Institut de la Francophonie pour l'Informatique (IFI), Hanoi, Vietnam

Université catholique de Louvain, Département d'ingénierie informatique, IP Networking Lab (INL), Belgium

AMÉLIORATION ET IMPLÉMENTATION D'UN ALGORITHME D'ÉVITEMENT DES BOUCLES TRANSITOIRES DURANT LA CONVERGENCE D'OSPF

Promoteurs: Prof. **Olivier Bonaventure**
Dr. **Pierre François**

Mémoire présenté en vue de l'obtention du grade de Master en informatique par

Nguyen Van Nam

Louvain-la-Neuve
Année académique 2008–2009

Remerciements

Tout d'abord, je voudrais adresser mes plus grands remerciements au professeur Olivier Bonaventure pour son acceptation de mon stage au sein de l'équipe réseaux (INL), de l'Université catholique de Louvain. Sa grande vision dans le domaine des réseaux m'a également aidé à trouver de bonnes solutions.

Je voudrais remercier mon encadrement direct, Pierre François, pour sa responsabilité de mon stage et ses connaissances profondes du sujet. De nombreux courriels électroniques et les discussions de chaque semaine avec lui m'a aidé à petit et petit comprendre le problème. Son intelligence m'a également aidé de mieux contrôler mes travaux et de surmonter des difficultés.

Je souhaiterais remercier à Damien Saucez, mon collègue de l'INL, qui m'a donné de bons commentaires et des conseils tout au long de mon stage sur la façon de travail, sur la vie quotidienne, sur la procédure administrative et notamment sur l'élaboration de ce mémoire.

Je cordialement remercie Damien Leroy, mon collègue de l'INL, qui m'a aidé à implémenter des algorithmes en routeur XORP et l'environnement expérimental de réseau NETKIT.

Je voudrais remercier Bruno Quotin, mon collègue de l'INL, qui m'a aidé à générer des topologies de test en IGEN pour évaluer les algorithmes implémentés.

Je souhaiterais remercier Sébastien Barré, Virginie Van den Shriecke et Benoit Donnet, mes collègues de l'INL, pour les discussions utiles avec eux pendant mon stage.

Abstract

Transient loops occur in IP networks using link state routing protocols like OSPF or IS-IS during their convergence due to topology changes. They can be avoided by applying a sequence of metric (the state of link) on the changed link so that the networks can adapt step by step with the changes. An algorithm is used to minimize the sequence. We propose in this works, an alternative algorithm that can produce faster a less optimized sequence.

Résumé

Des boucles transitoires peuvent être provoqués dans les réseaux IP durant la convergence des protocoles de routage de type « état de lien » utilisés comme OSPF ou IS-IS à cause des changements de topologie. Elles peuvent être évitées par l'application d'une séquence des métriques (l'état de lien) sur un lien pour que les réseaux puissent petit et petit s'adapter aux changements. Un algorithme permet de minimiser cette séquence. Nous proposons dans ce travail, un algorithme alternatif permettant produire plus rapidement une séquence moins optimisée.

TABLE DES MATIERES

LISTE DES FIGURES

1 Introduction

Les *protocoles de routages* sont utilisés dans les réseaux IP pour déterminer les routes entre deux routeurs. Les protocoles de routage interne (IGP : Interior Gateway Protocol) comme OSPF (Open Shortest Path First), RIP (Routing Information Protocol) fonctionnent sur un seul système autonome (AS : Autonomous System). Par contre, les routeurs dans un AS peuvent trouver les routes à l'extérieur grâce aux protocoles de routage externes (EGP : Exterior Gateway Protocol) comme BGP (Border Gateway Protocol) [27].

Deux types populaires d'IGP sont : *les protocoles de routage statiques et ceux dynamiques.* Ces derniers permettent aux routeurs d'automatiquement choisir les meilleures routes. En fonction de la façon dont les routeurs communiquent à leurs voisins et l'algorithme qu'ils utilisent pour sélectionner les chemins, nous en distinguons deux types : *« état de lien »* comme IS-IS (Intermediate System to Intermediate System) ou OSPF et *« vecteur à distance »* comme RIP [28].

Les boucles de routages peuvent être provoquées si les routes calculées par les routeurs sont inconsistantes. Il y en a deux types : *boucles persistantes et celles transitoires*. Les premières sont causées par des fautes de configurations des routeurs. Elles durent longtemps (des heures) à travers plusieurs AS. Les deuxièmes se passent dans les protocoles de routages dynamiques. Pour les protocoles de type « vecteur à distance » comme RIP, il y a une situation connue : *le contage jusqu'à l'infini (counting à infinity)*. De l'autre côté, les boucles transitoires sont provoquées lorsqu'il y a des *changements de la topologie* des réseaux à cause du délai de *la convergence* dans les protocoles de type « état de lien ». Elles peuvent donc causer *des ruptures* des réseaux parce que chaque paquet a un temps d'expiration (TTL : Time To Live). Après avoir passé ce temps là, le paquet va être supprimé [24][22].

Les figures 1.1 et 1.2 expriment *les ruptures* dans l'épine dorsale de Sprint aux État-Unis pendant quatre mois (de Décembre, 2001 à Avril, 2002). Sprint utilise le protocole de routage de type "état de lien" IS-IS (Intermediate System to Intermediate System). La figure 1.1 représente le pourcentage (l'axe « Failure event ») des ruptures en fonction du temps : chaque jour, chaque heure et chaque cinq minutes. La figure 1.2 exprime la distribution cumulée (l'axe « Cumulative distribution ») de la durée des ruptures (l'axe « Failure duration »). Dans la figure 1.1, les ruptures de réseaux existaient tout le temps particulièrement le 16 Janvier et ça comptais 6% du nombre total des ruptures. *Il y a 10% des ruptures dont la durée est de plus de 20 minutes* (Figure 1.2). C'est probablement à cause des coupures des câbles, le remplacement des équipements matériels [12].

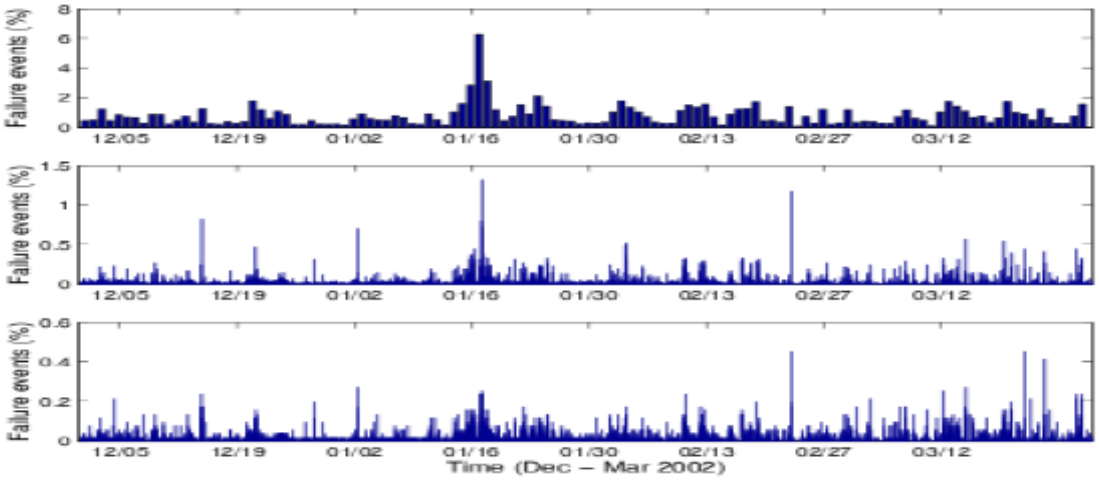

Figure 1-1: Ruptures du réseau en fonction du temps: chaque jour, chaque heure et chaque cinq minutes [12]

40% des ruptures durent d'une à moins de 20 minutes qui sont expliquées par la mise à jour des logiciels, le redémarrage des routeurs, etc. [12]. *Il y a 50% des ruptures durent moins d'une minute. Les causes de ces ruptures sont encore inconnues.* De plus, on a trouvé *que 47% des ruptures se passaient durant les opérations de maintenance* [12].

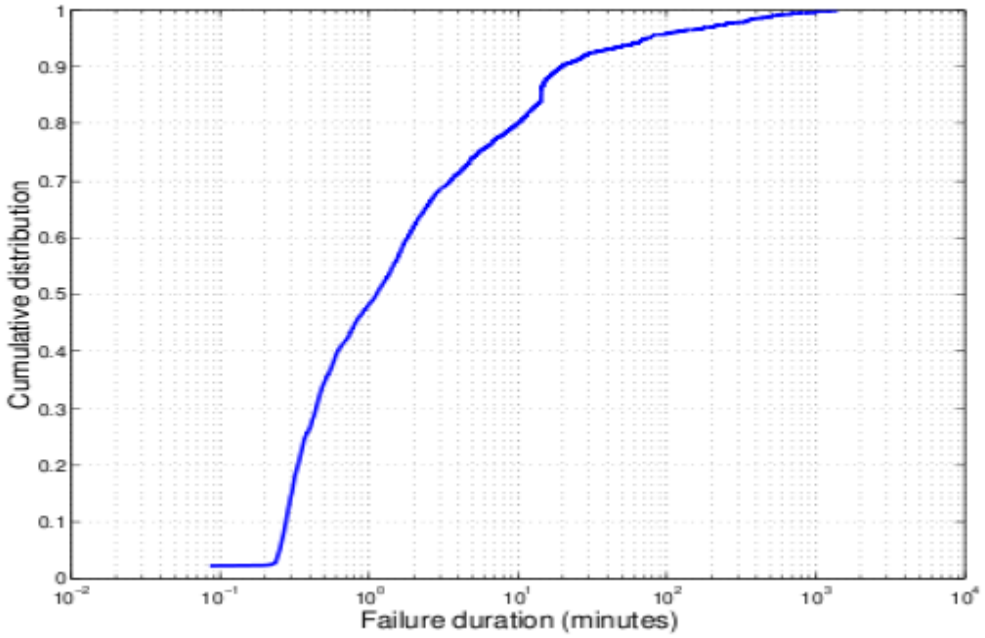

Figure 1-2: Durée des ruptures de réseau [12]

Dans littérature, **certaines solutions d'évitement des boucles transitoires durant la convergence d'OSPF** ont été proposées. Ces méthodes sont soit des modifications du protocole (oFIB : ordered FIB update)[1], soit des rejets des paquets (LISF : Loops through Interface-Specific Forwarding) [16]. L'idée d'oFIB est que si les FIBs sont mis à jour selon un ordre convenable lors du changement de la métrique d'un lien, il n'y aura pas de boucle dans le réseau. oFIB ordonne les routeurs dont le FIB est dans un arbre et si les fils de l'arbre actualisent avant leur père, les boucle seront évitées. oFIB modifie OSPF pour que les routeurs puissent négocier cet ordre. LISF associe les routes aux interfaces entrantes. Les paquets seront supprimés s'ils retournent au routeur à la prochaine fois, d'une autre interface. oFIB peut être utilisé pour éviter les boucles mais nous devons modifier le cœur du routeur et nous n'accepterons pas la perte de paquet selon LISF. C'est la raison pour laquelle ces approches ne sont pas populairement appliquées.

