Dédicaces

A Dieu qui m'a donné la vie. Ainsi que de m'avoir aidé à réaliser ce mémoire

A mes très chers parents qui ont fait de moi la personne qui je suis aujourd'hui.

*À tous les étudiants de l'**ENSTICP** et mes enseignants*

A ceux qui ont marqué leur présence dans ma vie.

A ceux qui sont en train de lire cette dédicace.

A moi-même.

ASSIA

Remerciement

Louange à Ellah, le tout puissant pour nous avoir donné la santé, le courage et la volonté, pour réaliser ce modeste travail.

Je remercie chaleureusement ma famille, mes parents, en premier lieu, pour leur soutien moral, leurs encouragements et leur patience durant les étapes difficiles de ce travail.

Je ne saurais jamais assez remercier mon encadreur Melle BOUCHAIR Maria, Ingénieur Réseau à ERICSSON. Je tiens à la remercier chaleureusement pour ses conseils, ses aides et de m'avoir fait confiance en me confiant la réalisation de ce projet d'actualité et de m'avoir réservé le meilleur accueil malgré ses obligations professionnelles.

J'adresse mes remerciements les plus sincères à mes co-encadreurs Monsieur TALEB Ahmed et Madame BENNOUR Zakiya pour ses précieux conseils et leurs aides permanentes.

Mes vifs remerciements vont aux membres du jury pour avoir accepté d'examiner et d'évaluer le travail réalisé dans ce mémoire.

Je tiens à remercier aussi l'ensemble des enseignants qu'ils nous ont prodigués depuis notre arrivée à l'ENSTICP pour leurs soutiens.

Enfin, je témoigne mes remerciements à tous ceux qui n'ont pas été cités mais qui ont participé de près ou de loin à la réussite de ce travail, ils trouvent ici l'expression de notre profonde gratitude.

Table des matières

Chapitre III : Docker et Kubernetes

Chapitre IV : Mise en place de la solution

Chapitre V : Étude managériale

Liste des acronymes

A

API Application Programming Interface

B

BSS Business Support Systems

C

CNF Container Network Function

CD Continuous delivery

CI Continuous integration

CPU Central Processing Unit

D

DCC Digitalization& Cloud Computing

E

ETSI European Telecommunications Standards Institute

F

FORCES Forwarding and Control Element Separation

FTP File Transfer Protocol

H

HPA Horizontal Pod Autoscaler

HDS Hyperscale Datacenter System

I

IP Internet Protocol

IT Information Technology

ICMP Internet Control Message Protocol

K

K8S Kubernetes

L

LXC Linux Containers

N

NFV Network function virtualization

NFVI Network Functions Virtualization Infrastructure

NF Network Function

NMSD Network Management Service Delivery

O

OSS Operations Support System

ONF Open Networking Foundation

OVS Open vSwitch

P

PCE Path Computation Element

PFE Projet de fin d'étude

Q

QOS Quality of Service

R

RAM Random Access Memory

S

SDN Software defined Network

T

TCP Transmission Control Protocol

V

VM Virtual Machine

VNF Virtualized Network Functions

VIM Virtualized Infrastructure Manager

VPA Vertical Pod Autoscaler

W

WBS Work breakdown structure

Y

YAML Yet Another Markup Language

Liste des figures

Chapitre I : Technologies SDN et NFV : Généralités et concepts

Chapitre II : Microservices et conteneurs

Chapitre III : Docker et Kubernetes

Chapitre IV : Mise en place de la solution

Chapitre V : Étude managériale

Liste des tableaux

Chapitre I: Technologies SDN et NFV : Généralités et concepts

Chapitre II : Microservices et Conteneurs

Chapitre IV : Mise en place de la solution

Annexe C :

Introduction Générale

Les technologies de réseaux programmables SDN (Software defined Network) et la virtualisation des fonctions réseau (NFV) ont le vent en poupe et leur adoption s'accélère que ce soit au sein du centre de données d'entreprises, des réseaux de fournisseurs de services ou des réseaux d'opérateurs.

Le SDN sépare le plan de contrôle (qui décide où le trafic est envoyé) du plan de données (qui transfert les paquets vers des destinations spécifiques) en rendant le réseau entièrement programmable. Il s'appuie pour cela sur des commutateurs programmés par un contrôleur SDN qui utilise un protocole standard comme OpenFlow. Cette solution permet aux administrateurs réseau de gérer les services réseaux grâce à un logiciel (contrôleur) qui centralise sa programmation et offre une configuration bien plus rapide et conviviale. Toutefois, malgré les avantages de SDN, l'utilisation d'un seul contrôleur centralisé a soulevé de nombreux défis, notamment le contrôleur peu agir comme un point de défaillance unique ainsi le volume de trafic sur les grands réseaux peut surcharger le contrôleur, pour relever ces défis, la distribution des contrôleurs SDN a été proposée pour réduire les problèmes des contrôleurs centralisés. Cependant, le cas de déploiement distribué utilisant plusieurs contrôleurs nécessite une augmentation énorme des ressources.

Dans l'architecture NFV, les fonctions réseau virtualisées (VNF) sont des applications logicielles qui fournissent des fonctions réseau telles que le partage de fichiers, le routage, la mise en pare-feu (firewalling) et l'équilibrage de charge via un hyperviseur. Cependant, ces fonctions trouvent leurs limites telles que : le poids volumineux et la surcharge des machines virtuelle, les coûts élevés des licences logicielles (l'hyperviseur) et l'évolutivité de plusieurs VNFs qui nécessitent des investissements considérables dans les serveurs et les équipements physiques.

Ericsson et de nombreuses entreprises optent pour l'approche microservices utilisant des conteneurs à la place des machines virtuelles dans le but de résoudre les problèmes liés aux VNFs et à la nouvelle alternative de VNF apparue, appelée CNF (Container network Function), elle apporte de nombreux avantages de légèreté, d'évolutivité, de disponibilité et de portabilité.

Notre travail se focalise sur l'implémentation d'un contrôleur SDN en tant qu'application conteneurisée (microservice) qui assure l'évolutivité et la haute disponibilité à l'intérieur d'un cluster Kubernetes pour faire évoluer l'infrastructure NFV d'Ericsson. Les objectifs techniques de notre travail consistent à :

- Le déploiement d'un contrôleur SDN en tant que un microservice s'exécutant sur un cluster Kubernetes.
- La mise en place d'une infrastructure SDN avec le contrôleur RYU.

- La mise en œuvre du découpage du réseau (Network Slicing) afin de déployer des fonctions réseau conteneurisé de manière indépendante.

Les objectifs managériaux de notre solution sont :

- Assurer une haute disponibilité de la solution en temps réel.
- Assurer l'évolutivité automatique (auto-scaling) de la solution.
- Réduction de la consommation de la RAM et CPU. Les anciennes solutions nécessitent une grande capacité de ressources.
- Déployer rapidement des fonctions réseau, au lieu des heures, il se réalisera en quelques secondes.

Ce mémoire est organisé en cinq chapitres présentés comme suit :

Le premier chapitre est dédié à la présentation des technologies du SDN et NFV, leurs composants et leurs avantages et limites.

Le deuxième chapitre fournit un aperçu de l'approche microservices et de la technologie de conteneurisation. Nous discutons les avantages de l'utilisation des conteneurs par rapport aux machines virtuelles. Puis, nous présentons les outils existants capables de gérer les conteneurs Docker, nous précisons également l'intérêt de Kubernetes par rapport aux autres plateformes d'orchestration.

Le troisième chapitre est dédié à la description et à l'explication des outils utilisés par Docker et Kubernetes.

Le quatrième chapitre est consacré aux différentes étapes de l'implémentation et au test de notre solution. Il se terminera par une évaluation objective.

Le dernier chapitre traitera la partie managériale qui explique l'impact du projet sur le management de l'entreprise.

En dernier lieu, nous avons conclu notre projet de fin d'études par une conclusion générale et perspectives.

Chapitre I:

Technologies SDN et NFV :

Généralités et concepts

Introduction

Au cours des dernières années, des initiatives à travers le monde proposent de nouveaux mécanismes permettant l'évolution des réseaux existant vers plus de flexibilité, une meilleure gestion des ressources réseau et une simplification du déploiement des nouvelles fonctions réseau virtualisés. La fusion de ces mécanismes tels que le réseau défini par logiciel (SDN) et la virtualisation des fonctions réseaux (NFV), modifient la façon dont les opérateurs de réseaux déploient et gèrent les services internet.

Dans ce chapitre, nous présentons une vue générale sur les technologies SDN et NFV. Nous commençons par présenter SDN, son architecture et le protocole utilisé (OpenFlow). Nous aborderons par la suite la technologie NFV, sa définition, son architecture, ainsi que ses avantages et limites.

I.1. Réseau défini par logiciel SDN :

I.1.1. Définition :

Selon l'ONF (Open Networking Foundation) SDN est une architecture qui sépare le plan de contrôle du plan de données, et centralise toute l'intelligence de réseau dans une entité programmable, afin de gérer plusieurs éléments du plan de données (switches ou routeurs, etc.) via des APIs (Application Programming Interface) ouverte et standard pour permettre une gestion multifournisseurs [1].

L'idée principale de ce nouveau paradigme, est de sortir la partie intelligente des équipements d'interconnexions et la placer dans un seul point de contrôle appelé « contrôleur », qui maintient une vue globale du réseau et injecte directement les règles de traitement des données sur chaque équipement, ce qui simplifie la gestion et la configuration de réseau. La communication entre le contrôleur et les équipements réseaux se fait à travers des (API) appelées les interfaces sud ou (Southbound APIs). La figure I.1 suivante illustre le routage d'un paquet entre deux réseaux distants via une implémentation de SDN avec le protocole de communication OpenFlow.

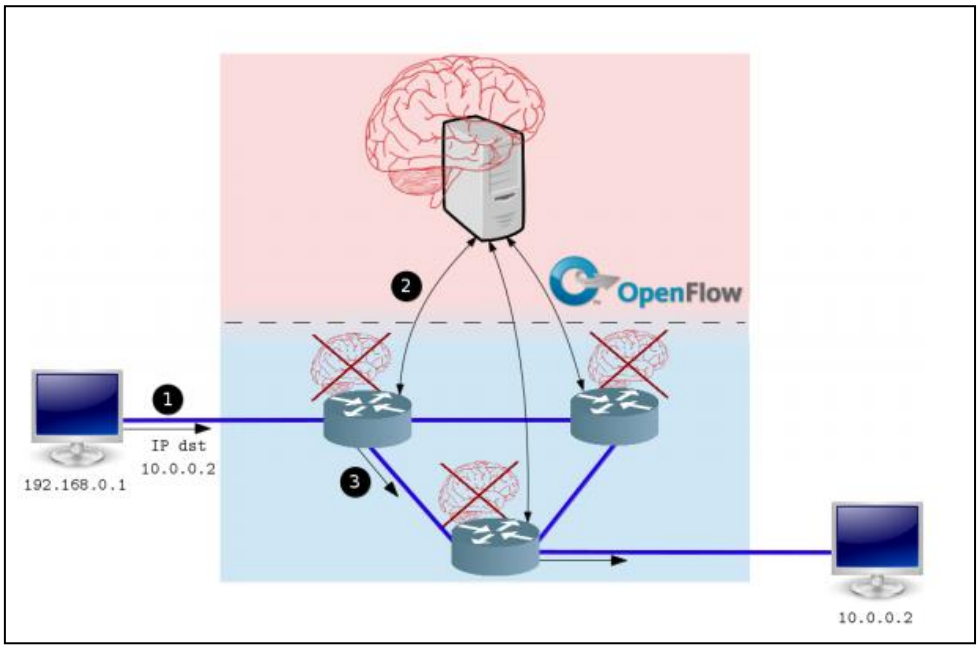

Figure .I.1 : Réseau SDN simple

I.1.2. Comparaison entre un réseau SDN et un réseau traditionnel :

Les réseaux traditionnels sont devenus de plus en plus complexes à administrer et à sécuriser malgré leur large adoption. C'est-à-dire que les administrateurs des réseaux passent beaucoup de temps à configurer manuellement les équipements réseaux à travers des lignes de commande. En plus, ces équipements réseaux tournent le plus souvent avec des logiciels propriétaires. Par exemple, dans un équipement réseau traditionnel, la partie qui contrôle le routage des paquets et la partie qui gère l'acheminement des paquets sont ensemble sur un seul dispositif, ce qui rend difficile son évolution. Autrement, ce couplage limite l'innovation puisque toute introduction d'un nouveau protocole ou service sur l'équipement réseau, passe obligatoirement par le fabricant et peut-être parfois très long. D'où le SDN a été introduite pour résoudre les problèmes et les limites des réseaux traditionnels.

Dans un réseau SDN, la séparation entre le plan de données et le plan de contrôle rend le réseau programmable. Le plan de contrôle est placé dans un contrôleur centralisé qui a une vision sur la topologie du réseau et le plan de données réside encore dans le commutateur ou le routeur. La figure I.2 illustre la différence architecturale entre un réseau traditionnel et un réseau SDN.

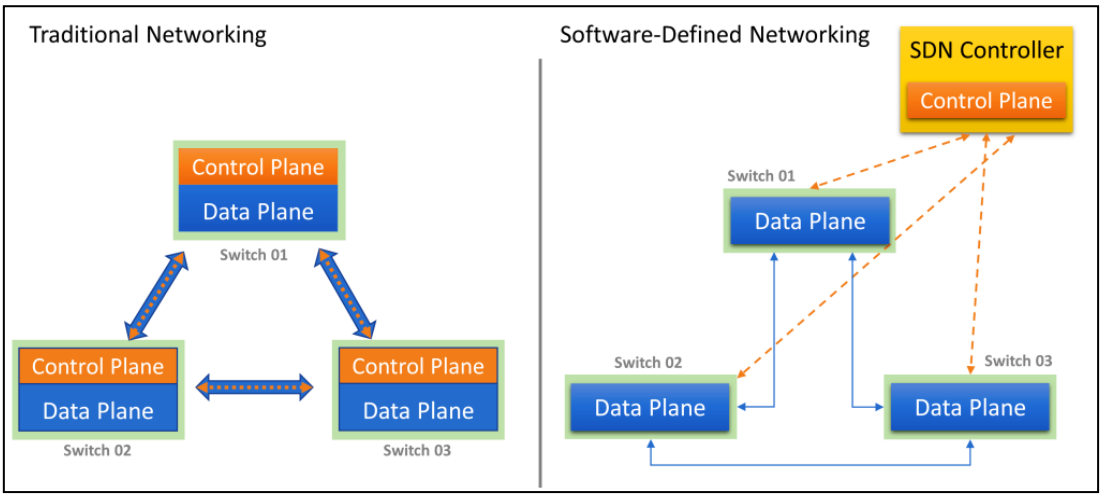

Figure .I.2: Comparaison entre un réseau traditionnel et un réseau SDN

I.1.3. Architecture du SDN [1]:

Selon la figure I.3 l'architecture d'un réseau SDN est divisée en trois couches principales, nous décrivons dans ce qui suit ces couches, ainsi que les interfaces de communications :

1. **Couche de données ou la couche infrastructure:** appelée aussi «plan de données», elle est composée des équipements d'acheminement tels que des commutateurs, des routeurs, des équipements virtuels, etc. L'objectif d'un plan de données est de transmettre le trafic réseau sur une base d'un certain ensemble de règles de transmission ordonnée par le plan de contrôle.
2. **Couche de contrôle :** appelée aussi «plan de contrôle», elle est constituée principalement d'un ou plusieurs contrôleurs SDN, son rôle est de contrôler et de gérer les équipements de l'infrastructure à travers une interface appelée « South-bound API ».
3. **Couche application :** représente les applications qui permettent de déployer de nouvelles fonctionnalités réseau, comme la surveillance de réseau, la qualité de service (QoS), la sécurité de réseau, le contrôle d'accès, la virtualisation de fonction de réseau (VNF), etc. Ces applications sont construites d'une interface de programmation appelée « Northbound API ».

Comme mentionné ci-dessus, le contrôleur fournit principalement deux interfaces de programmation, à savoir l'interface 'south-bound' et l'interface 'north-bound'.

- **Interface South-Bound :** appelé aussi « southbound APIs» représente les interfaces de communication, qui permettent au contrôleur SDN d'interagir avec les équipements de la couche d'infrastructure (plan de données), tel que les switchs, et les routeurs. Le protocole le plus utilisé, et le plus déployé comme interface Sud est le protocole OpenFlow, qui a été standardisé par l'ONF.

- **Interface North-Bound :** est surtout un écosystème logiciel, pas un matériel comme c'est le cas des APIs southbound. Il décrit la zone de communication couverte par le protocole OpenFlow entre le contrôleur et les applications ou les programmes de commande de la couche supérieure (application).

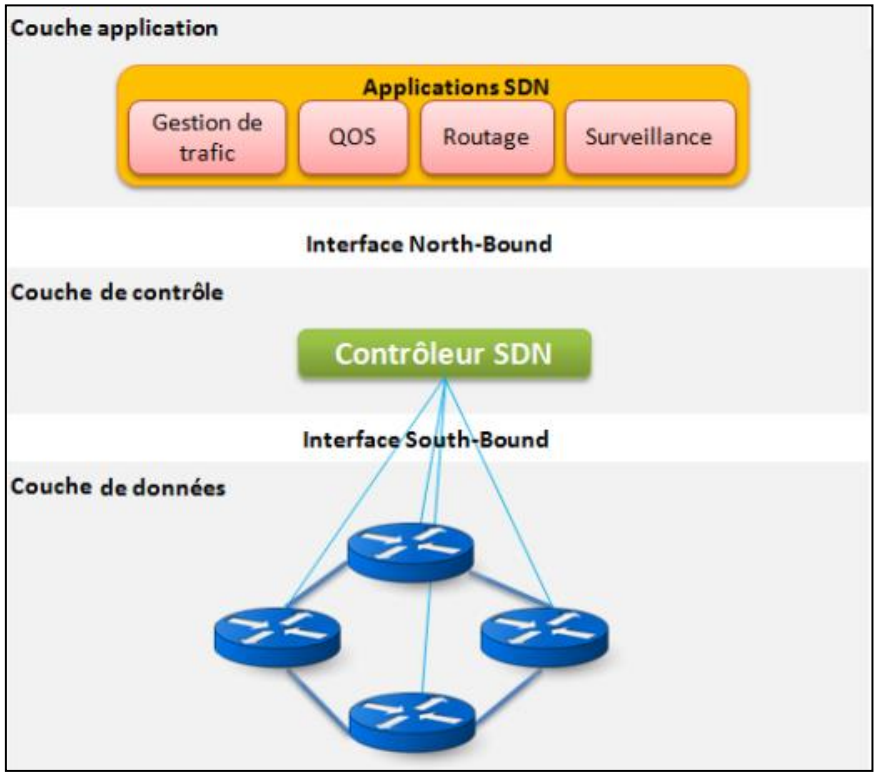

Figure .I.3: Architecture du SDN

I.1.4. Protocole de communication OpenFlow :

Il existe plusieurs protocoles de communication entre le plan de contrôle et le plan de données tels que ForCES (Forwarding and Control Element Separation), PCE (Path Computation Element) et NetConf (Network Configuration). Cependant, le protocole de communication OpenFlow reste le plus utilisé sur les équipements du réseau SDN [2].

Open Flow est un protocole de communication standard utilisé pour la communication entre le contrôleur et le commutateur dans un réseau SDN ainsi que d'accéder au plan de données des périphériques réseaux. OpenFlow a été standardisé par l'ONF, une organisation qui compte de nombreux fournisseurs de réseau comme membre. Ce protocole avait d'abord été développé pour tester les nouveaux protocoles dans les réseaux de campus. Mais plus tard, il a évolué pour être utilisé dans les réseaux SDN.

OpenFlow est donc un protocole ouvert qui permet aux administrateurs de réseau de programmer les tables de flux (flow tables) dans leurs différents commutateurs.

I.1.4.1. Architecture OpenFlow

Dans une architecture SDN, le protocole Open Flow est basé sur deux éléments principaux, les contrôleurs Open Flow et les commutateurs Open Flow.

1) Commutateur OpenFlow [2]

Un commutateur OpenFlow est un équipement réseau prenant en charge le protocole OpenFlow, tel qu'un commutateur, un routeur ou un appareil réseau intermédiaire. Chaque périphérique OpenFlow maintient une ou plusieurs tables de flux (tables d'acheminement) qui contiennent donc des entrées. Chaque entrée composée de trois champs comme montre la figure I.4.

Champs d'en-tête	Actions	Compteurs

Figure .I.4 : Entrée de la table de flux

a) Champs d'en-tête: permettent d'identifier le flux à base d'une multitude d'informations contenues dans l'entête du paquet, allant de la couche 1 à la couche 4 du modèle ISO tels qu'Ethernet, IPv4, IPv6 ou MPLS. Ce champ permet de vérifier si un paquet correspond avec l'entrée d'un flux.

b) Actions : spécifient comment les paquets d'un flux seront traités. Une action peut être l'une des suivantes : transférer le paquet vers un ou plusieurs ports, supprimer le paquet, transférer le paquet vers le contrôleur, ou modifier le champ d'en-tête de paquet.

c) Compteurs : pour la collecte des statistiques de flux. Ils enregistrent le nombre de paquets et d'octets reçus de chaque flux, et le temps écoulé depuis le dernier transfert de flux.

