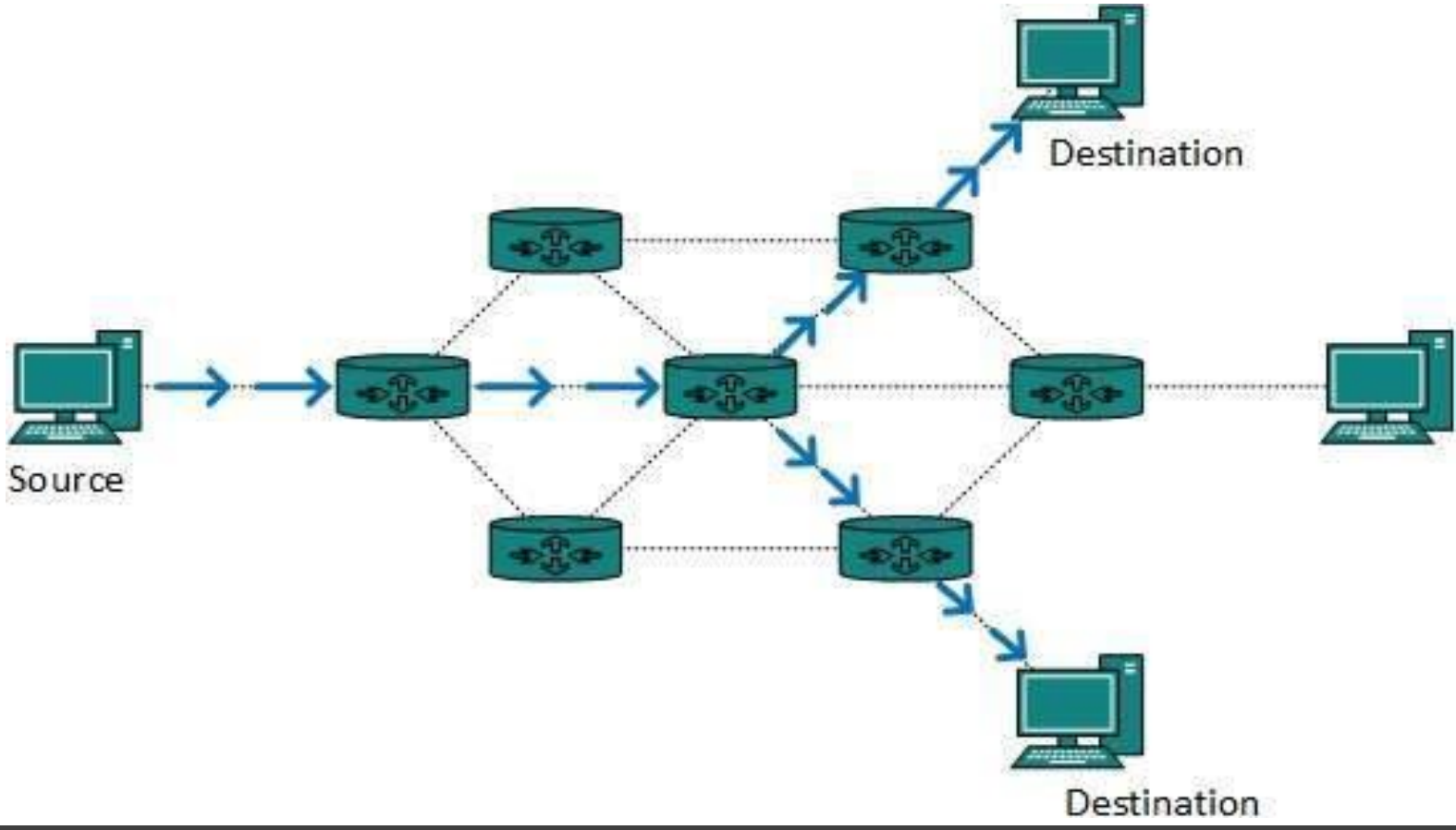
PROJET ADMINISTRATION RESEAU 4GI 2024



SDWMN : Routage multi chemins intelligent dans les réseaux maillés sans fil programmable