L'objectif de ce travail est d'améliorer et d'implémenter l'algorithme de reconfiguration de métrique (LIF) pour éviter des *boucles transitoire* durant la convergence du protocole de routage OSPF (Open Shortest Path First) lors des *changements de la topologie* en consistant à réduire le nombre des *ruptures* dans les réseaux IP.

L'algorithme de reconfiguration de métrique (LIF) : LIF est un nouvel algorithme qui a été présenté pour la première fois à INFOCOM 2007[1]. LIF considère les changements de la topologie d'un réseau qui causent ceux de la métrique des liens. LIF permet de détecter et éviter toutes les boucles possibles dans un réseau utilisant un protocole de routage de type « état de lien » (comme OSPF ou IS-IS) lors d'un changement de métrique étant donné le graphe représentant la topologie, un lien et sa nouvelle métrique. LIF est alors convenable à la phase de maintenance où nous souhaitons reconfigurer les interfaces d'un routeur.

L'idée de LIF est qu'au lieu d'appliquer immédiatement la nouvelle métrique sur le lien, nous calculons d'abord une séquence des métriques de l'ancienne à la nouvelle métrique de sorte que l'application de chacune ne provoque pas de boucles. LIF assure que l'application de la séquence sur le lien permet aux routeurs de mettre à jour les FIBs en ordre convenable (selon oFIB). LIF minimise également la longueur de séquence. La correction de LIF a été prouvée. Le succès de l'implémentation de cet algorithme dans XORP montre sa faisabilité. D'ailleurs, il est aussi efficace dans de grands réseaux grâce aux bons résultats de tests.

L'amélioration de l'algorithme : Néanmoins, LIF considère toutes les boucles comme les mêmes et le temps de calcul de LIF dépend du nombre des métriques de la séquence. En outre, nous trouvons que, dans LIF, la séquence à minimiser est encore longue. En analysant de différents types des boucles, nous voyons deux propriétés qui peuvent être utilisées pour réduire le nombre des métriques vérifiées et proposé un nouveau algorithme LE&DKM. LE&DKM est théoriquement plus lent mais en pratique il est plus rapide et moins optimisée que LIF. Toutefois, la raison de la rapidité de LE&DKM ouvre une nouvelle orientation de recherche sur ce problème.

La structure de ce mémoire : Ce mémoire présentera, d'abord, le protocole de routage de type « état de lien » OSPF. Nous analyserons, ensuite, la nature des boucles transitoires et le mécanisme pour les éviter. Nous introduirons différentes méthodes d'évitement des boucles et analyserons leur efficacité. Nous présenterons l'implémentation de deux méthodes en XORP et comparerons leurs performances.

Le chapitre 2 présente la vue générale d'OSPF. Nous concentrerons sur la description de l'algorithme Dijsktra utilisé pour calculer l'arbre des plus courts chemins d'un routeur OSPF aux autres. Nous aborderons aussi la manière de diffuser des LSAs (Link State Advertisement) entre les routeurs OSPF: des types de LSA concernés pour échanger des informations entre des routeurs, les zones utilisées pour diviser de grands réseaux et les types de liens qui connectent les routeurs OSPF.

Le chapitre 3 explique en détail les boucles transitoires. Nous voyons les nœuds influencés par le changement de la métrique d'un lien pour limiter le nombre de destinations à vérifier dans chaque itération de LIF. Nous discuterons aussi sur la potentialité des boucles sur de différentes topologies pour construire des cas de tests.

Dans le chapitre 4, nous concentrerons sur les deux algorithmes de reconfiguration des métriques (LIF et LE&DKM). Nous décrirons leurs idées: la construction des séquences des métriques (clefs pour LIF, décisives pour LE&DKM) et l'optimisation de LIF, montrerons leur correction, analyserons leur complexité afin de théoriquement les comparer. Particulièrement, nous présenterons le fonctionnement des algorithmes en pseudo code.

Dans le chapitre 5, nous présenterons notre implémentation des algorithmes en XORP pour éviter des boucles possibles causées par deux commandes du routeur OSPF. Nous expliquerons les étapes du développement et des difficultés rencontrées.

Le chapitre 6 présente la méthodologie de tests, des cas de test avec des réseaux réels et des réseaux générés. Nous évaluerons et analyserons des résultats pour voir l'efficacité de

LIF et la performance en practice de deux algorithmes (LIF et LE&DKM).

Lieu de stage : Ce mémoire de fin d'études a été réalisé au sein de l'équipe de réseau (INL) de l'Université catholique de Louvain en Belgique de Mars au Septembre, 2008. L'équipe est dirigée par le professeur Olivier Bonaventure. Les recherches de l'équipe sont applicables.

2 OSPF

2.1 Introduction

Les *protocoles de routage* spécifient la façon dont les routeurs se communiquent l'un avec l'autre pour diffuser des informations et qui leur permet de choisir des routes entre deux nœuds d'un réseau [27].

En fonction de la portée de fonctionnement des protocoles de routage, nous en distinguons deux types :

i) Les protocoles de routage internes (IGP - Interrior Gateway Protocol), comme RIP, IS-IS, EIGRP, OSPF, etc échangent les informations de routage dans un seul système autonome (AS).

ii) Les protocoles de routage externes (EGP-Exterior Gateway Protocol), comme BGP, EGP, échangent les informations de routage entre les systèmes autonomes (AS) [27].

En fonction de la manière dont nous utilisons pour déterminer les routes entre les routeurs, il y en a deux types des protocoles de routages : *statiques et dynamiques*. Les *protocoles de routage dynamiques* jouent un rôle important dans les réseaux de transit. Ils ont deux parties : *le protocole de routage* qui est utilisé entre les routeurs pour échanger des informations de leur réseau et *l'algorithme de routage* qui détermine les chemins via le réseau. Le protocole de routage définit la façon dont un routeur transfère des informations à l'extérieur, par contre, l'algorithme de routage traite les informations à l'intérieur du routeur [28].

Deux types communs des protocoles de routage dynamique sont :

i) « Vecteur à distance » : Les routes sont calculés par chaque routeur et sont échangés entre un routeur et ses voisins.

ii) « L'état de lien » : Les informations des voisins sont échangés entre les routeurs et ils calculent les routeurs en se basant sur la vue entière de la topologie du réseau [28].

OSPF est le protocole de routage IP interne (IGP) qui est largement utilisé sur Internet [9]. Dans ce chapitre, nous étudierons l'algorithme de routage Dijstra utilisé dans OSPF. Ensuite, nous étudierons la façon de la diffusion des informations entre les routeurs OSPF (OSPF version 2 pour IPv4).

2.2 Vue générale d'OSPF

OSPF est un protocole de type «état de lien». Chaque interface d'un routeur OSPF est assignée par une métrique (ou le coût) qui est considérée comme poids (ou l'état) du lien de ce routeur vers un autre pour les liens Point à Point et vers d'autres pour les liens Point à MultiPoint. Les routeurs OSPF échangent des états de liens associés à leurs interfaces avec leurs voisins [3].

Un routeur OSPF construit une vue entière de la topologie du réseau. Cette topologie est conceptuellement un graphe connecté, orienté et pondéré dans lequel chaque un sommet représente un routeur et les arêtes représentent les liens entre routeurs voisins. En se basant sur ce graphe et l'algorithme Dijkstra, chaque routeur calcule un arbre des plus courts chemins vers les autres destinations dont la racine est lui-même (Shortest Path Tree-SPT). Cet arbre est ensuite utilisé pour construire la table de routage du routeur et le FIB du routeur (Forwarding Information Base-FIB) [3].

Par rapport à RIP (Routing Information Protocol, un protocole de routage interne de type «vector à distance»)[19], le temps de convergence d'OSPF est particulièrement rapide (quelques secondes). En plus, OSPF consomme moins bande de passante: seuls les paquets HELLO, les LSAs (Link State Advertise) en période ou lors des changements de topologie sont envoyés [3].

Comme IS-IS (Intermediate Systeme to Intermediate System, un protocole de routage interne de type «état de liens» qui utilise Dijkstra)[20], OSPF demande aux routeurs d'une espace suffisante de mémoire pour stocker la LSDB (Link State Database) et d'un

processeur puissant pour calculer l'arbre des plus courts chemins notamment sur de grosses topologies. Sa configuration est complexe, particulièrement si le réseau est divisé en zones [3].
OSPF pour IPv4 (OSPF version 2) est détaillé dans RFC 2328 [29].

2.3 Algorithme Dijkstra

Cet algorithme a été proposé par Edsger Dijkstra en 1959 pour calculer les plus courts chemins dans un graphe dont les poids associés aux liens sont positifs ou nul en consistant à constituer un arbre (Shortest Path Tree-SPT) [7].
Étant donné un graphe G(V, E) où V, E est l'ensemble des N sommets et M arêtes, respectivement. Pour construire le SPT du nœud S qui appartient à V, on utilise deux sous ensembles P et Q où P contient des nœuds visités et Q contient ceux restants. Chaque nœud est assigné une étiquette qui se compose du nœud précédant et la distance temporaire dans le chemin de la racine (S). Au début, P ne contient que S et les étiquettes des (N-1) nœuds restants sont les mêmes: (S, ∞). À chaque itération, nous recalculons la distance temporaire et l'étiquette de chaque nœud en fonction de la procédure suivante (l'algorithme 2.1).

```
Pour chaque c in Q faire
   /* nous balayons toutes les noeuds qui ne sont pas visités   */
   {
    Pour chaque u in P do
      { /* nous recalculons la distance minimale temporaire et l'étiquette de chaque nœud  dans Q */
      Si u is neighbor of c et tentative_distance(c)>tentative_distance(u)+distance(u,c)
        alors
           {
           /* la nouvelle étiquette de c */
           label(c)=u;
             /* label(c)={u}, au cas d'ECMP:
           il y a plusieurs plus courts chemins d'un routeur vers   d'autres */

             /* la nouvelle distance minimale temporaire de c */
           tentative_distance(c)=tentative_distance d(u)    +distance(u,c)
           }
      }
}
```

Algorithme 2-1: L'assignation des étiquettes pour les nœuds à chaque itération de l'algorithme

Ensuite, les nœuds avec des distances temporaires minimales sont fixés et insérés dans P. Nous stockons également les étiquettes de ces nœuds. Cette boucle se termine lorsque Q est vide.

