الجمهورية الجزائرية الديمقراطية الشعبية
Populaire et Démocratique Algérienne République
وزارة التعليم العالي والبحث العلمي
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Département : Génie électrique Et Informatique Industrielle

Mémoire de fin d'étude pour l'obtention du diplôme d'Ingénieur d'État en Télécommunications

Spécialité : Système de Télécommunications et Réseaux

- Thème -

Conception et implémentation d'une solution SDN

Solution d'équilibrage de charge en SDN

Réalisé par :

M. Bentegar Sid Ahmed Abdelillah

M. Halouane Alla Eddine

Les membres de Jury :

Mme. BENDOUDA Djamila M.C.B	Présidente
M. KHOUMERI El Hadi M.C.B	Promoteur
Mme. ABBAD Leila M.A.A	Examinateur
M. BOUCHACHI Islam M.C.B	Examinateur

Alger, le 26/06/2022

Année universitaire : 2021 –2022

Remerciements

Tout d’abord, je remercie Allah le tout puissant de m’avoir donné le courage et la patience nécessaires pour mener ce travail à son terme.

Et en second lieu , on remercie nos parents pour leurs efforts pour rester confortable et heureux et pour obtenir une capacité suffisant de compléter ce travail,troisièmement on remercie nos frères de leurs encouragements toute l’année et tout le temps.

Et finalement on remercie notre promoteur M. KHOUMERI El Hadi pour leur disponibilité et on remercie aussi tous les professeurs de cette promotion et de tous les années passées pour nous donner toutes les informations dans nos domaines .

Table des matières

Remerciements **2**

Introduction générale **10**

1 Software Defined Network (SDN) **14**
1.1 Introduction . . . 14
1.2 Software Defined Network . . . 14
1.2.1 Définition . . . 14
1.2.2 Architecture . . . 14
1.2.3 Avantages . . . 18
1.2.4 Application de SDN . . . 19
1.3 Open Flow Protocol . . . 20
1.3.1 Vue Général sur OpenFlow . . . 20
1.3.2 Le Protocol OpenFlow . . . 21
1.3.3 L'évolution du protocole OpenFlow . . . 23
1.3.4 Limitations de OpenFlow . . . 28
1.4 Équilibrage de charge . . . 28
1.4.1 Les avantages de l'équilibreur de charge . . . 28
1.4.2 Types d'équilibrage de charge . . . 29
1.4.3 Les algorithmes d'équilibrage de charge . . . 29
1.4.4 Equilibreur de charge avec SDN . . . 31
1.5 Conclusion . . . 32

2 Logiciels et matériels **33**
2.1 Introduction . . . 34
2.2 Logiciels . . . 34
2.2.1 Émulateur . . . 34
2.2.2 Langages de programmation . . . 35
2.2.3 Bibliothèque et Framework . . . 36
2.2.4 Logiciel et Service . . . 37
2.2.5 SQLite . . . 39
2.3 Materiels . . . 39
2.3.1 Developement Local . . . 39
2.3.2 Serveur d'émulation . . . 39

3 Méthodologie **41**
3.1 Introduction . . . 41
3.2 Présentation du concept générale . . . 41

3.2.1 Réalisation d'une Preuve de concept . . . 41
3.2.2 Conclusion . . . 52
3.3 Amélioration d'équilibrage de charge . . . 52
3.3.1 Introduction . . . 52
3.3.2 Les méthodes de sélection du serveur . . . 54
3.3.3 Encapsulation de la méthodologie . . . 56
3.4 L'implémentation d'un North Bound API . . . 59
3.4.1 Implémentation d'un serveur HTTP[1] : . . . 60
3.4.2 Solution du problem CORS[2] . . . 60
3.4.3 Définition des routes . . . 61
3.5 Le développement d'une interface graphique . . . 62
3.5.1 Initiation du projet . . . 63
3.5.2 Visualisation de Topologies . . . 64
3.5.3 Gérance des équilibreurs de charge et serveurs . . . 65
3.5.4 Affichage des Flow . . . 66
3.6 Intégration de Docker Et CI[3]/CD[4] . . . 66
3.6.1 Docker containers . . . 66
3.6.2 Implémentation d'un CI/CD . . . 67

4 Implémentation et les Tests 70
4.1 Introduction . . . 70
4.2 Préparation des outils de test . . . 70
4.3 Préparation du Topologie . . . 70
4.4 Tests . . . 71
4.4.1 Test Initial : . . . 71
4.4.2 Test d'équilibrage de charge . . . 73
4.5 Conclusion . . . 77

5 Conclusion et travaux futurs 79
5.1 Conclusion . . . 79
5.2 Travaux futurs . . . 79

[1] *Hypertext Transfer Protocol*
[2] *Cross-Origin Resource Sharing*
[3] *Continuous integration*
[4] *continuous delivery*

Table des figures

1 Diagramme de Gantt du notre projet 11

1.1 L'architecture SDN 15
1.2 Séparation plan de contrôle au commutateur[46] 15
1.3 Fonctionnalité de commutateur Open Flow [4]. 22
1.4 Entré dans la table des Flow [6] 23
1.5 Tableau des Flow multiples [44] 25
1.6 OpenFlow Group [43] 27
1.7 Mécanisme de l'équilibreur de charge avec SDN 31

2.1 Configuration du serveur Azure 40

3.1 Réseau dans GNS3 48
3.2 Capture des requête ARP[5] avec wireshark 49
3.3 Capture des requête TCP entre le commutateur et l'hôte 1 50
3.4 Capture des requête TCP entre le commutateur et le serveur 1 50
3.5 Diagramme UML de la base des données 53
3.6 Diagramme d'activité d'une requête ARP 58
3.7 Diagramme d'activité du requête TCP 59
3.8 Visualisation du topolo 64
3.9 Visualisation du topologie dans l'interface Web 65
3.10 Affichage de la table des Flow d'un commutateur avec l'interface Web 66

4.1 Taux de latence 71
4.2 Taux de succès 71
4.3 Taux de latence 72
4.4 Taux de succès 72
4.5 Taux de latence 73
4.6 Taux de succès 74
4.7 Taux de latence des requêtes ICMP[6] 74
4.8 Taux de latence Client 1 75
4.9 Taux de latence Client 2 75
4.10 Taux de succès Client 1 75
4.11 Taux de succès Client 2 75
4.12 Taux de latence Client 1 76
4.13 Taux de latence Client 2 76
4.14 Taux de succès Client 1 76

[5] *Address Resolution Protocol*
[6] *Internet Control Message Protocol*

4.15 Taux de succès Client 2 76
4.16 Taux de latence Client 1 77
4.17 Taux de réponse Client 2 77
4.18 Taux de latence Client 1 77
4.19 Taux de succès Client 2 77

Liste des tableaux

1.1 Messages entre le contrôleur et le commutateur [46] 16
1.2 Messages échangés entre le contrôleur et le commutateur [51] 26

Liste des abréviations

API	*Application Programming Interface*
ARP	*Address Resolution Protocol*
ASIC	*Application Specific Integrated circuit*
CI	*Continuous integration*
CD	*continuous delivery*
CORS	*Cross-Origin Resource Sharing*
CPU	*Central Processing Unit*
DPID	*Depository Participant Identity*
FIFO	*First In, First Out*
GNS3	*Graphical Network Simulator-3*
HTTP	*Hypertext Transfer Protocol*
IaaS	*Infrastructure As A Service*
ICMP	*Internet Control Message Protocol*
IP	*Internet Protocol*
IPv4	*Internet Protocol version 4*
IPv6	*Internet Protocol version 6*
ISV	*Independent Software Vendor*
JSON	*JavaScript Object Notation*
KVM	*Kernel-based Virtual Machine*
LAN	*Local Area Network*
MPLS	*Multiprotocol Label Switching*
ONF	*Open Network Foundation*

OVSDB *Open vSwitch Database Management Protocol*

OXM *OpenFlowExtensible Match*

PaaS *Platform As A Service*

REST *Representational State Transfer*

SaaS *Software As A Service*

SDN *Softwate Defined Network*

SNMP *Simple Network Management Protocol*

SPA *Single Page Application*

SSH *Secure Shell*

SSL *Secure Socket Layer*

SVG *Scalable Vector Graphics*

TLS *Sécurité de la couche de transport*

TCP *Transmission Control Protocol*

UDP *User Datagram Protocol*

URL *Uniform Resource Locator*

VLAN *Virtual Local Area Network*

VPS *Virtual Private Server*

VPN *Virtual Private Network*

WSGI *Web Server Gateway Interface*

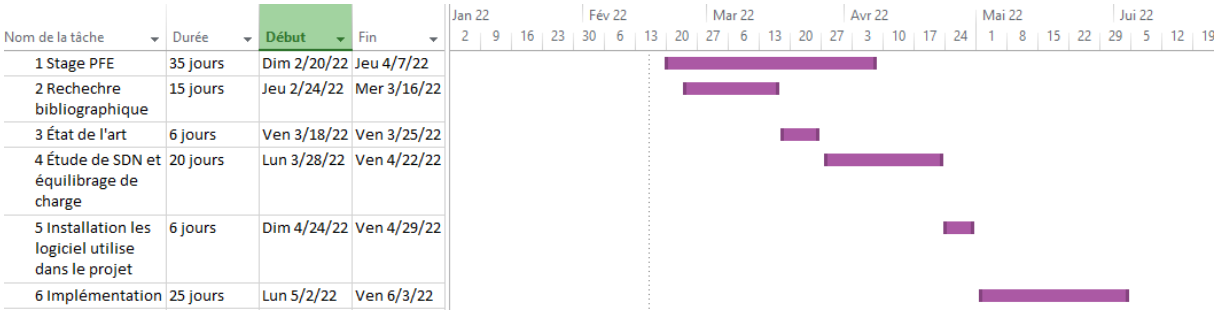
Introduction générale

Dans les entreprises, le réseau est devenu très important dans la vie professionnelle, car il facilite grandement la conduite de certains travaux grâce aux applications qui fonctionnent grâce à l'infrastructure réseau.
Cette structure se compose des équipements qui fonctionnent à leur tour pour communiquer et envoyer des paquets entre ordinateurs et serveurs, et comme nous le savons, les ordinateurs se développent de manière remarquable, il en va de même les infrastructures réseau pour traiter les énormes quantités de données générées par les ordinateurs.
Et quand on regarde notre réseau aujourd'hui, nous voyons qu'il a encore des performances mauvaises, manque de développement et est sujet à l'échec. C'est pourquoi la programmation réseau est apparue,notamment "Software-defined networking(SDN)", qui a su se faire remarquer dans le domaine des réseaux grâce à ses avantages et sa solution à un grand nombre de problèmes .
Et quand on parle de ses applications, l'équilibreur de charge est l'un des plus importants, ce qui contribue grandement à augmenter la qualité et la vitesse du réseau.

L'objectif principal de notre projet est de créer une solution équilibrage de charge implémentée dans le réseau SDN, et de donner à l'administrateur la liberté de gestion le réseau de manière dynamique et simple.

Diagramme de Gantt

Un diagramme de Gantt est un outil (souvent utilisé avec les réseaux PERT) de planification et de gestion de projet et permet de visualiser les différentes tâches qui composent un projet dans le temps[14]. Il s'agit d'une représentation graphique connectée, précieuse et dirigée qui peut représenter graphiquement l'avancement d'un projet[14].Les différéntes étape de notre projet dans un ordre chronologique présente à l'aide d'un digramme de Gantt est illustré dans la figure 1 .



Nom de la tâche	Durée	Début	Fin
1 Stage PFE	35 jours	Dim 2/20/22	Jeu 4/7/22
2 Rechechre bibliographique	15 jours	Jeu 2/24/22	Mer 3/16/22
3 État de l'art	6 jours	Ven 3/18/22	Ven 3/25/22
4 Étude de SDN et équilibrage de charge	20 jours	Lun 3/28/22	Ven 4/22/22
5 Installation les logiciel utilise dans le projet	6 jours	Dim 4/24/22	Ven 4/29/22
6 Implémentation	25 jours	Lun 5/2/22	Ven 6/3/22

Fig. 1 : Diagramme de Gantt du notre projet

Problématique

Notre idée a été inspirée par notre stage chez le département réseaux core chez Algerie Telecom Annaba. Au sein de cette entreprise, ils ont commencé le déploiement de la nouvelle technologie SDN. Cette transition est encore à ses débuts et se déroule à huis clos dans le cadre d'un contrat avec Juniper Networks.

Après avoir inspecté le réseau central d'Algerie Telecom nous avons trouvé là des implémentations de serveurs de cache de bord très intéressantes. Sur le bord du réseau national MPLS[7] ils utilisent des serveurs de cache fournis par Facebook et Google pour mettre en cache des données et essayer de limiter le trafic international aux services fournis par Facebook (Actuellement Méta) et Google. Ces deux derniers fournissent 4 serveurs de cache avec un niveau de préformation variable qui a été ajouté en tant qu'assurance de l'allocation des ressources la plus efficace. Le contexte actuel pousse à faire évoluer les réseaux car l'architecture des réseaux traditionnels commence à être saturé et ne satisfait réellement pas les nouvelles demandes avec l'arrivée des nouvelles technologies et l'évolution des besoins des clients. Avec toutes les limitations existantes sur le réseau traditionnel, les recherches ont menées à une nouvelle technique : Software Defined Networking (SDN).

Etude de l'Existance

Il existe plusieurs solutions pour l'équilibrage de charge, mais pour nous ils ne sont pas adaptés à la problématique qu'on avait, avoir un équilibreur de charge intégré dans la solution SDN qui fonctionne comme une partie de la solution. Les équilibreurs de charge les plus utilisés dans le domaine de IT ayant les problèmes suivants :

- **Liés à un fournisseur de cloud computing :** La solution est liée à un fournisseur de cloud computing. Cette solution doit être utilisée avec sa plateforme est ses services. Des exemples sur ces services sont Azure Application GateWay and AWS elastic.
- **Sont des services séparés :** Ça implique que la solution proposée pour l'équilibrage de charge doit être exécutée et gérée séparément de la solution existante. Les développeurs de ces solutions proposent le déploiement de ces solutions soit dans des nœuds dédiés. À notre avis, notre solution devrait faire partie de la solution SDN, et la fonction d'équilibrage de charge dans le contrôleur et les commutateurs devrait être incluse séparément dans un service isolé.

[7] *Multiprotocol Label Switching*

- **Coûteux** La plupart des solutions d'équilibrage de charge offrent aux marchés sont des solutions à prix très élevé avec un minimum de 2500$. D'une part nous savons que ce sont des solutions commerciales destinée aux entreprises, de l'autre part nous croyons en l'open source, nous croyons en vision de l'SDN au "Openness". c'est la vision adapter par plusieurs entreprises dans le domaine IT comme RedHat, Mysql, Apatche, Cloudbe.

L'état de l'art

Le SDN est un sujet chaud pour la recherche de nos jours ; ce qui signifie qu'il y a beaucoup de documents publiés au sujet de différentes enquêtes liées au SDN. Certaines des questions liées à l'énonce du problème sont abordées ci-dessous.

Équilibrage de charge à l'aide du contrôleur SDN OpenDaylight avec topologie fat-Tree

À l'Institut KLE en Inde, ils ont créé un algorithme d'équilibrage de charge dynamique qui distribue efficacement le trafic vers un réseau Fat-Tree via plusieurs chemins alternatifs entre une seule paire d'hôtes. Ils ont effectué plusieurs tests avant et après l'implémentation en se concentrant sur certains paramètres de qualité de service tels que le débit, le délai et la perte de paquets entre deux serveurs du réseau. Les résultats montrent que les performances du réseau sont améliorées après l'exécution de l'algorithme d'équilibrage de charge, et l'algorithme peut améliorer le débit et l'utilisation du réseau [27].

Implémentation Equilibrage de charge basée sur la bande passante dans SDN

À l'université de Benha en Egypte, on travaille à mette en ouvre une technique d'équilibrage de charge basée sur la bande passante pour les serveurs web et comparaison de la performance de differents algorithmes round-robin et connections-based en fonction de réponse et de débit dans des environnements d'émulation Mininet et de test Raspberry Pi en temps réel [23] . Outil HTttperf montre que l'algorithme d'équilibrage de charge proposé améliore le débit et le temps de réponse des serveurs Web en plus d'atteindre l'utilisation des serveurs [23].

Implémentation équilibrage de charge dynamique dans environnement virtuel avec open-flow

À la faculté des sciences de l'information et de l'ingénierie en chine, ils supposent un algorithme d'équilibrage de charge dynamique des serveurs dans l'environnement virtuel de tel façon que le le contrôleur utilise le protocole SNMP[8] pour collecter les informations d'état des serveurs, puis calcule la charge des serveurs selon l'algorithme supposé, et finalement sélectionne le serveur de charge le plus léger pour répondre aux utilisateurs [9]. l'algorithme réduit le temps de réponse du serveur, mais n'était pas adapté aux réseaux non structurés et aux centres de données.

[8] *Simple Network Management Protocol*

Implémentation équilibreur de charge de trafic Web avec OpenFlow

À l'Université de Stanford, un système d'équilibrage de charge du trafic Web basé sur OpenFlow appelé Plug-n-Server a été proposé, qui est basé sur la collecte dynamique de la charge du processeur et de l'état de congestion de chaque serveur [24]. Les résultats montrent que le système réduit la latence du réseau et améliore les performances, mais lorsque le nombre de serveurs augmente, cela augmente la charge sur le contrôleur, affectant l'expérience utilisateur et la qualité de service.

Présentation des chapitres

Chapitre 1

Dans ce chapitre, Nous parlerons de l'aspect théorique de notre projet, Nous introduirons les technologies SDN et OpenFlow ainsi que les méthodes d'équilibrage de charge entre les serveurs.

Chapitre 2

Dans ce chapitre, nous présentons les outils logiciels que nous avons utilisés lors de réalisation de notre projet.

Chapitre 3

Dans ce chapitre, nous présentons les différentes étapes de développement de notre projet.

Chapitre 4

Dans ce chapitre, nous allons implémenter notre solution avec GNS3 et tester les performances de notre solution dans différents scénarios.

Chapitre 1

Software Defined Network (SDN)

1.1 Introduction

Dans ce chapitre nous parlerons de ce qui est nécessaire et théorique dans notre projet, et à partir de là nous présentons du SDN en général, sa définition, ses fonctions, ses applications, ainsi que sa structure, et nous parlerons également du protocole open flow qui est la base du SDN, nous présentons également de l'équilibrage de charge, de ses avantages et de ses types.

1.2 Software Defined Network

1.2.1 Définition

Le Software-Defined Networking (SDN) est une architecture de réseau qui sépare le plan de contrôle et le plan de données, et permet de centraliser toutes les procédures du plan de contrôle dans un contrôleur SDN, ce qui simplifie les nœuds du réseau et optimise les performances du plan de données[50]. Cette architecture permet aux développeurs de mettre en œuvre de nouveaux algorithmes à placer dans le contrôleur qui peut gérer le réseau et modifier ses fonctionnalités [50].

1.2.2 Architecture

L'architecture de SDN est formée de 3 couches essentielles comme illustré dans la figure 1.1, chacun a un rôle principal qui sont couches d'infrastructure, couche de contrôle et couche d'application.
La couche de contrôle contient tous les éléments de plan de contrôle du réseau pour gérer des équipements de communication .la couche infrastructure est responsable de la transmission des paquets.Dans la couche d'application, les applications métier peuvent interagir avec les services et les capacités réseau. Il faut aussi des interfaces programmables sud et nord pour permettre au contrôleur de communiquer avec les deux autres couches[46].

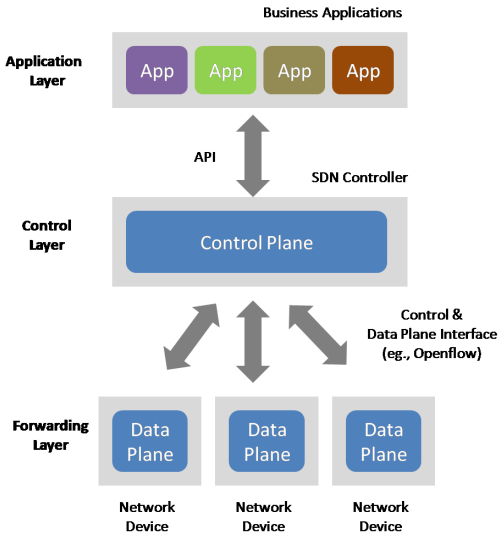


Fig. 1.1 : L'architecture SDN
[34]

Une description plus détaillée des composants du SDN est donnée dans cette section :

- **Couche de contrôle**

  Dans le plan de contrôle, un programme est placé au-dessus qui est séparé des commutateurs, comme indiqué sur la figure 1.2. Ce plan manipule la table de transfert pour chaque requête de commutateur en fonction de l'en-tête du message d'entrée de paquet qui est créé par le commutateur et envoie la réponse via le message de paquet ou de flux et suit toutes les demandes d'application[46]. Les contrôleurs peuvent être déployés de façon centralisée ou distribuée. L'un des avantages les plus évidents de l'utilisation d'un contrôleur centralisé est que la gestion et la récupération des informations seraient appliquées à partir d'un point logique (contrôleur) résultant réseau uniforme. Comme pour tout modèle centralisé, l'inconvénient d'un contrôleur centralisé est qu'un seul point de défaillance peut faire tomber tous les périphériques réseau en cours d'exécution [46]. Il existe plusieurs types et versions de plates-formes de contrôleur de réseau qui sont comparées dans le tableau 1.1.

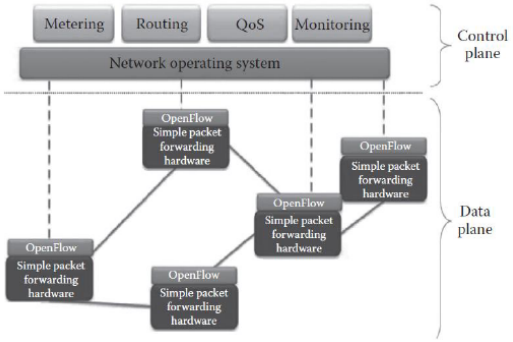


Fig. 1.2 : Séparation plan de contrôle au commutateur[46]

Contrôleur	Langage	Créé par	Open Source	Version Open-Flow	Description
NOX	Python C++	Nicira	oui	1.0 ,1.3	Asynchrone, modèle de programmation basé sur les événements, cadre basé sur les composants. NOX MT est multithread et améliore le débit et le temps de réponse. le débit et le temps de réponse
POX	Python	Nicira	oui	1.0	Cadre multiplateforme basé sur des composants
Beacon	Java	Stanford Univercity	oui	1.0.1	Développement multiplateforme, dynamique et rapide basé sur des événements.
Maestro	Java	Rice Univercity	oui	1.0	Application de contrôle de réseau modulaire
Floodlight	Java	Big Switch Networks	oui	1.0	Basé sur Beacon, l'architecture de base est modulaire, agent open source (Indigo)
Floodlight Plus	Java	Big Switch Networks	oui	1.3	New version of floodlight for supporting OF 1.3
Ryu	Python	NTT Labs	oui	1.0, 1.2, 1.3, 1.4, 1.5	Basé sur les composants, prenant en charge le développement de composants dans d'autres langages, la gestion des événements et la bibliothèque NETCONF réutilisable, la bibliothèque sFlow/Netflow
Open Daylight	Java	Linux Foundation	oui	1.0, 1.3	Plate-forme de contrôleur modulaire et flexible, prenant en charge plusieurs southbound protocoles

Tab. 1.1 : Messages entre le contrôleur et le commutateur [46]

Un réseau peut avoir plusieurs contrôleurs, de sorte que chaque contrôleur est responsable du contrôle d'un ensemble de commutateurs réseau qui peuvent interfèrer les uns avec les autres, donc un contrôleur est choisi comme contrôleur principal et les autres comme contrôleurs de secours[46].

