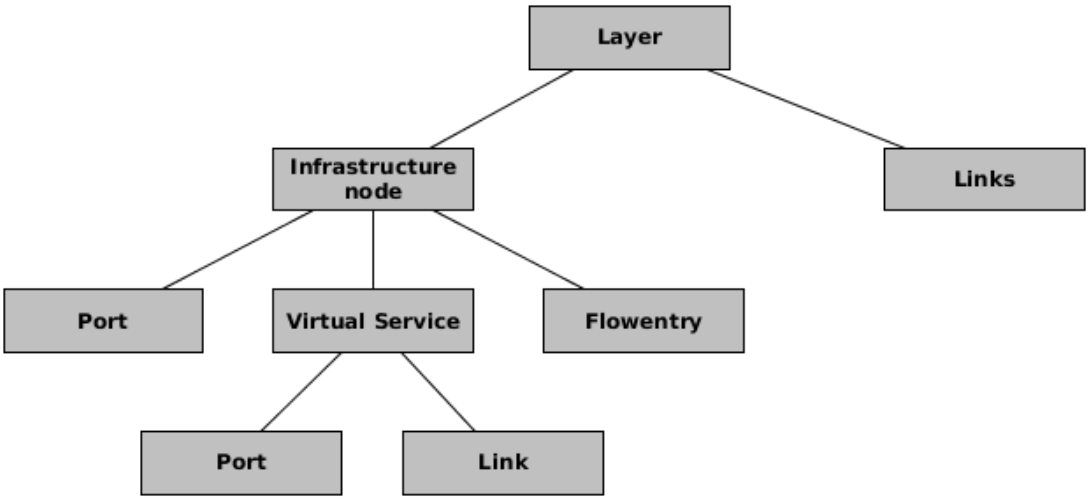

Figure 2.2: *The theoretical tree of network elements*

Incoming requests can contain operations refer to any of these elements. For example when some infrastructural report arrives (e.g. a new infrastructure node is available) the a create operation will come in to the proper infrastructure node. However if some virtual service is stopped the system gets a delete request which points to the virtual function inside an infrastructure node.

According to this the requests can be processed not only diversified by operation but along individual elements also. If we follow the exact procedure of identifying operations and elements we can construct the flowchart shown on figure 2.3. Here we can see how the input called diff object is processed within the Patch function which means the entry point for the algorithm. The procedure works recursively to fit perfectly for the tree-like manner. First we check if we already found on operation or we are still in the root. If we didn't do a search for the next element and the belongign operation. If we can't find any the processing ends as we are done otherwise we call the Patch process again with the found element and operation which will result the first condition evaluated true this time. Then we check if the actually processed element is a graph element or not. We already defined what elements are stored in the graph but notice these are equivalent to the elemetns shown in the tree (Figure 2.2). If the element we processing is one of them we continue the processing in the graph element related patch function. This step is responsible for the extendibility of the algorithm with other network elements. We let each element to know from itself what it requires when creating or deleting one. To give an example: if we are currently processing a create request for a virtual network function we pass the diff object and the operation type (create) to the proper specific patch method and let it perform the task itself.

If this step finished or the actual element is not from the stored elements then start to process its children recursively if it has one. Important to notice at the beginig of the patch we only process the request by operation and by element that means we don't get

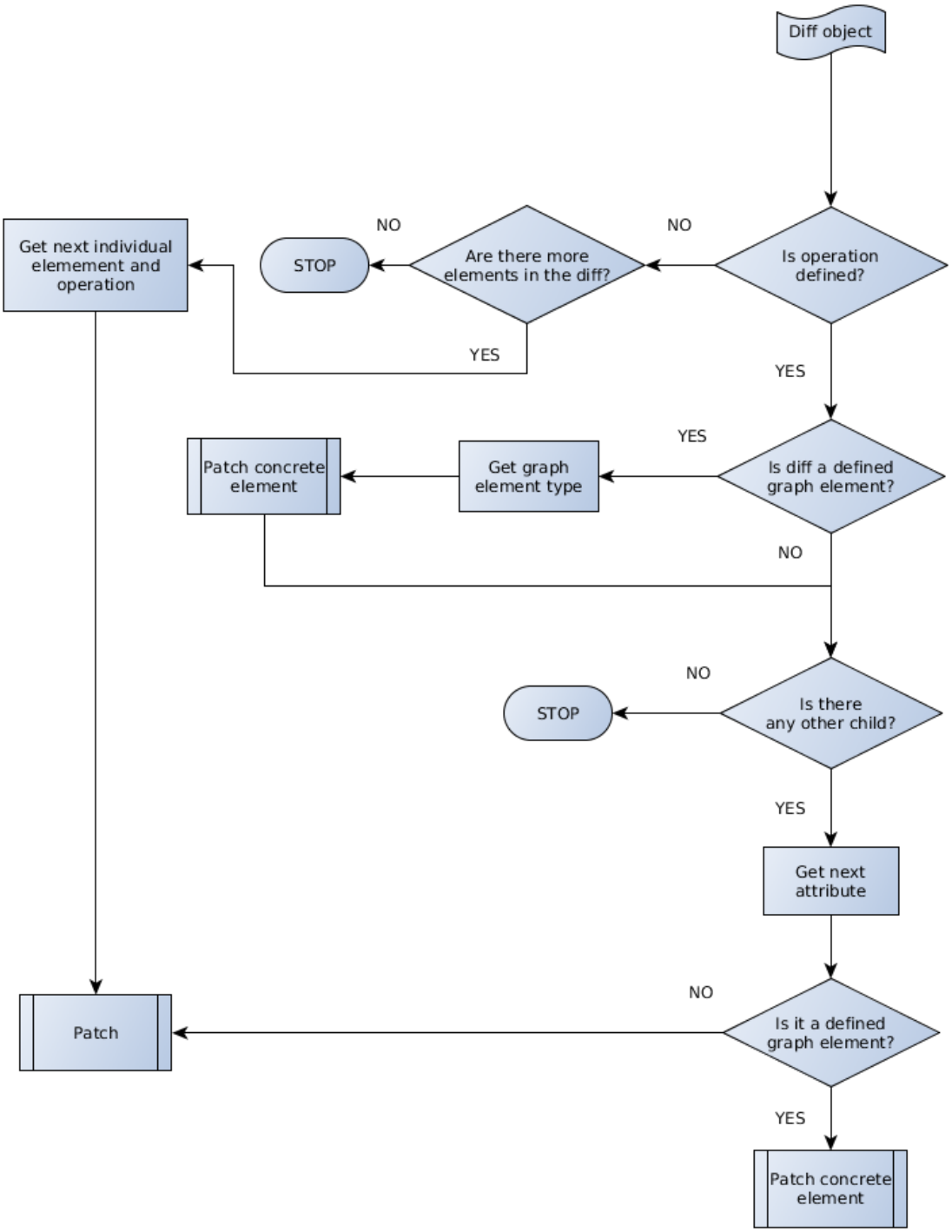

Figure 2.3: *Flowchart of identifying individual elemetns and belongig operation*

the children of an element if the operation comes to the parent. Consider the following example: we get a request that aims to create an infrastructure node with two ports. In this case the only element we get is the infrastructure node itself and not recursively its ports. Althought we want to process them so after we finished the creation of the parent object (the infrastructure node in the example) we recursively have to process its children by calling the same patch function on them recursively.

Inside the patch function of a proper graph element the only thing we have to do is decide what operation to perform and then continue the appropriate method. These methods will be detailed through the next sections.

2.7.3 Create operation

Creation of network elements naturally depends on what kind of element it is. There are some of them which is easy to create and others what requires a complex method to do so.

Infrastructure Node

This is one of the easiest element to parse because it is not stored in the graph only stored in a radix tree. This data structure can fit perfectly for storing tree type data. We use the path of nodes as their ID. In this case we can utilize the benefits of a radix tree data structure. To make it clear think of the next simple example: we have only one infrastructure node with two virtual services deployed on it. Call the node IN1 and the virtual network functions VNF1 and VNF2. In this simple case the radix tree will be shown on figure 2.4.

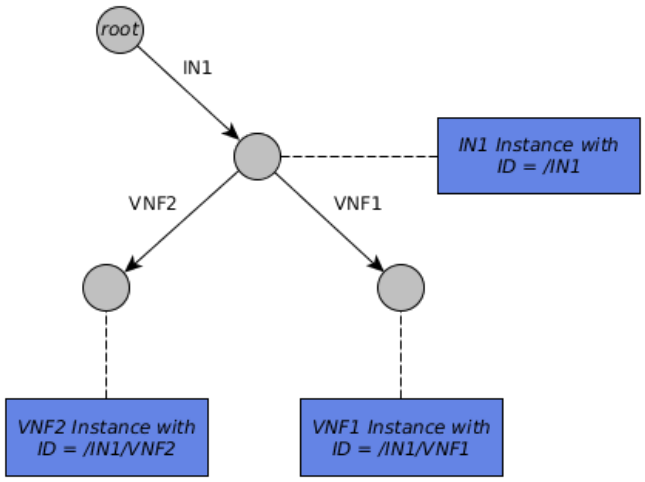

Figure 2.4: *Radix tree of the example*

If we would like to access the VNF1 element we stored we need to know its ID which is */IN1/VNF1* in this exmaple.