A partir des étiquettes des nœuds, nous pouvons trouver les arêtes d'un arbre des plus courts chemins (Shortest Path Tree (SPT)). C'est un arbre (un graphe connecté et sans boucle) parce que: il y a des chemins de la racine vers les autres nœuds (orienté, connecté) et s'il y a une boucle, la plus courte distance entre un nœud de la boucle et lui-même (0) sera égale à la longueur de la boucle.

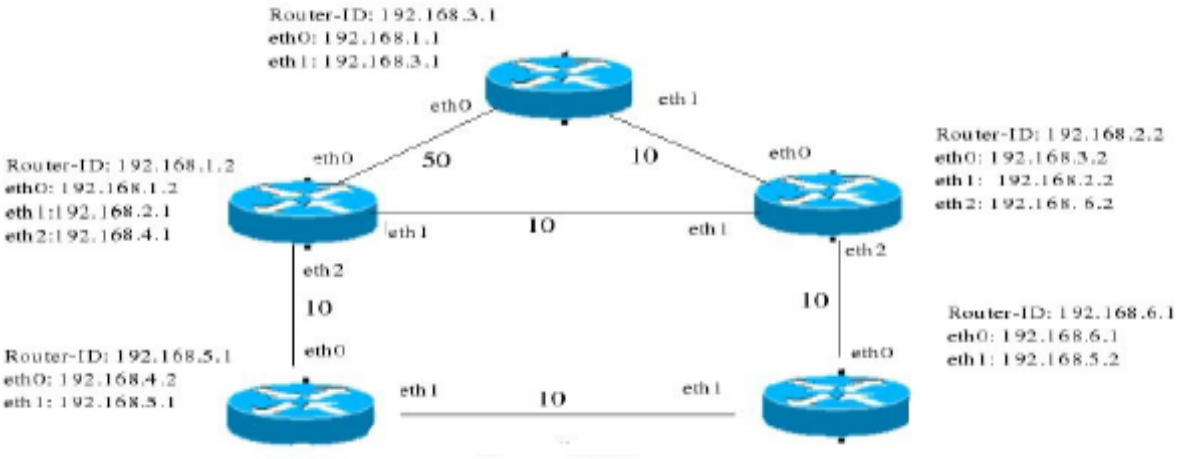

Figure 2-1: Une topologie d'un réseau OSPF

Par exemple, dans la topologie de la figure 2.1, le réseau contient cinq routeurs. Le routeur 192.168.3.1 contient deux interfaces dont les adresses sont 192.168.1.1/24 et 192.168.3.1/24. Le routeur 192.168.1.2 utilise l'interface eth2 avec l'adresse 192.168.4.1/24 pour se connecter au routeur 192.168.5.1 via l'interface 192.168.4.2/24. La métrique de ce lien est 10. Nous pouvons représenter la topologie du réseau sous forme d'un graphe comme suivant (la figure 2.2).

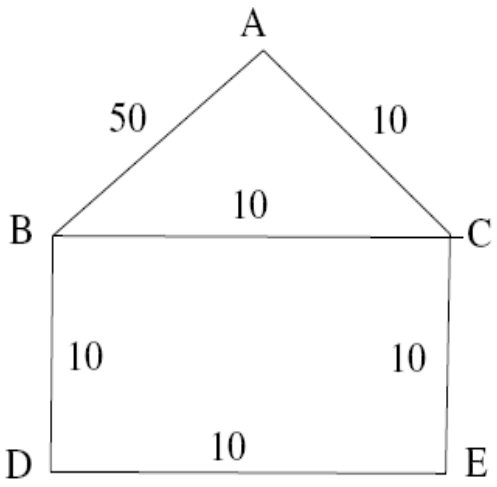

Figure 2-2: La représentation sous forme d'un graphe de la topologie

En plus, ce graphe peut être représenté par une forme compréhensive pour l'ordinateur :

une matrice des voisins (la figure 2.3).

Nœud	**Voisins (voisin : métrique)**
A	B :50 ; C :10
B	C :10 ; D :10 ; A :50
C	A :10 ; B :10 ; E :10
D	B :10 ;E :10
E	C :10 ;D :10

Figure 2-3: La matrice des voisins

Nous utilisons l'algorithme Dijkstra pour calculer les plus courts chemins du routeur C vers d'autres. Les itérations de l'algorithme sont dans la figure 2.4 et les étiquettes qui sont fixées (en anglais : fixed) sont dans la figure 2.5.

Itérations	A	B	C	D	E
0	(C, ∞)	(C, ∞)	(C,0)	(C,∞)	(C,∞)
1	(C,10)	(C,10)	Fixed	(C,∞)	(C,10)
2	Fixed	Fixed		(E,20), (B,20)	Fixed
3				Fixed	
4					

Figure 2-4: Les itérations de l'algorithme Dijkstra

Nœud	**Étiquette fixée (nœud précédent, distance)**	**Arête du SPT**
A	(C,10)	CA
B	(C,10)	CB
D	(E,20) ; (B,20)	ED,BD
E	(C,10)	CE

Figure 2-5: Les arêtes du SPT du routeur C

Notons que dans ce cas, il y a deux plus courts chemins de A vers D: ACBD et ACED (nous l'appelons Equal Cost MultiPath, ECMP). Pour traiter ce cas, nous avons modifié l'algorithme Dijsktra pour que les étiquettes temporaires de chaque nœud soient stockées dans une liste (l'algorithme 2-1).

Avec les arêtes dans la figure 2.5, le SPT du routeur C est reconstruit comme dans la figure 2.6.

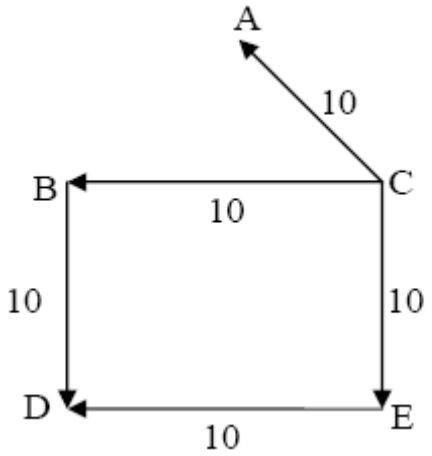

Figure 2-6: L'arbre des plus courts chemins du routeur C

2.4 La constitution de FIB

FIB (Forwarding Information Base) est une table qui contient les informations de « forwarding » d'un routeur. FIB est similaire à une table de routage ou une base d'informations. Il dynamiquement maintient une copie réflexive des informations de « forwarding » contenue dans la table de routage. Lorsqu'il y a un changement du routage ou de la topologie d'un réseau, la table de routage est mise à jour et ces changements sont référés dans le FIB. Le FIB maintient l'information de l'adresse du prochain hôte basant sur les informations de la table de routage [21].

Le FIB d'un routeur OSPF se base sur l'arbre des plus courts chemins du routeur vers d'autres.

Par exemple, en se basant sur l'arbre des plus courts chemins dans la figure 2-6, le FIB du routeur C de la topologie dans la figure 2.1 est dans la figure 2.7. Dans ce FIB, pour envoyer les paquets vers le routeur A (l'ID : 192.168.3.1), le routeur C

va les transférer à l'adresse est 192.168.3.1/24 (l'interface eth1 du routeur A) via son interface eth0. De même, les paquets vers routeur D(l'ID : 192.168.5.1) vont être transférés à l'adresse 192.168.2.1/24 (l'interface eth1 du routeur B) via son interface eth1 ou à l'adresse 192.168.6.1/24 (l'interface eth0 du routeur E) via son interface eth2 .

Destination	**Route** **(prochain hôte, via interface, distance)**
A	192.168.3.1/24, eth0, 10
B	192.168.2.1/24, eth1, 10
D	192.168.2.1/24, eth1, 20 192.168.6.1/24, eth2, 20
E	192.168.6.1/24, eth2, 10

Figure 2-7: Le FIB du routeur C de la topologie 2.1

2.5 Les zones

OSPF est utilisé dans des système autonomes (Autonomous System, AS) gérés par des fournisseurs des services (Internet Service Provider, ISP) qui développent maintenant de grands réseaux. Pour réduire le nombre des LSAs (Link State Advertise) envoyés ainsi que la taille de LSDB (Link State Database) que chaque routeur doit maintenir, OSPF propose la notation de zone qui permet aux administrateurs de diviser ces grands réseaux aux petites zones [3].

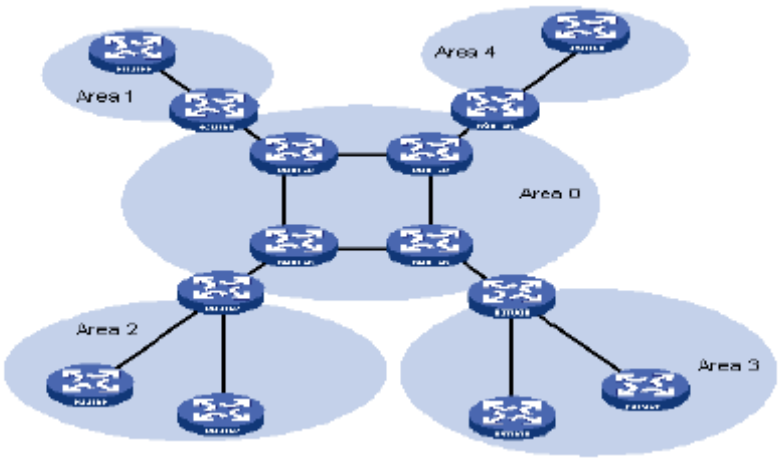

Figure 2-8: La partition en zone d'OSPF [9]

Les zones dans un réseau OSPF sont identifiées par area-ID. La figure 2.8 décrit un AS de cinq zones qui se connectent par des routeurs à la frontière (BR). La zone 0 dont l'ID est 0.0.0.0 (Area 0 dans la figure 2.8) est considérée comme l'épine dorsale. Toutes les autres zones (Area 1, 2, 3, 4 dans la figure 2.8) doivent être connectées physiquement à cette zone [9].