Chaque commutateur est contrôlé avec son contrôleur et les contrôleurs sont distribués dans le réseau, en utilisant la base de données et les applications qui sont écrites sur les contrôleurs considèrent tous les contrôleurs comme un contrôleur central. Bien que cette option permette de réduire la surcharge générale et d'assurer l'évolutivité et la disponibilité, mais parmi les problèmes les plus importants c'est la difficulté d'établir la cohérence et la compatibilité entre le système d'exploitation du réseau et tous type de version d'openflow , et même entre les contrôleurs [46].

La granularité des paquets nécessite une décision de plan de contrôleur chaque fois qu'un paquet est arrivé dans le plan de données. Lorsqu'on a une erreur de table dans un commutateur, le commutateur encapsule l'intégralité du paquet reçu comme un message paquet entrant et l'envoie au contrôleur, une décision sera prise et un message de sortie de paquet sera envoyé au commutateur. Dans ce cas, une énorme quantité de données passeraient

par le contrôleur et le commutateur pourrait causer un retard. Lorsqu'un flux est passé par un commutateur, pour la première fois que le commutateur reçoit ce flux, il nécessite une décision du plan du contrôleur, donc envoie un paquet qui ne contient que l'en-tête du paquet d'origine et du côté du contrôleur, la décision sera prise un paquet qui contenant l'en-tête du paquet d'origine sera envoyé au commutateur qui décidera en fonction de cette décision de contrôleur[46].

- **Couche d'infrastructure**

  La couche d'infrastructure se compose de nombreux dispositifs de communication qui créent un réseau qui peut être une combinaison de commutateurs et de routeurs. Un commutateur dans un réseau défini par logiciel est l'élément de transfert de base qui communique avec le contrôleur via une interface ouverte telle qu'OpenFlow, de sorte qu'un commutateur openflow contient au moins trois parties principales : table de flux, canal sécurisé et protocole openflow[46].
  Les tables de flux sont utilisées pour rechercher des paquets et les transmettre. Le canal sécurisé est généralement un canal TLS[1] ou SSL[2] entre le commutateur et le contrôleur. Le protocole OpenFlow est utilisé pour communiquer et gérer les commutateurs. Le contrôleur peut mettre à jour, supprimer et ajouter des entrées de flux de table de flux à l'aide de messages OpenFlow[46].

- **Couche d'application**

  Une couche d'application est composée d'applications qui communiquent avec le contrôleur dans une couche de contrôle en utilisant des connecteurs appelés API[3] Northbound. La couche d'application agit comme une zone ouverte pour développer de nombreuses applications, autant que possible, en utilisant les informations réseau complet par exemple son statut, ses données, etc.Il peut y avoir certains types d'applications qui peuvent être développés telles que les changements liés au réseau, la configuration et la gestion du réseau, la surveillance du réseau, le dépannage du réseau, les politiques du réseau et la sécurité. Ces applications SDN peuvent fournir une variété de résolutions globales pour les réseaux du monde réel ainsi que les réseaux de centres de données[20].

- **Northbound API**

  L'API Northbound fournit une abstraction des fonctionnalités du réseau, avec une interface programmable permettant aux applications de consommer des services réseau et de configurer dynamiquement et de permettre aux applications d'appliquer le comportement du réseau. Celui-ci est chargé d'établir la communication entre le plan de d'application et le plan de contrôle [52].

[1] *Sécurité de la couche de transport*
[2] *Secure Socket Layer*
[3] *Application Programming Interface*

- **Southbound API**

  Les API Southbound contribuent à un contrôle efficace du réseau et permettent au contrôleur du SDN d'apporter des modifications en fonction des demandes et des exigences en temps réel. OpenFlow est la première et peut-être la plus connue des interfaces sud. Il s'agit d'une norme industrielle qui décrit la manière dont le contrôleur du SDN doit travailler avec le plan de transfert pour contrôler le réseau.[1].

1.2.3 Avantages

La technologie SDN basée sur OpenFlow permet au service informatique de répondre à la nature dynamique et à une bande passante élevée des applications d'aujourd'hui, d'adapter le réseau à l'évolution des besoins opérationnels et de réduire considérablement la complexité opérationnelle et de gestion[19]. Les avantages que les entreprises et les opérateurs peuvent obtenir grâce à l'architecture SDN basée sur OpenFlow comprennent [19] :

- **Contrôle centralisé des équipements multi fournisseurs**

  Un contrôleur SDN peut contrôler n'importe quel périphérique réseau compatible OpenFlow, y compris les commutateurs, les routeurs et les commutateurs virtuels, de n'importe quel fournisseur[19]. Au lieu de gérer des groupes d'appareils de différents fournisseurs, le service informatique peut utiliser des outils d'orchestration et de gestion basés sur SDN pour déployer, configurer et mettre à jour rapidement les appareils sur le réseau[19].

- **Réduction de la complexité grâce à l'automatisation :**

  le SDN basé sur OpenFlow fournit un cadre flexible pour l'automatisation et la gestion du réseau, permettant de développer des outils qui automatisent de nombreuses tâches de gestion effectuées manuellement aujourd'hui. Ces outils automatisés réduiront les tâches opérationnelles et réduiront la gigue du réseau[19]. De plus, les applications basées sur le cloud peuvent être gérées à l'aide d'un système d'orchestration et de configuration intelligent, ce qui réduit encore les frais généraux opérationnels tout en augmentant l'agilité opérationnelle[19].

- **Un taux d'innovation plus élevé**

  L'adoption de SDN accélère l'innovation des entreprises en permettant aux opérateurs de réseaux informatiques de programmer et de reprogrammer littéralement le réseau en temps réel pour répondre aux besoins spécifiques des entreprises et des utilisateurs au fur et à mesure qu'ils surviennent[19]. En virtualisant l'infrastructure réseau et en la soustrayant à des services de réseau individuels, par exemple, SDN et OpenFlow donnent à service d'informatique et peut-être même aux utilisateurs la possibilité d'adapter le comportement du réseau et d'introduire de nouveaux services et capacités de réseau en quelques heures[19].

- **Augmentation de la fiabilité**

  Le SDN permet au service informatique de définir des déclarations de configuration et de politique de haut niveau, qui sont ensuite transmises à l'infrastructure via Open flow[19]. Une architecture SDN basée sur OpenFlow réduit la probabilité de défaillance du réseau en éliminant le besoin de configurer individuellement les périphériques réseaux, un service ou une application est ajouté ou changé, ou qu'une politique change[19].

1.2.4 Application de SDN

Le SDN est un moyen important d'innovation réseau de ces dernières années car il offre beaucoup plus de flexibilité et d'évolutivité que les réseaux existants. en plus, elle permettre d'ajouter dynamiquement des fonctionnalités et des services sous forme d'applications et de faciliter leur mise en œuvre grâce à son architecture. Et les applications qui peuvent être appliquées au SDN certaines d'entre elles résident dans la facilitation du trafic via l'équilibrage de charge, le contrôle du réseau .. etc., dont nous parlerons d'eux dans cette section :

- **Équilibrage des charges pour les serveurs d'applications**

  La fonction d'équilibrage de charge entre plusieurs serveurs est mise en œuvre par des équipements dédiés est installés dans le réseau. Malgré l'utilisation du SDN, les commutateurs OpenFlow peuvent gérer automatiquement cette fonction et distribuer le trafic vers différents serveurs,mais il n'est pas très évolutif ; il est donc possible d'écrire une application qui fonctionne au-dessus du contrôleur et qui peut fournir une application d'équilibrage de charge évolutive et efficace [7].

- **Mise à niveau des centres de données**

  Les centres de données sont une composante interne importante de nombreuses grandes entreprises qui disposent d'un grand nombre de centres de données pour gérer une demande massive et répondre rapidement. Ces centres de données sont extrêmement coûteux et compliqués à entretenir et à gérer. SDN et OpenFlow permettent aux entreprises de réduire les coûts d'installation et de configuration du centre de données, car la partie du réseau qui transporte les données peut être gérée à partir d'un contrôleur [26].

- **Facilite la gestion de Réseaux de campus**

  Les réseaux de campus peuvent être difficiles à gérer, notamment en raison du besoin constant d'unifier les réseaux Wi-Fi et Ethernet. Les contrôleurs SDN peuvent profiter aux réseaux de campus en fournissant une gestion et une automatisation centralisées, une sécurité renforcée et une qualité de service au niveau des applications sur le réseau [45].

- **Sécurité des centres de données**

  Les entreprises utilisent des pares-feux traditionnels pour protéger leurs centres de données, d'autre part, les SDN peuvent fournir une protection efficace et simplifier la gestion des pare-feu grâce à un contrôle et une automatisation centralisée, ce qui permet également aux administrateurs de visualiser, de modifier et de surveiller l'activité réseaux pour réduire les risques [45].

1.3 Open Flow Protocol

1.3.1 Vue Général sur OpenFlow

Open Flow est un protocole qui décrit la communication entre les commutateurs et le contrôleur ainsi que le comportement des commutateurs .Notamment ce protocole ne décrit pas le fonctionnement du contrôleur mais sa communication avec les commutateurs. Open Flow est le seul protocole Open Source de SDN, ce qui supporte un des principes de SDN "Openness".

- **La création d'Open Flow**

  Open Flow est développé par "openflow.org" qui est une organisation à but non lucratif créée en 2008 dans le but de développer le protocole Open Flow. Initialement, Open Flow était un protocole orienté vers les chercheurs dans le domaine de "Open Switching" et a été mis à disposition à des fins expérimentales.

  Au fur et à mesure de l'évolution des spécifications la première version publique était en 31 Décembre 2009, la version 1.0.0. Après la publication de cette version la vision a changé vers la commercialisation et l'accélération de distribution de cette technique. La fondation "Open Network Foundation" a été crée en 21 mars 2011 pour établir ce but. Toutes les versions qui suivent ont été publiées et développées par le ONF[4].

- **L'impact d'Open Flow**

  Après la commercialisation de Open Flow par la ONF et après la version 1.1, cette technologie a été considérée comme suffisamment mature pour être le standard pour le SDN. Il y a un certain nombre de promoteurs de SDN qui offrent des solutions SDN qui ne sont pas basés sur OpenFlow, mais pour ONF et la communauté des développeurs OpenFlow présente la vision de SDN, un protocole riche, ouvert et flexible.

  Pour ONF l'étape suivante était de convaincue les fabricants des commutateurs d'inclue ce protocole dans leur produit, pour cette raison cette fondation a créé la spécification Open Flow [18] et une liste officielle des commutateurs qui supporte le protocole [17] comme une méthode de promotion. Open Flow comme un protocole peut être utilisé avec

[4] *Open Network Foundation*

n'importe quel CPU[5] mais avec le standard utiliser et les méthodes des tables de commutateur implémente avec des ASIC[6] utilisé dans les commutateurs, pour que openFlow soit fiable comme une solution de production il faut d'une part que ONF profite des architectures des ASIC dans les protocoles, de l'autre part que les fabricants acceptent ce protocole et le prend en considération lors de développement de leur commutateur.

1.3.2 Le Protocol OpenFlow

Dans cette section,nous parlerons de OpenFlow. Par concept OpenFlow comme une technique est composé de trois parties les commutateurs, le contrôleur et le protocole on va expliquer chaque partie et son rôle pour établir la vision de SDN.

OpenFLow Switch

Un commutateur OpenFlow est un commutateur qui communique avec un contrôleur avec un canal sécurisé. La fonction principale du commutateur est de recevoir un paquet et de le transmettre au port correspondant après avoir apporté les modifications nécessaires au paquet.

Dans la figure 1.3 on voit une illustration des fonctionnalités d'un OpenFlow switch, l'aspect unique de ce switch est sa fonctionnalité d'analyse les paquets est trouvé un match dans le "Flow Table" afin de procéder une de ces fonctionnalités.

- Ré-acheminer le paquet après des modifications si c'est nécessaire .
- Déposer le paquet .
- Ré-acheminer le paquet vers le contrôleur.

Quand un paquet arrive au commutateur, le commutateur utilise la fonctionnalité de "matching" pour comparer le paquet avec le "Flow Table" pour décider comment gérer le paquet, si aucun match n'est trouvé le paquet est ré-acheminer vers le contrôleur avec un message PACKET_OUT.

[5] *Central Processing Unit*
[6] *Application Specific Integrated circuit*

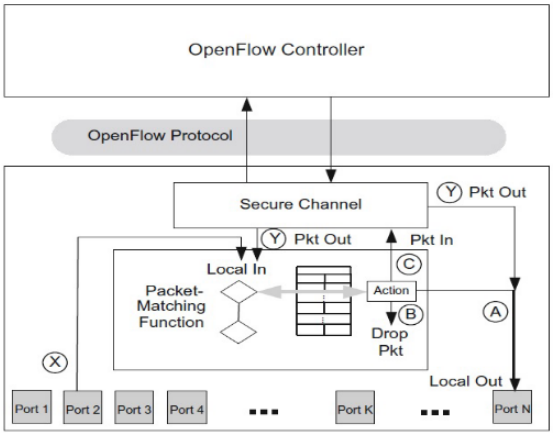


Fig. 1.3 : Fonctionnalité de commutateur Open Flow [4].

Il existe deux types des commutateurs OpenFlow :

- **OpenFlow-only :** un commutateur qui n'utilise que le protocole OpenFlow pour réacheminer les paquets comme décrit précédemment.

- **OpenFlow-Hybrid :** un commutateur qui supporte OpenFlow aussi que la méthode classique, ce type de commutateur utilise une fonctionnalité de pré traitement pour décider quelle méthode à utiliser pour réacheminer le paquet. Ce dernier est le plus utiliser pour supporter la migration vers cette nouvelle technique sans éliminer la rétro compatibilité.

Le Contrôleur OpenFlow

Nous avons discuté de l'importance de la séparation de plan des données et le plan de contrôle. Le plus grand avantage du contrôleur OpenFlow c'est qui est séparé au niveau du matériel , cet avantage le permettre de faire ces fonctionnalités sans déranger le fonctionnement des commutateurs. Autres avantages auxquels le contrôleur utilise le protocole OpenFlow pour communiquer avec les commutateurs. Les développeurs sont libres sur l'implémentation tant qu'ils utilisent OpenFlow pour communiquer avec les commutateurs.L'autre avantage est qu'un seul contrôleur peut contrôler plusieurs commutateurs au même temps en parallèle.

Le Protocole OpenFlow

Le Protocole OpenFLow est le protocole qui définit la communication entre le contrôleur et le commutateur. Le protocole consiste d'une collection des messages qui sont envoyés par le contrôleur au commutateur et vice-versa. Ce protocole permet au contrôleur de programmer les Flux dans le commutateur avec des actions basique (Ajouter, Modifier et Supprimer). Un flux dans les tableaux des flux définit par un match sur un ou plusieurs entêtes du paquet et une série des actions sur le paquet. Le protocole OpenFlow a évolué de manière significative avec chaque version d'OpenFlow, pour avoir une plus large gamme de matching d'en-tête et d'actions sur le paquet ce qu'on va discuter dans la section suivante.

1.3.3 L'évolution du protocole OpenFlow

La première version de OpenFlow,est la version 0.2.0 été publiée le 28 mars 2008 et la plus récente version est la version 1.5.1 [41] publié le 26 mars 2015. Pendant cette période, le protocole a énormément évolué. Dans cette section, nous parlerons des versions majeures d'OpenFlow et de leur évolution.

OpenFlow v1.0.0

Cette version de OpenFlow été considérer comme la première version majeure de la technologie. cette version été publié le 31 Décembre 2009 [38].

- **Porte and Queues (files d'attente)**

  Dans cette version de protocole une porte de OpenFlow correspond à un port physique du commutateur, et chaque port possède une file d'attente de paquets à acheminer. Ces files d'attente utilisent un algorithme de planification pour assurer la qualité de service nécessaire

- **Flow Table**

  Le flux Table est l'élément plus important dans un commutateur OpenFlow [6]. Il contient des entrées des flux, dans le figure 1.4 on voit qu'un flux est composé de trois parties

  - **Les entêtes :** les entêtes utilisaient pour comparer les paquets avec plus de détails sur la fonctionnalité de matching dans la partie au-dessus
  - **Action :** les actions à faire en cas de correspondance .
  - **Statistics :** Informations sur les statistiques de ce flux (compteur de paquets).

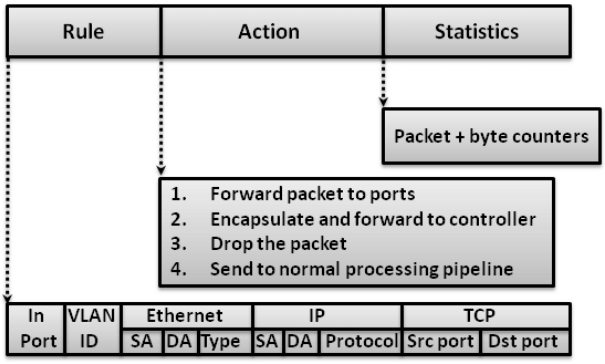


Fig. 1.4 : Entré dans la table des Flow [6]

- **Le matching des paquets**

  Dans le cas où un paquet arrive au switch il compare avec les flows dans le Flow Table, les entêtes du paquet sont comparés avec les entêtes suivants dans le Flow table.

- VLAN[7] ID : Identifiant de VLAN
- VLAN priority : Priorité du VLAN
- Ethernet source address : L'adresse MAC de l'émetteur .
- Ethernet destination address : L'adresse MAC de réception .
- Ethernet frame type : Type de protocole Ethernet .
- IP source address : L'adresse IP de l'émetteur .
- IP destination address : L'adresse IP de réception .
- IP protocol : Type de protocole IP
- IP Type of Service (ToS) bits .
- TCP[8]/UDP[9] source port
- TCP/UDP destination port

Dans le cas où aucun match n'est trouvé le paquet est réacheminé vers le contrôleur

- **Actions**

L'action à faire en cas d'une correspondance avec une entrée dans le Flow Table. Les actions plus courantes sont ignorées ou réacheminées le paquet, mais dans cette version de OpenFlow il existe quatre autres Actions :

- **LOCAL :** Signifie que ce paquet est destiné a être consommer par le software du switch. Par exemple les messages reçus par le contrôleur.
- **ALL :** ça signifie que ce paquet est destiné au domaine de broadcast, le paquet sera réacheminé vers tous les ports du commutateur.
- **CONTROLLER :** ça signifie que le message est destiné d'être réacheminé vers le contrôleur.
- **IN_PORT :** signifie que ce paquet est destiné à être renvoyé au port auquel il arrive.

- **Messages entre le contrôleur et le commutateur**

Le tableau 1.2 représente les messages envoyés entre le contrôleur et le commutateur et leur description .

[7] *Virtual Local Area Network*
[8] *Transmission Control Protocol*
[9] *User Datagram Protocol*

OpenFlow v1.1.0

OpenFlow version 1.1.0 [39] a été publié le 28 Février 2011 comme prochaine version majeure après la version 1.0.0. Elle a ajouté des nouvelles fonctionnalités comme la possibilité d'avoir multiple Flow Tables dans un switch et la possibilité d'utiliser les groupes. L'impact de cette version n'était pas aussi évident que son prédateur. Elle est considérée comme une version transitoire qui prépare les fondations pour la version majeur suivante.

- **Multiple Flow Table**

  Avec cette version de OpenFlow la possibilité d'avoir multiple Flow Table a été rajouter. Comme illustré dans le figure 1.5 la disponibilité de multiple Flow Table permettre le traitement d'un paquet en série. Un flow dans le tableau peut avoir l'action GOTO qui envoie le paquet pour continue sa fonctionnalité du Matching dans un autre Flow Table. Cette nouvelle fonctionnalité augmente considérablement la complexité des Flux et la logique du matching pour gérer des scénarios plus complexes.

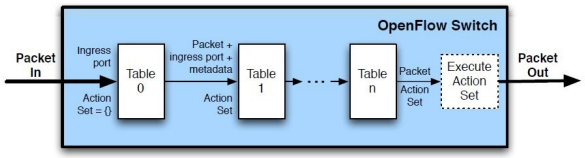


Fig. 1.5 : Tableau des Flow multiples [44]

Message	Description
Message envoyé du contrôleur au commutateur	
Packet Out	Envoyer des paquets pour la sortie du port donné
Flow Mod	Envoyer les entrées des Flows pour modifier le tableau des FLows dans le commutateur.
Port Mod	Changer l'état d'un port
Set Config	Modifier les configurations du commutateur
Message envoyé du contrôleur au commutateur	
Features Request/Reply	Récupérer les capacités supportées par commutateur
Stats Request/Reply	Collecter les statistiques d'un commutateur
Get Config Request/Reply	Interroger les paramètres de configuration du commutateur
Barrier Request/Reply	Confirmer l'achèvement des opérations demandées par le contrôleur
Queue Get Config Request/Reply	Configurations de la file d'attente des requêtes
Messages envoyés du commutateur au contrôleur	
Packet In	Envoyer les paquets reçus par le commutateur
Flow Removed	Informer que l'entrée d'un Flow a expiré
Port Status	Informer du changement de l'état d'un port
Messages envoyés à la fois par le contrôleur et le commutateur	
Hello	Pour initialiser la connection
Echo Request/Reply	Confirmer l'existence de la connexion contrôleur-commutateur
Error	Informer de toute erreur
Vendor	Pour les utilisations définies par le fournisseur

Tab. 1.2 : Messages échangés entre le contrôleur et le commutateur [51]

- **Groups**

  Cette nouvelle fonctionnalité est considérée comme une implémentations plus riche de la fonctionnalité FLOOD mentionnée dans 1.3.3 , c'est une opération de multicast plus flexible. Un group_ID contient plusieurs "Buckets" chacun a une ou plusieurs actions associées avec. Plusieurs Flow peuvent être associe avec ce "Bucket", dans le cas de nécessité de modification de tous ces Flow une seule modification sur le "Bucket" peut établir le même résultat.