Beyond that we only need to parse the attributes of the infrastructure node instance like ID or its name and store it in the tree.

Virtual Newtork Function

As this way included in the previous example it is easy to notice virtual functions will be stored with the same techique as infrastructure nodes. The only differenc here is in an additional attirbute. There is the possibility to define the inside links within the virtual service. As it is only an option if this not happens the algorithm tries to handle this. If there

are multiple ports and links are not defined by hand the algorithm assumes a full mesh work inside the virtual service. Altought tries to keep the number of links on minimum so insted of creating all edges at virtual service creation, the links added dinamically. This means when we define an incoming flowentry (a port of the virtual function is a destination port of a flowentry) we also define the edges between this very port and all the others in the virtual service. If links defined by hand there is nothing much to do but we have to create those edges. As a result service chains can only use these edges. It is important to store the information about a virtal function if its edges are added automatically by the algorithm or defined directly to be able to keep up dynamic processing of links.

Port

Parsing ports both of infrastructure nodes and virtual functions is a cardinal step as the graph works on them. However beyond parsing data of ports there are two steps to highlight: one makes ground for a later approach while the other one is the interconnection of infrastructure nodes via SAPs.

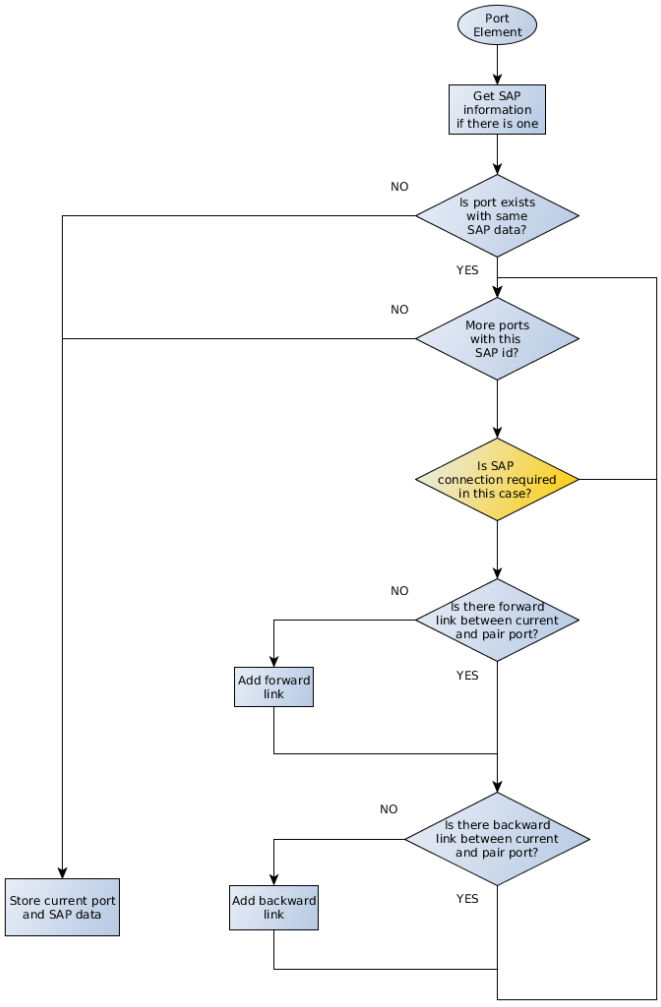

Figure 2.5: *Flowchart of applying create operation on a port element*

To be able to perform the latter one we maintain a dictionary of SAPs and when

a newone comes in check if we have to connect it somehow. The detailed procedure of this step shown on figure 2.5. We start with a parsed port element which contains SAP information. There are two information we need here: one is the SAP id which base of the connection between two SAPs. The other one is the SAP type which can be provider or consumer. The rule for these parameters is that we can only connect two SAPs if they own different type but the same SAP ID. If we follow with the figure the next condition we have to evaluate determines if there is a port added previously with the same SAP ID. In case of no we have nothing to do only store the port to the proper array with SAP ID as key. If there are already ports in the graph with the same ID we iterate through them and check if the rule is fullfilled or not (yellow condition). If no continue with the next port otherwise we apply the same steps to the individual ports of the found SAP pair. If there is no link to any of the direction defined by hand we construct this link and add it to the graph. After that we continue the iteration on SAP pairs.

At this point we also set a variable that stores the base of each port which will make sense later but now notice is is same as the port ID and it remains unchanged. This value called base port for a later created port.

Link

Parsing links is an easy operation again. We only get data from the preprocessed request and construct an edge with the proper attributes. Then create an edge between the defined ports of two infrastructure nodes. These added edges will supplement the links added based on SAP definitions.

Flowentry

The most complex element creation procedure is in the flowentry creation by far. The flowchart is on figure 2.7.

First we get the parsed data of the input flowentry which means we retrieve the start and destination port and the match action values also (detailed in 2.6). Then we check if the match value is defined or not. Here comes the essential idea of the whole algorithm. Cloning ports will play important role from now as this will help in separating service chains. The main idea behind it is when a service chain (defined through a flowentry) arrives to a port instead of connecting the flowentry into the port it was defined to we create a clone of that port along some rules detailed in a second and connect it into them. The rules are constructed a way that ensures: two different service chains will never go through the same exact port. Instead they will both have a cloned port which they hit with their paths. The first rule is in association with the match field: if we match for any tag we create a clone port or the Start node of the flowentry. See an example for port cloning on figure 2.6. In this very simple example we assume there are two infrastructure nodes IN1 and IN2 and they have the numbered ports connected in one direction. Here we would like to create a flowentry between IN1-1 and IN1-2 ports with a given match tag "A". In this case instead of adding an other edge between IN1-1 and IN1-2 we create a

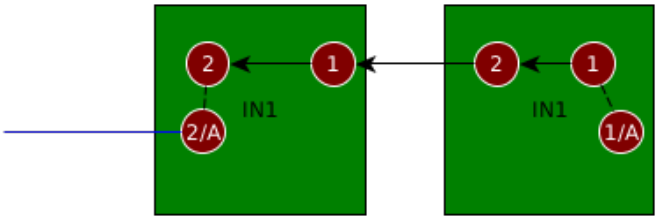

Figure 2.6: *Example for cloning a port 1.*

clone port of IN1-1 and add the match tag to its id with a separation character say "/". To keep the figure clean only the end of the ID id displayed the other can be constructed based on the element tree walking from the root to the current element. Later we expand this example as we move forward with the flowchart.

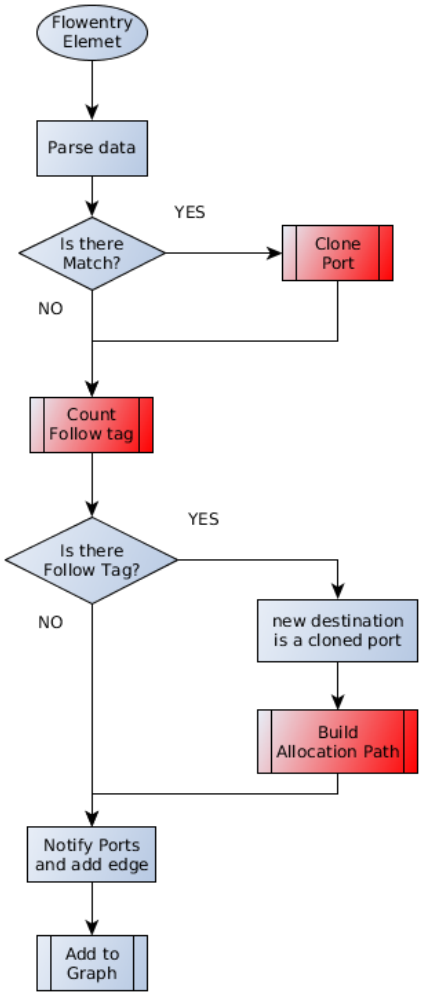

Figure 2.7: *Flowchart of applying create operation on a Flowentry element*

The next important step is depicted as count follow tag its flowchart can be found on figure F.1.1. Here some additional information required about what actions can we perform on tags. On the one hand we can push a tag to separate data from other and on the other

we can "pop tag" to concatenate it again together with all the data. The match fild filters for these tags if it is defined. In this step we need a tag we want to follow to identify the service chains. If the action is to pop the tag we the follow tag will be undefined to sign we would like to follow the default chain. If action tag is not defined within the request we avoid changing anything so follow the original tag which is in the match we filtered for. If tag is directly defined in action field we will follow this one.

If the result of the previous thought train is some defined value (Action is not pop tag) then we have to follow this tag until we can to identify the serveice chain we are working on. In this case we also clone the destination port of the flowentry to keep the rules and prevent collision of service chains. To perform this following procedure we created a recursive function that goes from port to port until some obstacle prevents it from the next jump or until it connects into a partial service chain which is a supplement part for the current one.