OSPF assigne un lien à exactement une seule zone. Le routeur dont les liens sont en plusieurs zones est appelé «routeur à la frontière» (Border Router, BR). Un routeur maintient le graphe complet de topologie de chaque zone à laquelle ses liens appartiennent. Le routeur ne connaît pas la topologie entière des zones à distance mais grâce aux routeurs à la frontière (BR), il connaît le poids total du chemin d'un ou plusieurs BRs de chaque zone à chaque routeur de cette zone. OSPF permet d'importer des informations de routage d'autres protocoles comme BGP (Border Gateway Protocol). Le routeur qui importe les informations d'autres protocoles à OSPF est appelé «routeur à la frontière d'AS» (AS Border Router) [6].

OSPF calcule les plus courts chemins en trois étapes. A la première étape, chaque routeur calcule le SPT pour la zone interne (*intra-area stage*). A la deuxième étape, il calcule les routes pour chaque routeur à distance en choisissant un routeur à la frontière comme nœud intermédiaire basé sur des informations du poids total (*inter-area stage*). En fin, la dernière étape, il calcule routes pour chaque nœud externe en choisissant un routeur à la frontière d'AS comme nœud intermédiaire (*external stage*) [6].
En réalité, pour réduire la complexité du calcul de SPT, la plupart des vendeurs des routeurs limitent la taille d'une zone entre 50 et 500 routeurs [8].

2.6 Types de LSAs (Link State Advertises)

Chaque routeur OSPF décrit sa connectivité locale dans un Link State Advertise (LSA). Ces derniers sont inondés aux autres routeurs dans le réseau qui leur permet à construire la vue entière de la topologie. L'ensemble des LSAs stockés en mémoire d'un routeur est

appelé Link State Database (LSDB)[6].

OSPF définit différents types de LSAs:

i) Router LSA (Type 1-LSA): ces LSAs sont origines de tous les routeurs et inondés dans une seule zone. Ce LSA décrit les informations des interfaces du routeur.

ii) Network LSA (Type 2-LSA): ces LSAs sont origines d'un routeur désigné (Designated Router, DR) et inondés dans une seule zone. Ce LSA contient une liste des routeurs connectés à un réseau.

iii) Network Summary LSA (Type 3-LSA): il est origine d'un routeur de frontière (BR) et inondé dans la zone associée au LSA. Chaque Summary-LSA décrit une route vers une destination à l'extérieur de la zone dans un AS (*inter-area route*)

iv) ASBR Summary LSA (Type 4-LSA): il est origine des routeurs de frontière (BR) et inondé dans la zone associé au LSA. Ce LSA décrit des routes vers le routeur de frontière d'AS (ASBR).

v) AS Extenal LSA (Type-5 LSA): ils sont origines des routeurs de frontière d'AS (ASBRs). Chaque LSA décrit une route vers un autre AS. [9].

2.7 Types de réseaux

Deux types de réseaux OSPF: les réseaux Point à Point (Point to Point) et les réseaux Point à MultiPoint (Multi-Access) . Trois possibles réseaux possibles sont: Broadcast Network, Non Broadcast Multi-Access Network et Point to MultiPoint Network:

i) Broadcast Network: Un message peut être envoyés à tous routeurs (par exemple Ethernet LAN).

ii) Non Broadcast Multi-Access Network (NBMA): il n'y a pas de possibilité de «broadcast» (Par exemple: ISDN, ATM, X25, Frame Relay).

iii) Point-to-Multipoint Network: ce type est utilisé en mode «group» du réseau Frame Relay.

[10].

2.8 Conclusion

Dans ce chapitre, OSPF est le protocole de routage de type « état de lien » qui est largement utilisé sur Internet. Les routeur OSPF utilisent l'algorithme Dijkstra pour calculer l'arbre des plus courts chemins (SPT) vers d'autres routeurs dans une zone. Basant sur ce SPT, les routes utilisées pour transférer les paquets du routeur sont constituées (FIB). Les routeurs OSPF utilisent deux types de LSAs: RouterLSA et Network LSA pour échanger les informations des états de tous les liens dans une zone. Différentes zones peuvent échanger des routes via les routeurs à la frontière (BR). Le routeur à la frontière d'AS permet aux routeurs dans un AS de se connecter à l'extérieur. Un routeur OSPF stocke les LSA dans une LSDB. Il y a deux types de liens entre les routeurs OSPF: Point à Point et Point à MultiPoint.

3 Boucle transitoire durant la convergence d'OSPF

3.1 Introduction

Les *boucles de routage (routing loops)* sont abordées depuis longtemps dans littérature. Il y en a deux types : les *boucles persistantes* et *celles transitoires*. Les premières peuvent durer des heures et sont causées par des fautes de configuration. Les deuxièmes sont plus courtes (à la minute ou seconde) et elles apparaissent lors de l'inconsistance des informations de routage. Les boucles persistantes sont plus difficiles à analyser parce qu'elles sont rares et elles se passent à travers plusieurs AS. Donc, nous concentrons, dans ce travail, sur l'analyse des boucles transitoires [24].

Dans les réseaux utilisant le protocole de routage de type « vecteur à distance » comme RIP où chaque nœud ne connaît pas la vue entière de la topologie du réseau, il y a une situation connue : *le contage jusqu'à l'infini (counting to infinity)* dans laquelle la métrique d'une route est répétitivement augmentée entre deux ou plus de deux nœuds jusqu'à ce qu'elle atteigne la valeur maximale [22].

Dans ce chapitre, nous étudierons le mécanisme des *boucles transitoires durant la convergence d'OSPF* lors du changement de la topologie d'un réseau.

Les boucles de routage peuvent être détectées par la trace de la répétition des instances *(replicas stream)* d'un paquet dans un seul lien [23].

Dans ce chapitre, nous étudierons une façon permettant de détecter et d'énumérer toutes les boucles transitoires possibles en basant sur la théorie de graphe, des types de boucles et leur potentialité aux différentes topologies.

3.2 Boucle transitoire durant la convergence OSPF

Dans OSPF chaque routeur calcule son SPT basant sur sa LSDB. Si le réseau est stable (il n'y a pas de changement de métrique des liens), les routes des FIBs des routeurs seront consistantes: il n'y a pas de boucles de transfert des paquets vers n'importe quelle destination du réseau parce qu'il n'y aura pas de boucles dans l'arbre des plus court

chemins de tous les autres vers cette destination.

Lorsqu'il y a un changement de métrique d'un lien, les échanges des LSAs ne permettent pas aux routeurs de mettre à jour leurs FIBs en même temps. Les routeurs près du lien le font avant ceux plus loin. Ils utilisent de nouvelles routes pour transférer leurs paquets. Ce trafic peut être envoyé vers les routeurs plus loin.

Pendant que les routes des routeurs plus loin restent les mêmes parce qu'ils ne reçoivent pas encore la notification du changement. Le trafic est donc re-transfert aux routeurs précédents. Cette boucle continue jusqu' à ce que les routeurs plus loin reçoivent la notification du changement (par un LSA). Lorsque tous les routeurs du réseau mettent à jour leurs FIBs, il n'y a plus de boucles. **Le processus du changement jusqu'à** ce que les routes de tous les routeurs soient consistantes est appelé la convergence d'OSPF. Sa durée est assez courte (à la minute) [1]. Les boucles sont donc transitoires pendant ce temps là.

Dans la topologie de la figure 2.1, supposons que la métrique du lien entre l'interface 192.168.2.1 du routeur 192.168.1.2 (B) et l'interface 192.168.2.2 du routeur 192.168.2.2 (C) est reconfigurée par 30.

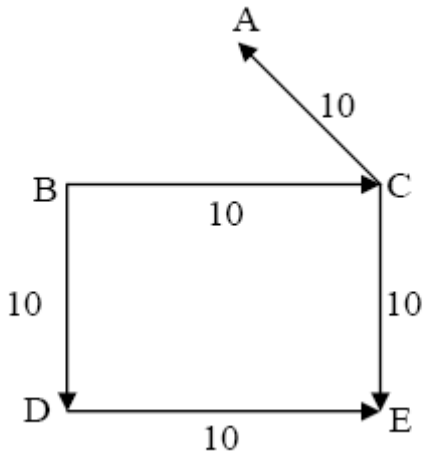

Figure 3-1: Le SPT du routeur B avant le changement

Destination	Route
A	Via C
C	directement à C
D	directement à D
E	Via C ou Via D

Figure 3-2: Le FIB de B avant le changement

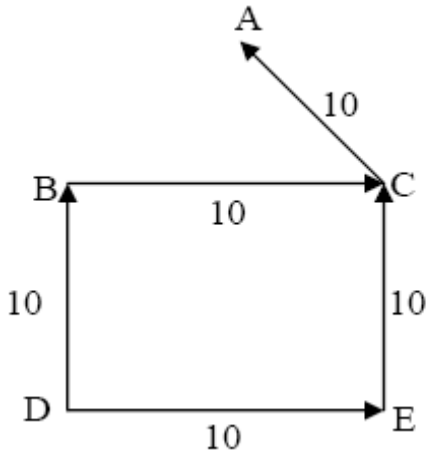

Figure 3-3: Le SPT du routeur D avant le changement

Destination	Route
A	Via B ou Via E
B	directement à B
C	Via B ou Via E
E	Directement à E

Figure 3-4: Le FIB de D avant le changement

Nous pouvons constater qu'avant la reconfiguration, selon le FIB de D, pour transférer ses paquets à la destination A, D les envoie à B (figure 3.2). Selon le FIB de B, ces paquets sont envoyés de B vers C et similairement de C vers A (figure 3.1). Les routes des FIBs des routeurs sont consistantes.