THEME 2 : IDENTIFICATION DES CHEMINS POSSIBLES ET ACCEPTABLES

SOUS LA SUPERVISION DE :

Pr Dr Ing Thomas Djotio Ndie

&

PhD Student Judith Nouho Noutat

MEMBRES DU GROUPE

- 20P263 MBO ALAIN GERARD (CHEF DE GROUPE)
- 20P159 DJIDDA MOUSTAPHA
- 20P292 NDONGO AMBASSA INGRID VANELLA
- 20P392 NJOUKOUMBE JAFAROU
- 14P124 NKODO ABANDA HILAIRE VICTOR FLAVIEN
- 22P146 KENFACK VOUFACK JODELLE

Table des matières

INTRODUCTION

Le Software-Defined Networking (SDN) est une approche du réseau qui s'appuie sur des contrôleurs logiciels ou des API pour diriger le trafic sur le réseau et communiquer avec l'infrastructure matérielle sous-jacente. Le SDN peut créer et contrôler un réseau virtuel ou contrôler un réseau matériel traditionnel à l'aide d'un logiciel. [1]. Parmi ces tâches de contrôle nous pouvons citer entre autres **l'identification des chemins possibles et acceptables**. Dans le contexte de ce projet, cette identification se fait grâce à un flux de données classifiées sur les différentes énergies résiduelles en fonction des nœuds fournis par un groupe au préalable. Ainsi dans cette analyse, il se pose le problème de savoir comment faire pour limiter les pertes en termes d'information dans le réseau. Dans l'optique de résoudre ce problème, notre analyse sera tout d'abord axée sur la description du problème et la présentation de l'architecture que nous allons utiliser pour mettre en évidence le plus possible notre problème, puis nous allons mettre en place un modèle mathématique nous permettant de voir la solution d'un point de vue abstrait en se basant sur les algorithmes de parcours et les heuristiques. Ensuite nous allons analyser et concevoir la solution à ce problème. Enfin, nous allons implémenter en langage python (sur Ryu avec Mininet-Wifi et Flowmanager) la recherche des chemins possibles et acceptables en se basant sur la modélisation mathématique.

I. DESCRIPTION DU PROBLÈME ET PRÉSENTATION DE L'ARCHITECTURE UTILISÉES

1. DESCRIPTION DU PROBLÈME

Dans le contexte de SDN, le réseau est géré de manière centralisée à l'aide d'un contrôleur SDN. Le contrôleur est responsable de la prise de décision concernant le routage du trafic et l'identification des chemins à travers le réseau. Le contrôleur utilise des informations sur l'état du réseau, telles que la topologie, la charge du trafic et les politiques de gestion, pour déterminer les chemins possibles et acceptables pour les paquets de données.

Le problème de l'identification des chemins possibles et acceptables ici est lié aux objectifs et aux exigences spécifiques du réseau. Le contrôleur SDN utilise des protocoles de communication, tels que OpenFlow, pour établir des règles dans les commutateurs réseau pour guider le trafic le long des chemins souhaités. De ce fait le contrôleur aura donc pour objectif dans le cadre de notre travail de trouver à partir d'une table le prochain saut d'un nœud et la liste des chemins possibles pour un paquet de données. Puis à partir de cette liste de chemins nous allons établir quels sont les chemins acceptables qui vont permettre de maintenir le trafic en état fonctionnel. D'où le problème de ce travail qui est de savoir comment **maintenir le trafic en état fonctionnel afin de pouvoir garder au mieux l'information dans un réseau.** Nous allons ensuite définir l'architecture utilisée afin de mieux comprendre comment résoudre ce problème.