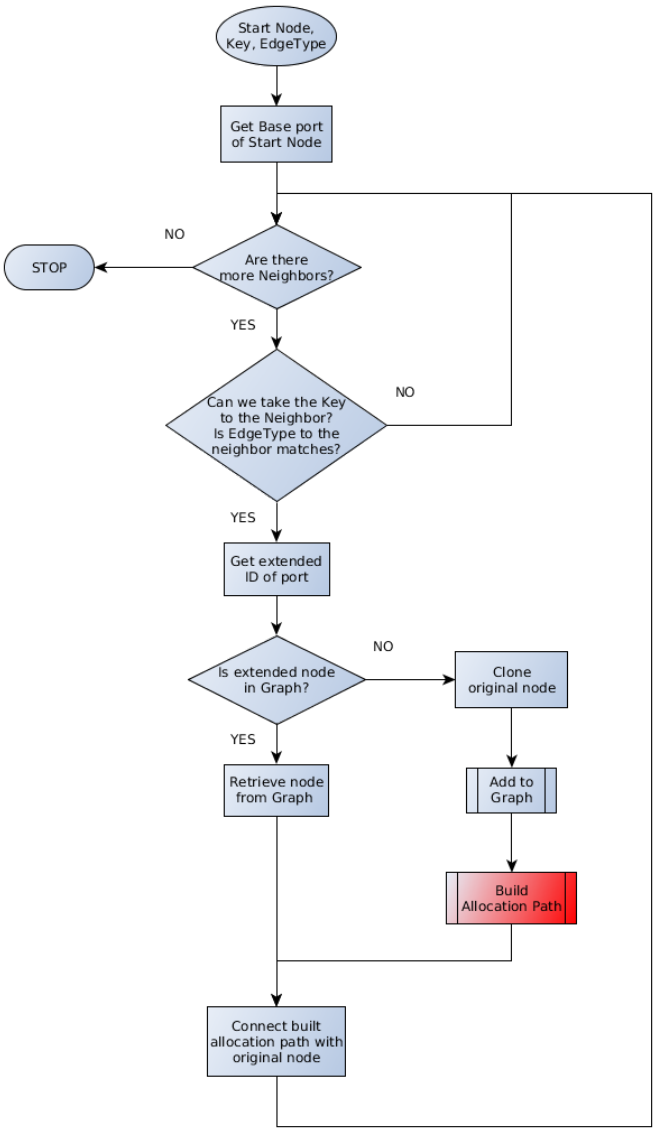

Figure 2.8: *Recursive method to build a service chain path*

The name allocaion path in this context equivalent with service chain paht. We are currently adding a flowentry to the graph while we had to clone the Start port to be able to separate the service chains and now we build this chain until we can. We determine the base port of the current cloned port which we got as an input. Now we run a depth first traversal on the graph and apply some steps along the path until some conditions evaluated ture. If follow with the flowchart we can see the start of an iteration on the neighbor nodes. If there is any of them we check the edge towards it for two contitions: edges can only carry given set of tags on them so we need to check if the current tag is among them. On the other hand it is important to avoid circles and to keep the path short we try to push out the service chain from the infrastructure nodes so it can only follow altering type of edges: once it goes through a flowentry next it can only follow a link (links are between infrastructure nodes and inside virtual functions). If any of these conditions fails we can not got towards the neighbor so we have to check other ones. If we can continue this way we calculate the extended ID of the neighbor port. This mean we guess what would be the ID if we would clone the neighbor port without actually performing the copy method. Then we check if this ID is already on the graph because in this case we can connect the current port to this negihbors cloned port which also connects this service chain into an other part what was constructed previously. This also results a cut in this branch of the search and we can countiue with other neighbors. However if the guessed ID is not appears in the graph we have to clone the neighbor and apply this methodology recursively with the new entry points (this port and the neighbor). If this recursion is done so the path is built until it was possible we only connect the currently examined node to the neighbors cloned port and continue with other neighbors.

If we go back to image 2.6 and apply this on that example after we cloned the first port we go through the following steps. Assume that the action here was undefined which means we follow the match tag what was "A". Then we have to clone the destination port which is IN1-2 So we create a node with IN1-2/A ID. Call the build allocation path recursive method and keep in mind we used a flowentry this time so next we can only go on a link. There is a neighbor of the base port which is reachable (assume the link can carry the tag "A"). Then we guess the cloned ID of the neighbor port what will be IN2-1/A. We see there is no port with this ID in the graph so we have to add it. Now we do it again recursively: we can also reach the neighbor which already has the cloned port. In this case we don't go deeper: only stepping backward we conncet the path until the original port. At this point the path is extended as mutch as it can be. Note that the dashed lines on

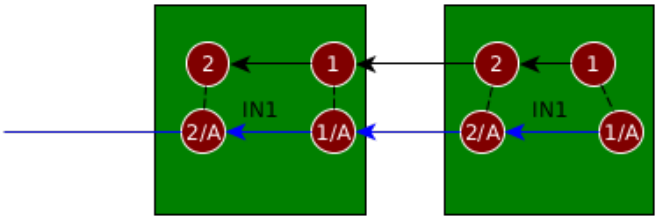

Figure 2.9: *Example for cloning a port 2.*

the figures are only logical relations these are not edges in the graph. We can now add all the network elements into the graph or store it somehow in paralell with the graph maintenance. If we follow these rules and methodologies the graph will step alter between consistent states only.

2.7.4 Delete operation

To provide the functionality of not only adding elements to graph but to delete them also we have to support an other operation. When the request contains an order to delete some elements the entry point for the algorithm remains the same: the patch function. Everything goes as in case of a create operation until the element related patch methods choose to apply delete proecdure instead of create. Althought every graph element responsible for its own removal to keep up the easily extensible worlflow.

Infrastructure node, Virtual Function

As these elements are not stored in the graph directly their removal is easy and similar at most points. If we would like to delete one of these elements for a clear remove we have to first iterate through the children of this virtual function or infrastructure node then we can safely delete the element itself. The maintained radix tree also has a good support for that method: to remove all children it is enough to iterate over the cildren in the radix tree. If we do so we can delete all children once. The only problem we have to deal with comes when we try to delete an infrastructure node which contains a virtual network function with port in it. At this point a sequence of deletes can come up where the virtual service deletes its children and itself however after that its port also notified to delete itself. To avoid this we have to examine if the element we would like to remove is since on the graph or not.

Port

Next we have to deal with is the removal of a port what has some common steps with removing a flowentry as they depend on each other.

The flowchart for removing a port appears on figure 2.10 where we can see how flowentries involved in removal of a port. The first thing we have to examine is if any flowentries are concerned about delete of the port to be concrete we have to check if the port has incoming or outgoing flowentry. If the asnwer is no we can simply remove the port itself from the graph. At this point we assume removing of a node from the graph entails the removal of its all edges which can be a link in this context. If we have a flowentry on the port we have to decide whether it is incoming or outgoing one. In both cases we remove the flowentry with the later detailed method but if the flowentry is an outgoing one we have to perform a process called dismount allocation path. This method can be thouth of as the reverse of the build allocation path function. This recursively removes the service chain while it can.

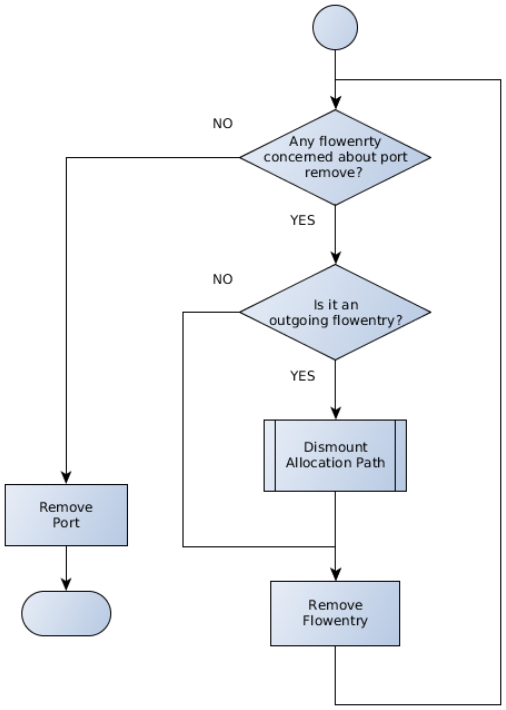

Figure 2.10: *Flowchart to remove a port from the graph*

Flowentry

As the flowentry was the most complex element to create it is the most difficult to remove from the graph. When we start the removing procedure the first what we have to examine if the indegree of the start port is 1 or more. If it is 1 we have to delete the port because this flowentry is the only reason this cloned port is in the graph. However if the source port is not added by the algorithm we can't delete it. Than we have to reverse the build allocation path procedure and dismount the built path. The exect steps of this method can be seen on figure F.1.2.

2.7.5 Remerge

At this point we have a graph algorithm that can create the graph and also delete from it to statisfy requriments defined previously. However if we examine the building graph we can see this is actually a set of disjoint graphs separated along the service chains with lots of cloned ports and their base ports which only hit by service chains if they don't have any tags. The question is how to create again a joint graph which contains only the ports were defined by the requests.

