

**JADAVPUR UNIVERSITY**

**Design and Analysis of SDN, NFV, and NV Topologies  
Using Neo4J Graph Database**

By

**Adwitiya Ghosh**

Master of Computer Application - III

Class Roll No.: 002010503040

Registration No.: 154248 of 2020-2021

Examination Roll No.: MCA2360019

Under the supervision of

**Prof. Chandan Mazumdar**

Project submitted in partial fulfilment for the

Degree of Master of Computer Application

In the

Department of Computer Science & Engineering

FACULTY OF ENGINEERING AND TECHNOLOGY

2023

**FACULTY OF ENGINEERING AND TECHNOLOGY**  
**JADAVPUR UNIVERSITY**

To Whom It May Concern

I hereby recommend that the project entitled “**Design and Analysis of SDN, NFV, and NV Topologies Using Neo4J Graph Database**” has been carried out by Adwitiya Ghosh (Reg. No.:154248 of 2020-21, Class Roll: 002010503040, Examination Roll No: MCA2360019) under my guidance and supervision and be accepted in partial fulfilment of the requirement for the degree of MASTER of COMPUTER APPLICATION in DEPARTMENT of COMPUTER SCIENCE and ENGINEERING, JADAVPUR UNIVERSITY during the academic year 2022-23.



(Prof. Chandan Mazumdar)

Project Supervisor

Dept. of Computer Science and Engineering

Jadavpur University

Jadavpur University, Kolkata-700032



Prof. (Dr.) Nandini Mukherjee

Head of the Department

Dept. of Computer Science & Engineering

Jadavpur University, Kolkata-700032



Prof. Ardhendu Ghoshal

Dean, Faculty of Engineering & Technology

Jadavpur University, Kolkata-700032

### **Statement of Originality**

I, Adwitiya Ghosh, student of Master of Computer Application of Jadavpur University of the Academic Year 2020-23 hereby declare that this thesis entitled **Design and Analysis of SDN, NFV, and NV Topologies Using Neo4J Graph Database** under the guidance of Prof. Chandan Mazumdar contains technology survey and original research work done by the undersigned candidate as part of fulfilment of the degree.

All information in this thesis have been obtained and presented in accordance with existing academic rules and ethical conduct. I declare that, as required by these rules and conduct, I have fully cited and referred all materials and results that are not original to this work.

I also declare that I have checked this thesis as per the “Policy on Anti-Plagiarism, Jadavpur University, 2019.”

Signature of the candidate

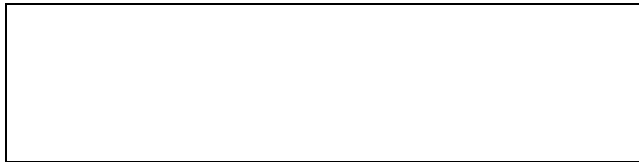
Date:

Name: Adwitiya Ghosh

Class Roll No: 002010503040

Examination Roll No: MCA2360019

Registration No: 154248 of 2020-21



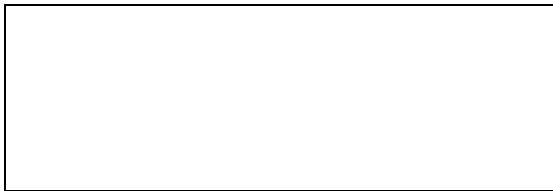
Signature of Supervisor

Prof. Chandan Mazumdar

Dept. of CSE, JU

## **CERTIFICATE OF APPROVAL**

This is to certify that the project entitled “**Design and Analysis of SDN, NFV, and NV Topologies Using Neo4J Graph Database**” is a bonafide record of work carried out by ADWITIYA GHOSH in fulfilment of the requirements for the award of the degree of Master of Computer Application in the Department of Computer Science and Engineering, Jadavpur University. It is understood that by this approval the undersigned do not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn therein but approve the thesis only for the purpose for which it has been submitted.



**Signature of Examiner 1**

**(With Date)**



**Signature of Examiner 2**

**(With Date)**

## **DECLARATION OF ORIGINALITY AND COMPLIANCE OF ACADEMIC PROJECT**

This is to certify that the work in the project entitled **Design and Analysis of SDN, NFV, and NV Topologies Using Neo4J Graph Database** submitted by Adwitiya Ghosh, is a record of an original research work carried out by him under the supervision and guidance of Prof. Chandan Mazumdar for the award of the degree of Master of Computer Application in the Department of Computer Science and Engineering, Jadavpur University, Kolkata-32. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Signature of the candidate

Date:

Name: Adwitiya Ghosh

Class Roll No: 002010503040

Examination Roll No: MCA2360019

Registration No: 154248 of 2020-21

## **ACKNOWLEDGEMENT**

The satisfaction and euphoria that accompanies the successful completion of this task would be incomplete without the mention of the people who made it possible. Their constant guidance and encouragement crowned my effort with success.

It is a great pleasure to express my sincerest thanks to my project supervisor Prof. Chandan Mazumdar, Professor, Department of Computer Science and Engineering, Faculty of Engineering and Technology, Jadavpur University, for his encouragement, valuable suggestion, and constant support during the course of this project. I would like to thank all the professors of the Department of Computer Science and Engineering, Jadavpur University, Kolkata for the guidance they provided me throughout the duration of the Master of Computer Application course.

A special note of thanks goes to Dr. Nandini Mukherjee, Head, Department of Computer Science and Engineering, Jadavpur University. I am also thankful to the Dean, Faculty of Engineering and Technology, for providing an excellent environment for completion of this project.

I am also indebted to Dr. Anirban Sengupta, Mr. Agniswar Chakraborty, Ms. Srijita Basu and all the associated persons of CDC JU for their seamless cooperation and help in completion of this project. I am thankful to my fellow classmates and my family for constant help and support.

Signature of the candidate

Date:

Name: Adwitiya Ghosh

Class Roll No: 002010503040

Examination Roll No.: MCA2360019

Registration No: 154248 of 2020-21

# Contents

List of Abbreviations .....	9
List of Figures .....	10
List of Tables .....	12
Abstract.....	13
Chapter 1: Introduction .....	14
1.1. Overview .....	14
1.1.1. Modelling .....	14
1.1.2. Querying.....	15
1.1.3. Analyzing .....	15
1.2. Virtualization .....	16
1.3. Advantages of Virtualization .....	17
1.4. Limitations of Virtualization.....	18
1.5. Types of Virtualization .....	19
1.6. Virtualization of Network Infrastructure .....	21
1.6.1. Key Components of Network Infrastructure Virtualization: .....	21
1.6.2. Benefits of Network Infrastructure Virtualization: .....	22
1.7. Distributed Computing Vs Centralized Computing .....	23
Chapter 2: Technology Survey .....	25
2.1. Software Defined Networking (SDN) .....	25
2.1.1. Differences between Traditional networking and SDN .....	25
2.1.2. SDN Architecture.....	26
2.1.3. The Controller .....	30
2.1.4. OpenFlow Protocol .....	31
2.2. Network Function Virtualization .....	35
2.2.1. Virtual Network Functions .....	36
2.2.2. Virtual Network Function – Chain Placement Problem .....	37
2.3. Network Virtualization .....	45
2.3.1. Key Components of Network Virtualization: .....	45
2.3.2. Benefits of Network Virtualization: .....	45
2.3.3. The Industry Perspective of NV: .....	46

2.4. Graph Database .....	56
2.4.1. Graph Database and Graph Theory .....	56
2.4.2. Using Graph Database For Handling Network Infrastructure.....	57
2.4.3. Difference with Relational Model .....	58
2.4.4. Conversion of Relational Database to Graph Database.....	60
2.5. Path Analysis .....	65
2.6. Cost Analysis .....	66
Chapter 3: SDN Topology Using Neo4J .....	69
3.1. Modelling .....	69
3.2. Visualization of Network Model .....	76
3.3. Updating.....	78
3.4. Deleting .....	79
Chapter 4: NFV Topology Using Neo4J .....	81
4.1. Modelling .....	81
4.2. Visualization of Network Elements: .....	86
4.3. Creating Links: .....	87
4.4. Updating:.....	90
4.5. Deleting .....	91
Chapter 5: NV Topology Using Neo4J .....	93
5.1. Modelling .....	93
5.2. Creating Links: .....	97
5.3. Visualization of Network: .....	99
5.4. Updating.....	100
5.5. Deleting .....	101
Chapter 6: Path Analysis Using Neo4J.....	103
Chapter 7: Cost Analysis Using Neo4J.....	105
7.1. Calculation .....	105
7.2. Analysis .....	107
Chapter 8: Conclusion and Future Scope.....	110
References .....	111



# List of Abbreviations

ACRONYM	FULL FORM
SDN	SOFTWARE DEFINED NETWORK(ING)
NFV	NETWORK FUNCTION VIRTUALIZATION
NV	NETWORK VIRTUALIZATION
NVI	NETWORK VIRTUALIZATION INSTANCE
VNI	VIRTUAL NETWORK INFRASTRUCTURE
VM	VIRTUAL MACHINE
VN	VIRTUAL NETWORK
VNF	VIRTUAL NEYWORK FUNCTION
VNF - CPP	VIRTUAL NETWORK FUNCTION – CHAIN PLACEMENT PROBLEM
RTR	ROUTER
NIC	NETWORK INTERFACE CARD
OVS	OPEN vSWITCH

# List of Figures

Figure 1: Virtualization.....	17
Figure 2: Types of Virtualization.....	21
Figure 3: The Key Components of Network Virtualization .....	22
Figure 4: Differentiation of Centralized and Distributed Computing.....	24
Figure 5: Differences between SDN and Traditional Network .....	26
Figure 6: SDN Architecture .....	28
Figure 7: Northbound API and Southbound API.....	29
Figure 8: The Controller .....	31
Figure 9: Packet-In and Packet-Out (OpenFlow) .....	32
Figure 10: Detailed Flow Chart of OpenFlow Protocol.....	34
Figure 11: NFV Infrastructure .....	36
Figure 12: A VNF Chain Infrastructure.....	38
Figure 13: A Feasible Projection (VNF-CPP) .....	40
Figure 14: Post Matrix Based Approach (VNF-CPP).....	43
Figure 15: Multiple Stage (VNF-CPP) .....	44
Figure 16: NIC Virtualization.....	47
Figure 17: Router Virtualization .....	49
Figure 18: Virtual Private Network(VPN).....	53
Figure 19: Virtual Sharing Network (VSN).....	54
Figure 20: VPN extends VSN.....	55
Figure 21: Neo4J Graph Database .....	60
Figure 22: Virtual machine in a Graph Database.....	63
Figure 23: Virtual network in a Graph Database .....	63
Figure 24: Network Performances .....	68
Figure 25: Importation of Configuration Files for SDN in the tool.....	69
Figure 26: A SDN Topology in the tool. ....	77
Figure 27: Updating a switch node in SDN in the tool.....	78
Figure 28: Deletion of a host node in SDN in the tool. ....	80
Figure 29: Importing configuration files for NFV Topology in the tool .....	81
Figure 30: Network Elements in a NFV Topology.....	87
Figure 31: Linking between nodes in a NFV .....	89
Figure 32: Link Visualization in NFV. ....	89
Figure 33: Updating a node in NFV Topology in the tool.....	91
Figure 34: Deletion of Nodes in NFV Topology in the tool.....	92
Figure 35: Importing Configuration files for NV Topology in the tool .....	93
Figure 36: Linking between nodes in a NV topology.....	98
Figure 37: Visualization of NV Topology in the tool.....	100
Figure 38: Updating node properties in a NV Topology .....	101
Figure 39: Deletion of a node in a NV Topology .....	102
Figure 40: The Network on which we will do Path Analysis.....	104
Figure 41: Path Details.....	104

Figure 42: The Shortest Path.....	104
Figure 43: Taking Link Properties.....	106
Figure 44: Calculation of Cost.....	106
Figure 45: Modified Weightage Vs Cost Graph.....	107
Figure 46: Bandwidth Vs Cost Graph.....	108
Figure 47: Latency Vs Cost Graph .....	109

# List of Tables

Table 1: Cost Measurement by giving weightage to bandwidth, jitter and latency.....	107
Table 2: Cost and Bandwidth.....	108
Table 3: Cost and Latency.....	108

# Abstract

Software-Defined Networking (SDN), Network Function Virtualization (NFV), and Network Virtualization (NV) technologies have revolutionized the networking landscape by providing flexible, scalable, and programmable network infrastructures. Efficiently managing and analyzing these complex networks require advanced tools that can model, query, and analyze the paths and associated costs within the network.

This thesis presents the development of an SDN-NFV-NV tool implemented in Django, a popular Python web framework, incorporating Neo4j, a graph database, for efficient modelling, querying, and path analysis. The tool aims to provide network administrators and researchers with a comprehensive platform to model SDN, NFV, and NV topologies, perform complex queries, and analyze paths and associated costs. The querying capabilities of the tool provide users with a flexible interface to define and execute queries against the SDN-NFV-NV network model. Users can specify search criteria based on various network attributes and traverse the graph to find desired network paths. Path analysis plays a crucial role in understanding network behavior and optimizing resource allocation. The tool incorporates shortest path algorithm, leveraging the graph database Neo4j's graph traversal capabilities, to efficiently find the shortest path and associated costs between network elements. Users can visualize and analyze these paths to gain insights into network performance and identify potential bottlenecks. The implementation of the SDN-NFV-NV tool demonstrates its effectiveness in managing and analyzing complex network infrastructures. Through Django and Neo4j integration, it provides a user-friendly interface for modelling, querying, and path analysis. The tool's modular architecture allows for future enhancements and integration with other network management frameworks.

*Keywords:* Django, Graph Database, Neo4j, NFV, NV, Path-Analysis, Python, SDN Virtualization.

# Chapter 1: Introduction

## 1.1. Overview

The project is divided into three major parts- modelling, querying, and analyzing. We discuss them in details.

### 1.1.1. Modelling

When modeling a network topology mathematically, we aim to represent the structure and connectivity of a network using mathematical concepts and equations. Network topology refers to the arrangement of nodes (devices) and links (connections) in a network.

Mathematically, a network topology can be represented using graph theory, which provides a mathematical framework for analyzing and describing the relationships between nodes in a network. In graph theory, a network can be represented as a graph, where the nodes are represented by vertices, and the links between nodes are represented by edges.

Let's consider a network with  $n$  nodes. Mathematically, we can represent this network topology as a graph  $G$ , defined as:

$$G = (V, E)$$

Where:

- $V$  represents the set of vertices or nodes in the network. Each vertex  $v_i \in V$  represents a specific node in the network, where  $i$  ranges from 1 to  $n$ .
- $E$  represents the set of edges or links between nodes. Each edge  $e_j \in E$  represents a connection between two nodes in the network, where  $j$  ranges from 1 to  $m$ .

To model the network topology mathematically, we need to define the connectivity between nodes, which can be represented using an adjacency matrix or an adjacency list.

Adjacency Matrix:

An adjacency matrix is a square matrix that represents the connections between nodes in a network. Let's denote the adjacency matrix as  $A$ , where  $A[i][j] = 1$  if there is a link between nodes  $i$  and  $j$ , and  $A[i][j] = 0$  otherwise. Mathematically, the adjacency matrix  $A$  can be defined as:

$$A = [a_{ij}]$$

Where  $a_{ij} = 1$  if there is a link between nodes  $i$  and  $j$ , and  $a_{ij} = 0$  otherwise.

Adjacency List:

An adjacency list is a collection of lists that represents the connections of each node in the network. Each element in the list represents a node and contains information about its neighboring nodes or the nodes it is connected to. Mathematically, the adjacency list can be defined as:

$$L = [I_i]$$

Where  $I_i$  represents the list of nodes that node  $i$  is connected to.

### 1.1.2. Querying

CRUD- Create, Read, Update, Delete. Querying a network topology graph database involves retrieving specific information or patterns from the stored network topology data using graph query languages or graph query APIs. Here, we use Cypher query language. Graph databases are designed to store and manage interconnected data, making them well-suited for representing and querying network topologies.

Data Modeling: Before querying, the network topology data needs to be appropriately modeled in the graph database. Nodes represent network devices or components, and edges represent the connections between them. Attributes and properties can be associated with nodes and edges to capture additional information about the network elements.

Graph Query Language: Graph databases often provide their own query language to interact with the data. We are using Cypher for Neo4j in this project.

Querying Node Properties: The network elements, which are entities (nodes) in the database, have several properties like IP Address, MAC Address, etc. We aim to update/delete the properties or/and the nodes and relationships.

### 1.1.3. Analyzing

Once the network topology is mathematically modeled using either an adjacency matrix or an adjacency list, we can apply various graph theory algorithms and techniques to analyze and comprehend the network's properties and behaviors. These analyses may include determining the shortest path between nodes, identifying central nodes, detecting network bottlenecks, and evaluating network performance metrics.

In summary, modeling a network topology mathematically involves representing the network structure and connectivity using graph theory concepts, such as vertices (nodes) and edges (links). This mathematical representation enables the analysis and exploration of various properties and behaviors of the network, supporting network design, optimization, and troubleshooting.

## 1.2. Virtualization

If we try to remember when we first came across the word 'virtual' in our life, we will be pegging away in getting an answer. Those who are far away from 'digital generation' may say that in their high school, while they came across 'Optics', they learnt the differences between real image and virtual image. A real image is formed by actual intersection of light rays, whereas a virtual image is formed by imaginary intersection of light rays. A real image can be formed in screen but a virtual image can only be seen in mirror. The difference between something that is unreal and something that is virtual can be subtle and context-dependent. Unreal refers to something that is fake or not real. It can describe something that is imagined, illusory, or deceptive. For example, an unreal creature might be a mythical beast that does not exist in the physical world. Unreal can also describe something that is not genuine or authentic. Virtual can refer to something that is not physically present but is simulated or represented in a digital environment. It can describe a computer-generated simulation of a real-world object or environment. For example, a virtual reality headset might allow a user to experience a simulated environment that feels real but is not physically present. Virtual can also describe something that is in essence or effect but not in fact or reality.

Virtualization is a method that allows the separation of a service from its physical infrastructure. It entails creating a virtual representation of computer hardware or other computing resources using specialized software. Originally developed during the mainframe era, virtualization enables the use of virtual or software-based versions of computing resources instead of their physical counterparts. By employing virtualization, it becomes possible to run multiple operating systems and applications simultaneously on a single machine, utilizing the hardware more efficiently and enhancing its flexibility. Virtualization leverages software to establish an abstraction layer that sits between computer hardware and virtual machines (VMs). [1] By utilizing this technique, the various hardware components of a single computer, such as processors, memory, and storage, can be partitioned into multiple VMs. Each VM operates with its own independent operating system (OS) and functions as a separate computer entity, despite utilizing only a portion of the physical hardware resources. In essence, virtualization enables the efficient sharing and allocation of computer resources, allowing multiple VMs to coexist on a single physical machine while maintaining isolation and independence. In virtualization, a software layer called a hypervisor or virtual machine monitor (VMM) [2] creates and manages the virtual environment. The hypervisor abstracts the underlying hardware and provides an interface for the virtual machines (VMs) to interact with it.



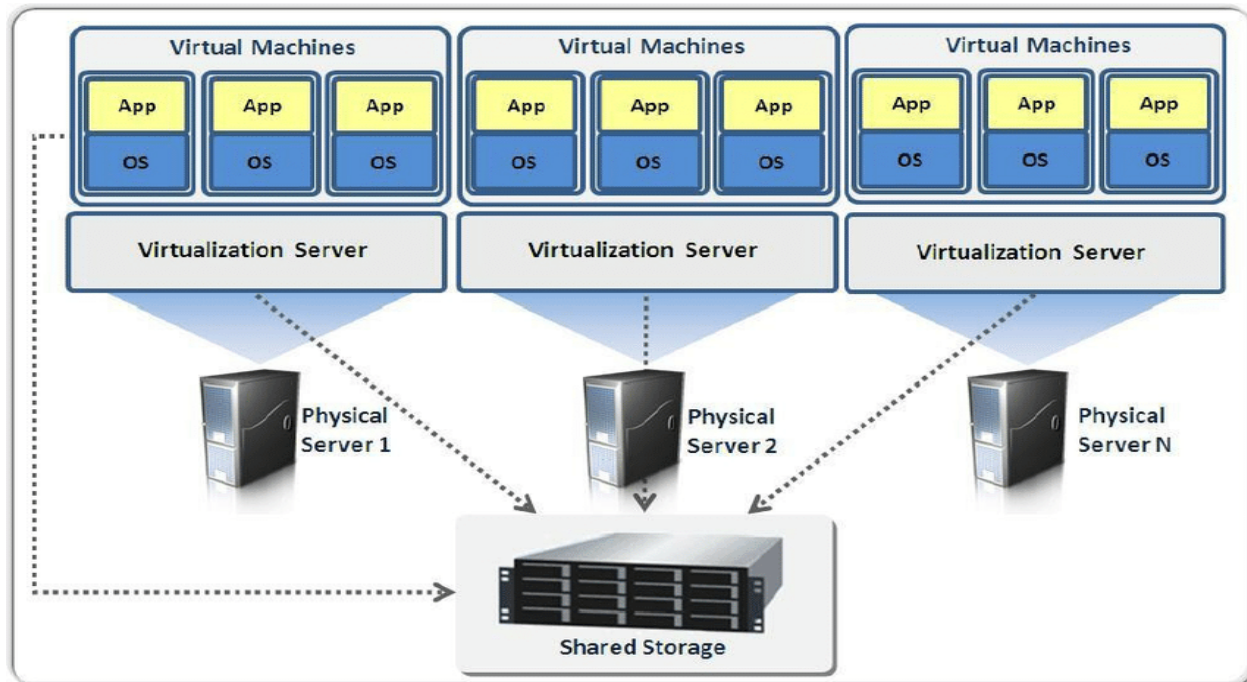


Figure 1: Virtualization

### 1.3. Advantages of Virtualization

Virtualization provides several benefits, including:

**Cost savings:** Virtualization helps organizations optimize their hardware utilization by consolidating multiple virtual machines onto fewer physical servers. This reduces the need for purchasing and maintaining a large number of physical machines, resulting in significant cost savings in terms of hardware acquisition, power consumption, cooling, and data center space.

**Improved resource utilization:** Virtualization allows for the efficient utilization of hardware resources. By running multiple virtual machines on a single physical server, the available processing power, memory, and storage capacity can be effectively shared and allocated as needed. This results in better resource utilization and eliminates the problem of underutilized servers, maximizing the return on investment for hardware resources. [3]

**Enhanced flexibility and scalability:** Virtualization provides the ability to dynamically allocate resources to virtual machines as demands fluctuate. This flexibility allows IT administrators to scale up or down the resources assigned to VMs based on changing workloads or business needs, ensuring optimal performance and responsiveness. [4]

**Simplified disaster recovery and business continuity:** Virtualization simplifies the process of backup, disaster recovery, and business continuity planning. Virtual machines can be easily replicated, backed up, and restored, making data protection and recovery more efficient. In

the event of hardware failures or disasters, virtual machines can be quickly migrated or restored to alternate hardware, minimizing downtime and ensuring business continuity. [5]

**Streamlined testing and development:** Virtualization is widely used in software development and testing environments. It allows developers to create and manage multiple virtual machines, each with a different configuration, operating system, or software stack. This enables efficient testing, debugging, and deployment of applications without the need for dedicated physical hardware for each development environment. [6]

**Increased security and isolation:** Virtualization provides strong isolation between virtual machines, enhancing security. Each virtual machine operates in its own isolated environment, preventing applications or processes from interfering with each other. This isolation helps contain security breaches, reducing the impact and spread of potential threats. [7]

**Green IT and energy efficiency:** Virtualization contributes to environmental sustainability by reducing energy consumption. By consolidating multiple virtual machines onto fewer physical servers, power consumption and cooling requirements are reduced, leading to lower energy costs and a smaller carbon footprint. [8] [9]

**Legacy system preservation:** Virtualization allows organizations to preserve and extend the life of legacy systems. By encapsulating older applications and operating systems in virtual machines, they can continue to run on modern hardware and software infrastructure without compatibility issues or the need for dedicated legacy hardware.

## 1.4. Limitations of Virtualization

While virtualization offers numerous benefits, it also has some limitations (and on which, the engineers and scientists are working and minimizing the gap) that should be considered:

**Performance overhead:** Virtualization introduces a layer of abstraction between the virtual machines and the underlying physical hardware. This abstraction can result in a slight performance overhead due to the need for the hypervisor to manage resource allocation and mediation between virtual machines. Although modern virtualization technologies have minimized this overhead, resource-intensive applications or real-time systems may experience a slight decrease in performance compared to running on bare metal. [10]

**Single point of failure:** When multiple virtual machines run on a single physical server, there is a risk of a single point of failure. If the physical server hosting multiple virtual machines experiences a hardware failure, it can affect all the virtual machines running on it. To mitigate this risk, organizations employ techniques like high availability clustering, distributed virtualization, or redundant hardware setups. [11]

**Complexity in management:** Virtualized environments can introduce complexity in terms of management and administration. IT administrators need to have expertise in managing

virtualization technologies and configuring virtual machines. Additionally, the dynamic nature of virtual machines and their resource allocation requires careful monitoring and optimization to ensure efficient performance and resource utilization.

**Limited hardware access:** Virtualization abstracts the underlying physical hardware, which can restrict direct access to certain hardware features or peripherals. Some applications or workloads that rely heavily on hardware-specific features may not perform optimally or may require additional configuration to access the required resources. [12]

**Licensing considerations:** Virtualization can have implications for software licensing. Some software vendors have specific licensing models or restrictions when it comes to virtualized environments. It is important to understand and comply with licensing agreements to avoid compliance issues and unexpected costs. [13]

**Security risks:** While virtualization can enhance security by isolating virtual machines, it also introduces potential security risks. The hypervisor and virtualization management layers become critical components in the virtualized environment, and any vulnerabilities in these layers can pose security threats. Proper security measures, such as regular updates and patches, secure configurations, and access controls, must be implemented to mitigate these risks. [14]

**Resource contention:** In a virtualized environment, multiple virtual machines share the physical resources of a single server. If resource allocation is not properly managed, resource contention can occur, leading to performance degradation. Careful planning and monitoring are required to ensure that virtual machines have adequate resources to operate efficiently and that resource allocation is balanced across the virtualized infrastructure.

It is important to evaluate these limitations in the context of specific use cases and requirements to make informed decisions about adopting virtualization technologies.

## 1.5. Types of Virtualization

There are several types of virtualization, each catering to different aspects of computing resources. Here's a brief description of some commonly used types of virtualization:

**Server Virtualization:** This type of virtualization involves creating multiple virtual machines (VMs) on a single physical server. Each VM runs its own operating system and applications, appearing as separate servers. Server virtualization allows for efficient utilization of server hardware and the consolidation of multiple servers onto fewer physical machines. [15]

**Desktop Virtualization:** Desktop virtualization enables the creation of virtual desktop infrastructure (VDI), where multiple virtual desktops run on a centralized server. [16] Users can access their virtual desktops remotely, providing flexibility, centralized management, and improved security. Desktop virtualization can be implemented using technologies such as Virtual Desktop Infrastructure (VDI) or Desktop as a Service (DaaS).

**Network Virtualization:** Network virtualization abstracts networking resources, such as switches, routers, and firewalls, into virtual entities. It allows for the creation of virtual networks that are independent of the underlying physical infrastructure. Network virtualization enables improved network agility, scalability, and easier management of complex network configurations. [17]

**Storage Virtualization:** Storage virtualization involves pooling physical storage resources from multiple storage devices and presenting them as a unified storage system. It allows for centralized management, efficient resource allocation, and simplified data migration. Storage virtualization abstracts the complexities of storage systems, making it easier to manage and scale storage infrastructure. [18]

**Application Virtualization:** Application virtualization separates applications from the underlying operating system and encapsulates them into self-contained packages. These packages, known as virtualized applications, can run on different operating systems without conflicts or dependencies. Application virtualization provides flexibility in deploying and managing applications, simplifies compatibility issues, and improves isolation between applications. [19]

**Operating System Virtualization:** Also known as containerization or operating system-level virtualization, this type of virtualization allows for the creation of multiple isolated user spaces within a single operating system instance. Containers share the same OS kernel but are isolated from one another, providing lightweight and efficient virtualization. [20] Operating system virtualization is commonly used in cloud environments and micro-services architectures.

**Hardware Virtualization:** Hardware virtualization enables the creation of virtual machines that run on top of physical hardware. It utilizes a hypervisor or virtual machine monitor (VMM) to abstract and manage the underlying hardware resources. Hardware virtualization allows multiple operating systems and applications to run concurrently on the same physical machine, providing isolation, flexibility, and efficient resource utilization. [21]

These types of virtualization can be used independently or in combination to meet specific IT requirements. Each type offers its own set of benefits and use cases, enabling organizations to optimize their infrastructure, improve resource utilization, and simplify management.

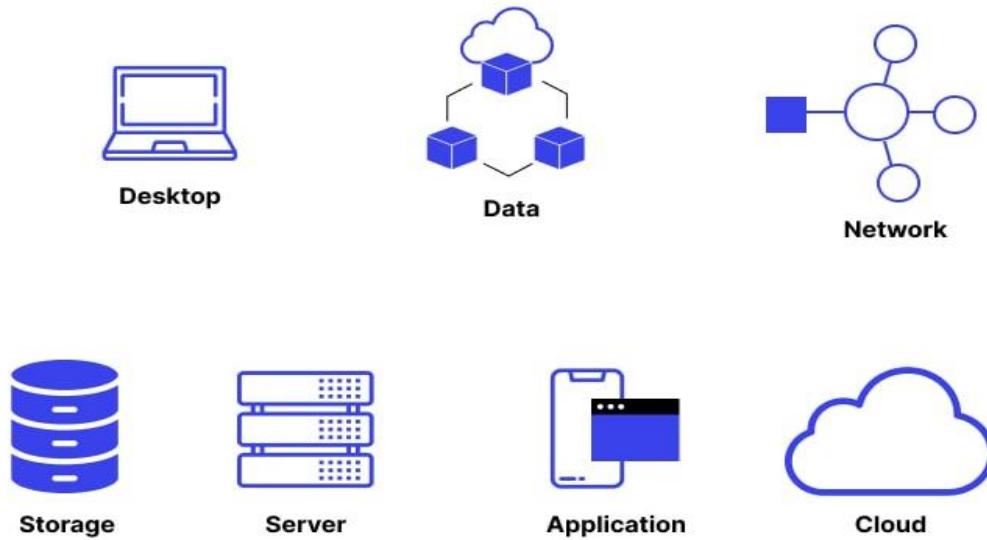


Figure 2: Types of Virtualization

## 1.6. Virtualization of Network Infrastructure

Network Infrastructure Virtualization is a concept that involves virtualizing and abstracting various components of the network infrastructure to create a more flexible, scalable, and efficient networking environment. It aims to decouple network services, functions, and resources from the underlying physical infrastructure, providing virtualized instances that can be dynamically allocated, provisioned, and managed.

Network Infrastructure Virtualization encompasses several technologies and techniques, including Software-Defined Networking (SDN), Network Function Virtualization (NFV), and Network Virtualization (NV). These technologies work together to virtualize and optimize different aspects of the network infrastructure, leading to improved agility, scalability, and cost-effectiveness.