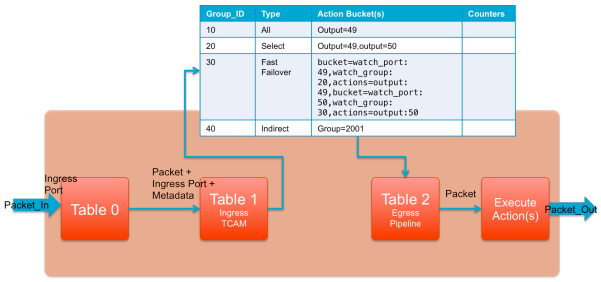


Fig. 1.6 : OpenFlow Group [43]

- **Prise en charge MPLS et VLAN**

  Comme nous l'avons mentionné dans la section 1.3.3 la version 1.0 avec le support des tags des Vlan, la possibilité d'ajouter ou supprimer des tags multiple des Vlan était déjà présenté donc l'implémentions des tags de MPLS était on continuation de même concept.

OpenFlow v1.2.0

OpenFlow version 1.2.0 [40] a été publiée le 5 Décembre 2011 avec des nouvelles fonctionnalités majeur comme un support extensible de fonctionnalité de matching et le support de l'IPV6.

- **Support extensible de matching**

  Les versions précédentes de OpenFlow avaient un support limite des entêtes pour la fonctionnalité de matching, c'était un des grand facteurs limiteurs de OpenFlow, les développeurs n'avait pas la possibilité pour une méthodologie de matching extensible. Avec cette nouvelle version une logique de matching plus riche et extensible le OpenFlow Extensible Match définit des valeurs couple qui peut décrit n'importe quel type des entêtes des paquets pour le matching même avec un Bitmask ou un Wildcard.

  Avec le OXM[10] les mêmes valeurs des entêtes disponibles dans la fonctionnalité de matching sont disponibles pour la modification des paquets

[10] *OpenFlowExtensible Match*

- **Contrôleur multiple**

  Dans les versions précédentes de OpenFlow, un OpenFlow switch était limité à une seule connection avec un seul contrôleur, dans le cas où cette connection est perdu, le switch entre dans un état défaillant. Avec cette nouvelle version un switch est autorisé de connecter a plusieurs contrôleurs. Le commutateur assure qu'il envoie le message de réponse au contrôleur qu'il a envoyé la demande d'information.

1.3.4 Limitations de OpenFlow

OpenFlow est en développement continue il n'est pas évident de juger des limitations parce qu'il peut être fixe dans les nouvelles versions. Mais avec notre expérience et nos besoins comme ingénieurs réseaux nous voyons que la limitation de matching sur les entêtes est une grande limitation pour le développement. Une fonctionnalité d'inspection plus riche qui inclut même les données application peut ouvrir la port pour des applications illimitées de cette technologie et qui peut avoir un grand impact sur le domaine de SDN.

1.4 Équilibrage de charge

Le rôle d'un équilibreur de charge est de répartir les charges de travail entre plusieurs ressources de communication. Cela permet d'optimiser l'utilisation des ressources, de maximiser le débit et de réduire le temps de réponse pour éviter de surcharger une seule ressource. Les répartiteurs de charge peuvent être implémentés en logiciel ou en matériel, selon le type de charge qu'ils distribuent [2].

1.4.1 Les avantages de l'équilibreur de charge

Lorsque le niveau de trafic du réseau approche, atteint ou dépasse le maximum prévu, on dit que le réseau est saturé.L'équilibreur de charge ne fonctionne pas seulement comme un régulateur de trafic mais offre également des avantages comme l'analyse prédicative qui identifie la congestion et y apporte une solution. L'équilibrage de la charge est une approche importante pour résoudre les problèmes de congestion du réseau résultant d'une allocation inefficace des ressources. Le routage peut être effectué sur des routes appropriées et les routes encombrées peuvent être évitées en fonction de la demande actuelle du trafic dans le réseau [2].Il existe d'autres avantages pour l'équilibrage de charge[2] :

- L'équilibrage de charge améliore les performances de chaque équipement et donc les performances globales du système.
- L'équilibrage des charges réduit le temps d'inactivité des tâches.
- Utilisation maximale des ressources
- Le temps de réponse devient plus court
- Débit plus élevé

- Une plus grande fiabilité
- Faible coût mais gain élevé
- Extensibilité et croissance incrémentale

1.4.2 Types d'équilibrage de charge

Plusieurs algorithmes d'équilibrage de charge ont été développés au cours des dernières années, mais aucun ne fonctionne pour toutes les applications. Le choix d'un équilibreur de charge approprié dépend des paramètres d'application tels que l'équilibrage de la qualité, les modèles de génération de charge et également des paramètres matériels tels que la surcharge de communication. En général, les algorithmes d'équilibrage de charge sont de deux types : le premier est l'équilibrage de charge statique et le second est l'équilibrage de charge dynamique [2] .

Équilibrage de charge statique

Dans un algorithme statique, le trafic est réparti uniformément entre les serveurs. L'algorithme nécessite une connaissance préalable des ressources système, de sorte que la décision de déplacer la charge ne dépend pas de l'état actuel du système. L'algorithme statique est correct dans les systèmes avec de petits changements de charge [13].

Équilibrage de charge dynamique

Les algorithmes dynamiques désignent les poids appropriés sur les serveurs et en recherchant dans le réseau entier un serveur le plus léger préféré pour équilibrer le trafic [13]. Contrairement à l'équilibrage de charge statique, l'équilibrage de charge dynamique nécessite les informations du système avant l'exécution et la sélection de serveur approprié va choisir en temps réel (pendent l'exécution) [13].

1.4.3 Les algorithmes d'équilibrage de charge

Algorithme d'équilibrage de charge Round-Robin

L'équilibrage de charge round robin est l'une des méthodes les plus simples pour répartir les demandes des clients sur un ensemble de serveurs. Le principe est parcourant la liste des serveurs du groupe puis d'envoyer tour à tour la requête du client à chaque serveur à tour de rôle. Lorsqu'il atteint la fin de la liste, l'équilibreur de charge ,revient en arrière et recommence à envoyer les demandes des clients à partir du premier serveur [25]. Le principal avantage de cet algorithme est qu'il est très facile à implémenter, Mais dans un scénario où les demandes de charge diffèrent considérablement, il ne peut pas distribuer les charges efficacement [25].

Weighted Round Robin

Cet algorithme est dérivé du round-robin, qui offre de meilleures performances que le round-robin traditionnel car il attribue des poids plus élevés aux serveurs les plus performants tel que le serveur le plus performant recevra plus de charge de travail entrante [35].

Least-connection

L'algorithme calcule dynamiquement le nombre de connexions associées à chaque serveur, puis choisit le serveur qui a le plus petit nombre de connexions et attribue une nouvelle charge de travail entrante au ce serveur choisi [35].

Weighted least-connection

L'algorithme compte également les connexions associées à un serveur, mais corrèle les nouvelles charges de travail entrantes selon un facteur calculé en multipliant le poids d'un serveur par son nombre de connexions, puis prendre le dernier temps de réponse de chaque serveur, puis en prenant le serveur ayant le temps de réponse le plus court est sélectionné comme le prochain serveur approprié [35].

Shortest expected delay

Dans cet algorithme, le dernier temps de réponse de chaque serveur est pris en compte et le serveur avec le temps de réponse le plus court est sélectionné comme prochain serveur approprié [35].

Random Algorithm (RAND)

Dans cet algorithme si la charge de travail est supérieure à un seuil qui est spécifiée par n'importe quel serveur, elle est transférée vers un autre serveur sélectionné au hasard sans vérifier les informations du ce dernier, qu'il soit surchargé ou sous-chargé. Cet algorithme ne conserve aucune information de charge reliée aux serveurs [48].

Weighted Response Time Load Balancing Algorithm

Cet algorithme est basé sur le temps de réponse en tenant compte de la valeur de poids sur chaque serveur. Lorsque le client envoie une demande, l'algorithme choisit serveur le plus petit charge et temps de réponse pour répondre aux exigences du client [37].

Hachage IP[11] source

Dans cette approche, un algorithme est utilisé qui prend en compte les adresses IP source et de destination du client et du serveur, puis les combine pour générer une clé de hachage unique.

[11] *Internet Protocol*

Les clés générées sont ensuite utilisées pour affecter des clients à des serveurs spécifiques[21]. Étant donné que les clés peuvent toujours être régénérées, en cas de suspension de session, la méthode d'équilibrage de charge de hachage IP source permet de garantir que les demandes des clients sont dirigées vers le même serveur que celui utilisé auparavant. Il est utile car il garantit que le client peut se connecter à la session après s'être déconnecté si la session est toujours active[21].

Équilibrage adaptatif des charges basé sur un agent.

Dans un groupe de serveurs, un agent est affecté à chaque serveur qui rend compte à l'équilibreur de charge de sa charge actuelle. Cette information en temps réel est utilisée pour décider quel serveur doit être placé pour traiter les requêtes au mieux[21]. Ceci est utilisé en combinaison avec d'autres techniques comme "Weighted Round Robin" et "Weighted Least Connection"[21].

1.4.4 Equilibreur de charge avec SDN

L'équilibrage de charge fonctionne comme un protocole de routage conscient dans le réseau défini par logiciel, c'est une entité nécessaire qui aide la disponibilité et l'évolutive qui conduit à obtenir le temps de réponse minimal de l'application. Lorsque le nombre de utilisateurs connectés au réseaux augmente, entraîne une augmentation de trafic qui provoque une congestion du réseau et des pertes de paquets[8]. Ainsi, pour résoudre ce problème, la méthode des techniques d'équilibrage de la charge augmente l'efficacité du réseau et maximise le débit et minimise l'intervalle du temps de réponse et réduit le brouillage. Les réseaux traditionnels ne sont pas approfondis dans la méthode d'équilibrage de la charge, mais les réseaux définis par un logiciel sont très efficaces et très performants. L'équilibre de la charge est un élément constitutif du gestionnaire du SDN, qui est situé dans un point central et les méthodes pourraient être connectées sur lui[8]. Le mécanisme équilibreur de charge avec SDN est illustré dans la figure1.7

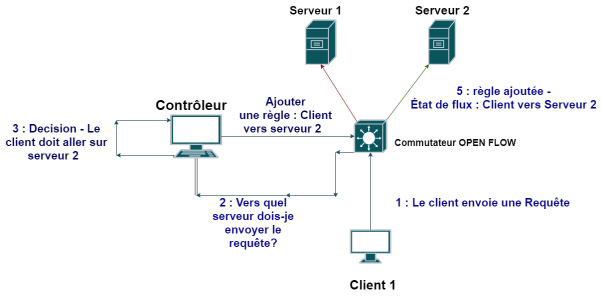


Fig. 1.7 : Mécanisme de l'équilibreur de charge avec SDN

1.5 Conclusion

Dans ce chapitre, nous présentons la technologie SDN et son protocole OpenFlow et décrivons tous les aspects de cette technologie. Nous avons également parlé de l'impact des équilibreurs de charge, de leurs types, avantages et mécanismes dans la solution SDN.

Chapitre 2

Logiciels et matériels

2.1 Introduction

Notre projet est fortement basé sur le Software(Logiciels). Dans ce chapitre, nous allons énumérer les outils logiciels que nous avons choisis d'utiliser pendent la réalisation de notre projet tout en expliquant les raisons pour lesquelles nous les considérons comme le choix approprié pour notre cas d'utilisation. D'autre part, bien que le matériel que nous avons choisi d'utiliser ne soit pas obligatoire pour notre cas d'utilisation, nous allons lister les outils que nous avons choisi d'utiliser,ces outils sont utilisés pour diverses parties du projet que nous allons énumérer tout en justifiant brièvement notre choix.

D'autre part, bien que le matériel n'a pas été très pertinent dans notre développement, nous avons vu qu'il vaut la peine d'une brève mention pour fournir le tableau complet de la suite d'outils utilisés dans ce projet.

Nous tenons aussi mentionner que tous les outils logiciels utilisés pendant le développement sont des outils open source soutenus par une licence gratuite et soutenus par une communauté forte qui a aidé les outils à prospérer et à évoluer.

2.2 Logiciels

2.2.1 Émulateur

Mininet

Mininet est un simulateur de réseau écrit en Python et C qui peut créer des hôtes virtuels, des commutateurs et des liens. Il prend en charge divers commutateurs OpenFlow basés sur des logiciels, des topologies personnalisées et fournit une API Python pour la programmabilité du réseau [36]. Il s'exécute sur un seul noyau Linux et utilise la virtualisation pour simuler un réseau complet à l'intérieur du système d'exploitation. Cependant, l'hôte, le commutateur, les routeurs et les liens créés sont des éléments du monde réel bien qu'ils soient créés au moyen d'un logiciel. Il nécessite un support réseau du noyau Linux, qui fournit à chaque processus ses propres tables de routage et ARP indépendantes [31].

On a choisi ce logiciel pour sa simplicité et sa rapidité. Pour l'avancement rapide et la meilleure expérience développée il était indispensable pour nous d'avoir la possibilité de lancer, modifier et terminer des topologie rapidement pour les prototype. dans notre plan initial on a prédit qu'on aura la nécessité de modifier notre topologie pendant le développement dont la possibilité d'avoir plu sieurs topologies définies en script est prête à lancer était un grand facteur dans notre choix de ce software

GNS3

GNS3 est une interface graphique et une plate-forme de contrôle frontale écrite en Python pour simuler des infrastructures professionnelles informatiques. Les technologies de visualisation utilisées par GNS3 sont les suivantes : Dynamips, VPCS, VMWare Workstation / ESXi, VirtualBox, QEMU / KVM, Docker. De plus, il permet d'émuler les technologies LAN/WAN

en interconnectant des ordinateurs émulés sur des commutateurs ou des routeurs virtuels [10].

Nous avons vu GNS3 comme le meilleur outil pour obtenir le résultat le plus proche de la réalité, GNS3 utilise différents émulateurs, ces émulateurs qui sont utilisés dans l'industrie par des profetionnels. dans notre workflow. Nous avons profité de son support Docker pour simuler des conteneurs Docker comme une solution légère pour profiter pleinement de notre matériel dans une solution hypervised. nous avons également profité de son support Qemu pour émuler des machines virtualisées à part entière, principalement des machines virtuelles linux.

Docker

Docker est une plateforme logicielle permettant d'assemblée les services ou fonctions de l'application et ses différentes bibliothèques, fichiers de configuration, dépendances et autres composants au sein du conteneur qui a été créé , déployer et gérer par docker sur système d'exploitation [32].

Docker container

Les conteneurs Docker sont une plate-forme de développement de logiciels open source. Son principal avantage est le regroupement des applications dans des conteneurs, ce qui les rend portables sur n'importe quel système exécutant un système d'exploitation (OS) Linux ou Windows par exemple une machine Windows peut exécuter des conteneurs Linux en utilisant une machine virtuelle (VM). Bien qu'il soit un acteur majeur dans le domaine des conteneurs, Docker n'est qu'une forme de technologie de conteneur [47].

Qemu

Qemu est un émulateur de machine qui permet d'exécuter le système d'exploitation et les programmes d'une machine sur une autre machine. Dans la plupart des cas, il est utilisé comme virtualiseur en combinaison avec les composants du noyau KVM, et non comme émulateur. Dans ce cas, on utilise la technologie de virtualisation matérielle pour virtualiser le visiteur [53].

2.2.2 Langages de programmation

Dans cette section, nous allons énumérer les langages de programmation utilisés dans notre projet. Tout d'abord, nous tenons à souligner que notre choix ne concerne pas le langage de programmation, mais les outils et les bibliothèques disponibles dans ce langage.

Python

Python est un langage de programmation interprété, orienté objet et de haut niveau avec une sémantique dynamique. Ses structures de données intégrées de haut niveau, combinées au typage dynamique et à la liaison dynamique, le rendent très attrayant pour le développement rapide d'applications. La syntaxe Python simple et facile à apprendre met l'accent sur la lisibilité et réduit donc le coût de maintenance du programme [54].

Python prend en charge les modules et les paquets, qui encouragent la modularité du programme et la réutilisation du code. L'interpréteur Python et la bibliothèque standard d'extension sont disponibles gratuitement sur toutes les principales plates-formes sous forme source ou binaire et peuvent être librement distribués [54].

Des autres outils utilisés qui ont été inclus sous python où PIP et PIENV. nous avons utilisé PIP pour installer les paquets python que nous avons utilisés pendant le développement et nous avons utilisé PYENV pour gérer un environnement python qui n'incluait que les paquets python dont nous avions besoin avec la version respectée pour éviter les conflits avec d'autres projets et les paquets utilisés à l'échelle du système

JavaScript

JavaScript est un langage de script qui a été créé en 1995 dans le cadre de l'interactivité des pages Web et donc une partie essentielle des applications Web. JavaScript est l'un des principaux langages utilisés par les développeurs Web car la grande majorité des sites Web l'utilisent et la plupart des navigateurs Web disposent d'un moteur JavaScript pour l'interpréter, également utilisé dans les serveurs Web avec le runtime Node.js[28].

TypeScript

TypeScript est un langage de programmation libre et open source développé par Microsoft qui a pour but d'améliorer et de sécuriser la production de code JavaScript. Il s'agit d'un surensemble syntaxique strict de JavaScript.

2.2.3 Bibliothèque et Framework

Ryu

Ryu est l'un des contrôleurs les plus populaires dans l'industrie avec le soutien de laboratoires NTT et une communauté active avec une documentation approfondie. Ryu est entièrement écrit en Python et utilise des threads verts pour implémenter le multi-threading. Il fournit des composants logiciels qui permettent aux développeurs de logiciels d'étendre la gestion du réseau et les applications de contrôle pour utiliser les fonctionnalités de contrôleur SDN. il prend actuellement en charge divers protocoles en direction sud tels que OpenFlow, OF-Config, NETCONF, etc.

Il supporte beaucoup de bibliothèques tierces qui incluent des liaisons Open vSwitch Python, de l'aide pour analyser VLAN, MPLS, etc des paquets de différents protocoles. Il prend en charge les protocoles sFlow et Netflow qui sont spécifiques à la gestion du trafic réseau. Ryu prend en charge jusqu'à OpenFlow version 1.4. Il utilise une bibliothèque d'encodeurs et de décodeurs pour les paquets OpenFlow et contient le contrôleur OpenFlow à son cœur. Il se compose d'un exécutable principal appelé gestionnaire Ryu où il écoute une adresse IP spécifique et sur le port 6633, le port OpenFlow standard. Les composants de base du Ryu l'architecture comprend le service de messagerie, la gestion d'événements, etc. Ryu prend en charge les plugins Neutron

Openstack et expose l'API REST[1] pour ses opérations OpenFlow de niveau inférieur.

Ryu utilise des pools de thread pour prendre soin des paquets entrants. Les événements asynchrones sont envoyés par les applications Ryu qui utilisent une file d'attente de réception pour ces événements. FIFO[2] est utilisé pour maintenir l'ordre des événements entrants et les threads sont utilisés pour gérer les événements. Dans la boucle principale du thread, un événement sort d'abord de la file d'attente et appelle le gestionnaire d'événement correspondant. Comme le gestionnaire d'événement est appelé par le thread, il vient dans son contexte ou bloc local, ce qui le fait fonctionner de manière bloquante. En raison de ce comportement ce gestionnaire d'événement ne sera pas en mesure de gérer un nouveau paquet comme son blocage du thread.

React

ReactJs est une bibliothèque Javascript qui a été crée par Facebook depuis 2013 dans le cadre faciliter la création d'un SPA[3], En créant des composants basés sur l'état et en générant une page ou une section HTML à chaque fois que l'état change [56].

2.2.4 Logiciel et Service

Linux

Est un système d'exploitation open source le plus utilise dans le monde basé sur les principes de conception d'Unix, Linux fonctionne dans la plupart des serveurs sur Internet et dans presque tous les produits électroniques grand public contrôlés par un microprocesseur [42].

Git

Git est un logiciel de gestion de versions a été créé en 2005 dans le but l'évolution des fichiers sources et conserve les anciennes versions de chacun d'eux sans rien écraser [16].

Github

GitHub est une plateforme utilisant le logiciel de gestion de versions Git a été lancé en 2008 dans le cadre d'hébergement des services Web et de gestion de développement de logiciels. en Plus le site fourni aussi le contrôle d'accès et des fonctionnalités destinées à la collaboration pour chaque projet [55].

Wireshark

Wireshark est un analyseur de protocole réseau, c'est une application qui capture les paquets de connexion réseau. Wireshark est le sniffeur de paquets le plus utilisé dans le monde, et Comme tout autre sniffeur de paquets, il fait trois choses [11] :

[1] *Representational State Transfer*
[2] *First In, First Out*
[3] *Single Page Application*

- Capture des paquets
- Filterage des packets
- Visualization des paquets

cette opération doit se faire automatiquement sans aucune assistance de notre part

Github Actions

Github Actions est une solution d'Intégration Continue et de Déploiement Automatisé (CI/CD). Il s'agit d'une extension Github. Son fonctionnement repose sur des événements Github tels que le commit push [22].

DockerHub

DockerHub est un service fourni par Docker pour trouver et partager des images de conteneurs entre les équipes de développement. Il s'agit du plus grand dépôt d'images de conteneurs au monde, avec un large éventail de sources de contenu, y compris des développeurs de la communauté des conteneurs, des projets open source et des éditeurs de logiciels indépendants (ISV[4]) qui créent et distribuent leur code dans des conteneurs [15].

Microsoft Azure

Microsoft Azure est un service de cloud computing de Microsoft. Azure offre une gamme d'options de logiciel en tant que service (SaaS[5]), de plateforme en tant que service (PaaS[6]) et d'infrastructure en tant que service (IaaS[7]) pour déployer des applications et des services sur une infrastructure de centre de données gérée par Microsoft [5].

ApacheBench

ApacheBench est un utilitaire qui permet de tester les performances du serveur HTTP Apache. Il est conçu pour vous donner une idée des performances de votre installation Apache. En particulier, il vous permet de déterminer combien de requêtes par seconde votre installation Apache peut gérer[3].

Kernel-based Virtual Machine

KVM (pour Kernel-based Virtual Machine) est une solution de virtualisation complète pour Linux sur du matériel x86 contenant des extensions de virtualisation (Intel VT ou AMD-V)[33].

[4] *Independent Software Vendor*
[5] *Software As A Service*
[6] *Platform As A Service*
[7] *Infrastructure As A Service*

OpenVpn

Openvpn est un VPN[8] open source optimisé pour le télétravail par création des réseaux privés virtuels au-dessus d'internet et sécurise les accès distants aux applications d'entreprise, et ainsi empêcher la fuite ou la captation des données échangées entre le navigateur et une application ou un serveur distant[29].

OpenVswitch

Open vSwitch est un logiciel de commutateur virtuel open source conçu pour les serveurs virtuels. Ce que fait ce logiciel est de transférer le trafic entre différentes machines virtuelless au sein du même hôte, et même entre les machines virtuelles et le réseau physique. Il prend en charge les interfaces de gestion standard telles que NetFlow, sFlow, etc. Open vSwitch peut accepter des extensions de programme et de contrôle à l'aide d'OpenFlow, ainsi qu'à l'aide du protocole de gestion OVSDB[9][30].