2. PRÉSENTATION DE L'ARCHITECTURE SDN - SDWMN

Le SDWMN (Software Defined Wireless Mesh Network) ou encore Réseau maillé sans fil à définition logicielle est une architecture réseau qui permet aux responsables informatiques d'utiliser des logiciels pour réaliser une gestion centralisée du réseau. Aussi, le routage multi chemin est une stratégie de routage qui permet à un paquet de données de suivre plusieurs chemins différents pour atteindre sa destination. Une architecture SDN standard est constituée de trois éléments : [1]

- **Applications** : elles communiquent des demandes de ressources ou des informations sur l'ensemble du réseau
- **Contrôleurs** : ils utilisent les informations provenant des applications pour décider de l'acheminement d'un paquet de données
- **Périphériques réseau** : ils reçoivent des informations du contrôleur sur l'endroit où déplacer les données [1]

Même si le principe d'un logiciel centralisé contrôlant le flux de données dans les commutateurs et les routeurs est commun à tous les réseaux software-defined, il existe différents modèles de SDN notamment : [1]

- **SDN ouvert** : les administrateurs réseau utilisent un protocole de type OpenFlow pour contrôler le comportement des commutateurs virtuels et physiques au niveau du plan de données.
- **SDN piloté par API** : au lieu d'utiliser un protocole ouvert, les interfaces de programmation d'applications contrôlent la façon dont les données circulent sur chaque terminal à travers le réseau.
- **SDN de superposition** : cet autre type de réseau software-defined exécute un réseau virtuel par dessus une infrastructure matérielle existante, créant des tunnels dynamiques vers différents Data Centers on premise et distants.

Le réseau virtuel alloue la bande passante à divers canaux et affecte des terminaux à chaque canal, laissant le réseau physique intact.

- **SDN hybride** : ce modèle combine un réseau software-defined à des protocoles réseau traditionnels dans un environnement unique pour la prise en charge des différentes fonctions du réseau. Les protocoles de réseau standard continuent à orienter une partie du trafic, tandis que le SDN prend en charge une autre partie du trafic, ce qui permet aux administrateurs réseau d'introduire le SDN par étapes au sein d'un environnement legacy.

II. ELABORATION DU MODÈLE MATHÉMATIQUE

Pour élaborer le modèle mathématique il est d'abord nécessaire de lever l'abstraction qu'est le réseau pour la ramener à un outil mathématique simple. En effet, notons que le choix de la structure mathématique dépend du problème dans son intégralité.

La structure utilisée ici est un **graphe non orienté G = (V, E)** avec V l'ensemble des routeurs et E l'ensemble des liens entre les routeurs.

Cette modélisation se fera en deux étapes. Premièrement la recherche des chemins possibles par l'algorithme de parcours en Largeur et s'en suivra la recherche des chemins acceptables en utilisant une heuristique sur les énergies résiduelles qui font office de poids pour nos différents liens.

1. Utilisation de l'algorithme de parcours en largeur et pour la détermination des chemins possibles

L'algorithme de parcours en largeur (ou BFS, pour Breadth-First Search en anglais) permet le parcours d'un graphe ou d'un arbre de la manière suivante : on commence par explorer un nœud source, puis ses successeurs, puis les successeurs non explorés des successeurs, etc. L'algorithme de parcours en largeur permet de calculer les distances de tous les nœuds depuis un nœud source dans un graphe non pondéré (orienté ou non orienté). [2]

a. Algorithme

L'algorithme de parcours en largeur est le suivant [3] :

```
Parcours Largeur(Graphe G, Sommet s):
    f = CreerFile();
    f.enfiler(s);
    marquer(s);
    tant que la file est non vide
        s = f.defiler();
        afficher(s);
        pour tout voisin t de s dans G
            si t non marqué
                f.enfiler(t);
                marquer(t);
```

b. Complexité

La complexité en temps dans le pire cas est en O(V+ E). En effet, chaque arc et chaque sommet est visité au plus une seule fois. Montrons le. Soit un graphe G=(V, E), dont aucun sommet n'est marqué à l'appel de l'algorithme. Tous les sommets insérés dans la file sont marqués et l'instruction conditionnelle assure donc que les sommets seront insérés au plus une fois, comme l'insertion dans la file se fait en Θ(1), la complexité est en Θ(V). La boucle sur les voisins t d'un sommet s consiste à lister les voisins a une complexité en temps de l'ordre de grandeur du nombre de voisins de

2. Utilisation de l'algorithme de parcours en largeur avec une heuristique pour la détermination des chemins acceptables.