### 1.6.1. Key Components of Network Infrastructure Virtualization:

**Software-Defined Networking (SDN):** SDN separates the control plane from the data plane in network devices, centralizing network management and control through a software-based controller. It allows administrators to programmatically configure and control the network, enabling greater agility, automation, and network orchestration. [22]

**Network Function Virtualization (NFV):** NFV virtualizes network functions such as routers, switches, firewalls, load balancers, and other network services. By running these functions as software instances on commodity hardware or virtual machines, NFV eliminates the need for dedicated physical appliances, leading to cost savings, scalability, and faster deployment of new services. [23]

Network Virtualization (NV): NV abstracts and virtualizes network resources, including network connectivity, addressing, and security, from the underlying physical infrastructure. It enables the creation of multiple virtual networks on top of a shared physical network, providing isolation, flexibility, and efficient resource utilization. NV allows organizations to partition and customize network services and configurations according to their specific requirements. [17]

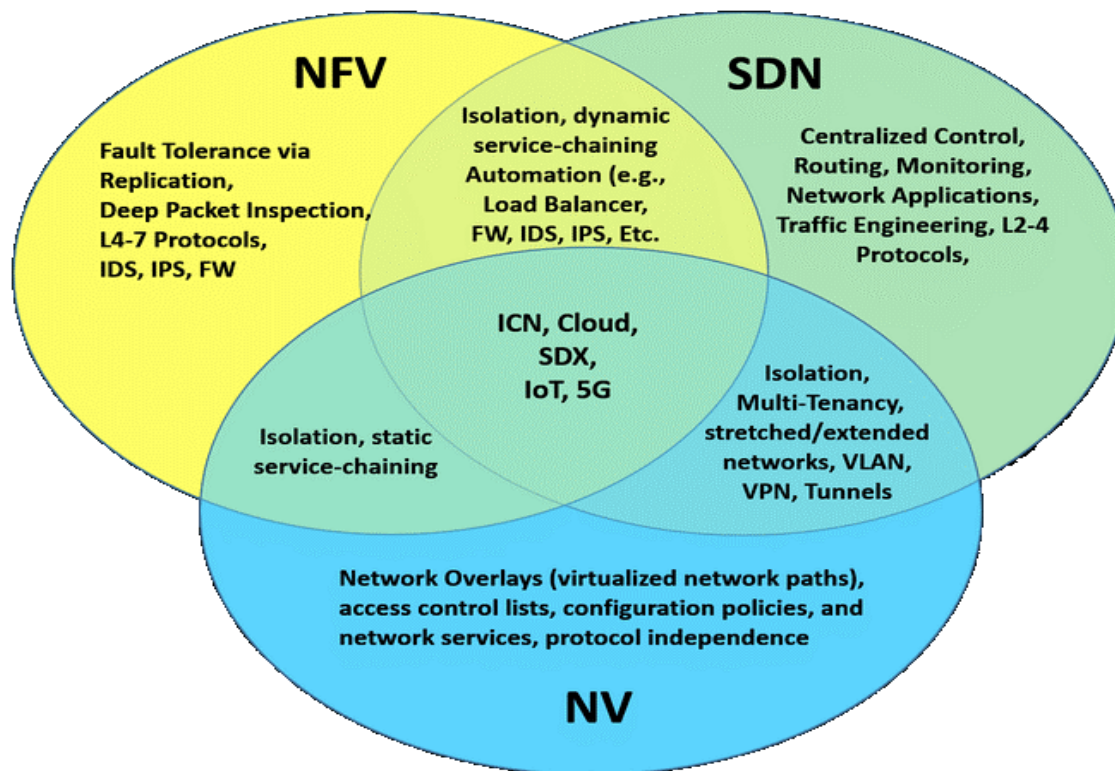


Figure 3: The Key Components of Network Virtualization

### 1.6.2. Benefits of Network Infrastructure Virtualization:

**Flexibility and Scalability:** By decoupling network services from the underlying physical infrastructure, virtualized network resources can be dynamically allocated and provisioned, enabling organizations to scale their networks rapidly and adapt to changing demands.

**Efficient Resource Utilization:** Network Infrastructure Virtualization allows for the consolidation of multiple virtual networks onto a shared physical infrastructure, optimizing resource utilization and reducing costs. It eliminates the need for dedicated hardware for each network service, leading to better efficiency and cost savings.

**Agility and Rapid Service Deployment:** With virtualized network functions and centralized control, organizations can quickly deploy new network services and applications, reducing time-to-market and enabling faster innovation.

**Improved Network Security and Isolation:** Network virtualization provides enhanced security by isolating virtual networks from each other. It enables granular control over traffic segmentation, firewall policies, and access controls, reducing the risk of unauthorized access and improving overall network security.

**Simplified Network Management:** Virtualization of network infrastructure simplifies network management by providing a centralized management and control plane. It allows for programmable configuration, automation, and orchestration of network resources, leading to better network control and operational efficiency.

## **1.7. Distributed Computing Vs Centralized Computing**

Centralized and distributed computing are two contrasting approaches to organizing and processing computational tasks. Here are the key differences between the two:

### **1. Control and Decision Making:**

- **Centralized Computing:** In a centralized computing model, a single central node or server has control and decision-making authority over the entire system. It manages and coordinates all the computing resources and makes decisions on task allocation, data storage, and processing. [24]

- **Distributed Computing:** In a distributed computing model, there is no single central authority. Multiple nodes or servers work together as a network, sharing responsibilities and making collective decisions. Each node has some degree of autonomy and can make decisions locally based on its own resources and knowledge. [25]

### **2. Resource Allocation:**

- **Centralized Computing:** In a centralized model, the central server controls and allocates resources such as processing power, memory, and storage. It determines how resources are distributed among different tasks or users based on predefined policies.

- **Distributed Computing:** In a distributed model, resources are distributed across multiple nodes, and each node independently manages its own resources. Nodes can allocate resources based on local availability and requirements, which enables better scalability and fault tolerance. [24]

### **3. Communication and Data Sharing:**

- **Centralized Computing:** In a centralized model, communication and data sharing primarily happen between clients and the central server. Clients send requests to the server, and the server responds with the requested data or performs the necessary computations.



- Distributed Computing: In a distributed model, communication and data sharing occur between nodes in the network. Nodes exchange data, messages, or tasks to collaborate on solving a problem or achieving a common goal. This decentralized communication pattern enables parallel processing and efficient utilization of distributed resources. [24]

#### 4. Scalability and Fault Tolerance:

- Centralized Computing: Centralized systems can face scalability challenges as the workload increases since the central server may become a performance bottleneck. Additionally, if the central server fails or experiences issues, the entire system can be affected.

- Distributed Computing: Distributed systems can scale more effectively by adding more nodes to the network, enabling parallel processing and increased computational capacity. They also tend to be more fault-tolerant since failures in individual nodes typically do not impact the entire system.

#### 5. Complexity and Maintenance:

- Centralized Computing: Centralized systems are generally easier to design, develop, and maintain since all components are under the control of a single entity. Updates and changes can be implemented more easily, but it also means that a single point of failure can disrupt the entire system.

- Distributed Computing: Distributed systems are more complex to design and maintain due to the decentralized nature of the architecture. Coordination between nodes, data consistency, and fault management require additional considerations. However, distributed systems can offer higher resilience and availability through redundancy and load balancing.

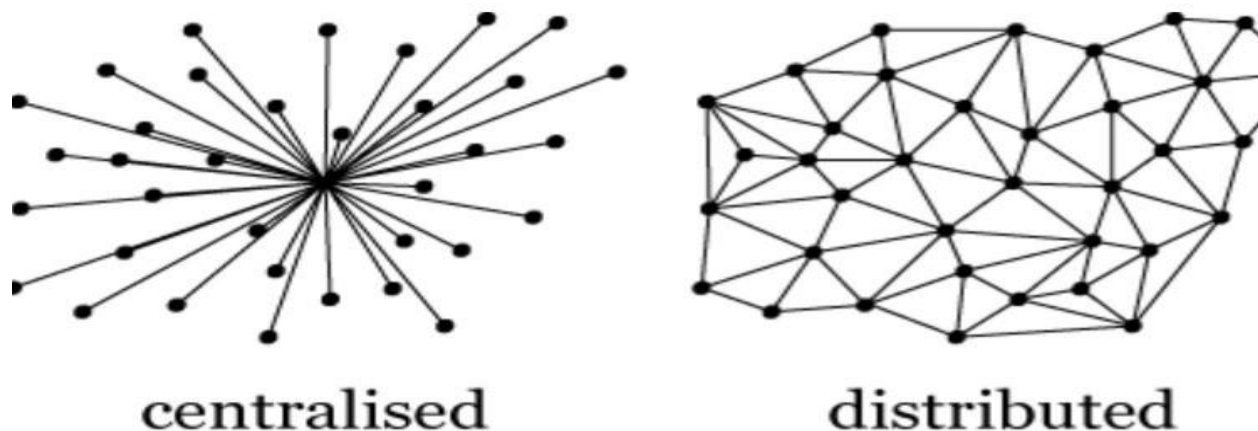


Figure 4: Differentiation of Centralized and Distributed Computing



# Chapter 2: Technology Survey

## 2.1. Software Defined Networking (SDN)

It is an approach to network architecture and management that separates the control plane from the data plane of a network, enabling centralized control and programmability of network resources.

In traditional networks, network devices such as routers and switches have their control plane and data plane tightly coupled. The control plane handles tasks like routing protocols, network management, and policy enforcement, while the data plane is responsible for forwarding packets based on the established rules.

In an SDN architecture [26], the control plane is decoupled from the data plane and moved to a centralized controller. The controller is a software-based entity that manages and controls the behavior of network devices. It interacts with network devices through open protocols such as OpenFlow, which allows it to directly program the forwarding behavior of switches.

### 2.1.1. Differences between Traditional networking and SDN

**Centralized control:** In SDN, the control plane is centralized, which provides a global view of the network and allows administrators to define and manage network policies from a single point of control. [27] In traditional networks, control is distributed across individual network devices.

**Programmability:** SDN offers programmability through the controller, enabling dynamic configuration and management of network resources. Network administrators can use software applications and APIs to control and automate network behavior. Traditional networks typically lack this level of programmability. [28]

**Network agility:** SDN allows for easier network configuration, management, and provisioning. It enables rapid deployment of new services and applications by programmatically adapting the network to changing requirements. Traditional networks often require manual configuration and are less flexible.

**Network virtualization:** SDN can provide network virtualization, allowing the creation of multiple virtual networks (network slices) on a shared physical infrastructure. Each virtual network can have its own policies, routing, and security settings. Traditional networks usually do not provide native support for network virtualization.

**Simplified troubleshooting:** With a centralized view of the network and programmable control, SDN simplifies network troubleshooting and monitoring. It allows administrators to identify and resolve issues more efficiently by analyzing network-wide data from the controller. Traditional networks may require more manual and distributed troubleshooting.

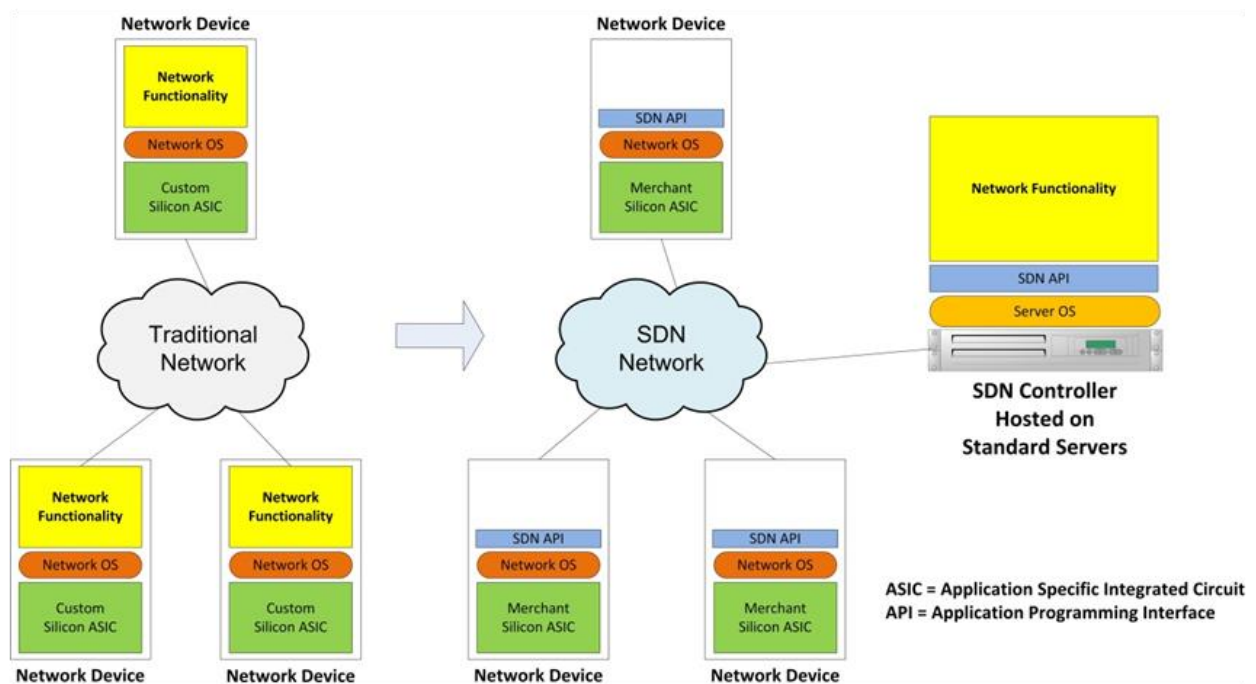


Figure 5: Differences between SDN and Traditional Network

### 2.1.2. SDN Architecture

The architecture of Software-Defined Networking (SDN) typically consists of three key components: the application layer, the control layer, and the infrastructure layer. Let's explore each of these components in more detail:

#### 1. Application Layer:

The application layer consists of software applications that run on top of the SDN architecture. [26] These applications interact with the SDN controller to define and manage network policies, services, and configurations. They leverage the programmability of SDN to control network behavior based on specific requirements, such as traffic engineering, load balancing, security, and quality of service (QoS). Examples of SDN applications include network monitoring tools, firewall management applications, and traffic optimization services.

#### 2. Control Layer:

The control layer is the central component of the SDN architecture and is responsible for managing and controlling the network devices. It consists of an SDN controller, which acts as the brain of the network. The controller communicates with the network devices in the

infrastructure layer using a standardized protocol such as OpenFlow or NETCONF. The primary functions of the control layer include:

- a. Network Management: The controller manages and configures network devices, including switches, routers, and virtual switches, by sending instructions to the devices via the control protocol. [27]
- b. Network Control: The controller defines and enforces network policies, routing decisions, and traffic flows across the network. [29] It collects information about the network state, topology, and traffic from the infrastructure layer and makes decisions to optimize network performance.
- c. API and Interface: The control layer provides APIs and interfaces that enable communication between the applications in the application layer and the network devices in the infrastructure layer. These APIs allow applications to program the behavior of the network, retrieve network statistics, and receive event notifications.

### **3. Infrastructure Layer:**

The infrastructure layer consists of the physical and virtual network devices that forward network traffic based on the instructions received from the control layer. This layer includes switches, routers, firewalls, load balancers, and other network devices. The infrastructure layer may also include virtual switches and virtual network functions (VNFs) that run on hypervisors in virtualized environments.

The communication between the control layer and the infrastructure layer occurs through an open protocol, such as OpenFlow [30] [31] [32] [33]. The control layer instructs the network devices on how to process and forward packets, configure forwarding tables, apply access control rules, and handle network events. The infrastructure layer implements the data plane functionality and performs the actual packet forwarding based on the instructions received from the control layer. Overall, the SDN architecture separates the control plane from the data plane, providing centralized control and programmability of the network. This separation enables network administrators and operators to have a holistic view of the network, simplify network management, and dynamically adapt the network to changing requirements.

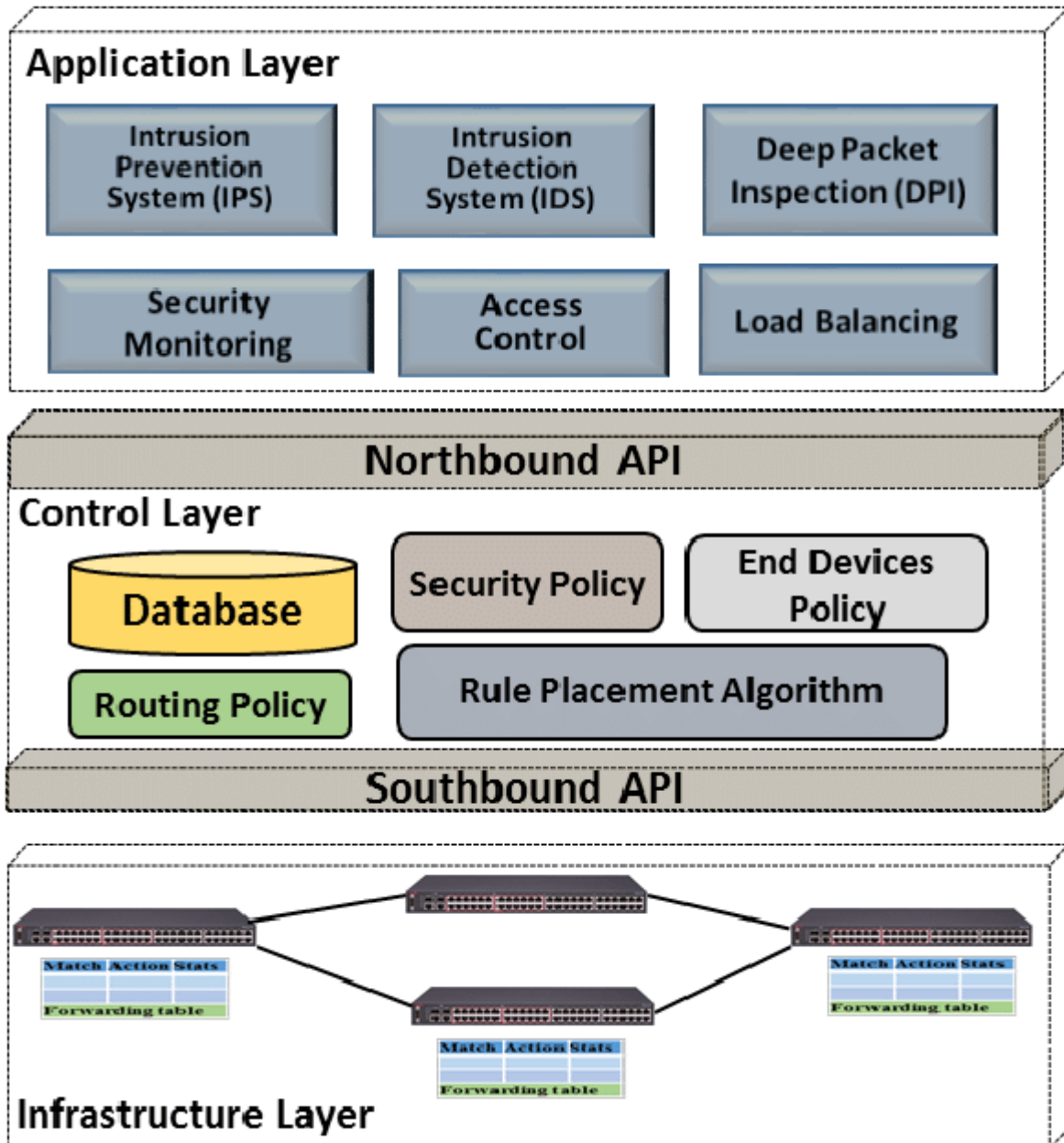


Figure 6: SDN Architecture

#### 2.1.2.1. Northbound API

The Northbound API [34] refers to the interface or set of application programming interfaces (APIs) that enable communication between the SDN controller (control layer) and the applications running in the application layer. [26] [34] It allows applications to interact with the SDN controller and leverage its capabilities to control and manage the network. Here are some key points about the Northbound API:

- **Functionality:** The Northbound API exposes functions and methods that applications can use to communicate with the SDN controller. It enables applications to define network policies, configure network resources, retrieve network state information, and receive event notifications.
- **Application Interaction:** Applications in the Northbound API interact with the controller to convey their requirements and intentions. They can request specific network behavior, such as setting up virtual networks, defining traffic policies, or obtaining real-time network statistics.
- **Programmability:** The Northbound API provides a programmable interface that allows developers to build applications that control and manage the network. By using the API, applications can dynamically adapt the network to changing conditions and requirements.
- **Standardization:** The Northbound API can follow industry-standard protocols such as RESTful APIs or other custom-defined protocols. The API provides a standardized and well-defined interface, allowing applications from different developers to interact with various SDN controllers in a consistent manner.

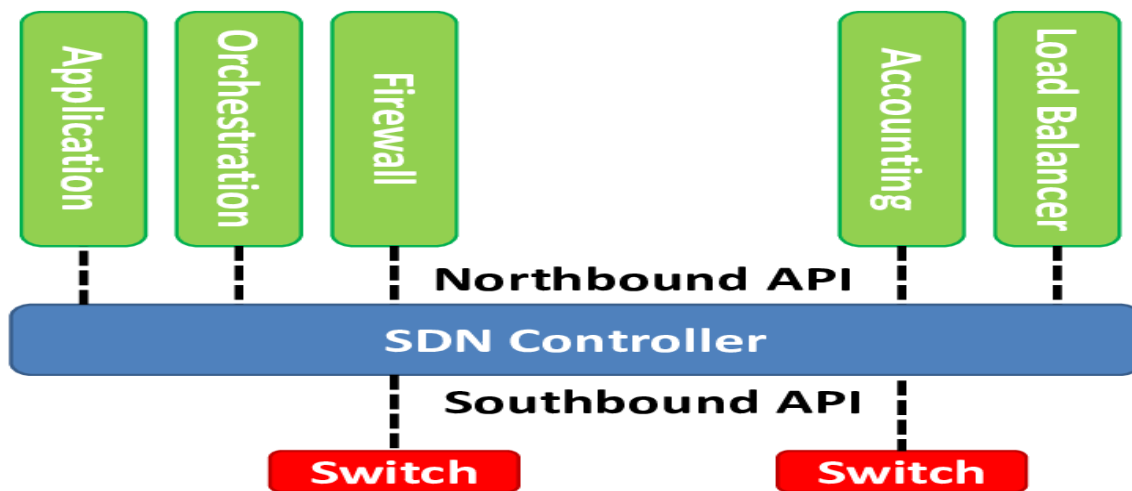


Figure 7: Northbound API and Southbound API

#### 2.1.2.2. Southbound API

The Southbound API [35] refers to the interface or set of APIs that facilitate communication between the SDN controller (control layer) and the network devices in the infrastructure layer. It allows the controller to control and manage the behavior of network devices. Here are some key points about the Southbound API:

- **Device Interaction:** The Southbound API enables the SDN controller to communicate with network devices such as switches, routers, and virtual switches. It provides a standardized interface for the controller to configure, monitor, and control the behavior of these devices.
- **Protocol Compatibility:** The Southbound API supports specific protocols that network devices understand, allowing the controller to interact with a wide range of devices. One commonly used protocol in the Southbound API is OpenFlow, which allows the controller to program the forwarding behavior of OpenFlow-enabled switches.
- **Control Instructions:** The Southbound API allows the controller to send instructions to network devices regarding forwarding rules, routing decisions, and traffic flow handling. It provides a way for the controller to exert control over the data plane of the network devices.
- **Data Collection:** The Southbound API facilitates the collection of network state information from network devices. It enables the controller to gather data such as traffic statistics, link statuses, and device health information. This information helps the controller make informed decisions and optimize network performance.

In summary, the Northbound API focuses on the communication between applications and the SDN controller, enabling applications to control and manage the network. On the other hand, the Southbound API focuses on the communication between the SDN controller and network devices, allowing the controller to control and configure the behavior of the devices in the network infrastructure.

### **2.1.3. The Controller**

Software-defined networking (SDN) is a new paradigm of networking in which the architecture transitions from a completely distributed form to a more centralized form. In SDN, the control plane is separated from the data plane, and a centralized controller manages the network. The SDN controller is a critical component of the SDN architecture, responsible for managing the network and implementing policies. In this article, we will discuss the SDN controller, its functions, and some of the approaches used to optimize its placement.

#### **2.1.3.1. Functions of an SDN Controller**

The SDN controller is responsible for managing the network and implementing policies. Some of the functions of an SDN controller include:

**Network Management:** The SDN controller manages the network by configuring network devices, monitoring network traffic, and collecting network statistics.

**Policy Enforcement:** The SDN controller enforces policies by programming network devices to forward traffic according to specific rules.

**Network Virtualization:** The SDN controller creates virtual networks by abstracting the underlying physical network.

**Traffic Engineering:** The SDN controller optimizes network traffic by dynamically adjusting network paths based on network conditions.

### 2.1.3.2. Approaches to SDN Controller Placement

The placement of the SDN controller is an important factor in the performance of an SDN network. Several approaches have been proposed to optimize the placement of the SDN controller, including:

**Static Placement:** In this approach, the SDN controller is placed in a fixed location in the network. This approach is simple but may not be optimal in dynamic networks.

**Dynamic Placement:** In this approach, the SDN controller is placed dynamically based on network conditions. This approach is more complex but can lead to better performance in dynamic networks.

**Sub modularity-Based Placement:** This approach uses submodular optimization to find the optimal placement of the SDN controller. Sub modularity is a property of set functions that captures the notion of diminishing returns. This approach has been shown to be effective in optimizing the placement of the SDN controller in IoT networks.

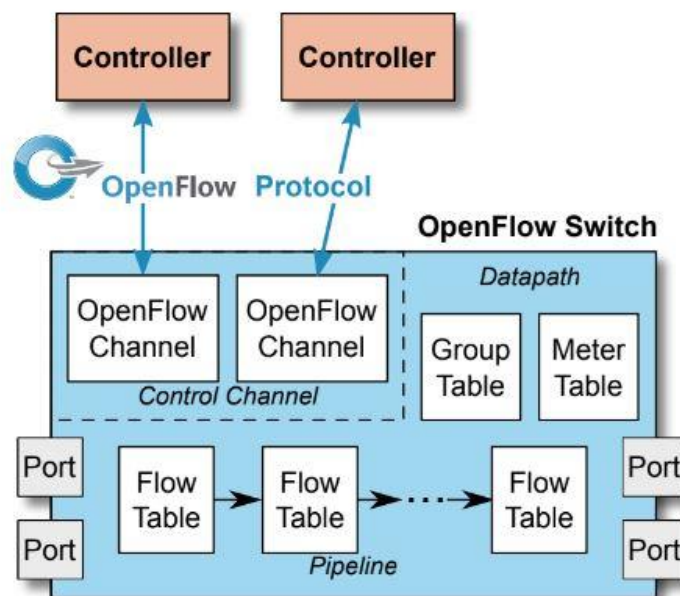


Figure 8: The Controller

### 2.1.4. OpenFlow Protocol

OpenFlow [36] is a widely adopted communication protocol that enables the programmability and control of network devices in Software-Defined Networking (SDN)

environments. It defines a standardized way for the SDN controller to communicate with the forwarding plane of network switches, routers, and other devices. Here's an overview of the OpenFlow protocol:

### 1. Purpose and Objectives:

The primary goal of the OpenFlow protocol is to separate the control plane from the data plane in network devices, allowing centralized control and programmability. It provides a standardized interface for the SDN controller to define and manage the forwarding behavior of network devices, facilitating dynamic network configuration and management.

### 2. Architecture:

OpenFlow operates as a communication protocol between the SDN controller (control layer) and the network devices (data plane) such as switches. It uses a secure channel, typically TCP/IP, for communication between the controller and the OpenFlow-enabled devices.

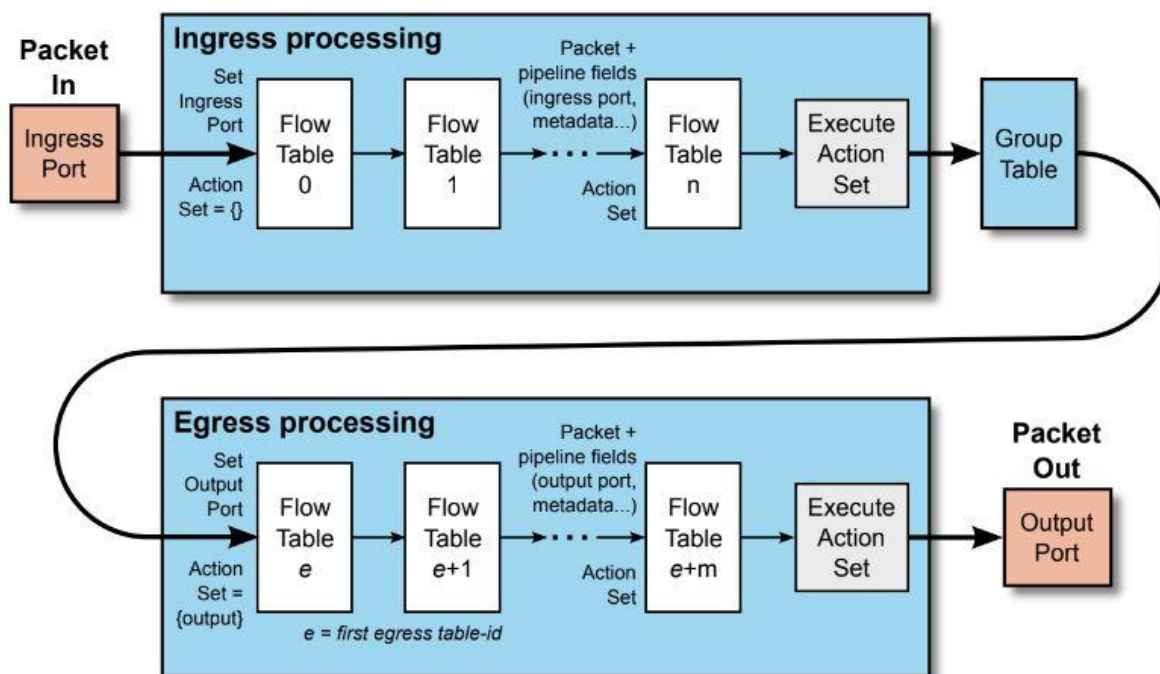


Figure 9: Packet-In and Packet-Out (OpenFlow)

### 3. Flow Tables and Match-Action Paradigm:

At the heart of OpenFlow is the concept of flow tables within network devices. A flow table consists of flow entries, each representing a specific set of packet header fields and corresponding actions to be taken for matching packets. When a packet arrives at an



OpenFlow-enabled switch, the switch looks up the packet's header fields in its flow table and applies the associated actions, such as forwarding, dropping, or modifying the packet.

#### **4. Message-based Protocol:**

OpenFlow uses a message-based protocol for communication. The controller and the network devices exchange different types of messages to perform various functions. Some common message types include:

- Features Request/Reply: The controller requests information about the capabilities and features of the network device, and the device responds with its capabilities.
- Flow Mod: The controller sends flow modification instructions to the network device, such as adding, modifying, or deleting flow entries in the flow table.
- Packet-In/Packet-Out: When a packet does not match any flow entry in the flow table, the network device sends a Packet-In message to the controller. The controller can then instruct the device to take appropriate action with a Packet-Out message.

#### **5. Centralized Control and Programmability:**

OpenFlow enables centralized control of network devices through the SDN controller. The controller has a global view of the network and can program the forwarding behavior of switches dynamically. It can define routing decisions, traffic policies, and implement network-wide optimizations based on real-time network conditions and requirements.

#### **6. OpenFlow Versions:**

OpenFlow has evolved over time, with each version introducing new features, enhancements, and capabilities. The initial versions of OpenFlow, such as OpenFlow 1.0 and 1.1, provided fundamental functionality. Subsequent versions, like OpenFlow 1.3 and 1.4, introduced features like group tables, IPv6 support, multiple tables, and more advanced matching capabilities.

#### **7. Industry Adoption:**

OpenFlow has gained significant adoption and support from the networking industry. Many networking vendors and open-source projects have implemented OpenFlow in their products, allowing interoperability and promoting the development of SDN ecosystems.

OpenFlow has been instrumental in enabling the flexibility, programmability, and central control that are central to Software-Defined Networking. By defining a standardized protocol for communication between the controller and network devices, OpenFlow has played a crucial role in the widespread adoption and evolution of SDN technology.