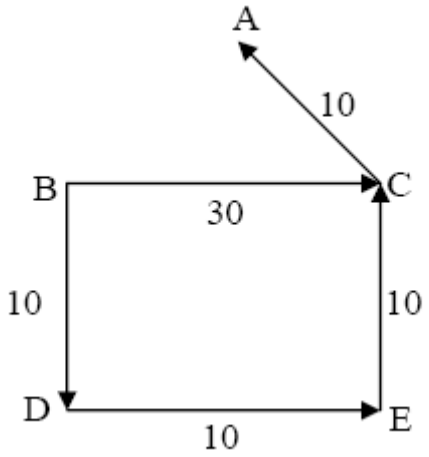

Figure 3-5: Le SPT du routeur B si la métrique du lien B→C est 30

Destination	Route
A	Via C ou via D
C	directement à C ou via D
D	Directement à D
E	Via D

Figure 3-6: Le FIB de D après le changement

Si la métrique du lien B→C est reconfiguré par 30, B mettra sa FIB avant D parce qu'il reçoit la notification de changement avant D. Selon le nouveau FIB de B (figure 3.5, 3.6), B transférera ses paquets vers C ou vers D pour la destination A. Pendant que D ne reçoit pas la notification de changement, il ne met pas encore sa table de routage. D envoiera les paquets reçus de B vers B. Il y aura donc une boucle entre B et D.
Lorsque D reçoit la notification de changement d'un LSA envoyé de B ou E, les routes deviennent consistantes et la boucle sera disparue.

3.3 Détection des boucles transitoires

Dans cette section, nous utiliserons le théorie de graphe pour analyser des boucles transitoires dans les réseaux OSPF.
Nous définissons, d'abord, *l'arbre inversé des plus courts chemins rSPT (reverse Shortest Path Tree)*.

Définition 3.3.1

Etant donné un graphe connecté et pondéré G=(V,E) où V est l'ensemble des nœuds et E est l'ensemble des arêtes, pour une destination D qui appartient à V, un lien X → Y qui appartient à E, le $rSPT_D$ (X→Y=m) est la fusion des plus courts chemins d'autres routeurs vers D dans le graphe lorsque la métrique du lien X → Y est égale à m [1].

C'est un arbre parce que comme analysé dans la partie 2.3, il est orienté et il n'y a pas de boucles.

Nous pouvons utiliser l'algorithme Dijkstra pour calculer cet arbre sur le graphe obtenu en échangeant les poids de chaque arête (ou la métrique des liens) du graphe, la métrique du lien X→ Y devient celle du lien Y → X et vice versa [1].

Comme analysé dans la partie 3.2, les boucles transitoires n'existent qu'au cas où il y a de nouvelles et d'anciennes routes utilisées dans le réseau. Le théorème 3.3.1 permet de détecter des boucles pour le transfert de paquets vers une destination.

Théorème 3.3.1

Étant donné un graphe G=(V,E), une destination D, un lien X→Y, *s'il y a une boucle transitoire dans le réseau lorsque la métrique du lien X→Y passe de m à M, les autres métriques ne changent pas, il y aura un cycle dans le graphe qui est la fusion de $rSPT_D$ (X→Y=m) et $rSPT_D$ (X→Y=M) et vice-versa[1].*

Un exemple est illustré dans les figures 3.4, 3.5, 3.6, 3.7.

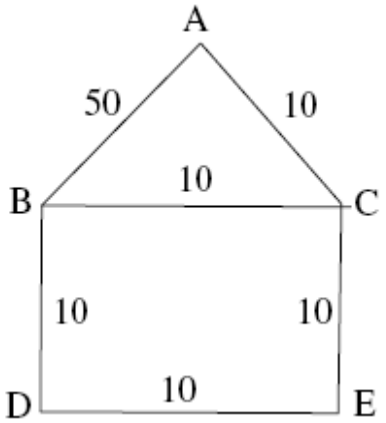

Figure 3-7: Le graphe original

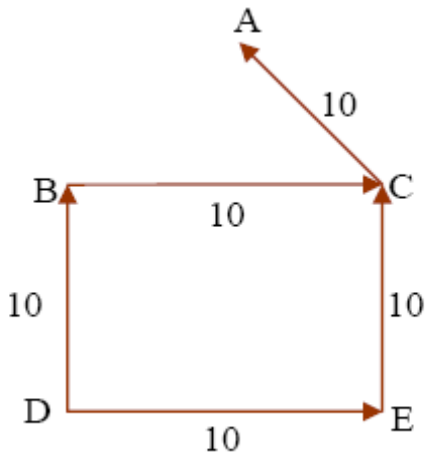

Figure 3-8: rSPT $_{A}$ (B→C=10)

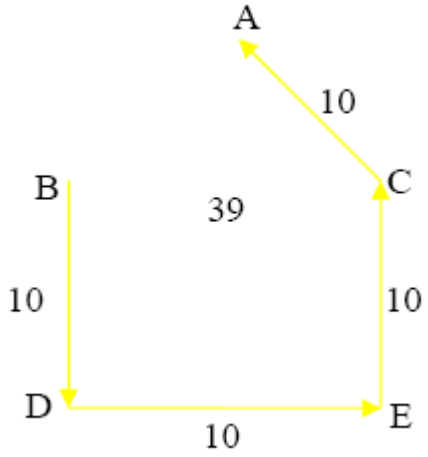

Figure 3-9: rSPT$_{A}$ (B→C= 39)

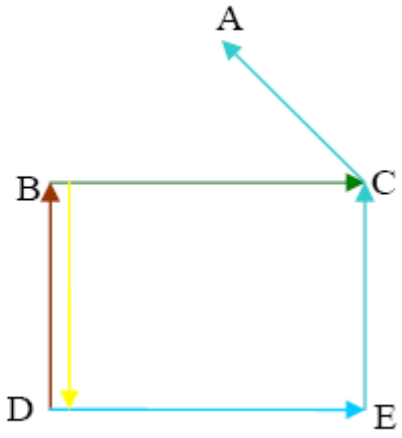

Figure 3-10: La fusion des rSPT$_{A}$ et une boucle *B→D→B*

Dans ces figures, lorsque la métrique du lien B→C passe de 10 à 39, il y a une boucle entre B et D parce que B met à jour son FIB avant D. Pour éviter cette boucle, il faut que D le fasse avant B. C'est l'idée d'un algorithme d'évitement des boucles qui s'appelle « ordered FIB update » présenté dans [1].

3.4 Destinations influencées par un changement de métrique d'un lien

Dans la partie 3.3, nous pouvons trouver les boucles possibles dans un réseau pour une destination D dans la fusion de deux $rSPT_D$: avant le changement et après la convergence. Si nous balayons toutes destinations, nous détecterons toutes boucles dans un réseau.

Néanmoins, lorsqu'il y a un changement de la métrique d'un lien $X \rightarrow Y$, seuls les chemins utilisent ce lien pour atteindre une destination Z sont influencées. Y est donc dans le plus court chemin de X vers Z.

En revanche, si Y n'est pas dans le plus court chemin de X vers Z, les routes vers Z des autres nœuds restent les mêmes lorsqu'il y a un changement de la métrique du lien $X \rightarrow Y$.

Par exemple, dans la figure 3.11, A est dans le plus court chemin de B vers E, mais non dans ceux de B vers C ou D. Lorsque la métrique du lien $B \rightarrow A$ change, le $rSPT_D$ et $rSPT_C$ ne change pas. Par contre, $rSPT_E$ peut être changé lorsque certains chemins vers E n'utilisent plus $B \rightarrow A$.

Cette idée est détaillée par la propriété 3.4.1.

Propriété 3.4.1 *Étant donné un graphe G=(V,E), un lien X→Y qui appartient à E, les destinations influencées par un changement de la métrique du lien X→Y se trouvent dans le sous arbre de l'arbre des plus courts chemins de X dont la racine est Y [1].*

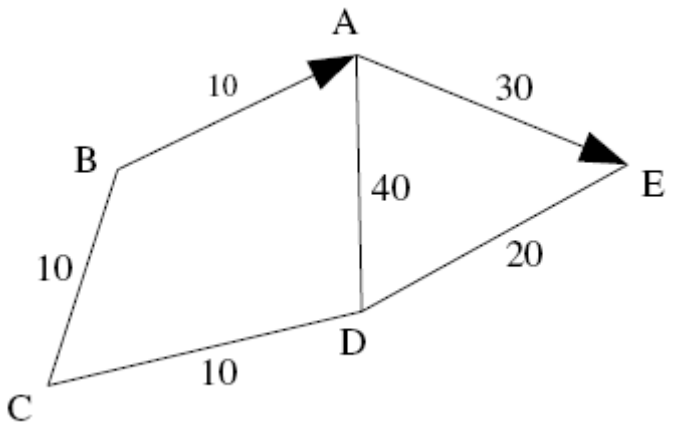

Figure 3-11: Destination influencée par un changement de la métrique du lien B→A: E

3.5 Analyse de topologie

Cette section étudie la potentialité d'occurrence des boucles transitoires durant la convergence d'OSPF aux topologies particulières.

Tout d'abord, nous distinguons deux types de boucles transitoires: ***micro boucle*** et ***macro boucle***.

Micro boucle ne contient que deux routeurs (la boucle longueur-2). La longueur des macro boucles est plus de 2.

Selon [1], on a fait une recherche de l'occurrence les micro boucles lorsqu'il y des ruptures des liens sur quatre topologies Tier1 A, Tier1 B, ISP 1, ISP 2. Tier 1 A contient 200 nœuds et 800 liens, Tier1 B se compose de 110 nœuds et 400 liens. ISP 1 et ISP 2 sont plus petits que les Tier1s: il y a 50 nœuds et 200 lien dans la topologie ISP 1 et 30 nœuds, 60 liens dans ISP 2.

Le résultat est que dans Tier 1-A, la rupture de 50% liens peut provoquer une micro boucle. Pour ISP 1, ISP 2 et Tier B, 40 % liens peuvent causer des micro boucles.

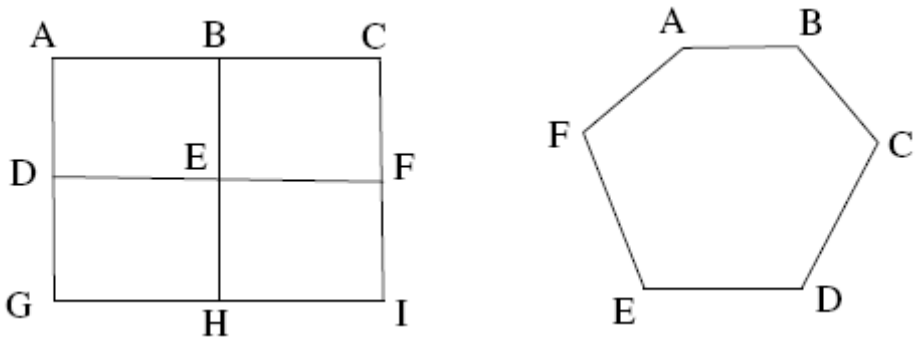

Figure 3-12: Topologies potentielles pour les boucles transitoire qui contiennent des carrés ou/et des anneaux

Concernant la forme des topologies, nous pouvons généralement constater que le nombre des boucles dépend du celui des ***anneaux*** (rings) comme dans la figure 3.12 et leur longueur dans le graphe du réseau. Alors que plus il y a de longs anneaux dans la topologie, plus des boucles transitoires apparaîtrent. En effet, dans un anneau, le changement de la métrique de chaque lien $X \rightarrow Y$ peut causer une boucle. Vu que pour deux nœuds A et B quelconques qui se situent dans le même anneau avec ce lien, il y a au moins deux routes entre eux : l'une passe le lien et l'autre non. Si le plus court chemin de A vers B passe $X \rightarrow Y$, A pourra changer son plus court chemin vers B lors du changement de la métrique de ce lien. Le changement du chemin de A vers B peut provoquer une boucle.