To perform this we only have to iterate ovel all of the ports in the graph and check if it is a cloned port or not. If it is we find its base port and reconnect all edges to that makred with a mach-action pair to be able to differentiate them. If we passed all links and flowentries to the base port we can simply remove the cloned port from the graph. For the visualization purposes before this remerge step we assign color to each disjoint graph in the set and color its edges.

This step ends the workflow of the algorithm where we ended up with a graph we only have to serialize with the neccessary data about it. We saw how requests from multiple source senders arrive to the algorithm input and get familiar with the graph built up from these files. The idea behind service chain identification was also described so as the remerging to a visible form of the graph. This is an important result in the project as we can now discover cross domain service chains based on only local knowledge of each domain orchestrators.

Chapter 3

Implementation

In chapter 2 we introduced the algorithm with a detailed but only theoretical approach. Now we describe what technologies and tools we used to create a working system from the theory.

The main language of the program based on python 2.7 to fit it into the parent project which also uses this version. With these borders of the system where there are two connection points to the outter world and both of them is a file we are language independent form other projects however python is a perfect language for creating prototypes as it is high level enough and has a great community support as a result almost everithing was needed to create this algoritm was already impelented in some packages.

We don't have to talk much about the radix tree package called pygtrie[3] which supports all functions we required. It provides option to store elements at every node of the tree what we can utilized. It also able to give back all children under a given node what was benefitial at infrastructure node or virtual function removal.

3.1 NewtorkX

A more complex package we used was NetworkX[2] which is a graph library to work with graphs. It supports many features what was useful through the implementation period. To see the how class hierarchy built up take figure 3.1. In this image you can disconver how the model of individual element structure applied and stored in the graph. On the right side there is the hierarchy which represents the network elements: basically there are two types of them. A set of elements mapped for nodes in the graph others will create edges instead. These all can be inherited from a common class called Graph Element. As there is no abstract class in python we use this class to notify inherited classes to override patch function by throwning a not implemented exception. Here we define ID and Name what attibutes are associated with each element in the graph. Now depends on edge or node created from an element we can inherit from the proper class. To do so we have to implement Patch function to support create and delete operations right now. We have to define what steps should be performed in case of these operations.

On the left side near this element tree we can see the other components of the system.

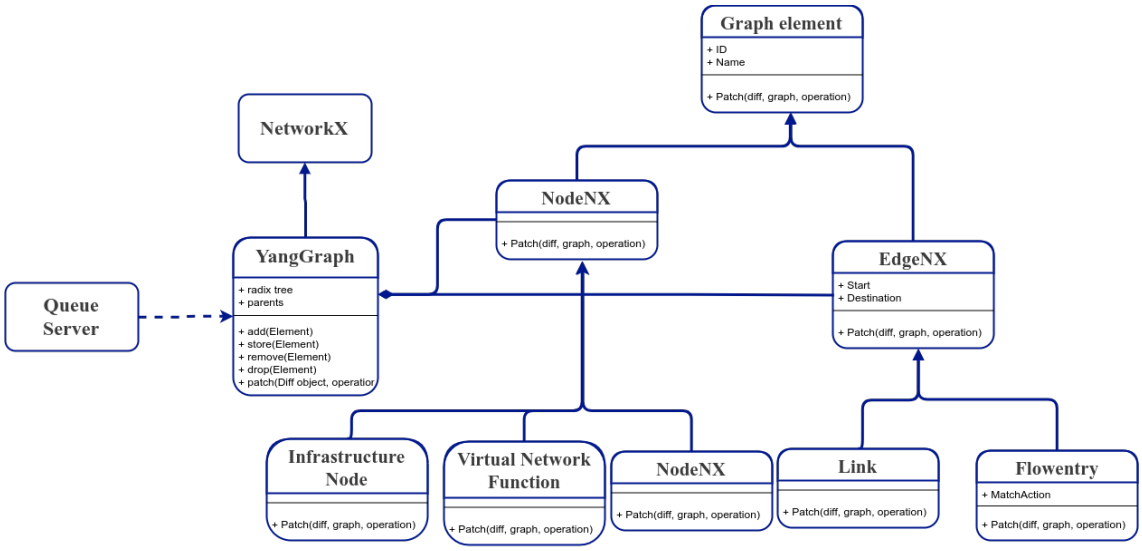

Figure 3.1: *Class diagram of python implementation*

There is the YangGraph class which coordinates the whole process on the graph. Also the graph instance itself is stored within this class so as the radix tree and the parent nodes. The latter one contains the infrastructure nodes and virtual services as they are not in the graph but we need them through the creation procedure and at the visualization also. We can see from the figure this class uses EdgeNX and NodeNX classes to store. It only cares about this difference the other details are irrelevant here. The type of the concrete instance only determined on runtime and there the proper patch function will be called. By only keeping this function as an entry point we aimed to keep the program as extensible as possible for the future element types may come in later.

We can also recognize the inheritance from the NetworkX class. In a previous version the usage this package happened through a class variable what instanciated the package class. By this approach change we achieved several benefits as we can acceess all function right in this wrapper class instead of its variable. We can use YangGraph class as a graph itself while we can override any neccessary function. We extended the methods with some others depicted on figure.

On the very left side there is the Queue Server which uses this class and keep this kind of graph up to date inside. These graphs are associated with layers as detailed before. This is how these classes work together to perform the algorithm.

3.2 CytoscapeJS

After running the algorithm we would like to evaluate the constructed graph what requires some kind of representation. The fastest way to process graph information is maybe through visual data. In this case we need a graph visualization library that can draw these special graphs. However NetworkX owns a builtin visualization based on mathplotlib it has small set of options and doesn't provide an interactive interface.

We decided to use a graph visualization tool but also implement it in a web-based

environment. The main requerements toward this visualization was an interactive operation and to support hierarchical visualization as infrastructure nodes, virtual services and ports are considered as a hierarchical tree (see figure 2.2). After trying several tools we started work with cytoscape.js[1] which fullfills all of these requirements. We just had to set some properties and generate the requred data structure and the visualization was ready and interactive.

We used the Tornado[4] python package to implement the webserver. An additional responsibility of this component is to monitor the input file with a given frequency. As the algorithm only serialize the graph into the same file again and again somehow we have to detect changes. To perform this the webserver generates a hash value of the serialized graph at every 500ms and if the hash changed there was change in the file.

The user interface has playback support also which means the changes are stored and can be replayed and analyze individually. Also the graphs are separated along the layer attribute. With these functionalities the visualization tool currently provides the following user interface.

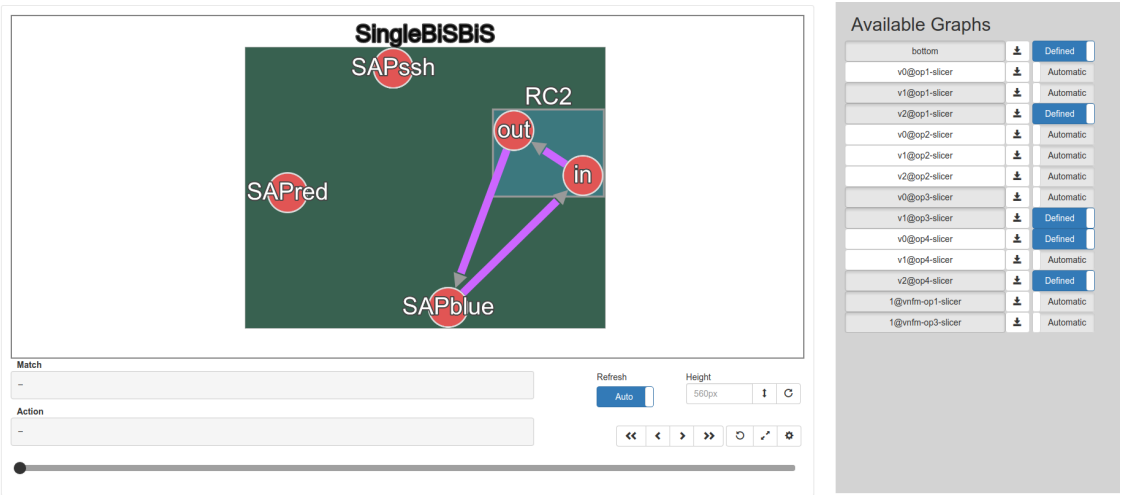

Figure 3.2: *The graphical user interface to visualize the result of algorithm*

An important benefit of cytoscape is to easili extensible with third-party layout algorithms. Also there is a way to define a layout by hand. In this setup a hybrid solution works and we can chagne between the two on the right side of each layer.

3.3 Queue Server

This server coordinates the system from recieving requests through the execution of the algtorithm to the serialization. This server was also implemented in pyton 2.7. To power up the system we have to start this server and to see the results the visualization webserver also. When this server is up and running it is ready to recieve requests at a given port and IP address. When these start to arrive the server order them by time and starts to process them. After recognizing the layer they belong to it calls the patch function of the proper graph instance. However this is currently not implemented we can expect a growth in the

performance if we apply paralell processing by layer. Other point where we can consider multi-thread approach are mentioned at the Evaluation chapter while we are searching for performance bottlenecks.