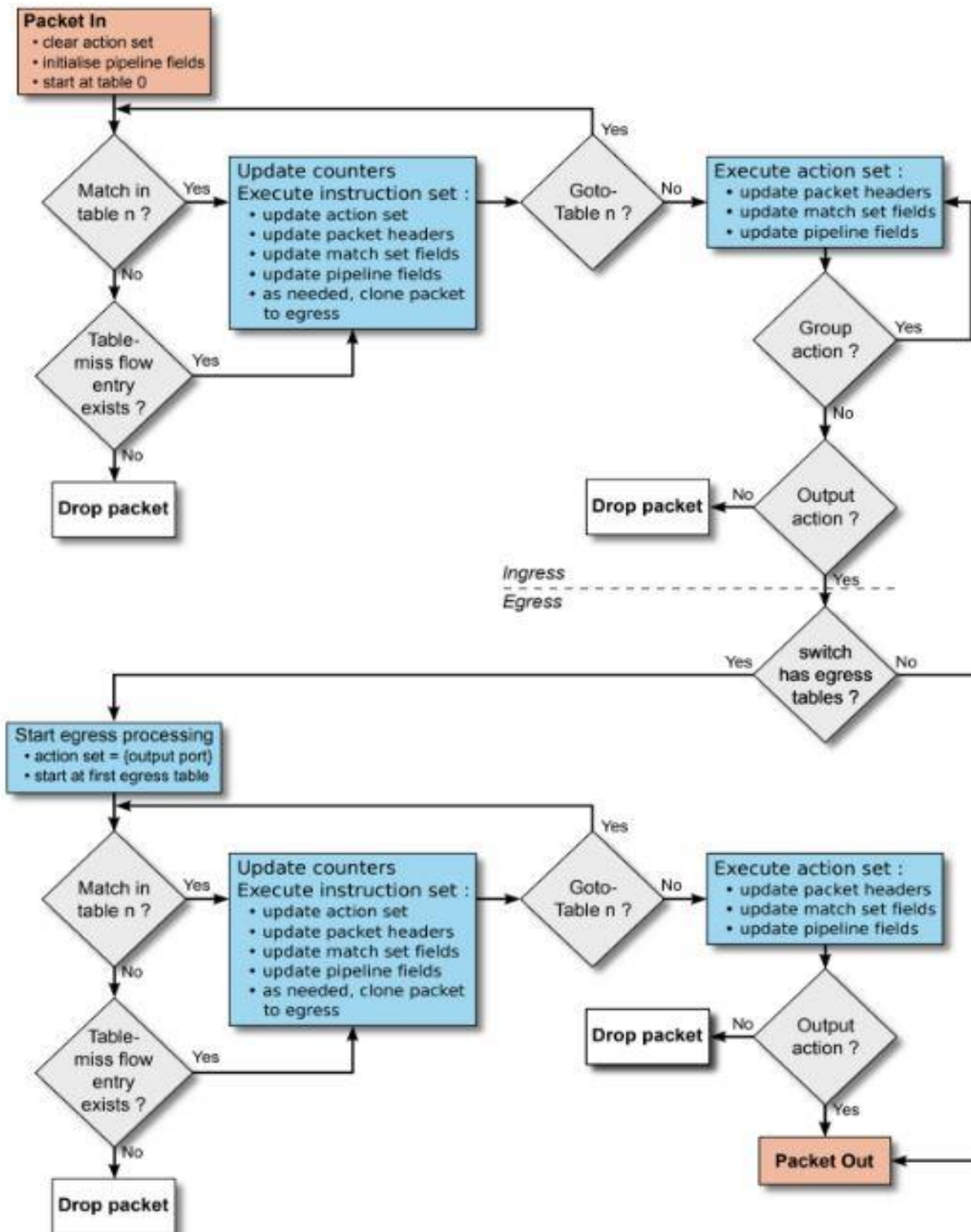


Figure 10: Detailed Flow Chart of OpenFlow Protocol

## 2.2. Network Function Virtualization

Network Function Virtualization (NFV) is an architectural framework and technology that aims to virtualize and consolidate traditional network functions onto standard hardware, such as servers, switches, and storage devices. It decouples network functions from dedicated physical appliances and enables them to run as software-based instances on virtual machines (VMs) or containers. NFV offers several benefits, including flexibility, scalability, cost savings, and faster deployment of network services. Here are some key aspects of Network Function Virtualization:

**1. Virtualized Network Functions (VNFs):** NFV replaces specialized and dedicated network appliances with Virtualized Network Functions (VNFs) [37]. VNFs are software implementations of network functions that traditionally run on proprietary hardware. Examples of network functions that can be virtualized include firewalls, routers, load balancers, intrusion detection systems (IDS), and WAN accelerators. VNFs are deployed on standard servers or cloud infrastructure, leveraging virtualization technologies.

**2. Virtualization Technologies:** NFV leverages virtualization technologies like hypervisors (such as VMware ESXi, KVM) or containerization platforms (such as Docker, Kubernetes) to create and manage virtual instances of network functions. These virtualized instances can be dynamically provisioned, scaled up or down, and migrated across physical hosts, providing agility and resource efficiency.

**3. Orchestration and Management:** NFV deployments typically involve orchestration and management frameworks to automate the lifecycle of VNFs. These frameworks handle tasks such as VNF instantiation, scaling, chaining, and service composition. They provide centralized management and control, allowing operators to efficiently manage and orchestrate network services across distributed infrastructure.

**4. Scalability and Elasticity:** NFV enables network services to scale based on demand. VNFs can be easily replicated or instantiated in response to increased traffic or workload requirements. This scalability helps organizations optimize resource utilization, dynamically allocate resources, and improve service performance.

**5. Cost Savings:** By virtualizing network functions, organizations can reduce capital and operational expenses. NFV eliminates the need for dedicated hardware appliances, reducing equipment costs, power consumption, and physical space requirements. It also simplifies maintenance and reduces operational complexity by enabling centralized management and automated provisioning.

**6. Service Agility and Rapid Deployment:** NFV allows network services to be provisioned and deployed more quickly compared to traditional methods. VNFs can be dynamically instantiated and chained together to create complex service topologies. This agility facilitates faster service deployment, testing, and scaling, enabling organizations to respond rapidly to changing business needs and market demands.

**7. Ecosystem and Interoperability:** NFV promotes an open ecosystem and interoperability among different vendors' solutions. It encourages the use of standardized interfaces and APIs to ensure compatibility between VNFs from various vendors and different NFV infrastructure (NFVi) components.

Overall, Network Function Virtualization offers a flexible and scalable approach to network infrastructure by replacing dedicated hardware appliances with software-based VNFs. It brings numerous advantages in terms of cost savings, service agility, scalability, and easier management of network services, making it a key technology in modern network deployments.

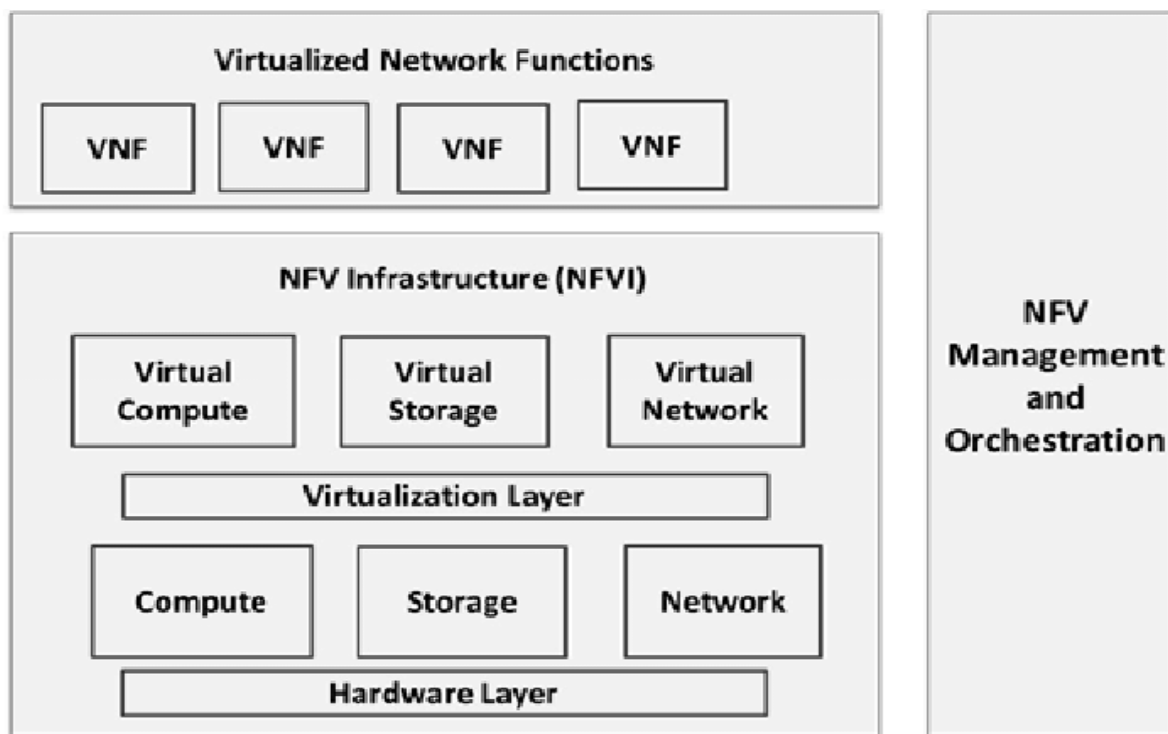


Figure 11: NFV Infrastructure

### 2.2.1. Virtual Network Functions

Virtual Network Functions (VNFs) refer to network functions that can be virtualized and run on virtual machines (VMs) or containers within a software-defined networking (SDN) or network functions virtualization (NFV) environment. These functions traditionally existed as dedicated hardware appliances deployed in physical networks, but with the advent of virtualization and cloud computing, they can now be implemented as software instances. VNFs represent specific network services or functions, such as firewalls, routers, load balancers, intrusion detection systems, WAN accelerators, or network address translation (NAT) devices. By virtualizing these functions, they can be deployed and managed more

flexibly and dynamically, allowing network operators to optimize resource utilization, scale services as needed, and reduce costs.

The virtualization of network functions brings several benefits. It enables network operators to deploy and provision services faster, as VNFs can be spun up or down based on demand. It allows for more efficient utilization of hardware resources by running multiple VNFs on shared physical infrastructure. Additionally, VNFs can be centrally managed and orchestrated, providing greater agility and automation in network operations.

VNFs are typically deployed within a virtualized infrastructure, which includes hypervisors, virtual switches, and other software-defined networking components. These VNFs communicate with each other and with physical network elements using virtual network interfaces, enabling the creation of complex network topologies and services. It's important to note that the concept of VNFs is closely related to the broader concept of Network Functions Virtualization (NFV), which aims to virtualize and consolidate various network functions into software-based solutions. VNFs are the specific instances of virtualized network functions that are deployed and run within an NFV environment.

### **2.2.2. Virtual Network Function – Chain Placement Problem**

The Chain Placement Problem refers to the challenge of determining the optimal placement of a virtual network function chain within a network infrastructure. A virtual network function chain represents a sequence of interconnected VNFs that together provide a specific network service. For example, a chain might consist of a firewall followed by a load balancer and then a deep packet inspection function. The goal is to deploy these VNFs in a way that satisfies performance requirements, minimizes resource consumption, and ensures efficient traffic routing.

The Chain Placement Problem involves deciding which physical or virtual resources within the network infrastructure should host each VNF in the chain. Factors that are typically considered include network topology, available resources (such as computing power, memory, and bandwidth), latency requirements, and traffic patterns.

Solving the Chain Placement Problem efficiently is crucial to ensure that the VNF chain operates effectively and meets the desired performance objectives. Different optimization techniques, algorithms, and heuristics can be employed to address the Chain Placement Problem and find an optimal or near-optimal solution based on the specific requirements and constraints of the network environment.

In the deployment of service functions in large-scale environments, the optimal location of VNF (service) chains in hosting infrastructures is one of the primary challenges. The Problem of Virtualized Network Functions Chain Placement VNF-CPP is NP-Hard, and there is a need for placement strategies that can scale with the size of the problem and find effective solutions in a reasonable amount of time.

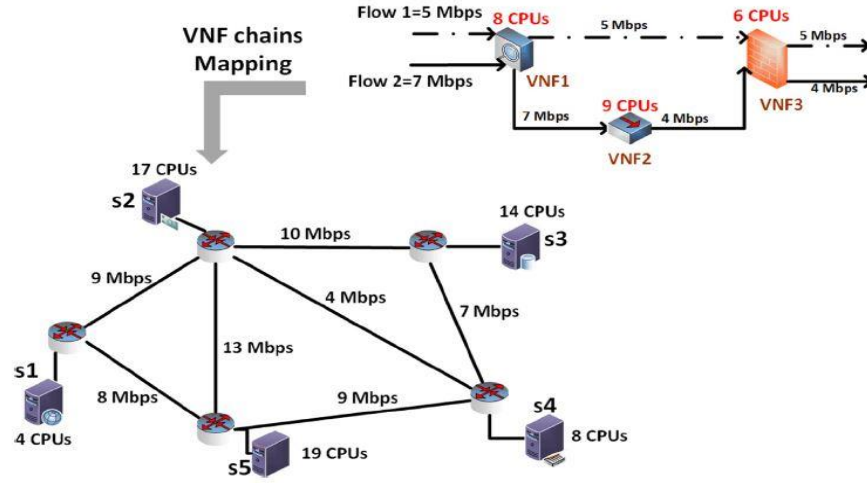


Figure 12: A VNF Chain Infrastructure

There are some proposed algorithms to optimize the VNF-CPP like-

1. A precise approach that is developed from the *Perfect 2-Matching theory* [38] and is meant to handle in polynomial time [39] the problem of VNF chains that comprise three VNFs. This method can solve the issue of VNF chains because it is exact and it is derived from the theory of Perfect 2-Matching. This polynomial case will function as a lower limit, and it will operate in concert with a new family of valid inequalities (facet) to help speed up the resolution of the VNF-CPP for VNF chains of longer lengths (chains that include at least 4 VNFs).

The task known as Perfect 2-Matching requires that all of the nodes in the substrate network be connected to one another in such a manner that the degree value of each node is equal to 2. This is the same as finding cycles in the substrate graph that contain the minimum amount of resources required to meet or verify the resource constraints and restrictions. This is the same as finding cycles in the substrate graph that contain the smallest number of resources necessary. As a result of this transformation and finding, we are now in a position to deal with the VNF-CPP by using the convex hull technique of linear programming for the Perfect 2-Matching polytope. Because of this, we are able to do so for VNF chains whose length (measured in terms of the number of arcs) does not exceed 2. A VNF chain with a length that is less than two has exactly one and only one arc, and the two VNFs that make up the chain are connected to one another by this one and only arc. We enhance the VNF-CPP convex hull by include extra valid inequalities when the number of VNFs (nodes) in a VNF chain is more than three in order to speed up our convergence towards the optimum solution. This method is not going to work very well when used to really large graphs. On the other hand, it will provide optimal solutions for graphs of a size that is manageable, and it may be used to test the efficacy of various alternative solutions or heuristic algorithms.

Let the substrate graph comprises of set of physical edges,  $E_s$  and a set of physical hosts,  $V_s$ .

Let us assume that the bandwidth available on substrate graph link  $e (e \in E_s)$  is denoted by  $C_e$ , and that the necessary throughput of VNF chain  $c$  is  $R_c$ . Then, finding a solution to the VNF-CPP consists of locating edges and nodes that have appropriate processing capabilities and sufficient capacity to host the demand. We choose just edges that are capable of hosting the demands or of managing the chain flow so that we may speed up the convergence process. We suggest minimizing the following utility function as a means of removing all of the edges of  $E_s$  that are unable to meet the demand for the product:

$$\min \sum_{(e \in E, e=(i,j))} (C_e - R_c) 1^+ x_e \quad \dots (1)$$

Where  $x_e$  is a binary variable indicating if an edge  $e$  is selected in the solution i.e.,  $x_e = 0$  or  $1$ . Since we are looking for reduction of available bandwidth consumption, we use:

$$1^+ = \begin{cases} 1, & \text{if } (C_e - R_c) > 0 \\ \infty, & \text{otherwise} \end{cases}$$

To get a better hold on the Perfect 2-matching problem convex hull, we generally use the equality to make sure that all the nodes in the substrate graph will have degree 2, so that they are covered by a cycle:

$$\sum x_e = 2 \quad \dots (2)$$

(The summation runs through the set of edges that are incident to the nodes,  $V_s$ .)

The following set of valid inequalities, noted by Blossom Inequalities [39] [40] [41] (essentially used to solve NP Hard Problems), strengthens the previous constraints to get integer optimal solutions of the VNF-CPP problem using continuous variables.

$$\sum x_e + x(F) \leq \left\lfloor \frac{1}{2} (2|A| + |F|) \right\rfloor \quad \dots (3)$$

Where  $A \in V_s$  and  $F \subseteq \delta(A)$ ,  $\delta$  represents the set of edges that are incident to the node  $A$ .

And,

$$\sum_{v \in A} 2 + |F| \text{ is odd.}$$

... (4)

The mathematical model is thus:

$$\min \sum_{(e \in E, e=(i,j))} (C_e - R_c) 1^+ x_e,$$

*subject to:*

$$\sum x_e = 2$$

$$\sum x_e + x(F) \leq \left\lfloor \frac{1}{2}(2|A| + |F|) \right\rfloor$$

$$\sum_{v \in A} 2 + |F| \text{ is odd.}$$

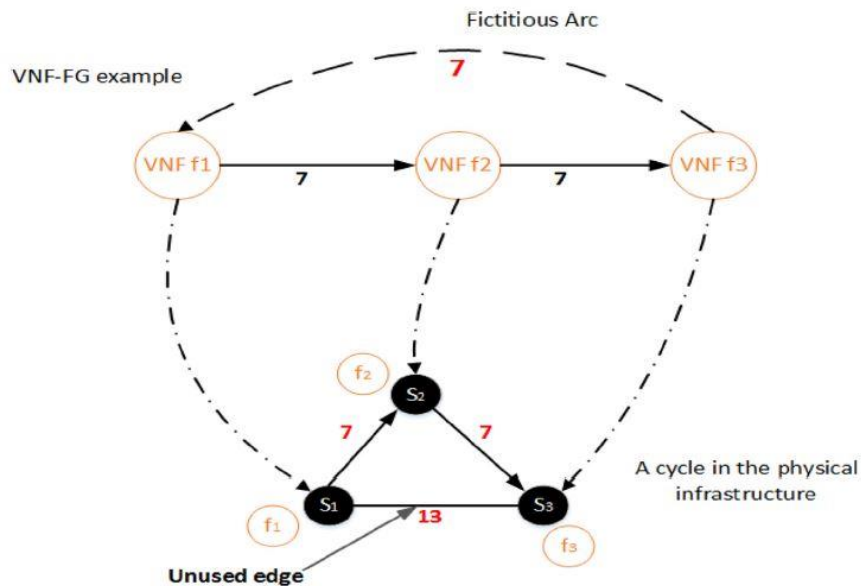


Figure 13: A Feasible Projection (VNF-CPP)

2. A strategy that is based on *matrix analysis* [38]: this method combines the products of matrices with a simple linear programming algorithm in order to determine the ideal way to guide the traffic flows on each VNF that has been set.

Before we can derive an alternative mathematical model and algorithms that are scalable and can quickly solve the VNF-CPP, we need to first define and characterize the service chain requests and the infrastructure that will host these chains. This will allow us to develop a model and algorithms that are both scalable and capable of solving the VNF-CPP. When



looking for the best possible placement in the VNF-CPP, we need to take into account a variety of different VNFs and chains, therefore we construct two variables that correspond to this requirement:  $C$  and  $K$ . These variables stand for the number of VNF chains in a request and the number of distinct VNFs that are involved, respectively.  $N\{s_1, s_2, \dots, s_N\}$  is the number of physical nodes or servers that are accessible for hosting virtual network functions (VNFs). We define  $N$  as the number of such nodes or servers for the infrastructure. We will make the assumption, by default, that VNFs from the same chain may be hosted by the same infrastructure nodes unless it is specifically stated otherwise. The algorithms may be modified to fulfil the requirements of any given need, including the separation of certain VNFs from a specific chain.

We employ a matrix-based approach to propose a new technique, in which we introduce a matrix  $M_1$  that reflects the presence (or absence) and the order of a VNF of a particular type in the requested chains. This matrix may be thought of as a representation of both the presence and absence of a VNF.

$M_1 = (a_{kc})$  for  $k = 1, 2, \dots, K$  and  $c = 1, 2, \dots, C$ ; with  $a_{kc}$  referring to the order of VNF  $k$  in chain  $c$  of the forwarding graph,  $a_{kc} = 0$  if chain  $c$  does not contain services of VNF types  $k$ .

Matrix  $M_1$  is stated as: if chain<sub>1</sub> requires VNF<sub>1</sub>, then  $M_1[VNF_1, chain_1] = a_{11}$ , otherwise  $M_1[VNF_1, chain_1] = 0$ .

$chain_1/chain_2$

$$M_1 = \begin{pmatrix} VNF_1 & 1 & 1 \\ VNF_2 & 0 & 2 \\ VNF_3 & 2 & 3 \end{pmatrix}$$

This first matrix, which represents the composition of the chains in terms of the VNF types, is combined with a second matrix  $M_2$ , which has rows that correspond to all of the involved chains  $C$ , in the forwarding graph requests, and columns that represent the  $N$  physical infrastructure nodes that will host the VNFs according to the placement optimization as described in the subsequent section. The values of the elements of matrix  $M_2$ , namely the  $(b_{cj})$  (for values of  $c = 1, 2, \dots, C$  and  $j = 1, 2, \dots, N$ ), represent the total number of chain flows that have been allocated to a server  $j$  (or that are scheduled to be handled by a server  $j$ ). When adding together all of the  $C$  chains, the total quantity of chain flows that may be allocated to a specific server cannot exceed the limit of what is considered to be its acceptable processing capabilities. Matrix  $M_2$  is expressed as:

$$M_2 = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1N} \\ \vdots & \vdots & \ddots & \vdots \\ b_{c1} & b_{c2} & \dots & b_{cN} \end{pmatrix}$$

The constraint equation:

$$\sum_{c=1}^c b_{cj} \leq Capa_j \quad \dots (5)$$

where the right hand side expression denotes the processing capability limit of the server  $j$ .

If  $req(k)$  is the necessary processing capacity of a VNF type  $k$ , then a server  $j$  will only be picked if its flow processing capability is at least equal to the required processing capability of this VNF. In other words, if  $req(k)$  is the needed processing capability of a VNF type  $k$ , then this server will be selected. This need may be represented by the constraint, which states that, for any virtual network function (VNF) of type  $k$  and a hosting node or server  $j$ , given by the equation:

$$\sum_{c=1}^c 1_{kc} b_{cj} \geq req(k) \quad \dots (6)$$

Where

$$1_{kc} = \begin{cases} 1, & \text{if } 0 < a_{kc} \\ 0, & \text{otherwise} \end{cases}$$

To find the near-optimal distribution of the flows reported in  $M_2$ , and taking into account constraints, we need to find the optimal (minimal) values of  $b_{cj}$  that satisfy them and minimize the physical resource consumption. Therefore, the mathematical model is:

$$\min Z = \sum_{c=1}^c b_j,$$

subject to:

$$\sum_{c=1}^c b_{cj} \leq Capa_j \text{ and}$$

$$\sum_{c=1}^C 1_{kc} b_{cj} \geq req(k), \text{ where } b_{cj} \in \mathbf{R}^+, \forall c = 1, 2, \dots, C.$$

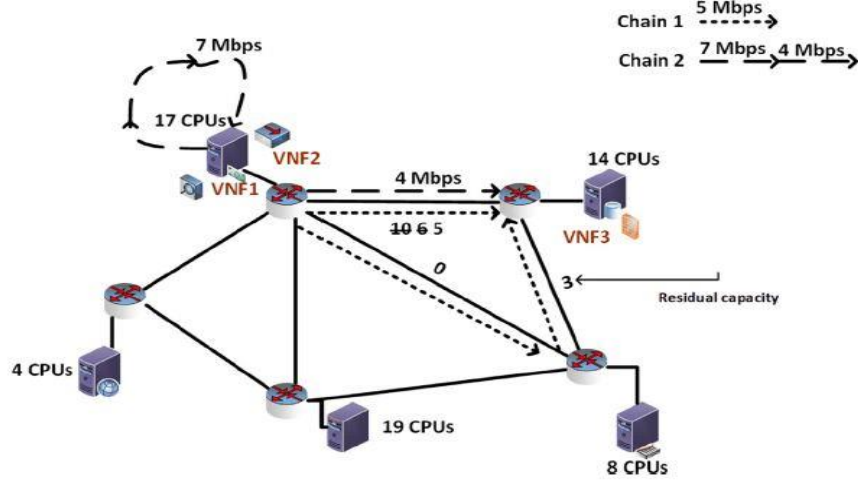


Figure 14: Post Matrix Based Approach (VNF-CPP)

3. A method predicated on the use of **multiple stage** [38] graphs, which is generated as a new extended multi-stage graph to represent the servers that are accessible for hosting the necessary VNFs and their respective interconnections. After that, the chains that make up the Forwarding network are arranged such that there is the greatest possible flow between the vertices of the various phases of this network.

In order to fill the graph MG, we make the assumption that each physical node (server) is capable of hosting a number of VNFs of various sorts, and that the combination of these VNFs leads in an amount of needed resources being made available on the physical node or server. If a particular node, for example say  $s_2$ , is capable of hosting both VNF<sub>2</sub> and VNF<sub>3</sub>, then this  $s_2$  will appear at the VNF<sub>2</sub> and VNF<sub>3</sub> levels of the multi-stage graph. This is significant from the perspective of the building of the multi-stage graph. A node (host) may appear at as many levels as it is capable of managing different VNF kinds. A rule is followed by the set of arcs in the multi-stage graph in order to guarantee that virtual network functions (VNFs) from the same service chain are allocated to hosting nodes without exceeding the available resources in the nodes that have been chosen:

In order to create a multi-stage graph, we make advantage of the fact that there is an arc with the direction  $(i_k, j_{k+1})$  connecting each pair of nodes  $i$  and  $j$  at levels  $k$  and  $k + 1$ . This multi-stage building rule guarantees that Virtual Network Functions (VNFs) belonging to the same service chain are allowed to share the same server to the extent that the capacity of that server permits. In addition, the weight of the arc  $(i_k, j_{k+1})$  is determined by the greatest flow (splittable flow) that can be routed in the physical substrate from server  $i_k$  to server  $j_{k+1}$ . This flow may be routed from the server  $i_k$  to server  $j_{k+1}$ .

We employ a flow marking method that begins at a hypothetical node  $T$  and ends at a fictitious sink  $S$ , in order to discover the ideal positions of the VNFs in the order that is specified in the chain request. This allows us to determine the optimal placements of the VNFs in the chain request. The technique for marking continues in the manner that is detailed below until it reaches node  $S$ . The figure illustrates the number of CPUs that are still available on the hosting nodes after allocations of CPUs (barred values) are removed from the available compute resources to indicate a transition from the current state to a new state (server  $s$  4 transitions from 8 CPUs down to only 2 available CPUs since VNF<sub>3</sub> consumes 6 CPUs in the requested VNF-FG). This transition allows for an indication of a change from the current state to the new state.

- Beginning at  $T$ , which has already been indicated, choose the node (on level  $K$ ) that has the lowest/highest possible processing capability;
- Let's say that a node with the index  $j_k$  is picked out and marked. It is necessary to update the server  $j$ 's capability at every level between 1 and  $K$  in which the letter  $j$  occurs. Explore all of the predecessors  $i_{k1}$  of  $j_k$ , and then designate one of the predecessors (or the predecessor) as the one that represents an extreme of an arc with the smallest amount of bandwidth that is now accessible. If we have more than one that has the same capacity, then we should choose the one that has an extremity node that has a minimal (or maximum, depending on the chosen criteria) processing capability. Consolidation is fostered by the lowest criteria, while load balance is protected by the maximum criterion.
- Carry out this procedure again and again until the node  $S$  is located and identified.

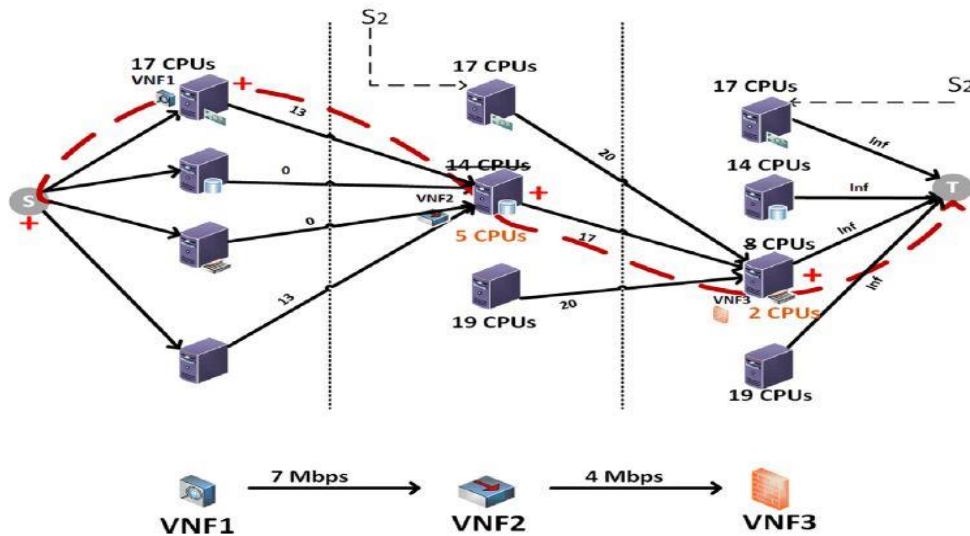


Figure 15: Multiple Stage (VNF-CPP)

## 2.3. Network Virtualization

The term "network virtualization" refers to a technique that permits the development of conceptually segregated network partitions across common physical network infrastructures. This allows different users to use the same infrastructure while maintaining their own distinct network identities. It is possible for several kinds of virtual networks to coexist at the same time over common infrastructures. In addition, network virtualization makes it possible to aggregate many resources and present the combined results as a single resource. This process is known as resource consolidation.

Network Virtualization [17](NV) is a technology that abstracts and decouples network resources from the underlying physical infrastructure, creating multiple virtual networks on top of a shared physical network. It provides a software-defined environment that allows for the creation, provisioning, and management of isolated and programmable virtual networks, independent of the physical network infrastructure. NV enables the logical partitioning of the network into multiple virtual segments, each with its own set of virtualized network services, configurations, and security policies. These virtual networks operate as if they were separate physical networks, providing enhanced flexibility, scalability, security, and efficiency.

### 2.3.1. Key Components of Network Virtualization:

*1. Hypervisor:* The hypervisor is a fundamental component of NV. It abstracts the underlying physical network and enables the creation and management of virtual networks. It allocates and isolates network resources, such as bandwidth, latency, and security policies, to different virtual networks.

*2. Virtual Switching:* Virtual switches, also known as vSwitches, play a crucial role in network virtualization. They facilitate traffic forwarding between virtual machines (VMs) and virtual networks within a hypervisor. Virtual switches provide capabilities such as network segmentation, VLAN tagging, and traffic isolation within the virtualized environment.

*3. Network Overlays:* Network overlays create virtual networks that operate on top of the physical infrastructure. They encapsulate virtual network traffic and enable seamless communication between virtual machines across different physical hosts. Network overlays employ tunneling protocols to encapsulate and transport virtual network traffic over the physical network infrastructure.

### 2.3.2. Benefits of Network Virtualization:

**1. Network Isolation:** Network virtualization enables the creation of isolated virtual networks, providing logical separation and isolation between different applications, tenants, or departments. This isolation enhances security, prevents unauthorized access, and ensures privacy and compliance.

**2. Agility and Scalability:** NV allows for the rapid creation, provisioning, and scaling of virtual networks, independent of the physical network infrastructure. It eliminates the need for

manual reconfiguration or hardware modifications, enabling businesses to adapt quickly to changing requirements and scale their networks as needed.

**3. Efficient Resource Utilization:** By consolidating multiple virtual networks on a shared physical infrastructure, NV optimizes resource utilization. It reduces the need for dedicated hardware for each network, leading to cost savings, improved energy efficiency, and better overall utilization of network resources.

**4. Simplified Management and Automation:** Network virtualization simplifies network management by providing centralized control and management capabilities. It enables administrators to provision, configure, and monitor virtual networks through a single management interface, leading to improved operational efficiency and automation.

**5. Application Mobility and Flexibility:** NV allows virtual machines and applications to be easily moved or migrated between physical hosts without disrupting network connectivity. This mobility and flexibility enable workload balancing, disaster recovery, and seamless application deployment across different environments.

### **2.3.3. The Industry Perspective of NV:**

#### **A. Network Device Virtualization:**

To begin, we will go through the virtualization technologies of the primary components that make up a network. These components are known as network interface cards (NICs, located at the network edge) and routers, which are located in the network core.

##### *1) NIC Virtualization:*

a) Software-Enabled NIC Virtualization: VMware [42], Microsoft, Citrix Systems (offering Xen [43]), and Oracle are among the most prominent commercial vendors of operating system (OS) virtualization solutions. NIC virtualization is accomplished via software. The ability to share network interface controller (NIC) hardware across different instances of the virtual operating system is an essential function of such systems. The virtual network interface card (vNIC), which is a software emulation of a physical network interface card and may be given its own, dedicated IP and MAC addresses, is the essential component of NIC virtualization. A client that intends to utilise a virtual NIC might be considered a vNIC client, as seen in the image. The vast majority of vNIC clients are either a virtual operating system, also known as a virtual machine (VM) in VMware or a domain in Xen, or an OS-level virtualization instance, such as a Solaris Zone. In VMware, a VM is referred to as a virtual machine, and in Xen, a domain. A software emulation of a physical switch is called a virtual switch (vSwitch). A virtual switch may not provide all of the functions that are available on a real switch. A vSwitch is a kind of network switch that can switch traffic, perform multiplexing and scheduling, and bridge virtual NICs to physical NICs if the situation calls for it. It is important not to mistake the idea of a "virtual link" with the "software-emulated links" that exist between vNIC and vSwitch. These links are generated by the programme. The computing power of the host system is the only thing that may restrict the bandwidth available across these mimicked networks. Although it is feasible to put an upper limit on

the speed of each simulated connection in order to preserve the overall balance of traffic, most implementations of NIC virtualization do not enable guaranteed bandwidth. This is despite the fact that it is possible to set an upper limit on the speed of each emulated link. The Sun Crossbow project is an important and prominent exception. Crossbow not only offers a more refined framework for the management of resources, but it also has a guaranteed bandwidth feature that enables virtual network interface cards (vNICs) to reserve a hardware traffic lane. Either the hypervisor, which is a programme that controls OS resources to enable virtualization, or the host operating system is where all of these abstractions are stored.