2.2.5 SQLite

SQLite est une bibliothèque écrite en C qui propose un moteur de base de données relationnelle accessible via SQL[49].

2.3 Materiels

2.3.1 Developement Local

Notre Environnement de développement locale était composé de nos ordinateurs portables avec un processeur Intel i5 série mobile avec deux coeur et 8Gb de RAM. Nous avons utilisé cet environnement pour développer le contrôleur et l'interface et faire des simples simulations sous Mininet qui ne nécessite pas beaucoup de puissance de traitement.

2.3.2 Serveur d'émulation

Pour notre émulation réseaux sous **GNS3** nous avons opté pour une solution de serveur distant hébergée par **Azure**, avec notre abonnement étudiant nous avons obtenu un serveur D4s_v3 qui est un serveur quad core avec 16gb de RAM et une visualisation imbriquée activée ce qui était important pour nous afin de nous permettre d'utiliser KVM pour notre visualisation. Tout d'abord, nous avons acquis un VPS[10] d'Azure qui a été configuré comme indiqué sur la figure 2.1. Après la connexion à ce serveur avec une clé SSH[11], nous avons installé GNS3 en utilisant la configuration à distance en utilisant la commande suivante :

[8] *Virtual Private Network*
[9] *Open vSwitch Database Management Protocol*
[10] *Virtual Private Server*
[11] *Secure Shell*

```
#!/bin/bash
gns3-remote-install.sh --with-openvpn
```

Cette commande nous permet d'installer GNS3 pour être accédé comme un serveur distant en utilisant une connexion **OpenVPN**. Après la récupération de la clé pour connecter au serveur avec OpenVPN on doit redémarre le serveur pour finalisé l'installation. Après le redémarrage on peu se connecter avec le serveur avec OpenVpn en utilisant la commande suivante :

```
#! /bin/bash
sudo openvpn cle.ovpn
```

Après que la connection a été établit en peut se connecter au serveur GNS3 on utilisant l'interface WEB ou bien avec une installation GNS3 local qui utilise le serveur pour l'émulation.

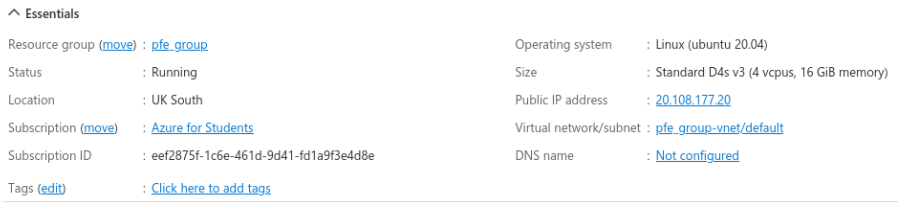


Fig. 2.1 : Configuration du serveur Azure

Chapitre 3

Méthodologie

3.1 Introduction

Dans ce chapitre, nous aborderons la méthodologie et la conception pour notre solution SDN. La solution est divisée en deux parties, premièrement un contrôleur SDN avec une topologie proposée pour faire les tests, deuxièmement une interface d'application WEB qui permet à l'utilisateur de contrôler et visualiser la solutions.

3.2 Présentation du concept générale

La solution de "Software Defined Networking" est basée sur la technologie OpenFlow. Nous avons choisi la version 1.3 de OpenFlow en raison de son large éventail de caractéristiques qui fournissent les fonctionnalités exactes dont nous avons besoin. Nous pensons que l'utilisation d'une version plus récente de OpenFlow crée un obstacle à l'implémentation de notre solution. Pour la conception de notre idée on va définir une adresse virtuelle qui présente le cluster des serveurs, ou bien le service fourni par ces serveurs. L'idée de notre solution est quand un paquet est destiné au cluster avec l'adresse IP du cluster il sera considéré comme un Miss-Flow (Flow avec aucun match), ce Flow sera envoyé au contrôleur. Le contrôleur reçoit ce paquet ensuite il détecte vers quel cluster est destiné puis, effectue le travail nécessaire pour faire l'équilibrage des charges. Le commutateur va mémoriser la décision du contrôleur donc il n'aura pas à faire des étapes non nécessaires.

3.2.1 Réalisation d'une Preuve de concept

Pour mettre en évidence la conception de notre solution on utilisera le scénario suivant : on a un réseau LAN[1] avec cinq hôtes et un commutateur OpenFlow liés à un contrôleur on considère trois de ces hôtes comme des clients qui vont accéder à un service qui est fournit par les deux autres qui sont considérés comme des serveurs. Notre but est de réaliser une solution SDN telle que les requêtes sont distribues entre les deux commutateurs d'une manière autonome et transparente. Les serveurs ne doivent pas connaître qui est en train d'être considérer comme

[1] *Local Area Network*

un cluster et les hôtes ne doivent pas connaître que leurs requêtes sont distribuées entre plusieurs hôtes, la solution doit être transparente.

Réalisation de équilibrage de charge Preuve de concept

Pour réaliser et simuler la solution définit dans 3.2.1, nous avons réalisé un réseau avec Mininet en utilisant le code python suivant.

```
class ProofOfConceptTopo(Topo):
    """ Topology Proof Of Concept """

    def __init__(self, **opts):
        """Create custom topo."""

        # Initialize topology
        super(ProofOfConceptTopo, self).__init__(**opts)
        #Topo.__init__(self)

        # Add hosts and commutateures
        h1 = self.addHost('h1', ip='10.0.0.1', mac="ca:7c:a4:4f:a7:6f")
        h2 = self.addHost('h2', ip='10.0.0.2', mac="9a:02:11:eF:0c:14")
        h3 = self.addHost('h3', ip='10.0.0.3', mac="0a:e6:a2:73:a3:65")
        h4 = self.addHost('h4', ip='10.0.0.4', mac="9e:fd:8f:98:3a:d1" )
        h5 = self.addHost('h5' ,ip='10.0.0.5', mac="2e:31:d0:72:a1:59")
        h6 = self.addHost('h6' ,ip='10.0.0.6', mac="36:d7:8e:e8:7f:60")

        opts = dict(protocols='OpenFlow13')

        # Adding commutateures
        s1 = self.addcommutateur('s1', dpid="0000c671162b8843")

        # Add links
        self.addLink(h1, s1)
        self.addLink(h2, s1)
        self.addLink(h3, s1)
        self.addLink(h4, s1)
        self.addLink(h5, s1)
        self.addLink(h6, s1)
```

Ce code génère un réseau avec 5 hôtes avec les adresses IP 10.0.0.1/24 à 10.0.0.5/24, les trois premiers hôtes sont considérés comme des clients et les deux derniers sont considérés comme des serveurs. on définit l'adresse 10.0.0.100 comme une adresse virtuelle qui représente le service ou le cluster des serveurs. On note qu'aucun hôte n'est configuré avec cette adresse. Notre but est de faire un ping vers cette adresse et recevoir une réponse.

Comme on a mentionné dans 2.2.3, on va utiliser RYU comme un framework pour développer un contrôleur. La première étape est d'ajouter un Flow qui fonction comme une Flow d'un

”Miss-Flow” dans un commutateur quand il connecte au contrôleur celle-ci assure que le commutateur envoie les paquets au contrôleur dans le cas où il n’y a pas de match dans la table des Flows. Pour effectuer cette opération on implémente l’évènement ”ofp_event.EventOFPcommutateurFeatures”. La fonction décorer avec le décorateur de ce évènement sera lancer au moment ou un commutateur connecte au contrôleur.

```
@set_ev_cls(ofp_event.EventOFPcommutateurFeatures, CONFIG_DISPATCHER)
def commutateur_features_handler(
    self, ev : ofp_event.EventOFPcommutateurFeatures):
        datapath : Datapath= ev.msg.datapath


        self.datapaths.append(datapath)
        self.round_robin_indexes[dpid_to_str(datapath.id)] = 0

        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)
```

Quand un client vient de faire un PING vers l’adresse de cluster la première étape est de chercher l’adresse mac relié a cette adresse IP, donc le client envoie une requête ARP qui suit la convention ”Who Has 10.0.0.100”,comme le commutateur n’as aucun Flow qui correspond à cette requête sera renvoyé au contrôleur.

Dans notre code du contrôleur on implémente l’événement PACKET_IN. La fonction décorée avec le décorateur de cette évènement ce lance au moment ou une paquet est envoyé au contrôleur dans cette fonction on implémente la partie qui détecte est traite les requêtes ARP visé au adresse 10.0.0.100. C’est la première étape de notre solution d’équilibrage de charge. Dans le début de notre contrôleur on définit l’ensemble de nos serveurs et leur adresse Mac correspondante comme des Constants.

```
class LoadBalncer(app_manager.RyuApp):
    VIRTUAL_IP = '10.0.0.100'  # The virtual server IP

    SERVER1_IP = '10.0.0.5'
    SERVER1_MAC = '2e:31:d0:72:a1:59'
    SERVER1_PORT = 5
    SERVER2_IP = '10.0.0.6'
    SERVER2_MAC = '36:d7:8e:e8:7f:60'
    SERVER2_PORT = 6
```

Génération du Réponse ARP

Dans le cas où le contrôleur détecte une requête ARP qui cherche l'adresse physique de l'adresse IP 10.0.0.100. l'algorithme sélectionne un des deux serveurs d'une manière aléatoire, le serveur sélectionné sera le serveur qui reçoit les requêtes de l'hôte qui a envoyé la requête ARP. la réponse ARP sera générée par le contrôleur, la requête contient l'adresse Mac du serveur sélectionné et l'adresse IP 10.0.0.100 et envoyé au hôte qui a envoyé la requête ARP, donc pour le client, le cinquième serveur possède l'adresse 10.0.0.100, une requête envoyée par 10.0.0.1 au 10.0.0.100 sera directement envoyé au serveur, il n'existe pas un noeud qui fonctionne comme un équilibreur de charge et qui prend la requête, Pour transmettre le paquet, il est envoyé au commutateur puis directement au serveur sans nœud intermédiaire, c'est le même chemin que n'importe quel paquet suivit avec aucun nœud intermédiaire. C'est la transparence de notre solution.

```
if eth.ethertype == ether_types.ETH_TYPE_ARP:
            ARP_header = pkt.get_protocol(ARP.ARP)

            if (
            ARP_header.dst_ip == self.VIRTUAL_IP
            and
            ARP_header.opcode == ARP.ARP_REQUEST):
                self.logger.info("**************************")
                self.logger.info("---Handle ARP Packet---")
                # Build an ARP reply packet using source IP and source MAC
                reply_packet = self.generate_ARP_reply(
                                        ARP_header.src_ip,
                                        ARP_header.src_mac)

                actions = [parser.OFPActionOutput(in_port)]

                packet_out = parser.OFPPacketOut(
                                    datapath=datapath,
                                    in_port=ofproto.OFPP_ANY,
                                    data=reply_packet.data,
                                    actions=actions,
                                    buffer_id=0xffffffff)

                datapath.send_msg(packet_out)

                self.logger.info("Sent the ARP reply packet")
                return
# ...
# ...
# ...
# ...
```

```
    def generate_ARP_reply(self, dst_ip, dst_mac):
        self.logger.info("Generating ARP Reply Packet")

        ARP_target_ip = dst_ip  # the sender ip

        ARP_target_mac = dst_mac  # the sender mac
        # Making the load balancer IP as source IP
        src_ip = self.VIRTUAL_IP

        if(random.random() > 0.5):
            src_mac = self.SERVER1_MAC
        else:
            src_mac = self.SERVER2_MAC


        pkt = packet.Packet()

        pkt.add_protocol(
            ethernet.ethernet(
                dst=dst_mac,
                src=src_mac,
                ethertype=ether_types.ETH_TYPE_ARP)
        )

        pkt.add_protocol(
            ARP.ARP(
                opcode=ARP.ARP_REPLY,
                src_mac=src_mac,
                src_ip=src_ip,
                dst_mac=ARP_target_mac,
                dst_ip=ARP_target_ip)
        )

        pkt.serialize()
        return pkt
```

L'algorithme précédent fait les actions définies précédemment, dans le cas ou la requête envoyée au contrôleur est une requête ARP qui cherche l'adresse du serveur virtuel, le contrôleur utilise la fonction generate_ARP_reply qui sélectionne un serveur aléatoirement et génère un paquet qui simule la réponse du serveur sélectionner comme s'il occupe l'adresse virtuelle, le paquet est envoyé en broadcast par le commutateur.

Les Flux des paquets TCP

La première étape de notre algorithme d'équilibrage de charge a été déjà fait, maintenant dans l'hôte, l'adresse virtuelle 10.0.0.100 est déjà reliée à l'adresse Mac du serveur sélectionné.

Le paquet envoyé par l'hôte a l'adresse IP 10.0.0.100, le serveur pourra refuser la réception si le paquet n'est pas destiné à lui. Dans l'étape suivante on va configurer le commutateur pour manipuler les paquets déstinés au serveur pour assurer la transparence de l'équilibrage de charge.

Dans le cas où un paquet arrive au contrôleur, si le paquet est un paquet TCP destiné à l'adresse virtuelle du cluster et contient l'adresse mac d'un des serveurs (le serveur sélectionner pour cet hôte), Pour assurer la transparence de l'opération , il faut que le commutateur change l'adresse IP de destination par l'adresse IP de serveur sélectionner, comme ça quand le va recevoir le paquet il va contient sans adresse IP. Dans l'autre sens le commutateur changera l'adresse IP de l'émetteur du paquet par l'adresse virtuelle donc pour l'hôte la réponse a été envoyé par l'adresse 10.0.0.100, cette opération assure la transparence.

Dans cette étape, nous allons profiter de la fonctionnalité de matching avancée et de manipulation des en-têtes des paquets du protocole OpenFlow, nous allons écrire un nouveau Flow dans la table de Flow du commutateur qui correspondra à tout paquet qui sera envoyé de cet hôte au serveur virtuel. Dans ce cas, tous les paquets futurs envoyés de cet hôte spécifique au serveur virtuel seront gérés par ce Flow et n'auront pas besoin d'être interférés par le contrôleur. Le code suivant s'occupe de cette opération.

```
    # Handle TCP Packet
        if eth.ethertype == ETH_TYPE_IP:
            self.logger.info("****************************")
            self.logger.info("---Handle TCP Packet---")
            ip_header = pkt.get_protocol(ipv4.ipv4)

            packet_handled = self.handle_tcp_packet(datapath, in_port, ip_header,
                                                    parser, dst_mac, src_mac)
            if packet_handled: return

    def handle_tcp_packet( self, datapath, in_port, ip_header,
                           parser, dst_mac, src_mac ):
        packet_handled = False
        if ip_header.dst == self.VIRTUAL_IP:
            if dst_mac == self.SERVER1_MAC:
                server_dst_ip = self.SERVER1_IP
                server_out_port = self.SERVER1_PORT
            else:
                server_dst_ip = self.SERVER2_IP
                server_out_port = self.SERVER2_PORT

            match = parser.OFPMatch( in_port=in_port, eth_type=ETH_TYPE_IP, ip_proto=
                                     ipv4_dst=self.VIRTUAL_IP )

            actions = [parser.OFPActionSetField(ipv4_dst=server_dst_ip),
                       parser.OFPActionOutput(server_out_port)]

            self.add_flow(datapath, 20, match, actions)
```

```
            # Reverse route from server
            match = parser.OFPMatch(
                                in_port=server_out_port, eth_type=ETH_TYPE_IP,
                                ip_proto=ip_header.proto,
                                ipv4_src=server_dst_ip,
                                eth_dst=src_mac )

            actions = [parser.OFPActionSetField(
                                ipv4_src=self.VIRTUAL_IP),
                       parser.OFPActionOutput(in_port)]

            self.add_flow(datapath, 20, match, actions)

            packet_handled = True
        return packet_handled
```

Test Initial du Preuve de concept

Dans cette section on va tester notre Preuve de concept. Pour un test initial qui assure que les Ping entre les hôtes et l'adresse virtuelle des serveurs fonctionne, pour cela on va utiliser Mininet.

D'abord on crée la topologie définie dans la section 3.2.1 avec la commande

```
sudo Python3 topology.py
```

Après on va tester le ping avec cette commande et le ping réussit.

```
ping 10.0.0.100
```

Émulation et analyse de preuve de concept

Dans cette étape on va émuler le même réseau définit précédemment avec GNS3 dans la figure 3.1.

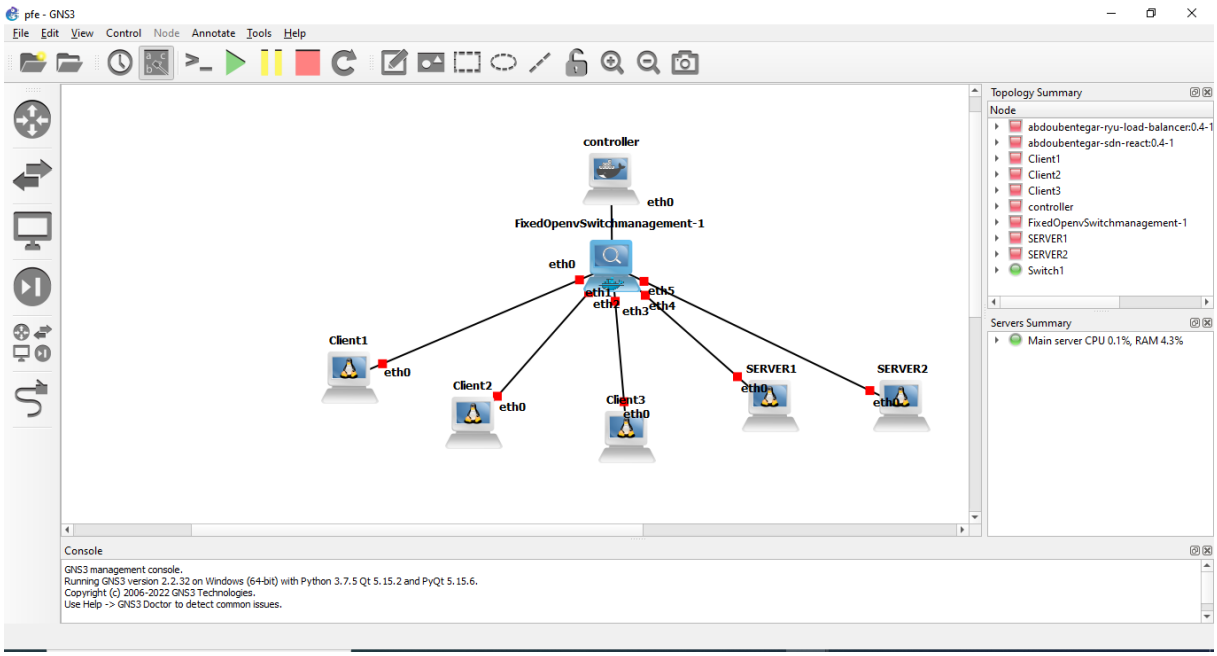


Fig. 3.1 : Réseau dans GNS3

D'abord on a initialiser le réseau et les adresses IP, Ce réseau est constitué de conteneurs Docker suivant :

- **OpenvSwitch with management** : Une image docker qui implémente OpenvSwitch avec une interface management sur le port 0, ce commutateur est déjà configuré avec OpenFlow pour tous les ports sauf le port 0 qui est utilisé pour le management du commutateur.

- **Ubuntu Docker Guest** : Une image docker qui implémente une version léger de Ubuntu.

On a installé Python et Ryu sur le noeud qui sera considéré comme le contrôleur. On a télécharger le script du contrôleur avec Git et Github et on a lancé le script du contrôleur. On a configuré le commutateur avec le contrôleur sur l'adresse 10.0.0.20 et vérifier la connectivité comme suit :

```
ovs-vsctl set-controller br0 tcp:10.0.0.20:6633

ovs-vsctl show

77b326d9-2c77-4968-80de-ec837e74fe98
   Bridge "br0"
       Controller "tcp:10.0.0.20:6633"
           is_connected: true
```

Avec ces étapes de préparation, on est prêt pour tester et inspecter le fonctionnement d'équilibrage de charge, On a utilisé Wireshark pour capturer les paquets qui sont sur les liens entre le commutateur et l'hôte 1, et les deux serveurs, ensuite on a testé le Ping.

Interprétation des paquet avec WireShark

Dans cette étape, on va vérifier que les étapes qu'on a prévue précédemment fonctionnent correctement. Quand on lance la commande du Ping , La première requête envoyée est une requête ARP qui cherche l'adresse Mac 10.0.0.100 à laquelle le contrôleur doit répondre par une réponse ARP contenant l'une des adresses Mac des serveurs.
Dans le figure 3.2, cette capture d'écran assure que le serveur 1 a été choisi pour répondre aux requêtes de l'hôte 1, et donc son adresse Mac est dans la réponse ARP.

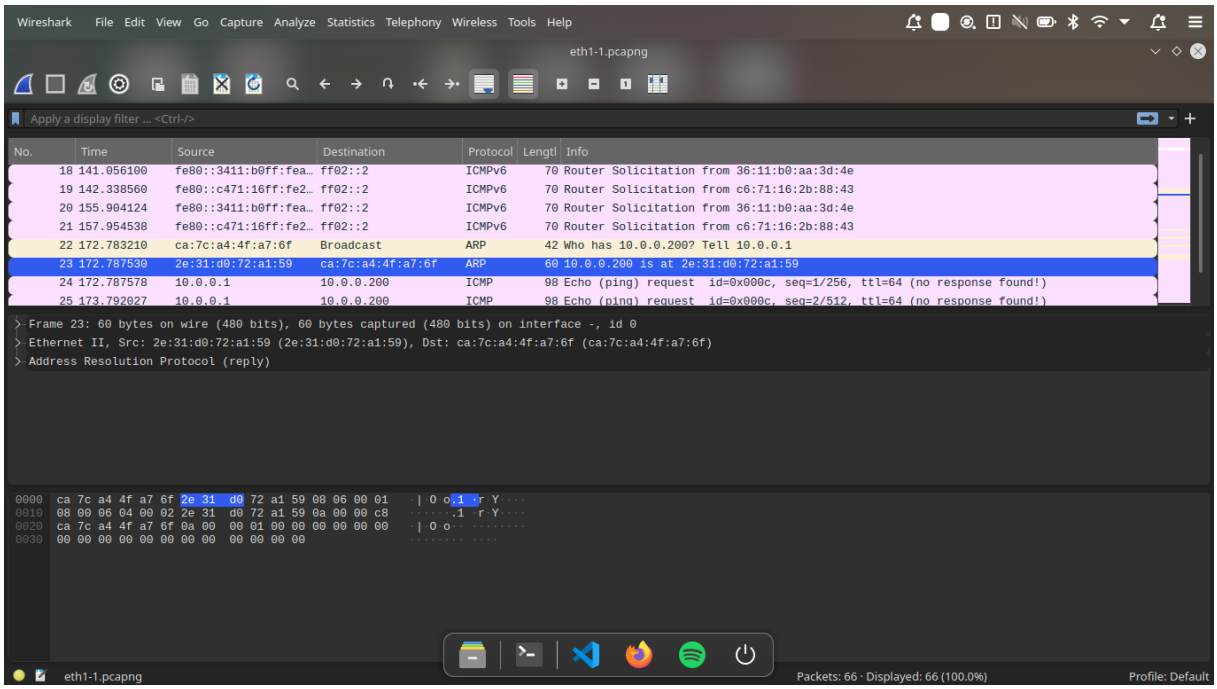


Fig. 3.2 : Capture des requête ARP avec wireshark

Après on suit les paquets ICMP envoyés entre le serveur 1 et le client 1 dans les deux liaisons. Dans la figure 3.3 et 3.4 on voit les captures des paquets entre le commutateur et l'hôte, et aussi entre le commutateur et le serveur.