2) Contrôleur OpenFlow

Le contrôleur est l'élément de base de la couche contrôle dans une architecture SDN. Un contrôleur SDN a pour rôle d'ajouter, de supprimer et de modifier d'une façon dynamique les entrées de flux dans la table des flux de chaque commutateur OpenFlow (virtuel ou pas) pour des raisons d'acheminement des flux (routage). Nous allons évoquer dans ce qui suit les contrôleurs (open source) les plus utilisés:

- **Ryu :** est un contrôleur SDN à base de composants écrits dans le langage de programmation Python. L'un des principaux avantages de ce contrôleur, contrairement à de nombreux autres contrôleurs SDN, est qu'il supporte toutes les versions du protocole OpenFlow de 1.0 à 1.6. De plus, en raison de sa conception basée sur les composants et

de la prise en charge complète des versions OpenFlow, il est souvent utilisé pour le développement rapide d'un module SDN **[3]**.

- **OpenDaylight :** le contrôleur OpenDaylight est une plateforme à base de la machine virtuelle Java et peu être exécuté depuis n'importe quel système d'exploitation. Il est utilisé pour automatiser des réseaux plus étendus **[3].**
- **ONOS :** est un système distribué spécifiquement pour l'évolutivité et la haute disponibilité. Avec cette conception, ONOS se projette comme un système d'exploitation de réseau, avec une séparation des plans de contrôle et de données pour les réseaux étendus (WAN) et les réseaux de fournisseurs de services **[3].**

La mise en réseau SDN utilise deux approches (ou architectures) pour le plan de contrôlée (Centralisée ou Distribuée), le tableau suivant présente une comparaison entre ces deux architectures.

Approche	Centralisée	Distribuée
Définition	Le mode de contrôle dans ce type d'architectures est complètement centralisé autour d'un seul contrôleur SDN pour contrôler tout le réseau.	Le mode de contrôle dans ce type d'architectures est distribué et basé sur une hiérarchie de multi-contrôleurs SDN, tel que chaque contrôleur SDN gère une partie du réseau.
Avantages	- Configurer l'ensemble du réseau à partir d'un seul appareil - Le contrôleur central a un accès complet et un aperçu de tout ce qui se passe dans le réseau.	- l'évolutivité et la fiabilité du réseau, même en cas de très fortes charges. - Chaque contrôleur a pris la responsabilité d'une certaine partie du réseau donc, moins de charge sur les contrôleurs.
Inconvénients	- Un contrôleur central peut agir comme un point de défaillance unique. - L'augmentation de trafic sur les grands réseaux peut surcharger le contrôleur.	- Cette approche nécessite une importante synchronisation des données entre les différents contrôleurs afin de conserver une vue globale du réseau. - Cette approche est très coûteuse en ressource

Tableau .I.1 : Comparaison entre l'approche centralisée et l'approche distribuée.

Le contrôleur Ryu support l'approche centralisé, en revanche les contrôleurs Opendaylighit et ONOS supportant l'approche distribuée. Nous avons choisi le contrôleur Ryu dans notre solution pour profiter les avantages de l'architecture centralisé tout en assurant les bénéfices de l'architecture distribué en terme de l'évolutivité et la fiabilité (disponibilité) par l'implémentation de contrôleur Ryu dans un cluster Kubernetes.

I.1.5. Avantages et limite du SDN :

La technologie SDN offre un réseau qui peut dynamiquement s'adapter afin de répondre au mieux aux besoins des utilisateurs. Le SDN fournit plusieurs améliorations par rapport au fonctionnement classique. Il offre les avantages suivants :

- Les réseaux programmables permettent de répondre aux besoins d'automatisation du réseau. Les administrateurs de réseau peuvent utiliser des APIs afin de programmer les équipements réseau.
- La gestion et le contrôle du tout le réseau peuvent se faire via une seule interface au lieu de configurer chaque périphérique individuellement. La conception de gestion centralisée simplifie l'administration des infrastructures de grande taille.
- L'homogénéité grâce à la standardisation du protocole de communication OpenFlow, permet une interopérabilité entre les différents matériels réseaux utilisés. Par conséquent, les administrateurs du réseau peuvent aisément gérer les flux directement depuis l'interface du contrôleur quelle que soit la marque des composants matériels.
- Le contrôle centralisé fourni par SDN permet une gestion et une orchestration efficaces des fonctions réseau virtuelles. SDN est combiné parfaitement avec une autre technologie, appelée Network Functions Virtualization (NFV).

Malgré ses nombreux avantages, le SDN présente tout de même quelques limites liées notamment à son architecture :

- Le passage à l'échelle ou la scalabilité du réseau, plus la taille de réseau augmente, plus des demandes sont envoyées au contrôleur et à un moment donné, le contrôleur devient incapable de traiter toutes ces demandes.
- Le fait de centraliser toute l'intelligence du réseau dans un seul contrôleur présente un risque de fiabilité et de sécurité en cas défaillance du contrôleur, s'il est compromis ou devient inaccessible peut provoquer la panne de l'ensemble du réseau.
- La distribution des contrôleurs SDN (multi-contrôleurs) est très coûteuse en ressources

I.2. Virtualisation [4]:

La virtualisation est un mécanisme informatique qui nous permet de faire fonctionner sur une même machine physique plusieurs systèmes, serveurs ou applications comme s'ils fonctionnaient sur des machines physiques distinctes. Elle nous permet d'exploiter toute la capacité d'une machine physique en la répartissant en plusieurs environnements sécurisés distincts. C'est ce que l'on appelle les machines virtuelles.

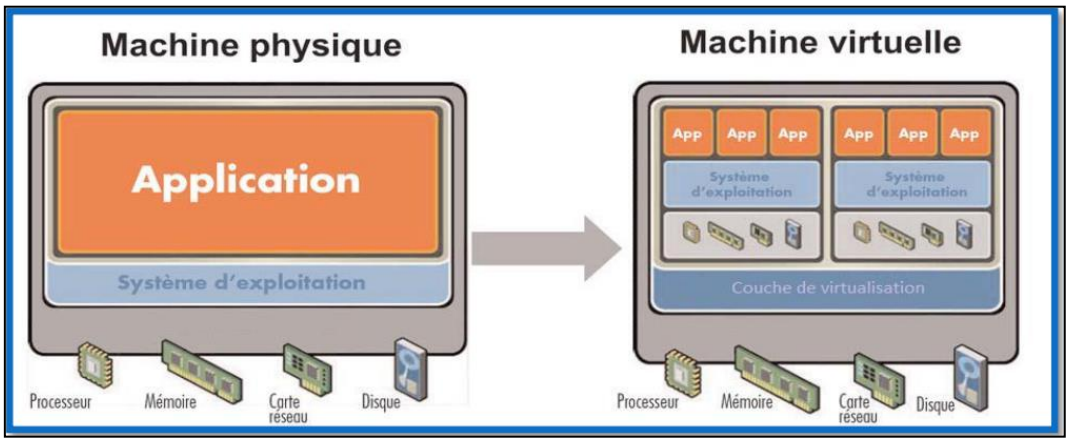

Figure .I.5 : Concept de la virtualisation

Ces dernières exploitent la capacité de l'hyperviseur à séparer les ressources du matériel et à les distribuer convenablement. Comme on le voit à la figure I.5.

I.3. Virtualisation des fonctions réseau NFV :

I.3.1. Définition [4]:

Selon l'Institut européen des normes de télécommunications (ETSI: European Telecommunications Standards Institute), l'idée de la virtualisation des fonctions réseau est reconnue comme une architecture réseau qui transforme la façon de construire et d'exploiter des réseaux en exploitant les technologies de virtualisation informatique standard et en consolidant les fonctions réseaux basées sur le matériel informatique propriétaire aux appareils standards.

La virtualisation des fonctions réseau (NFV) est une technologie réseau émergente. Au lieu de déployer des équipements matériels pour chaque fonction du réseau, les fonctions réseau virtualisées dans NFV sont réalisées via des machines virtuelles (VM) exécutant divers logiciels au-dessus des serveurs à haut volume ou de l'infrastructure cloud.

I.3.2. Architecture NFV [4]:

Selon l'ETSI, l'architecture NFV est composée de trois éléments clés : Les fonctions de réseau virtuelles (VNFs), L'infrastructure (NFVI) dans laquelle les fonctions seront exécutées et NFV MANO pour la gestion et l'orchestration, comme l'illustrer la figure suivante :

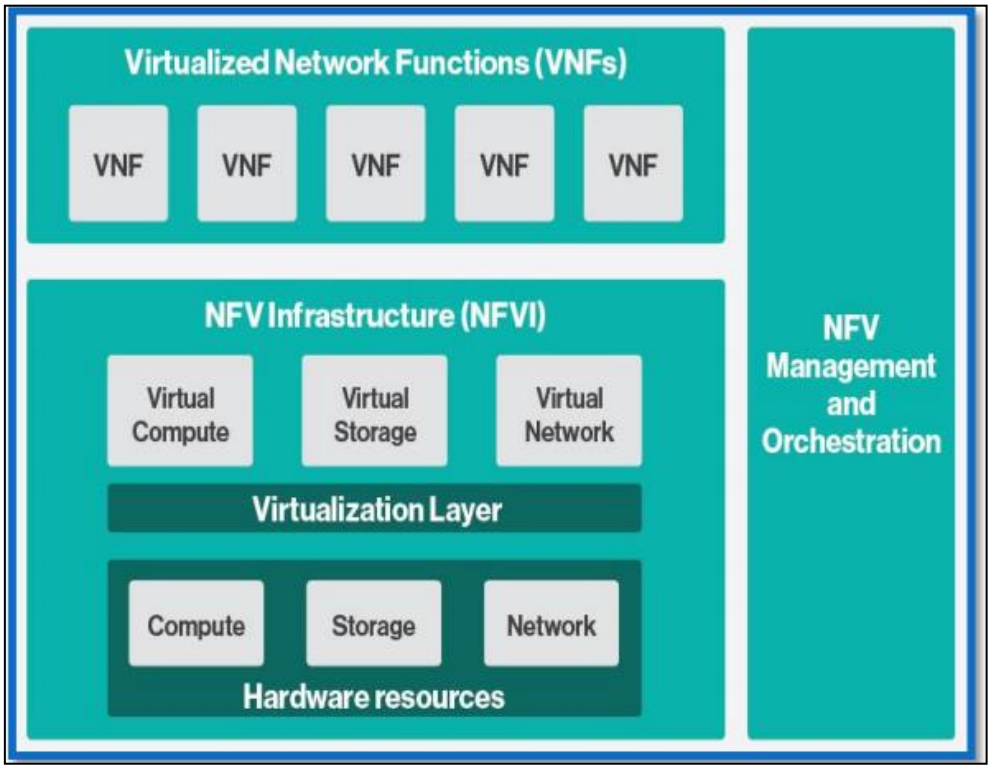

Figure .I.6 : Architecture NFV

I.3.2.1 VNF (Virtualized Network Functions) :

Les VNFs correspondent aux fonctions de réseaux virtualisées mise en œuvre sur un serveur physique et déployées sous forme de machines virtuelles. Par exemple, un VNF routeur, un VNF commutateur OpenFlow, un VNF Firewall, etc.

I.3.2.2 NFVI (Network Functions Virtualization Infrastructure):

Les fonctions de réseau virtuel sont exécutées dans un environnement appelé infrastructure de virtualisation de fonction de réseau. Ceci comprend :

- **Ressources physiques** : Cette partie de l'infrastructure est utilisée pour les ressources informatiques, de stockage et de mise en réseau.
- **Ressources virtuelles** : Les ressources physiques sont transformées en ressources virtuelles abstraites à utiliser par les fonctions de réseau virtuel.
- **Virtualisation Layer** : Un hyperviseur dans lequel les ressources physiques sont abstraites en ressources virtuelles.

I.3.2.3 NFV M&O (Management and Orchestration) :

NFV M&O (Management and Orchestration) : permettant de gérer les services réseaux de bout en bout, il contient 3 blocs :

- **NFV Orchestrator** : se charge notamment de la gestion du cycle des services réseau, de leur production à leur exploitation, et de la supervision des processus d'allocation de ressources à ces services.
- **VNF Manager** : Il permet la gestion du cycle de vie des instances VNF, il est responsable de : l'initialisation, la mise à jour, les tests, la mise à l'échelle et la résiliation (terminate). Plusieurs VNF Manager peuvent être déployé pour gérer une ou un ensemble de VNFs.
- **VIM (Virtualized Infrastructure Manager)** : Contrôle et gère les ressources de calcul, de stockage et de réseau du NFVI, c'est typiquement la que l'on trouve des éléments comme OpenStack.

Ericsson est l'un des principaux fournisseurs de technologies de l'information et de la communication (TIC) qui propose, dans ce domaine aux opérateurs tels qu'Ooredoo, OTA Djezzy et AT Mobilis la solution Ericsson NFVI.

I.3.3. Solution Ericsson NFVI [5]

Ericsson NFVI se compose de produits logiciels et matériels ainsi que de services de support et d'intégration de systèmes formant une solution complète pour les opérateurs télécoms. La solution Ericsson NFV permet aux opérateurs de déployer rapidement des charges de travail virtuelles télécoms, OSS, BSS, IT et médias tout en maintenant un faible coût total de possession. En fournissant une solution pré-intégrée, testée et vérifiée, les opérateurs disposeront de bases solides pour saisir de nouvelles opportunités de revenus dans les domaines de la consommation et de l'entreprise.

Afin de permettre aux opérateurs de la téléphonie mobiles de virtualiser leurs architectures réseaux convenablement, la solution NFVI d'Ericsson propose des différents VNF appropriés aux applications de service .La Figure I.7.présente les différentes solutions Ericsson qui constituent la solution NFVI convenable à l'architecture du NFV.

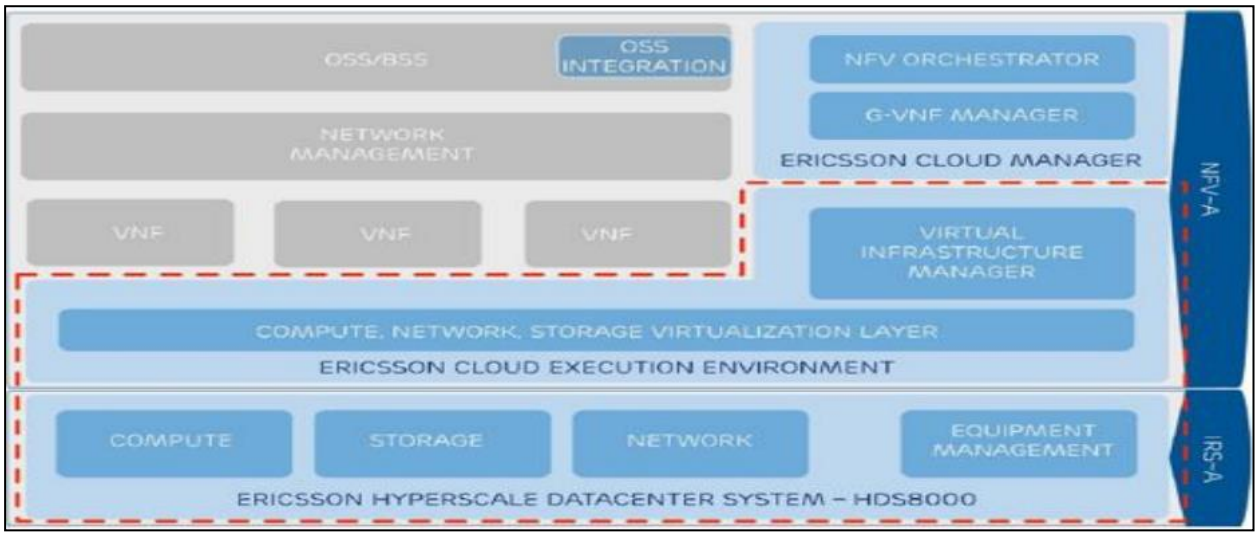

Figure .I.7 : Architecture NFVI Ericsson.

NFVI« Ericsson Network FunctionsVirtualization Infrastructure », une solution complète pour la création d'un cloud capable de prendre en charge les charges de travail Telco et IT. La solution NFVI se constitue de trois couches:

1. Couche d'infrastructure sous la forme de l'Ericsson Hyperscale Datacenter System (HDS).
2. Couche VIM « Virtualized Infrastructure Manager » sous la forme d'ECEE « Ericsson Cloud Execution Environment ».
3. Couche réseau : comprenant à la fois contrôle plane et data, elle transportera le trafic NFVI entre les locataires et les réseaux backbone de opérateurs mobiles.

Ericsson utilise le Network Slicing ou le découpage de réseau pour les réseaux 5G qui permet de maximiser la rentabilité des investissements via une utilisation et une gestion efficaces des ressources du réseau et de fournir des services différenciés à grande échelle.

I.3.4. Network-slicing [6]

SDN et NFV permettent une meilleure flexibilité du réseau grâce au partitionnement des architectures réseau en éléments virtuels. En substance, le « Network Slicing » permet la création de plusieurs réseaux virtuels au sommet d'une infrastructure physique partagée. Ces différents réseaux logiques sont appelés des tranches de réseaux ou network slices, et correspondent à la mise en réseau de bout en bout entre un client et un service réseau en s'appuyant sur une infrastructure réseau virtuelle. En langage NFV, une tranche serait généralement déployée en tant qu'instance de service réseau. Chaque tranche de réseau est isolée des autres et gérée/contrôlée de manière indépendante et s'adapte à la demande.

I.3.4. Avantages de NFV :

La solution NFV apporte de nombreux avantages pour les opérateurs de réseau tels que :

- La technologie NFV utilise du matériel standard, les opérateurs de réseau ont la liberté de choisir et de construire le matériel de la manière la plus efficace pour répondre à leurs besoins et exigences.
- Les ressources de calcul, de mémoire et de stockage des serveurs peuvent être partagées simultanément par plusieurs machines virtuelles de manière flexible tout en optimisant les coûts.
- La technologie NFV est agile et peut évoluer en fonction de la demande de services, tout en hébergeant plusieurs services sur un même serveur physique, économiquement plus rentable.
- Réduire le temps nécessaire au déploiement de nouveaux services.

I.3.5. Limites de NFV :

Malgré de nombreux avantages des réseaux NFV, nous retrouvons certains inconvénients tels que :

- L’impact sur les performances en raison de la surcharge des machines virtuelles ainsi que le temps de démarrage long.
- Le poids lourd des machines virtuelles et le manque d'optimisation des ressources entravent l'efficacité des VNF.
- L'évolutivité multi-VNF nécessite des investissements considérables dans les serveurs et l'équipement PNF.

Conclusion

Nous avons vu dans ce chapitre les deux technologies SDN et NFV qui sont interdépendants, mais lorsqu'ils sont déployés ensemble, ils permettent d'obtenir des infrastructures réseaux flexibles et agiles. SDN assume la responsabilité de la gestion et l’orchestration globale des opérations réseau et NFV fournit le déploiement et l'exécution des fonctions réseau désormais virtualisées.

L’architecture NFV basée principalement sur la virtualisation au niveau matérielle, elle intègre des VNF qui sont des machines virtuelles, mais malheureusement cette dernière possède plusieurs inconvénients qui entravent leur efficacité surtout dans les grands réseaux pour cela un nouveau type de virtualisation au niveau du système d’exploitation a été créé, c’est ce qu’on appelle la conteneurisation.

Chapitre II:

Microservices et conteneurs

Introduction

Dans le monde d'aujourd'hui, les applications doivent être déployées fréquemment et rapidement pour être compétitives. L'architecture des microservices a émergé pour aider les entreprises à décomposer leurs applications monolithiques en plus de petits éléments. Ces microservices sont généralement déployés à l'aide de conteneurs, et avec de plus en plus de conteneurs déployés, il est nécessaire de disposer un outil d'orchestration de ces conteneurs. Le rôle de l'orchestrateur est d'automatiser de nombreux processus manuels, tels que le déploiement, la gestion et la mise à l'échelle des conteneurs.

Dans ce chapitre, nous décrivons les deux architectures monolithiques et microservices en précisant les avantages de l'architecture microservices, et présenter également une vue globale sur les technologies de conteneurisation et l'orchestration de conteneur.

II.1 Microservices [7]

II.1.1. Passage de l'architecture monolithique vers microservices :

- **Concept de monolithique :**

L'approche monolithique utilise une seule base de code pour créer des applications qui simplifient les processus de déploiement, de débogage et de test. Cela fonctionne bien si les applications sont à leurs étapes initiales et que la structure est plus simplifiée. À mesure que la taille de l'application augmente les interdépendances entre les composants augmentent, ce qui entraîne une application complexe.

L'architecture monolithique comporte plusieurs inconvénients, parmi lesquels :

- **Manque de fiabilité :** une seule erreur dans n'importe quel module peut faire tomber toute l'application monolithique.
- **Évolutivité :** il n'est pas possible de faire évoluer les composants de manière indépendante, mais uniquement l'application dans son ensemble.
- **Limites technologiques :** il est très difficile d'adopter une nouvelle technologie bien adaptée à une fonctionnalité particulière, car elle affecte l'ensemble de l'application.
- **Modification coûteuse :** une modification dans une petite partie de l'application requiert un redéploiement entier.

Pour remédier à ces limites, il existe une nouvelle approche de développement des applications qu'est l'architecture microservices.

II.1.2. Concept de microservices:

Les microservices est une approche moderne du développement logiciel. Martin Fowler décrit les microservices comme une approche pour développer une application unique en tant que suites de petits services indépendants qui communiquent sur les API bien définis. Chaque microservice possède son propre logique métier et sa propre base de données. En plus, le code est beaucoup moins volumineux et donc plus simple à tester, à maintenir et à faire évoluer. La figure II.1 montre une comparaison entre l'approche monolithique et l'approche microservices.