Chapter 4

Evaluation and Validation

4.1 Robotics Example

The robotics use-case mentioned in section 1.3 provides a great opportunity to test the algorithm. Now we can understand this in details, follow and identify the steps of the algorithm based on the visualized data.

There are 3 type of virtual services which will be deployed on one of the infrastructure nodes. One is to help balance the LEGO robot, and two additional to help share these virtual services: a splitter and a PID Helper virtual function. There are also two marked SAPs the SAPblue and the SAPred where the robots can access the network. The requests can come in from 4 source senders as there are 4 domains in this setup. Please note some of the steps are only included in the appendix.

If the request start to arrive from the working system components we first recieve infrastructure information which means the domains notify us about the available infrastructure nodes and the connections between them. As all of the requests arrived in association with the infrastructure we get the following state:

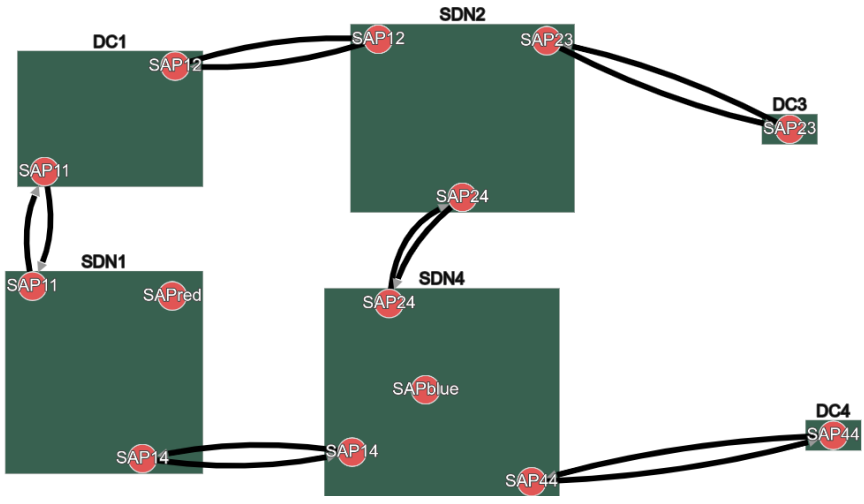

Figure 4.1: *State ot the graph after 2. request of robotics demo*

After that in the demo the first virtual was deployed on Docker Container 3. It was a PID controller with two flowentries defined: from and to the SAP23 port. Figure 4.2

depicts this step on the right side of the picture. On the next requests two flowentries were requested within DC1 which caused changes on the left side of the image.

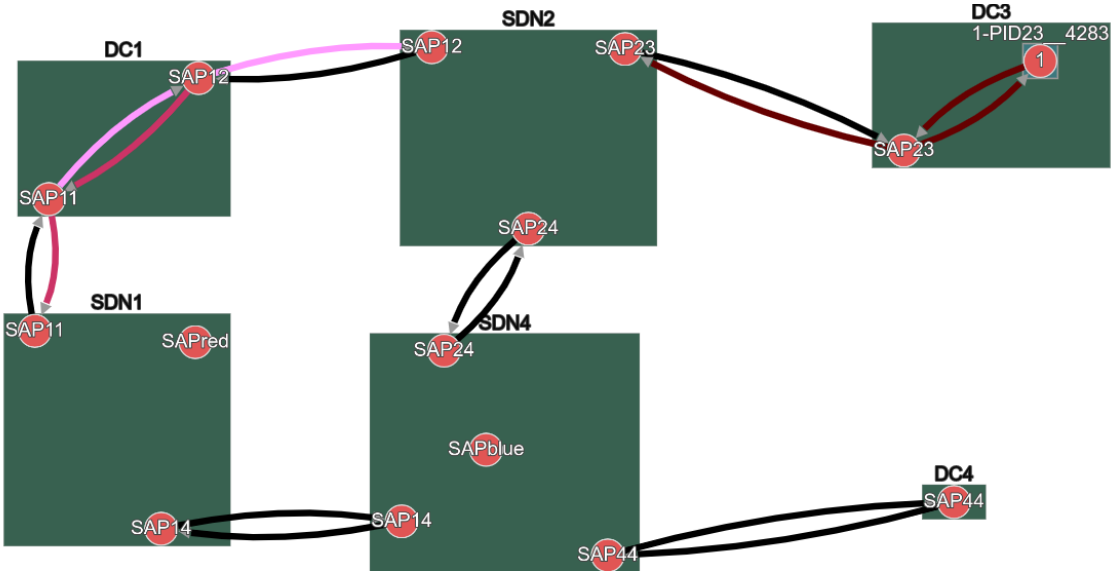

Figure 4.2: *State ot the graph after 4. request of robotics demo*

If we examine the first request of this two we can identify the steps of the algorithm: On the one hand we created a virtual service with a port inside it. Then we created the defined flowentries with red color within the DC3 node and if we remember to the flowentry creation shown on figure 2.7 we can recognize why there is an additional red edge if we only requested two flowentries. This is the result of the build allocation path recursive function because the destination port of a flowentry had available neighbors. Also important to notice that the flowentry is logically not connected into the SAP23 itself instead it ends in the clone port of SAP23. This also true for the port inside the virtual function. The figure we see is only the result of the remerge procedure.

Interesting step happened on the left side also. In the request there were two flowentries with create operation between the two ports of DC1 in the two direction. The match and action fields of these were not the same. Two thigs have to be considered here: once we can see again how the allocation path is built across the available link and however the backward link is also available the edge types must alter on the path so the recursion stops there. Second we have to notice the two different color we see. In the logical graph these two path parts are disjoint nevertheless they are connected on the drawn graph. Because the match-action fields are different they will connect to diversified cloned ports. After coloring these with two colors after remerge we see the expected result.

On figure 4.3 we can again see the result of two requests. In the first one we created two flowentries within SDN2 between SAP12 and SAP23. As before, the building of allocation path drilled deeper. Not stopped because of altering edges conditoin but because it found the cloned port and only connected the current one into it. Thanks to the similarity of match and action pairs now the two partial path connected together which means in the logical graph they are not disjoint anymore. The other is easy to see: in SDN1 we requested

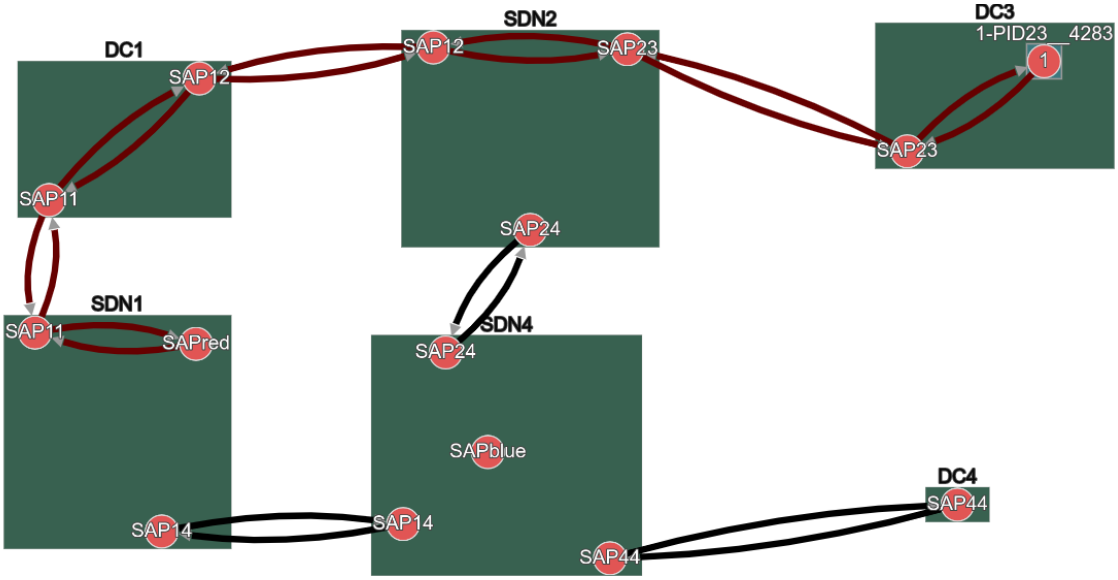

Figure 4.3: *State ot the graph after 6. request of robotics demo*

for two flowentries between SAP11 and SAPred to connect the service chain to the access point where the robot can access it. Now we got one PID deployed and connected with a service chain.

On the next step we repeat this but in different order. First we create flowentries in SDN4 between SAPblue and SAP44. They will appear with different colors as the match and action pairs differ. We can also see the build allocation path working. Than we deploy the second virtual function within DC4 node and connect it with flowentries to the proper port. The aggregated result of this two steps will be two completly separated but one-by-one joint allocations. That state shown on figure 4.4.