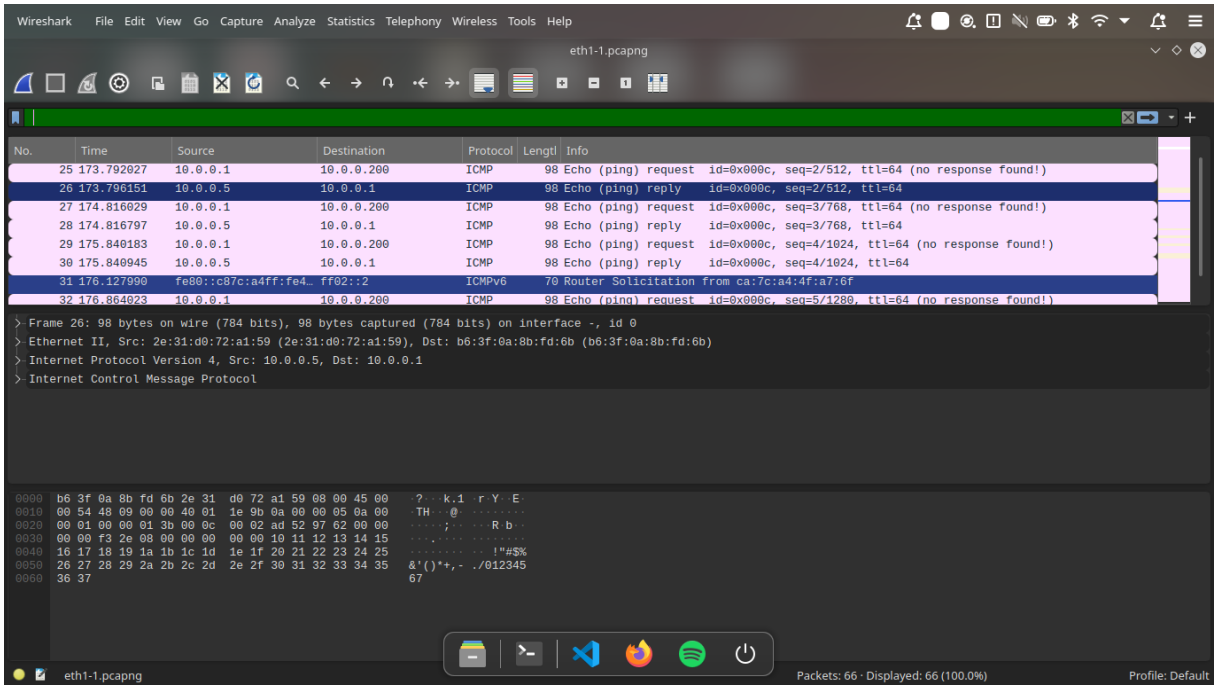


Fig. 3.3 : Capture des requête TCP entre le commutateur et l'hôte 1

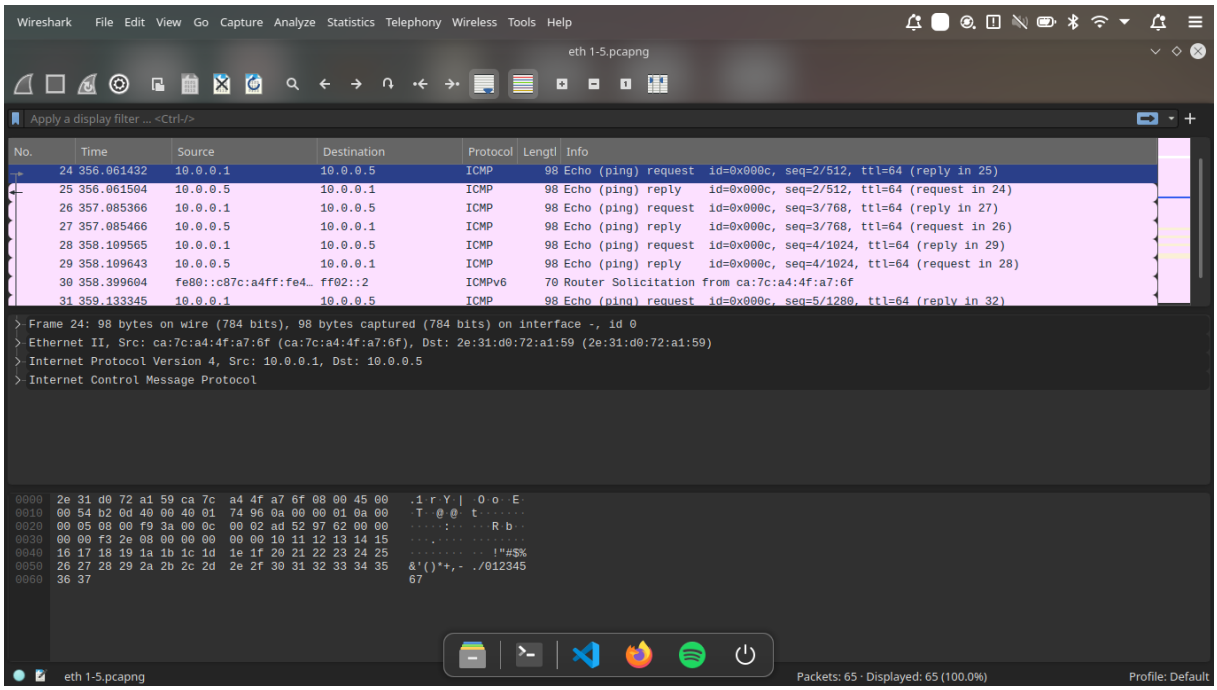


Fig. 3.4 : Capture des requête TCP entre le commutateur et le serveur 1

Dans la figure 3.3 qui représente le capture des paquets avec Wireshark entre le commutateur et le client 1 qui procède l'adresse IP 10.0.0.1 on remarque que les paquet ICMP et leur réponse ont les adresses du client et l'adresse virtuelle du serveur. De l'autre côté dans la figure 3.4 qui représente le capture des paquet entre le commutateur et le serveur 1 qui a été sélectionné pour répondre au requêtes du client 1 on remarque que dans les mêmes paquets envoyé du client 1 a l'adresse virtuelle, les entêtes IP ont été changé a l'adresse du serveur sélectionné.

Inspection du table des Flow

La dernière étape dans notre vérification du bon fonctionnement de notre concept et de vérifier les Flow qui sont dans le commutateur, pour vérifier les Flow on exécuter cette commande et nous obtenus les résultats ce dessus.

```
ovs-ofctl dump-flows br0
```

```
cookie=0x0, duration=203.854s, table=0, n_packets=0, n_bytes=0, priority=65535,
dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535

cookie=0x0, duration=203.861s, table=0, n_packets=15, n_bytes=1118, priority=0
actions=CONTROLLER:65535

cookie=0x0, duration=166.908s, table=0, n_packets=18, n_bytes=1764, priority=15,
ip,in_port=eth1,nw_dst=10.0.0.100 actions=mod_nw_dst:10.0.0.5,output:eth5

cookie=0x0, duration=166.908s, table=0, n_packets=18, n_bytes=1764, priority=13,
ip,in_port=eth5,dl_dst=ca:7c:a4:4f:a7:6f,nw_src=10.0.0.5,
actions=mod_nw_src:10.0.0.100,output:eth1

cookie=0x0, duration=166.908s, table=0, n_packets=1, n_bytes=42, priority=10,
in_port=eth1,dl_src=26:f2:f6:d6:ea:97,dl_dst=72:27:aa:c4:0e:1c actions=output:eth5

cookie=0x0, duration=160.683s, table=0, n_packets=0, n_bytes=0, priority=10,
in_port=eth5,dl_src=72:27:aa:c4:0e:1c,dl_dst=26:f2:f6:d6:ea:97
actions=output:eth1
```

On remarque qu'il y a six Flux enregistres dans le commutateur :

- Le premier Flux permet d'envoyer le paquet au contrôleur avec une priorité 0 donc si aucun Flux n'est trouvé pour le paquet.
- Le troisième Flux s'occupe des paquets envoyés au serveur 1 par le client 1, le Flux spécifie un match sur le port d'où le paquet arrive avec le port "eth1" et l'adresse IP de destination avec celle du serveur virtuel. s'il y a un match, le commutateur change l'adresse IP de destination avec celle de serveur 1.
- Le quatrième Flux s'occupe du paquet envoyé au client 1 par le serveur 1, le Flux spécifie un match sur l'adresse mac de destination avec l'adresse mac du client 1, et l'adresse IP source avec celle du serveur 1, s'il y a un match, le commutateur change l'adresse IP de destination avec celle du serveur virtuel.
- Les autres Flows fonctionnent comme une table Mac de commutateur, chaque adresse Mac de destination est lié à un port de sortie.

3.2.2 Conclusion

Après l'inspection du fonctionnement de notre solution, on a confirmé le comportement de la solution ainsi que les paquets envoyer entre les clients et le serveur. On a conclu que notre concept fonctionne comme prévu et répond à notre besoin. Dans la section suivante, nous allons améliorer ce concept pour le rendre plus convivial et gérer plusieurs serveurs et équilibreurs de charge.

3.3 Amélioration d'équilibrage de charge

3.3.1 Introduction

Après qu'on a prouvé le bon fonctionnement de notre solution, on passe vers les étapes suivantes. Comme on a mentionné dans , nous avons besoin d'une solution adaptative, donc on doit ajouter une méthode pour contrôler l'équilibreur de charge. nous avons besoin d'une méthode pour visualiser et gérer la solution, donc on a besoin d'une interface graphique. On a aussi besoin de chercher la meilleure méthode pour délivrer une solution portable et facile à installer.

Support du multiple équilibreurs de charge et serveurs

Dans la première section, nous avons créé une solution qui ne fonctionne que sur le réseau que nous avons défini précédemment. Pour améliorer l'extensibilité de notre réseau, nous allons ajouter la fonction de la découverte du réseau et la liberté d'ajouter et de supprimer des serveurs au cluster de l'équilibreur de charge.

Conception de base des donnés :

Pour implémenter une solution avec plusieurs équilibreurs de charge et serveurs, chacun avec sa propre méthode d'équilibrage. On va ajouter une base des données simples pour sauvegarder les équilibreurs de charge et leurs informations ainsi que leurs serveurs associés.

Dans notre base des données on aura trois entité :

- **LoadBalancer :** Représente un équilibreur de charge dans le réseau, pour que le commutateur OpenvSwitch soit un équilibrer de charge il faut que la fonctionnalité d'équilibrage de charge soit activer, pour activer cette fonctionnalité dans un commutateur il suffit d'ajouter une entrée dans le tableau du "LoadBalancers" dans la base de données en indiquant sont DPID[2] , la méthode d'équilibrage utilisée et l'adresse virtuelle du cluster des serveurs associer avec ce commutateur.

  Avec cette entité on peut chercher un équilibreur de charge et récupérer la liste des serveurs qui est gérée par cet équilibreur de charge, et obtenir la méthode par laquelle on va faire l'équilibrage et choisir le serveur désiré.

[2] *Depository Participant Identity*

- **Server :** Représente un serveur lié a un équilibreur de charge pour chaque serveur on enregistre l'adresse MAC du serveur, l'adresse IP du serveur, le port de commutateur par lequel le serveur est connecté et le DPID de l'équilibreur de charge lié avec ce serveur comme une clé étrangère. On a une relation père-fils entre le serveur et l'équilibreur de charge, donc chaque serveur a un seul équilibreur de charge qui le gère et l'équilibreur de charge gère plusieurs serveurs.
- **Migration :** Cette entité n'est pas nécessairement liée avec notre méthode d'équilibrage de charge, par contre elle assure la disponibilité de la base de données et sa structure et la possibilité du développement continu de notre solution. On définit une migration comme un changement de structure de la base de données. Et cela pour assurer la création automatique de notre base de données, le bon structure de notre base de données et la traçabilité des changements à la structure de la base de données pendent le développement.la figure 3.5 illustre le diagramme UML de la base des données .

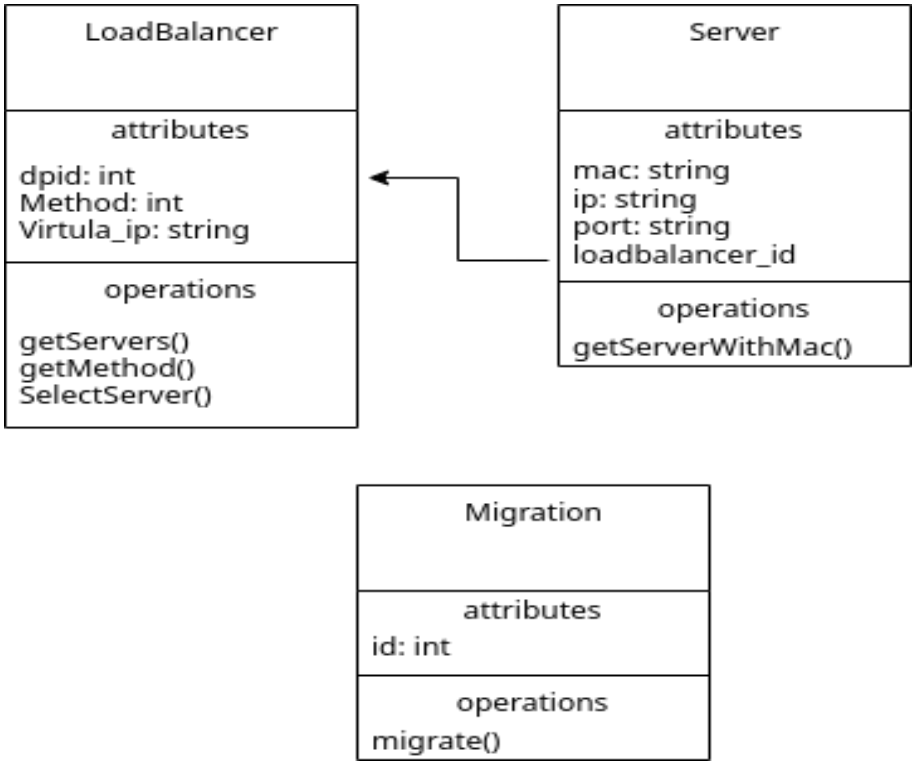


Fig. 3.5 : Diagramme UML de la base des données

Migration de la base de données

Lors de la première exécution du contrôleur, une base de données sera créée automatiquement, après la création de la base de données, elle sera automatiquement remplie avec les tables nécessaires, un enregistrement sera ajouté dans la base de données pour suivre ce changement dans la structure de la base de données, au cas où une mise a jour est faite à notre application ou il est nécessaire de changer la base de données, le contrôleur va vérifier les derniers changements dans la base de données et continuer la migration de la base de données avec les changements nécessaires sans faire aucun travail destructif sur les données de la base de données.

Cette fonctionnalité assure un des principes de notre application, la portabilité et la facilite d'installation et de maître à jour notre solution, la migration joue un rôle important dans le déploiement de notre solution.

3.3.2 Les méthodes de sélection du serveur

Après qu'on a prouvé notre concept d'équilibrage de la charge avec un commutateur OpenFlow. Notre approche initiale était de choisir la destination du paquet au hasard avec une méthode aléatoire, c'était juste une preuve de notre concept. Dans cette section on va définir les autres méthodes ainsi que leur implémentions dans notre solution.

Méthode Round-Robin :

Dans 1.4.3 on a définit la méthode de round robin, le principe est de grouper les serveurs liés a un équilibreur de charge du Round-Robin et envoyer les requêtes au serveur tour a tour un après l'autre.

Dans notre algorithme pour établir cette fonctionnalité on a défini un dictionnaire qui a une paire clé-valeur qui relie chaque DPID de commutateur avec l'index actuel de la méthode round-robin, cet index représente l'index du serveur suivant dans le tableau des serveurs lié à cet équilibreur de charge. Lorsqu'un commutateur est connecté et activé comme équilibreur de charge, un élément est ajouté à ce dictionnaire avec le DPID du commutateur comme clé et 0 comme valeur. Dans ce cas, lorsqu'une requête est envoyée à l'adresse IP virtuelle de cet équilibreur de charge, le serveur avec l'index 0 dans la liste des serveurs lie à cet équilibreur de charge est sélectionné, la valeur dans le dictionnaire est incrémentée, ainsi à la prochaine requête, le serveur avec l'index 1 sera sélectionné. Lorsque la valeur dans le dictionnaire dépasse la longueur de liste des serveurs, il sera réinitialisé à 0. Le code ci dessus représente l'implémentation de cette méthode dans notre contrôleur.

```
class LoadBalncer(app_manager.RyuApp):
    ....

    round_robin_indexes = {}

    def chose_server_round_robin(self ,
        servers : List[Server] , load_balancer : LoadBalancer) -> Server:

        size = len(servers)
        if (not size ):
            return False

        next_index = (self.round_robin_indexes[load_balancer.datapath] + 1) % size

        self.round_robin_indexes[load_balancer.datapath] = next_index

        return servers[self.round_robin_indexes[load_balancer.datapath]]
```

Méthode de hachage de l'adresse MAC

Dans 1.4.3 on a défini la méthode de hachage IP. Dans notre implémentations on va utiliser le même principe mais avec l'adresse MAC. Cette méthode assure que chaque client sera connecter au même serveur pour toutes ces requêtes, cette fonctionnalité assure la continuité de connexions. Pour l'implantation on a choisi une simple fonction de hachage qui transforme l'adresse Mac de l'hexadécimale vers la décimale et prend le reste d'une division de l'adresse Mac sur le nombre des serveurs associé à l'adresse virtuelle, le résultat sera l'index du serveur qui gère les requêtes de ce client. Le code ci dessus représente l'implémentation de cette méthode dans notre contrôleur.

```
    def chose_server_mac_hash(self,mac: str,
        servers :List[Server]) -> Server:
            int_mac = haddr_to_int(mac)
            server_count = len(servers)

            server_index = int_mac % server_count
            return servers[server_index]
```

Méthode basée sur la charge du serveur

Cette méthode montre la véritable puissance du SDN, dans cette méthode le contrôleur est capable d'évaluer la charge qui est appliquée sur les serveurs dans cet espace et de réagir en choisissant le serveur le moins chargé avec la plus grande puissance de traitement disponible pour traiter cette demande. Cela montre la programmabilité, l'autonomie et la flexibilité d'un répartiteur de charge SDN.

Pour cette implémentation , nous avons configuré un simple serveur TCP sur chacun de nos

serveurs, ce serveur TCP écoute une demande du contrôleur pour répondre avec la charge CPU actuelle. D'autre part, le contrôleur se connecte via TCP à chacun des serveurs via un autre script simple, enfin le serveur qui a la charge la plus faible et donc la plus grande puissance de traitement disponible a été sélectionné.

Le script dans le contrôleur

```
def get_server_with_load(self,servers: List[Server]):
    min = 101
    index = 0
    for i ,server in enumerate(servers):
        HOST = server.ip
        PORT = 2345
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect((HOST, PORT))
            s.sendall(b"Hello, world")
            data = s.recv(1024)

        cpu = float(data.decode())
        if(cpu < min):
            min = cpu
            index = i

    return servers[i]
```