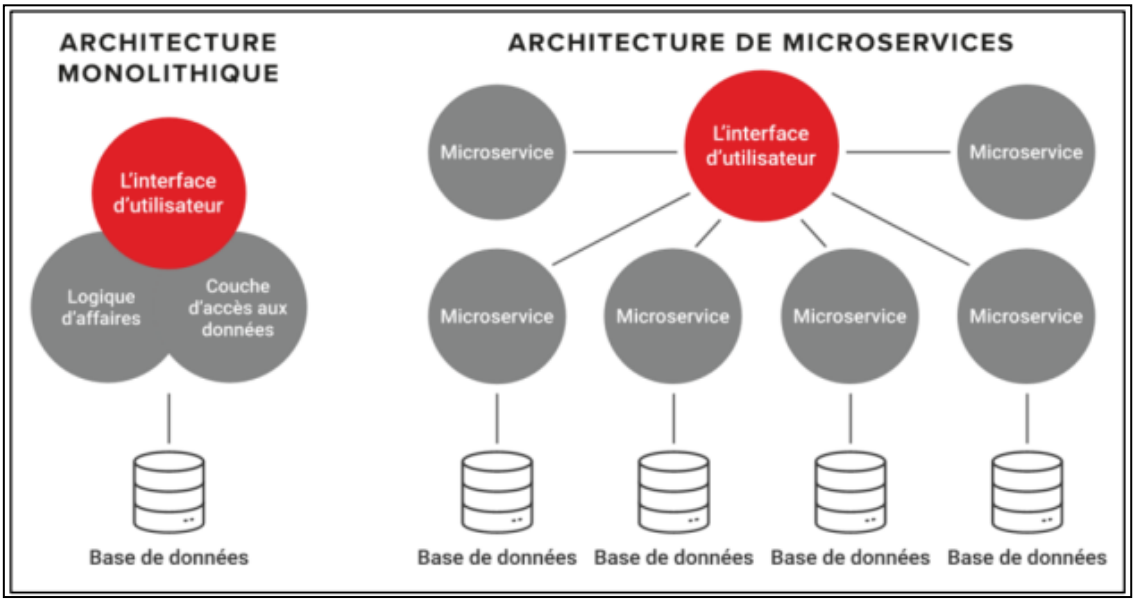

Figure .II.1 : Comparaison entre l'approche monolithique et l'approche microservices

II.1.3. Avantages d'une architecture microservices :

L'architecture microservices comporte plusieurs avantages, parmi lesquels :

- **Les microservices sont indépendamment évolutifs :**

À mesure que la demande pour une application augmente, il est plus facile de faire évoluer l'utilisation des microservices. Nous pouvons augmenter les ressources des microservices les plus nécessaires plutôt que de mettre à l'échelle une application entière. Cela signifie également que la mise à l'échelle est plus rapide et souvent plus rentable.

- **Les microservices réduisent les temps d'arrêt grâce à l'isolation des pannes :**

Si un microservice spécifique échoue, vous pouvez isoler cette défaillance de ce service unique et empêcher les défaillances en cascade qui entraîneraient le blocage de l'application. Cette isolation des pannes signifie que votre application critique peut rester opérationnelle même en cas de défaillance de l'un de ses modules.

- **Facilité de déploiement :**

Les microservices individuels et isolés sont beaucoup plus faciles et rapides à déployer dans les pipelines d'intégration et du déploiement continu. Seul le service concerné est modifié et redéployé lorsqu'une mise à jour est nécessaire.

- **Flexibilité dans le choix de la technologie :**

Les équipes d'ingénieurs ne sont pas limitées par la technologie choisie dès le départ. Cette flexibilité permet d'appliquer librement diverses technologies pour chaque microservice en fonction des besoins de l'entreprise et aux compétences de leurs équipes.

II.2 Conteneurisation

II.2.1. Définition :

Il s'agit d'une forme de virtualisation du système d'exploitation dans laquelle s'exécutent des applications dans des espaces utilisateurs isolés appelés conteneurs qui utilisent le même système d'exploitation partagé. Les conteneurs sont des unités exécutables de logiciel dans lesquelles le code d'application est empaqueté, avec ses bibliothèques et ses dépendances, de manière commune, afin qu'il puisse être exécuté n'importe où et déplacée facilement.

Les environnements d'exécution de conteneurs sont des composants logiciels qui exécutent des conteneurs sur le système d'exploitation hôte et gèrent leur cycle de vie. Ils travaillent avec le noyau du système d'exploitation pour lancer et soutenir la conteneurisation, et peuvent être contrôlés et automatisés par des systèmes d'orchestration de conteneurs tels que Kubernetes.

Les conteneurs sont populaires pour les microservices, car ils sont légers, portables, et démarrent plus vite que les machines virtuelles. La figure II.2 illustre le concept de la conteneurisation.

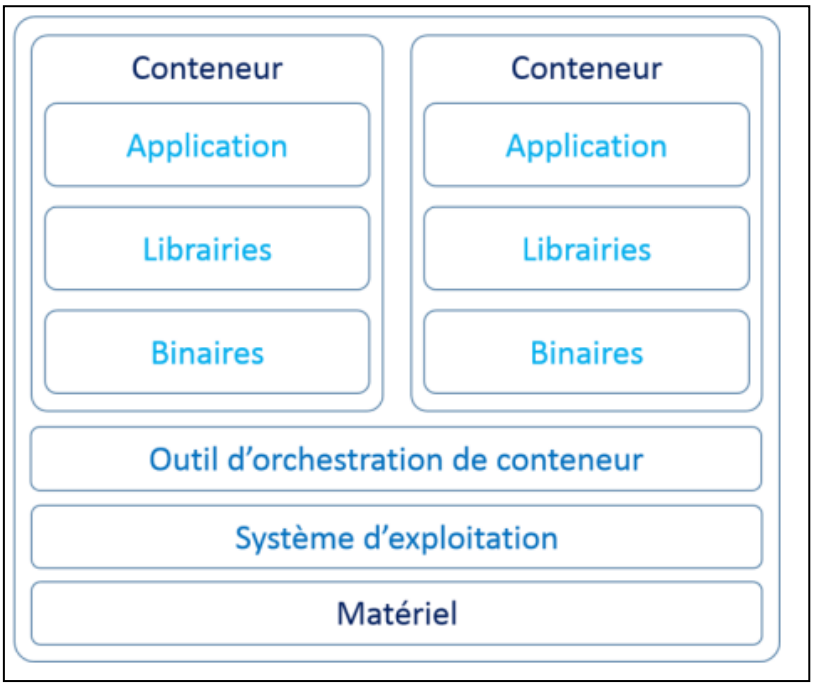

Figure .II.2 : Principe de la conteneurisation

II.2.2. Différences entre les VMs et les conteneurs

Pour mieux comprendre un conteneur, il convient de savoir ce qui les distingue des machines virtuelles (VM) traditionnelles. Voici le tableau qui conclut sur les différences entre une machine virtuelle et un conteneur.

VM	CONTENEUR
Isolation des processus au niveau matériel.	Isolation des processus au niveau du système d'exploitation.
Chaque machine virtuelle a un système d'exploitation distinct.	Chaque conteneur peut partager le système d'exploitation.
Reprise en quelques minutes.	Reprise en quelques seconds.
Les machines virtuelles sont de quelques Go.	Les conteneurs sont légers (Ko / Mo).
Le portage d'une machine virtuelle est difficile et prend également beaucoup de temps en raison de sa taille.	Un conteneur peut être porté facilement sur un autre système d'exploitation et il peut démarrer immédiatement.
La création de VM prend un temps relativement plus long.	Les conteneurs peuvent être créés en quelques seconds.
Plus d'utilisation des ressources.	Moins d'utilisation des ressources.
Très peu de logiciel gratuite pour faire de la virtualisation, on peut citer virtualbox sinon,	La plupart des logiciels de conteneurisation tels que Docker, LXC, Kubernetes sont basés

c'est des licences payantes et on peut citer VMWare qui est le leader de la Virtualisation mais qui coûte très cher.	sur linux et donc OpenSource . On ne paie que les machines physiques qui ont besoin de moins de performance que pour des VM.

Tableau .II.1 : Différences entre les VMs et les conteneurs

Ericsson a évolué leur infrastructure NFV (NFVI) pour adopter les technologies cloud natives basées sur les conteneurs et microservices donc une nouvelle alternative de VNF est apparue qui s'appelle CNF (Container network function). L'adoption des CNFs peut résoudre les principaux problèmes liés aux VNF grâce à la migration de nombreuses fonctions réseau virtualisées vers des conteneurs.

II.2.3. Avantages de CNF (Container network function) [8]

La migration des VNFs vers des CNFs basés sur des conteneurs apporte de nombreux avantages :

1. **Légèreté :** ils partagent le noyau du système d'exploitation de la machine hôte, ce qui évite d'utiliser une instance complète du système d'exploitation pour chaque fonction réseau, ainsi que les conteneurs beaucoup moins volumineux que les VMs, ils permettent d'économiser les ressources
2. **Réduction des coûts :** l'infrastructure de mise en place des CNFs n'a plus besoin de fonctionner sur du matériel spécialisé. Elle peut fonctionner sur des serveurs de base connectés dans un cluster privé, ou même dans des infrastructures de cloud public comme OpenStack ou AWS.
3. **Évolutivité :** offrent une grande évolutivité. Les CNFs peuvent automatiquement gérer des charges de travail croissantes en reconfigurant l'architecture existante afin d'activer les ressources.
4. **Tolérance aux pannes :** chaque CNF s'exécute indépendamment de sorte que la défaillance de l'un n'aura pas d'impact sur la continuité des autres.

II.3. Orchestration des conteneurs

Lorsque Ericsson a commencé d'adopter les conteneurs, souvent dans le cadre d'architectures microservices modernes et cloud natives, la simplicité du conteneur a commencé à se heurter à la complexité de la gestion de centaines et milliers de conteneurs dans un système réparti.

L'orchestration des conteneurs est un processus qui automatise le déploiement, la gestion, la mise à large échelle et la mise en réseau des conteneurs via un outil d'orchestration de

conteneur. Kubernetes, Docker Swarm et Apache Mesos sont quelques-uns des outils d'orchestration de conteneurs open source populaires.

II.3.1. Orchestrateurs les plus populaires

De nombreux outils d'orchestration sont disponibles pour les conteneurs, certaines des plus courantes sont décrites ici.

1. **Docker Swarm [9]:**

Docker Swarm est le moteur d'orchestration de conteneurs natif de Docker. Initialement publié en novembre 2015, il est également écrit en Go. Swarmkit est la version native Docker de Swarm incluse à partir de la version 1.12, qui est la version recommandée de Docker pour ceux qui souhaitent utiliser Swarm.

2. **Apache Mesos [9]:**

Développé par Apache et lancé en juillet 2016, Mesos est en production avec certaines grandes entreprises telles que Twitter, Airbnb et Netflix. Une application s'exécutant sur Mesos comprend un ou plusieurs conteneurs et est appelée framework. Mesos offre des ressources à chaque framework, et chaque framework doit ensuite décider lequel accepter. Mesos est moins riche en fonctionnalités que Kubernetes et peut impliquer un travail d'intégration supplémentaire.

3. **Kubernetes [9]:**

Kubernetes, communément appelé "k8s", est un large écosystème en rapide expansion. Initialement développé par Google, il permet de gérer les applications conteneurisées dans un environnement en cluster. Ce système d'orchestration de conteneurs permet l'automatisation du déploiement, la mise à l'échelle de nombreux conteneurs d'application sur des clusters. Le projet Kubernetes est devenu open source en 2014.

Le tableau II.2 présente une comparaison entre quelques orchestrateurs de conteneurs les plus courantes dans le marché.

	Docker swarm	**Kubernetes**	**Mesos**
License	Open Source	Open Source/Commercial	Open Source/Commercial
Developpé par	Docker,Inc	Google	Apache Software Foundation
Frais	Paiement par nombres de nœuds.	Gratuit	Paiement par nombres de nœuds.
Interface web native (GUI)	Non	Oui	Oui
Configuration des conteneurs	la configuration des conteneurs est fournie et limitée par l'API Docker**.**	Kubernetes utilise sa propre API client et ses définitions au format YAML	la configuration des conteneurs est basée sur des définitions au format JSON**.**
Fournisseur de cloud public	Azure	Google, Azure, AWS, OTC (Open Telekom Cloud), OpenStack .	Azure, AWS.
Taux d'utilisation dans le marché	21 %	83 %	9 %
Synopsis	Swarm fournit des capacités de planification basiques et ne répond pas aux besoins d'évolution de l'entreprise	Kubernetes est en train d'évoluer rapidement, il est prometteur en matière de développement d'applications modernes à l'échelle de l'entreprise.	la force de Mesos est dans le Big Data et l'analyse de données , il est moins centré sur l'orchestration des conteneurs et le développement d'applications modernes.

Tableau .II.2 : Comparaison entre quelques orchestrateurs de conteneurs.

Malgré les différentes fonctionnalités qu'offrent Docker Swarm et Apache Mesos, ils restent moins évolués par rapport à Kubernetes. Ce dernier est l'outil d'orchestration de conteneurs le plus largement utilisé dans le marché et celle qui a été choisie par Ericsson. C'est pour cela, nous avons choisi l'outil d'orchestration « Kubernetes » dans notre travail. Cette technologie sera abordée d'une manière plus détaillée dans le chapitre suivant.

La figure II.3 montre que kubernetes est la solution la plus populaire selon Google Trend.

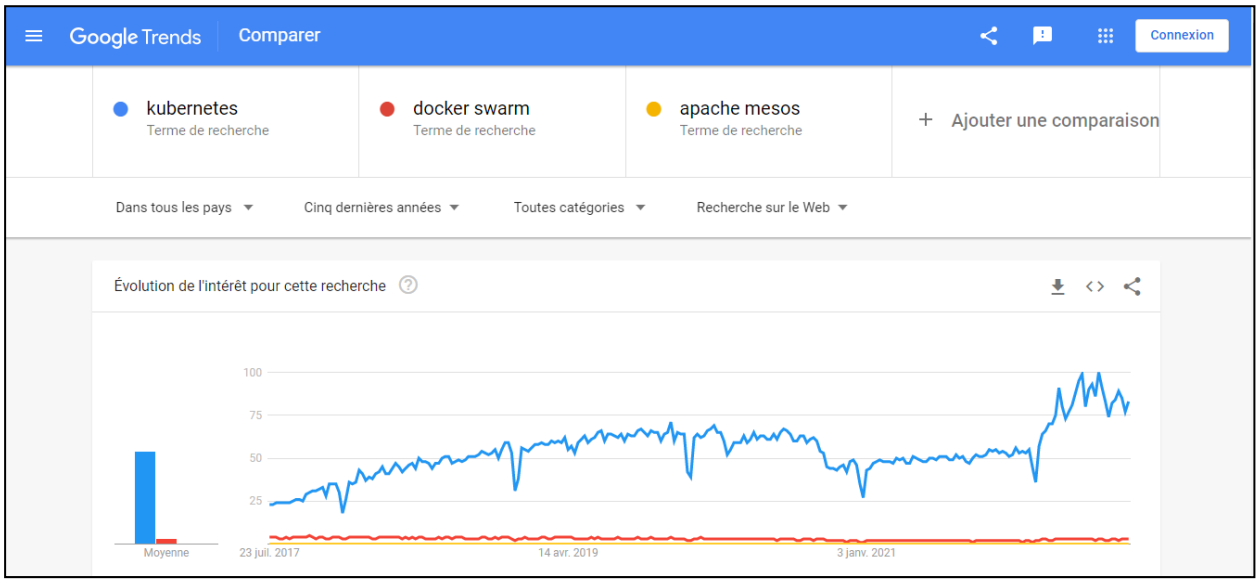

Figure .II.3 : Graphique de Google Trends.

Conclusion

De nombreuses entreprises optent aujourd'hui pour une architecture basée sur les microservices. Et cela tient très certainement au fait que les microservices sont étroitement associés au concept de conteneur. L'utilisation de conteneurs à la place de machines virtuelles dans l'architecture réseau permet d'améliorer l'infrastructure NFVI par l'adoption des CNF.

Le chapitre suivant sera consacré aux deux technologies (Docker et Kubernetes) nécessaires pour réaliser notre projet.

Chapitre III:

Docker et Kubernetes

Introduction

Au cours des dernières années, l'utilisation des technologies de virtualisation dans les infrastructures réseaux considérablement augmentée. Ceci est principalement dû aux avantages en termes d'efficacité d'utilisation des ressources et de robustesse que procure la virtualisation. La virtualisation par conteneurs Docker et la virtualisation par hyperviseurs sont les deux principales technologies apparues sur le marché. Parmi elles, la virtualisation par conteneurs Docker se distingue par sa capacité de fournir un environnement virtuel léger et efficace, mais qui nécessite la présence d'un orchestrateur. Par conséquent, les entreprises adoptent la plateforme ouverte Kubernetes comme étant le gestionnaire standard des applications conteneurisées.

Dans ce chapitre, nous allons étudier deux technologies qui sont Docker et Kubernetes en décrivant leurs composants, leurs architectures et leurs avantages.

III.1 Docker

III.1.1 Définition

Docker est une plateforme logicielle open source qui permet de créer, de déployer et de gérer des conteneurs d'applications virtualisées sur un système d'exploitation (OS) commun. La technologie de conteneur Docker lancée en 2013 par la société Docker Inc en deux éditions Docker Communauté qui est une version gratuite pour la communauté et Docker Entreprise qui est une version payante destinée aux entreprises [10].

Le conteneur docker regroupe le service ou la fonction d'application avec toutes les bibliothèques, fichiers de configuration, dépendances et autres paramètres nécessaires au fonctionnement dans une image complète, portable et fonctionne toujours avec la même manière quel que soit l'environnement. Aussi, il est très simple de détruire une image avec Docker et c'est comme si rien ne s'est passé. Pour cela, le docker est considéré comme un outil qui aide à résoudre les problèmes courants liés à l'installation, la distribution et la gestion du logiciel.

Docker a été créé pour fonctionner sur la plate-forme Linux , mais s'est étendu pour offrir une meilleure prise en charge des systèmes d'exploitation non Linux, notamment Microsoft Windows, Apple OS ainsi des versions de Docker pour Amazon Web Services (AWS) et Microsoft Azure sont disponibles.

III.1.2 Architecture Docker

Docker est une application client-serveur, qui comprend les trois composants principaux : Docker Client, Docker Host et Docker Registry comme illustrer la figure III.1 :

- **Docker Client :** le client docker est utilisé pour déclencher des commandes Docker. Lorsqu'une commande est envoyée (construction, exécution de conteneur), le client docker envoi ces commandes vers le démon Docker, qui les traitera ensuite.
- **Docker Host :** Fournis un environnement complet pour exécuter des applications. Il comprend le démon docker qui traite les requêtes docker afin de gérer les différents objets docker tels que les images, les conteneurs ou les volumes de stockage.
- **Docker Registry :** permet de stocker et de distribuer des images de conteneurs. Il existe deux types de registres dans le Docker : un registre public également appelé Docker hub que tout le monde peut utiliser et qui est configuré par défaut pour rechercher des images, un registre privé utilisé pour partager des images au sein de l'entreprise.

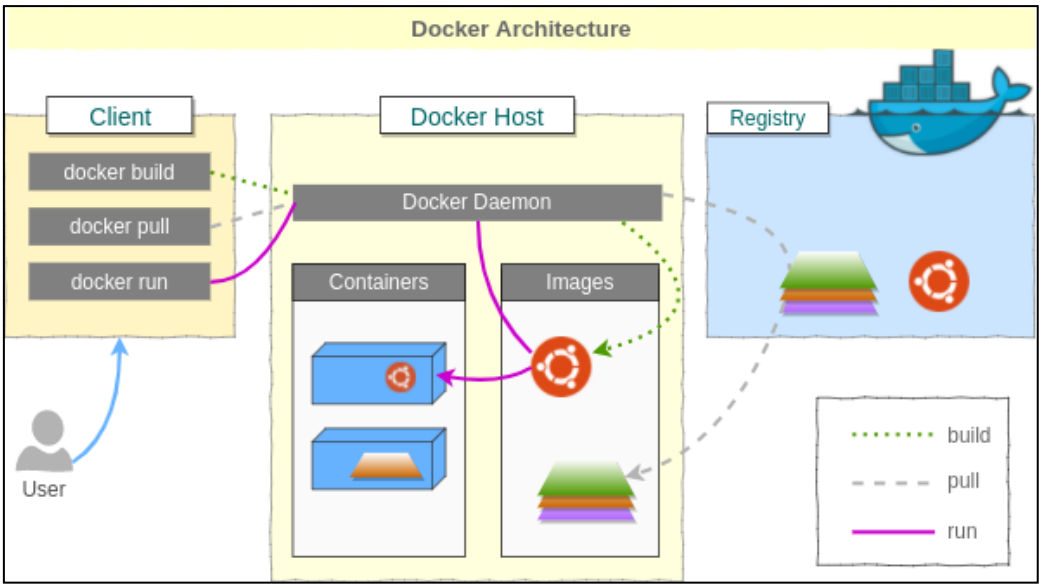

Figure .III.1 : Architecture docker

III.1.3 Objets Docker [11]

L'utilisation de Docker se base sur un ensemble d'objets qui sont générés par le Docker daemon, tels que les images, les conteneurs, les réseaux et les volumes (voir figure III.2).

1. Image

Une image est l'élément de base d'un conteneur. Il s'agit d'un ensemble de fichiers systèmes et de paramètres d'exécutions. Une image n'a pas d'état et est exécutée par la runtime docker. On peut créer une nouvelle image soit en modifiant un conteneur déjà en marche, soit en créant un Dockerfile basé sur une image qui existe déjà. Dockerfile est un fichier texte qui décrit une image, il est constitué d'instructions qui servent à décrire une image Docker et permet d'automatiser la « build » (création) d'images.