3.6 Conclusion

Dans ce chapitre, nous avons abordé les boucles de routage : les boucles persistances et celles transitoires. Les boucles transitoires durant la convergence d'OSPF sont à cause de la mise à jour des FIBs des routeurs lors du changement de la métrique d'un lien. Elles sont détectées dans la fusion de deux arbres inversés des plus courts chemins pour une destination. Les boucles pour toutes les destinations sont trouvées en ne balayant que celles influencées par le changement de la métrique d'un lien. En général, les topologies qui contiennent de longueur anneaux peuvent provoquer ces boucles.

4 Éviter des boucles transitoires par la reconfiguration des métriques

4.1 Introduction

Beaucoup de solutions d'évitement des boucles de routage ont été proposées dans littérature. Les solutions précédentes utilisent les échanges des messages entre les routeurs.

DUAL (Diffusion Update Algorithm) [24], une partie du protocole EIGRP de CISCO, est le plus connu algorithme d'évitement des boucles transitoires et le contage jusqu'à l'infini du protocole de routage de type « vecteur à distance » lors des changements de la topologie ou de la métrique d'un lien. La *condition de nœud source (Source Node Condition-SNC)* définit *l'ensemble des successeurs faisables* comme l'ensemble des voisins dont le *« coût vers la destination » (cost to destination)* courant est plus petit que celui déjà connu par le nœud. Un nœud peut choisir n'import quel voisin dans l'ensemble des successeurs faisables comme son *successeur (le prochain hôte - next hop)* sans besoin de notifier ses voisins, ni causer de boucles pourvu que ses voisins suivent ce règle [26].

Si le voisin vers lequel le coût vers la destination d'un nœud est minimal est dans l'ensemble des successeurs faisables, il sera choisi comme le successeur du nœud. Si cet ensemble est vide ou ne contient pas le meilleur successeur, le nœud va initialiser une procédure de la mise à jour synchrone, appelé *le calcul par la diffusion (diffusing computation),* en envoyant des requêtes à tous ses voisins et en attendant leur acquittement avant de changer de son successeur [26].

LFI (Loop Free Invariants Algorithm) [24] introduit une paire d'invariants basée sur le coût vers la destination d'un nœud et ses voisins. L'algorithme a prouvé que si les nœuds maintiennent ces invariants, aucune boucle n'est constituée [25].

Dans LFI, nous avons besoin un mécanisme pour maintenir la condition LFI. Il unifie deux approches pour être applicable aux deux types de protocoles : *Multiple-path Partial-topology Dissemination Algorithm* (MPDA) pour les protocoles de type « état de lien » et *Multipath Distance Vector Algorithm* (MDVA) pour les protocoles de type « vecteur à distance »[25].
Pourtant, l'efficacité de DUAL et de LFI dépend de la convergence des informations échangées entre les routeurs pour maintenir les conditions de faisabilité.

En utilisant la théorie de graphe, **oFIB (ordered FIB Update)** indique clairement l'ordre convenable de la mise à jour des FIBs. Avec le protocole de type « état de lien » comme OSPF ou IS-IS, les nœuds s'organisent dans un arbre inversé des plus courts chemins dont la racine est l'autre côté du lien. Pour éviter les boucles transitoires lors du changement de la métrique d'un lien, chaque nœud doit mettre à jour son FIB après ses fils [1].

Néanmoins, oFIB utilise la modification du cœur du protocole de routage pour que les nœuds puissent échanger des messages et négocier cet ordre.

Dans ce chapitre, nous étudierons une technique (**LIF- LIF Largest Increase First)** [1] d'évitement de boucles transitoires durant la convergence d'OSPF basée sur une reconfiguration progressive des métriques d'un lien dont l'état doit être modifié pour effectuer une maintenance.

LIF ne modifie pas le cœur du protocole, ni besoin d'échanges d'informations supplémentaires entre les routeurs.

Dans ce chapitre, nous proposons aussi une méthode alternative **(LE&DKM- Loop enumeraiton and decisive key metric)** pour calculer de la séquence de métrique à appliquer, et nous comparons la complexité théorique de ces méthodes.

4.2 Reconfiguration des métriques

La technique de reconfigurations des métriques peut être utilisée pour éviter des boucles transitoires. Lorsque le changement est non urgent, l'opérateur dispose du temps nécessaire pour effectuer des changements de métrique intermédiaires, étant donné que la connectivité est maintenue au moment du changement.

Cette technique se base sur la détection des boucles potentiellement créées par le changement d'état du lien, et sur le calcul d'une séquence de métriques intermédiaires. Cette séquence est telle que l'application successive des métriques ne crée pas de boucles transitoires. Afin de réduire le temps requis pour atteindre l'état final du lien (sa nouvelle métrique), le nombre des métriques de la séquence doit être minimisé.

La technique ne demande pas de modifications à OSPF, étant donné que la reconfiguration de métrique d'un lien est une fonctionnalité déjà présente dans les implémentations d'OSPF.

Si l'opération de maintenance du lien nécessite une désactivation complète du lien, la métrique terminale du lien est MAX_METRIC. Ainsi, la technique supporte également ce type de maintenance [1].

Dans cette partie, nous introduirons le concept d'ensemble de métriques clefs correspondant à un changement d'état de lien. Cet ensemble sert de base pour la constitution de la séquence de métriques permettant d'éviter les boucles transitoires.

Ensuite, nous présenterons le pseudo code de l'algorithme permettant de réduire la longueur de la séquence de reconfiguration et nous en analyserons la complexité.

Avant d'aller en le détail, nous introduisons quelques notations utilisées dans ce chapitre. Supposons un graphe G=(V,E), V est l'ensemble de sommets, E est l'ensemble d'arêtes et une destination D Є V.

1. **$Distance_D$ (N, X→Y =m):**

la plus courte distance du nœud N vers la destination D lorsque la métrique du lien X→Y est égale à m.

2. **d(X→Y) :**

la plus courte distance de X à Y dans le graphe initiale.

3. **KMS_D (X→Y:m, M):**

la séquence des métriques clefs pour la destination D lorsque la métrique du lien X→Y passe de m à M.

4. **RMS_D (X→Y:m, M):**

la séquence des métriques clefs de reconfiguration pour la destination D lorsque la métrique du lien X→Y passe de m à M.

5. **$ORMS_D$ (X→Y:m, M):**

la séquence optimisée des métriques clefs de reconfiguration pour la destination D lorsque la métrique du lien X→Y passe de m à M.

6. **$Path_{AB}$(X→Y=m):**

le plus court chemin de A vers B dans $rSPT_D$ (X→ Y =m) du graphe G

4.2.1 Métrique clefs

Comme analysé dans le chapitre 3, dans un graphe G=(V,E), une destination D Є V et un lien X→Y Є E, le changement de la métrique du lien X→Y influence le plus court chemin de certains nœuds du graphe vers D, par exemple nœud C. Le plus court chemin de C vers D (noté par r_1) passe par X→Y pour atteindre D. Si il existe un autre chemin de C vers D (par exemple r_2) ne passant pas par X→ Y, un changement de la métrique du lien X→ Y peut forcer C à choisir r_2 au lieu de r_1 pour atteindre D. La plus petite métrique du lien X→ Y *provoquant ce changement du plus courts chemin de C vers D est* appelée la métrique clef du lien *X*→ Y en correspondant à C.

Définition 4.2.1.1

Supposons un graphe connecté et pondéré G=(V,E), D Є V, X→Y Є E. Lorsqu'il y a un changement de la métrique du lien X→Y de m à M, la métrique clef du lien X→Y en

correspondant à un nœud N qui appartient à V est:

$$m + Distance_D (N, X \rightarrow Y = M) - Distance_D (N, X \rightarrow Y = m) \quad [1] \qquad (1)$$

$Distance_D$ (N, X→Y =m) est dans la notation 1) de la cette partie.

Nous reprenons l'exemple de la section 3.3: pour la destination A, les métriques clef pour le lien B→C correspondant aux nœuds B,D,E sont respectivement 30, 10 et 10. Dans ce cas, si la métrique du lien B→C passe à 30, B a deux plus courts chemins par rapport à 1. Si la métrique du lien B→C continue à augmenter (à partir de 31), B aura seulement un plus court chemin vers A, et celui-ci ne contient pas B→C .
Notons que pour un nœud N et une reconfiguration de lien $X \rightarrow Y$ donnés, il n'existe qu'une seule métrique clef.

4.2.2 Séquence des métriques de reconfiguration

La séquence ordonnée reprenant l'ensemble des métriques clefs correspondant à un changement d'état d'un lien est appelée « Key Metric Sequence » (KMS) [1]. Notons que cette séquence est croissante dans le cas d'une augmentation de la métrique d'un lien, et décroissant dans le cas de la diminution de la métrique d'un lien. Nous ne considérons que l'augmentation de la métrique d'un lien. Les techniques visant à gérer les cas de la diminution de la métrique d'un lien sont très similaires à celles des cas d'augmentation de métrique [1].

Définition 4.2.2.1

Supposons un graphe connecté et pondéré G=(V,E), D ∈ V, X→Y ∈ E, l'ensemble des métriques clefs du lien X→Y pour chaque nœud du graphe constitue une séquence des métriques clefs [1].

Dans l'exemple de la partie 3.3, chapitre 3, nous avons :

$$KMS_A(B \rightarrow C{:}10, 39) = \{10,30,39\}$$

Théorème 4.2.2.1

Étant donné G=(V,E), D ∈ *V,* X→Y ∈ *E, si la métrique de ce lien est augmentée par 1, il n'y a pas de boucle transitoire dans le réseau [1].*

Etant donné le Théorème 4.2.2.1, si nous augmentons progressivement la métrique du lien X→Y par 1 jusqu'à atteindre la métrique terminal, nous pourrons éviter toutes les boucles transitoires. Cependant, cette technique n'est pas réaliste étant donné que la séquence obtenue pourrait être extrêmement longue.