Le script dans le serveur

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind(("0.0.0.0" , 2345))
    s.listen()
    while True:
        conn, addr = s.accept()
        with conn:
            while True:
                data = conn.recv(1024)
                a = psutil.cpu_percent()
                if not data:
                    break
                conn.sendall(str.encode(str(a)))
```

3.3.3 Encapsulation de la méthodologie

Après l'explication de notre méthode, Dans cette section, nous allons encapsuler notre méthode d'équilibrage de charge dans les SDN avec des diagrammes explicatifs.

Diagramme d'activité

Un diagramme d'activités visualise un graphe d'activité qui modélise le comportement interne d'une méthode (la réalisation d'une opération), d'un cas d'utilisation ou plus généralement d'un processus impliquant l'utilisation d'un ou de plusieurs classificateurs.

- **Diagramme d'activité pour une requête ARP :** Dans ce diagramme on a représenté les étapes qui passe lors d'une requête ARP de la part du client à la cherche de l'adresse Mac de l'adresse IP virtuelle du cluster des serveurs. On note qu'il y a une condition au début de cette procédure si le client a déjà reçu une adresse MAC pour l'adresse IP virtuelle du cluster, donc cette requête se produire juste avant la première requête, donc la procédure ne sera pas prise en compte dans les communications à venir et la latence sera meilleure.La figure 3.6 illustre le diagramme d'activité d'une requête ARP .

- **Diagramme d'activité pour une requête TCP :** Dans ce diagramme on présente les étapes qui passe lors d'une requête TCP. Premièrement on note que cette étape se passe après la vérification d'existence d'une entrer pour l'adresse virtuelle dans la table ARP du client. Par contre, on note que le paquet ne sera pas envoyé au contrôleur que s'il y a un "FLOW TBALE MISS", pour que le contrôleur ajoute les Flows dans le commutateur , donc la procédure ne sera pas prise en compte dans les communications à venir et la latence sera meilleur. La figure 3.7 illustre diagramme d'activité du requête TCP .

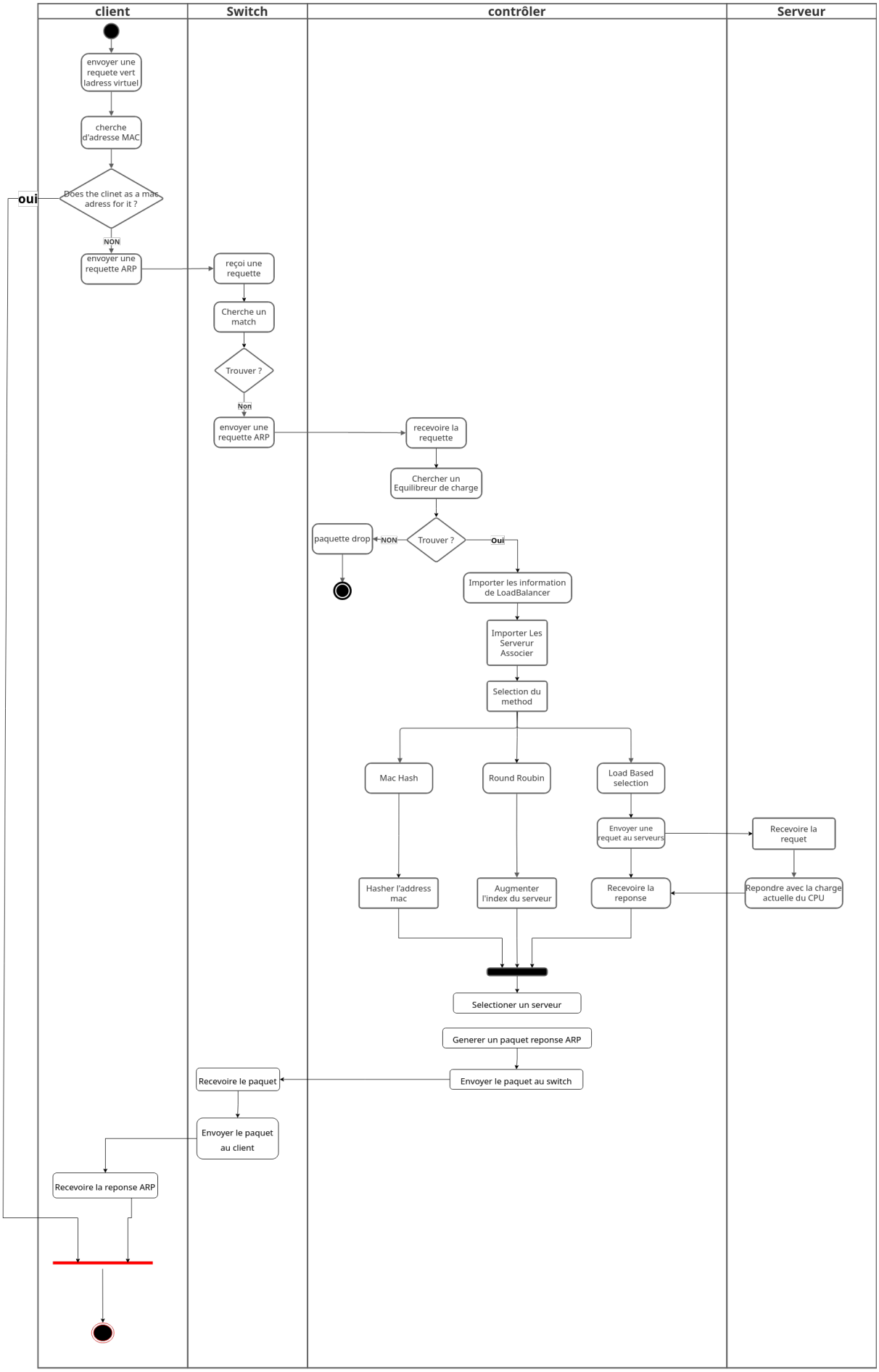


Fig. 3.6 : Diagramme d'activité d'une requête ARP

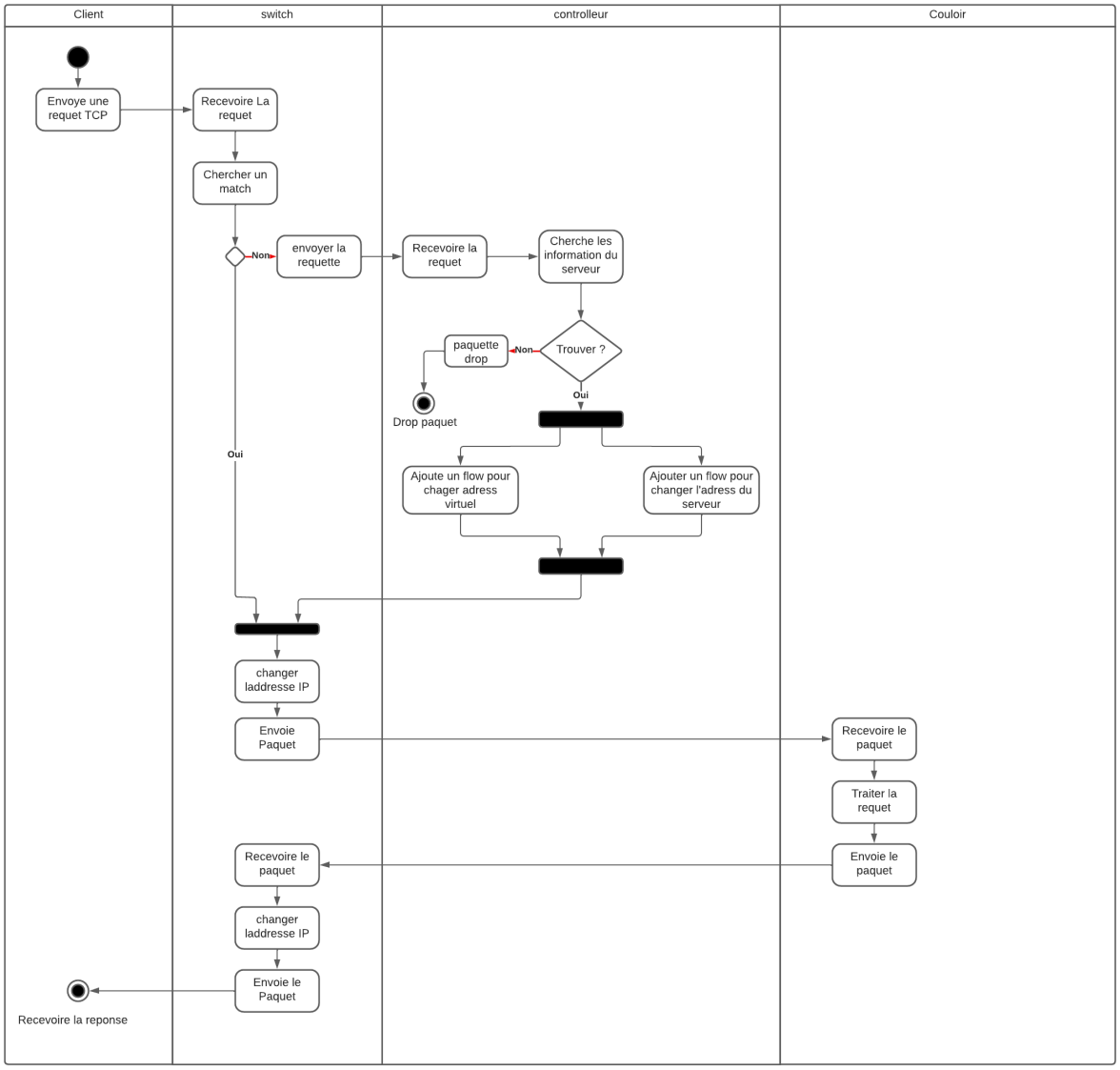


Fig. 3.7 : Diagramme d'activité du requête TCP

D'après ces deux diagrammes, on peut noter que la partie qui est considérée comme le goulot de bouteille de notre solution n'est exécutée qu'une seule fois pour chaque client. C'est une optimisation qui diminue énormément la latence de notre solution.

3.4 L'implémentation d'un North Bound API

Après s'être assuré le bon fonctionnement et l'autonomie de notre solution, dans cette section on va fixer le problème de la programmabilité de notre solution. On doit fournir une solution aux développeurs pour interfacer le contrôleur. Notre objectif est de fournir un moyen pour l'administrateur réseau et les développeurs de communiquer avec le contrôleur via un protocole standard. Dans l'architecture SDN, cela se fait via le Northbound API.

Pour avoir un objectif raisonnable à atteindre dans les délais impartis, nous avons fournir aux utilisateur de notre application les fonctionnalités suivantes :

- Avoir un serveur HTTP standard qui peut être appelé depuis n'importe quel endroit du réseau et fournir des données via JSON[3].
- Fournir des informations sur la topologie du réseau, y compris les commutateurs, les hôtes et les liens. Fournir aussi la possibilité d'ajouter et de supprimer des fonctionnalités d'équilibrage de charge vers et depuis les commutateurs tout en sélectionnant l'adresse IP virtuelle et la méthode d'équilibrage de charge.
- Fournir la possibilité d'ajouter et de supprimer un nœud au cluster des serveurs.

3.4.1 Implémentation d'un serveur HTTP :

Le cadre qu'on a choisi pour développer notre application fournit un outil simple pour implémenter un serveur REST http en utilisant WSGI[4]. La façon de l'implémenter dans RYU est de créer une classe contenant la bibliothèque WSGI dans le variable _CONTEX et peut accéder à l'instance de la classe du contrôleur principal pour la gestion des données. Dans cette classe on a l'accès aux décorateurs qui lient la fonction à un point de terminaison URL[5] sur le serveur HTTP.

```
class LoadBalncerRest(ControllerBase):
    _CONTEXTS = { 'wsgi': WSGIApplication }
    def __init__(self, req, link, data, **config):
        super(LoadBalncerRest, self).__init__(req, link, data, **config)
        self.load_balancer_app : LoadBalncer  = data[load_balancer_instance_name]

    # un exemple d'un route avec un decorateur
    @route('loadbalancer' ,  '/v1/loadbalancer/servers' , methods=["OPTIONS"])
    def _response_function(self,req,**kwargs):
        response = Response(content_type=content_type,
                                body={"ok" : "ok"}, status=status)
        return response
```

3.4.2 Solution du problem CORS

Nous voulons que notre API HTTP Northbound puisse être appelée depuis n'importe quel endroit avec toutes les méthodes possibles. Pour se faire, nous devons résoudre le problème CORS.

Definition CORS

Le "Cross-Origin Resource Sharing" (CORS) ou "Partage de ressources entre plusieurs origines" est un mécanisme qui consiste à ajouter des en-têtes HTTP pour permettre aux agents utilisateurs d'accéder aux données d'un serveur situé sur une autre origine que le site courant[12].

[3] *JavaScript Object Notation*
[4] *Web Server Gateway Interface*
[5] *Uniform Resource Locator*

Pour notre cas, nous voulons présenter la console et l'interface Web séparément, dans une implémentation simple sans configurations supplémentaires.

Fonctionnement de CORS

La norme CORS fonctionne en ajoutant de nouveaux en-têtes HTTP qui permettent aux serveurs de décrire un ensemble d'origines autorisées pour lire des informations à partir d'un navigateur Web[12] .Pour les méthodes de requête HTTP il y a deux types,la première appelé les requêtes simples qui nécessitent pas de requête préliminaire comme GET et POST, et la deuxième appelée les requêtes nécessitant une requête préliminaire qui envoie d'abord une requête HTTP avec la méthode OPTION comme PUT et DELETE [12].

Solution Propose

Pour résoudre ce problème, nous avons défini une fonction d'aide qui prend le corps de la réponse et crée un objet réponse puis ajoute les en-têtes nécessaires montrés dans le script ci-dessous.

```
def create_response(body, status=200, content_type='application/json'):
    r = Response(content_type=content_type,
                                 body=body, status=status)
    r._headerlist.append(('Access-Control-Allow-Origin', '*'))
    r._headerlist.append(('Access-Control-Allow-Methods', '*'))
    r._headerlist.append(('Access-Control-Max-Age', 86400))
    r._headerlist.append(('Access-Control-Allow-Headers' , '*'))

    return r
```

3.4.3 Définition des routes

Les route Topologies

- **Commutateur :** Cette route est localisée sous l'URL "/v1.0/topology/switches" elle permet de récupérer avec une requête de méthode GET la liste des commutateurs avec leur DPID ainsi que les informations de ses ports qui incluent le nom du port et l'adresse MAC.

- **Hôtes :** Cette route est localisée sous l'URL "/v1.0/topology/hosts" elle permettre avec une requête de méthode GET de récupérer la liste des hôtes avec leur adresses MAC, IPv4 et IPv6 ainsi que les informations de ses ports qui incluent le nom du port et l'adresse MAC.

- **Liens :** Cette route est localisée sous l'URL "/v1.0/topology/hosts" elle permettre avec une requête de méthode GET de récupérer les liaisons qui existe dans le réseau.

Les routes équilibreurs de charge

- **Les équilibreurs de charge :** Les routes associées aux équilibreurs de charge sont localisés sous l'URL "/v1/loadbalancer/loadbalancers" avec les méthodes suivants :

  - **GET équilibreur de charge :** avec la méthode GET on peut récupérer la liste des équilibreur de charge avec leur adresse virtuelle, méthode d'équilibrage et DPID.
  - **POST équilibreur de charge :** avec la méthode POST on peut ajouter un équilibreur de charge. Cette requête active la fonctionnalité d'équilibrage de charge dans un commutateur qui existe dans notre topologie en indiquant son DPID, adresse virtuelle et méthode d'équilibrage dans le corps de la requête.
  - **DELETE équilibreur de charge :** Avec la méthode DELETE on peut supprimer un équilibreur de charge en indiquant son DPID dans l'URL. Cette requête désactive la fonctionnalité d'équilibrage de charge dans un commutateur qui existe dans notre topologie, donc elle supprime l'entrée de la base de données et supprime les Flow dans le commutateur, donc elle le rend comme un commutateur simple qui ne fait pas l'équilibrage de charge.

- **Les serveur :** les routes associées aux serveurs sont localisées sous l'URL "/v1/loadbalancer/servers" avec les méthodes suivantes :

  - **GET serveur :** avec la méthode GET on peut récupérer la liste des serveurs dans la topologie avec leur port, adresse IP et MAC.
  - **POST serveur :** avec la méthode POST on peut ajouter un serveur au cluster des serveurs en indiquant le DPID d'équilibreur de charge qui est associé avec, son port et son adresse IP et MAC.
  - **DELETE serveur** avec la méthode DELETE on peut supprimer un serveur de cluster des serveurs en indiquant son adresse MAC dans l'URL .

3.5 Le développement d'une interface graphique

l'étape suivante consiste à développer une interface graphique Web afin de fournir un moyen plus convivial de visualiser la topologie du réseau et de gérer le cluster des serveurs et les équilibreurs de charge. Notre vision pour cette interface Web est qu'elle sera packagée séparément du contrôleur, elle sera donc considérée comme un add-on pour notre solution.

Dans cette section, nous parlerons du processus de développement de cette partie de la solution, nous nous concentrerons sur la communication entre le SouthBound API du contrôleur et cette application car nous pensons que c'est le moyen le plus intéressant. Il n'aura pas trop de détail sur l'affichage. Cette application sera développée en TypeScript avec la bibliothèque react-js précédemment définit dans la section 2.2.3 .

3.5.1 Initiation du projet

Vite

La première étape est de générer notre projet avec Vite. On va utilise Vite comme un outil de ”build” de notre application, cette outil va compiler notre code Ts, Jsx et les autres bibliothèque vers des fichiers HTML et JS qui vont être consommés par le navigateur. La première étape est de générer notre projet avec Vite.

React Router

Notre application est une application SPA donc il n’y aura pas d’actualisation entre les différentes pages. La navigation sera prise en charge par la bibliothèque ”React Router”, cette bibliothèque est spécifiquement important dans nos Flows des données car on l’utilise pour récupérer des paramètres encoder dans l’URL

Zustand Global state

Cette bibliothèque est utilisé pour garder l’état global de notre application. Dans cet état on crée des ”store” qui garde les informations des différente données de notre application et qui peuvent être utilisés globalement dans notre application Web et partagés entre les composants, ci-dessus on peut voir un exemple d’un store défini par Zustand qui englobe les données de topologie.

```
import create from 'zustand'
import { ServerInterface } from '../types/Topology'

interface ServerStore {
  servers: ServerInterface[]
  setServers: (servers: ServerInterface[]) => void
  getIsServer: (mac: string | undefined) => boolean
  getServer: (mac: string) => ServerInterface | undefined
}

const useServersStore = create<ServerStore>((set, get) => ({
  servers: [],
  setServers: servers =>
    set(state => ({
      ...state,
      servers: servers,
    })),
  getIsServer: mac =>
    get().servers.findIndex(server => server.mac === mac) > -1,
  getServer: mac => get().servers.find(server => server.mac === mac),
}))
export default useServersStore
```

3.5.2 Visualisation de Topologies

Dans cette partie nous allons expliquer comment notre interface communique avec l'API de contrôleur pour récupérer les informations du topologie et comment les afficher.

Requête au contrôleur

La première étape est de faire un requête HTTP vers le contrôleur avec les URL pour récupérer les commutateurs, les hôtes et les liens on a définit des fonctions spéciales à la bibliothèque React appeler des Hook pour récupérer ces données, ces fonctions sont :

- useAxiosGet : C'est une fonction qui envoie une requête HTTP à une URL de récupération de données, et cette fonction a la possibilité de redémarrer la requête à un intervalle pour actualiser les données .

- useTopologyApi : cette fonction utilise la première fonction pour récupérer les informations de la topologie qui inclue les commutateurs, les hôtes, les équilibreurs des charges et les liens entre eux.

Traitement des informations et visualisation

Pour la visualisation des données on a choisi une bibliothèque qui accepte les données d'un format spécifique. On doit reformuler les données de la topologie récupérer à partir du contrôleur vers deux listes, une liste des noeuds qui englobe les hôtes, commutateurs et les équilibreurs de charge chacun avec ses informations. Et une liste qui contient les liens. Les résultats de cette fonction sont entré au composant qui va afficher la topologie dans notre interface avec des composants qu'on a développés pour réagir au changement de notre topologie.

Dans le figure 3.8 on voie la topologie dans notre interface Web.

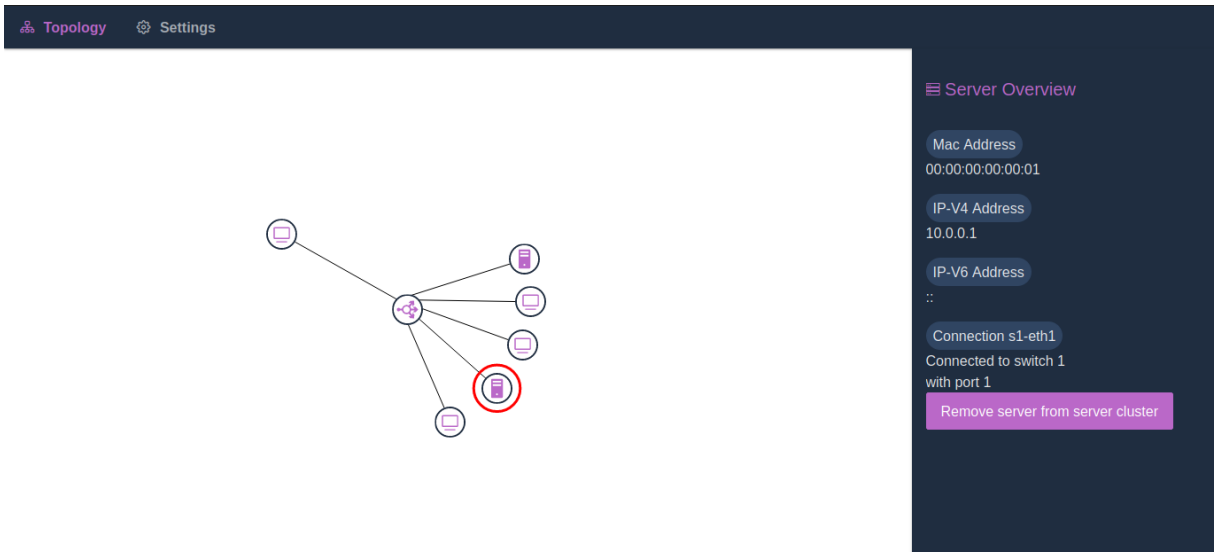


Fig. 3.8 : Visualisation du topolo

3.5.3 Gérance des équilibreurs de charge et serveurs

Dans cette partie on va démontrer comment gérer les équilibreurs de charge avec notre interface graphique. Pour activer la fonctionnalité d'équilibrage de charge à un commutateur il suffit d'envoyer une requête HTTP au contrôleur qui contient sont DPID , l'adresse virtuelle et la méthode d'équilibrage de charge.

Pour ajouter un serveur a un cluster des serveur il suffit d'envoyer une requête HTTP au contrôleur qui contient le DPID d'équilibreur de charge qui est associé avec, son port et son adresse IP et MAC.

Le figure 3.9 présente la procédure d'ajouter un équilibreur de charge avec notre interface Web.

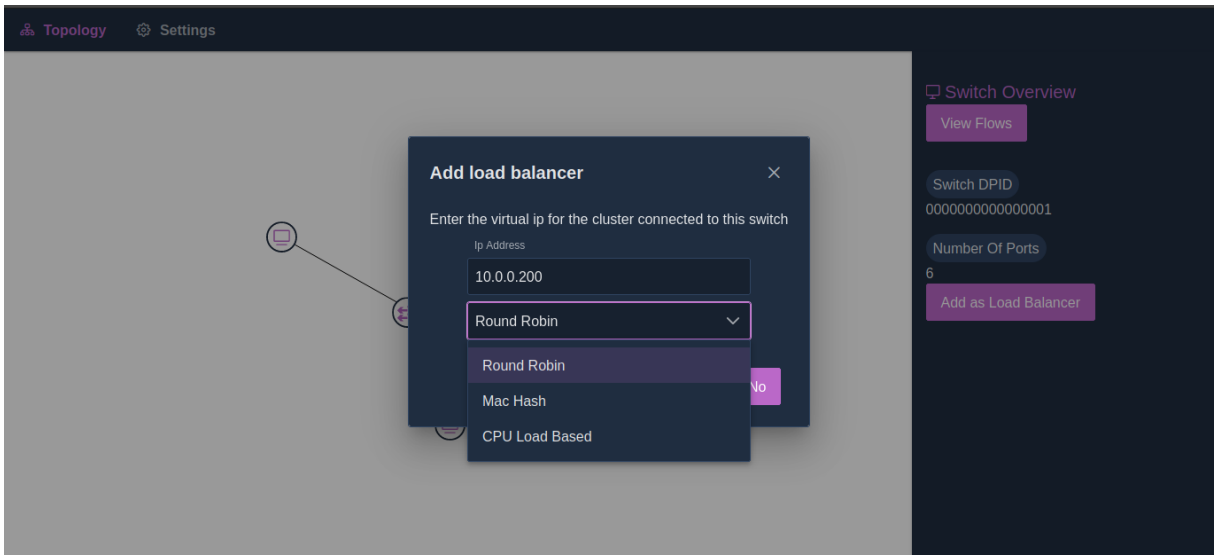


Fig. 3.9 : Visualisation du topologie dans l'interface Web

3.5.4 Affichage des Flow

Avec notre interface Web on peut visualise la table des Flow d'un commutateur dans notre topologie.

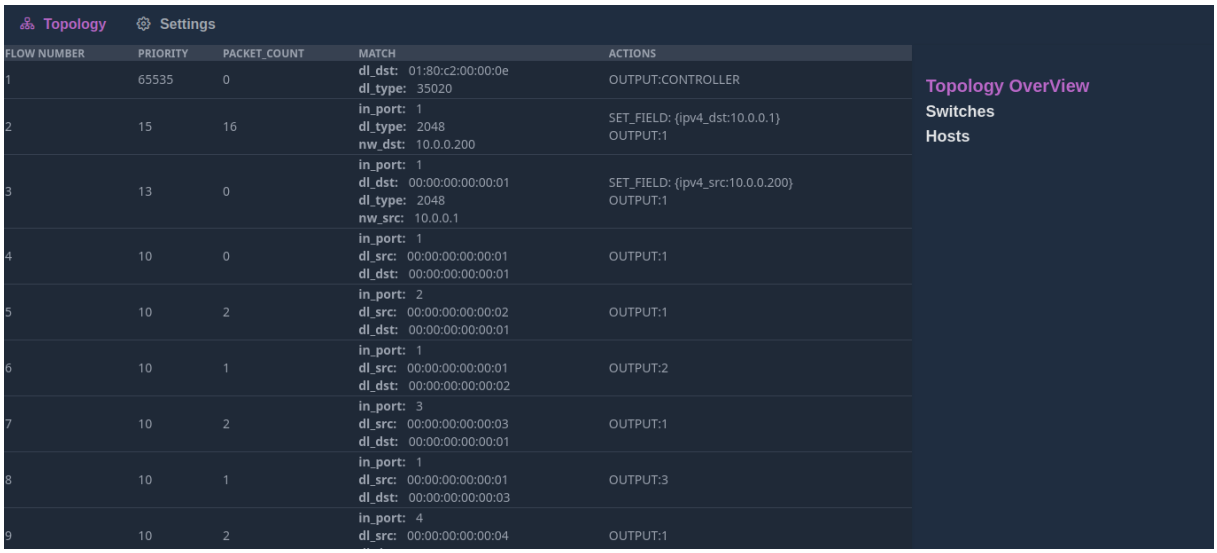


Fig. 3.10 : Affichage de la table des Flow d'un commutateur avec l'interface Web

3.6 Intégration de Docker Et CI/CD

La facilité d'intégration est un autre critère important de notre solution. Dans l'état actuel de notre solution, nous sommes confrontés à un dilemme : l'application source devra être exécutée par l'interpréteur python après l'installation des dépendances dans le cas du contrôleur, elle devra être construite après l'installation de NodeJs et des dépendances du projet. Le résultat sera un fichier HTML, JS et CSS qui devra être servi sur un serveur WEB. Nous comprenons que ce processus est fastidieux et prend du temps, que les administrateurs réseau doivent faire toutes les étapes précédentes pour configurer notre solution ou même pour nous lorsque nous intégrons la solution dans notre emulateur GNS3.

3.6.1 Docker containers

Nous avons choisi Docker pour construire la version livrable de notre solution parce que, comme nous l'avons mentionné dans la section 2.2.1, Docker construit une image d'application qui inclut toutes les dépendances de l'application et qui est prête à fonctionner sur n'importe quel serveur ou ordinateur, quelle que soit la plate-forme.

Construction des images docker

Pour construire un conteneur docker, nous devons ajouter un fichier docker qui décrit au démon docker comment construire l'image. Ce fichier inclut une image de base dont il hérite et il l'a commande pour construire et servir l'application.

- **Controller Docker :** Pour construire l'image du contrôleur on utilise l'image de base Ubuntu, on installe les dépendances du système après les dépendances de python enfin on lance les script avec RYU