L'heuristique ou euristique est « l'art d'inventer, de faire des découvertes » en résolvant des problèmes à partir de connaissances incomplètes. Ce type d'analyse permet d'aboutir en un temps limité à des solutions acceptables. Celles-ci peuvent s'écarter de la solution optimale. Pour Daniel Kahneman, c'est une procédure qui aide à trouver des réponses adéquates, bien que souvent imparfaites, à des questions difficiles 3.

L'algorithme que nous utiliserons se base sur une heuristique qui va nous permettre cette fois ci non seulement de réduire le temps d'exécution de la recherche mais aussi de nous permettre de maintenir le trafic en état en éliminant les chemins qui vont être compromettant à la survie du trafic sur le long terme.

Mathématiquement parlant nous allons toujours utiliser la même structure que précédemment pour avoir les différents chemins sauf que cette fois ci

l'heuristique nous permettra non seulement d'améliorer la complexité en temps et aussi à maintenir le réseau en bon état. En effet, la complexité en temps est de l'ordre du nombre des nœuds et d'arcs. Nous allons donc avoir $O(V + E)$ avec V le nombre de nœuds et E le nombre d'arcs. Maintenant en soustrayant certains nœuds sous la forme : $E' = E - R$ avec R le nombre de nœuds ne vérifiant pas l'heuristique et E' le nombre de nœuds vérifiant celle-ci. Nous allons donc avoir comme nouvelle complexité $O(V + E')$. Cette complexité est meilleure que la précédente et nous montre en effet que l'heuristique est une meilleure façon d'améliorer le trafic.

Dans cette partie, nous utilisons l'algorithme de parcours en largeur qui en fonction du type de trafic et l'énergie qui en résulte, trouver les chemins acceptables, l'algorithme est le suivant:

Algorithme1 : Parcours en largeur BFS(G, s,typeTrafic)

Données : graphe G, sommet de départ s , energie, energieSeuil

File q (initialisée à vide), marque des sommets (initialisé à Faux)

début

energieSeuil ← const

marque[s] ← Vrai ;

enfiler s à la fin de q ;

tant que q non vide faire

u ← tête(q) ;

pour chaque v voisin de u faire

si energie <= energieSeuil (1)

```
                    si v non marqué alors
                        marque[v ] ← Vrai ;
                        enfiler v à la fin de q ;
                    fin
                fin
            fin
            défiler u de la tête de q ;
        fin
    fin
```

Notons qu'avec cet algorithme il est naturellement possible de trouver les chemins possibles en ajoutant la condition (1). Ainsi dans la suite nous allons faire une implémentation de ces algorithmes en Python en se basant sur l'orienté objet.

III. ANALYSE ET CONCEPTION DE LA SOLUTION

1. Analyse de la Solution

La méthode pour trouver les chemins possibles et acceptables dans les réseaux SDN vise à optimiser le routage des données en utilisant une approche centralisée de gestion du réseau. Cette solution nécessite la collecte d'informations sur la topologie du réseau SDN et de connaitre l'état du réseau en temps réel. En utilisant ces informations, un algorithme de calcul de chemin est appliqué pour trouver les chemins possibles et acceptables. La programmation du contrôleur SDN permet de mettre en œuvre ces chemins en établissant des règles de flux dans les commutateurs du réseau.

La première étape de la solution consiste à collecter les informations nécessaires. Cela comprend la découverte de la topologie du réseau SDN à l'aide de protocoles de découverte de topologie tels que OpenFlow. Ensuite, des mécanismes de surveillance du réseau, tels que SNMP, sont utilisés pour collecter les informations sur l'état du réseau, y compris la charge du trafic, la bande passante disponible et la latence. Mais cette partie a déjà été effectué par un autre groupe à intérêt connexe. Naturellement ces informations sont liées aux nœuds et à leur énergie résiduelle sans oublier l'énergie résiduelle seuil pour un type de service.

Une fois les informations collectées dans une table un algorithme de calcul de chemin est appliqué pour trouver les chemins possibles et acceptables. Ces algorithmes tiennent compte de la topologie du réseau et de l'état du réseau en termes d'énergie consommée en un temps réel.

Une fois les chemins possibles et acceptables calculés, le contrôleur SDN est programmé pour mettre en œuvre ces chemins dans les commutateurs du réseau.