If the clients of the NIC virtualization are servers, such as web servers, DNS servers, or firewalls, then the NIC virtualization truly creates a virtual network that is comprised of virtual servers, virtual NICs, virtual switches, and virtual connections. This functionality is sometimes referred to as a "network-in-a-box." In this scenario, the virtual network is really a software-emulated network that creates traffic that is injected into the real world via a hardware NIC that is neither virtual nor emulated. In other words, the virtual network is a software simulation. Because to vSwitch, the interactions that take place between vNIC clients inside a single hypervisor are no longer visible to the outside world. As a result, more work is required to monitor or govern these connections. Network managers have an additional challenge as a result of the fact that various server virtualization technologies offer a variety of vSwitch flavours. Therefore, it is desirable to have all traffic routed to the physical switch even if it is meant to return to the same physical server. This is because having all traffic directed to the switch improves performance.

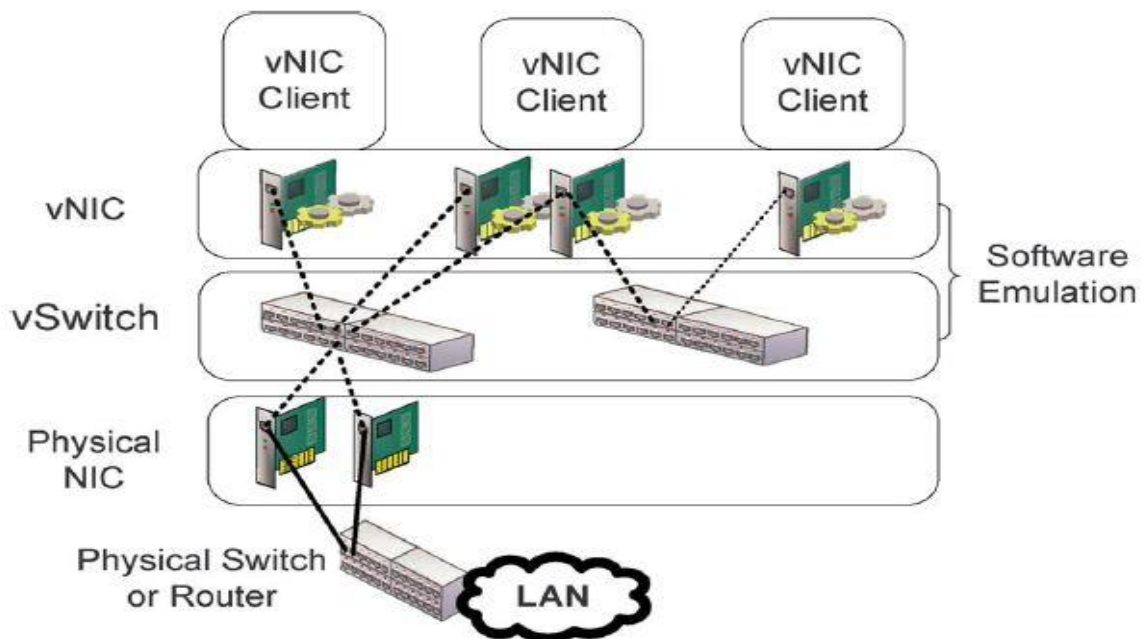


Figure 16: NIC Virtualization

b) Hardware-Assisted Network Interface Controller Virtualization: A regular NIC only offers one peripheral component interconnect express (PCIe) channel, which often becomes the I/O bottleneck in an aVMcentric data server. Hardware-assisted network interface controller [44] virtualization solves this problem. Single root I/O virtualization, often known as SR-IOV, is a hardware innovation that has the potential to construct numerous instances of PCI functions via the use of its Virtual Functions (VFs). Instead of connecting vNIC clients to a vSwitch, each virtual machine (VM) might be directly mapped to a VF in order to get access to NIC resources in a more expedient manner. In general, this module offers improved throughput and scalability, in addition to decreased CPU utilization.

## *2) Router Virtualization:*

In this part, we will refer to a network device that is capable of routing or switching activities as a "router" when we use the word "router" in its more generic sense. The three technologies for virtualizing routers that are discussed in the next subsections each represent a distinctive set of capabilities and are distinguished from one another in key ways.

a) Routers in Virtual Operating Systems: Software and hardware in a typical router are strongly tied together. Operating systems like Cisco's Internetwork Operating System (IOS), VxWorks, Linux, and BSD are common examples of those that have had their software altered to work more efficiently with hardware that has been specifically developed. Within a specialised operating system, router software, which also includes routing protocols, is executed.

Several attempts have been made to decouple the hardware and software components of routers. These efforts have been focused mostly on certain Linux distributions, such as Alpine Linux, Mikrotik RouterOS, Untangle, and Vyatta. The majority of these distributions are capable of operating on hardware with a typical X86 architecture, and they may be deployed in a virtual operating system such as VMware or Xen.

The phrase "virtual routers" may be used to refer to routers that are used in virtual operating systems. In this context, the word "virtual" refers to the fact that these routers share hardware resources with other instances of any virtual operating system that are running on the same physical computer. Examples of these resources include the central processing unit (CPU), memory, hard drive, and network interface card (NIC).



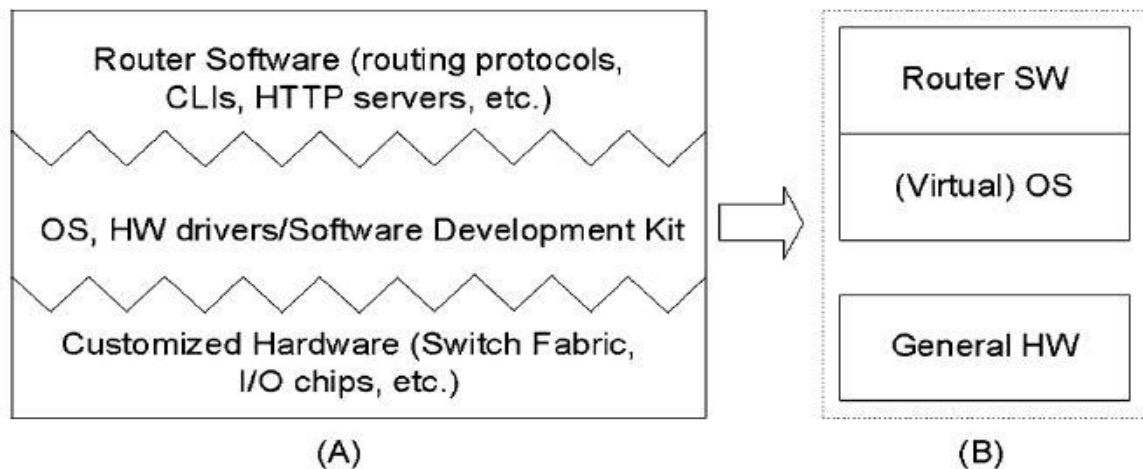


Figure 17: Router Virtualization

b) Router Control/Data Plane Virtualization: The routing table, which maps incoming packets to output ports, is a crucial part of routers and is handled by this kind of virtualization. It's possible that certain routers only have a single routing table that handles all of the packets, and that this routing table is only updated by a single process. There is the possibility that some routers will contain more than one routing table, with each table catering to a distinct routing context. It is possible for a single process or numerous processes (for example, one process for each routing table) to be responsible for maintaining the different routing tables. "virtual routing and forwarding" (VRF) is the name given to this particular technological advancement. The Routing Information Base (RIB) is virtualized as numerous routing tables in the control plane, while the Forwarding Information Base (FIB) is virtualized as multiple forwarding tables in the data plane.

Although we may consider it a type of control-plane and data-plane virtualization, technically speaking, it is not a virtualized device.

Routers with a higher level of sophistication may enable (logically) different routing protocols, settings, and other aspects in addition to having separate routing tables. In routers like these, a "virtual routing instance" is the precise logical combination of a configuration, routing protocol, and routing table [44] . Virtual routing instances are used to route data more efficiently. This kind of virtual routing instance has been referred to by a variety of names, including "virtual router," "logical router," and "routing context."

c) Hardware-Partitioned Router: Routers [44] that enable hardware partitioning may host several routing instances in a single device. Juniper Networks refers to them as "protected system domains," whereas Cisco Systems refers to them as "logical routers." Hardware-partitioned routers are supported by Cisco Systems. Hardware-partitioned routers are often used at the Points of Presence (PoP) of network carriers. This is done in order to conserve space and power while also lowering the cost of management. Because the hardware is

normally partitioned on a line card-by-line card basis, this technology may be conceptualized as a router on a line card basis.

## **B. Link Virtualization**

The word "virtual links" may refer to a variety of different things depending on the context in which it is used, and link virtualization technologies are responsible for their creation. In this part, we will go through the numerous interpretations that have been given, as well as reveal the multiple levels of link virtualization.

*1) Multiplexing of the channels via physical means:* When talking about link virtualization, the first thing that has to be defined is precisely what makes up a "link." This is the problem that comes up immediately. It's possible that multiplexing and link virtualization are the same thing if the connection in question is a physical media. It is possible for a physical media to be either wired (such as fibre or copper cable) or wireless (such as the wireless spectrum). Multiplexing various communication channels across a single physical media is often accomplished with the assistance of multiplexing technologies such as time division multiplexing (TDM), frequency division multiplexing (FDM), and code division multiple access (CDMA). We point out that on a fundamental level, multiplexing performs a function that is very similar to virtualization. The physical medium is split into distinct channels, and the sender and receiver are given the illusion that they own the physical medium. Although multiplexing is not generally considered to be a link virtualization technology, we point out that on a fundamental level, it performs this function very similarly. As a result, virtualization may be thought of as an expansion on the concept of multiplexing.

*2) Virtualization of Bandwidth:* The term "link virtualization" refers to technologies in this context that combine the bandwidth of separate channels to construct virtual connections. Bandwidth virtualization is a subset of link virtualization.

a) Circuit: When making a standard telephone call, a circuit is created for a call by stringing together a number of channels (time slots on a physical connection) along the links of the route that connects the two parties involved in the conversation. The communication takes place within these time blocks, giving the appearance to the people involved that they are using a dedicated queue. In this instance, the stringing together of several time intervals produces a virtual connection.

b) Reverse Multiplexing: Developments in optical technology have made it feasible to combine bandwidth in order to build flexible services (i.e., pools of bandwidth) that are independent of the capacity of the physical connections and devices that are underlying the network. These types of technology for reverse multiplexing may work at the granularity of either sub-wavelength or full-wavelength rates.

For example, SONET virtual concatenation joins several STS-frames together to create channels with flexible bandwidth that cannot be supported by the traditional SONET bandwidth hierarchy. The combined STS-frames may be contiguous or non-contiguous frames on the same SONET link, or they may even belong to different SONET links.

Additionally, the combined STS-frames may be able to span multiple SONET links. In a similar fashion, one may combine four (or ten) 10 G channels (wavelengths) to produce a single 40 G channel (or 100 G channel). Virtual connections are the terms used to refer to the channels that are produced by technologies such as reverse multiplexing. Techniques of a similar kind for virtualizing optical bandwidth may also be used in the provisioning of Layer 1 virtual private networks (L1 VPNs). Services are firmly tied to specific physical devices in an optical network that operates in the conventional fashion. Bandwidth virtualization makes it possible to decouple services from specific devices by making them dependent instead on the virtualized bandwidth itself.

*3. Data Path Virtualization:* Data Path Virtualization refers to the methods that do not modify the channel itself but instead manipulate the data that is transported on this channel in the form of packets. In this scenario, a data channel with certain characteristics is equivalent to a virtual connection that has those characteristics.

This kind of virtual connection does not rely (directly) on the physical attributes of the links (such their bandwidth), but rather it is supplied by the nodes themselves. To be more specific, nodes make use of a variety of technologies in order to route data via these virtual connections (data routes); the following paragraphs will examine two of the most common technologies.

a) Labels: Labels, which may also be referred to as tags, IDs, or other similar terms, are found in certain fields of the header of a packet and serve as identification and sharing methods. Nodes are aware of labels, which allows them to guide traffic in the appropriate direction. This is a crucial aspect since nodes are responsible for enabling the virtual connection, also known as the data path.

Tags for 802.1q virtual local area networks (VLANs) make it possible for several VLANs to share a same physical media while being conceptually distinct. Sharing is the primary function of VLAN tags, which may also be used to differentiate between data coming from various VLANs. In addition to this, they are used in order to assist in the formation of data routes for the broadcasting domain. The labels that are used in technologies such as asynchronous transfer mode (ATM), frame relay (FR), and multiprotocol label switching (MPLS) are also used to designate the direction that data packets travel. Virtual circuits are the terms used to refer to these data pathways. In the second section. The term "circuit" refers to the fact that time slots are chained together beginning to finish. When used in this context, the word "virtual" refers to connections that are concatenated rather than actual time periods.

b) Tunnels and Encapsulation: Virtual (logical) linkages may be established between network devices that are not physically close via the use of tunnels, which often make use of encapsulation methods. For instance, tunnels may be used to provide the impression to certain protocols that are operating on a network device that this device has a direct connection to another device, even while there is no actual physical link between the two devices. This can be done even when there is no connectivity between the two devices.

Generic route encapsulation, also known as GRE tunnels, Internet Protocol security, also known as IPsec tunnels, GPRS Tunneling Protocol, or GTP tunnels, and MPLS label switched path, or LSP tunnels, are all examples of prominent tunneling technologies. Tunnels, which are essentially the same thing as overlay connections, are the basic building components that constitute overlay networks.

### C. Virtual Networks

The terms "network-in-a-box" and "bandwidth-virtualized network" have been used to describe the two varieties of virtual networks that we have come across so far. In this part, we will discuss a number of other network virtualization technologies, as well as explain how these technologies connect to one another and how they stack up against one another. Only commercial technologies provided by the industry are taken into consideration here, since this is in line with the general concept of this part. In the next section, we will discuss virtual networks, which are the subject of study conducted by academics.

*1) Overlay Networks:* A network that is established on top of another network, often using tunneling and encapsulation technologies, is referred to as an overlay network. [45] The capability of overlay networks to install new network services in a cost-effective manner by using pre-existing network infrastructure is one of the primary reasons for their growing popularity. Take for example a physical network, in which three nodes are linked by two connections. Imagine for a moment that there is a need to put in place a new network service between two of the nodes, such as a new routing protocol. By using tunnels to link the nodes, only the two nodes that are directly involved in providing the service will need to be updated. On the other hand, the third node does not require any modifications; it will continue to pass data between and, but it will be unaware of the newly introduced service. In the overlay network, the tunnel creates the illusion for the nodes and links that they are linked to each other through link. However, the nodes and links are unaware of the existence of the other node. Numerous overlay networks have been built in order to extend the capabilities of preexisting infrastructure and so enable the provision of brand new services. Internet access networks, such as a digital subscriber line (DSL) or cable network that is superimposed on an existing public switched telephone network (PSTN) or cable TV infrastructure, are some examples. Other examples are the MBone and 6Bone, which add multicast and IPv6 capabilities to the Internet, respectively. In an overlay network, the topology of the network is what is virtualized, and the virtual network is formed by all of the nodes that are aware of any new services.

*2) Virtual Private Networks:* A virtual private network, often known as a VPN is a collection of private networks that are able to communicate with one another but are shielded from access to public networks like the Internet. Individuals who work from home generally use a virtual private network (VPN) [46] [47] in order to get access to the internal network of their employer. By way of illustration, businesses utilize VPNs in order to link their offices that are located in various parts of the world.

Layer 2 (L2) and Layer 3 (L3) VPN technologies are mature and widely deployed. In a L2 VPN, the VPN provider's network is virtualized as a Layer 2 switch, and the responsibility for developing the client sites' own routing infrastructure falls on the customer sites themselves. On the other hand, an L3 virtual private network (VPN) transforms the network of the service provider into a Layer 3 router. The virtual private network (VPN) technology known as Layer 1 (L1) has been going through a fast process of standardization since 2005. The different networks that the L2/L3 VPN and the L1 VPN technologies run on is the primary distinction between the two types of VPN technology. L1/L2 VPNs are often constructed on an IP/MPLS BGP core, but L1 VPNs are primarily intended to function on TDM networks, such as SONET/SDH or wavelength division multiplexing (WDM) optical networks. L2/L3 VPNs are typically built on an IP/MPLS BGP core. Customers, or the edge routers of private networks, will have the ability to make a request for Layer 1 data channels over the provider's network while using L1 VPN. These data paths are what will be utilized to link customer sites to one another. However, a higher layer protocol is responsible for the provisioning of an L1 VPN. This means that protocols such as generalized MPLS (GMPLS) are the ones responsible for determining and establishing the data route that was requested. The L1 VPN technology is still in the process of being developed and is very sometimes used in operational settings.



Figure 18: Virtual Private Network(VPN)

*3)Virtual Sharing Networks:* We refer to technologies that facilitate the sharing of physical resources across many network instances as "virtual sharing networks" (VSN), and we use the term "virtual sharing network" (VSN) to describe these technologies. These technologies also provide clear distinction between the various network instances. In the business world, a network instance of this kind can simply be referred to as a "virtual network", which contributes even further to the oversimplification of the latter word. We came up with the acronym VSN in order to avoid any confusion and to separate this idea from overlay networks and VPNs, which were both previously covered in this article. Virtual LANs use the same physical network infrastructure as traditional LANs, but their broadcast domain boundaries ensure that they operate independently of one another. Sharing and segmentation are the two most important characteristics of a VSN, which is a generalization

of the VLAN idea to apply to a wider network. There is a possibility that the access points (wireless hot spots, LAN) for the employee network, the visitor network, and the administrator network will all be shared. Access including physical switches and routers, as well as servers of the VPN-extended VSN. On the other hand, these networks are also correctly divided as virtual networks, each with its own unique set of access rights. On the other hand, only the administrator network may provide the ability to configure physical devices or to access certain parts of the infrastructure, including the security surveillance network and the building automation system. For instance, all virtual networks are able to access the Internet, but only the employee and administrator networks may access email servers, Intranet Web servers, and certain document and database servers. Provisioning a virtual network for each user group is a better answer in this scenario than constructing a separate physical network for visitors, workers, and administrators. This is true in terms of cost, efficiency, and the amount of effort required for maintenance, among other considerations. The most important need is to make sure that different virtual networks may use the same physical infrastructure while yet maintaining the appropriate level of segmentation.

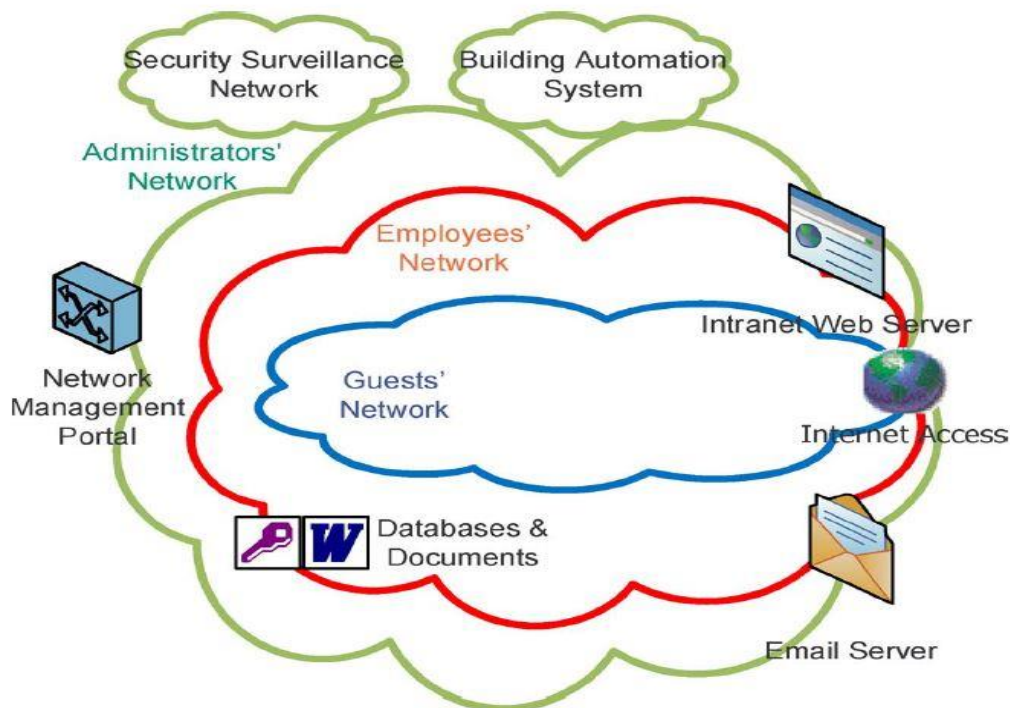
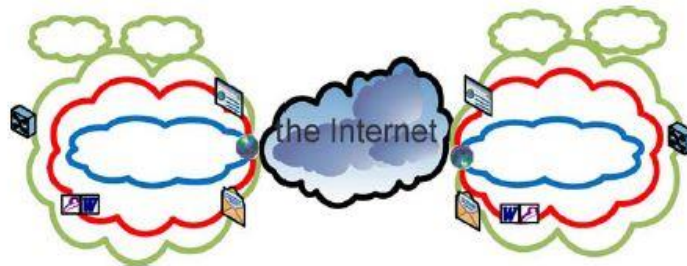


Figure 19: Virtual Sharing Network (VSN)

The user group-specific virtual networks of the VSN may be expanded over many physical locations when an AVSN is combined with a VPN, as was previously mentioned. This allows the user group-specific virtual networks to be extended throughout the AVSN. In this

scenario, the term "virtual" refers to two different things: first, from the perspective of the VSN, each virtual network (user group-specific) shares the same physical resources with other VSN virtual networks; [48] second, from the perspective of the VPN, the VSN itself forms a virtual network because several peer VSN networks at geographically dispersed locations are interconnected transparently; third, the term "virtual" refers to both of these concepts.



*Figure 20: VPN extends VSN*

## 2.4. Graph Database

A graph database [49] is a type of database that uses graph structures to store and represent data. It is designed to efficiently store and query highly interconnected data. In a graph database, data is organized as nodes (also known as vertices) and relationships (also known as edges) between those nodes. In a graph database, nodes represent entities such as people, places, or objects, and relationships represent the connections or associations between those entities. For example, in a social network, nodes can represent users, and relationships can represent friendships between users.

Graph databases provide a flexible data model that can handle complex relationships and interconnected data more effectively than traditional relational databases. They excel at handling queries that involve traversing relationships and finding patterns or paths between nodes.

Some key characteristics of graph databases include:

1. Graph data model: Graph databases have a native graph data model, which means the data is represented as nodes and relationships.
2. Relationships as first-class citizens: Relationships are as important as nodes in a graph database and can have properties of their own. This allows for rich representations of connections and the ability to store additional information about relationships.
3. Traversal and pattern matching: Graph databases offer powerful query languages that enable efficient traversal of the graph and pattern matching across nodes and relationships. This makes it easier to find and analyze complex relationships and patterns in the data.
4. Scalability and performance: Graph databases are designed to scale horizontally and can handle large-scale graphs efficiently. They are optimized for queries that involve traversing relationships, which makes them well-suited for use cases such as social networks, recommendation engines, fraud detection, and knowledge graphs.

Overall, graph databases provide a flexible and efficient way to model, store, and query highly connected data, making them particularly useful for applications that require complex relationship analysis.

### 2.4.1. Graph Database and Graph Theory

Graph databases are closely related to graph theory, as they both deal with the representation and analysis of graphs. Graph theory is a branch of mathematics that studies the properties and relationships of graphs, which are mathematical structures consisting of nodes and edges. [50] [51]

Graph theory provides the foundation for understanding and analyzing graphs in various domains, including computer science, network analysis, operations research, and more. It defines fundamental concepts and algorithms for working with graphs, such as:



**1. Nodes and edges:** Graph theory defines the basic elements of a graph, which are nodes (vertices) and edges (arcs or edges). Nodes represent entities, and edges represent connections or relationships between those entities.

**2. Graph properties:** Graph theory defines various properties of graphs, such as the degree of a node (number of edges connected to a node), connectivity (how well connected the graph is), and path finding algorithms (finding the shortest path between nodes).

**3. Graph algorithms:** Graph theory provides algorithms for analyzing graphs, such as breadth-first search (BFS) and depth-first search (DFS) for traversing graphs, Dijkstra's algorithm for finding the shortest path, and various clustering and centrality algorithms for analyzing the structure of a graph.

Graph databases leverage the principles and concepts from graph theory to store, represent, and query highly connected data. They utilize the graph data model, where nodes represent entities and relationships represent connections between those entities. By doing so, graph databases can efficiently handle complex relationships and enable powerful graph-based queries.

Graph theory provides the theoretical foundation for understanding the structure and behavior of graphs, while graph databases provide a practical implementation of graph theory concepts for storing and querying real-world data.

#### **2.4.2. Using Graph Database For Handling Network Infrastructure**

Graph databases are well-suited for modeling network infrastructure due to the inherent graph-like structure of networks. Here are several reasons why graph databases are beneficial in this context:

1. Relationship-centric modeling: Network infrastructure consists of interconnected components, such as devices, servers, switches, and routers. Graph databases excel at representing and managing these relationships. By modeling network infrastructure as a graph, you can capture the connections, dependencies, and hierarchies between various components accurately.

2. Flexibility and scalability: Graph databases offer a flexible data model that can accommodate changes and additions to the network infrastructure. As networks grow and evolve, new devices, connections, and configurations can be easily added to the graph without requiring significant schema changes. Graph databases can also handle large-scale networks with millions of nodes and relationships efficiently.

3. Efficient querying and traversal: Graph databases provide powerful query languages and traversal algorithms that allow you to navigate and explore the network infrastructure efficiently. You can easily find paths between devices, analyze dependencies, detect bottlenecks, and perform other complex network analysis tasks. Traversal-based queries can help identify network paths, optimize routing, and uncover potential performance issues.

4. Real-time monitoring and analysis: Graph databases can be integrated with real-time monitoring systems to capture and store network events, such as device status, performance metrics, and network traffic. By continuously updating the graph with real-time data, you can analyze and visualize the network in real-time, detect anomalies, monitor network health, and perform predictive analysis.

5. Network visualization: Graph databases offer rich visualization capabilities that can help network administrators understand and communicate the network structure effectively. Graph visualizations can provide an intuitive representation of network components, their relationships, and the overall network topology, aiding in troubleshooting, planning, and decision-making processes.

6. Security and access control: Graph databases can enforce security measures and access controls at the node and relationship levels. This allows us to define permissions and restrictions for different users or groups, ensuring that only authorized personnel can access and modify specific parts of the network infrastructure graph.

In summary, graph databases provide a natural and effective way to model, store, and analyze network infrastructure. They offer the flexibility, scalability, and powerful querying capabilities required to manage and understand complex network relationships, making them an excellent choice for network infrastructure modeling and management.

### **2.4.3. Difference with Relational Model**

Relational databases and graph databases differ in their data models, query languages, and use cases. Here are some key differences between the two:

#### **Data Model:**

- Relational Database: Relational databases use a tabular data model where data is organized into tables with predefined schemas. Each table consists of rows (tuples) and columns (attributes). Relationships between tables are established using foreign keys.
- Graph Database: Graph databases use a graph data model where data is represented as nodes (vertices) and relationships (edges). Nodes represent entities, and relationships represent connections between entities. Both nodes and relationships can have properties associated with them.

#### **Query Language:**

- Relational Database: Relational databases use Structured Query Language (SQL) for querying and manipulating data. SQL is primarily based on set theory and operates on tables using declarative queries.
- Graph Database: Graph databases typically provide specialized query languages, such as Cypher (used by Neo4j) or GraphQL, which are specifically designed to navigate and traverse the graph data model. These query languages allow for expressive and efficient querying of relationships and patterns within the graph.

**Performance:**

- Relational Database: Relational databases are optimized for handling large volumes of structured data and performing complex aggregations and joins across multiple tables. They excel at transactional consistency and can handle ACID (Atomicity, Consistency, Isolation, Durability) properties. [52]
- Graph Database: Graph databases are optimized for querying and traversing highly connected data. [53] They excel at queries that involve navigating relationships, finding patterns, and calculating graph-based metrics. Graph databases are often used in applications where the relationships between data points are critical, such as social networks, recommendation systems, fraud detection, and knowledge graphs.

**Use Cases:**

- Relational Database: Relational databases are commonly used in applications where structured data consistency, transactional integrity, and complex data relationships are important. They are suitable for applications like e-commerce, inventory management, financial systems, and most traditional business applications.
- Graph Database: Graph databases are well-suited for applications that involve highly connected data, complex relationships, and pattern analysis. They are often used in social networks, recommendation systems, fraud detection, network analysis, knowledge graphs, and any domain where relationships between entities are crucial.

In summary, while relational databases excel at structured data and complex joins, graph databases provide a powerful way to model and query highly interconnected data. The choice between the two depends on the nature of your data, the types of queries you need to perform, and the specific requirements of your application.

A Neo4J Graph Database [54] image is given in the next page. (Figure 21).

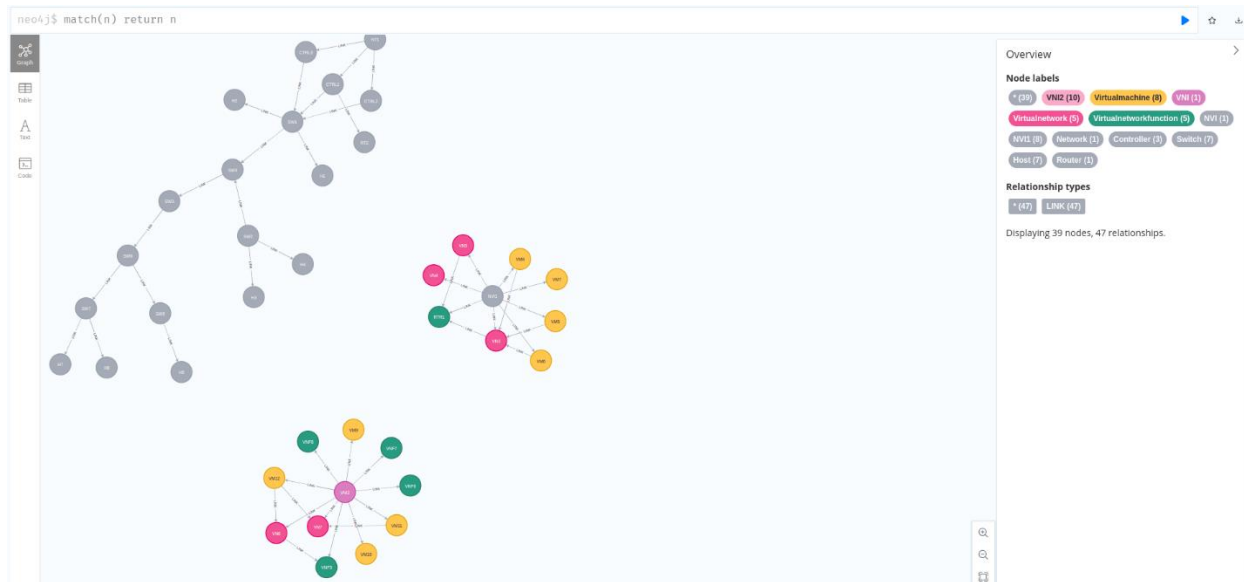


Figure 21: Neo4J Graph Database

#### 2.4.4. Conversion of Relational Database to Graph Database

Compared with relational databases and NoSQL databases, graph databases have obvious advantages in relational query and analysis and mining of data. Such related data sets mostly appear in social networks, recommendation systems, network and data center management, and logistics management. When traditional relational databases and graph databases process this type of data separately, the analysis and processing efficiency of graph databases is several orders of magnitude higher than that of relational databases. At this time, it is necessary to deal with such a problem, that is, how to convert the tables in the data set under the relational model and the connections between the tables into nodes and corresponding relationships in the graph database, and to migrate the related data. [55]

##### A. Mapping Rules for nodes and relationships

###### 1) Node Conversion

To begin, a fundamental structure of the table may be plainly achieved by drawing an E-R diagram. The primary key in the diagram is used to name the node, and all candidate codes are used after that as the attributes of this node. Afterwards, a complete list of candidate codes is employed. In addition, some of the data tables will have both business primary keys and logical primary keys as their primary keys. First, delete the logical primary key, then maintain the primary key that corresponds to the business, and last, name the node using the primary key that corresponds to the company. Enhancing the effectiveness of queries may also be accomplished by simultaneously adding an index for the main key of the company. There are composite primary keys in some of the data tables, and the technique of conversion is the same as what was described earlier. Last but not least, in the case of non-canonical data tables that do not have a primary key, a combination of many fields is used

for the purpose of conversion and naming. The following is an explanation of how the node conversion method works:

**Algorithm:**

**Input:** Table name (table I, table II, ...) and the field name of the corresponding table (table I.field I, table II.fieldI, ...)