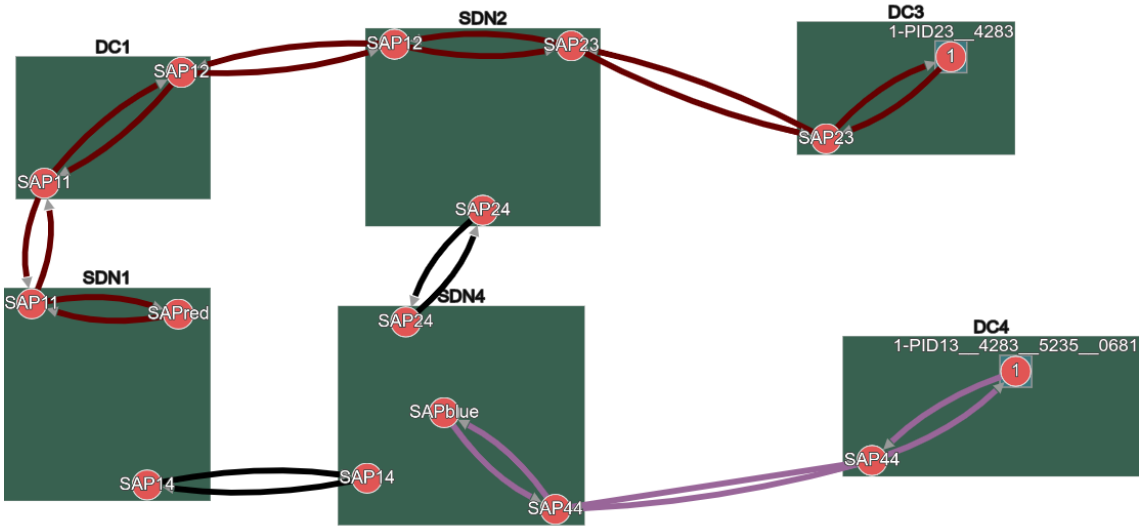

Figure 4.4: *State ot the graph after 8. request of robotics demo*

Here comes the interesting part: we start an other virtual service within the DC3 however we want to separate it from the previously built service chain. First of all we create

a service on the proper node and connect it with the other service with two flowentries. We use match and acition fields to separate the service chain when we connect it to the SAP23 port. To achieve this we have to push an unused tag to the flowentry right before we connect it to the common port. Also we have to filter with the match tag for the flowenty coming from the common port to the service. The result of these steps shown on figure 4.5. Here we can also see how the recursive function added an extra edge. Note here the

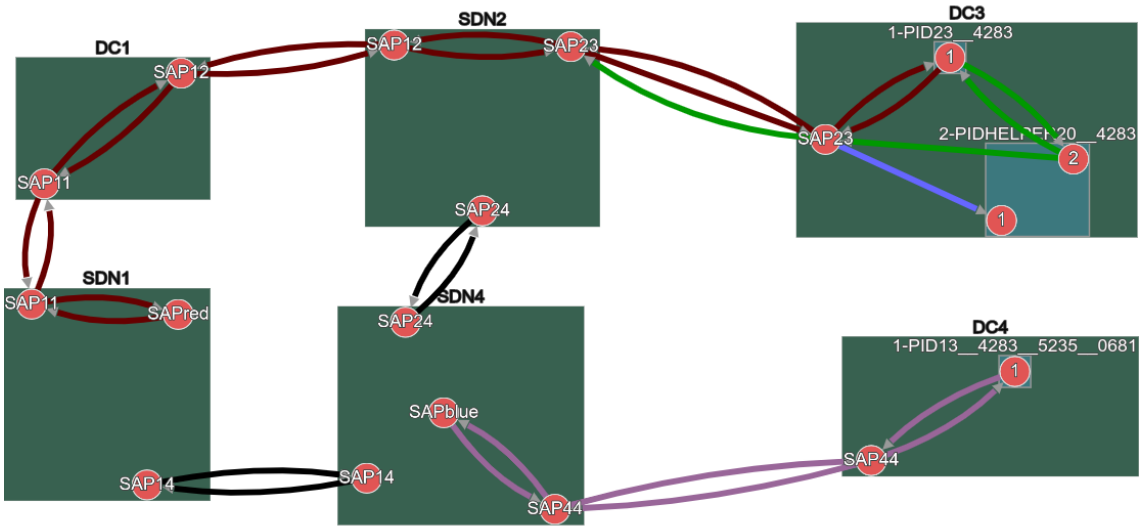

Figure 4.5: *State ot the graph after 9. request of robotics demo*

three colors: one is the original red stay unchanged. Second is the green which represents the outgoing data from the virtual service and the important result is the separation of these chains on the link between SDN2 and DC3. The next virtual service we would like to start is a splitter funciton which will be deployed on SDN4. To complete the service chain we create two flowentries in SDN2 from SAP23 to SAP24 where we can continue the path to SDN4. The build allocation path method will extend it with two additional edges on the two concerned links. Now between DC3 and SDN2 there is a service chain with two directions colored with red and two other disjoint partial service chains with one and riverse directional edges.

Now we deploy the splitter and an other PID Helper near the other PID Controller and we also define flowentries to connect them into a service chain. The splitter sheperds the data into two directions. Here we can also recognize the smart link addition inside the virtual service. While in PID Helpers it was defined by hand there is no link within the virtual function here the links added automatically as soon as it is neccessary. We can also notice there are no unwanted links between the ports where is no incoming flowentry. The end of these steps can be seen on figure 4.6. What is important to see here is that there are three service chains marked with greed purple and red. Sometimes these share a common link and goes throuh the same port on this visualized graph. Thanks to the algorithm we can separate these service chains and differentiate with various colors. To perform this we only use requests from local actors with limited knowledge on the whole system. This

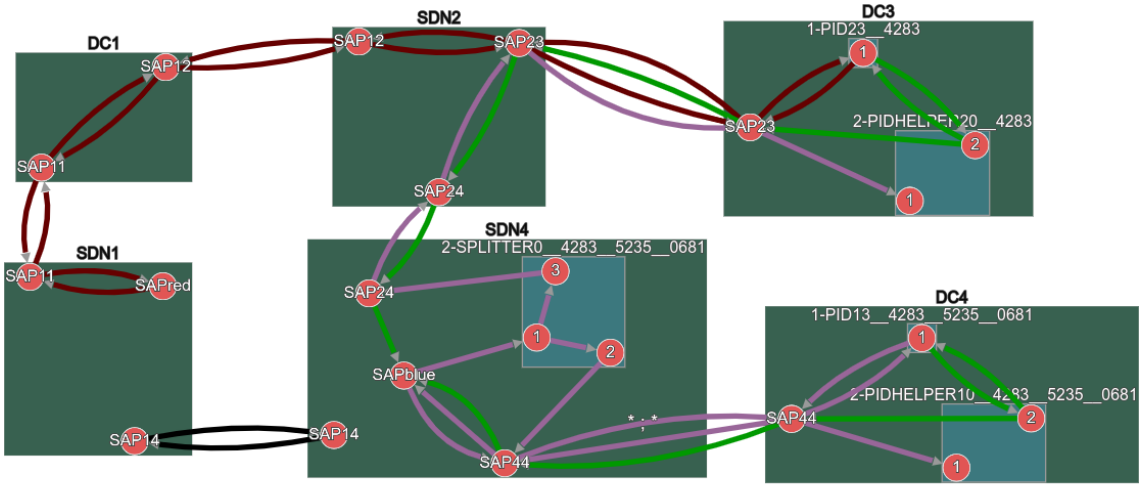

Figure 4.6: *State ot the graph after 12. request of robotics demo*

also means we can use the system as part of a distributed ecosystem where not only one information source presented.

To validate the work of delete operation take a request which removes the first PID Helper from the graph. We can see not only the virtual service disappeared itself but the flowentries connected to it also. In addition we can see the reverse of building an allocation path: the recursive dismount method which clears up the service chain over the link where it was defined by hand so it can't be removed further.

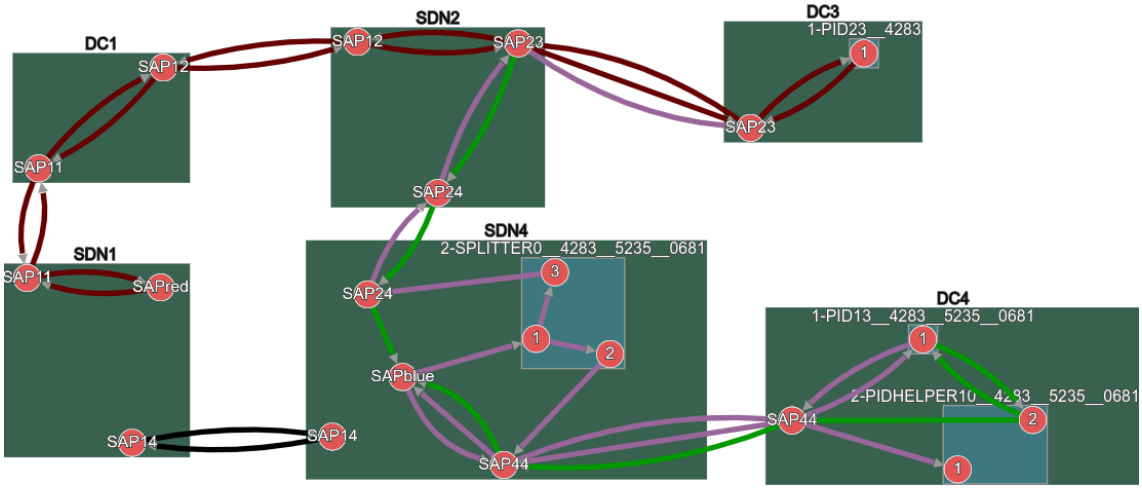

Figure 4.7: *State ot the graph after 13. request of robotics demo*

By these examples we covered the requirements towards the algorithm an as it was shown all of them is fullfilled.