2. Conteneur

Un conteneur Docker représente une instance active et en cours d'exécution d'une image. Chaque conteneur Docker consomme une certaine part des ressources du système lors de son exécution, comme le processeur, la RAM, les interfaces réseau, etc. Un conteneur Docker peut être créé, lancé, interrompu et détruit.

3. Réseaux

Les réseaux permettent aux conteneurs isolés d'un système de communiquer entre eux. Il y a plusieurs types de réseau qui sont installés par défaut avec le Docker :

- **Bridge :** réseau par défaut de docker, il permet d'interconnecter les conteneurs connectés au même bridge. Bridge fourni une isolation entre l'hôte et les conteneurs.
- **Host :** permet l'accès au réseau de l'hôte sans aucune isolation.
- **Overlay :** réseau par défaut pour docker swarm. Il permet l'interconnexion de conteneur entre plusieurs nœuds docker.
- **None :** désactive tous les réseaux, aucun réseau attaché au conteneur.

4. Volume

Le Docker Volume est l'emplacement où l'on stocke les données utilisées par les conteneurs. L'idée est de stocker les données dans un répertoire spécifique du serveur hôte Docker plutôt que dans le système de fichiers par défaut du conteneur. Ce stockage externe, appelé volume de données, permet de rendre les données indépendantes de la vie du conteneur.

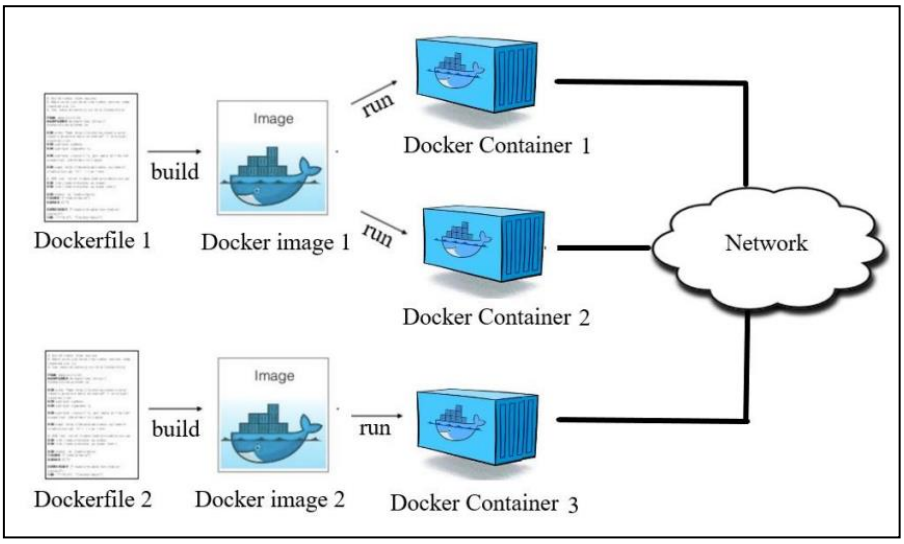

Figure .III.2 : Objets Docker

À mesure que les applications s'étendent sur plusieurs conteneurs déployés sur plusieurs serveurs, leur utilisation et mise à l'échelle devient plus complexe, il est nécessaire de pouvoir approvisionner, configurer, étendre et surveiller les conteneurs sur l'architecture de microservice. Pour cela, on utilise des outils d'orchestration de conteneurs, L'un de ces outils est Kubernetes.

III.2 Kubernetes

Kubernetes a pour objectif principal de dissimuler la complexité de la gestion d'un parc de conteneurs situés sur différents serveurs hôtes, qu'ils soient physiques ou virtuels ou alors dans le Cloud.

III.2.1 Architecture Kubernetes

L'architecture Kubernetes suit une architecture maître-esclave. Le composant principal de Kubernetes est le cluster constitué de deux types de machines. Les machines « Master» (ou Maitre) qui sont en charge du pilotage du cluster et des machines « Worker » sur lesquels sont déployés des Pods.

Un Pod est un environnement isolé permettant d'exécuter un ou plusieurs conteneurs formant une application conteneurisée partageant ainsi la même stack réseau (chaque pod se voit attribuer une adresse IP unique) et le même stockage, plus précisément un volume partagé, ce qui permet à ces conteneurs de partager plus facilement des données [12].

La figure suivante représente les différents composants d'un cluster Kubernetes :

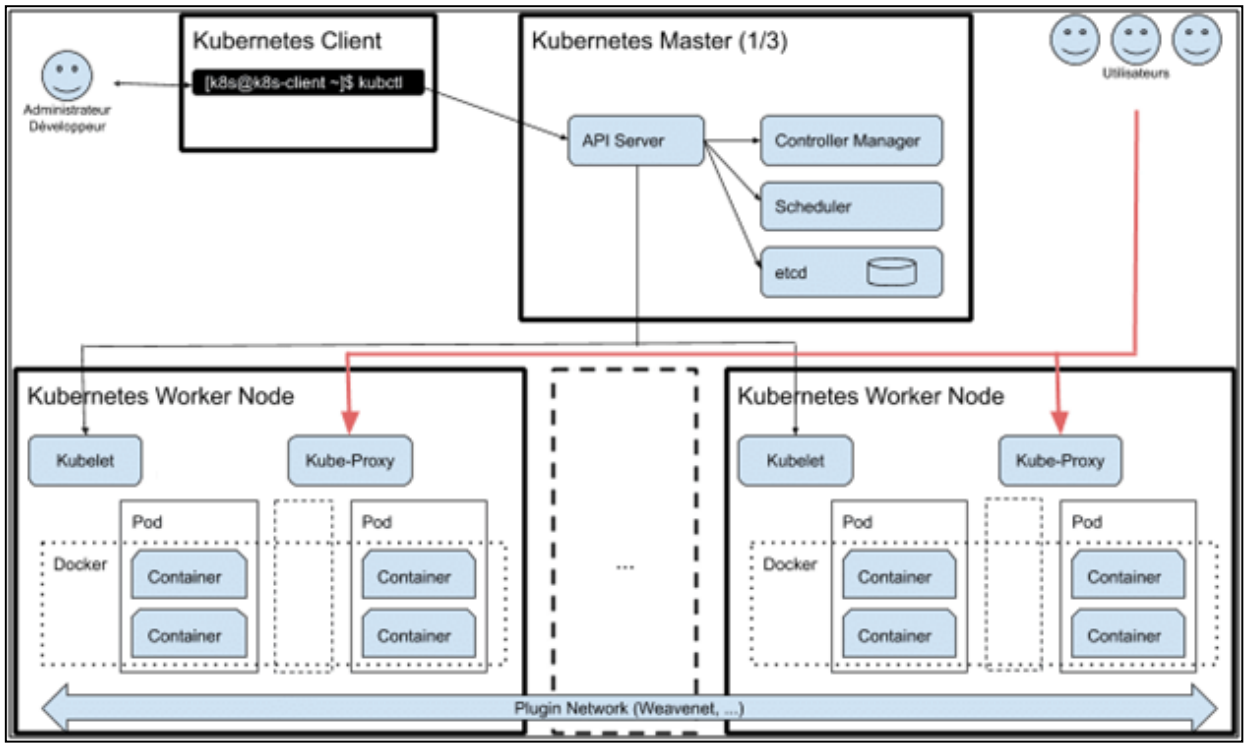

Figure .III.3 : Architecture Kubernetes.

III.2.1.1 Maitre (Master) [12]

C'est le plan de contrôle du cluster Kubernetes, la fonction principale du maître est d'organiser la création ou la suppression de workers ou de conteneurs. Il se charge également de la répartition du trafic selon les besoins. C'est à partir de cette machine « Maître » que les administrateurs paramètrent et interagissent avec l'ensemble de l'architecture. Les composantes d'un « Maître » sont les suivantes :

A. API server

API Server est un composant de communication utilisé pour mettre en relation le master avec les Workers. Il est aussi le seul à communiquer avec l'etcd et communique via une interface REST utilisée par l'outil en ligne de commande « kubectl ». Ce dernier permet d'exécuter des commandes dans les clusters pour gérer le cluster et les applications du cluster

B. Controller Manager

Le contrôleur surveille l'état du cluster via l'API serveur et apporte des modifications afin de ramener le cluster de l'état actuel instable vers un état stable. Ce composant est le garant de l'intégrité et de la disponibilité de l'architecture.

C. Scheduler (Planificateur)

Le scheduler est le service qui planifie le déploiement des pods sur les workers. En effet, il s'agit du composant en charge de la répartition des tâches entre les workers. Et ce, en fonction de la charge et des ressources nécessaires et disponibles.

D. Etcd

Etcd est une base de données qui stocke et assure la persistance de la configuration du cluster. Ce composant enregistre l'état actuel de l'ensemble des composants du cluster.

III.2.1.2 Workers (nœuds) [12]

Sont lesquels s'exécutent les applications conteneurisées. Sur chaque worker, nous avons des services qui s'exécutent à l'intérieur de pods, les éléments présents sur les workers sont les suivants :

A. Kublete

Il s'agit d'un agent qui s'exécute dans chaque nœud chargé de relayer les informations au Master. Il interagit avec la base de données Etcd du Master pour récupérer des informations afin de connaître les tâches à effectuer. Il assume la responsabilité de maintenir en bon état de fonctionnement les conteneurs d'un pod et s'assure qu'ils tournent conformément à la spécification. Il communique avec le Master et redémarre le conteneur défaillant en cas de crash.

B. Kube-Proxy

Le kube-proxy est un proxy réseau qui est exécuté sur chaque worker, il permet de gérer les adresses IP virtuelles des pods. Ceux-ci sont alors accessibles à la fois au sein du cluster et depuis l'extérieur. Il est également utilisé pour équilibrer la charge (load balancer) des services.

C. Container runtime :

Un moteur de conteneur utilisé pour gérer l'ensemble des conteneurs, tels que l'extraction d'images, le démarrage et l'arrêt de conteneurs. Le plus souvent, le moteur de conteneur utilisé par Kubernetes est Docker

Le plugin network est déployé au travers de tous les serveurs du cluster Kubernetes. Il a pour rôle la mise en œuvre du réseau entre les workers et les pods et entre les pods eux-mêmes. Ces plugins sont des solutions tierces telles que Calico, Flannel ou Weavenet.

Kubernetes peut être exécuté dans un cloud public, un cloud privé, sur site ou une combinaison de ceux-ci, à l'aide de machines physiques ou virtuelles. Les workers peuvent exécuter sur Linux ou Windows, tandis que les composants du plan de contrôle (Maitre) ne s'exécutent que sous Linux.

III.2.2 Fonctionnement du Kubernetes

Kubernetes s'exécute au-dessus d'un système d'exploitation et interagit avec les pods de conteneurs qui s'exécutent sur les workers. Le serveur maître Kubernetes reçoit les commandes de la part d'un administrateur et relaie ces instructions aux nœuds qui lui sont subordonnés. Ce système de transfert fonctionne avec une multitude de services et choisit automatiquement le nœud le plus adapté pour chaque tâche. Il alloue ensuite les ressources aux pods désignés dans ce nœud pour qu'ils effectuent la tâche requise.

Docker continue de jouer son rôle. Lorsque Kubernetes planifie un pod dans un nœud, le kubelet de ce nœud ordonne à Docker de lancer les conteneurs spécifiés. Le kubelet collecte ensuite en continu le statut de ces conteneurs via Docker et rassemble ces informations sur le serveur maître. Docker transfère les conteneurs dans le nœud et démarre/arrête ces conteneurs, comme d'habitude. La différence est l'origine des ordres : ils proviennent d'un système automatisé et non plus d'un administrateur qui assigne manuellement des tâches à tous les nœuds pour chaque conteneur.

III.2.3 Avantages de Kubernetes

Kubernetes est une solution open-source qui aide les entreprises à améliorer leur administration des applications dans des environnements IT hétérogènes. Ci-dessous, nous expliquons les principaux avantages du Kubernetes :

1. **Réduire les délais de développement et de la publication**

Kubernetes simplifie considérablement les processus de développement, de publication et de déploiement dans les scénarios où l'architecture est basée sur des microservices, l'application est décomposée en unités fonctionnelles qui communiquent entre elles via des API **:** l'équipe de développement peut ainsi être décomposée en groupes plus restreints, chacun spécialisé dans une seule fonctionnalité. Cette organisation permet aux équipes informatiques de travailler avec plus de concentration et d'efficacité, en accélérant les délais de publication.

2. **Optimisation des coûts en IT**

Grâce à une administration dynamique et intelligente des conteneurs, Kubernetes peut aider les organisations à économiser sur la gestion de leur écosystème, en garantie l'évolutivité sur plusieurs environnements. L'allocation des ressources est automatiquement modulée en fonction des besoins réels de l'application, tandis que les opérations manuelles de bas niveau sur l'infrastructure sont considérablement réduites, en partie grâce aux logiques natives d'autoscaling

(HPA horizontal pod autoscaler , VPA vertical pod autoscaler) et aux intégrations avec les principaux fournisseurs de cloud, capables de fournir des ressources de manière dynamique.

3. **Evolutivité et disponibilité**

Kubernetes est capable de faire évoluer les applications et les ressources d'infrastructure sous-jacentes en fonction des besoins contingents de l'organisation. Grâce à ses API natives d'autoscaling, telles que HPA et VPA, Kubernetes pourra demander dynamiquement de nouvelles ressources matérielles allouées à l'infrastructure fournissant le service, afin d'assurer les performances de celle-ci. Une fois l'urgence passée, Kubernetes réduira alors les ressources qui ne sont plus nécessaires en évitant ainsi le gaspillage.

4. **Flexibilité dans les environnements multi-cloud**

La conteneurisation et Kubernetes permettent de réaliser les promesses des nouveaux environnements hybrides et multi-cloud, garantissant le fonctionnement des applications dans n'importe quel environnement public et privé, sans pertes fonctionnelles ou de performances. Donc réduire le risque de manque d'interopérabilité de certaines solutions informatiques, qui oblige les organisations à se lier à un seul fournisseur.

5. **Self Healing**

Un autre avantage important de Kubernetes est de bien gérer les conteneurs individuels. Si un conteneur tombe en panne en raison d'une défaillance matérielle/logicielle, le système Kubernetes s'assure qu'il est remplacé en temps réel.

Cette caractéristique assure la haute disponibilité du système et sa capacité de rebondir après des pannes. Il s'agit d'une norme essentielle pour les applications métier en temps réel, car les temps d'arrêt peuvent entraîner directement une perte de revenus.

Conclusion

Ericsson et de nombreuses entreprises ont tendance à opter vers les microservices afin d’adopter des services réseau plus flexible, hautement évolutive et plus fiable avec des coûts réduit. Cependant, une architecture basée sur les microservices est également complexe et exige de nouvelles technologies, notamment docker et Kubernetes qui peuvent aider à simplifier et à gérer le déploiement de ces microservices. Toutes ces notions théoriques seront mises en pratique dans le prochain chapitre.

Chapitre IV:

Mise en place de la solution

Introduction

Après avoir accompli toute une étude théorique sur les réseaux SDN, NFV et la nouvelle approche de développement des applications moderne « microservices », nous allons mettre ces notions théoriques en pratique.

Ce chapitre sera dédié à la présentation et la mise en place de la solution SDN dans un cluster Kubernetes afin d'implémenter des fonctions réseaux conteneurisées basé sur les conteneurs Docker avec un ensemble de tests de fonctionnement.

IV.1 Présentation de la topologie

La figure IV.1 montre la topologie utilisée dans la mise en place de notre solution sur un cluster Kubernetes qui se compose d'un Master et deux Workers.

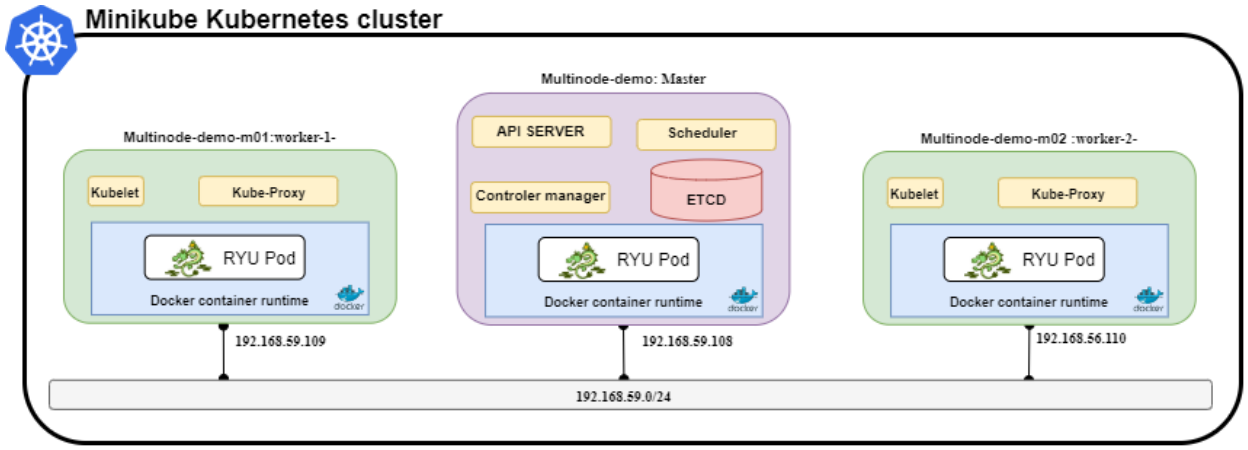

Figure. IV.1 : Topologie mise en œuvre.

Le déploiement de la solution passe par quatre étapes principales :

1. Première étape : Déploiement du cluster Kubernetes.
2. Deuxième étape : Préparation de l'image de Ryu.
3. Troisième étape : Déploiement du contrôleur Ryu sur le cluster Kubernetes
4. Quatrième étape : Création de la topologie du réseau SDN.

IV.2 Table d'adressage

Le tableau IV.1 résume les différentes adresses IP et caractéristiques de composants qui composent notre cluster Kubernetes.

Nom de composant	Caractéristiques	Adressage
Multinode-demo (Master)	2GB Ram, 2 vcpus	192.168.59.108
Multinode-demo-m01 (Worker1)	2GB Ram, 2 vcpus	192.168.59.109
Multinode-demo-m02 (Worker2)	2GB Ram, 2 vcpus	192.168.59.110

Tableau IV.1. Adressage et caractéristique de chaque composant.

IV.3 Déploiement de la solution

IV.3.1 Déploiement du cluster Kubernetes

Pour créer un cluster Kubernetes nous avons utilisé l'outil « Minikube » qui est une distribution allégée de Kubernetes, qui met en place facilement un cluster Kubernetes local et assure une performance maximale pour utiliser les fonctions de Kubernetes tout en profitant d'une charge de travail minimale. L'installation du Docker et du Minikube est expliquée dans l'Annexe B.

Nous allons mettre en place un cluster Kubernetes multi-nœud dans une machine virtuelle qui exécute trois nœuds (1 Master et 2 Workers) tournant sous la distribution Ubuntu en version 20.04.

La figure IV.2 représente la création de notre cluster kubernetes.

```
assia@ubuntu:~/Desktop/PFE_assia/SDN_Network$ minikube start  -p multinode-demo
😄  [multinode-demo] minikube v1.25.2 on Ubuntu 20.04
✨  Using the virtualbox driver based on existing profile
👍  Starting control plane node multinode-demo in cluster multinode-demo
🔄  Restarting existing virtualbox VM for "multinode-demo" ...
🐳  Preparing Kubernetes v1.23.3 on Docker 20.10.12 ...
    ▪ kubelet.housekeeping-interval=5m
    ▪ kubelet.cni-conf-dir=/etc/cni/net.mk
🔗  Configuring CNI (Container Networking Interface) ...
    ▪ Using image kubernetesui/dashboard:v2.3.1
    ▪ Using image k8s.gcr.io/metrics-server/metrics-server:v0.4.2
    ▪ Using image kubernetesui/metrics-scraper:v1.0.7
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🔎  Verifying Kubernetes components...
🌟  Enabled addons: storage-provisioner, default-storageclass, metrics-server, dashboard
👍  Starting worker node multinode-demo-m02 in cluster multinode-demo
🔄  Restarting existing virtualbox VM for "multinode-demo-m02" ...
🌐  Found network options:
    ▪ NO_PROXY=192.168.59.108
🐳  Preparing Kubernetes v1.23.3 on Docker 20.10.12 ...
    ▪ env NO_PROXY=192.168.59.108
🔎  Verifying Kubernetes components...
👍  Starting worker node multinode-demo-m03 in cluster multinode-demo
🔄  Restarting existing virtualbox VM for "multinode-demo-m03" ...
🌐  Found network options:
    ▪ NO_PROXY=192.168.59.108,192.168.59.109
🐳  Preparing Kubernetes v1.23.3 on Docker 20.10.12 ...
    ▪ env NO_PROXY=192.168.59.108
    ▪ env NO_PROXY=192.168.59.108,192.168.59.109
🔎  Verifying Kubernetes components...
🏄  Done! kubectl is now configured to use "multinode-demo" cluster and "default" namespace by default
assia@ubuntu:~/Desktop/PFE_assia/SDN_Network$
```

Figure. IV.2 : Création d'un cluster Kubernetes multi-nœud.