**Output:** A set of nodes with labels (A1, A2, ...) and respective node properties (A1.property1, A2.property1,...)

```

1.  $i \leftarrow 1, j \leftarrow 1$ 

2. for table[i]  $\neq \emptyset$  do
    for table[i].field[j]  $\neq \emptyset$  do
         $A[i][j] \leftarrow \text{table } i \text{ and table } i.\text{field } j$ 
         $j \leftarrow j + 1$ 
     $i \leftarrow i + 1$ 

3. return A[i][j]
```

*2) Relationship Conversion*

There are two different methods that the connection may be built in the graph data model. The first is made up of many tables that each contain the same information. Direct construction is the name given to this procedure. The second method is made up of foreign key combinations in a number of distinct tables. In certain circles, this technique is also referred to as the indirect building approach. In the first step of the process, an intermediate connection table is constructed using the tables' respective foreign keys. Each field in the connection table becomes an attribute of the relationship once it is used to establish a directed link between two or more nodes, which is the primary function of the connection table. The following is an example of the relationship conversion algorithm:

**Algorithm:**

**Input:** Table name (table I, table II, ...) and the field name of the corresponding table (table I.field I, table II.fieldI, ...)

**Output:** Relationship between nodes (*Relationship*:  $A1 \rightarrow [\dots] \rightarrow A2$ )

```

1.  $i \leftarrow 1, j \leftarrow 1, m \leftarrow 1, n \leftarrow 1, \text{tag} \leftarrow 0$ 

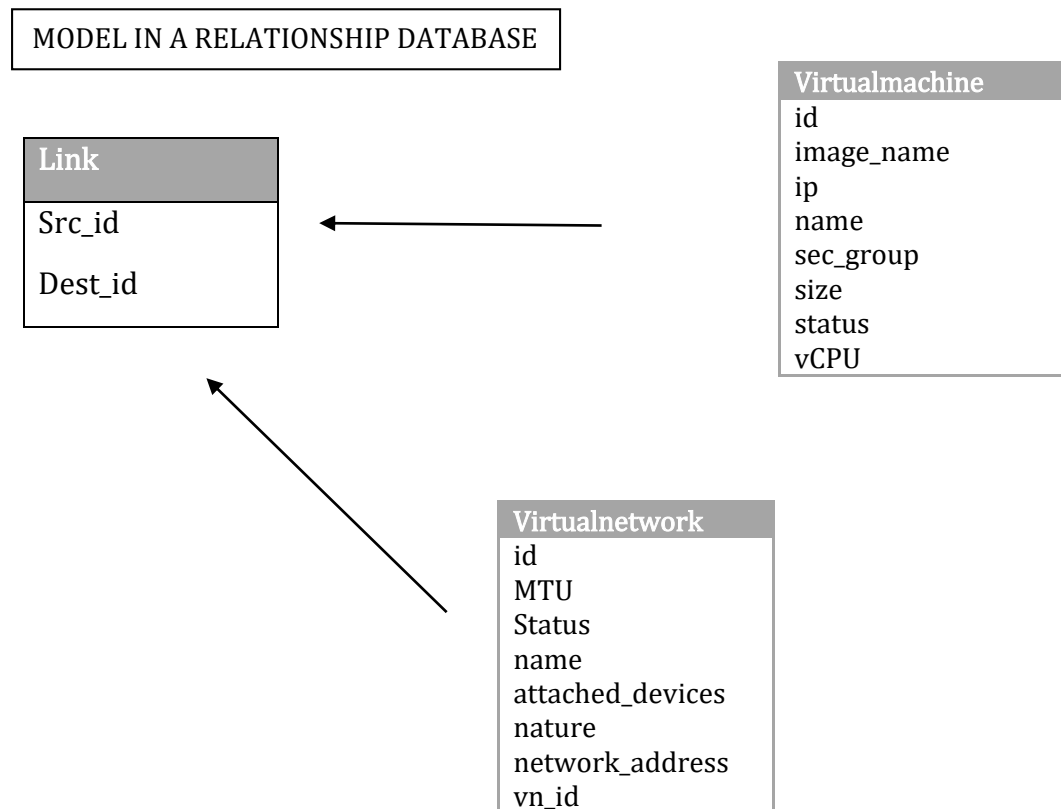
2. for table [i]  $\neq \emptyset$  and table[j]  $\neq \emptyset$  do
    for table[i].field[m]  $\neq \emptyset$  and table[j].field[n]  $\neq \emptyset$  do
```

```

        if table[i].field[m] = table[j].field[n] then
            tag ← 1
            break

        m ← m + 1
        n ← n + 1
    i ← i + 1
    j ← j + 1
3. if tag = 1 then
    use direct construction
    return relationship
else
    use indirect construction
    return relationship

```



## Corresponding Graph Database:

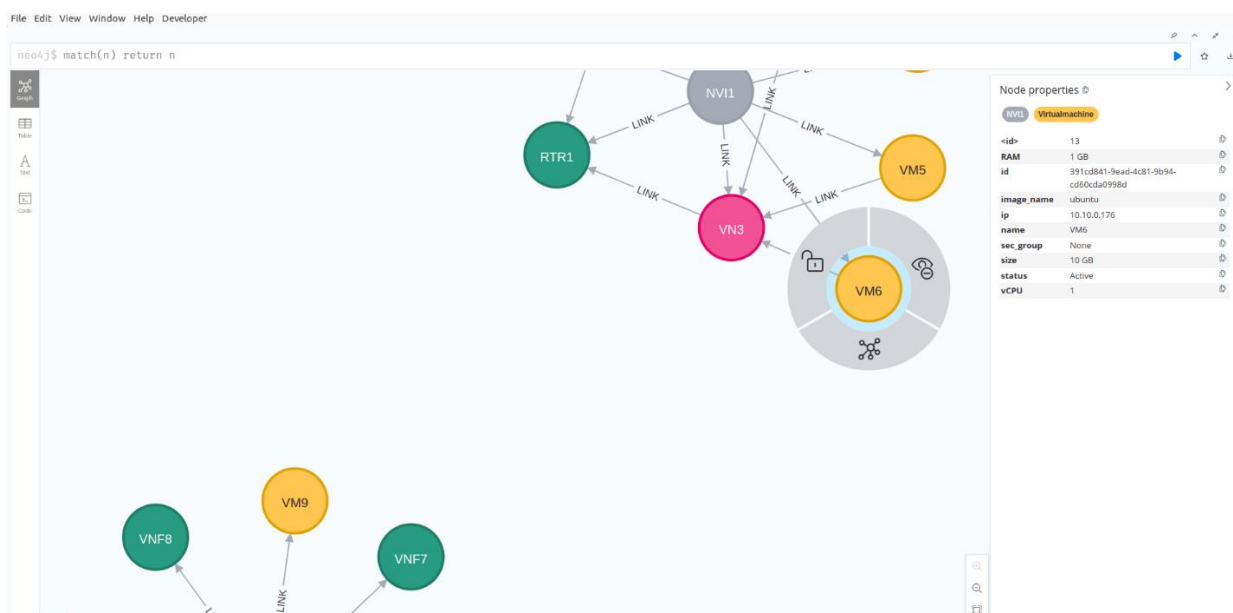


Figure 22: Virtual machine in a Graph Database

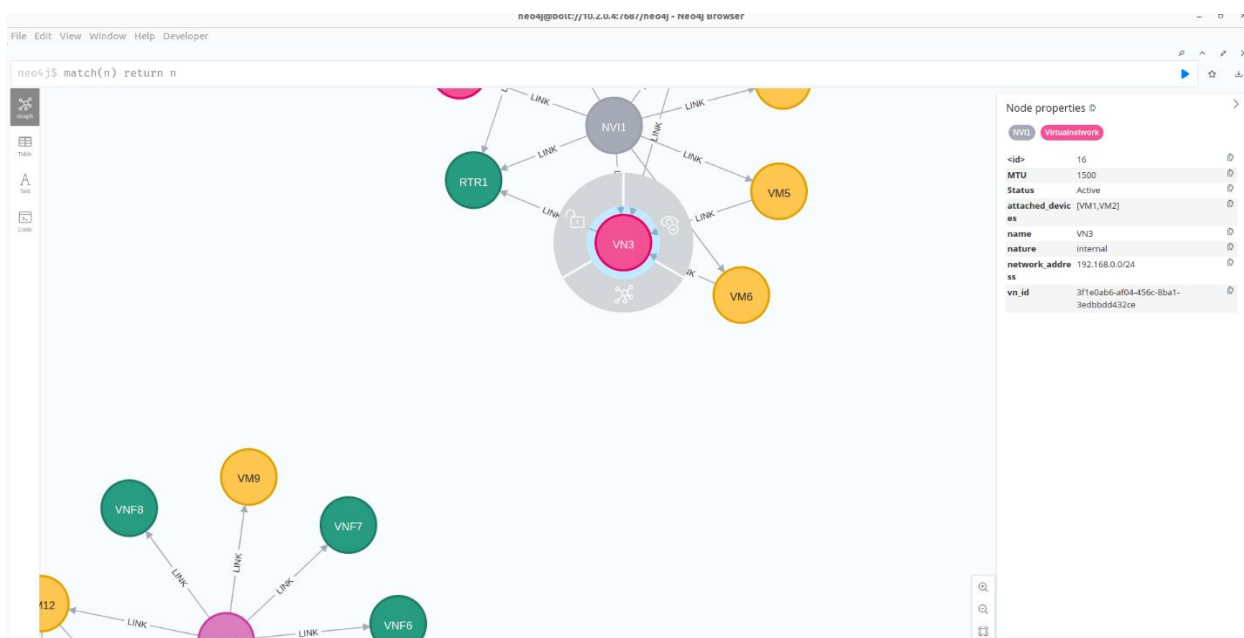


Figure 23: Virtual network in a Graph Database

### B. Constraint handling

Relational data tables will add certain restrictive criteria to particular fields, such as initial default values, uniqueness of field values, and auto-increment fields situated on the logical primary key. These constraints will limit the data that may be entered into those fields.

Constraint type	Approach
With initial default value	Delete the default value
Field Uniqueness	Use constraint statements (#)
Auto-increment field	Delete auto increment field

#

*CREATE CONSTRAINT ON (variable name: label name) ASSERT variable name.Attribute name IS UNIQUE*

In addition, we need to establish distinct nodes for some irregular data tables that include duplicate entries since such tables contain them. Last but not least, the processing of data types and null values is taken care of by the CQL (Cypher Query Language) function that is officially supplied by graph databases.

### C. Data Migration

The fundamental foundation of the graph data model has been constructed on the basis of the conversion of the data. The subsequent activity is to transmit certain data. The typical way of data import, which is the CSV file import technique officially offered by the graph database, is not the same as the data migration method that is being presented in this part since it is distinct from that approach. However, the CSV import technique is unable to filter some unlawful data, which compromises the authenticity and accuracy of the data that is imported using this approach. The ability to convey the data precisely and error-free is one of the benefits of using this technology.

The requirements for data conversion dictate that data migration must be performed in two stages: the first stage involves the extraction of metadata and table data, and the second stage involves the import of the data that was extracted into the graph database. The term "data extraction" refers to the process of using Python technology to crawl the relational database in order to get the fields and records that are necessary for successful metadata. The importation of data is accomplished on the basis of the existing graph data model, and the extracted valid data is used as the input terminal in accordance with the established graph data model. The data is then entered into the API that is made available by the graph database.



## 2.5. Path Analysis

Neo4j is a powerful graph database that provides various algorithms to find paths between nodes efficiently. These algorithms help in traversing and analyzing the graph structure to discover relationships and optimize queries. Here are some common path algorithms available in Neo4j:

1. **Shortest Path Algorithm:** The Shortest Path algorithm finds the shortest path between two nodes based on the sum of relationship weights. It uses the Dijkstra algorithm or the A\* algorithm to efficiently find the shortest path.

2. **All Shortest Paths Algorithm:** The All Shortest Paths algorithm finds all possible shortest paths between two nodes in the graph. It provides a comprehensive view of different paths with equal shortest distances.

3. **A\* Algorithm:** The A\* algorithm is an informed search algorithm that uses heuristics to guide the search process. It considers both the distance from the starting node and an estimated cost to the target node, enabling efficient path finding.

4. **Breadth-First Search (BFS):** BFS is an algorithm that explores all the neighbors of a node before moving on to the next level of nodes. It is useful for finding the shortest path in an unweighted graph or exploring the graph in a breadth-first manner.

5. **Depth-First Search (DFS):** DFS is an algorithm that explores as far as possible along each branch before backtracking. It is useful for traversing the entire graph or finding paths in a non-weighted graph.

6. **Bidirectional BFS:** Bidirectional BFS is an optimized version of BFS that simultaneously performs two BFS searches from both the starting and ending nodes. It reduces the search space and improves performance, especially for finding shortest paths.

Neo4j provides these algorithms through the Graph Data Science Library (GDS), which offers an extensive collection of graph algorithms optimized for Neo4j's graph processing capabilities. The algorithms are implemented in a parallel and distributed manner to handle large-scale graphs efficiently.

Using the Cypher query language or the GDS library, we can leverage these path algorithms to find shortest paths, explore relationships, perform graph analysis, and gain insights from the graph data stored in Neo4j.

## 2.6. Cost Analysis

Performing a cost analysis of a network based on bandwidth, jitter, and latency involves evaluating the impact of these factors on network performance and the associated financial implications. Here's a general approach to conducting the analysis:

1. Define Requirements: Determine the specific requirements of the network in terms of bandwidth, jitter, and latency. This may involve understanding the expected traffic volume, application needs, and user expectations.

2. Identify Cost Factors:

- Bandwidth: Calculate the cost of acquiring and maintaining the required bandwidth capacity. Consider the expenses related to network infrastructure, equipment, internet service providers (ISPs), or dedicated lines.

- Jitter: Assess the impact of jitter on applications or services that are sensitive to variations in latency. Determine if additional measures like quality of service (QoS) mechanisms or traffic prioritization are necessary, and evaluate the associated costs.

- Latency: Analyze the financial implications of latency on network performance. For example, high latency may affect real-time applications, customer satisfaction, or productivity, leading to potential revenue losses or increased operational costs.

3. Quantify Impact:

- Bandwidth: Estimate the potential consequences of insufficient bandwidth, such as reduced network performance, increased data transfer times, or downtime. Quantify the impact on productivity, customer satisfaction, or revenue generation.

- Jitter: Assess how jitter affects specific applications or services. Determine if jitter causes service disruptions, poor user experiences, or additional support costs. Quantify the impact on customer satisfaction, service-level agreements (SLAs), or operational efficiency.

- Latency: Evaluate the impact of latency on critical applications or services. Consider the potential loss in revenue, increased customer churn, or decreased employee productivity. Quantify the impact on response times, transaction speed, or overall business operations.

4. Compare Costs:

- Consider different scenarios: Analyze multiple options for network configuration, bandwidth capacity, QoS mechanisms, or latency reduction techniques. Compare the costs associated with each scenario.

- Factor in operational costs: Include ongoing operational expenses, such as maintenance, support, or upgrades, when evaluating the overall cost of the network.

- Consider opportunity costs: Assess the potential revenue gains or competitive advantages that can be achieved by investing in higher bandwidth, reducing jitter, or minimizing latency.

## 5. Decision-making:

- Evaluate cost-benefit trade-offs: Compare the financial implications of different network configurations or improvements. Consider the impact on network performance, user experience, and revenue generation potential.
- Optimize for cost-effectiveness: Identify the most cost-effective approach that meets the network requirements. Strike a balance between the desired network performance and the associated costs.
- Plan for scalability: Anticipate future growth and scalability requirements when making cost-related decisions. Consider the long-term benefits and potential cost savings of scalable network solutions.

We must remember that the specific cost analysis will vary depending on the unique characteristics of your network and business requirements. It's essential to gather accurate data, consult with network experts, and tailor the analysis to your organization's specific needs.

There is no universal formula that can be used to calculate the minimum cost path in a network topology based on bandwidth, jitter, and latency. The reason for this is that the cost of a path depends on various factors, such as the network topology, the type of traffic, and the specific requirements of the application.

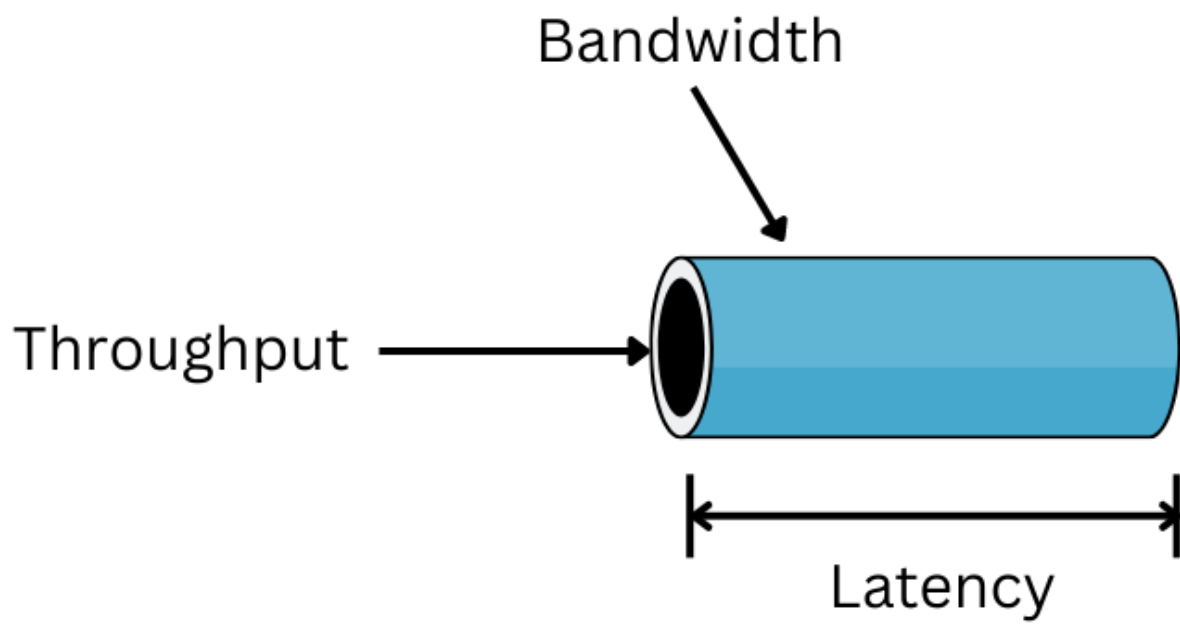
However, it is possible to use mathematical models and algorithms to calculate the cost of a path based on different factors, including bandwidth, jitter, and latency. For example, the cost of a path can be calculated as a weighted sum of these factors, where each factor is assigned a weight that reflects its importance. The specific weights used will depend on the requirements of the application and the characteristics of the network.

In general, bandwidth is an important factor in determining the cost of a path, as it reflects the capacity of the network to transmit data. Jitter and latency are also important factors, as they reflect the delay and variability of data transmission in the network.

To use these factors in a formula for calculating the cost of a path, you can assign a weight to each factor based on its relative importance. For example, if bandwidth is the most important factor, it might be assigned a weight of 0.6, while jitter and latency are assigned weights of 0.2 each. The cost of a path can then be calculated as the weighted sum of these factors:

$$\text{Cost} = (0.6 \times \text{Bandwidth}) + (0.2 \times \text{Jitter}) + (0.2 \times \text{Latency})$$

This formula provides a way to calculate the cost of a path based on bandwidth, jitter, and latency. However, it is important to note that the specific weights used in the formula will depend on the requirements of the application and the characteristics of the network.



*Figure 24: Network Performances*

# Chapter 3: SDN Topology Using Neo4J

## 3.1. Modelling

We model a software-defined network as:

$$G = \{C, S, H, L\}$$

Where,

$C$  = Set of Controllers,

$S$  = Set of Switches,

$H$  = Set of Hosts,

$L$  = Set of Links.

### *Import Network:*

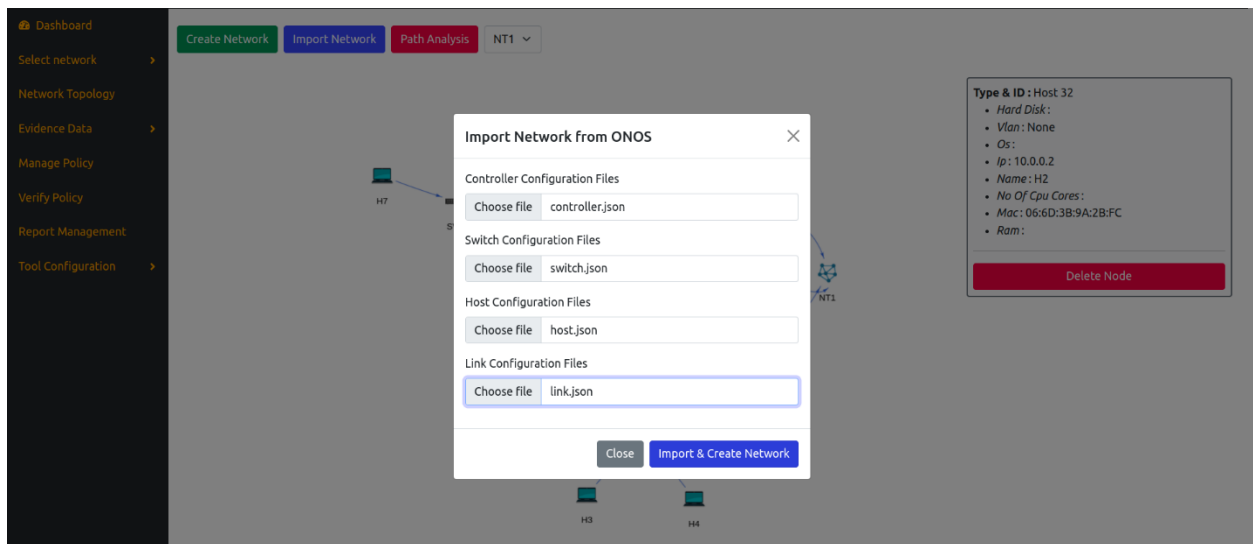


Figure 25: Importation of Configuration Files for SDN in the tool.

### *Format of C: The Set of Controllers*

```
{  
  "controller": [  
    {  
      "id": "C1",  
      "IPListincluster": "1",  
      "controllerType": "ONOS",  
    }  
  ]  
}
```

```

        "noofhosts": "3",
        "noofdevicescon": "3",
        "nooflinks": "6",
        "IP": "192.168.0.130",
        "connectedSwitchId": "of:00000000000000003"
    },
    {
        "id": "C2",
        "IPListincluster": "1",
        "controllerType": "ONOS",
        "noofhosts": "3",
        "noofdevicescon": "3",
        "nooflinks": "6",
        "IP": "192.168.0.131",
        "connectedSwitchId": "of:00000000000000003"
    }
]
}

```

### ***Create Controller method in Neo4JController.py:***

As we give the details of the controllers in a json file, the tool parses it, and creates the controller(s) as node(s) under the then current network infrastructure.

---

#### **Algorithm: Function to create controller**

---

**Input:** parent id, name, properties={}, link properties={}

**Output:** partial creation of the controller node

1. Function create\_controller(parent\_id, name, properties={}, link\_properties={}):
2. Define a nested function \_create\_controller(tx, \_parent\_id, \_name):
3. Create a query string using the following steps:
4. MATCH (parent) WHERE ID(parent) = \$parent\_id.
5. CREATE(c:Controller{name: \$name}).
6. Neo4JController.generate\_query\_for\_properties('c', properties).
7. MERGE (parent)-[r:LINK]->(c).
8. Neo4JController.generate\_query\_for\_properties('r', link\_properties).
9. RETURN ID(c), c.name.
10. Execute the query using tx.run(query, parent\_id=\_parent\_id, name=\_name, \*\*link\_properties, \*\*properties).
11. Return the single result from the query.
12. End.
13. Call the NetworkController.from\_neo4j\_response() method with the result of Neo4JController.write\_transaction(\_create\_controller, parent\_id, name).
13. Return the result of the NetworkController.from\_neo4j\_response() method.
14. End.

Note: The steps 3-9 describe the construction of the query string, where Neo4JController.generate\_query\_for\_properties() is a placeholder for a method that generates query strings based on the provided properties and their values.

The algorithm defines a function `create_controller()` that takes the parent ID, name, and optional properties and `link_properties` as input. Inside this function, a nested function `_create_controller()` is defined, which performs the actual database operation. The `_create_controller()` function constructs a query string and executes it using the provided parameters. Finally, the result is processed and returned using the `NetworkController.from_neo4j_response()` method.

### ***Format of S: The Set of Switches***

```
{
  "devices": [
    {
      "id": "of:000000000000000003",
      "type": "SWITCH",
      "available": false,
      "role": "MASTER",
      "mfr": "Nicira, Inc.",
      "hw": "Open vSwitch",
      "sw": "2.9.8",
      "serial": "None",
      "driver": "ovs",
      "chassisId": "3",
      "lastUpdate": "1674010507107",
      "humanReadableLastUpdate": "connected 12m8s ago",
      "annotations": {
        "channelId": "172.0.0.3:37228",
        "datapathDescription": "None",
        "managementAddress": "172.0.0.3",
        "protocol": "OF_14"
      }
    },
    {
      "id": "of:000000000000000004",
      "type": "SWITCH",
      "available": false,
      "role": "MASTER",
      "mfr": "Nicira, Inc.",
      "hw": "Open vSwitch",
      "sw": "2.9.8",
      "serial": "None",
      "driver": "ovs",
      "chassisId": "4",
      "lastUpdate": "1674010507105",
      "humanReadableLastUpdate": "connected 12m8s ago",
      "annotations": {
        "channelId": "172.0.0.3:37228",
        "datapathDescription": "None",
        "managementAddress": "172.0.0.3",
        "protocol": "OF_14"
      }
    }
  ]
}
```

### ***Create Switch method In Neo4JController.py:***

As we give the details of the switch(es) in a json file, the tool parses it, and creates the switch(es) as node(s) under the then current network infrastructure.

---

#### **Algorithm: Function to create switch**

---

- Input:** parent id, name, properties={}, link properties={}
- Output:** creation of the switch node
1. Function create\_switch(parent\_id, name, properties={}, link\_properties={}):
  2. Define a nested function \_create\_switch(tx, \_parent\_id, \_name):
  3.     Create a query string using the following steps:
  4.     MATCH (parent) WHERE ID(parent) = \$parent\_id.
  5.     CREATE(s:Switch{name: \$name}).
  6.     Neo4JController.generate\_query\_for\_properties('s', properties).
  7.     MERGE (parent)-[r:LINK]->(s).
  8.     Neo4JController.generate\_query\_for\_properties('r', link\_properties).
  9.     RETURN ID(s), s.name.
  10.    Execute the query using tx.run(query, parent\_id=\_parent\_id, name=\_name, \*\*link\_properties, \*\*properties).
  11.    Return the single result from the query.
  12.    End.
  13. Call the NetworkSwitch.from\_neo4j\_response() method with the result of Neo4JController.write\_transaction(\_create\_switch, parent\_id, name).
  13. Return the result of the NetworkSwitch.from\_neo4j\_response() method.
  14. End.

Note: The steps 3-9 describe the construction of the query string, where Neo4JController.generate\_query\_for\_properties() is a placeholder for a method that generates query strings based on the provided properties and their values.

The algorithm defines a function create\_switch() that takes the parent ID, name, and optional properties and link\_properties as input. Inside this function, a nested function \_create\_switch() is defined, which performs the actual database operation. The \_create\_switch() function constructs a query string and executes it using the provided parameters. Finally, the result is processed and returned using the NetworkSwitch.from\_neo4j\_response() method.

### ***Format of H: The Set of Hosts***

```
{
  "hosts": [
    {
```



```

      "id": "82:F6:F2:6D:6F:CC/None",
      "mac": "82:F6:F2:6D:6F:CC",
      "vlan": "None",
      "innerVlan": "None",
      "outerTpid": "0x0000",
      "configured": false,
      "suspended": false,
      "ipAddresses": [
        "10.0.0.5"
      ],
      "locations": [
        {
          "elementId": "of:000000000000000006",
          "port": "1"
        }
      ]
    },
    {
      "id": "36:E2:A1:87:50:96/None",
      "mac": "36:E2:A1:87:50:96",
      "vlan": "None",
      "innerVlan": "None",
      "outerTpid": "0x0000",
      "configured": false,
      "suspended": false,
      "ipAddresses": [
        "10.0.0.1"
      ],
      "locations": [
        {
          "elementId": "of:000000000000000003",
          "port": "1"
        }
      ]
    },
    {
      "id": "82:A0:59:34:53:56/None",
      "mac": "82:A0:59:34:53:56",
      "vlan": "None",
      "innerVlan": "None",
      "outerTpid": "0x0000",
      "configured": false,
      "suspended": false,
      "ipAddresses": [
        "10.0.0.4"
      ],
      "locations": [
        {
          "elementId": "of:000000000000000004",
          "port": "2"
        }
      ]
    }
  ],
  {

```

```

        "id": "06:6D:3B:9A:2B:FC/None",
        "mac": "06:6D:3B:9A:2B:FC",
        "vlan": "None",
        "innerVlan": "None",
        "outerTpid": "0x0000",
        "configured": false,
        "suspended": false,
        "ipAddresses": [
            "10.0.0.2"
        ],
        "locations": [
            {
                "elementId": "of:000000000000000003",
                "port": "2"
            }
        ]
    }
}

```

### ***Create host method in Neo4JController.py:***

As we give the details of the host(s) in a json file, the tool parses it, and creates the host(s) as node(s) under the then current network infrastructure.

---

#### **Algorithm: Function to create host**

---

- Input:** parent id, name, properties={}, link properties={}
- Output:** creation of the host node
1. Function create\_host(parent\_id, name, properties={}, link\_properties={}):
  2. Define a nested function \_create\_host(tx, \_parent\_id, \_name):
  3. Create a query string using the following steps:
  4. MATCH (parent) WHERE ID(parent) = \$parent\_id.
  5. CREATE(h:Host{name: \$name}).
  6. Neo4JController.generate\_query\_for\_properties('h', properties).
  7. MERGE (parent)-[r:LINK]->(h).
  8. Neo4JController.generate\_query\_for\_properties('r', link\_properties).
  9. RETURN ID(h), h.name.
  10. Execute the query using tx.run(query, parent\_id=\_parent\_id, name=\_name, \*\*link\_properties, \*\*properties).
  11. Return the single result from the query.
  12. End.
  13. Call the NetworkHost.from\_neo4j\_response() method with the result of Neo4JController.write\_transaction(\_create\_host, parent\_id, name).
  13. Return the result of the NetworkHost.from\_neo4j\_response() method.
  14. End.

Note: The steps 3-9 describe the construction of the query string, where Neo4JController.generate\_query\_for\_properties() is a placeholder for a method that generates query strings based on the provided properties and their values.

The algorithm defines a function `create_host()` that takes the parent ID, name, and optional properties and `link_properties` as input. Inside this function, a nested function `_create_host()` is defined, which performs the actual database operation. The `_create_host()` function constructs a query string and executes it using the provided parameters. Finally, the result is processed and returned using the `NetworkHost.from_neo4j_response()` method.

### ***Format of L: The Set of Livks***

```
{
  "links": [
    {
      "src": {
        "port": "2",
        "device": "of:000000000000000001"
      },
      "dst": {
        "port": "3",
        "device": "of:000000000000000005"
      },
      "type": "DIRECT",
      "state": "ACTIVE"
    },
    {
      "src": {
        "port": "3",
        "device": "of:000000000000000004"
      },
      "dst": {
        "port": "2",
        "device": "of:000000000000000002"
      },
      "type": "DIRECT",
      "state": "ACTIVE"
    }
  ]
}
```

### ***Create link method in Neo4JController.py:***

As we give the details of the links(s) in a json file, the tool parses it, and creates the link(s) as relationships(s) under the then current network infrastructure.