Cela se fait en utilisant le protocole OpenFlow pour établir des règles de flux qui guident le trafic le long des chemins identifiés. De plus, la solution va surveiller en permanence l'état du réseau afin de mettre à jour dynamiquement les chemins lorsque cela est nécessaire. Cela garantit une adaptation continue aux changements du réseau et aux besoins du système.

2. Conception de la Solution

La conception de la solution pour trouver les chemins possibles et acceptables dans les réseaux SDN repose sur une architecture centralisée. Le contrôleur SDN joue un rôle central en étant responsable de la programmation des commutateurs et de la gestion du flux de données. Les commutateurs SDN sont utilisés pour acheminer le trafic selon les règles établies par le contrôleur. Les protocoles de communication, tels qu'OpenFlow, permettent la communication entre le contrôleur et les commutateurs. Les modules d'analyse et de prise de décision utilisent les informations sur la topologie, l'état du réseau et les politiques pour calculer les chemins possibles et acceptables.

Les modules d'analyse et de prise de décision sont essentiels pour la conception de la solution. Ils prennent en compte les informations sur la topologie du réseau SDN, l'état du réseau et les politiques de routage et de gestion pour calculer les chemins possibles et acceptables. Les modules d'analyse peuvent également inclure des mécanismes de surveillance du réseau pour collecter en temps réel les informations sur la charge du trafic, la bande passante disponible et la latence, afin d'optimiser les chemins calculés.

La programmation du contrôleur SDN est une étape cruciale de la conception. Le contrôleur doit être capable de prendre les chemins calculés par les modules d'analyse et de les convertir en règles de flux pour les commutateurs

du réseau. Cela se fait en utilisant le protocole OpenFlow, qui permet de définir les règles de correspondance et d'action pour diriger le trafic. Le contrôleur doit également être capable de gérer la mise à jour dynamique des chemins en fonction des changements de l'état du réseau et des politiques, en reprogrammant les règles de flux en conséquence.

La conception de la solution doit également inclure des mécanismes de mise à jour dynamique des chemins. Cela permet de s'adapter aux changements du réseau, tels que l'ajout ou la suppression de liens ou de commutateurs, ainsi qu'aux changements des politiques de routage. Lorsqu'un changement est détecté, les modules d'analyse recalculent les chemins possibles et acceptables et le contrôleur met à jour les règles de flux dans les commutateurs en conséquence.

IV. IMPLEMENTATION

1- MISE EN EVIDENCE DE LA LOGIQUE ORIENTEE OBJET

Dans la suite, nous allons montrer comment identifier les chemins possibles dans un réseau. Nous allons toujours nous baser sur la structure de graphes non orienté. Nous allons nous attarder sur le parcours en largeur pour trouver les chemins possibles dans le graphe en question d'un nœud source S vers un nœud destination D. Nous avons donc utilisé une approche orientée objet en appelant notre Graphe comme une classe qui a les attributs voisins et énergies. Ces attributs permettent respectivement de connaître les voisins d'un nœud et l'énergie résiduelle dans ce nœud. Le principe pour les chemins acceptables cette fois-ci se base sur les chemins possibles en retirant le chemin possible contenant un nœud

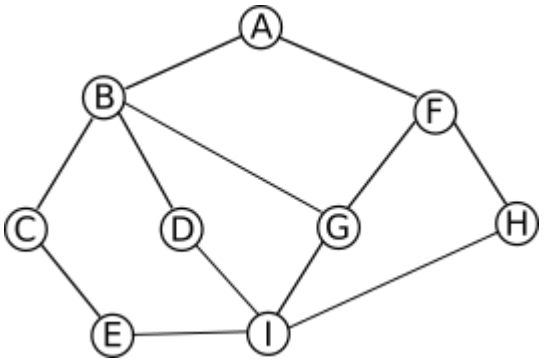
« endommagé ». Un nœud est dit endommagé lorsqu'il ne peut pas acheminer un paquet. Dans ce cas, son énergie résiduelle est inférieure à l'énergie résiduelle seuil pour le service en question. Le bout de code ci-dessous montre la structure de la classe **Graph** que nous avons utilisé :

```
class Graph:
    def __init__(self):
        self.voisin = {}
        self.energy = {}
```

2- IDENTIFICATION DES CHEMINS POSSIBLES EN UTILISANT LE PARCOURS EN LARGEUR

Ici, il s'agira de determiner les chemins possibles partant d'un nœud source à un nœud destination. Naturellement nous allons utiliser l'objet Graph défini plus haut.