4.2 cProfile

While the algorithm passed the functional requiremets it is important to analyze the methodology by performance. cProfile is a profiling tool for pyhton. It measures times and counts function calls within a script. We made these tests at every notification and to choose the most relevant we show the one that last longer than others.

In this case we can get the following values:

Number of calls	Total Time	Cummulated Time	Percall	Function
1012913	0.680	2.810	0.001	Python deepcopy
35395	0.235	2.785	0.021	Python deepcopy reconstruct
1	0.000	2.625	2.625	Visual serialization
1	0.019	1.413	1.413	Remerge

These are the top few lines of the created cProfile log. In the first and second row we can see the program calls the pythons builtin deepcopy function more than a million times and however it runs only 0.001 second as an avarage it costs 2.810 second to perform this amount of copy. On the third row we can see the longest own function is when we serialize the data also the remerge of disjoint ports lasts long. It is worthy to consider to transform these methods to use multiple threads as they can be altered with relative easily.

If we would like to evaluate the performance we can say there are some functions that lasts quite long compared to others. However these methods' call time is in order of second if we take in account the incoming frequency of requests we can still call the algorithm online.

Chapter 5

Conclusion

In this paper we introduced the desing procedure of an algorithm that ia capable to identify service chains in a distributed orchestration environment. An other important benefit of this new approach lies in the scope of knowledge what each domain orchestrator handles. They only familiar with the related part of the infrastructure nevertheless we can identify service chains within a global range. However at this point it supports all operations those are essential to fit into the workflow we succesfully kept the exntesibility on a high level.

We also have to keep in mind the performance measurements as this algorithm should work online. According to the results of section 4.2 we know what methods should be reconstructed to decrease runtime projected on one request processing.

Bibliography

[1] Reference page of cytoscape.js. `http://js.cytoscape.org/`. 2017.10.27.

[2] Reference page of networkx. `https://networkx.github.io/index.html`. 2017.10.27.

[3] Reference page of pygtrie. `http://pygtrie.readthedocs.io/en/latest/`. 2017.10.27.

[4] Reference page of tornado web server. `http://www.tornadoweb.org/en/stable/`. 2017.10.27.

[5] Maysam Mirahmadi Hassan Hawilo, Abdallah Shami. NFV: state of the art, challenges, and implementation in next generation mobile networks (vEPC). *IEEE Network*, 28(6):18–26, Nov-Dec 2014.