Ensuite, nous avons utilisé l'outil en ligne de commande « kubectl » afin de récupérer la liste des nœuds de notre cluster Kubernetes, comme le montre la figure suivante :

```
assia@ubuntu:~/Desktop/PFE_assia/SDN_Network$ kubectl get node -o wide
NAME                 STATUS   ROLES                  AGE     VERSION   INTERNAL-IP      EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION   CONTAINER-RUNTIME
multinode-demo       Ready    control-plane,master   4d21h   v1.23.3   192.168.59.108   <none>        Buildroot 2021.02.4   4.19.202         docker://20.10.12
multinode-demo-m02   Ready    <none>                 12m     v1.23.3   192.168.59.109   <none>        Buildroot 2021.02.4   4.19.202         docker://20.10.12
multinode-demo-m03   Ready    <none>                 11m     v1.23.3   192.168.59.110   <none>        Buildroot 2021.02.4   4.19.202         docker://20.10.12
```

Figure. IV.3 : Description du Master et des Workers

IV.3.2 Préparation de l'image de Ryu

La deuxième étape pour déployer la solution SDN sur Kubernetes consiste à créer l'image Ryu à partir d'un Dockerfile. Un Dockerfile est un document texte qui contient un ensemble d'instructions, chacune présente sur une ligne séparée. Les instructions sont exécutées l'une après l'autre pour créer une image Docker. La figure suivante (Figure IV.4) illustre le Dockerfile de l'image Ryu.

```
assia@ubuntu:~/Desktop/PFE_assia$ cat Dockerfile
FROM python:3.8.10

WORKDIR /root

COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

RUN git clone https://github.com/martimy/flowmanager

COPY SDN_Network /root
```

Figure. IV.4 : Dockerfile de l'image Ryu.

Les instructions du Dockerfile sont :

- **FROM :** Définit l'image de base à partir de laquelle la nouvelle image est dérivée.
- **WORKDIR :** Définit le répertoire de travail.
- **COPY :** Permet de copier des fichiers depuis notre machine locale vers le conteneur Docker. Le fichier « requirements.txt » contient tous les packages nécessaires pour la création d'une image Ryu (Voire la figure IV.5).
- **RUN :** Exécute des commandes dans l'image durant le build process. Chaque instruction RUN va créer une nouvelle couche.

```
assia@ubuntu:~/Desktop/PFE_assia$ cat requirements.txt
certifi==2021.10.8
charset-normalizer==2.0.12
debtcollector==2.5.0
dnspython==1.16.0
eventlet==0.30.2
greenlet==1.1.2
idna==3.3
msgpack==1.0.3
netaddr==0.8.0
oslo.config==8.8.0
oslo.i18n==5.1.0
ovs==2.16.0
pbr==5.8.1
PyYAML==6.0
repoze.lru==0.7
requests==2.27.1
rfc3986==2.0.0
Routes==2.5.1
ryu==4.34
six==1.16.0
sortedcontainers==2.4.0
stevedore==3.5.0
tinyrpc==1.1.4
urllib3==1.26.9
WebOb==1.8.7
wrapt==1.14.0
```

Figure. IV.5 : Packages de l'image Ryu.

Après la création de Dockerfile, nous avons exécuté la commande Docker Build (Figure IV.6) qui nous permet de construire une image Docker à partir de Dockerfile.

```
assia@ubuntu:~/Desktop/PFE_assia$ docker build -t assiapfe2022/ryu-ns-ids:v1 .
[+] Building 13.0s (12/12) FINISHED
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 230B
 => [internal] load metadata for docker.io/library/python:3.8.10
 => [auth] library/python:pull token for registry-1.docker.io
 => [1/6] FROM docker.io/library/python:3.8.10@sha256:f7981c32c931f07d053f6867d78
 => [internal] load build context
 => => transferring context: 3.27MB
 => CACHED [2/6] WORKDIR /root
 => CACHED [3/6] COPY requirements.txt ./
 => CACHED [4/6] RUN pip install --no-cache-dir -r requirements.txt
 => CACHED [5/6] RUN git clone https://github.com/martimy/flowmanager
 => [6/6] COPY SDN_Network /root
 => exporting to image
 => => exporting layers
 => => writing image sha256:3b17260b5a9af4ce5d6360273320bdb3c82b0bcfbb5b36099776a
 => => naming to docker.io/assiapfe2022/ryu-ns-ids:v1
assia@ubuntu:~/Desktop/PFE_assia$
```

Figure. IV.6 : Build de l'image docker.

Nous allons maintenant publier l'image docker sur le Docker Hub. Pour cela, nous devons créer un répertoire nommé « assiapfe2022 » et on utilise la commande Docker Push (Figure IV.7).

```
assia@ubuntu:~/Desktop/PFE_assia$ docker push  assiapfe2022/ryu-ns-ids:v1
The push refers to repository [docker.io/assiapfe2022/ryu-ns-ids]
651d6e055e1d: Pushed
17791dfe37b5: Layer already exists
cc904ad6ebd6: Layer already exists
77bfdfb2e29f: Layer already exists
5f70bf18a086: Layer already exists
6ab97ebc930b: Layer already exists
e726038699f2: Layer already exists
b8e0cb862793: Layer already exists
4b4c002ee6ca: Layer already exists
cdc9dae211b4: Layer already exists
7095af798ace: Layer already exists
fe6a4fdbedc0: Layer already exists
e4d0e810d54a: Layer already exists
4e006334a6fd: Layer already exists
v1: digest: sha256:683ad5fc1cead6a39828a363192c2622d7d8a1a370a2c92d884fdc62e8f5c1b2 size: 3264
assia@ubuntu:~/Desktop/PFE_assia$
```

Figure. IV.7 : Push de l'image docker.

IV.3.3 Déploiement du contrôleur Ryu sur le cluster kubernetes

Après la création de l'image de contrôleur Ryu et la publier sur le Docker Hub, nous allons passer à l'étape suivante, qui est le déploiement et l'exécution d'un contrôleur RYU en tant que microservice à l'intérieur des pods du cluster Kubernetes, en suivant les étapes suivantes:

- **La création des PODs**

Nous allons créer un fichier texte au format YAML de type déploiement dans lequel nous avons défini notre état souhaité pour le cluster.

Notre fichier de déploiement « ryu-deployment.yaml » est le suivant :

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ryu-deployment
  labels:
    app: ryu
spec:
  replicas: 3
  selector:
    matchLabels:
      app: ryu
  template:
    metadata:
      labels:
        app: ryu
    spec:
      containers:
      - name: ryu
        image: assiapfe2022/ryu-ns-ids:v1
        command: ["/bin/sh","-c"]
        args: ["ryu-manager --observe-links ~/flowmanager/flowmanager.py  ./ryu_controller_slicing.py "]
        resources:
          limits:
            cpu: 300m
          requests:
            cpu: 200m
```

Figure. IV.8 : Fichier YAML de déploiement

Ce fichier de déploiement YAML contient les principales spécifications suivantes :

- apiVersion : Ceci spécifie la version de l'API de l'objet « déploiement Kubernetes ».
- kind : Décrit le type d'objet à créer. Dans ce cas, c'est un objet de type déploiement.
- metadata : C'est l'ensemble de données permettant d'identifier de manière unique le déploiement, il s'agit essentiellement du nom de déploiement.
- spec : Nous avons déclaré l'état et les caractéristiques de déploiement souhaitées, nous avons spécifié le nombre de répliques (le nombre de pods) et le nom de l'image docker de « assiapfe2022/ryu-ns-ids:v1 » que nous allons créer précédemment.

Créons maintenant notre déploiement, à l'aide de la commande ci-dessous :

```
assia@ubuntu:~/Desktop/PFE_assia/SDN_Network/kube$ kubectl create -f ryu-deployment.yaml
deployment.apps/ryu-deployment created
```

Figure. IV.9 : Exécution du fichier Yaml.

Revérifions ensuite que nos pods ont bien été créés et qu'ils sont en mode running, comme il est montré dans la figure IV.10.

```
Every 2.0s: kubectl get pod -o wide

NAME                              READY   STATUS    RESTARTS   AGE    IP             NODE
ryu-deployment-cbfcb6fd4-5pmgk    1/1     Running   0          6s     10.244.0.13    multinode-demo
ryu-deployment-cbfcb6fd4-qw6g5    1/1     Running   0          6s     10.244.1.7     multinode-demo-m02
ryu-deployment-cbfcb6fd4-v9zm7    1/1     Running   0          132m   10.244.2.2     multinode-demo-m03
```

Figure. IV.10 : Description des pods.

- **Création des services :**

Les pods sont conçus pour être accessible qu'en interne du cluster, pour permettre aux pods d'être accessible depuis l'extérieur de notre cluster Kubernetes. Nous exploiterons deux services de type Nodeport, le premier service nommé « ryu-controller-service » qui permet un accès au contrôleur Ryu et le deuxième service nommé « ryu-service» qui permet un accès à l'interface graphique FlowManager. Les deux services sont créés depuis de fichiers YAML comme il est montré dans la figure IV.11.

```yaml
#Fichier yaml de serice 1 : contrôleur Ryu
kind: Service
apiVersion: v1
metadata:
  name: ryu-controller-service
spec:
  selector:
    app: ryu
  type: NodePort
  ports:
  - protocol: TCP
    port: 6653
    targetPort: 6653
    nodePort: 30897
```

```yaml
#Fichier yaml de serice 2 : GUI FlowManager
kind: Service
apiVersion: v1
metadata:
  name: ryu-service
spec:
  selector:
    app: ryu
  type: NodePort
  ports:
  - protocol: TCP
    port: 8080
    targetPort: 8080
    nodePort: 30952
```

Figure. IV.11 : Fichiers YAML de services.

À la suite de l'exécution de deux fichiers YAML, nous allons vérifier que nos services ont bien été créés à l'aide de la commande ci-dessous :

```
assia@ubuntu:~/Desktop$ kubectl get svc -o wide
NAME                    TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE    SELECTOR
kubernetes              ClusterIP   10.96.0.1       <none>        443/TCP          3d2h   <none>
ryu-controller-service  NodePort    10.109.63.147   <none>        6653:30897/TCP   3d1h   app=ryu
ryu-service             NodePort    10.110.163.89   <none>        8080:30952/TCP   3d1h   app=ryu
```

Figure. IV.12 : Description de services

Les services prennent par défaut l'adresse IP du nœud master et le numéro de port indiqué dans la phase de la création du service (Nodeport).

Maintenant, nous allons vérifier l'accessibilité de l'interface graphique FlowManager de contrôleur Ryu en utilisant URL : 192.168.59.108 :30952.

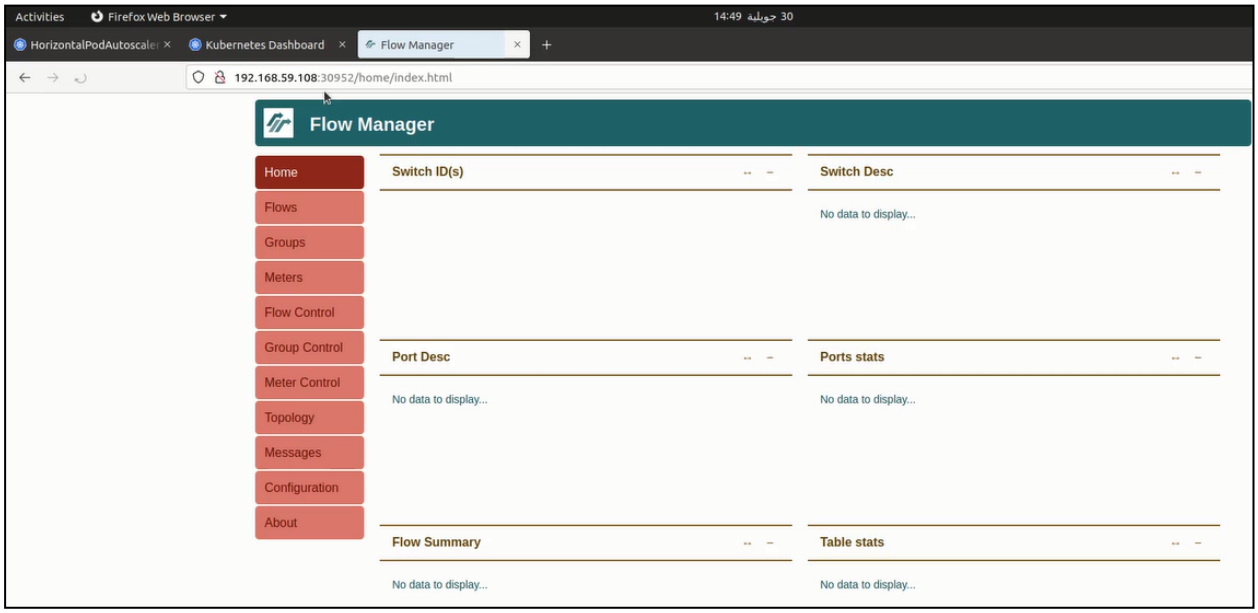

Figure. IV.13 : Accès à FlowManager

La figure IV.13 montre que nos pods qu'exécutent le contrôleur Ryu sont accessibles depuis l'interface web FlowManager, cependant notre réseau SDN n'est pas encore déployé. Dans la section suivante, nous avons détaillé la création de notre architecture du réseau SDN.

IV.3.4 Création de la topologie SDN

Notre topologie de réseau défini par logiciel (SDN) utilise des conteneurs Docker comme hôtes à l'aide de l'émulateur ContainerNet. Ce dernier est une extension de l'émulateur de réseau Mininet, il permet de mettre en place des typologies réseaux conteneurisées et d'exécuter des fonctions réseau à l'intérieur de conteneurs Docker.

La figure suivante montre la topologie de réseaux SDN basé sur le contrôleur Ryu utilisée dans notre projet :

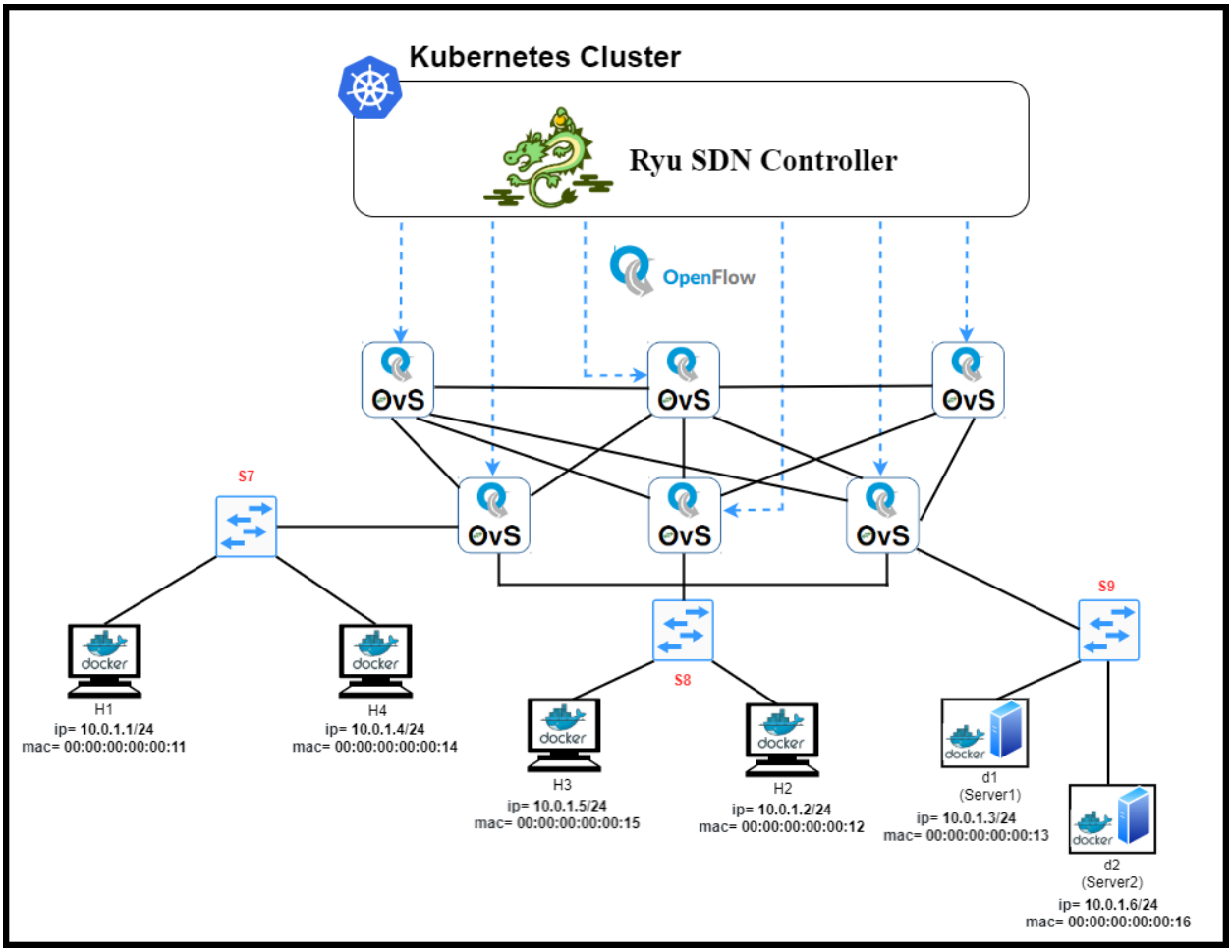

Figure. IV.14 : Topologie de réseau SDN.

Dans cette architecture, six Open vSwitch (OVS) sont utilisés pour construire le réseau core et trois commutateurs (Switch S7, S8 et S9) sont utilisés pour fournir un réseau d'accès aux quatre machines hôtes et deux serveurs. La définition de cette topologie est basée sur un script Python appelé : Topologie_ryu.py (voire la figure IV.15).

```
c1=net.addController('c1', controller=RemoteController,ip='192.168.59.108', port=30897)

h1 = net.addDocker('h1', ip='10.0.1.1/24',mac="00:00:00:00:00:11",dimage=client_image)
h2 = net.addDocker('h2', ip='10.0.1.2/24',mac="00:00:00:00:00:12", dimage=client_image)
h3 = net.addDocker('h3', ip='10.0.1.5/24', mac="00:00:00:00:00:15", dimage=client_image)
h4 = net.addDocker('h4', ip='10.0.1.4/24', mac="00:00:00:00:00:14", dimage=client_image)
d1 = net.addDocker('d1', ip='10.0.1.3/24',mac="00:00:00:00:00:13", dimage=server_image)
d2 = net.addDocker('d2', ip='10.0.1.6/24', mac="00:00:00:00:00:16", dimage=server_image)

s1 = net.addSwitch("s1")
s2 = net.addSwitch("s2")
s3 = net.addSwitch("s3")
s4 = net.addSwitch("s4")
s5 = net.addSwitch("s5")
s6 = net.addSwitch('s6')
s7 = net.addSwitch('s7', cls=OVSSwitch)
s8 = net.addSwitch('s8', cls=OVSSwitch)
s9 = net.addSwitch('s9', cls=OVSSwitch)