---

#### **Algorithm: Function to establish links**

---

**Input:** source id, target id, properties={}

**Output:** creation of the links

1. Function `create_link(source_id, target_id, properties={})`:
2. Define a nested function `_create_link(tx, _source_id, _target_id)`:
3. Execute the query using the following steps:
4. MATCH (a), (b) WHERE ID(a) = \$source\_id AND ID(b) = \$target\_id
5. CREATE (a)-[r:LINK]->(b)

6. Neo4JController.generate\_query\_for\_properties('r', properties)
7. Execute the query using tx.run(query, source\_id=\_source\_id, target\_id=\_target\_id, \*\*properties)
8. Neo4JController.generate\_query\_for\_properties('r', link\_properties).
9. RETURN ID(h), h.name.
10. Execute the query using tx.run(query, parent\_id=\_parent\_id, name=\_name, \*\*link\_properties, \*\*properties).
11. End.
12. Try executing the Neo4JController.write\_transaction(\_create\_link, source\_id, target\_id) block:
13. If successful, return True.
14. If an exception occurs, print the error and return False.
15. End.

Note: The steps 3-6 describe the construction of the query string, where Neo4JController.generate\_query\_for\_properties() is a placeholder for a method that generates query strings based on the provided properties and their values.

The algorithm defines a function create\_link() that takes the source ID, target ID, and optional properties as input. Inside this function, a nested function \_create\_link() is defined, which performs the database operation. The \_create\_link() function constructs a query string and executes it using the provided parameters. The write\_transaction() block is executed within a try-except block to handle any exceptions that may occur. If the transaction is successful, the function returns True. Otherwise, it prints the error message and returns False.

## 3.2. Visualization of Network Model

*Get graph method in Neo4JController.py:*

The algorithm defines a function get\_graph() that takes the parent ID and an optional format as input. Inside this function, a nested function \_get\_network\_tree() is defined, which performs the database query to retrieve the network tree. The response is processed based on the format type and returned accordingly.

---

### Algorithm: Function to get graphs

---

- Input:** parent id  
**Output:** Get graphs
1. Function get\_graph(parent\_id, format="HierarchicalFormat"):
  2. Define a nested function \_get\_network\_tree(tx, \_parent\_id):
  3. Execute the query using the following steps:
  4. Set the query to match the parent node with ID = \$parent\_id
  5. Use the apoc.path.subgraphAll() procedure to get the subgraph with label filter "\*" and relationship filter "LINK"

6. Extract the necessary information from the subgraph, including parent ID, label, name, and children
7. Return the result of the query execution
8. Call the `Neo4JController.read_transaction()` method with `_get_network_tree` and `parent_id` as arguments, and store the response in the "response" variable
9. Check if the response is empty or has an empty inner list:
10. If true, return an empty dictionary if the format is "HierarchicalFormat",
11. Otherwise return an empty dictionary for nodes and relationships if the format is "VisJSFormat"
12. Otherwise, extract the first element from the response list and assign it to the "response" variable
13. Check the format type:
14. If the format is "HierarchicalFormat", call the `format_data_hierarchical_json()` function with the necessary data and the "children" from the response, and return the formatted data
15. If the format is "VisJSFormat", call the `format_data_to_vis_js_format()` function with the necessary data and the "children" from the response, and return the formatted data
16. Otherwise, return None

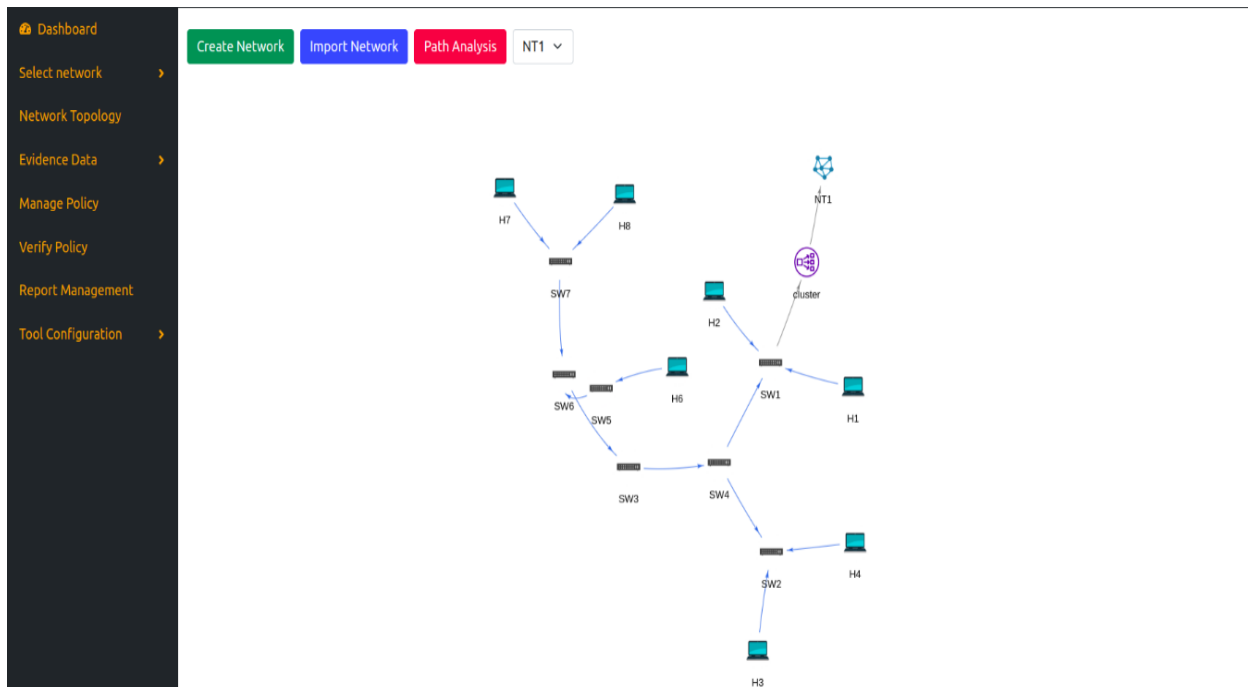


Figure 26: A SDN Topology in the tool.

### 3.3. Updating

*Update node method in Neo4JController.py:*

The algorithm defines a function `updatenode()` that takes the data dictionary and optional properties as input. Inside this function, the `network_id` and `id` are extracted from the data dictionary. Then, a nested function `_updatenode()` is defined, which performs the database update operation. The `_updatenode()` function constructs a query string based on the provided data and properties, and executes the query using the `write_transaction()` method. If the transaction is successful, the function returns `True`. Otherwise, it prints the error message and returns `False`.

---

**Algorithm: Function to update nodes**

---

**Input:** data, properties={}

**Output:** updated nodes

1. Function `updatenode(data, properties={})`:
2. Extract the "network\_id" and "id" from the data dictionary and assign them to `network_id` and `id` variables, respectively.
3. Remove "network\_id" and "id" keys from the data dictionary.
4. Define a nested function `_updatenode(tx, _id)`:
5.     Construct the query string using the following steps:
6.         `MATCH (n) WHERE id(n) = id`
7.         "SET" followed by a comma-separated list of property updates using data dictionary values
8.         `Neo4JController.generate_query_for_properties('n', properties)`
9.         Execute the query using `tx.run(query_string, id=_id, **properties)`
10.        Return the result and the `query_string` as a dictionary
11. Try executing the `Neo4JController.write_transaction(_updatenode, id)` block:
12.     If successful, return `True`
13.     If an exception occurs, print the error and return `False`

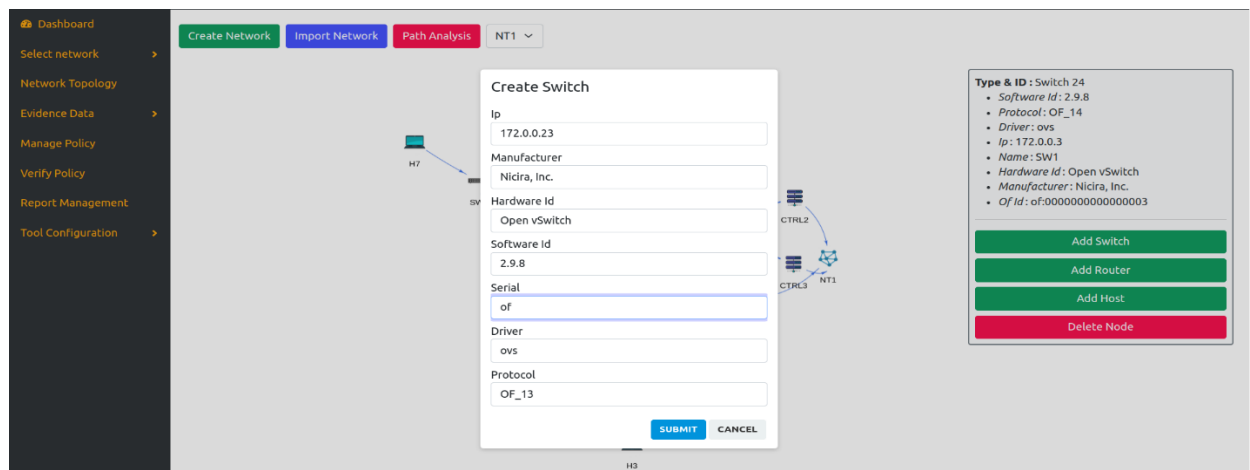


Figure 27: Updating a switch node in SDN in the tool

### 3.4. Deleting

*Delete node method in Neo4JController.py:*

The algorithm defines a function `delete_node()` that takes the node ID and an optional `delete_all_nodes_relationship` parameter as input. Inside this function, there are two nested functions: `_delete_all_nodes_relationships` and `_delete_node`. The `_delete_all_nodes_relationships` function deletes all nodes and relationships connected to the specified node ID, while the `_delete_node` function deletes only the specified node. The appropriate nested function is executed based on the value of `delete_all_nodes_relationship` using the `Neo4JController.write_transaction()` method.

---

**Algorithm: Function to delete nodes**

---

- Input:** id, delete\_all\_nodes\_relationship=False  
**Output:** delete nodes
1. Function `delete_node(id, delete_all_nodes_relationship=False)`:
  2. Define a nested function `_delete_all_nodes_relationships(tx, _id)`:
  3.     Execute the query using the following steps:
  4.         Set the query to match all nodes and relationships connected to the node with ID = id
  5.         Use OPTIONAL MATCH to also match relationships connected to the connected nodes
  6.         Delete the relationships and the connected nodes
  7.         Execute the query using `tx.run(query, id=_id)`
  8. Define a nested function `_delete_node(tx, _id)`:
  9.     Execute the query using the following steps:
  10.         Set the query to match the node with ID = id
  11.         Use DETACH DELETE to delete the node and its relationships
  12.         Execute the query using `tx.run(query, id=_id)`
  13. Check if `delete_all_nodes_relationship` is True:
  14.     If true, call the `Neo4JController.write_transaction()` method with `_delete_all_nodes_relationships` and id as arguments
  15.     Otherwise, call the `Neo4JController.write_transaction()` method with `_delete_node` and id as arguments

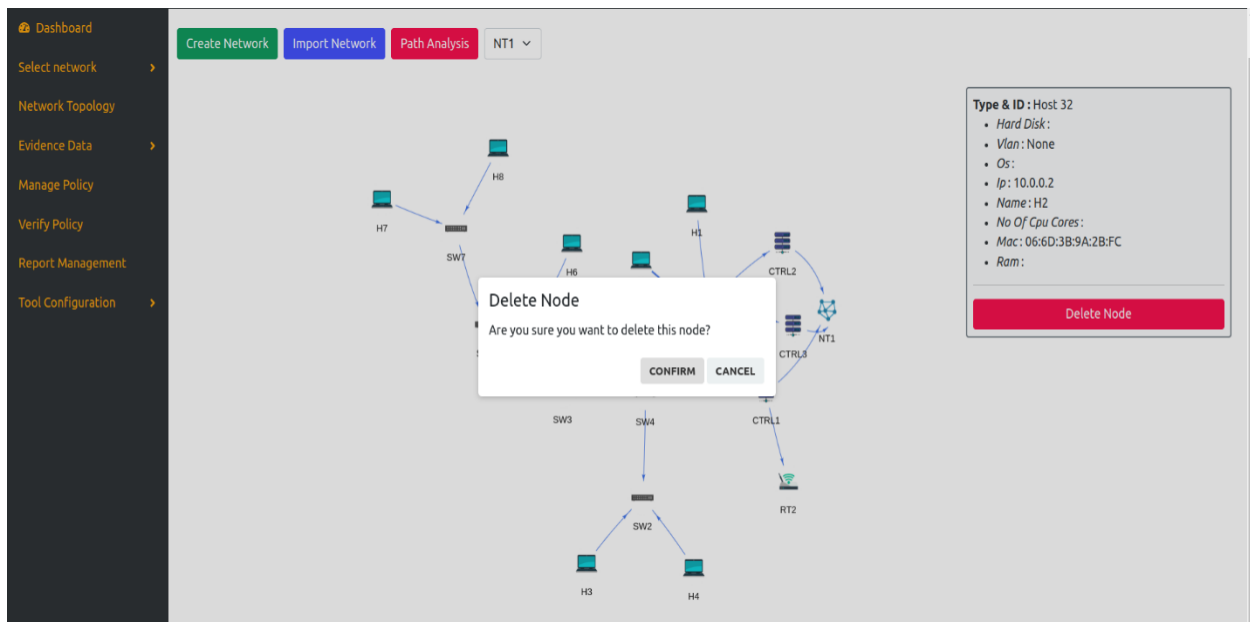


Figure 28: Deletion of a host node in SDN in the tool.





```

{
  "id": "ebc97359-012a-48c2-a005-88a5aa7519af",
  "status": "Active",
  "name": "client",
  "ip": "10.10.1.115",
  "sec_group": "default",
  "vCPU": "1",
  "RAM": "1 GB",
  "size": "10 GB",
  "image_name": "cirros"
},
{
  "id": "391cd841-9ead-4c81-9b94-cd60cda0998d",
  "status": "Active",
  "name": "server",
  "ip": "10.10.0.176",
  "sec_group": "None",
  "vCPU": "1",
  "RAM": "1 GB",
  "size": "10 GB",
  "image_name": "ubuntu"
},
{
  "id": "391cd841-9ead-4c81-9b94-cd60cda2589d",
  "status": "Active",
  "name": "client",
  "ip": "10.10.0.178",
  "sec_group": "None",
  "vCPU": "1",
  "RAM": "1 GB",
  "size": "10 GB",
  "image_name": "cirros"
}
]
}

```

### ***Create virtual machine method in Neo4JConroller.py:***

The algorithm defines a function `create_virtualmachine()` that takes the `parent_name`, `name`, and optional properties as input. Inside this function, a nested function `_create_virtualmachine()` is defined, which performs the database operation. The `_create_virtualmachine()` function constructs a query string and executes it using the provided parameters. Finally, the result is processed and returned using the `Vnvm.from_neo4j_response()` method, which is called with the result of the `Neo4JController.write_transaction()` method as input.

---

#### **Algorithm: Function to create virtual machine**

---

**Input:** parent id, name, properties={}

**Output:** partial creation of the virtual machine node

1. Function `create_virtualmachine(parent_name, name, properties={})`:
2. Define a nested function `_create_virtualmachine(tx, _name, _parent_name)`:
3. Execute the query using the following steps:
4. `CREATE(h:{parent_name}:Virtualmachine{name: $name})`
5. `Neo4JController.generate_query_for_properties('h', properties)`
6. `RETURN ID(h), h.name`
7. Execute the query using `tx.run(query, parent_name=_parent_name, name=_name, **properties)`
8. Return the single result from the query
9. Call the `Neo4JController.write_transaction()` method with `_create_virtualmachine, name, and parent_name` as arguments
10. Call the `Vnvm.from_neo4j_response()` method with the result of the `Neo4JController.write_transaction()` method
11. Return the result from the `Vnvm.from_neo4j_response()` method
12. End.

### ***Format of VN: The Set of Virtual Networks***

```
{
  "vns": [
    {
      "name": "net3",
      "vn_id": "3f1e0ab6-af04-456c-8ba1-3edbbdd432ce",
      "Status": "Active",
      "MTU": "1500",
      "network_address": "192.168.0.0/24",
      "attached_devices": ["VM1", "VM2"]
    },
    {
      "name": "net2",
      "vn_id": "4f1e0ab6-af04-456c-8ba1-3edbbdd432ce",
      "Status": "Active",
      "MTU": "1400",
      "network_address": "192.168.1.0/24",
      "attached_devices": ["VM3", "VM4"]
    }
  ]
}
```

### ***Create virtual network method in Neo4JController.py:***

The algorithm defines a function `create_virtualnetwork()` that takes the `parent_name, name,` and optional `properties` as input. Inside this function, a nested function `_create_virtualnetwork()` is defined, which performs the database operation. The

\_create\_virtualnetwork() function constructs a query string and executes it using the provided parameters. Finally, the result is processed and returned using the Vnivn.from\_neo4j\_response() method, which is called with the result of the Neo4JController.write\_transaction() method as input.

---

#### **Algorithm: Function to create virtual network**

---

- Input:** parent id, name, properties={}  
**Output:** partial creation of the virtual network node
1. Function create\_virtualnetwork(parent\_name, name, properties={}):
  2. Define a nested function \_create\_virtualnetwork(tx, \_name, \_parent\_name):
  3.     Execute the query using the following steps:
  4.     CREATE(h:{parent\_name}:Virtualnetwork{name: \$name})
  5.     Neo4JController.generate\_query\_for\_properties('h', properties)
  6.     RETURN ID(h), h.name
  7.     Execute the query using tx.run(query, parent\_name=\_parent\_name, name=\_name, \*\*properties)
  8.     Return the single result from the query
  9.     Call the Neo4JController.write\_transaction() method with \_create\_virtualnetwork, name, and parent\_name as arguments
  10. Call the Vnivn.from\_neo4j\_response() method with the result of the Neo4JController.write\_transaction() method
  11. Return the result from the Vnivn.from\_neo4j\_response() method
  12. End.

#### ***Format of VNF: The Set of Virtual Network Functions***

```
{
  "vnfs": [
    {
      "num_cpus": "1",
      "mem_size": "512 MB",
      "disk_size": "16 GB",
      "image": "ubuntu_custom",
      "cps":["net0", "net1", "net2"],
      "vnfID": "6c42ghjut-3357-4096-85b3-2a098124547f",
      "vimID": "9e5866c6-317c-40fd-8446-adea8beaf4a1"
    },
    {
      "num_cpus": "1",
      "mem_size": "512 MB",
      "disk_size": "16 GB",
      "image": "ubuntu_custom",
      "cps":["net3", "net4", "net5"],
      "vnfID": "6c42dfgt-1999-4096-85b3-2a098124547f",
      "vimID": "9e5866c6-317c-40fd-7474-adea8beaf4a1"
    }
  ]
}
```

```

    },
    {
      "num_cpus": "1",
      "mem_size": "512 MB",
      "disk_size": "16 GB",
      "image": "ubuntu_custom",
      "cps":["net6", "net4", "net2"],
      "vnfID": "6c4895hh-1999-4896-85b3-2a098124547f",
      "vimID": "9e5866c6-317c-40fd-7474-adea1jdk19b2"
    },
    {
      "num_cpus": "1",
      "mem_size": "512 MB",
      "disk_size": "16 GB",
      "image": "ubuntu_custom",
      "cps":["net7", "net1", "net8"],
      "vnfID": "7k52ea97-1099-1096-75b3-3b55huo124547f",
      "vimID": "8h4563c6-345b-40fd-7478-xdea8beaf4a1"
    }
  ]
}

```

### ***Create virtual network function method in Neo4JController.py:***

The algorithm defines a function `create_virtualnetworkfunction()` that takes the `parent_name`, `name`, and optional properties as input. Inside this function, a nested function `_create_virtualnetworkfunction()` is defined, which performs the database operation. The `_create_virtualnetworkfunction()` function constructs a query string and executes it using the provided parameters. Finally, the result is processed and returned using the `Vnivnf.from_neo4j_response()` method, which is called with the result of the `Neo4JController.write_transaction()` method as input.

---

#### **Algorithm: Function to create virtual network function**

---

**Input:** parent id, name, properties={}

**Output:** partial creation of the virtual network function node

1. Function `create_virtualnetworkfunction(parent_name, name, properties={})`:
2. Define a nested function `_create_virtualnetworkfunction(tx, _name, _parent_name)`:
3. Execute the query using the following steps:
4. `CREATE(h:{parent_name}:Virtualnetworkfunction{name: $name})`
5. `Neo4JController.generate_query_for_properties('h', properties)`
6. `RETURN ID(h), h.name`
7. Execute the query using `tx.run(query, parent_name=_parent_name, name=_name, **properties)`
8. Return the single result from the query

9. Call the Neo4JController.write\_transaction() method with \_create\_virtualnetworkfunction, name, and parent\_name as arguments
10. Call the Vnivnf.from\_neo4j\_response() method with the result of the Neo4JController.write\_transaction() method
11. Return the result from the Vnivnf.from\_neo4j\_response() method
12. End.

## 4.2. Visualization of Network Elements:

*Create and Get VNI method in Neo4JController.py:*

The algorithm defines two functions: create\_vni() and get\_vni().

The create\_vni() function takes the name and optional properties as input. Inside this function, a nested function \_create\_vni() is defined, which performs the database operation. The \_create\_vni() function constructs a query string and executes it using the provided parameters. Finally, the result is processed and returned using the Network.from\_neo4j\_response() method.

The get\_vni() function is a static method that retrieves all VNI nodes from the database. Inside this method, a nested function \_get\_vni() is defined, which executes the query to retrieve the VNI nodes. The values from the query execution are processed, and a list of Network objects is returned, created using the Network.from\_neo4j\_response() method for each response.

---

### Algorithm: Function to create virtual network infrastructure

---

- Input:** name, properties={}
- Output:** creation of the virtual network infrastructure
1. Function create\_vni(name, properties={}):
  2. Define a nested function \_create\_vni(tx, \_name):
  3. Execute the query using the following steps:
  4. CREATE (n:VNI {name: \$name}) followed by Neo4JController.generate\_query\_for\_properties('n', properties) and RETURN ID(n), n.name
  5. Execute the query using tx.run(query, name=\_name, \*\*properties)
  6. Return the single result from the query
  7. Call the Neo4JController.write\_transaction() method with \_create\_vni and name as arguments
  8. Call the Network.from\_neo4j\_response() method with the result of the Neo4JController.write\_transaction() method as input
  9. Return the result from the Network.from\_neo4j\_response() method
  10. Static method get\_vni():
  11. Define a nested function \_get\_vni(tx):
  12. Execute the query to retrieve all VNI nodes using tx.run("MATCH (n:VNI) RETURN ID(n), n.name")

13. Return the values from the query execution
14. Call the `Neo4JController.read_transaction()` method with `_get_vni` as an argument and store the response in the "result" variable
15. Iterate over each response in the "result" variable
16. Call the `Network.from_neo4j_response()` method with the response as input and create a `Network` object for each response
17. Return a list of the `Network` objects

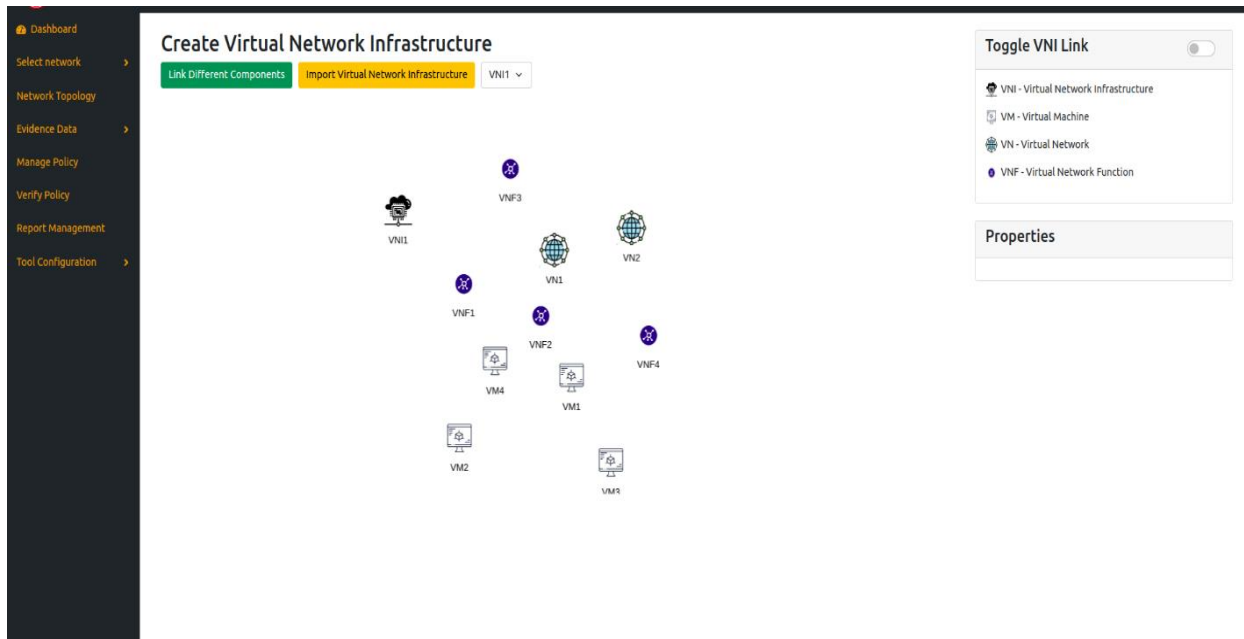


Figure 30: Network Elements in a NFV Topology

## 4.3. Creating Links:

*Create link methods in `Neo4JController.py`:*

The algorithm defines a function `createvnlink_vn_to_vnf()` that takes `vni`(virtual network infrastructure), `vn`(virtual network), `vnf`(virtual network function), and optional properties as input. Inside this function, a nested function `_createvnlink_vn_to_vnf()` is defined, which performs the database operation. The `_createvnlink_vn_to_vnf()` function constructs a query string to match nodes of the specified types and create a relationship between them. The query is executed using the provided parameters, and the result is stored in the "result" variable as a dictionary.

The function then tries to execute the `Neo4JController.write_transaction()` method for each vnf in the vnf list. If all transactions are successful, True is returned. Otherwise, the error is printed, and False is returned.

[Similar algorithm for establishing link between vm(virtual machine) and vn(virtual network) within a vni(virtual network infrastructure).]

---

**Algorithm: Function to create links**

---

**Input:** vni,vn,vnf, properties={}

**Output:** creation of relationship between virtual network to virtual network function

1. Function `createvnlink_vn_to_vnf(vni, vn, vnf, properties={})`:
2. Define a nested function `_createvnlink_vn_to_vnf(tx, _vni, _vn, _vnf)`:
3.     Execute the query using the following steps:
4.     Set the query to match nodes of type vni and vnf and create a relationship between them
5.     Use MATCH to find nodes with `vn_id = vn` and `vnfID = vnf`
6.     Use MERGE to create a relationship (a)-[r:LINK]->(b)
7.     Append `Neo4JController.generate_query_for_properties('r', properties)`
8.     Execute the query using `tx.run(query, vni=_vni, vn=_vn, vnf=_vnf, **properties)` and store the result in the "result" variable as a dictionary
9.    Try executing the following block for each vnf in the vnf list:
10.     Call the `Neo4JController.write_transaction()` method with `_createvnlink_vn_to_vnf, vni, vn,` and `x (vnf)` as arguments, and store the result in the "response" variable
11.    If all transactions are successful, return True
12.    If an exception occurs, print the error and return False



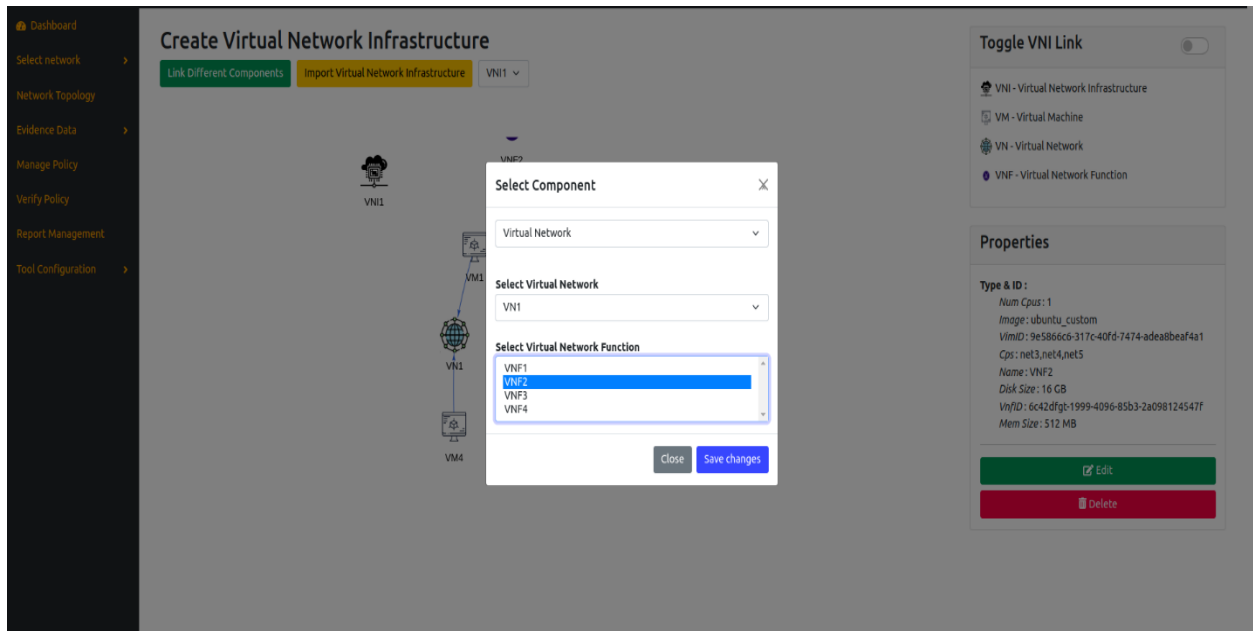


Figure 31: Linking between nodes in a NFV

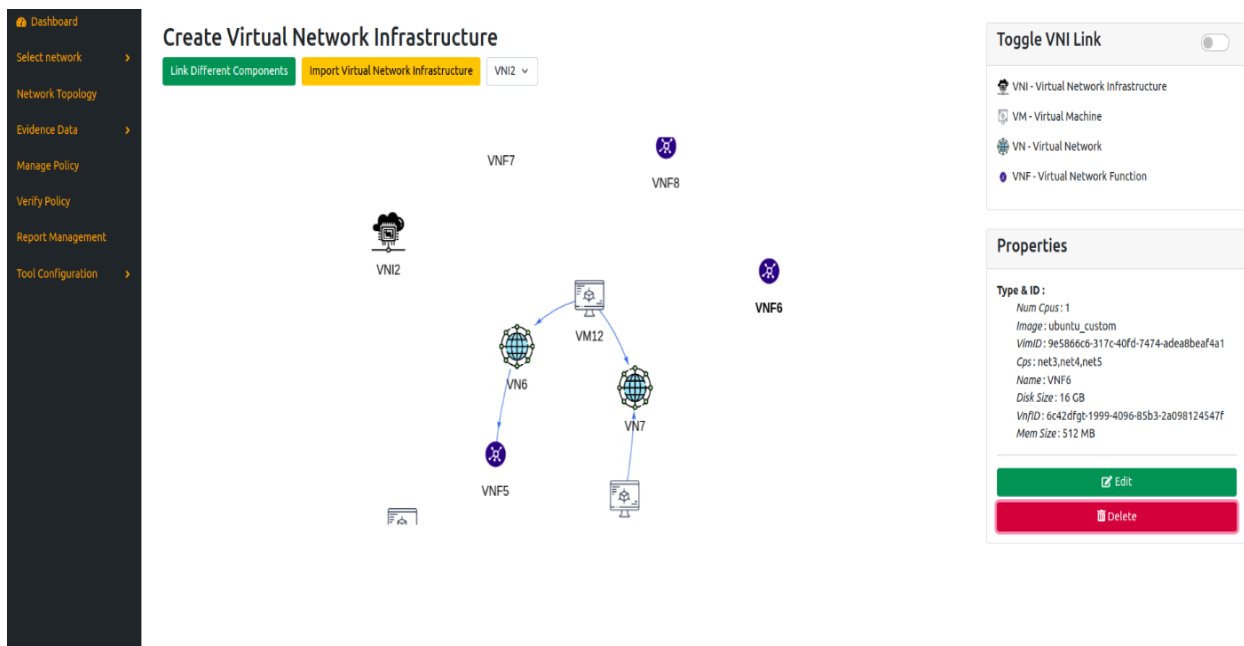


Figure 32: Link Visualization in NFV.

## 4.4. Updating:

*Update node method in Neo4JController.py:*

The algorithm defines a function `updatenode()` that takes the data dictionary and optional properties as input. Inside this function, the `network_id` and `id` are extracted from the data dictionary. Then, a nested function `_updatenode()` is defined, which performs the database update operation. The `_updatenode()` function constructs a query string based on the provided data and properties, and executes the query using the `write_transaction()` method. If the transaction is successful, the function returns `True`. Otherwise, it prints the error message and returns `False`.