Dans [1], il a été démontré que la séquence des métriques de reconfiguration composée des métriques clefs et des métriques clefs plus 1 est une séquence qui permet d'éviter les boucles.

Dans l'exemple de la partie 3.3 du chapitre 3, nous avons :

$$KMS_A(B\rightarrow C{:}10, 39)=\{10,30,39\},$$

la séquence de reconfiguration correspondante, notée $pRMS_A(B\rightarrow C{:}10, 39)$, est égale à :

$$pRMS_A(B\rightarrow C{:}10, 39)=\{10,11,30,31,39\}$$

Définition 4.2.2.2

Étant donné un graphe G=(V,E), D ∈ V, X→Y ∈ E avec la métrique initiale m et la métrique terminale M, sa KMS correspondante

$$KMS_{X\rightarrow Y}(D)=\{km_1, km_2, ..,km_n\},$$

la séquence des métriques reconfiguration du lien X→Y, $nRMS_D(X\rightarrow Y{:}m, M)$ est

$$\{rm_0, rm_1, rm_2, .., rm_n, rm_{n+1}\}$$

où

$$rm_0 = km_0 = m, \text{ et } rm_{n+1}=km_{n+1}=M \text{ et } km_{i-1}<rm_i<km_i \text{ pour } i=1..n+1$$

Dans l'exemple de la section 3.3, la séquence des métriques de reconfiguration correspondant à cette définition, notée $nRMS_A(B\rightarrow C{:}10, 39)$, est égale à :

$$nRMS_A(B\rightarrow C{:}10, 39)= \{10,11,31,39\}.$$

Nous constatons que l'application de cette séquence provoquera une mise à jour des FIBs des routeurs du réseau correspondant à l'ordre introduite dans le chapitre 1. Cette constations est détaillée dans le Théorème 4.2.2.2.

Théorème 4.2.2.2

Étant donné un graphe G=(V,E), D ∈ V, X→Y ∈ E avec la métrique initiale, terminale m. En appliquant progressivement nRMS $_{X→Y}$ (D) sur le lien X→ Y, il n'y a pas de boucle transitoire dans le réseau représenté par G.

Preuve

Etant donné une séquence

$$nRMS_{X→Y}(D) = \{rm_0, rm_1, rm_2, .., rm_n, rm_{n+1}\},$$

Nous allons montrer qu'il n'y pas de cycle dans la fusion de :

$rSPT_D$ $(X→ Y = rm_i)$ *et* $rSPT_D$ $(X→ Y = rm_{i+1})$, *pour tout* $i ∈ [0,n]$.

Notons que $\{km_1, km_2, .., km_n\}$ est la séquence des métriques clefs pour X→ Y correspondant à la destination D.

1. Etant donné que $km_{i-1} < rm_i < km_i$, et qu'il n'y a pas de changement dans les chemins lors du passage du poids de X→Y de rm_i à km_i-1, on a que $rSPT_D$ (X→Y=rm_i) et $rSPT_D$ (X→Y=km_i-1) sont identiques.
2. $rSPT_D$ (X→Y=rm_{i+1}) est un sous graphe de $rSPT_D$ (X→Y=km_i) car
   1. $km_i < rm_{i+1} < km_{i+1}$.
   2. Il ne peut y avoir que des retraits de liens dans les rSPT correspondant au passage d'une métrique clef à la métrique clef suivante moins 1 [1].
3. Selon le **Théorème 4.2.2.1** , il n'y a pas de cycle dans la fusion de $rSPT_D$ (X→Y=km_i-1) et $rSPT_D$ (X→Y=km_i).
4. Etant donné (1) et (3), il n'y a pas de cycle entre $rSPT_D$ (X→Y=rm_i) et $rSPT_D$ (X→Y=km_i).

5. Etant donné (4) et (2), *il n'y pas de cycle dans la fusion de* $rSPT_D$ $(X \rightarrow Y = rm_i)$ *et* $rSPT_D$ $(X \rightarrow Y = rm_{i+1})$.

Le théorème est donc prouvé *!!!*

L'exemple de la section 3.3 va illustrer l'application des métriques basée sur le Théorème 4.2.2.2, pour la reconfiguration du poids du lien B→C de 10 à 39. Nous avons eu :

$$nRMS_A(B \rightarrow C{:}10, 39) = \{10,11,31,39\}.$$

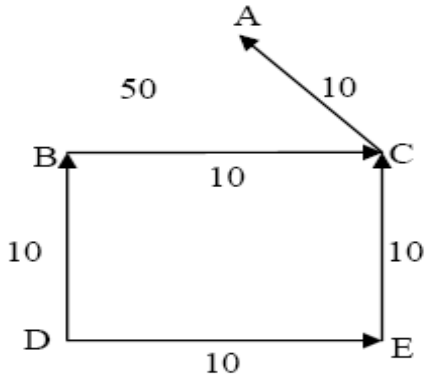

Figure 4-1: rSPT $_A$(B→C=10)

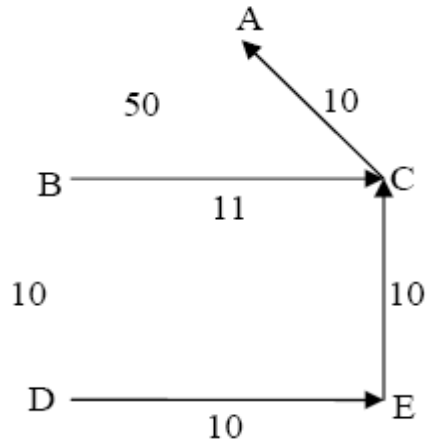

Figure 4-2: rSPT $_A$(B→C=11)

La figure 4.1 exprime l'arbre initial des plus courts chemins de tous les nœuds du graphe vers la destination A. Supposons que la métrique du lien B→C passe de 10 à 39. Dans ce cas,

$$nRMS\ B \rightarrow C\ (A) = \{10,11,31,39\}.$$

Initialement, D a deux plus courts chemins pour atteindre A: l'un via B et l'autre via E. Dans la figure 4.2, la métrique de B→C passe à 11, D n'utilise plus le chemin via B vers A. Les plus courts chemins d'autres nœuds ne changent pas.

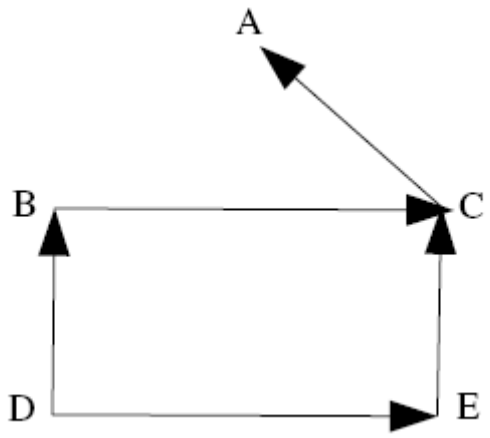

Figure 4-3: La fusion du $rSPT_A$(B→C=10) et $rSPT_A$(B→C=11)

Dans la figure 4.3, il n'y a pas de cycles dans la fusion des rSPT de A lorsque la métrique du lien B→C passe de 10 à 11. Il n'y a donc pas de boucles dans le réseau lors de cette transition (Théorème 4.2.2.1).

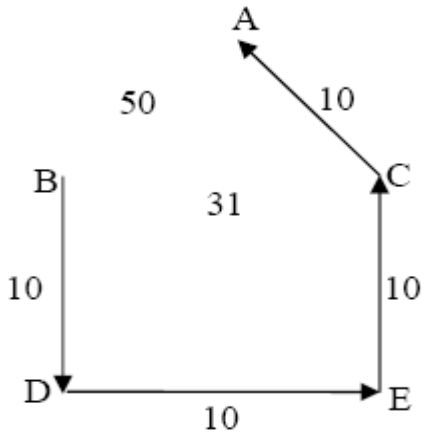

Figure 4-4: $rSPT_A$(B→C=31)

Dans la figure 4.4, la métrique de B→C passe de 11 à 31, par rapport à l'arbre précédent $rSPT_A$ (B→C=11), seul B change de son plus court chemin vers A. En plus, l'arbre des

plus courts chemins lorsque le poids de B→C est égal à 29 est le même que quand le poids de B→C est égal à 11 (point 2 du théorème 4.2.2.2).

Notons que la fusion de $rSPT_A$(B→C=11) et de $rSPT_A$(B→C=30) est égal à la fusion de $rSPT_A$(B→C=29) et $rSPT_A$(B→C=30). Le passage de la métrique de B→C de 30 à 31 forcerait B à arrêter d'utiliser le lien B→C pour joindre A. $rSPT_A$(B→C=31) est donc un sous ensemble de $rSPT_A$(B→C=30), et la fusion $rSPT_A$(B→C=11) et $rSPT_A$(B→C=31) ne contient donc pas de cycle (la figure 4.5).

Cet exemple illustre donc bien que l'application conjointe des théorèmes 4.2.2.1 et 4.2.2.2 permet de montrer que le passage de 11 à 31 est sans boucle.

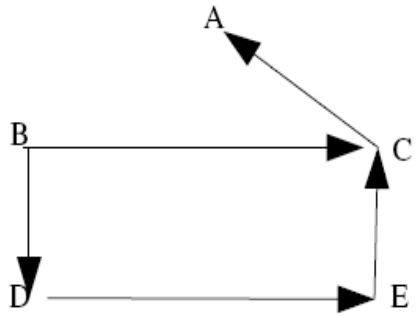

Figure 4-5: La fusion du $rSPT_A$(B→C=11) et $rSPT_A$(B→C=31)

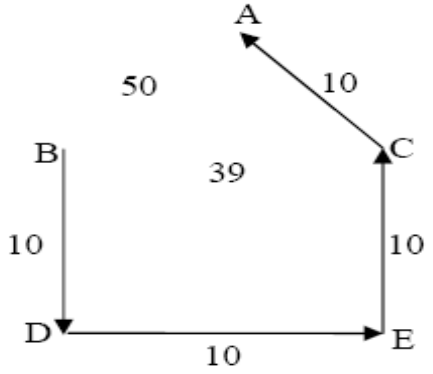

Figure 4-6: $rSPT_A$(B→C=39)

Dans les figures 4.4 et 4.6, le $rSPT_A$(B→C=31) et le $rSPT_A$(B→C=39) sont les mêmes. En effet, le lien B→C n'est pas utilisé lorsque la métrique du lien à égale à 31. Les plus

courts chemins dans le graphe sont donc identiques à ceux obtenus lorsque la métrique de lien est égale à 39. Ainsi, il n'y a pas de cycles dans la fusion de ces deux rSPTs .