net.addLink(s1,s7)
net.addLink(s2,s8)
net.addLink(s5,s9)
net.addLink(s1, s2)
net.addLink(s1, s3)
net.addLink(s2, s4)
net.addLink(s3, s5)
net.addLink(s2, s3)
net.addLink(s4, s5)
net.addLink(s6, s1)
net.addLink(s6, s2)
net.addLink(s6, s3)
net.addLink(s6, s4)
net.addLink(s6, s5)
net.addLink(h1,s7)
net.addLink(h2,s8)
net.addLink(d1,s9)
net.addLink(h4,s7)
net.addLink(h3,s8)
net.addLink(d2,s9)
```

Figure. IV.15 : Fichier python de notre Topologie SDN.

Après l'exécution de script, L'application FlowManager affiche dans la page « Topologie » tous les Open vSwitch et les liens entre les commutateurs et les hôtes.

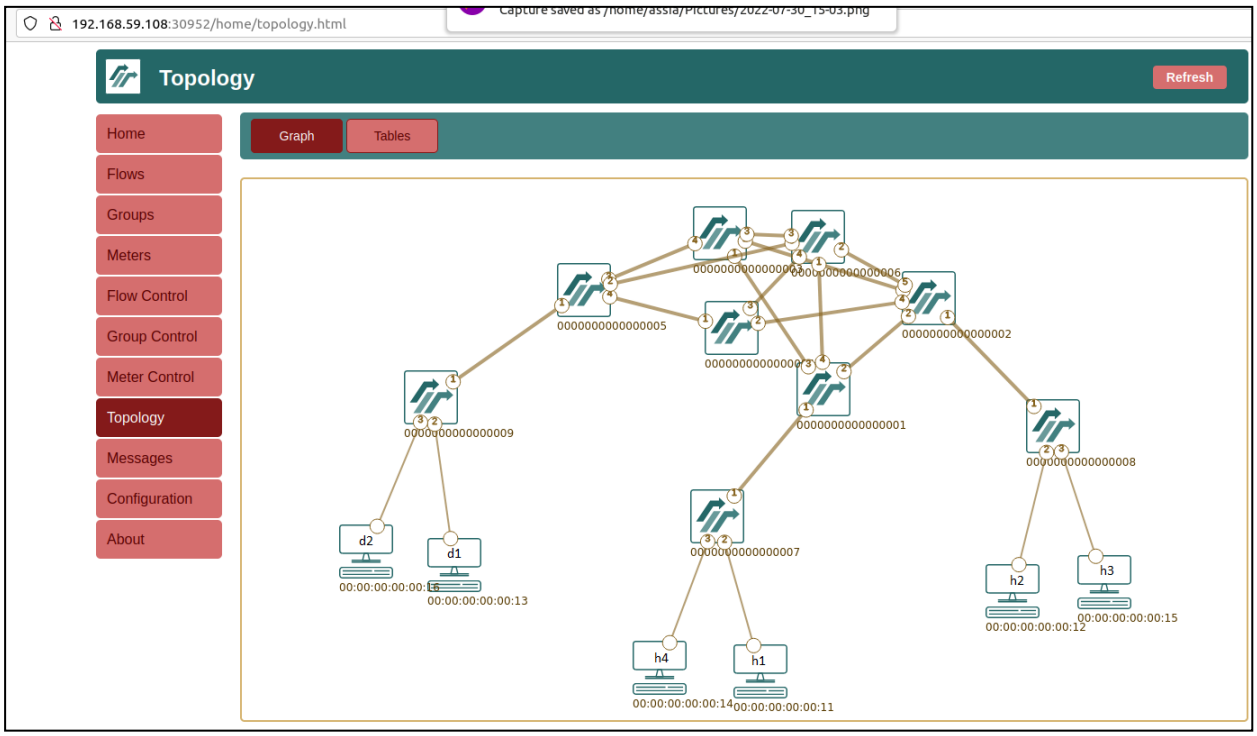

Figure. IV.16 : Illustration de topologie SDN depuis FlowManager .

Les adresses des différents hôtes et des serveurs sont données par le tableau suivant :

Conteneurs	Nom	Adresse IP	Adresse MAC
Hôtes	**h1 (Host 1)**	10.0.1.1	00:00:00:00:00:11
	h2 (Host 2)	10.0.1.2	00:00:00:00:00:12
	h3 (Host 3)	10.0.1.5	00:00:00:00:00:15
	h4 (Host 4)	10.0.1.4	00:00:00:00:00:14
Serveurs	**d1 (serveur 1)**	10.0.1.3	00:00:00:00:00:13
	d2 (serveur 2)	10.0.1.6	00:00:00:00:00:16

Tableau IV.2. Adressage d'hôtes et de serveurs.

IV.3.4.1 Déploiement le découpage du réseau (Network Slicing)

Le découpage du réseau ou Network Slicing consiste à découper un réseau en sous-réseaux logiques, appelés "tranches" ou "slices". Chaque tranche de réseau est conçue, déployée et gérée de manière indépendante, mais déployée sur une infrastructure physique commune.

Dans ce contexte, nous allons utiliser cette technique pour découper notre topologie de réseau SDN en tranches (voire figure IV.17)

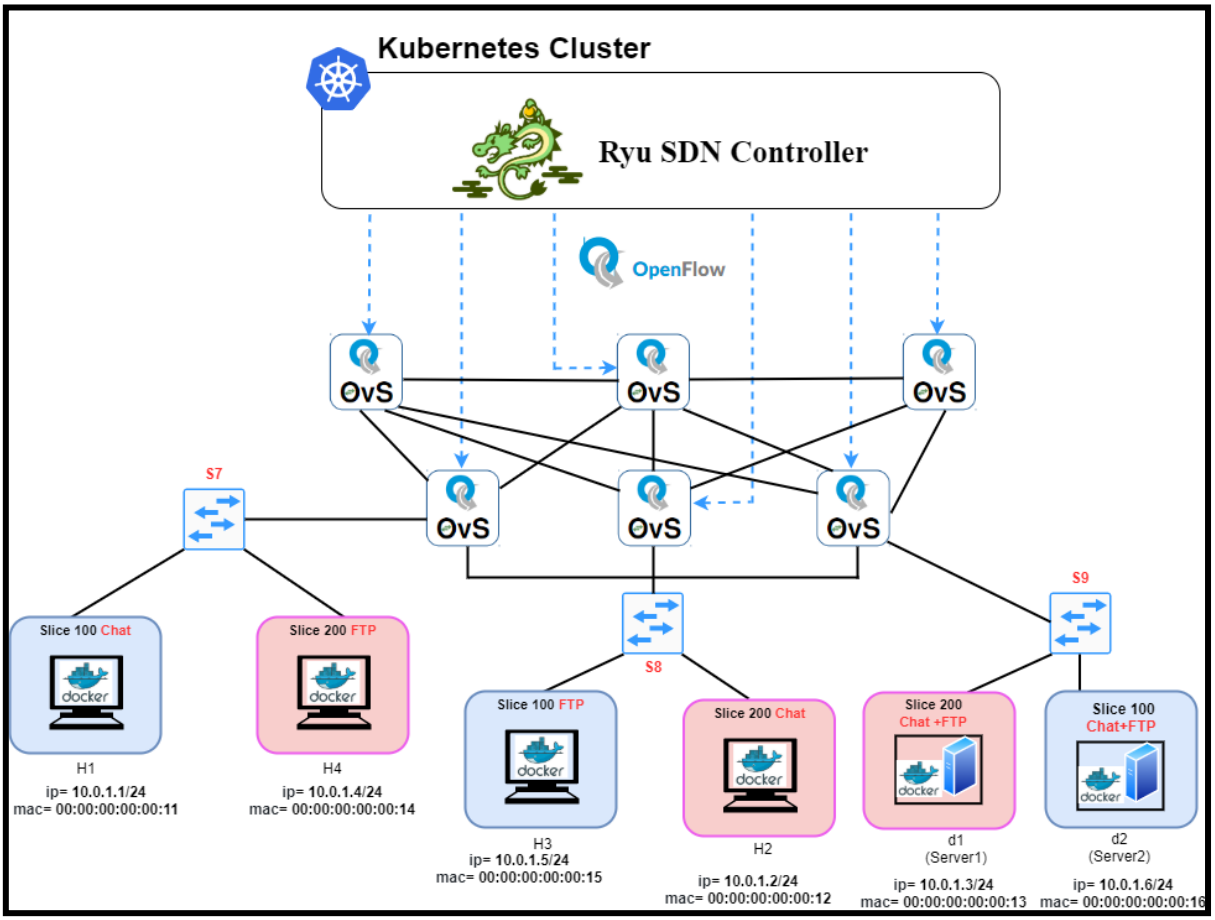

Figure. IV.17 : Déploiement du découpage du réseau.

Le but de cette topologie est de créer deux tranches à l'aide du protocole ICMP, à savoir la tranche 100 et la tranche 200. La tranche 100 comprend Host 1, Host 3 et le serveur 2, tandis que la tranche 200 comprend Host 4, Host 2 et le serveur 1. Les appareils Slice 100 ne peuvent communiquer entre eux que via le ping ICMP, qui est totalement indépendant des appareils Slice 200. Cela s'applique également de la même manière à la tranche 200.

De plus, des tranches basées sur le service sont également implémentées dans cette topologie à l'aide du protocole TCP et ces tranches sont définies comme tranche 1 et tranche 2. La tranche 1 composée de Host 1, Host 2 peut obtenir que le service chat avec le port TCP 1060, tandis que la tranche 2 composée de Host 3, Host 4 peut obtenir que le service FTP (File Transfer Protocol) avec le port TCP 21. Les deux services ont été exécutés sur le même serveur 1 mais un seul service qui se trouve dans une tranche spécifique peut être atteint par les hôtes de cette tranche.

Le contrôleur RYU est responsable de la déclaration des tranches (Slices) ICMP et TCP comme l'illustre la figure suivante :

```
# Slices 100 et 200 pour le découpage réseau à base du protocole ICMP
slices_data2 = [(100,"00:00:00:00:00:11","00:00:00:00:00:15","00:00:00:00:00:16",),
                (200,"00:00:00:00:00:14","00:00:00:00:00:12","00:00:00:00:00:13",) ]

# Slices 1 et 2 pour le découpage réseau à base du protocole TCP
slices_data4 = [(1,"00:00:00:00:00:11","00:00:00:00:00:12","00:00:00:00:00:13",1060),
                (2,"00:00:00:00:00:14","00:00:00:00:00:15","00:00:00:00:00:13",21)]
```

Figure. IV.18 : Configuration le découpage du réseau.

La description des slices permet à un utilisateur de profiter d'une fonction réseaux dédiée et d'un accès partagé. Le client (utilisateur) étant ici le demandeur de service auprès d'un opérateur et celui qui utilise le service en bout de chaîne (terminal). L'isolation opérationnelle permet donc de partager des ressources matérielles et logicielles comme des hébergeurs de cloud, en isolant les fonctions réseaux entre elles.

Maintenant nous allons observer la connectivité entre les différentes tranches du réseau :

- Le ping entre les hôtes dans la même Slice ICMP passe sans aucun problème et le ping entre les différentes tranches a échoué (Voire figure IV.19).

```
*** Testing connectivity
*** Ping: testing ping reachability
h1 -> X X X h3 d2
h2 -> X d1 h4 X X
d1 -> X h2 h4 X X
h4 -> X h2 d1 X X
h3 -> h1 X X X d2
d2 -> h1 X X X h3
*** Results: 60% dropped (12/30 received)
*** Running CLI
*** Starting CLI:
containernet>
```

Figure. IV.19 : Test de la connectivité.

IV.4 Tests de la solution

Après la mise en place de la solution SDN au sein d'un cluster Kubernetes, il serait nécessaire d'effectuer les tests de bon fonctionnement de notre application. Nous avons subdivisé ces tests en 3 catégories.

IV.4.1 Test du fonctionnement du réseau SDN

Pour ce test, nous avons utilisé l'analyseur de paquets Wireshark. Celui-ci nous permettra de visualiser le trafic openflow de réseau SDN

- La figure IV.20 assure que la communication entre le contrôleur et les Open vSwitch est bien établie, un message hello envoyé du contrôleur Ryu via le port TCP 30897 aux OVS a été capturé. Il s'agit de l'initialisation du canal OpenFlow, qui est le chemin de données du protocole OpenFlow entre le contrôleur et les OVS.

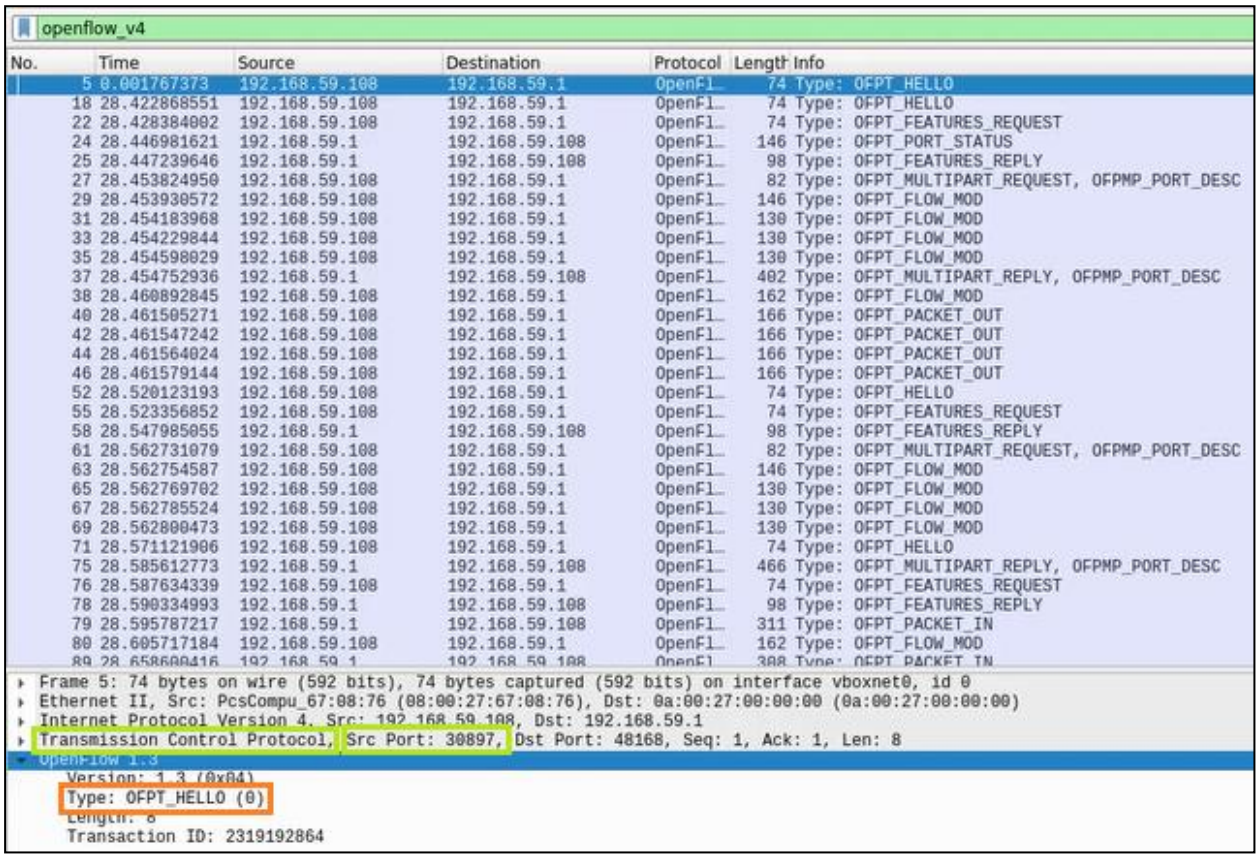

Figure. IV.20 : Initialisation du canal openflow

- Un paquet ICMP a été capturé lorsque nous allons lancer un ping de Host 1 vers Serveur1 (d1).

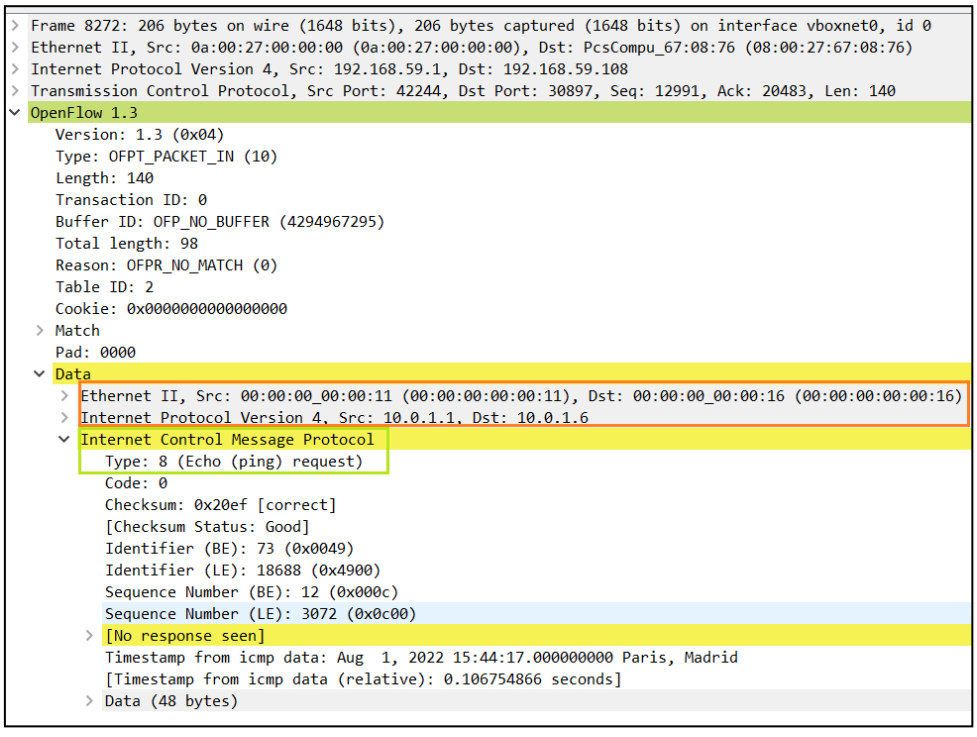

Figure. IV.21 : Capture packet ICMP

IV.4.2 Test Auto-résiliente (Self Healing)

Kubernetes vérifie constatement l'état du cluster pour s'assurer qu'il correspond à l'état optimal. Si l'un des pods de contrôleurs RYU s'arrête de manière inattendue, l'autre pod commencera automatiquement à s'exécuter pour maintenir le nombre de pods à trois à tout moment. Tout cela est géré automatiquement par le déploiement Kubernetes. Pour montrer le « Self healing » de kubernetes, nous avons effectué ce test comme suit :

Nous allons exécuter la commande suivante pour vérifier que le nombre de pods dans le cluster est trois :

```
Every 2.0s: kubectl get pod -o wide

NAME                                READY   STATUS    RESTARTS   AGE    IP             NODE
ryu-deployment-cbfcb6fd4-5pmgk      1/1     Running   0          10s    10.244.0.13    multinode-demo
ryu-deployment-cbfcb6fd4-qw6g5      1/1     Running   0          10s    10.244.1.7     multinode-demo-m02
ryu-deployment-cbfcb6fd4-v9zm7      1/1     Running   0          132m   10.244.2.2     multinode-demo-m03
```

Figure. IV.22 : Description des pods.

Ensuit, nous avons supprimé un pod parmi ces 3 pods en utilisant la commande montrée dans la figure suivante :

```
assia@ubuntu:~/Desktop/PFE_assia/SDN_Network$ kubectl delete pod ryu-deployment-cbfcb6fd4-qw6g5
pod "ryu-deployment-cbfcb6fd4-qw6g5" deleted
```

Figure. IV.23 : Suppression d'un pod.

Pour voir le comportement de Kubernetes face à ce problème, nous avons vérifié de nouveau le nombre de pods.

```
Every 2.0s: kubectl get pod -o wide

NAME                                READY   STATUS              RESTARTS   AGE    IP             NODE
ryu-deployment-cbfcb6fd4-5pmgk      1/1     Running             0          23s    10.244.0.13    multinode-demo
ryu-deployment-cbfcb6fd4-fclk6      0/1     ContainerCreating   0          3s     <none>         multinode-demo-m02
ryu-deployment-cbfcb6fd4-qw6g5      1/1     Terminating         0          23s    10.244.1.7     multinode-demo-m02
ryu-deployment-cbfcb6fd4-v9zm7      1/1     Running             0          132m   10.244.2.2     multinode-demo-m03
```

Figure. IV.24 : Vérification des pods

La figure IV.21 montre qu'un nouveau pod vient juste d'être créé par Kubernetes et que notre déploiement contient 3 pods au lieu de 2 pods. Cela signifie que Kubernetes a rapidement recréé un nouveau pod pour remplacer celui que nous avons supprimé et cela d'une manière automatique (sans intervention de l'utilisateur).

IV.4.3 Test Auto-Scaling

L'objectif est de gérer la situation lorsqu'il y a une augmentation de la charge du travail sur le contrôleur SDN a cause d'une augmentation du trafic réseau ou d'un changement dans la topologie du réseau. Si le nombre de requetes continue d'augmenter, l'utilisation du processeur et la consommation de mémoire du contrôleur SDN augmenteront, ce qui ralentira la performance de réseau. Nous avons donc dû créer un HorizontalPodAutoscaler (HPA) dans le but de mettre automatiquement à l'échelle la charge de travail pour répondre à la demande.

D'abord, nous allons créer un HPA qui maintient entre 3 et 10 répliques des Pods que nous avons créés dans la première étape de dépliement lorsque l'utilisation du processeur dépasse le seuil de 50 % à l'aide de la commande suivante :

```
assia@ubuntu:~/Desktop/PFE_assia/SDN_Network/kube$ kubectl autoscale deployment ryu-deployment --cpu-percent=50 --min=3 --max=10
horizontalpodautoscaler.autoscaling/ryu-deployment autoscaled
```

Figure. IV.25 : Configuration de la mise à l'échelle automatique.

Le HPA augmente et diminue le nombre de répliques (en mettant à jour le déploiement) pour maintenir une utilisation moyenne du CPU de 50 % pour tous les pods. Comme chaque pod demande 200 milli-cores par exécution de kubectl, cela signifie une utilisation moyenne du CPU de 100 milli-cores.

Ensuite, nous allons voir comment l'autoscaling réagit à une augmentation de la charge. Pour ce faire, nous allons lancer un conteneur appelé « load-generator » pour agir comme un client et envoyer une boucle infinie de requêtes au contrôleur RYU (Voire figure IV.26).

```
assia@ubuntu:~/Desktop/PFE_assia/SDN_Network/kube$ kubectl run -i --tty load-generator --rm --image=busybox:1.28 --restart=Never -- /bin/sh \
> -c "while sleep 0.01; do wget -q \
>  -O- http://192.168.59.108:30952/home/index.html; done"
```

Figure. IV.26 : Lancer une charge de trafic au contrôleur RYU.

Maintenant, nous allons vérifier l'état actuel de notre autscaling HPA, au bout d'une minute, nous allons voir que la consommation du CPU a augmenté à 305% (supérieur à la 50 %). En conséquence, le HPA augmentait automatiquement les répliques de pods à 7 en raison de l'augmentation de la charge (Voire les figures IV.27 et IV.28).

```
Every 2.0s: kubectl get hpa

NAME             REFERENCE                   TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
ryu-deployment   Deployment/ryu-deployment   305%/50%  3         10        7          27m
```

Figure. IV.27 : Vérification l'état de HPA.

```
Every 2.0s: kubectl describe hpa ryu-deployment

Warning: autoscaling/v2beta2 HorizontalPodAutoscaler is deprecated in v1.23+, unavailable in v1.26+; use autoscaling/v2 HorizontalPodAutoscaler
Name:                                                  ryu-deployment
Namespace:                                             default
Labels:                                                <none>
Annotations:                                           <none>
CreationTimestamp:                                     Tue, 23 Aug 2022 14:46:29 +0100
Reference:                                             Deployment/ryu-deployment
Metrics:                                               ( current / target )
  resource cpu on pods  (as a percentage of request):    305% / 50%
Min replicas:                                          3
Max replicas:                                          10
Deployment pods:                                       7 current / 7 desired
Conditions:
  Type            Status  Reason            Message
  ----            ------  ------            -------
  AbleToScale     True    SucceededRescale  the HPA controller was able to update the target scale to 3
  ScalingActive   True    ValidMetricFound  the HPA was able to successfully calculate a replica count from cpu resource utilization (percentage of request)
  ScalingLimited  True    TooFewReplicas    the desired replica count is less than the minimum replica count
Events:
  Type    Reason             Age                From                       Message
  ----    ------             ----               ----                       -------
  Normal  SuccessfulRescale  3s (x3 over 2m19s)  horizontal-pod-autoscaler  New size: 3; reason: All metrics below target
```

Figure. IV.28 : Description de HPA.

D'après la figure IV.29, nous voyons que le nombre de pods qui exécute le contrôleur RYU est augmenté à sept répliques après l'utilisation de HPA.