Soit le graphe non orienté suivant :



Pour implémenter celui-ci, nous utiliserons le langage python pour commencer par créer le graphe. Le code de création est le suivant :

```
edges = [
    ['A', 'B'],
    ['A', 'F'],
    ['B', 'C'],
    ['B', 'D'],
    ['B', 'G'],
    ['C', 'E'],
    ['D', 'I'],
    ['E', 'I'],
    ['I', 'G'],
    ['I', 'H'],
    ['F', 'G'],
    ['F', 'H'],
]
```

Ce code permet d'initialiser les voisins directs qui représentent dans SDN ce que l'on va appeler les prochains sauts pour un nœud donné. Maintenant pour la suite utiliser une méthode statique de parcours en larguer qui va pour un graphe donné nous retourner une liste de chemins :

```
def parcours_en_largeur(self, start, end):
    queue = deque([(start, [start])])
    paths = []

    while queue:
        node, current_path = queue.popleft()

        if node == end:
            paths.append(current_path)
        else:
            if node not in self.voisin:
                continue

            neighbors = self.voisin[node]
            for neighbor in neighbors:
                if neighbor not in current_path:
                    queue.append((neighbor, current_path + [neighbor]))

    return paths
```

3- IDENTIFICATION DES CHEMINS ACCEPTABLES

Avant de chuter sur la détermination des chemins acceptables nous allons expliquer comment prendre en compte les énergies dans le graphe. Nous allons pour le graphe précédent mettre en évidence les énergies.

```
energies = {
    'A': 190,
    'B': 4,
    'C': 44,
    'D': 99,
    'E': 190,
    'F': 40,
    'G': 190,
    'H': 80,
    'I': 122,
}
```

Pour l'identification des chemins acceptables, nous le faisons avec le code ci-dessus :

```
def chemins_acceptables(self,PATH_LIST,energy):
    TEMP = []
    for i in range(1000):
        for path in PATH_LIST:
            for node in path:
                if self.get_energy(node) < energy and path in PATH_LIST:
                    print(path)
                    PATH_LIST.remove(path)
                    print(PATH_LIST)
    return PATH_LIST
```

Notons que ce code sue base aussi sur l'algorithme écrit dans la partie modélisation mathématique et se base aussi sur l'heuristique que nous allons développer plus bas.

4- FORMULATION DE L'HEURISTIQUE ET PREUVE DE L'HEURISTIQUE

Dans le cas du problème dont nous voulons apporter une solution à l'identification des chemins possibles et acceptables, nous devons prendre en compte non seulement le réseau qui est représenté sous forme de graphe mais aussi la consommation en énergie des paquets dans le réseau fonction du type de service. Pour cela, il était nécessaire de tenir compte des énergies nécessaires pour le fonctionnement des équipements, des énergies nécessaires aux équipements de transférer un paquet reçu vers une destination précise ; cette énergie qui appelée énergie seuil. Les 03 principaux services supportés par notre architecture réseau sont :

- Le "**Share services**"
- Le " **Dedicated services**"
- Le "**Real Time services**"

Il est à noter que tous ces 3 services ont une consommation énergétique et une énergie résiduelle seuil différentes.

Dans un réseau peu complexe, la transmission se limiterait à l'utilisation des algorithmes de parcours classique. Mais dans notre cas, considérant que les équipements dans réseau peuvent atteindre un très nombre, les algorithmes conventionnels seront plus lents à traiter les transactions dans le réseau car leur complexité atteindra une puissance carrée. Pour ce fait, l'utilisation d'une heuristique était nécessaire. Cette heuristique (compte tenu du fait qu'un paquet dans un réseau a une énergie résiduelle) sera définie en comparant l'énergie qu'il faut à l'équipement pour envoyer le paquet vers les équipements suivant à l'énergie résiduelle du paquet, si cette différence sera positive ou négative.