---

**Algorithm: Function to update nodes**

---

- Input:** data, properties={}  
**Output:** updated nodes
1. Function `updatenode(data, properties={})`:
  2. Extract the "network\_id" and "id" from the data dictionary and assign them to `network_id` and `id` variables, respectively.
  3. Remove "network\_id" and "id" keys from the data dictionary.
  4. Define a nested function `_updatenode(tx, _id)`:
  5.     Construct the query string using the following steps:
  6.         MATCH (n) WHERE id(n) = id
  7.         "SET" followed by a comma-separated list of property updates using data dictionary values
  8.         `Neo4JController.generate_query_for_properties('n', properties)`
  9.         Execute the query using `tx.run(query_string, id=_id, **properties)`
  10.        Return the result and the query\_string as a dictionary
  11. Try executing the `Neo4JController.write_transaction(_updatenode, id)` block:
  12.     If successful, return `True`
  13.     If an exception occurs, print the error and return `False`

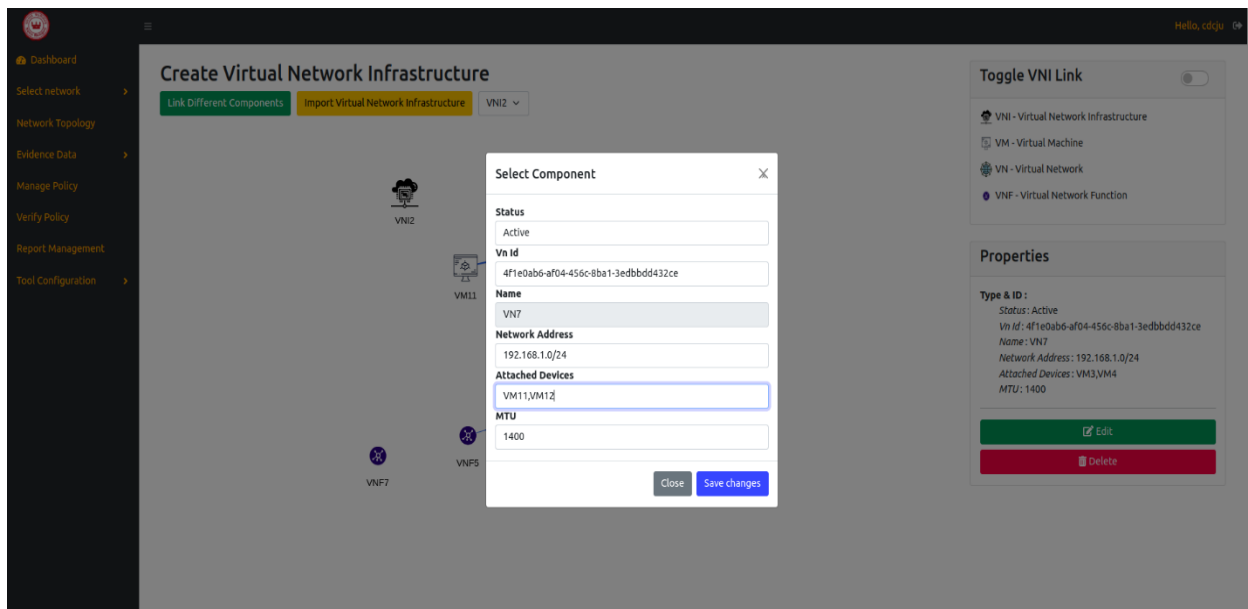


Figure 33: Updating a node in NFV Topology in the tool

## 4.5. Deleting

*Delete node method in Neo4JController.py:*

The algorithm defines a function `delete_node()` that takes the node ID and an optional `delete_all_nodes_relationship` parameter as input. Inside this function, there are two nested functions: `_delete_all_nodes_relationships` and `_delete_node`. The `_delete_all_nodes_relationships` function deletes all nodes and relationships connected to the specified node ID, while the `_delete_node` function deletes only the specified node. The appropriate nested function is executed based on the value of `delete_all_nodes_relationship` using the `Neo4JController.write_transaction()` method.

---

### Algorithm: Function to delete nodes

---

**Input:** id, delete\_all\_nodes\_relationship=False

**Output:** delete nodes

1. Function `delete_node(id, delete_all_nodes_relationship=False)`:
2. Define a nested function `_delete_all_nodes_relationships(tx, _id)`:
3.     Execute the query using the following steps:
4.         Set the query to match all nodes and relationships connected to the node with ID = id
5.         Use OPTIONAL MATCH to also match relationships connected to the connected nodes
6.         Delete the relationships and the connected nodes
7.     Execute the query using `tx.run(query, id=_id)`
8. Define a nested function `_delete_node(tx, _id)`:
9.     Execute the query using the following steps:

10. Set the query to match the node with ID = id
11. Use DETACH DELETE to delete the node and its relationships
12. Execute the query using tx.run(query, id=\_id)
13. Check if delete\_all\_nodes\_relationship is True:
14. If true, call the Neo4JController.write\_transaction() method with \_delete\_all\_nodes\_relationships and id as arguments
15. Otherwise, call the Neo4JController.write\_transaction() method with \_delete\_node and id as arguments

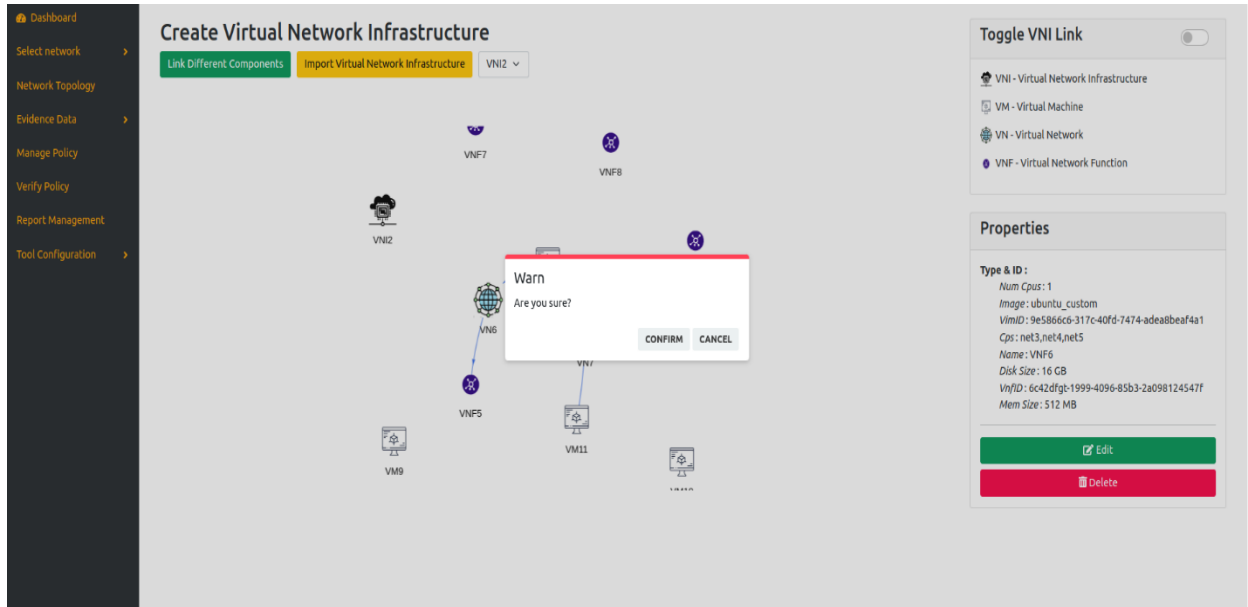


Figure 34: Deletion of Nodes in NFV Topology in the tool

# Chapter 5: NV Topology Using Neo4J

## 5.1. Modelling

We model a network function virtualization infrastructure as:

$$G = \{VM, VN, R, E\}$$

Where,

$VM = \text{Set of virtual machines}$

$VN = \text{Set of virtual networks}$

$R = \text{Set of Routers}$

$E = \text{Set of edges}$

**Import Network:**

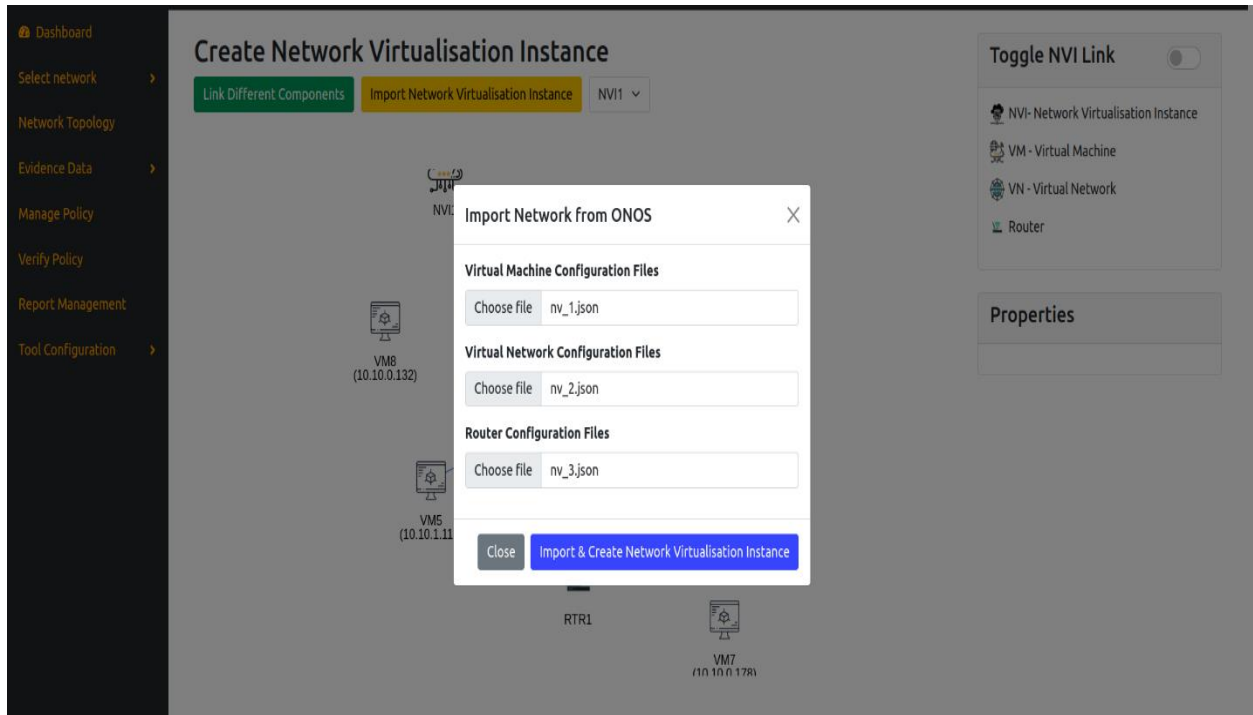


Figure 35: Importing Configuration files for NV Topology in the tool

**Format of VM: The Set of Virtual Machines**

```
{
  "vms": [
    {
```

```

        "id": "ebc97359-012a-48c2-a005-88a5aa7519af",
        "status": "Active",
        "name": "client",
        "ip": "10.10.1.115",
        "sec_group": "default",
        "vCPU": "1",
        "RAM": "1 GB",
        "size": "10 GB",
        "image_name": "cirros"
    },
    {
        "id": "391cd841-9ead-4c81-9b94-cd60cda0998d",
        "status": "Active",
        "name": "server",
        "ip": "10.10.0.176",
        "sec_group": "None",
        "vCPU": "1",
        "RAM": "1 GB",
        "size": "10 GB",
        "image_name": "ubuntu"
    },
    {
        "id": "391cd841-9ead-4c81-9b94-cd60cda2589d",
        "status": "Active",
        "name": "client",
        "ip": "10.10.0.178",
        "sec_group": "None",
        "vCPU": "1",
        "RAM": "1 GB",
        "size": "10 GB",
        "image_name": "cirros"
    }
]
}

```

### ***Create virtual machine method in Neo4JConroller.py:***

The algorithm defines a function `create_virtualmachine()` that takes the `parent_name`, `name`, and optional properties as input. Inside this function, a nested function `_create_virtualmachine()` is defined, which performs the database operation. The `_create_virtualmachine()` function constructs a query string and executes it using the provided parameters. Finally, the result is processed and returned using the `Nvivm.from_neo4j_response()` method, which is called with the result of the `Neo4JController.write_transaction()` method as input.

---

#### **Algorithm: Function to create virtual machine**

---

**Input:** parent id, name, properties={}

**Output:** partial creation of the virtual machine node

1. Function create\_virtualmachine(parent\_name, name, properties={}):
2. Define a nested function \_create\_virtualmachine(tx, \_name, \_parent\_name):
3.     Execute the query using the following steps:
4.     CREATE(h:{parent\_name}:Virtualmachine{name: \$name})
5.     Neo4JController.generate\_query\_for\_properties('h', properties)
6.     RETURN ID(h), h.name
7.     Execute the query using tx.run(query, parent\_name=\_parent\_name, name=\_name, \*\*properties)
8.     Return the single result from the query
9.     Call the Neo4JController.write\_transaction() method with \_create\_virtualmachine, name, and parent\_name as arguments
10. Call the Nvwm.from\_neo4j\_response() method with the result of the Neo4JController.write\_transaction() method
11. Return the result from the Nvwm.from\_neo4j\_response() method
12. End.

### ***Format of VN: The Set of Virtual Networks***

```
{
  "vns": [

    {
      "name": "net3",
      "vn_id": "3f1e0ab6-af04-456c-8ba1-3edbbdd432ce",
      "Status": "Active",
      "MTU": "1500",
      "network_address": "192.168.0.0/24",
      "attached_devices": ["VM1", "VM2"],
      "nature": "internal"
    },
    {
      "name": "net2",
      "vn_id": "4f1e0ab6-af04-456c-8ba1-3edbbdd432ce",
      "Status": "Shut-off",
      "MTU": "1400",
      "network_address": "192.168.1.0/24",
      "attached_devices": ["VM3", "VM4"],
      "nature": "internal"
    },
    {
      "name": "public_1",
      "vn_id": "4f1e0ab6-af04-456c-8ba1-3edbbdd432ce",
      "Status": "Active",
      "MTU": "1400",
      "network_address": "172.24.4.0/24",
      "attached_devices": ["net2", "net3"],
      "nature": "public"
    }
  ]
}
```

```

    ]
}

```

### ***Create virtual network method in Neo4JController.py:***

The algorithm defines a function `create_virtualnetwork()` that takes the `parent_name`, `name`, and optional properties as input. Inside this function, a nested function `_create_virtualnetwork()` is defined, which performs the database operation. The `_create_virtualnetwork()` function constructs a query string and executes it using the provided parameters. Finally, the result is processed and returned using the `Nvivn.from_neo4j_response()` method, which is called with the result of the `Neo4JController.write_transaction()` method as input.

---

#### **Algorithm: Function to create virtual network**

---

- Input:** parent id, name, properties={}
- Output:** partial creation of the virtual network node
1. Function `create_virtualnetwork(parent_name, name, properties={})`:
  2. Define a nested function `_create_virtualnetwork(tx, _name, _parent_name)`:
  3. Execute the query using the following steps:
  4. `CREATE(h:{parent_name}:Virtualnetwork{name: $name})`
  5. `Neo4JController.generate_query_for_properties('h', properties)`
  6. `RETURN ID(h), h.name`
  7. Execute the query using `tx.run(query, parent_name=_parent_name, name=_name, **properties)`
  8. Return the single result from the query
  9. Call the `Neo4JController.write_transaction()` method with `_create_virtualnetwork`, `name`, and `parent_name` as arguments
  10. Call the `Nvivn.from_neo4j_response()` method with the result of the `Neo4JController.write_transaction()` method
  11. Return the result from the `Nvivn.from_neo4j_response()` method
  12. End.

### ***Format of R: The Set of Routers***

```

{
  "vrouters": [
    {
      "name": "VR1",
      "routerID": "9f3475ef-e390-4541-a30f-0d1bd0456848",
      "status": "Active",
      "interfaces": ["172.24.4.208", "192.168.1.1", "192.168.0.1"],
      "ex_gateway": "172.24.4.208"
    }
  ]
}

```



```
]
}
```

***Create virtual router method in Neo4JController.py:***

The algorithm defines a function `create_router()` that takes the `parent_name`, `name`, and optional properties as input. Inside this function, a nested function `_create_router()` is defined, which performs the database operation. The `_create_router()` function constructs a query string and executes it using the provided parameters. Finally, the result is processed and returned using the `NetworkRouter.from_neo4j_response()` method, which is called with the result of the `Neo4JController.write_transaction()` method as input.

---

**Algorithm: Function to create virtual router**

---

- Input:** parent id, name, properties={}
- Output:** partial creation of the virtual router node
1. Function `create_router(parent_name, name, properties={})`:
  2. Define a nested function `_create_router(tx, _name, _parent_name)`:
  3.     Execute the query using the following steps:
  4.     `CREATE(h:{parent_name}:Router{name: $name})`
  5.     `Neo4JController.generate_query_for_properties('h', properties)`
  6.     `RETURN ID(h), h.name`
  7.     Execute the query using `tx.run(query, parent_name=_parent_name, name=_name, **properties)`
  8.     Return the single result from the query
  9.     Call the `Neo4JController.write_transaction()` method with `_create_router`, `name`, and `parent_name` as arguments
  10. Call the `NetworkRouter.from_neo4j_response()` method with the result of the `Neo4JController.write_transaction()` method
  11. Return the result from the `NetworkRouter.from_neo4j_response()` method
  12. End.

## 5.2. Creating Links:

***Create link method in Neo4JController.py:***

The algorithm defines a function `createnlink_vn_to_rtr()` that takes `nvi`(network virtualization instance), `vn`(virtual network), `rtr`(router), and optional properties as input. Inside this function, a nested function `_createnlink_vn_to_rtr()` is defined, which performs the database operation. The `_createnlink_vn_to_rtr()` function constructs a query string to match nodes of the specified types and create a relationship between them. The query is executed using the provided parameters, and the result is stored in the "result" variable as a dictionary.

The function then tries to execute the `Neo4JController.write_transaction()` method with the `_createnvilink_vn_to_rtr` function and the provided arguments. If the transaction is successful, `True` is returned. Otherwise, the error is printed, and `False` is returned.

---

#### Algorithm: Function to create links

---

**Input:** `nvi, vn, rtr` properties={}

**Output:** creation of relationship between virtual network to router

1. Function `createnvilink_vn_to_rtr(nvi, vn, rtr, properties={})`:
2. Define a nested function `_createnvilink_vn_to_rtr(tx, _nvi, _vn, _rtr)`:
3.     Execute the query using the following steps:
4.     Set the query to match nodes of type `nvi` and `rtr` and create a relationship between them
5.     Use `MATCH` to find nodes with `vn_id = vn` and `routerID = rtr`
6.     Use `MERGE` to create a relationship (a)-[r:LINK]->(b)
7.     Append `Neo4JController.generate_query_for_properties('r', properties)`
8.     Execute the query using `tx.run(query, nvi=_nvi, vn=_vn, rtr=_rtr, **properties)` and store the result in the "result" variable as a dictionary
9.     Try executing the `Neo4JController.write_transaction()` method with `_createnvilink_vn_to_rtr`, `nvi`, `vn`, and `rtr` as arguments, and store the result in the "response" variable
11.    If all transactions are successful, return `True`
12.    If an exception occurs, print the error and return `False`

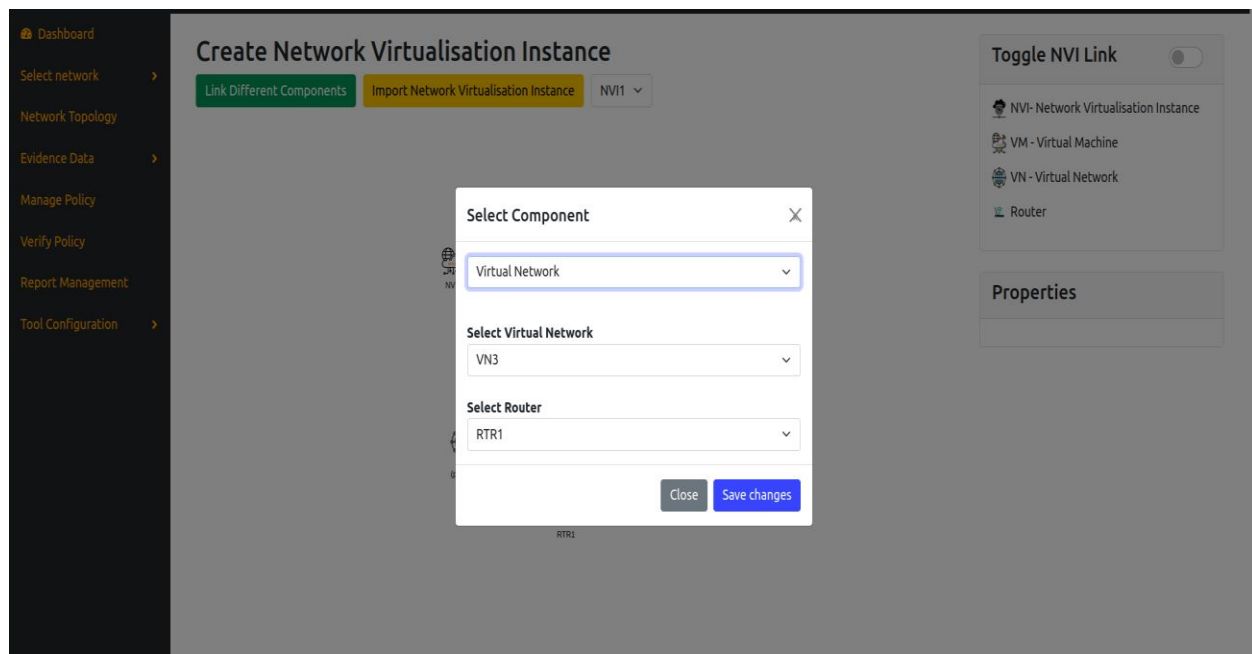


Figure 36: Linking between nodes in a NV topology

## 5.3. Visualization of Network:

### *Create and Get NVI method in Neo4JController.py:*

The algorithm defines two functions: `create_nvi()` and `get_nvi()`.

The `create_nvi()` function takes the name and optional properties as input. Inside this function, a nested function `_create_nvi()` is defined, which performs the database operation. The `_create_nvi()` function constructs a query string and executes it using the provided parameters. Finally, the result is processed and returned using the `Network.from_neo4j_response()` method.

The `get_nvi()` function is a static method that retrieves all VNI nodes from the database. Inside this method, a nested function `_get_nvi()` is defined, which executes the query to retrieve the NVI nodes. The values from the query execution are processed, and a list of Network objects is returned, created using the `Network.from_neo4j_response()` method for each response.

---

**Algorithm: Function to create network virtualization instance**

---

- Input:** name, properties={}
- Output:** creation of the network virtualization instance
1. Function `create_nvi(name, properties={})`:
  2. Define a nested function `_create_nvi(tx, _name)`:
  3. Execute the query using the following steps:
  4. `CREATE (n:NVI {name: $name})` followed by `Neo4JController.generate_query_for_properties('n', properties)` and `RETURN ID(n), n.name`
  5. Execute the query using `tx.run(query, name=_name, **properties)`
  6. Return the single result from the query
  7. Call the `Neo4JController.write_transaction()` method with `_create_nvi` and `name` as arguments
  8. Call the `Network.from_neo4j_response()` method with the result of the `Neo4JController.write_transaction()` method as input
  9. Return the result from the `Network.from_neo4j_response()` method
  10. Static method `get_nvi()`:
  11. Define a nested function `_get_nvi(tx)`:
  12. Execute the query to retrieve all VNI nodes using `tx.run("MATCH (n:NVI) RETURN ID(n), n.name")`
  13. Return the values from the query execution
  14. Call the `Neo4JController.read_transaction()` method with `_get_nvi` as an argument and store the response in the "result" variable
  15. Iterate over each response in the "result" variable
  16. Call the `Network.from_neo4j_response()` method with the response as input and create a Network object for each response
  17. Return a list of the Network objects

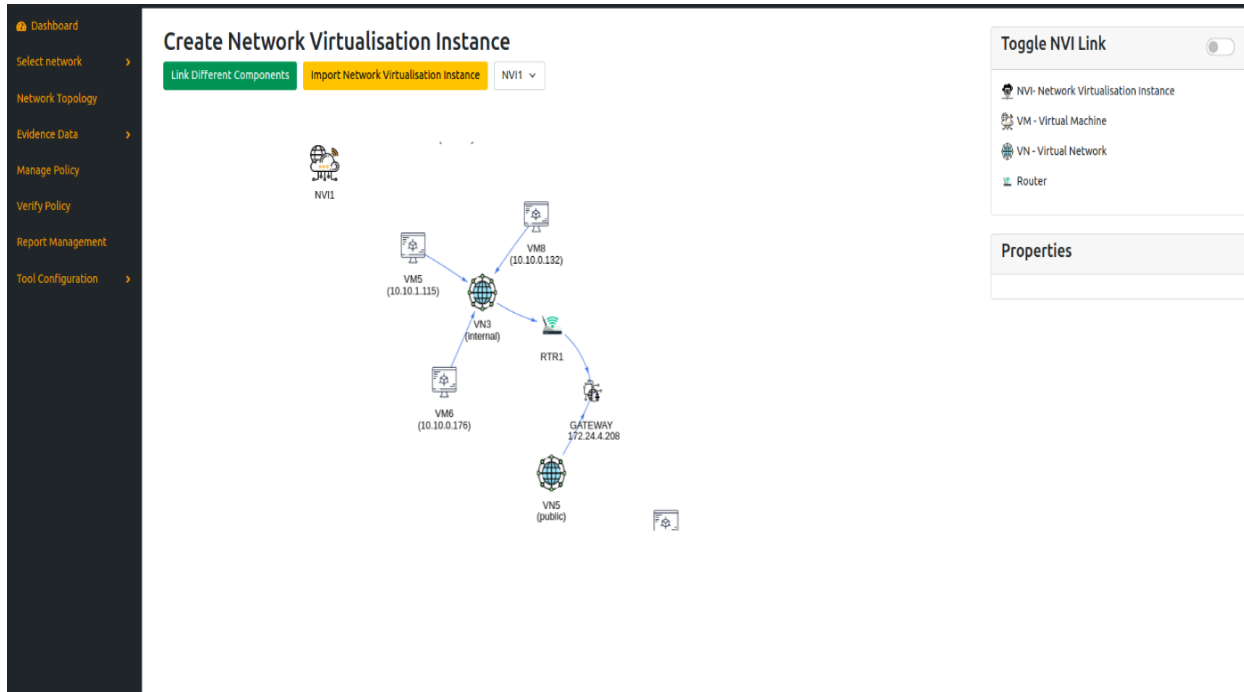


Figure 37: Visualization of NV Topology in the tool

## 5.4. Updating

*Update node method in Neo4JController.py:*

The algorithm defines a function `updatenode()` that takes the data dictionary and optional properties as input. Inside this function, the `network_id` and `id` are extracted from the data dictionary. Then, a nested function `_updatenode()` is defined, which performs the database update operation. The `_updatenode()` function constructs a query string based on the provided data and properties, and executes the query using the `write_transaction()` method. If the transaction is successful, the function returns `True`. Otherwise, it prints the error message and returns `False`.

---

### Algorithm: Function to update nodes

---

**Input:** data, properties={}

**Output:** updated nodes

1. Function `updatenode(data, properties={})`:
2. Extract the "network\_id" and "id" from the data dictionary and assign them to `network_id` and `id` variables, respectively.

3. Remove "network\_id" and "id" keys from the data dictionary.
4. Define a nested function \_updatenode(tx, \_id):
5.     Construct the query string using the following steps:
6.         MATCH (n) WHERE id(n) = id
7.         "SET" followed by a comma-separated list of property updates using data dictionary values
8.         Neo4JController.generate\_query\_for\_properties('n', properties)
9.         Execute the query using tx.run(query\_string, id=\_id, \*\*properties)
10.        Return the result and the query\_string as a dictionary
11. Try executing the Neo4JController.write\_transaction(\_updatenode, id) block:
12.     If successful, return True
13.     If an exception occurs, print the error and return False



Figure 38: Updating node properties in a NV Topology

## 5.5. Deleting

*Delete node method in Neo4JController.py:*

The algorithm defines a function delete\_node() that takes the node ID and an optional delete\_all\_nodes\_relationship parameter as input. Inside this function, there are two nested functions: \_delete\_all\_nodes\_relationships and \_delete\_node. The \_delete\_all\_nodes\_relationships function deletes all nodes and relationships connected to the specified node ID, while the \_delete\_node function deletes only the specified node. The appropriate nested function is executed based on the value of delete\_all\_nodes\_relationship using the Neo4JController.write\_transaction() method.

---

**Algorithm: Function to delete nodes**

---

**Input:** id, delete\_all\_nodes\_relationship=False

**Output:** delete nodes

1. Function delete\_node(id, delete\_all\_nodes\_relationship=False):
2. Define a nested function \_delete\_all\_nodes\_relationships(tx, \_id):
3.     Execute the query using the following steps:
4.         Set the query to match all nodes and relationships connected to the node with ID = id
5.         Use OPTIONAL MATCH to also match relationships connected to the connected nodes
6.         Delete the relationships and the connected nodes
7.         Execute the query using tx.run(query, id=\_id)
8. Define a nested function \_delete\_node(tx, \_id):
9.     Execute the query using the following steps:
10.         Set the query to match the node with ID = id
11.         Use DETACH DELETE to delete the node and its relationships
12.         Execute the query using tx.run(query, id=\_id)
13. Check if delete\_all\_nodes\_relationship is True:
14.     If true, call the Neo4JController.write\_transaction() method with \_delete\_all\_nodes\_relationships and id as arguments
15.     Otherwise, call the Neo4JController.write\_transaction() method with \_delete\_node and id as arguments

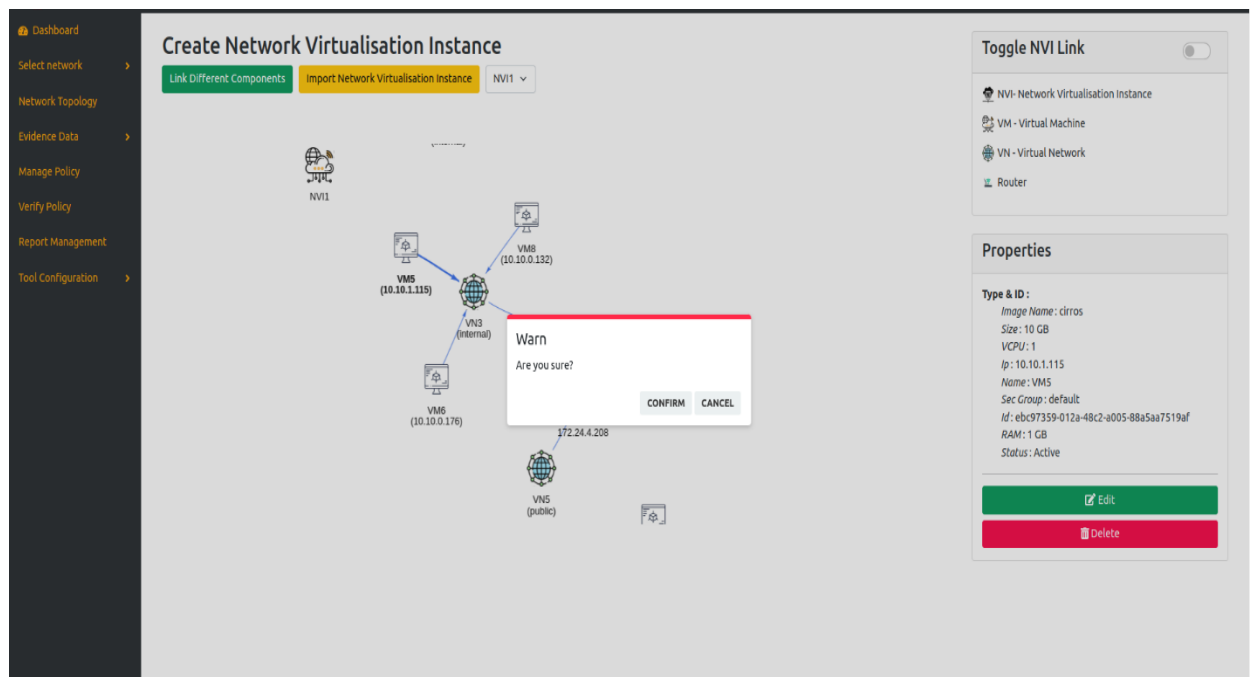


Figure 39: Deletion of a node in a NV Topology

# Chapter 6: Path Analysis Using Neo4J

## *Function in Neo4JController.py:*

The algorithm defines a function `get_shortest_path()` that takes `hid1` and `hid2` as input. Inside this function, a nested function `_get_shortest_path()` is defined, which performs the database operation. The `_get_shortest_path()` function constructs a query to find the shortest path between the specified node IDs. The query is executed using the provided parameters, and the result is stored in the "result" variable.