```
Every 2.0s: kubectl get pods

NAME                                READY   STATUS              RESTARTS   AGE
load-generator                      1/1     Running             0          21m
ryu-deployment-cbfcb6fd4-9d6f6      0/1     ContainerCreating   0          9s
ryu-deployment-cbfcb6fd4-d94mw      1/1     Running             0          51m
ryu-deployment-cbfcb6fd4-hmwqc      1/1     Running             0          84s
ryu-deployment-cbfcb6fd4-lssxw      0/1     ContainerCreating   0          9s
ryu-deployment-cbfcb6fd4-mz28p      0/1     ContainerCreating   0          9s
ryu-deployment-cbfcb6fd4-pvnx7      0/1     ContainerCreating   0          9s
ryu-deployment-cbfcb6fd4-qslhf      1/1     Running             0          51m
```

Figure. IV.29 : Description des pods après l'utilisation de HPA.

Donc, nous avons réussi à implémenter la solution SDN en tant qu'application conteneurisée basée sur la plateforme d'orchestration Kubernetes qui apporte de nombreux avantages à l'infrastructure NFV d'Ericsson notamment les points suivants :

1. Utilisation des ressources du système efficace :

Nous allons exécuter la solution SDN en tant qu'application conteneurisée légère et simple, nous pouvons déployer plusieurs instances partageant la même ressource de système d'exploitation dans une seule machine virtuelle. Plutôt que de fournir une machine virtuelle entière pour exécuter une seule application de contrôleur qui nécessite autant de mémoire et CPU pendant les périodes d'inactivité et gaspille de la bande passante.

2. Evolutivité automatique:

Notre application peut être mise à l'échelle automatiquement (auto-scaling). Cela génère immédiatement de nouveaux pods au nombre de répliques mentionnées dans le fichier déploiement YAML lorsque la charge de trafic est augmentée.

3. Haute disponibilité :

La haute disponibilité est obtenue grâce à la création des services de type Endpoint. Ce dernier garantit que la requête de l'utilisateur atteint l'un des pods à partir d'un groupe de pods exécutant la même l'application. Cela garantit l'accessibilité de l'application même si un pod est supprimé/cassé. La demande sera redirigée vers l'autre pod.

4. Auto-résiliente (Self Healing)

L'objet kubernetes Replica-Set garantit que le nombre de répliques souhaitées de nos pod doit être bien respecté. Ainsi, si un pod est détruit ou supprimé d'une manière ou d'une autre, le Replica-Set entre en action et déploie un nouveau pod automatiquement en assurant le nombre de pod spécifié. Cela facilite une architecture résiliente plus rapide et fiable avec une bande passante acceptée.

Conclusion

Dans ce chapitre, nous avons déployé la solution SDN en exécutant du contrôleur RYU comme un microservice à l'intérieur des PODs d'un cluster Kubernetes, ce qui nous a permis d'atteindre notre objectif, où nous pouvons fournir une architecture de réseau SDN hautement disponible et auto-résiliente en cas de défaillance grâce à l'objet Kubernetes Replica-Set et le service Eendpoint. Ainsi, nous allons assurer la mise à l'échelle automatique de l'application du contrôleur en cas de surcharge avec la fonctionnalité de Kubernetes l'auto-sacling.

Chapitre V:

Étude managériale

Introduction

Notre travail dans le cadre du projet de fin d'études (PFE) s'est déroulé au niveau de l'entreprise Ericsson. Depuis sa création, ERICSSON s'est fixé des objectifs principaux à savoir, la satisfaction et la fidélisation des clients, l'innovation et le progrès technologique. Ce qui lui a permis d'augmenter la confiance, de garantir de bonnes relations avec les différents clients et d'acquérir de nouveaux abonnés.

Dans cette partie du mémoire, nous allons présenter les différentes phases de notre projet avec illustration par un diagramme de GANTT & WBS. Ainsi, qu'une enquête sur terrain et nous finissons par une synthèse globale.

V.1. Présentation de l'organisme d'accueil

ERICSSON est une société multinationale suédoise de réseautage et de télécommunications dont le siège est à Stockholm. La société propose des services, des logiciels et des infrastructures dans le domaine des technologies de l'information et de la communication pour les opérateurs de télécommunications [15].

Ericsson d'Algérie contient deux départements importants NMSD (Network Management Service Delivery) et DCC (Digitalisation & Cloud Computing). Notre projet s'est déroulé au département NMSD qui a pour mission de fournir des services de gestion de réseau et d'assurer un bon fonctionnement de réseau.

V.2. Outils de réalisation de Management de PFE

Le Management de Projet nécessite de disposer d'un cadre de référence unique et commun entre les différents acteurs afin d'identifier toutes les tâches nécessaires et suffisantes pour en maîtriser la gestion, tant sur le plan technique, économique et administratif. Il constitue une méthode d'organisation qui nécessite un minimum d'outillage notamment dans le cadre du suivi du projet.

V.2.1 WBS (Work breakdown structure)

Le WBS aide à identifier et à définir l'ensemble des éléments à prendre en considération afin d'organiser un projet et pour pouvoir ensuite évaluer périodiquement son avancement. Chaque élément correspond à une tâche ou à un ensemble de tâches du projet. Le premier élément d'une WBS est le projet lui-même et donc, il reçoit le nom du projet. À partir de celui-ci, d'autres éléments sont créés en dessous pour représenter chaque élément du projet [16].La figure IV.1 montre la structure des tâches de notre projet.

V.2.2 Diagramme de Gantt

Le diagramme de Gantt est un outil utilisé en gestion de projet, est l'un des outils les plus efficaces pour représenter visuellement l'état d'avancement des différentes activités (tâches) qui constituent un projet. La colonne de gauche du diagramme énumère toutes les tâches à effectuer, tandis que la ligne d'en-tête représente les unités de temps les plus adaptées au projet [17]. (Voir figure V.2).

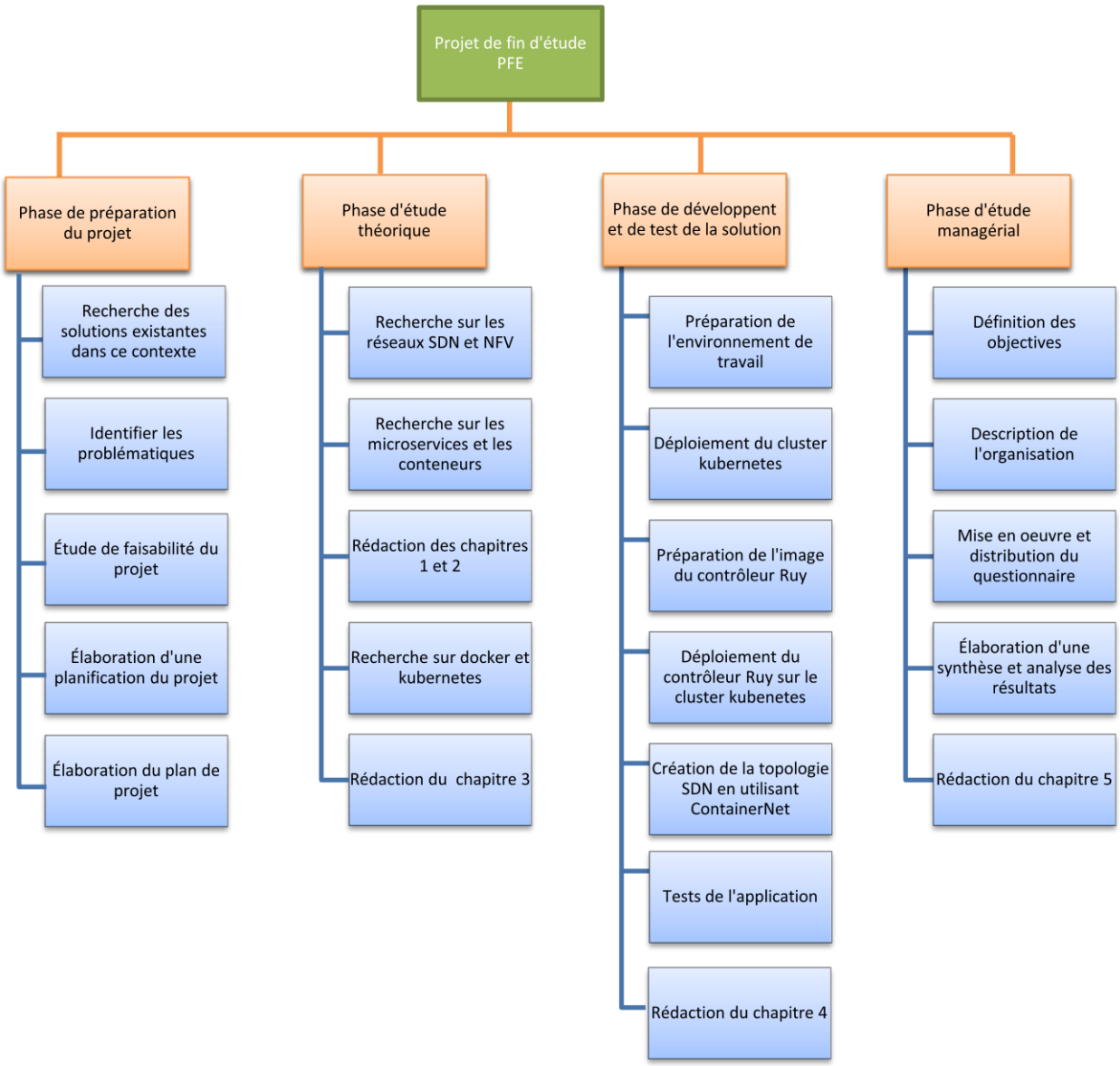

Figure .V.1 : Structure WBS.

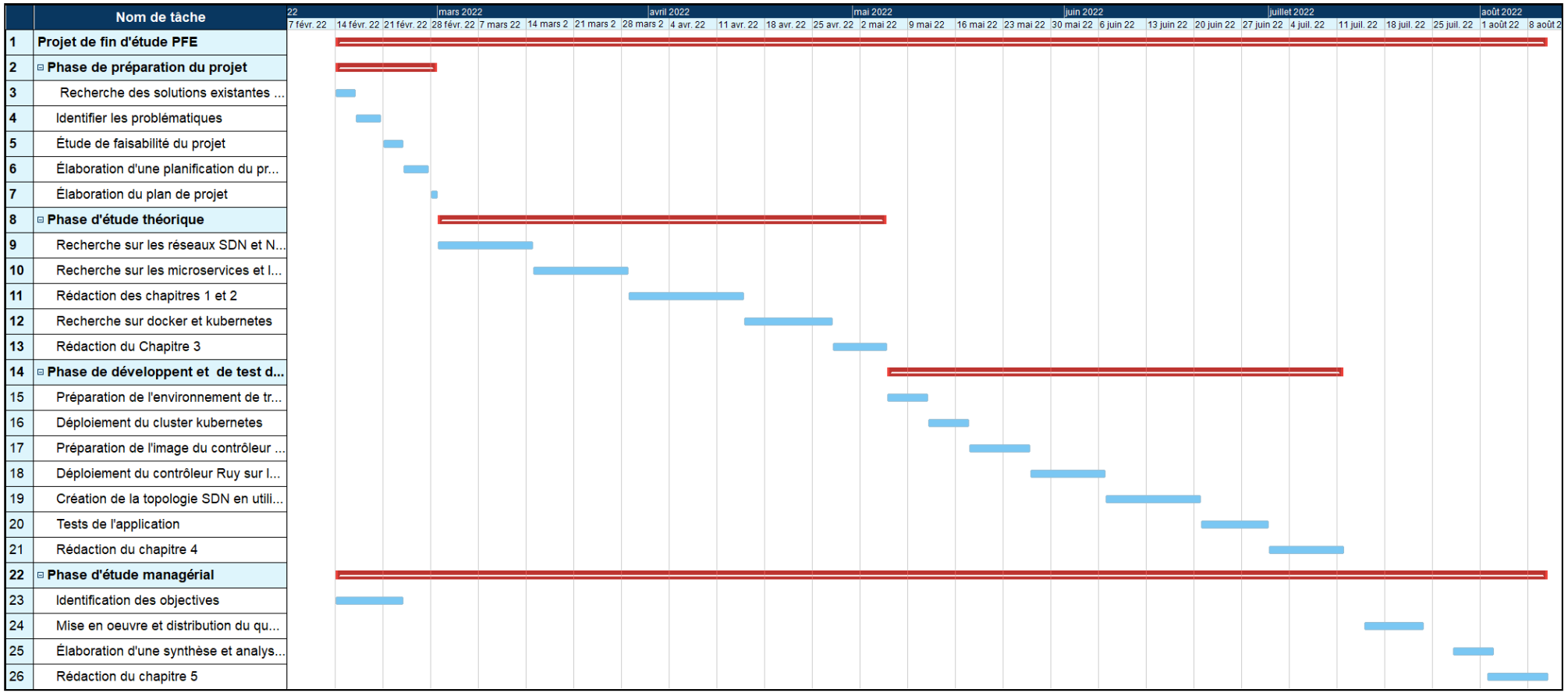

Figure .V.2 : Diagramme de Gantt

V.3. Impact de la solution sur l'entreprise

Pour voir l'impact de notre solution sur le management de l'entreprise, nous avons distribué un questionnaire à plusieurs personnes qui font partie des parties prenantes de notre solution :

- La partie externe : c'est la partie prenante la plus importante, il s'agit essentiellement des clients d'Ericsson comme les managers d'Ooredoo, de Djezzy et de Mobilis qui achètent la solution pour prendre part de son avantage.
- La partie interne : les ingénieurs, les administrateurs et les techniciens qui travaillent dans le département NMSD au sein de l'entreprise Ericsson, le responsable managérial du département NMSD, Directeur Général Ericsson d'Algérie.

Le questionnaire a été envoyé à leurs boîtes mails et les réponses reçues seront traitées dans les sections suivantes. Nous avons envoyé 39 questionnaires et n'en avons reçu que 21 en retour.

Nous avons calculé la tendance des avis pour chaque question. Les résultats obtenus sont récapitulés dans le tableau C.1 (voir annexe C). Le questionnaire comporte une grille de pondération d'échelle à quatre (04) niveaux de consentement, présentée ci-dessous, qui met en évidence l'impact de notre solution :

• Pas du tout d'accord (1)

• Plutôt en désaccord (2)

• D'accord (3)

• Tout à fait d'accord (4)

Formule de calcul des tendances des avis : ((a*1) + (b*2) + (c*3) + (d*4)) /N.

Sachant que : N= a + b + c + d = 21.

V.3.1. Interprétation des résultats

V.3.1.1. Organisation du travail

La plupart des parties prenantes estiment que cette solution aura un impact important pour l'entreprise et qu'il est nécessaire de l'appliquer le plus tôt possible car au nombre important des avantages et des bénéfices apportés comme la réduction des coûts, le gain du temps. Et vu que cela n'empêche pas le fonctionnement des autres solutions déjà déployées.

V.3.1.2. Moyens mis à la disposition pour réussir la mise en place de la solution

L'environnement de travail au niveau d'Ericsson permet l'intégration de notre solution à n'importe quel moment sans se soucier du facteur financier. Cette solution ne coûte rien à Ericsson qui n'a pas besoin d'aucun investissement, car les équipements nécessaires à sa mise en place existent déjà au niveau d'Ericsson, et les outils logiciels sont open source et gratuit.

V.3.1.3. Moyens humains (collègues, support en compétences)

La majorité des employés questionnés disent que l'implémentation de notre solution dans l'entreprise permet d'améliorer leurs compétences techniques et développent de nouvelles compétences suite à la mise en place de la solution, en revanche, certains se trouvent confrontés au changement qui nécessite un certain niveau de connaissance et d'expertise dans la technologie de conteneurisation, ainsi que la plateforme Kubernetes qui est un peu compliquée. Cependant, les avis favorables reçus émanaient de la parfaite conviction du personnel de la nécessité d'effectuer des formations continues dans ce domaine dans le but d'améliorer leurs compétences techniques et leur savoir-faire professionnel.

V.3.1.4. Gestion (encadrement et hiérarchie)

Cette solution permet d'automatiser et de faciliter la gestion du réseau NFV d'Ericsson et d'avoir une vue globale, ce qui conduit à mieux définir les tâches et les responsabilités. En effet, cette solution va permettre de diminuer l'intervention humaine dans la maintenance d'une infrastructure réseau due à l'impact de l'utilisation de l'outil Kubernetes.

V.3.1.5. Gestion du changement

Les ingénieurs réseaux d'Ericsson adhèrent pleinement à cette nouvelle initiative et ils sont aptes pour adapter leurs habitudes techniques pour comprendre les nouveaux aspects de la solution une fois implémentée, Excepté certains administrateurs et techniciens qui éprouvent quelques difficultés. Pour cela, nous suggérons qu'une formation à ces individus sera minimiser leur résistance au changement et éliminer leurs soucis vis-à-vis du fonctionnement de l'entreprise.

V.3.2. Synthèse globale

Suite aux réponses au questionnaire soumis aux parties prenantes, nous avons obtenu les informations suivantes :

- La majorité des avis sont positifs en ce qui concerne cette solution qui permet d'améliorer l'infrastructure NFV d'Ericsson et donc fournir des fonctions réseau (services) aux opérateurs mobile (clients) plus rentable non seulement en termes

de ressources, mais aussi une haute disponibilité et auto-résiliente en cas de défaillance de ces services.

- Ericsson dispose de tous les outils logiciels et les moyens matériels nécessaires pour le déploiement de cette solution.
- Les ingénieurs réseaux sont en faveur du changement, ça revient aux avantages fournis par notre solution. De même, les administrateurs et les techniciens acceptent le passage vers les microservices pour déployer des fonctions réseau conteneurisées car elle est consciente de l'intérêt et l'avantage que peut apporter cette solution, cependant, elle réclame des formations à son profit afin d'approfondir ses connaissances et de mieux s'adapter à la nouvelle solution.

Conclusion

Dans ce chapitre, nous avons abordé la partie managériale de notre projet. Nous avons d'abord présenté l'organisme d'accueil. Ensuite, nous avons présenté la planification de notre projet en utilisant des outils de management comme le diagramme de Gantt et le WBS. Enfin, après avoir répondu au questionnaire par les différentes parties prenantes, nous avons déduit l'impact managérial de notre solution sur l'entreprise.

Conclusion Générale

Avec l'augmentation des utilisateurs et la demande exponentielle des services, les infrastructures réseaux sont devenues très larges, complexes et difficiles à gérer. Pour cela, la nécessité de développer les réseaux traditionnels et de déployer de nouvelles architectures fondées sur le SDN et le NFV est devenue primordiale, car ils apportent d'énormes bénéfices aux fournisseurs de services et aux opérateurs de télécommunications.

La centralisation de plan de contrôle SDN dans une seule unité et l'adoption de plusieurs machines virtuelles volumineuses entravent l'efficacité des réseaux SDN et NFV pour les grands réseaux de 5G, de cloud qui exigent un haut niveau d'agilité, de disponibilité et d'évolutivité ainsi que des coûts d'exploitation réduits.

La migration vers les microservices et l'utilisation de conteneurs à la place de machines virtuelles afin de déployer et de gérer des fonctions réseaux conteneurisées peuvent résoudre les problèmes liés aux réseaux SDN et NFV, surtout avec l'outil d'orchestration Kubernetes qui est devenu rapidement la solution la plus populaire d'orchestration de ces fonctions conteneurisées sur un cluster de machines en utilisant des méthodes d'auto-résiliente, d'auto-scaling et de haute disponibilité.

C'est précisément l'objectif de notre projet de fin d'étude, au sein d'Ericsson, en proposant une solution SDN basé sur un contrôleur RYU déployé en tant qu'application conteneurisée à l'intérieur d'un cluster Kubernetes à l'aide de conteneurs docker et de l'orchestrateur Kubernetes.

Pour arriver à une solution qui satisfait nos exigences, nous avons en premier lieu parlé des réseaux SDN et NFV ainsi que leurs avantages et limites. En deuxième lieu, nous avons expliqué l'approche microservices et la technologie de conteneurisation en précisant l'avantage d'utiliser Kubernetes. Enfin, nous avons montré les avantages de la solution proposée par des tests de fonctionnement afin d'améliorer significativement les performances globales du réseau. Il s'agit d'optimiser l'utilisation des ressources existantes, de garantir la haute disponibilité, d'assurer la tolérance aux pannes automatique (auto-résiliente) et la mise à l'échelle automatique (auto-scaling) de la solution en cas de surcharge.

Au cours du développement de ce travail, certaines difficultés ont été rencontrées, en raison d'un problème dans l'analyse des performances pour déployer l'autoscaling, car ce dernier a besoin d'un outil qui fournit des métriques sur l'utilisation ressources. Nous avons surmonté ce problème en utilisant les deux serveurs intégrés dans minikube (add-ons) 'Heapster 'et 'Metrics server'.

En dernier lieu, nous avons abordé la partie managériale dans laquelle nous avons mis en évidence l'impact de notre solution sur le management de l'entreprise Ericsson en termes de réduction de coûts et d'utilisation efficace de ressources.

Enfin, nous espérons que notre travail servira d'appui pour les futurs étudiants afin d'apporter d'éventuelles améliorations. Dans cet objectif, nous proposons comme perspectives l'implémentation de notre solution dans un environnement Cloud sur OpenStack et de développer cette solution en utilisant l'apprentissage automatique.

Références

Bibliographiques

Références bibliographiques

[1] Ihssane Choukri, « Mohammed Ouzzif, Khalid Bouragba », Software Defined Networking (SDN): Etat de L'art, 27 Sep 2019

[2] Lobna DRIDI, « Mitigation des attaques de déni de service dans les réseaux définis par logiciel» mémoire en ligne, Université Du Québec, 11 septembre 2017.

[3] Djamel Eddine HENNI, « Gestion de la qualité de service des flux multimédia dans les réseaux SDN» Thèse doctorat. Université Oran 1, Algérie. 2019/2020

[4] Hind ERRAHMOUNI, « Modélisation des chaines de services hautement disponibles » mémoire en ligne, Université Du Québec, 11 décembre 2020.

[5] « Documentation propre à Ericsson ». EPG 16A ON SSR Product Documentation, Version : V16.3. Ericsson AB 2017.

Webographies

[4] Comprendre la virtualisation [Enligne].
Adresse URL : https://www.redhat.com/fr/topics/virtualization
Consulté le : 20 février 2022

[6] Network Slicing [Enligne].
Adresse URL : https://whatis.techtarget.com/definition/network-slicing
Consulté le : 2 Mars 2022

[7] Lorsque l'architecture monolithique devient trop imposante, il est peut-être temps de passer aux microservices [Enligne].
Adresse URL : https://www.atlassian.com/fr/microservices/microservices-architecture/microservices-vsmonolith?fbclid=IwAR0LhWPUzuRfO6xVXyUzHXICsdrsckvaua_913AXG82cDU3ifxEbMtUbg_E
Consulté le : 19 Mars 2022

[8] What is a CNF? [Enligne].
Adresse URL : https://ligato.io/cnf/cnf-def/, Consulté le : 10 Avril 2022.

[9] Kubernetes vs d'Autres Plateformes d'Orchestration de Conteneurs [Enligne].
Adresse URL :
https://medium.com/@rewelle/kubernetes-vs-dautres-plateformes-d-orchestration-de-conteneurs-233a1568e571, Consulté le : 10 Avril 2022.

[10] Apprendre Docker [Enligne].
Adresse URL : https://devopssec.fr/article/decouverte-et-installation-de-docker#begin-article-section
Consulté le : 5 Mai 2022.

[11] Docker [Enligne].

Adresse URL : https://www.ibm.com/fr-fr/cloud/learn/docker

Consulté le : 14 Mai 2022.

[12] Objets et architecture Kubernetes [Enligne].

Adresse URL : https://containers.goffinet.org/content/k8s-01-introduction-a-kubernetes/02-objets-architectures-kubernetes.html

Consulté le : 20 Mai 2022.

[14] What is Kubernetes? [Enligne].

Adresse URL : https://www.vmware.com/topics/glossary/content/kubernetes.html

Consulté le : 12 juin 2022.

[15] Ericsson - Eric Stoupel[Enligne].

Adresse URL : https://fr.wiki5.ru/wiki/Ericsson

Consulté le : 1 août 2022.

[16] WBS (Work Breakdown Structure) [Enligne].

Adresse URL : http://www.gestiondeprojet.wabyl.com/wbs.html

Consulté le : 2 août 2022.

[17] Qu'est-ce qu'un diagramme de Gantt ? [Enligne].

Adresse URL : https://www.gantt.com/fr/

Consulté le : 2 août 2022.

Annexes

A.1. Messages OpenFlow

Les messages OpenFlow décrit comment les commutateurs OpenFlow communiquent avec les contrôleurs. Les messages sont ont transmis via un canal sécurisé, implanté via une connexion TLS sur TCP. Il existe trois types de messages : contrôleur-commutateur, asynchrone et symétrique.

1) Messages contrôleur-commutateur

Les messages contrôleur-commutateur sont lancés par le contrôleur et utilisés pour gérer ou inspecter directement l'état du commutateur. Le contrôleur est capable interroger un Switch ou de modifier sa configuration (Table de Flux) avec les messages suivants :

- **Features :** Le contrôleur peut transmettre un message Feature Request pour obtenir les capacités et fonctionnalités supportées par un commutateur. Ce dernier répond par un Feature Reply.
- **Configuration:** Permet de changer ou d'obtenir la configuration d'un commutateur.
- **Modify-State:** Le contrôleur envoie des messages de modification d'état pour gérer l'état sur les commutateurs. Leur objectif principal est d'ajouter, de supprimer et de modifier les entrées de flux dans les tables OpenFlow et de définir les propriétés de ports de commutation.
- **Read State:** Permet de demander les statistiques du commutateur
- **Send-Packet:** Demande au commutateur de transmettre le paquet sur un port donné

2) Messages asynchrones

Ce type de message est initié lorsqu'un événement réseau est détecté. Les messages suivants tombent dans cette catégorie :

Packet-In: Lorsqu'un paquet est reçu par un commutateur qui ne correspond à aucune entrée de la Table de Flux ou bien l'action correspondante est égale à CONTROLLER, un Packet-In est généré et transmis vers le Contrôleur. Avant de transmettre le Packet-In au Contrôleur, le paquet initial (celui pour lequelle aucune entrée de la Table de Flux ne correspond) est placé dans le buffer et un identifiant est affecté à celui-ci. Le Packet-In transmis au Contrôleur contiendra l'identifiant buffer et une copie de l'entête du paquet. Lorsque le Contrôleur renvoie une réponse Send-Packet, celui-ci comprendra l'identifiant buffer.

Flow-Removal: Si une entrée de la Table de Flux est supprimée lorsque aucun paquet entrant n'a de correspondance avec cette entrée pendant un temporisateur (Timeout) spécifié par

le contrôleur lors de la création de cette entrée au niveau de la table de flux du commutateur, un message Flow-Removal est transmis au Contrôleur.

3) Messages symétriques

Les messages symétriques peuvent être initiés par le Contrôleur ou le commutateur.

HELLO: Ce type de message est transmis par le canal sécurisé entre le commutateur et le contrôleur a été établi.

ECHO: Ce type de message est utilisé comme les messages Echo request / reply ICMP pour s'assurer que la connexion est toujours en vie et afin de mesurer la latence et le débit courants de la connexion.

A.2 DevOps

DevOps est une approche qui permet de réunir et de faire collaborer les équipes de développement et d'exploitation dans le but de pouvoir créer et livrer des produits plus performants et plus fiables pour mieux répondre aux besoins du client. Cette collaboration et cette productivité améliorées permettent d'atteindre des objectifs commerciaux tels que la publication plus fréquente ou la mise à disposition plus rapide de versions, de fonctionnalités ou de mises à jour des produits, le tout en assurant les niveaux de qualité et de sécurité appropriés. Autre objectif: améliorer les délais de détection, de résolution de bogues ou d'autres problèmes et de republication d'une version, et aussi :

- ✓ Raccourcir le délai de commercialisation
- ✓ S'adapter au marché et à la concurrence
- ✓ Améliorer le temps de récupération

A.2.1 Développement continu (continuous development) :

Le développement continu veut dire qu'au lieu de faire les mises à jour pour le logiciel en un seul lot, les mises à jour seront effectuées en continu, bloc par bloc, permettant au code logiciel d'être livré aux clients dès qu'il est terminé et testé.

A.2.2 Déploiement continue (continuous deployment) :

Le processus de déploiement est effectué de telle manière que les modifications apportées au code ne devraient pas affecter le fonctionnement de l'application à fort trafic.

A.3 La relation entre microservices et DevOps :

L'architecture micro services et l'approche DevOps présentent des pratiques qui offrent une plus grande agilité et une efficacité opérationnelle pour l'entreprise, Elles rendent

l'entreprise plus productive que leurs concurrents, ce qui signifie qu'elles peuvent innover plus rapidement à moindre coût.

La combinaison entre microservices et DevOps a été utilisée pour la première fois par les grandes entreprises telles qu'Amazon, Netflix, Google ...etc.

Cette combinaison permet d'accélérer l'innovation, réduire les erreurs, améliorer la qualité des produits, augmenter la valeur commerciale, réduisez les coûts de développement logiciel, et rendre les équipes DevOps plus productives.

Un mélange des deux méthodologies tel qu'agile et DevOps peut être utilisé pour assurer une efficacité accrue. Les deux ont un rôle majeur à jouer en matière de développement et de déploiement de logiciels, et l'un peut être utilisé pour activer l'autre.

A.4 Cycle de vie d'un conteneur

La figure suivante explique le cycle de vie complet d'un conteneur :

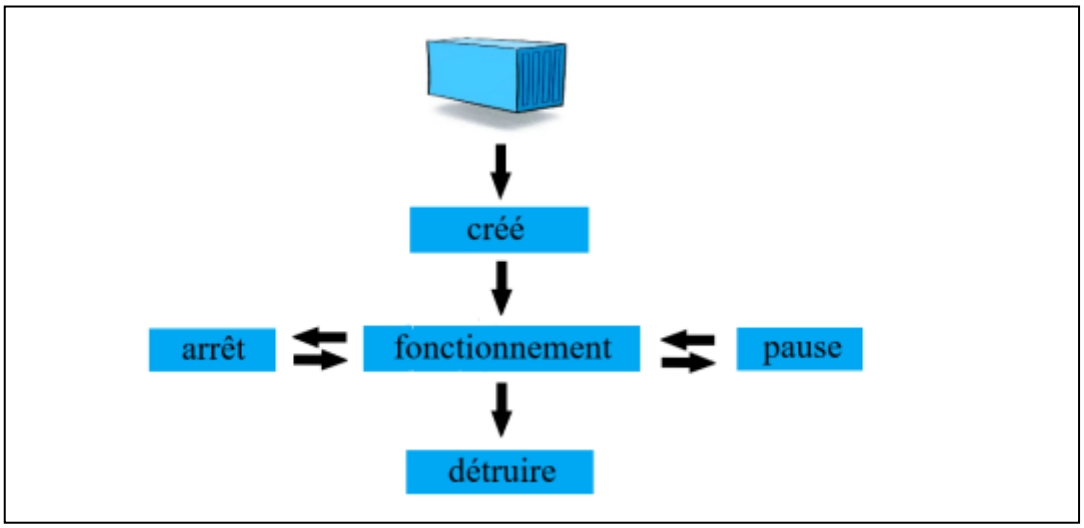

Figure A.1. Cycle de vie d'un conteneur

Initialement, le conteneur sera à l'état création, Ensuite, le conteneur Docker passe à l'état de fonctionnement lorsque la commande d'exécution est utilisée. On passe à l'état de détruire quand ont utilisé la fonction Kill sur un fonctionnel conteneur pour le fermer (tuer). On peut passer le conteneur vers l'état pause (stopper le conteneur pendant une courte période puis le démarrer) à travers la commande pause, comme on peut aussi l'arrêter (stopper le conteneur avec la possibilité de le redémarrer ou pas) avec l'utilisation de la commande stop.

B.1 Instalation de Docker :

Nous avons utilisé la commande suivante pour installer Docker :