Appendix

F.1 Additional images

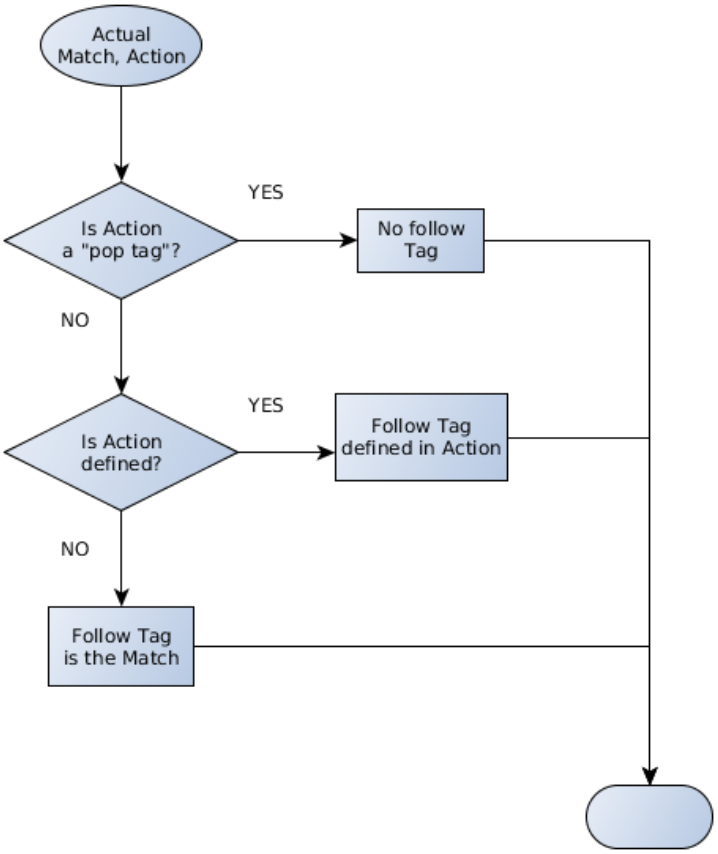

Figure F.1.1: *Flowchart of counting the follow tag.*

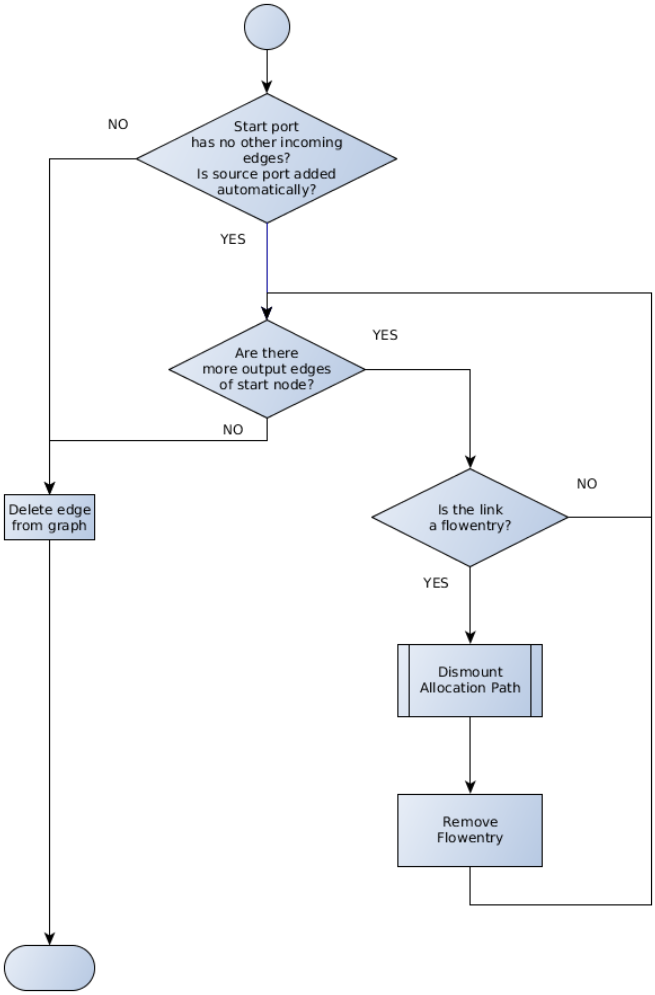

Figure F.1.2: *Flowchart of applying delete operation on a flowentry element*

Figure F.1.3: *State ot the graph after 1. request of robotics demo*

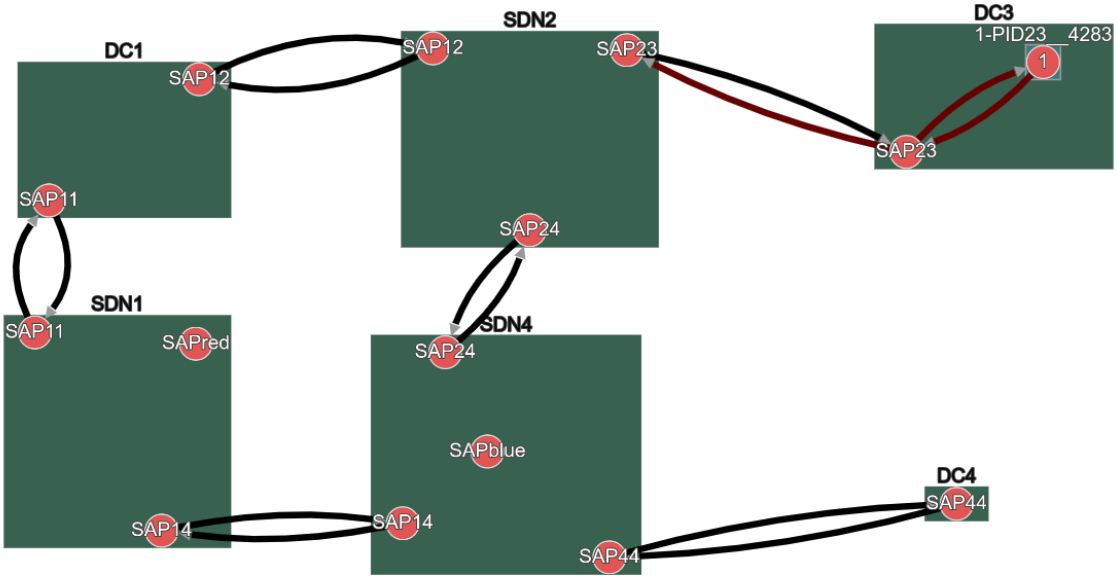

Figure F.1.4: *State ot the graph after 3. request of robotics demo*

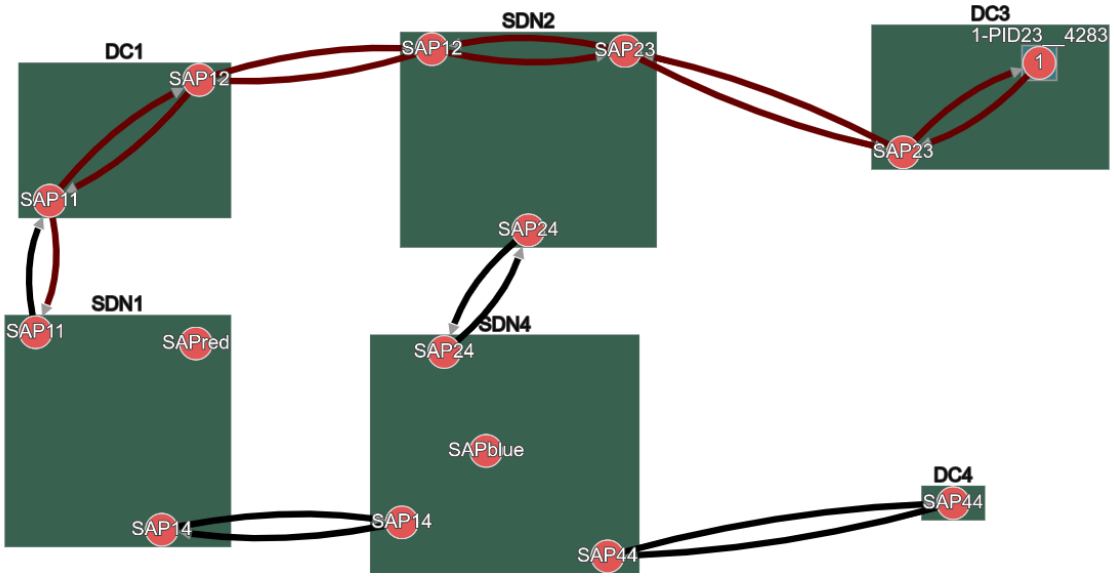

Figure F.1.5: *State ot the graph after 5. request of robotics demo*

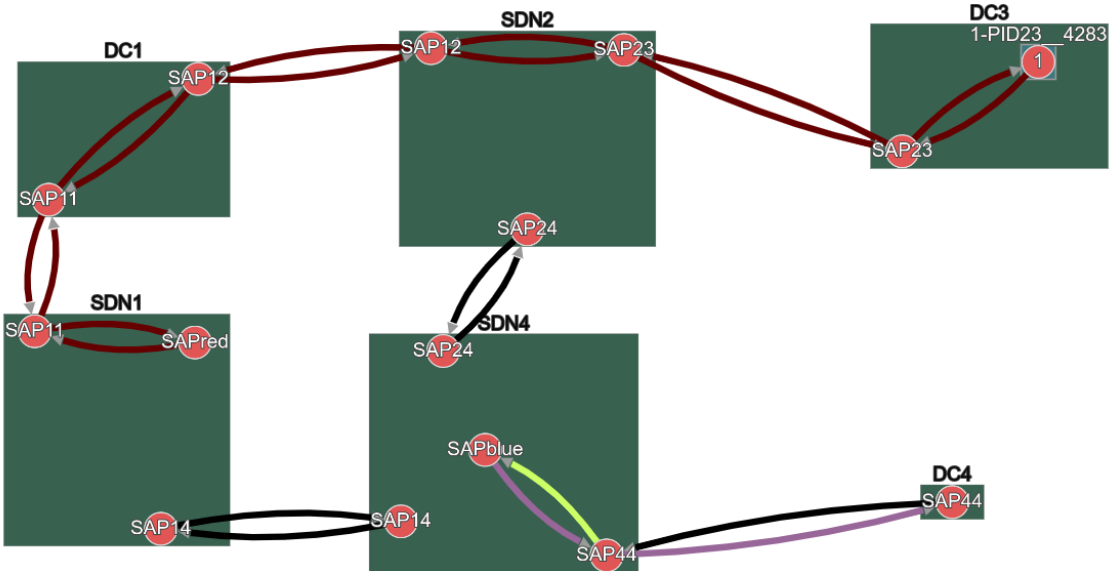

Figure F.1.6: *State ot the graph after 7. request of robotics demo*

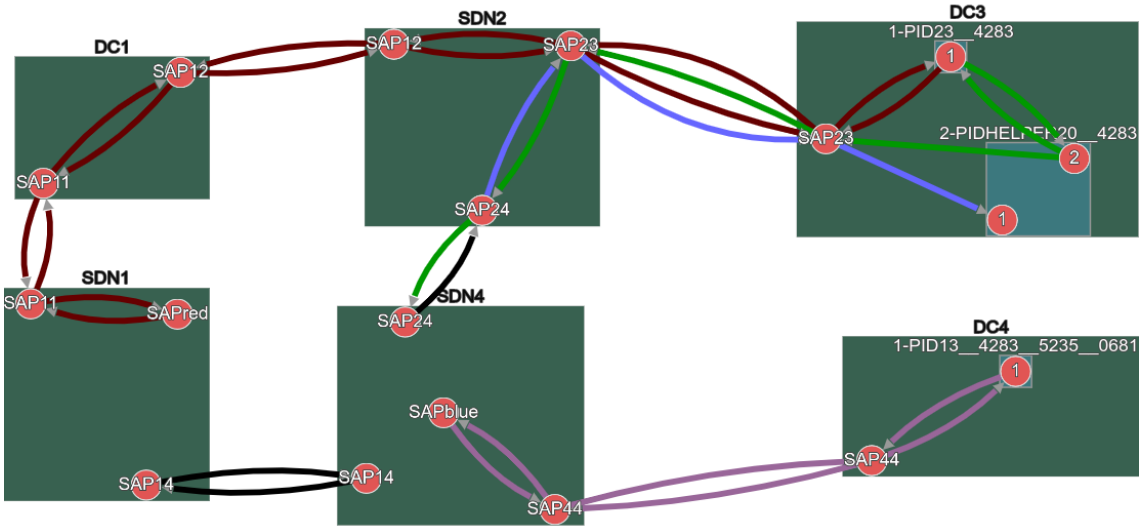

Figure F.1.7: *State ot the graph after 10. request of robotics demo*

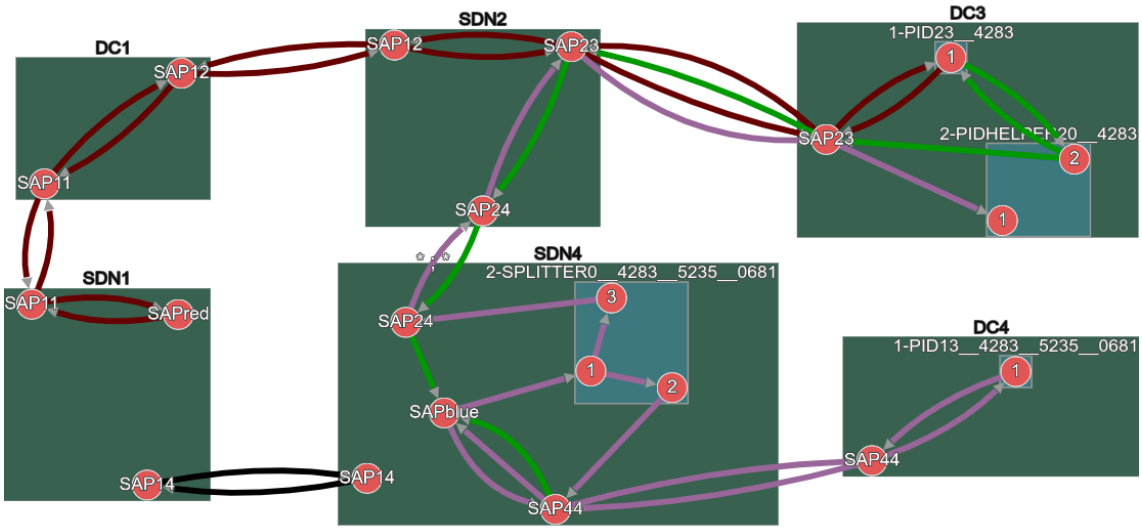

Figure F.1.8: *State ot the graph after 11. request of robotics demo*