Dans le cas où cette différence est négative, on supprime ce chemin dans la liste des chemins acceptables sinon on l'ajoute.

Donc notre heuristique se base sur les énergies des équipements et les énergies des paquets à transiter pour déterminer les chemins possibles, et les chemins acceptables qui nécessitent le minimum d'énergie pour transférer un paquet d'un équipement à un autre. Tout en sachant que l'énergie résiduelle varie d'un type de trafic à un autre.

Compte tenu de l'heuristique une adaptation du programme prennant en compte les types de services est mise en œuvre :

```
def chemins_acceptables(self,PATH_LIST,services_type):
    if services_type == "Shared_Service":
        energy_seuil = 80
    elif services_type == "Real_Time_Service":
        energy_seuil = 40
    elif services_type == "Dedicated_Service":
        energy_seuil = 90
    TEMP = []
    for i in range(1000):
        for path in PATH_LIST:
            for node in path:
                if self.get_energy(node) < energy_seuil and path in PATH_LIST:
                    print(path)
                    PATH_LIST.remove(path)
                    print(PATH_LIST)
    return PATH_LIST
```

Ces algorithmes vont être mis en œuvre dans un contrôleur Ryu Manager et simulé sur Mininet Wifi puis illustré graphiquement via FlowManager.

5- MISE EN ŒUVRE D'UN CONTROLEUR RYU POUR LA DETERMINATION DES CHEMINS POSSIBLES ET ACCEPTABLES

Pour la mise en place de ce contrôleur nous allons utiliser les évenements de Ryu pour le contrôle du trafic dans notre réseau grace à Mininet Wifi. Nous allons naturellement nous assurer que Mininet n'utilise pas son propre contrôleur pour éviter les conflits de contrôleur avec Ryu. Voici donc le code de Python permettant de gérer le contrôleur :

```python
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER, set_ev_cls
from ryu.ofproto import ofproto_v1_3
from collections import deque

class MyRyuController(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(MyRyuController, self).__init__(*args, **kwargs)
        self.graph = Graph()

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, MAIN_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        parser = datapath.ofproto_parser
        ofproto = datapath.ofproto
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER, ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

    def add_flow(self, datapath, priority, match, actions):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, actions)]
        mod = parser.OFPFlowMod(
            datapath=datapath, cookie=0, cookie_mask=0,
            table_id=0, command=ofproto.OFPFC_ADD, idle_timeout=0, hard_timeout=0,
            priority=priority, buffer_id=ofproto.OFP_NO_BUFFER,
            out_port=ofproto.OFPP_ANY, out_group=ofproto.OFPG_ANY,
            flags=0, match=match, instructions=inst
        )
        datapath.send_msg(mod)

    def determine_paths(self, src, dst, services_type):
        edge_list = [(1, 2), (2, 3), (3, 4)]
        self.graph.add_edge(edge_list)

        energy_dict = {1: 75, 2: 85, 3: 60, 4: 95}
        self.graph.set_energy_Dict(energy_dict)

        paths = self.graph.parcours_en_largeur(src, dst)
        acceptable_paths = self.graph.chemins_acceptables(paths, services_type)

        return acceptable_paths
```