The function then calls the `Neo4JController.read_transaction()` method with the `_get_shortest_path` function and the `hid1`, `hid2` arguments. The response is then converted to JSON format using the `json.dumps()` method and returned.

---

**Algorithm: Function to find shortest path between two nodes**

---

**Input:** `hid1`, `hid2`

**Output:** shortest path between `hid1` and `hid2`

1. Function `get_shortest_path(hid1, hid2)`:
2. Define a nested function `_get_shortest_path(tx, _hid1, _hid2)`:
3.     Execute the query using the following steps:
4.     Set the query to find the shortest path between two nodes with IDs `hid1` and `hid2`
5.     Use `MATCH` to find the shortest path between start and end nodes
6.     Return the nodes and relationships of the path as "nodes" and "links"
7.     Execute the query using `tx.run(query, hid1=_hid1, hid2=_hid2)` and store the result in the "result" variable
8. Call the `Neo4JController.read_transaction()` method with `_get_shortest_path`, `hid1`, and `hid2` as arguments, and store the response in the "response" variable
9. Convert the "response" to JSON format using the `json.dumps()` method
10. Return the JSON-formatted response

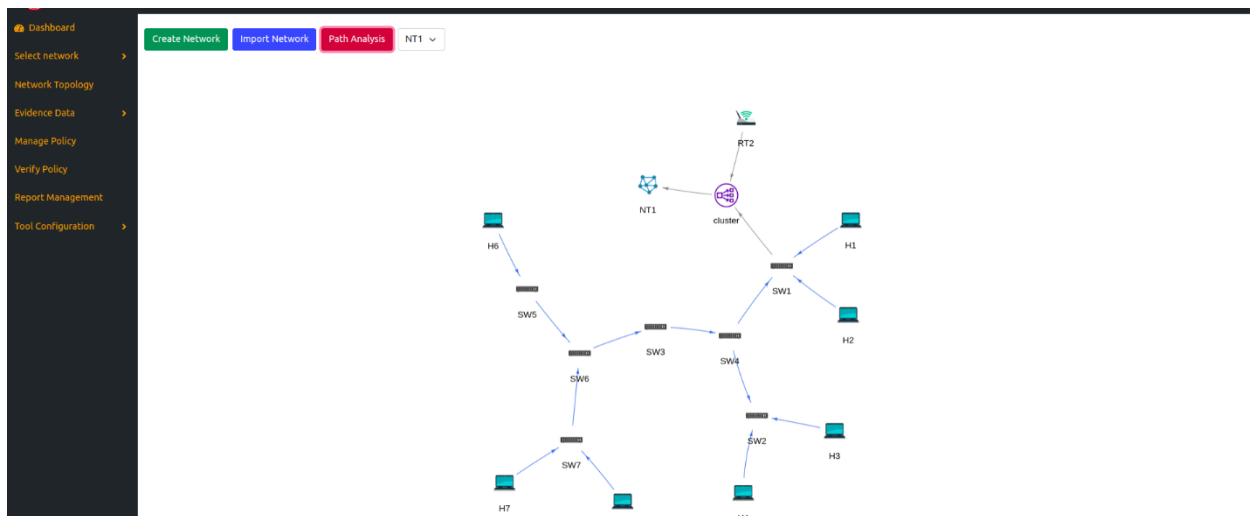


Figure 40: The Network on which we will do Path Analysis

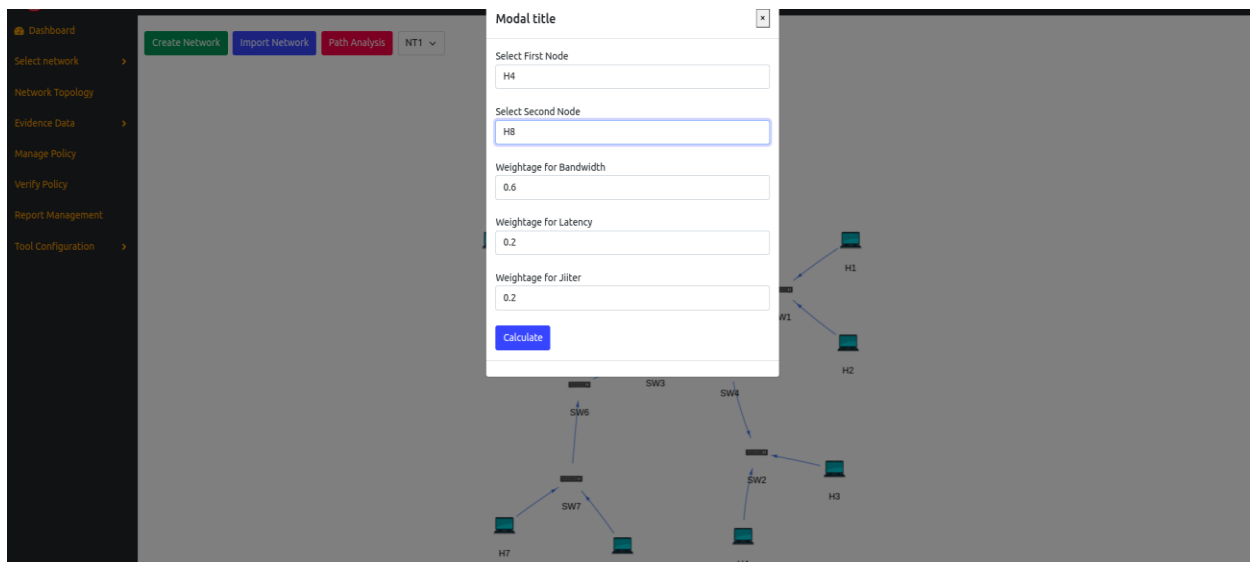


Figure 41: Path Details

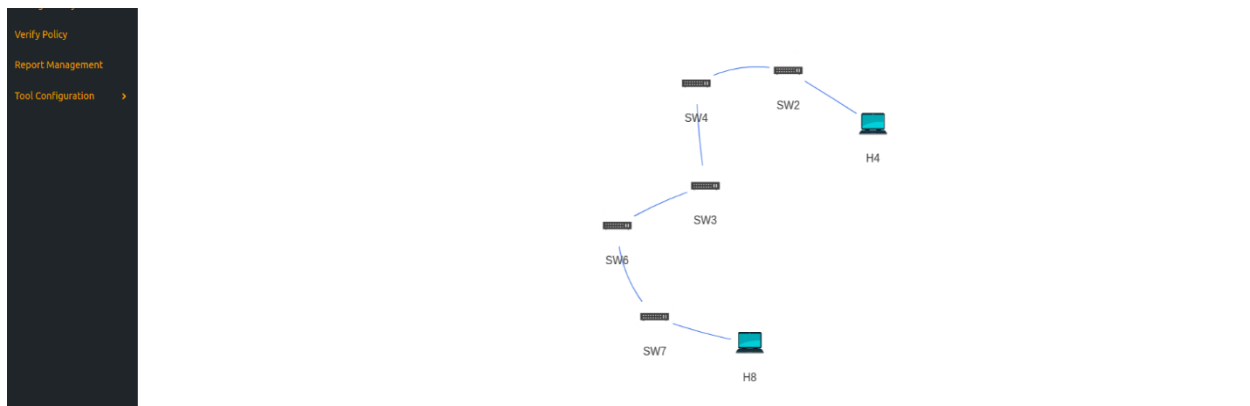


Figure 42: The Shortest Path



# Chapter 7: Cost Analysis Using Neo4J

## 7.1. Calculation

*Methods: Calculation of cost of the shortest path between two nodes.*

---

**Algorithm:** To Calculate Cost

---

**Input:** edgeId, property, value, bw (bandwidth weightage), jw (jitter weightage), lw(latency weightage)

**Output:** cost of path

1. Declare a global variable called linkData and set it as an empty object.
2. Declare a function called updateLinkProperty(edgeId, property, value):
3. Retrieve the edge object using the edgeId from the edges collection.
4. Update the specified property of the edge object with the provided value.
5. Update the edge in the edges collection.
6. Store the link data in the linkData object, including bandwidth, latency, and jitter values.
7. Declare a variable called cost and calculate the cost using the calculateCost function with the linkData object.
8. Retrieve the cost <div> element using its ID ("cost") and assign it to the costDiv variable.
9. Set the innerHTML of the costDiv to display the calculated cost.
10. Declare a variable called edges.
11. Declare a function called calculateCost(linkData):
12. Initialize a variable called totalCost as 0.
13. Initialize a variable called count as 0.
14. Iterate through each edgeId in the linkData object:
  15. Retrieve the link object using the edgeId.
  16. Calculate the cost based on the provided weights (bw, lw, jw) and the corresponding link properties (bandwidth, latency, jitter).
  17. Add the calculated cost to the totalCost.

18. Increment the count.
19. Check if the count is greater than 0:
20. If true, calculate the average cost by dividing the totalCost by the count and return the result.
21. If false, return the totalCost.
22. If the totalCost is NaN, return "Calculating..." as the result.

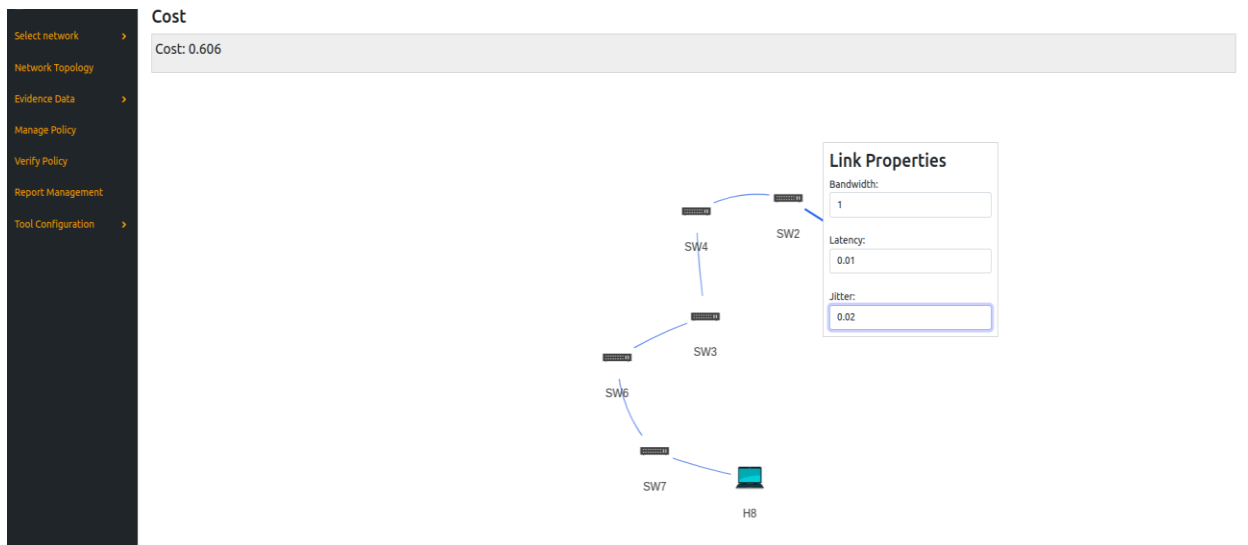


Figure 43: Taking Link Properties

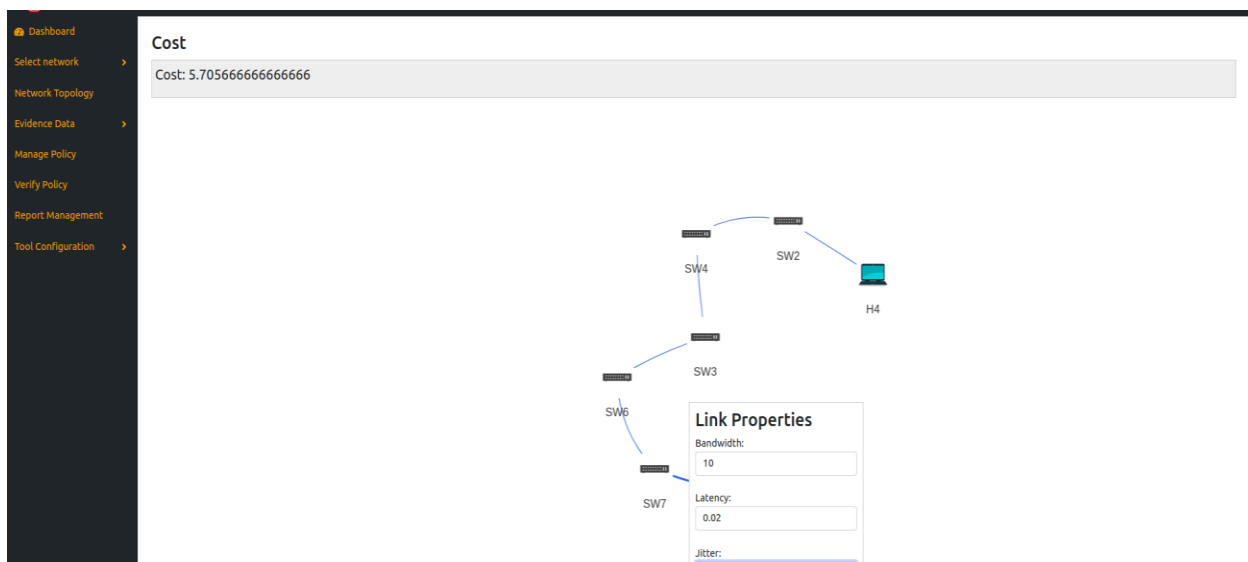


Figure 44: Calculation of Cost

## 7.2. Analysis

**Case 1:** The data table and the corresponding graph. In this case, we have changed the weightage for bandwidth, latency and jitter and kept the data of the links unchanged. Now we see how the cost of the path changes in this case. (The graph is W vs C.).

Table 1: Cost Measurement by giving weightage to bandwidth, jitter and latency

Bandwidth weightage, w1	Latency weightage, w2	Jitter Weightage, w3	Modified weightage, $W=w1*w2*w3$	C= Avg. Cost
0.6	0.2	0.2	0.024	7.6
0.4	0.3	0.3	0.036	6.4
0.8	0.1	0.1	0.008	8.8
0.2	0.4	0.4	0.032	5.2

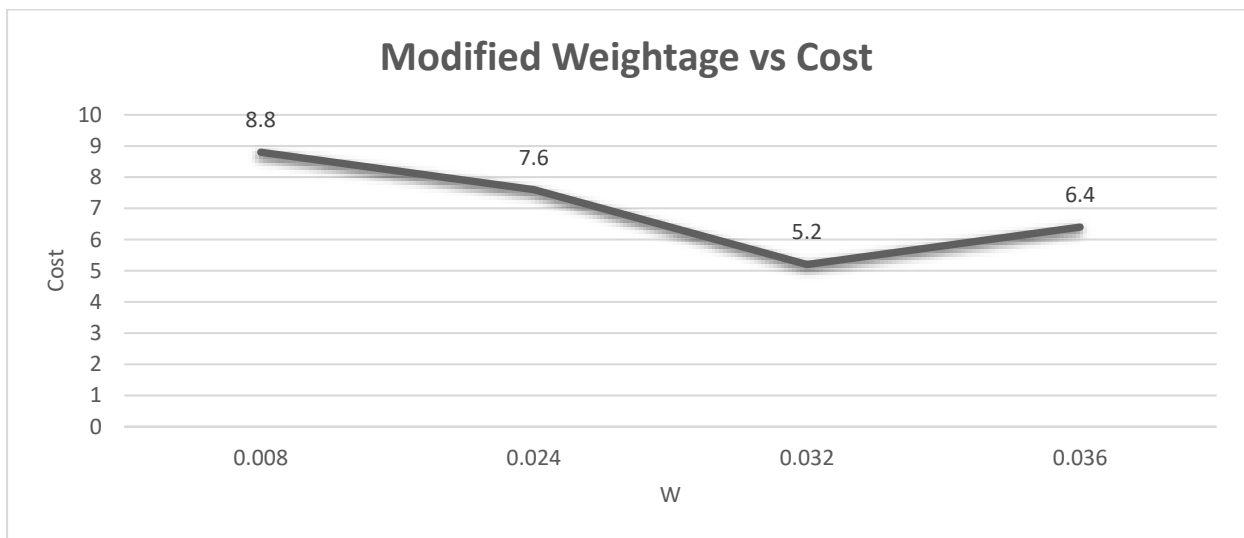


Figure 45: Modified Weightage Vs Cost Graph

### Observation:

We see that when one of the weightage becomes too large compared to the rest two, then there is a hike in the cost change. When the difference between the weightages are small, the cost changes smoothly.

### Case 2:

In this case, we keep the weightage same, but we change the bandwidth and latency of a link. The data and corresponding graph is given below:

Table 2: Cost and Bandwidth

Cost	Bandwidth
7.6	10
10.6	20
13.6	30
16.6	40

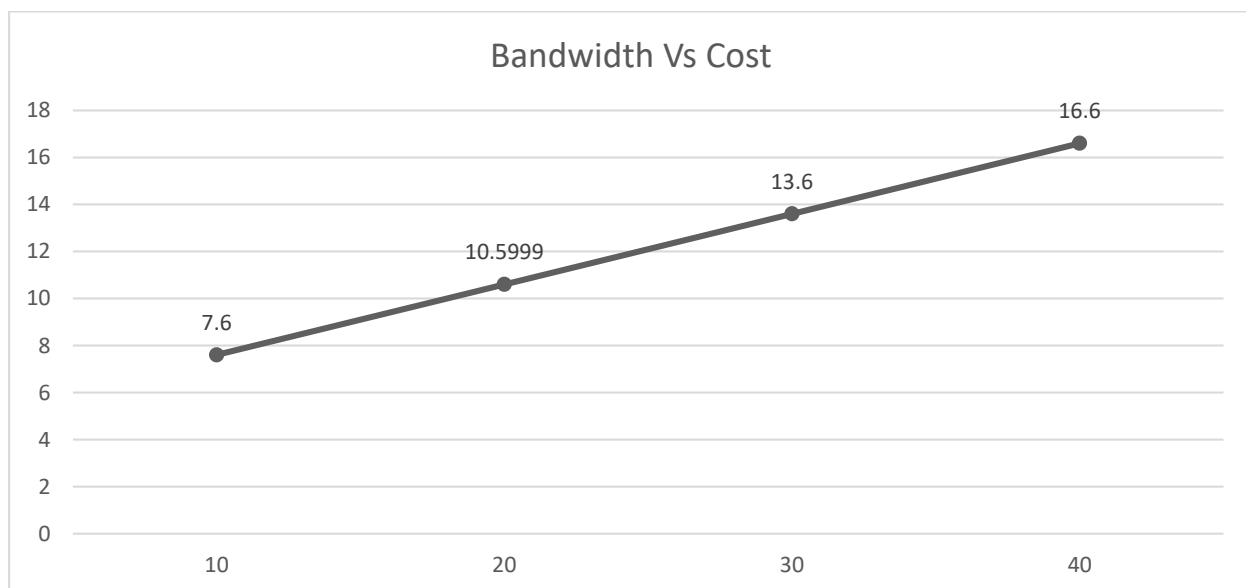


Figure 46: Bandwidth Vs Cost Graph

### Observation:

We see that when bandwidth increases, the average cost also increases, (taking weightages unchanged as a constraint). Similarly, if the bandwidth decreases, the cost will decrease.

Table 3: Cost and Latency

Cost	Latency
7.4	6
7.6	7
7.8	8
7.9	9

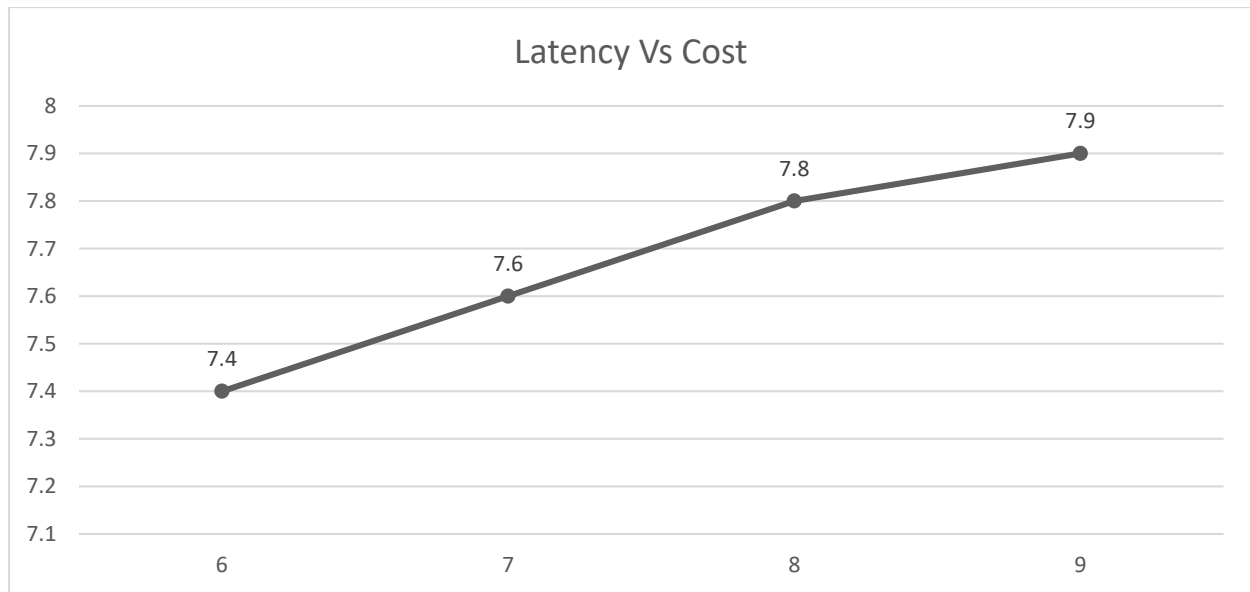


Figure 47: Latency Vs Cost Graph

*Observation:*

We see that when latency increases, the average cost also increases, (taking weightages unchanged as a constraint). This means if the delay increases, the cost increases.

## Chapter 8: Conclusion and Future Scope

This project focused on three major domains – modelling the network topology, querying them using Cypher language and analyzing the path and respective cost between two nodes. Modelling is nothing but creating a logical viewpoint and abstraction of physical entities. Here, we have done that only. Taking into consideration the main three domains of virtualization of network infrastructure, SDN, NFV, and NV, we have thoroughly gone through the theoretical perspectives and the enhancement of modern technology. But, main problems are unsolved like we have stated the VNF-Chain Placement Problem. Future works should come up with better and optimized algorithms. Also, implementation of several policies like mutual authentication policy, NAT-PT policy, network segregation policy, etc. in the SDN controller plane will be an enhancement of the project. Similarly, developing new virtual network functions in the NFV infrastructure and working upon them will be a next step of this project. Calculation of minimum cost with better Quality of Service should be achieved. And, the most important two factors that should be the key foci of any developer are nothing but –Security and Privacy.

# References

- [1] G. R. P and M. P. S, "Virtual Machine Technology: A Bridge From Large Mainframes To Networks Of Small Computers," in *Comcon Fall 79. Proceedings*, Washington, 1979.
- [2] R. M. and G. T., "Virtual machine monitors: current technology and future trends," *Computer*, vol. 38, no. 5, pp. 39-47, 2005.
- [3] L. A. Barroso and U. Holzle, "The Case for Energy-Proportional Computing," *Computer*, vol. 40, no. 12, pp. 33-37, December 2007.
- [4] N. Omnes, M. Bouillon and e. al, "A programmable and virtualized network & IT infrastructure for the internet of things: How can NFV & SDN help for facing the upcoming challenges," in *International Conference on Intelligence in Next Generation Networks*, Paris, 2015.
- [5] M. D. Mattei and S. Eric, "Integrating Virtualization and Cloud Services into a Multi-Tier, Multi-Location Information System Business Continuity Plan," *Journal of Strategic Innovation and Sustainability*, vol. 11, no. 2, pp. 70-81, 2016.
- [6] N. Rathos and A. Surve, "Test orchestration a framework for Continuous Integration and Continuous deployment," in *International Conference on Pervasive Computing (ICPC)*, Pune, 2015.
- [7] F. Lombardi and R. D. Pietro, "Secure virtualization for cloud computing," *Journal of Network and Computer Applications*, vol. 34, no. 4, pp. 1113-1122, 2011.
- [8] Y. Jin, Y. Wen and C. Qinghua, "Energy efficiency and server virtualization in data centers: An empirical investigation," in *Proceedings IEEE INFOCOM Workshops*, Orlando, 2012.
- [9] R. Bolla, C. Lombardo, R. Bruschi and M. S, "DROPv2: energy efficiency through network function virtualization," *Network*, vol. 28, no. 2, pp. 26-32, 2014.
- [10] R. McDougall and J. Anderson, "Virtualization performance: perspectives and challenges ahead," *SIGOPS Operating Systems Review*, vol. 44, no. 4, pp. 40-56, 2010.
- [11] S. Loveland, E. Dow and F. LeFevre, "Leveraging virtualization to optimize high-availability system configurations," *IBM Systems Journal*, vol. 47, no. 4, pp. 591-604, 2008.
- [12] G. Neiger, A. Santoni, F. Leung and e. al, "Intel Virtualization Technology: Hardware support for efficient processor virtualization," *Intel Technology Journal*, vol. 10, no. 3, pp. 167-177, 2006.

- [13] L. Helali and M. Omri, "Software License Consolidation and Resource Optimization in Container-based Virtualized Data Centers," *J Grid Computing*, vol. 20, no. 13, 2022.
- [14] O. AbdelRahem, Bahaa-Eldin, A. M. and A. Taha, "{Virtualization security: A survey," in *11th International Conference on Computer Engineering & Systems (ICCES)*, 2016.
- [15] J. Daniels, "Server virtualization architecture and implementation," *The ACM Magazine for Students*, vol. 16, no. 1, pp. 8-12, 2009.
- [16] L. Yan, "Development and application of desktop virtualization technology," in *IEEE 3rd International Conference on Communication Software and Networks*, 2011.
- [17] R. Boutaba and K. M. N. M. Chowdhury, "A survey of network virtualization," in *Computer Networks*, 2010, pp. 826-876.
- [18] A. Singh, M. Korupolu and D. Mohapatra, "Server-storage virtualization: Integration and load balancing in data centers," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.
- [19] L. Zhang, Z. Yang, Y. He, M. Li, S. Yang, M. Yang, Y. Zhang and Z. Qian, "App in the Middle: Demystify Application Virtualization in Android and Its Security Threats," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 3, no. 1, 2019.
- [20] L. Oren and J. Nieh, "Operating system virtualization: practice and experience," in *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, 2010.
- [21] G. Pék, L. Buttyán and B. Bencsáth, "A survey of security issues in hardware virtualization," *ACM Comput. Surv.*, vol. 45, no. 3, pp. 300-360, 2013.
- [22] W. Xia, Y. Wen, H. Foh, D. Niyato and H. Xie, "A Survey on Software-Defined Networking," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 27-51, 2015.
- [23] B. Yi, X. Wang, K. Li, D. Sajal and M. Huang, "A comprehensive survey of Network Function Virtualization," *Computer Networks*, vol. 133, pp. 212-262, 2018.
- [24] King and J. Leslie, "Centralized versus Decentralized Computing: Organizational Considerations and Management Options," *ACM Comput. Surv.*, vol. 15, no. 4, pp. 319-349, 1983.
- [25] N. N. S. Ahituv and M. Zviran, "Factors Affecting the Policy for Distributing Computing Resources," *MIS Quarterly*, vol. 13, no. 4, pp. 389-401, 1989.
- [26] D. B. a. R. S. R. Rawat, "Software Defined Networking Architecture, Security and Energy Efficiency: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 1, pp. 325-346, 2017.



- [27] M. F. Bari, A. R. Roy, R. Chowdhury S and Q. Zhang, "Dynamic controller provisioning in Software Defined Networks," in *Proc. of 9th Int. Conference on Network and Service Management*, 2013.
- [28] M. I. Lali, R. U. Mustafa, M. S. Nawaz and W. Aslam, "Performance Evaluation of Software Defined Networking vs. Traditional Networks," *The Nucleus*, vol. 54, no. 1, 2017.
- [29] Medved, V. R, T. A and G. K, "OpenDaylight: Towards a model-driven SDN controller architecture," in *Proc. of 15th Int. Symposium on A World of Wireless, Mobile and Multimedia Networks*, 2014.
- [30] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford and S. S. a. J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69-74, 2008.
- [31] Y. S. Wang, C. C. L and Y. C. M, "EstiNetOpenFlow network simulator and emulator," *IEEE Communications Magazine*, vol. 51, no. 9, pp. 110-117, 2013.
- [32] H. Q, F. Hu and B. K, "A survey on Software Defined Networking (SDN) and OpenFlow: From concept to implementation," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2181-2206, 2015.
- [33] T. Turletti and K. Obraczka, "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 3, pp. 1617-1634, 2014.
- [34] T. P and V. D, "The Northbound APIs of Software Defined Networks," *INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY*, vol. 5, no. 10, pp. 501-513, 2016.
- [35] V. P. L, T. M, S. S and Filsfils, "SDN Architecture and Southbound APIs for IPv6 Segment Routing Enabled Wide Area Networks," *IEEE Transactions on Network and Service Management*, vol. 15, no. 4, pp. 1378-1392, 2018.
- [36] N. McKeown, A. T, P. L and R. J, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Commun. Review*, vol. 38, no. 2, pp. 69-74, 2008.
- [37] F. Schardong and I. Nunes, "NFV Resource Allocation: a Systematic Review and Taxonomy of VNF Forwarding Graph Embedding,," *Computer Networks*, vol. 185, 2021.
- [38] S. Khebbache, M. Hadji, Zeghlache and Djamal, "Virtualized network functions chaining and routing algorithms," *Computer Networks*, vol. 114, pp. 95-110, 2017.
- [39] K. B and V. J, *Combinatorial Optimization: Theory and Algorithms*, Springer, 2007.

- [40] L. Lovász, "Review of the book by alexander schrijver: combinatorial optimization: polyhedra and efficiency," *Oper. Res. Lett*, vol. 33, no. 4, pp. 437-440, 2005.
- [41] S. D. B, A. Rinnooy and J. K. Lenstra, *The Traveling Salesman Problem : A Guided Tour of Combinatorial Optimization*, Wiley, 1985.
- [42] Virtualization Overview.
- [43] B. P, D. B and F. K, "Xen and the art of virtualization," in *SOSP'03: Proc. the Nineteenth ACM Symp. Operating*, 2003.
- [44] V. M and K. R, "A virtualization technologies primer: Theory," in *Network Virtualization*, Cisco, 2006.
- [45] J. X and X. D, "VIOLIN: Virtual Internetworking on Overlay," 2003.
- [46] T. T, "Framework and Requirements for Layer 1 Virtual Private Networks," *NTT*, 2007.
- [47] N.-K. T, "VPN in brief," in *Building VPNs: With IPsec and*, McGraw-Hill Professional, 2003, pp. 9-11.
- [48] A. Wang, G. N. Rouskas, M. Iyer and R. Dutta, "Network Virtualization: Technologies,," *JOURNAL OF LIGHTWAVE TECHNOLOGY*, vol. 31, no. 4, 2013.
- [49] R. Angles and C. Gutierrez, "Survey of Graph Database Models," *ACM Computer Surveys*, vol. 40, no. 1, pp. 1-39, 2008.
- [50] Vandana and P. R. Singh, "Application of Graph Theory in Computer Science and Engineering," *International Journal of Computer applications*, vol. 104, no. 1, 2014.
- [51] Y. n, F. X and S. M, "Design and Implementation of Movie Recommender System Based on Graph Database," in *14th Web Information Systems and Applications Conference (WISA)*, 2017.
- [52] M. PW and R. D., "Comparison of Relational Database and Graph Database in the context of web application development," in *Proceedings of 36th International Conference on Information Systems Architecture and Teechnology*, 2016.
- [53] B. K, M. H and B. G. Comparative, " Comparative Study Between t h e MySQL Relational Database and theMongoDB NoSQL Database," *International Journal of Software Science and Computational Intelligence*, vol. 13, no. 3, pp. 38-63, 2021.
- [54] J. Miller, "Graph Database Applications and Concepts with Neo4J," in *SAIS 2013 Proceedings*.

- [55] H. M and F. H, "An Approach to Converting Relational Database to Graph Database: from MySQL to Neo4j," in *IEEE 2nd International Conference on Power, Electronics and Computer Applications (ICPECA)*, 2022.