```
# dockerfile
from ubuntu
RUN  apt-get update
RUN    apt-get install -y --no-install-recommends python-setuptools    py
RUN apt-get update
RUN apt-get install -y python2 pip
RUN   pip install ryu
EXPOSE 6633/tcp
RUN mkdir /app
WORKDIR /app
COPY ./ .
WORKDIR /app
CMD /usr/local/bin/ryu   run ./load_balance.py ./rest_topology.py
./ofctl_rest.py --observe-links
```

- **Image Application Web :** L'image docker de l'interface WEB est un peu plus compliqué que celle du contrôleur. la procédure de la construction de l'image est divisé en deux étapes, dans la première étape le code source sera compilé en fichiers HTML, JS , et CSS qui devront être servis de manière statique au client avec un serveur web. Nous avons choisi l'image de base de Node pour le début de la construction où nous devons installer les dépendances de Node en utilisant le gestionnaire de paquets yarn, puis nous devons construire les fichiers de distribution en utilisant l'outil vite build. Pour la phase de production, nous avons choisi le serveur web NGINX pour servir les fichiers de l'application web. Nous avons donc eu besoin du fichier nginx.conf pour configurer le serveur Web sur notre image docker afin de servir l'application Web sur le port 2345.

```
#dockerfile
FROM node:14 as build-stage
WORKDIR /app
COPY package*.json ./
RUN yarn
COPY ./ .
RUN yarn build
FROM nginx as production-stage
RUN mkdir /app
COPY --from=build-stage /app/dist /app
COPY nginx.conf /etc/nginx/nginx.conf
```

3.6.2 Implémentation d'un CI/CD

Notre script pour construire des images docker est prêt. L'étape suivante consiste à trouver un moyen de distribuer notre solution SDN aux utilisateurs de la manière la plus simple possible,

à la fois pour nous en tant que développeurs et pour nos consommateurs. Nous avons besoin d'un moyen de faire de la construction d'images Docker une tâche automatique qui fait partie de notre processus de développement. Nous avons également besoin que le processus de construction se déroule loin de nos ordinateurs pour nous permettre de continuer à développer pendant que la construction se déroule quelque part plus rapidement. Après le processus de construction, l'image doit être téléchargée à un endroit où nos utilisateurs finaux peuvent l'obtenir facilement. Cette opération doit se faire automatiquement sans aucune assistance de notre part

Github Actions

On va utiliser Github Action définit dans la section 2.2.4 . Nous utilisons ce service pour créer deux images Docker et les publier automatiquement sur DockerHub. Une Github Action est définie par une séquence des étapes qui s'exécute sur une plate-forme prédéfinit. Pour nos actions on va définir les étapes suivantes :

- **Checkout :** Cette étape permettait de télécharger le code source du projet sur la machine qui va exécuter notre action
- **Check skip :** Cette étape vérifie si la construction d'image est nécessaire, pour éviter les constructions non nécessaires on a défini que la dernière commit de Git doit avoir un message qui contient "build react" ou "build-ryu" pour que l'action continue.
- **Setup QEMU et Setup DockerX :** Cette étape prépare l'environnement dans la machine pour effectuer la construction.
- **Login to DockerHub :** Cette étape nous permettait de se connecter au DockerHub pour que l'image sera téléchargée sur DockerHub après la construction.
- **Docker build :** Cette étape construit des images et après le chargement de l'image sur DockerHub. Nous devons fournir des arguments pour cette action afin de spécifier le tag de l'image et le contexte de construction.

Ci dessus on trouve la définition de notre Github Action

```
name: build-react
on:
  push:
    branches:
      - 'master'
jobs:
  docker:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v2

      - name: check skip?
```

```
    env:
      COMMIT_FILTER: "build-react"
    run: |
      readonly local last_commit_log=$(git log -1 --pretty=format:"%s")
      echo "last commit log: $last_commit_log"

      readonly local filter_count=
          $(echo "$last_commit_log" | grep -c "$COMMIT_FILTER" )

      if [[ "$filter_count" -eq 0 ]]; then
        exit 78
      else
        echo "all good, continue"
      fi
  - name: Set up QEMU
    uses: docker/setup-qemu-action@v1