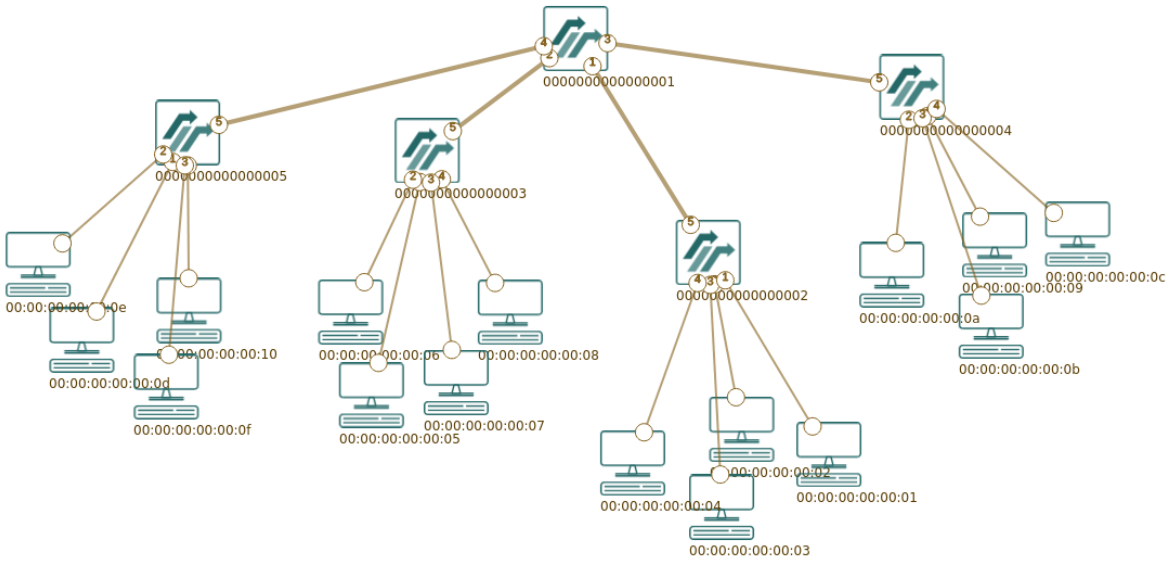
Nous allons ensuite nous assurer que le trafic des paquets ICMP via pingall sur Mininet est bien effectif puis nous allons donc nous mettre dans la détermination des prochains sauts puis des chemins possibles et acceptables.

6- SIMULATION MINIET ET ILLUSTRATION GRAPHIQUE AVEC FLOWMANAGER

Nous allons mettre en place une topologie réseau simple composée de 16 hotes et 5 switch avec un contrôleur avec la commande Mininet suivante:

```
sudo mn --controller=remote,ip=127.0.0.1 --mac -i 10.0.0.1/24 --topo=tree,depth=2,fanout=4
```

Notons que le contrôleur utilisé n'est pas celui de Mininet par défaut grâce au mot clé **remote.** Maintenant nous allons maintenant utiliser FlowManager pour visualiser le trafic ainsi que les chemins possibles et acceptables sur la base du code du contrôleur Ryu précédent :



CONCLUSION

En somme, les réseaux maillés sans fil programmables (SDWMN) offrent une approche flexible et centralisée pour la gestion des réseaux sans fil. La recherche de chemins possibles et acceptables dans ces réseaux joue un rôle crucial dans l'optimisation des performances. L'identification des chemins possibles et acceptables dans un SDWMN se fait en modélisant le réseau sous forme d'un graphe pondéré. Les poids des liens peuvent représenter des contraintes spécifiques telles que l'énergie résiduelle ou les coûts de transmission. Dans cette analyse nous avons mis en evidence les algorithmes de parcours en largeur et celui basé sur l'heuristique pour déterminer les chemins possibles et acceptables respectivement en se basant sur une analyse détaillée pour lever les abstractions liées au réseau. De plus nous avons fait des implémentation python en se basant sur l'orienté objet après une analyse et une conception détaillée. Nous avons aussi mis en évidence Ryu et de Mininet pour une implémentation et une visualisation avec FlowManager. De cette analyse il en ressort que ces chemins acceptables nous permettent effectivement de maintenir le réseau en bon état mais pendant combien de temps ? Ne faudrait-il pas trouver un chemin optimal ?

REFERENCES :

[1]https://www.vmware.com/fr/topics/glossary/content/software-defined-networking.html

[2] https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_largeur

[3]https://www.youtube.com/watch?v=k63VbjhsBA&list=PLccoFREVAt_4nEtrkl59mjjf5ZzRX8DZA&index=2

[4]https://www.youtube.com/watch?v=GveqSJ88aEo&list=PLIaCSU6kgAqYcA8MZSVkJqZ8uPdOJ38fD

[5]https://www.youtube.com/watch?v=7LBgXno9fcg&list=PLIaCSU6kgAqYcA8MZSVkJqZ8uPdOJ38fD&index=3