```
assia@ubuntu:~$ sudo apt install docker-ce
Lecture des listes de paquets... Fait
Construction de l'arbre des dépendances
Lecture des informations d'état... Fait
Les paquets supplémentaires suivants seront installés :
  containerd.io docker-ce-cli docker-ce-rootless-extras docker-scan-plugin pigz slirp4netns
Paquets suggérés :
  aufs-tools cgroupfs-mount | cgroup-lite
Les NOUVEAUX paquets suivants seront installés :
  containerd.io docker-ce docker-ce-cli docker-ce-rootless-extras docker-scan-plugin pigz slirp4netns
0 mis à jour, 7 nouvellement installés, 0 à enlever et 4 non mis à jour.
Il est nécessaire de prendre 3 652 ko/102 Mo dans les archives.
Après cette opération, 422 Mo d'espace disque supplémentaires seront utilisés.
Souhaitez-vous continuer ? [O/n] o
```

Figure B.1. Installation Docker

Une fois l'installation terminée, nous allons vérifier l'état du Docker avec la commande suivante :

```
assia@ubuntu:~$ systemctl status docker
● docker.service - Docker Application Container Engine
     Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
     Active: active (running) since Sat 2022-08-27 20:21:30 CET; 2min 3s ago
TriggeredBy: ● docker.socket
       Docs: https://docs.docker.com
   Main PID: 1336 (dockerd)
      Tasks: 10
     Memory: 108.0M
     CGroup: /system.slice/docker.service
             └─1336 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
```

Figure B.2. Vérifier l'état de Docker

La figure B.2 montre que l'installation a réussi et que Docker est actif et en cours d'exécution.

B.2 Installation de Minikube :

D'abord, nous allons installer VirtualBox sur Ubuntu en exécutant la commande suivante :

```
assia@ubuntu:~$ sudo apt install virtualbox virtualbox-ext-pack
Lecture des listes de paquets... Fait
Construction de l'arbre des dépendances
Lecture des informations d'état... Fait
Les paquets suivants ont été installés automatiquement et ne sont plus nécessaires :
  libfprint-2-tod1 libllvm9 linux-headers-5.13.0-30-generic linux-hwe-5.13-headers-5.13.0-30 linux-image-5.13.0-30-generic
  linux-modules-5.13.0-30-generic linux-modules-extra-5.13.0-30-generic
Veuillez utiliser « sudo apt autoremove » pour les supprimer.
Les paquets supplémentaires suivants seront installés :
  virtualbox-qt
Paquets suggérés :
  vde2 virtualbox-guest-additions-iso
Les NOUVEAUX paquets suivants seront installés :
  virtualbox virtualbox-ext-pack virtualbox-qt
0 mis à jour, 3 nouvellement installés, 0 à enlever et 4 non mis à jour.
Il est nécessaire de prendre 43,3 Mo dans les archives.
Après cette opération, 172 Mo d'espace disque supplémentaires seront utilisés.
Souhaitez-vous continuer ? [O/n] o
```

Figure B.3. Installation VirtualBox

Une fois VirtualBox installé, nous passons à l'installation de Minikube en utilisant la commande suivante :

```
assia@ubuntu:~$ wget https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
--2022-08-27 19:09:36--  https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
Résolution de storage.googleapis.com (storage.googleapis.com)… 216.58.215.48, 216.58.213.80, 142.250.179.80, ...
Connexion à storage.googleapis.com (storage.googleapis.com)|216.58.215.48|:443… connecté.
requête HTTP transmise, en attente de la réponse… 200 OK
Taille : 75566620 (72M) [application/octet-stream]
Enregistre : «minikube-linux-amd64»

minikube-linux-amd64                100%[=====================================================================>]  72,07M  1,00MB/s    ds 61s

2022-08-27 19:10:38 (1,17 MB/s) - «minikube-linux-amd64» enregistré [75566620/75566620]
```

Figure B.4. Installation Minikube

Enfin, nous allons vérifier que Minikube est bien installé en vérifiant la version du logiciel :

```
assia@ubuntu:~$ minikube version
minikube version: v1.25.2
commit: 362d5fdc0a3dbee389b3d3f1034e8023e72bd3a7
assia@ubuntu:~$
```

Figure B.5. Version de Minikube

B.3 Développement du module Network Slicing :

Lorsqu'un évènement OpenFlow (PacketIn) est détecté par le contrôleur RYU, ce dernier va déclencher le module « Network Slicing »afin de découper le réseau en sous-réseaux logiques isolés, appelés "tranches" ou "slices". Chaque tranche de réseau fournie une fonction réseau dédiée (Voir figure B.6).

```python
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath
        dpid = datapath.id
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        in_port = msg.match['in_port']

            if eth.dst in mac_to_port_table:

                src=eth.src
                dst=eth.dst
                protocol = ip_pkt.proto
                out_port = mac_to_port_table[eth.dst]
                actions = [parser.OFPActionOutput(out_port)]

                    if protocol == in_proto.IPPROTO_TCP:
                        dst_port = _tcp.dst_port
                        src_port = _tcp.src_port
                        for net_slice in slices_data4:
                            slice_id = net_slice[0]     # extraire le slice ID de src et dst
                            net_slice = net_slice[1:]

                            if src in net_slice and dst in net_slice and (dst_port in net_slice or src_port in net_slice):
                                self.logger.info(f"dpid {dpid} in eth {in_port} out eth {out_port}")
                                self.logger.info(f"Slice pair [{src},{dst}] in slice {slice_id} protocol {protocol} dst_port {dst_port}")
                                match = parser.OFPMatch(in_port=in_port, eth_dst=eth.dst, eth_src=eth.src,eth_type=ether_types.ETH_TYPE_IP,
                                 ip_proto=protocol,tcp_src= src_port, tcp_dst= dst_port )
                                self.add_flow(datapath, 0, 5, match, actions)
                                self.send_packet_out(datapath, msg.buffer_id, in_port,
                                                     out_port, msg.data)
                        else: # src MAC et dst Mac ne sont pas dans le même Slice,alors sautez vers le prochain slice
                            return
```

Figure B.6. Script Network Slicing à base de protocol TCP.

Annexe C : Questionnaire de la partie managériale

Le tableau suivant présente la liste des questions qu'on a posées sur les 39 personnes qui font partie des parties prenantes de notre solution :

1. Organisation de travail					
Question	Réponses				Tendance des avis
	1	2	3	4	
1- Je vois un grand intérêt à mettre en place cette solution	2	1	7	11	3.28
2- De nouveaux procédés de travail peuvent être développés par cette solution technique	1	2	11	7	3.14
3- Je peux utiliser cette solution sans être dérangé (pas d'interférence, pas de conflit de priorité avec d'autres technologies).	0	1	7	13	3.57
4- Cette solution proposée prendra du temps pour être mise en œuvre d'une façon officielle au niveau de l'entreprise	2	2	9	8	3.09
5- Cette solution technique a permis d'améliorer notre mode de fonctionnement	1	2	4	14	3.47
2. Les moyens mis à la disposition pour réussir la mise en place de la solution					
Question	Réponses				Tendance des avis
	1	2	3	4	
6- J'ai à ma disposition tous les outils de travail nécessaires à la mise en place de cette solution technique	1	2	9	9	3.23
7- J'ai toute la documentation nécessaire pour concrétiser cette solution technique.	0	3	10	8	3.23
8- L'aménagement de mon environnement de travail est favorable à cette solution.	2	1	7	11	3.28
9- Cette solution peut être compatible avec l'infrastructure existante	0	1	9	11	3.47
3. Moyens humains (collègues, support en compétence)					
Question	Réponses				Tendance

	1	2	3	4	des avis
10- Je pourrais avoir le support nécessaire pour accomplir cette solution	0	2	7	12	3.47
11- Cette solution me permet d'améliorer mes compétences techniques.	1	1	8	11	3.38
12- De nouvelles compétences sont développées au sein de notre structure suite à la mise en place de cette solution.	2	3	8	8	3.04
13- Est-il nécessaire de programmer une formation en tout ce qui concerne le fonctionnement et l'utilisation de la solution technique.	2	3	7	9	3.09
4. Gestion (encadrement et hiérarchie)					
Question	Réponses				Tendance des avis
	1	2	3	4	
14- Mes comportements et mes résultats au travail pourront être améliorés	0	2	10	9	3.33
15- Mes tâches et responsabilités sont mieux définies par les responsables dans le cadre de cette solution technique.	1	1	12	7	3.19
16- Notre travail est de meilleure qualité.	1	1	10	9	3.28
5. Gestion du changement					
Question	Réponses				Tendance des avis
	1	2	3	4	
17- On m'a expliqué pourquoi ils ont mis en place cette solution	1	1	7	12	3.42
18- Vous acceptez la mise en place de cette solution dans l'entreprise.	0	0	9	12	3.57
19- Vous acceptez de changer vos habitudes de travail après la mise en place de la solution.	3	3	8	7	2.9

Tableau C.1. Questionnaire de la partie managériale

.

.

Résumé

Le réseau défini par logiciel est une nouvelle architecture de réseau. Ce type de réseau est d'une grande importance, car il permet une gestion centralisée du réseau, mais la centralisation complète des fonctions réseau soulève des problèmes potentiels liés à la disponibilité, à la tolérance aux pannes et à l'évolutivité.

Ce travail vise à implémenter le contrôleur RYU SDN en tant que une application conteneurisée à l'intérieur d'un cluster Kubernetes en utilisant l'approche microservices pour créer plusieurs répliques de contrôleur RUY ainsi améliorer les performances globales du réseau. Il s'agit d'optimiser l'utilisation des ressources existantes, de garantir la haute disponibilité, d'assurer l'auto-résiliente et l'évolutivité automatique de réseau SDN.

Mots-Clés : SDN, NFV, OpenFlow, RYU, Kubernetes, Docker.

ملخص

الشبكات المعرفة بالبرمجيات هي بنية شبكة جديدة. هذا النوع من الشبكات له أهمية كبيرة لأنه يسمح بإدارة مركزية للشبكة، لكن المركزية الكاملة لوظائف الشبكة تثير مشاكل محتملة تتعلق بتوفر واستمرار خدمة مع الأعطال وقابلية التوسع.

يهدف هذا العمل إلى تنفيذ وحدة تحكم RYU SDN كوظيفة شبكة حاويات داخل مجموعة Kubernetes باستخدام نهج الخدمات المصغرة لإنشاء نسخ متماثلة متعددة لوحدة تحكم RUY وبالتالي تحسين الأداء العام للشبكة. هذا لتحسين استخدام الموارد الحالية، لضمان التوفر العالي، لضمان المرونة الذاتية والقابلية للتوسع التلقائي لشبكة SDN.

.

Abstract

Software defined networking is a new network architecture. This type of network is of great importance as it allows centralized network management, but the full centralization of network functions raises potential issues related to availability, fault tolerance and scalability.

This work focuses on implementing the SDN RYU controller as a containerised network function inside a Kubernetes cluster using the microservices approach to create multiple RUY controller replicas thus improving overall network performance. This includes optimising the use of existing resources, ensuring high availability, self-healing and automatic SDN scalability.

Keywords: SDN, NFV, OpenFlow, RYU, Kubernetes, Docker.