  - name: Set up Docker Buildx
    uses: docker/setup-buildx-action@v1

  - name: Login to DockerHub
    uses: docker/login-action@v1
    with:
      username: ${{ secrets.DOCKERHUB_USERNAME }}
      password: ${{ secrets.DOCKERHUB_TOKEN }}
  - name: docker build
    uses: docker/build-push-action@v2
    with:
      push: true
      tags: abdoubentegar/sdn-react,abdoubentegar/sdn-react:0.4.5
      file: nuxt/network-v/dockerfile
      context: ./nuxt/network-v
```

Chapitre 4

Implémentation et les Tests

4.1 Introduction

Dans ce chapitre, nous allons implémenter et évaluer notre solution d'équilibrage de charge SDN. Pour évaluer notre solution, nous comparerons la solution dans différents scénarios en évaluant différents aspects. Toutes les simulations dans GNS3 seront sur le serveur que nous avons configuré dans la section 2.3.2.

4.2 Préparation des outils de test

Pour nos tests nous avons besoin des outils suivants :

- **iperf :** Iperf est un outil unix pour calculer la bande passante maximum entre deux machine.
- **Http-perf :** httperf est un outil unix pour évaluer les performances d'un serveur Web.
- **Python :** Nous avons besoin de l'interpréteur Python pour lancer nos script des tests.
- **matplot-lib :** matplotlib est une bibliothèque de python qu'on a utilisé pour dessiner les graphes et visualiser nos résultats.

Selon notre expérience acquise grâce à ce projet, il nous a été facile de créer une image docker avec tous les scripts et les outils pré installés.

4.3 Préparation du Topologie

Pour nos topologies, nous avons choisi de lancer deux machines virtuelles qui simulent en KVM[1], l'avantage de cela est que ce sont des machines virtuelles chacune avec sa propre allocation de RAM et son propre processeur, contrairement aux conteneurs docker les ressources ne

[1] *Kernel-based Virtual Machine*

sont pas partager entre les instants. Cette fonctionnalité assure l'indépendance de chaque serveur à l'autre pour des résultats plus précis. c'est une faute qu'on a remarque dans notre recherche.

4.4 Tests

4.4.1 Test Initial :

Au début, nous devrions faire des tests préliminaires sans aucune configuration. Pour ces tests, nous avons pris les scénarios suivants :

Commutateur Simple avec deux hôtes :

Dans ce test, nous avons testé avec HTTPerf un scénario avec 2000 requêtes entre deux clients avec un taux de requête entre 150 et 600 et obtenons les résultats dans les deux figures 4.1 et 4.2 .

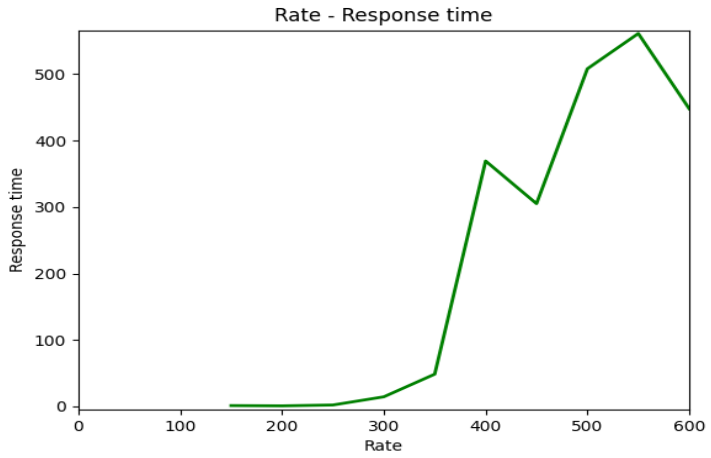


Fig. 4.1 : Taux de latence

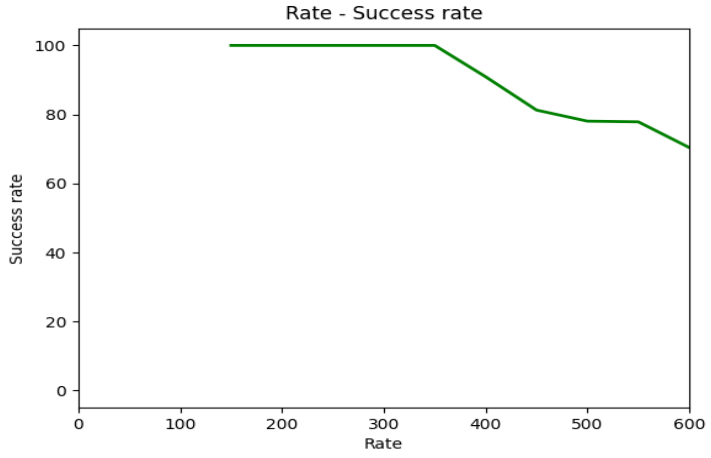


Fig. 4.2 : Taux de succès

Les tests de ce scénario sont destinés à tester les limites du serveur. Avec un seul serveur et un simple commutateur sans l'implémentation de SDN, le seul variable qui contrôle cette résultat est la capacité de notre serveur.

Nous remarquons qu'à partir de 300 requêtes simultanées, le serveur commence à avoir des problèmes de traitement des requêtes et la latence commence à augmenter. A 400 requêtes simultanées, le serveur est saturé, le taux de réussite chute à moins de 80%, donc 400 requêtes sur 2 000 échouent. De l'autre côté, la latence a augmenté jusqu'à 400 ms. Nous concluons donc qu'un seul serveur a une limite de 400 requêtes simultanées.

Commutateur OpenVswitch avec deux hôtes et deux serveurs identiques

Ce scénario est identique avec le scénario précédent, mais la seule différence est qu'on a changé le commutateur simple avec un commutateur OpenVswitch qui est contrôler par le contrôleur qu'on a développé. Nous avons effectué les mêmes tests et obtenons les résultats qui ont dans les figures 4.3 et 4.4.

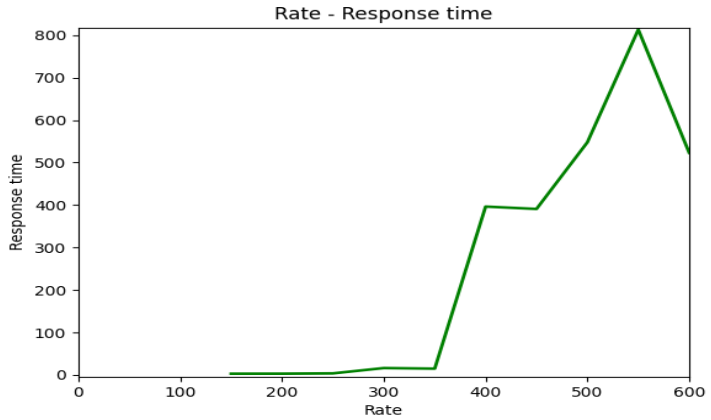


Fig. 4.3 : Taux de latence

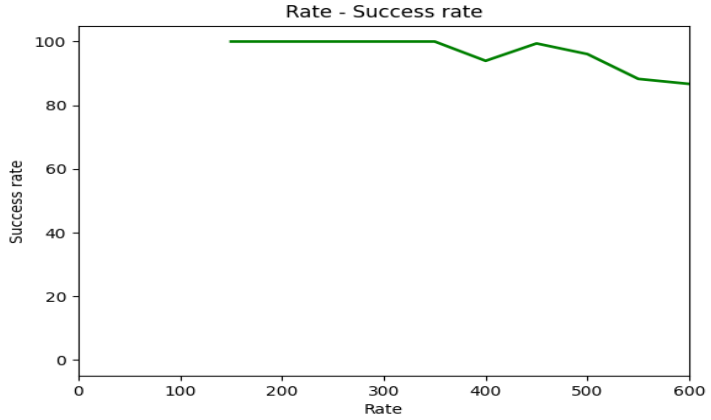


Fig. 4.4 : Taux de succès

Les tests de ce scénario ont l'objectif de tester l'impact de l'implémentation d'OpenVswitch

avec les mêmes conditions du test précédent. Il faut noter que notre commutateur OpenVswitch est émulé dans un conteneur Docker. Suite à la limitation de ressources matériels à notre possession pour pouvoir lancer le commutateur software comme un machine virtuelle. Donc les ressources sont partagés entre le commutateur et les autres deux hôtes qui seront lancer avec l'hyperviseur Docker, donc il y aura aussi l'impact de l'implémentations de Docker.

4.4.2 Test d'équilibrage de charge

Dans cette section, nous allons tester notre solution d'équilibrage de charge avec SDN. Nous sommes fortement limités par notre serveur d'émulation, malgré on a choisi les capacités maximum avec 16 Gb de RAM et 4 coeurs de processeur mais il n'était pas suffisant pour lancer tous les tests dans les meilleures conditions.

Équilibrage de charge avec la méthode round robin

Dans ce scénario, nous avons testé l'équilibreur de charge round robin entre deux serveurs et deux hôtes, un équilibrer de charge avec une adresse virtuelle 10.0.0.200 lié avec les deux serveurs émulent en KVM. On note que ces deux serveurs sont identiques. Les figures 4.5 ,4.6 et 4.7 montrent les résultats obtenus.

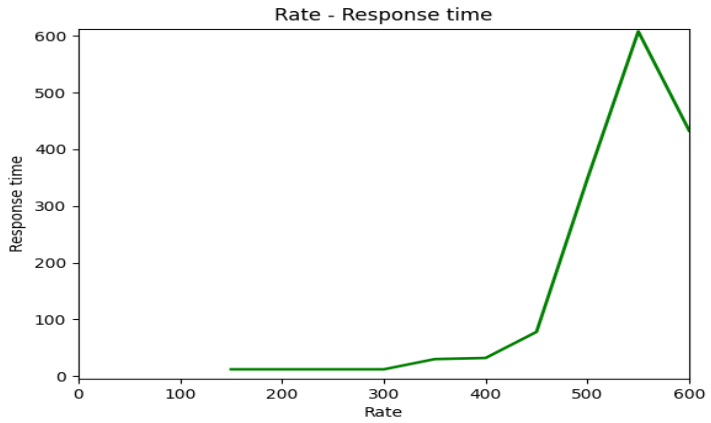


Fig. 4.5 : Taux de latence

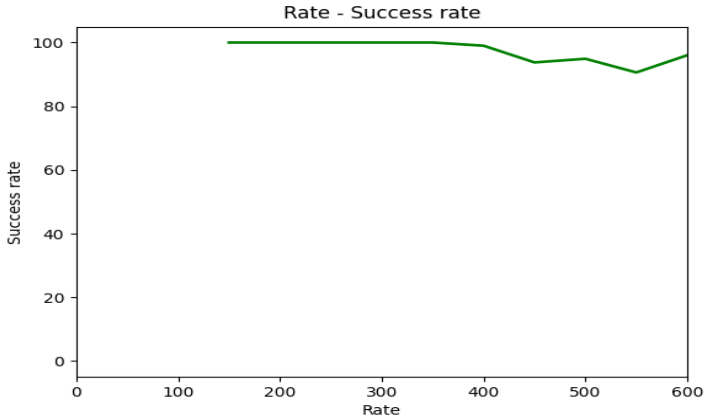


Fig. 4.6 : Taux de succès

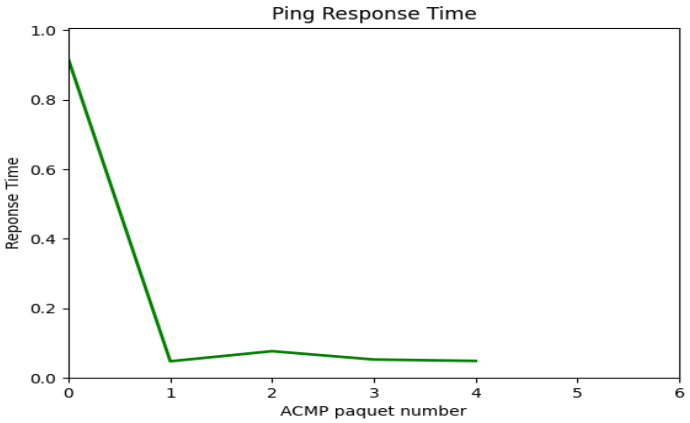


Fig. 4.7 : Taux de latence des requêtes ICMP

Les tests ce scénario visent à tester l'efficacité de notre solution d'équilibrage de charge avec SDN dans le cas où les serveurs sont identiques. Nous avons choisi la méthode de Round-Robin pour ce scénario parce qu'elle distribue les requêtes tour par tour en équivalence, donc c'est la meilleure méthode pour ce scénario avec deux serveurs identiques.

L'autre test qu'on a lancé est un test des temps des réponses d'un PING en fonction du numéro de séquence ICMP. Le but de ce test est de tester l'impact de notre solution dans le cas où il n'y a pas une nécessité de traitement des données ou de gérer un grand nombre des requêtes, le seule variable est le temps que notre solution prend pour traiter un paquet et sélectionner un serveur.

On remarque qu'à 400 requêtes simultanées le serveur commence à avoir des difficultés à traiter les requêtes et le taux de latence commence à augmenter. En 500 simultanés requêtes le serveur devient saturé. Le taux de réussite diminue à moins de 85%. Si on compare ces résultats avec les résultats du deuxième scénario, on observe une augmentation des performances de 60%.

Le troisième graphe confirme notre hypothèse initiale que seulement le premier paquet sera traité par le contrôleur. On remarque que le premier paquet dans la séquence ICMP a un temps de réponse d'environ 0.9ms alors que les autres ont un temps de réponse qui ne dépasse pas 0.1 ms.

Commutateur OpenVswitch avec deux hôtes et deux serveurs non identique

Dans ce scénario nous avons testé l'équilibrage de charge entre deux serveurs et deux hôtes. Nous avons utilisé deux hôtes et un équilibreur de charge avec une adresse virtuelle 10.0.0.200 qui est lié avec deux serveur émule en KVM. On note que ces deux serveurs ne sont pas identique, pour un de ces serveurs on a diminué la capacité de la RAM vers 512MB et le nombre des coeurs vers 1.

Pour ce scénario nous avons testé les deux méthodes Round-Robin et la méthode basée sur les charges des serveur. Les figures 4.16 ,4.17, 4.18 et 4.19 montrent les résultats obtenus.

Résultat de Round Robin

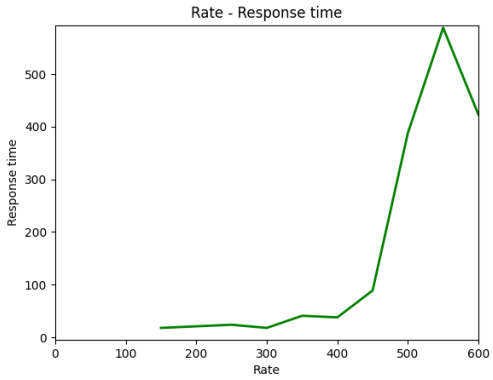


Fig. 4.8 : Taux de latence Client 1

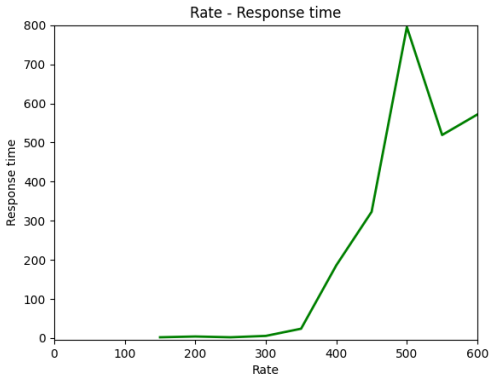


Fig. 4.9 : Taux de latence Client 2

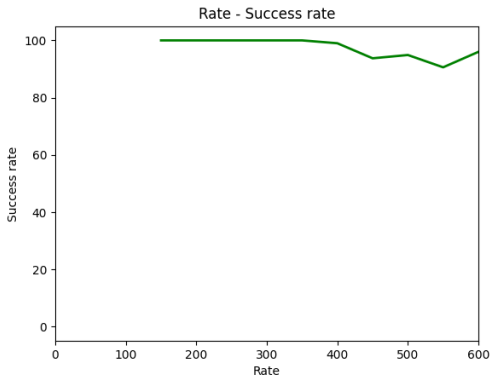


Fig. 4.10 : Taux de succès Client 1

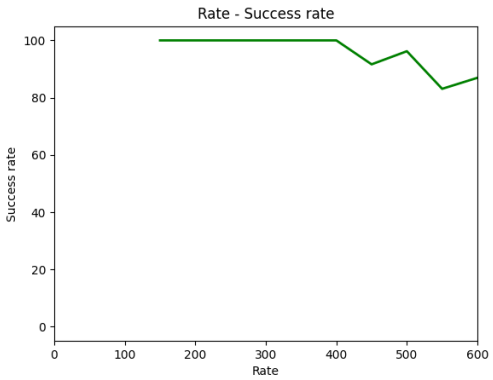


Fig. 4.11 : Taux de succès Client 2

Le but de ce test est de connaître l'efficacité de la méthode de Round-Robin quand les serveurs n'ont pas le même performance. Le principe de la méthode Round-Robin est la répartition des requêtes égales entre les serveurs. Dans ce scénario, nous n'avons pas une distribution égale des ressources entre les serveurs donc les résultats changent d'un utilisateur à l'autre. Le premier hôte allait ses requêtes au le premier serveur qui dans lequel nous n'avons rien changé, donc les résultats sont similaires aux résultats précédent. Et pour le deuxième hôte, ses requêtes sont allées au deuxième serveur, dont nous avons réduit leur capacité, donc il ne supporte que 400 requêtes au lieu de 500 requêtes. On sait que lorsque nous réduisons les capacités, elles diminuent, mais il existe une relation directe entre les capacités et la qualité.

Résultat d'équilibrage base sur la charge

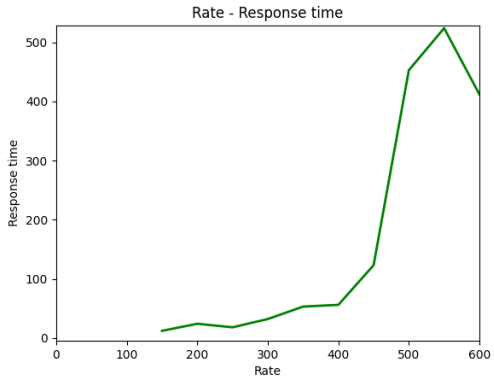


Fig. 4.12 : Taux de latence Client 1

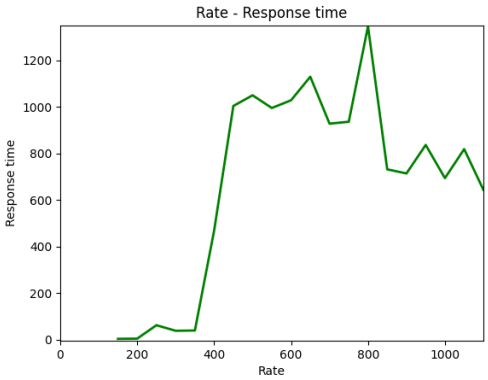


Fig. 4.13 : Taux de latence Client 2

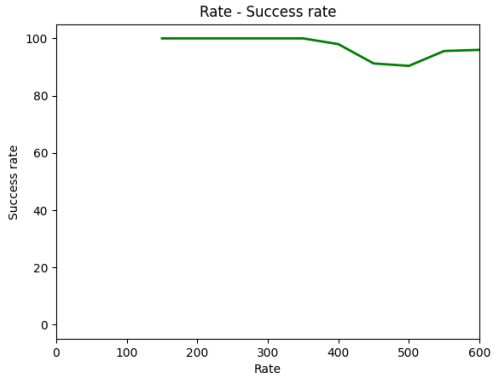


Fig. 4.14 : Taux de succès Client 1

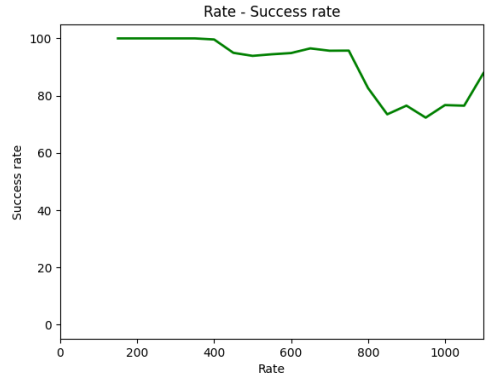


Fig. 4.15 : Taux de succès Client 2

Ces résultats se sont rapprochés des résultats de méthode Round-Robin. La raison de cette différence c'est qu'il y a une étape supplémentaire dans cette méthode qui est la communication entre les serveurs et le contrôleur afin que ce dernier récupère les rapports des charges. Dans notre solution quand on a deux hôtes et deux serveurs on aura la meme distribution des charge de la méthode Round-Robin. Nous utiliserons donc cette algorithme dans le prochain test avec un autre scénario qui pourra montrer les points forts de cette méthode.

Commutateur OpenVswitch avec six hôtes et deux serveurs non identique

Dans ce scénario, nous utiliserons les mêmes serveurs avec lesquels nous avons travaillé dans le scénario précédent (il y a une différence entre les capacités de serveurs), la différence est que nous utiliserons six hôtes avec deux serveurs, et pour le contrôleur a un service supplémentaire qui est l'allocation. Selon nos travaux, nous voyons cela comme un scénario compatible avec cette méthode. Parce qu'en round robin, il donne trois hôtes au chaque serveur, et c'est un point négatif car il y a une différence entre les capacités des serveurs, donc on va utiliser la méthode d'équilibrage base sur la charge uniquement, dans ce cas notre scénario sera de six hôtes, c'est à dire que la charge sera de trois fois plus qu'un scénario précédent, Donc pour comparer bien,

ont diminué le pas de changement de numéro de la requête de 50 à 10, ainsi mettre l'intervalle de requête envoyer entre 50 à 300 requêtes.

Dans les résultats, on va afficher les résultats du le premier et le deuxième client car ils sont l'échantillon qui est répété. Notre stratégie est que lorsque le premier hôte envoie une requête, il ira au premier serveur, et lorsque le deuxième utilisateur envoie une requête, le premier serveur sera occupé par les requêtes du premier utilisateur, c'est-à-dire qu'il enverra au deuxième serveur. Nous n'avons que deux serveurs et à partir de là, nous aurons les mêmes résultats pour les autres clients.

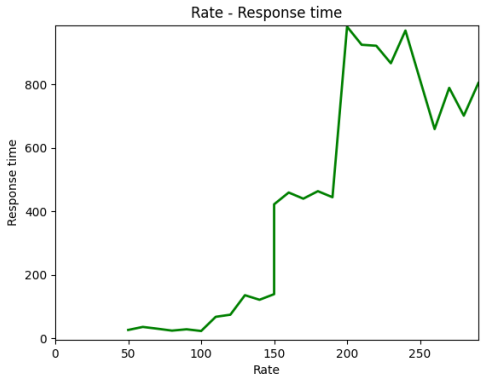


Fig. 4.16 : Taux de latence Client 1

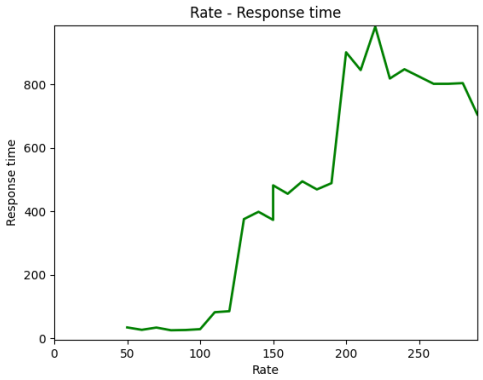


Fig. 4.17 : Taux de réponse Client 2

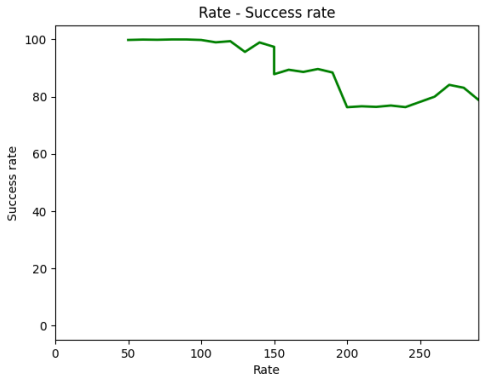


Fig. 4.18 : Taux de latence Client 1

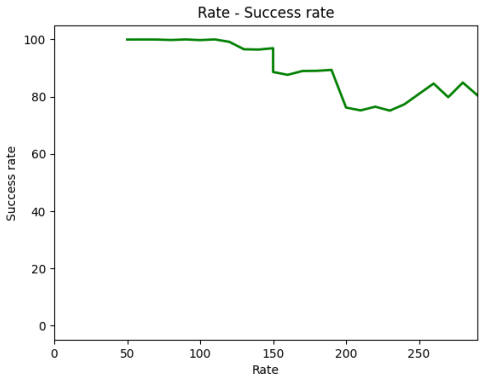


Fig. 4.19 : Taux de succès Client 2

Selon les résultats, c'est une méthode qui correspond avec ce scénario car dans ce cas, l'un des serveurs est moitié moins l'autre, ce qui signifie que la performance devrait être inférieure de moitié à l'autre, mais grâce à notre algorithme, cela a bien fonctionné, les charges sont égales. Et on a aussi les temps des réponse et latence presque les mêmes avec un décalage de 20%, alors cette méthode est travail bien avec ce scénario.

4.5 Conclusion

Dans ce chapitre, nous avons conclu que chaque scénario a une méthode de l'équilibreur de charge appropriée pour cela grâce aux bons résultats que nous avons obtenons, tels que Round-

Robin est bon lorsque les serveurs ont les mêmes capacités et également les requêtes réparties dans peu de temps, dans ce cas Round-Robin meilleure par rapport la méthode qui base sur la charge car ça fait perdre du temps à savoir la charge du serveur.

L'algorithme basé sur la charge est bon lorsque les serveurs n'ont pas les mêmes capacités, c'est-à-dire qu'il fonctionne pour être les serveurs sont équilibrés malgré son déférent dans les capacités, et ça le but de l'équilibreur de charge .

Nous voulions faire un gros test avec 50 hôtes et 5 serveurs ou plus, mais malgré l'émulation exécutée dans un serveur dédié avec 16 RAM et 4 cœurs, nous n'avons pas pu exécuter tous ces hôtes et serveurs, nous avons essayé mais cela donne de mauvais résultats et le serveur devient surchargé.

Nous pourrions avoir tout les serveur émule avec Docker, mais c'est l'erreur que beaucoup qui sont tombés dedans plusieurs les articles que nous avons lus, car les containers Docker fonctionnent dans un hypervisor qui partage les ressources entre les conteneurs, c'est l'un des avantages de Docker, mais pour nous c'est un inconvénients. Nous avons besoin d'isolation des ressource entre les serveur c'est pour ça qu'on a choisi d'émuler les serveur avec KVM.

Chapitre 5

Conclusion et travaux futurs

5.1 Conclusion

Dans ce projet on a présenté une solution d'équilibrage de charge avec la technique de "Software Defined Networking". Tout d'abord on a présente la problématique de là quel on a inspiré notre idée. Ensuite on a présente l'aspect théorique de la technique SDN et le protocole OpenFlow qui à nous permet de développer notre solution. On a développé notre solution qui fait l'équilibrage de charge entre les serveurs avec trois méthodes différentes, Puis on a amélioré l'aspect de la programmabilité et autonomie de notre solution avec un API HTTP et une application Web comme une interface graphique.

Finalement on a eu la chance de tester notre solution dans une topologie émulée avec GNS3 dans un serveur dédié et assurer la bonne fonctionnalité de notre solution et son adaptation aux différents scénarios.

Dans ce projet on a eu la chance de découvrir la nouvelle technique SDN et implémente notre logique on explorant le protocole OpenFlow. Durant cette implémentions on a eu la chance de travailler avec plusieurs technologie comme Linux, Azure, GNS3, Docker et CI/DC pour n'en citer que quelques-uns. Il était un grand plaisir et une grande expérience pour nous.

5.2 Travaux futurs

On aimerait continue le développement de cette solution dans le future, il sera notre but d'améliorer cette solution pour explorer des autres méthodes d'équilibrage de charge, explorer l'aspect de sécurité de cette solution et améliorer sons support des technologie du reseaux comme VLAN, MPLS et BGP.

On aimerait d'améliorer notre solution vers une solution complète qui aura le potentiel d'être utilise dans l'industrie .

Bibliographie

[1] Mustafa Albowarab, Nurul Zakaria et Zaheera Zainal Abidin. “Software Defined Network : Architecture and Programming Language Survey”. In : *International Journal of Pure and Applied Mathematics* 119 (jan. 2018), p. 561-572.

[2] Md Firoj Ali et Rafiqul Zaman Khan. “The study on load balancing strategies in distributed computing system”. In : *International Journal of Computer Science and Engineering Survey* 3.2 (2012), p. 19.

[4] N S Athmiya, K R Shobha et Valluri Sarimela. “Feasibility study and implementation of openflow based SDN controller for tactical scenario”. In : *2016 IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*. 2016, p. 789-794.

[6] Siamak Azodolmolky et al. “Integrated OpenFlow–GMPLS control plane : an overlay model for software defined packet over optical networks”. In : *Opt. Express* 19.26 (déc. 2011), B421-B428.

[7] Wolfgang Braun et Michael Menth. “Software-Defined Networking Using OpenFlow : Protocols, Applications and Architectural Design Choices”. In : *Future Internet* 6 (mai 2014), p. 302-336.

[8] Jasmeen Chahal et Abhinav Bhandari. “Load Balancing in Software Defined Networking : A Review”. In : *Asian Journal of Computer Science and Technology* 7 (avr. 2019).

[9] Wenbo Chen et al. “Dynamic server cluster load balancing in virtualization environment with openflow”. In : *International Journal of Distributed Sensor Networks* 11.7 (2015), p. 531538.

[13] Tushar Desai et Jignesh Prajapati. “A survey of various load balancing techniques and challenges in cloud computing”. In : *International Journal of Scientific & Technology Research* 2.11 (2013), p. 158-161.

[19] Open Networking Foundation. “Software-defined networking : the new norm for networks”. In : *ONF white paper* 2 (2012), p. 2-6.

[20] Kuntal Gaur et al. “Software Defined Networking : A review on Architecture, Security and Applications”. In : *IOP Conference Series : Materials Science and Engineering* 1099.1 (mars 2021), p. 012073.

[21] Anish Ghosh et Mrs Manoranjitham. “A study on load balancing techniques in SDN”. In : *International Journal of Engineering Technology* 7 (mars 2018), p. 174.

[23] Mohamed I Hamed et al. “A new approach for server-based load balancing using software-defined networking”. In : *2017 Eighth International Conference on Intelligent Computing and Information Systems (ICICIS)*. IEEE. 2017, p. 30-35.

[24] Nikhil Handigol et al. “Plug-n-Serve : Load-balancing web traffic using OpenFlow”. In : *ACM Sigcomm Demo* 4.5 (2009), p. 6.

[25] Taufik Hidayat, Yasep Azzery et Rahutomo Mahardiko. “Load balancing network by using round Robin algorithm : a systematic literature review”. In : *Jurnal Online Informatika* 4.2 (2020), p. 85-89.

[26] Fei Hu, Qi Hao et Ke Bao. “A survey on software-defined network and openflow : From concept to implementation”. In : *IEEE Communications Surveys & Tutorials* 16.4 (2014), p. 2181-2206.

[27] Nalini Iyer, Nikhil S Hugar, Manjunath MN et al. “Load balancing using openday light SDN controller : Case study”. In : *International Research Journal on Advanced Science Hub* 2.9 (2020), p. 59-64.

[31] Faris Keti et Shavan Askar. “Emulation of software defined networks using mininet in different simulation environments”. In : *2015 6th International Conference on Intelligent Systems, Modelling and Simulation*. IEEE. 2015, p. 205-210.

[35] Mohammadreza Mesbahi et Amir Masoud Rahmani. “Load balancing in cloud computing : a state of the art survey”. In : *Int. J. Mod. Educ. Comput. Sci* 8.3 (2016), p. 64.

[37] Heru Nurwasito et Risna Rahmawati. “Weighted Response Time Algorithm for Web Server Load Balancing in Software Defined Network”. In : *2021 3rd International Conference on Electronics Representation and Algorithm (ICERA)*. 2021, p. 143-148.

[46] Shiva Rowshanrad et al. “A survey on SDN, the future of networking”. In : *Journal of Advanced Computer Science Technology* 3 (nov. 2014), p. 232-248.

[48] Shantanu Shukla et Raghuraj Singh. “Survey on Load Balancing Techniques”. In : fév. 2019.

[50] Hadar Sufiev et al. “Dynamic SDN controller load balancing”. In : *Future Internet* 11.3 (2019), p. 75.

[51] Kazuya Suzuki et al. “A survey on OpenFlow technologies”. In : *IEICE Transactions on Communications* 97.2 (2014), p. 375-386.

[52] PV Tijare et D Vasudevan. “The northbound APIs of software defined networks”. In : *International journal of engineering sciences and research technology* 5.10 (2016), p. 501-513.

Webographie

[3] apache. *ApacheBench*. url : https://httpd.apache.org/docs/2.4/fr/programs/ab.html.

[5] atmosera. *Microsoft Azure*. url : https://www.atmosera.com/resources/glossary/azure/.

[10] cisco-goffinet. *GNS3*. url : https://cisco.goffinet.org/ccna/cisco-ios-cli/installer-et-configurer-gns3/.

[11] comptia. *Wireshark*. url : https://www.comptia.org/content/articles/what-is-wireshark-and-how-to-use-it.

[12] *CORS*. url : https://developer.mozilla.org/fr/docs/Web/HTTP/CORS.

[14] *Diagramme de Gantt*. url : https://fr.wikipedia.org/wiki/Diagramme_de_Gantt.

[15] Docs-hub. *Docker-Hub*. url : https://docs.docker.com/docker-hub/.

[16] edutechwiki. *Git*. url : https://edutechwiki.unige.ch/fr/Git_et_Github.

[17] Open Networking Foundation. *ONF OpenFlow® Conformant : Certified Product List*. url : https://opennetworking.org/product-registry/.

[18] Open Networking Foundation. *ONF Specifications*. url : https://opennetworking.org/software-defined-standards/specifications/.

[22] *Github action*. url : https://tkt.paris/comment-creer-une-action-github-pour-tester-son-application-avec-des-services/.

[28] *Java-script*. url : https://fr.wikipedia.org/wiki/JavaScript.

[29] journaldunet. *OpenVPN*. url : https://www.journaldunet.fr/web-tech/guide-de-l-entreprise-digitale/1506369-openvpn-tout-savoir-sur-le-vpn-open-source-pour-les-pros/.

[30] journaldunet. *OpenVswitch*. url : https://www.techopedia.com/definition/28936/open-vswitch.

[32] lebigdata. *Docker*. url : https://www.lebigdata.fr/docker-definition.

[33] linux-kvm. *Kernel-based Virtual Machine*. url : https://www.linux-kvm.org/page/Main_Page.

[34] logicfinder. *SDN Technology The Transformer of Network Automation*. url : https://www.logicfinder.net/sdn-technology-the-transformer-of-network-automation/.

[36] *Mininet*. url : http://mininet.org/.

[38] ONF. *OpenFlow 1.1.0 Switch Specification*. Mars 2015. url : https://opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf.

[39] ONF. *OpenFlow 1.1.0 Switch Specification*. Mars 2015. url : https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.1.0.pdf.

[40] ONF. *OpenFlow 1.2.0 Switch Specification*. Mars 2011. url : https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.2.pdf.

[41] ONF. *OpenFlow 1.5.1 Switch Specification*. Mars 2015. url : https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.

[42] pcmag. *Linux*. url : https://www.pcmag.com/encyclopedia/term/linux.

[43] Pica8. *Pica8 Group Tables*. url : https://docs.pica8.com/display/PICOS2111cg/Creating+a+Group+Table.

[44] Pica8. *Pica8 Hardware OpenFlow Multi-table Limitations*. Mars 2015. url : https://docs.pica8.com/display/PICOS2111cg/Configuring+Multi-Tabl.

[45] Linda Rosencrance. *software-defined networking (SDN)*. url : https://www.techtarget.com/searchnetworking/definition/software-defined-networking-SDN.

[47] sdxcentral. *Docker-container*. url : https://www.sdxcentral.com/cloud/containers/definitions/what-is-docker-container/.

[49] *SQLite*. url : https://fr.wikipedia.org/wiki/SQLite.

[53] Ubuntu. *Wireshark*. url : https://ubuntu.com/server/docs/virtualization-qemu.

[54] *What is Python ?* . url : https://www.python.org/doc/essays/blurb/.

[55] Wikipedia. *GitHub*. url : https://fr.wikipedia.org/wiki/GitHub.

[56] Wikipedia. *React*. url : https://fr.wikipedia.org/wiki/React.

Résumé

Le réseau définie par logiciel est une technique qui permet de séparer le plan de contrôle du plan de données d'un équipement réseau. le réseau définie par logiciel est considéré comme l'un des techniques modernes le plus important dans le domaine des réseaux, car il contient sa dynamique et sa structure qui facilitent la gestion et donnent de l'espace pour l'application d'un grand nombre d'applications, et c'est ce qui a poussé les chercheurs à le considérer et le développer pour le mieux.
L'objectif principal de notre projet est de créer un équilibreur de charge implémenter dans le réseau SDN, et de donner à l'administrateur la liberté de gestion le réseau de manière dynamique et simple.

Mots clés : Flux ouvert , commutateur , contrôleur , internet , réseau .

Abstract

Software-defined network is a technique for separating the control plane from the data plane of a network equipment. the software-defined network is considered to be one of the most important modern networking techniques because it contains its dynamics and structure which facilitate management and give space for the application of a large number of applications, and this is what prompted researchers to consider it and develop it for the better.

The main objective of our project is to create a load balancing solution for a Software Defined Network, and to give the administrator the freedom to manage the network in a dynamic and simple way.

Keywords : OpenFlow ,switch , internet , network , controller .

ملخص

الشبكات المعرفة بالبرمجيات هي تقنية تفصل مستوى التحكم عن مستوى البيانات لجهاز الشبكة. تعتبر الشبكة المعرفة بالبرمجيات من أهم التقنيات الحديثة في مجال الشبكات ، لاحتوائها على ديناميكياتها وبنيتها التي تسهل الإدارة وتوفر مساحة لتطبيق عدد كبير من التطبيقات ، وهذا ما دفع الباحثين إلى النظر فيه وتطويره للأفضل.

الهدف الرئيسي لمشروعنا هو إنشاء موازن تحميل لتنفيذه في شبكة SDN ، ومنح المسؤول حرية إدارة الشبكة بطريقة ديناميكية وبسيطة.

كلمات مفتاحية :. فتح تيار ، تبديل ، متحكم ، إنترنت ، شبكة.