Remarquons qu'il n'y pas non plus de cycle dans la fusion de $rSPT_A(B \rightarrow C=11)$ et le $rSPT_A(B \rightarrow C=39)$. Autrement dit, nous n'avons pas besoin d'appliquer la métrique 31 sur le lien.
Une séquence optimale pour effectuer la maintenance est donc :

$$nRMS\ B \rightarrow C\ (A) = \{10,11,39\}$$

au lieu de

$$nRMS\ B \rightarrow C\ (A) = \{10,11,31,39\}.$$

La raison pour laquelle il y a des métriques inutiles dans la séquence de reconfiguration est que chaque élément de cette séquence provient de l'évitement d'une boucle potentielle. Cependant, la métrique correspondant à l'évitement d'une boucle peut également servir à l'évitement d'une autre boucle, bien que la métrique clef correspondant au nœud créant la boucle soit différente.

Dans la partie suivante, nous présenterons l'algorithme qui permet minimiser la séquence proposée dans [1].

4.2.3 Optimisation de séquence

Dans cette section, nous présenterons l'algorithme *Largest Increase First* (LIF) [1] permettant d'optimiser les RMS. L'idée est que pour chaque métrique clef, nous chercherons la plus grande augmentation de métrique possible (parmi l'ensemble des métrique clefs), qui ne produit pas de boucle. Cette recherche est effectuée en utilisant la recherche binaire et la technique de détection des boucles décrite à la section 3.3. En plus détail, supposons

$$RMS_D\ (X \rightarrow Y{:}rm_1,\ rm_n) = \{rm_1,\ rm_2,\ldots,rm_n\}.$$

Pour trouver l'augmentation la plus longue de rm_1, nous vérifions la présence de boucle lors de la transition entre rm_1 et rm_n. Si des boucles sont détectées, nous continuons la vérification avec la métrique clef rm_i qui est au milieu de rm_1 et rm_n. Cette procédure se termine jusqu'à ce que nous trouvions le plus grand j tel que le passage de rm_1 à rm_j ne cause pas des boucles. Toutes les métriques entre rm_1 et rm_j sont supprimées de la séquence. Nous continuerons ensuite en partant de rm_j dans RMS_D $(X \rightarrow Y: rm_j, rm_n)$.

Par exemple, avec la séquence $pRMS_A$ (B→C:10, 39) = {10, 11, 30, 31, 39}, pour la métrique 10, nous avons vérifié qu'il y des boucles entre 10 et 39.

Nous cherchons par recherche binaire la plus grande transition sans boucle depuis 10 et nous trouvons que la métrique 11 est la plus grande qui ne crée pas des boucles. Nous effectuons ensuite la même opération à partir de 11. Il apparaît que le passage de 11 à 39 est sans boucle. Les valeurs 30 et 31 peuvent donc être supprimées.

La séquence obtenue $pORMS_A$ (B→C:10, 39), est {10, 11, 39}.

4.2.4 Implémentation de LIF en pseudo code

Dans cette partie, nous présenterons l'implémentation de LIF en pseudo code proposé dans [1].
La première procédure correspond au calcul de la séquence de reconfiguration étant donné un lien et une métrique terminale pour ce lien.
Le programme en pseudo code se compose de quatre petits modules: le premier est pour combiner toutes les RMS optimisées, le deuxième calcule un RMS pour une destination, le troisième sert à optimiser un RMS et le quatrième permet de stocker les rSPTs dans une cache afin d'éviter les re-calculs de rSPT lors de la vérification de présence de boucles lors d'une transition entre deux métriques clefs.

```
/* L'augementation de métrique à mt pour le lien L : A → B */
/* Le calcul des destinations inluencées */

        AffectedDest = Suivre(A → B, SPTinit (A));

/* Le calcul des ORMS */
        ORMSSet = {};

Pour tout Destination D Є AffectedDest faire
        {
        RMS = PrendreRMS(D,A → B, mt );
        ORMS = OptimiserRMS(D, RMS, A → B, L.metric, mt );
        ORMSSet.ajouter(ORMS);
        }
        MergedRMS = Sequences_Combiner(ORMSSet);
```

Algorithme 4-1: La combinaison des ORMS pour toutes les destinations

```
MetricSequence PrendreRMS(Destination dest, Link L, Metric target_metric)
        {

        RMS={L.metric, target_metric};

        /*  Le calcul de rSPT pour la destination dest et la métrique initiale de L  */

        initialRSPT=calculer_rSPT(dest, L,L.metric);

        /* Le calcul de rSPT pour la destination dest et la métrique finale de L  */

        targetRSPT= calculer_rSPT (dest, L,target_metric);
        Pour tout Node S | PathLength(S,D,initial_rSPT) ≠ PathLength(S,D,target_rSPT)
        faire  {
        KeyMetric=L.metric + PathLength(S,D,targetRSPT)- PathLength (S,D,initialRSPT);
        RMS.add(KeyMetric);
        /*   ajouter des métriques intermédiaires   */
        Si  (KeyMetric ≠ target_metric ) RMS.add(KeyMetric +1);
        }
        retourne RMS;
        }
```

Algorithme 4-2: La calcul de RMS pour une destination

```
MetricSequence OptimiserRMS (Destination D, MetricSequence RMS, Link L, Metric
StartMetric, Metric TargetMetric)
        {
        tempORMS = {StartMetric};
        currentMetric = StartMetric;
        Tant que (!(currentMetric==TargetMetric))  faire

        /* trouver la plus longue métrique M dans RMS de sorte que la transition de la
        métrique courante à M ne cause pas de boucle pour la destination D*/
        {
        M = TargetMetric;
        bool loopfree=false;
        Tant que (! loopfree) faire
                {
                MergedrSPT = combiner(rSPT(D,L,currentMetric),rSPT(D,L,M));
                Si MergedrSPT.containsCycle() alors M = La Métrique Avant M dans RMS;
                sinon loopfree = true;
                }

        tempORMS.ajouter(M);
        CurrentMetric = M;
        }
        retourne tempORMS;
        }
```

Algorithme 4-3: L'optimisation de RMS pour une destination

```
ShortestPathTree rSPT(Destination Dest, Link L, metric m)
{
        Si (rSPTCache.contains(Dest,L,m)) alors
        retourne prendre_rSPTCache(Dest,L,m) ;
        Sinon
                {
                rSPT = Calculer  rSPT de Dest  où  la métrique de L est mise à m;
                insérer_rSPTCache(Dest,L,m,rSPT);
                }
}
```

Algorithme 4-4: Le calcul de rSPT en utilisant cache

4.2.5 Analyse de complexité

La complexité de l'algorithme réside essentiellement dans l'optimisation des RMS (LIF). Cette dernière se compose de deux phases importantes: la recherche binaire et la vérification de la présence de boucles. La complexité de la première est O(logL) où L est la longueur de la RMS [17]. Selon la définition de RMS, la valeur maximale de L est égale au nombre de nœuds (N) du graphe de réseau.

Pour la deuxième phase, étant donné un graphe G=(V,E) la complexité de l'algorithme Dijkstra est O(|E| log N). En somme, la complexité de l'algorithme LIF pour une destination est O(|E| log² N) et, pour toute les destinations, est O(N |E| log² N) .

Si nous limitons le nombre d'arrêtes du graphe est de O(N), la complexité de cet algorithme pour une destination est de $O(N \log^2 N)$ et pour toutes destinations est de $O(N^2 \log^2 N)$.

4.3 Énumération des boucles et métriques clefs décisives (Loop enumeration and decisive key metric-LE&DKM)

Dans un graphe G=(V,E), nous utilisons les notations suivantes:

1) $\mathbf{Cycles_{X\rightarrow Y}(D)}$

pour indiquer l'ensemble des cycles présents dans la fusion de $rSPT_D(X\rightarrow Y=m)$

2) $\mathbf{rSPT_D(X\rightarrow Y=M)}$

lorsque la métrique du lien X→Y passe de m à M.

3) $\mathbf{KM_{X\rightarrow Y}(N)}$:

la métrique clef du lien X→Y correspondant au nœud N du graphe G.

4.3.1 Métrique clef décisive (Decisive Key Metric- DKM)

Une métrique clef est dite « décisive » pour un cycle donné si, elle est la plus petite des

métriques de KMS qui, au moment de leur application, peuvent provoquer une boucle transitoire. Nous allons montrer que l'application d'une métrique intermédiaire égale à la métrique décisive moins 1 sur le lien permet d'éviter la boucle transitoire correspondant à ce cycle.

Définition 4.3.1.1

Étant donné un graphe G=(V,E), D ∈ V, X→Y ∈ E avec la métrique initiale de X→Y=m et la métrique terminale de X→Y = M, tel que

$$KMS_{X\rightarrow Y}(D)=\{km_1, km_2, ..,km_n\},$$

une métrique clef km est décisive pour un cycle C ∈ $Cycles_{X\rightarrow Y}(D)$, résultat de la fusion de

$$rSPT_D(X\rightarrow Y=km) \text{ et de } rSPT_D(X\rightarrow Y=m),$$

si :

pour tout km'<km, C n'est pas dans la fusion du $rSPT_D(X\rightarrow Y=km')$ et le $rSPT_D(X\rightarrow Y=m)$.

Dans l'exemple de la partie 3.3, chapitre 3, $KMS_A(B\rightarrow C:10, 39)=\{10,30,39\}$, la métrique clef décisive pour le cycle B→D→B est 30 parce que si 30 est appliquée à B→C, une boucle apparaît alors que si 29 est appliquée, la fusion des rSPTs correspondant ne contient pas de cycle.

4.3.2 La recherche de DKM d'une boucle

Nous présenterons dans cette section, deux propriétés permettant trouver DKM pour chaque cycle dans la fusion de deux rSPT.

La première propriété concerne la relation entre un nœud et ses fils dans l'arbre $rSPT_Y$ (X→ Y =m) [1]. L'idée est que si un nœud change son chemin via son fils pour atteindre la destination après le changement de la métrique d'un lien, une micro boucle sera constituée entre deux nœuds et nous pouvons appliquer la métrique du nœud moins 1 sur le lien pour que le fils mette à jour son FIB avant son père.

Propriété 4.3.2.1

Étant donné un graphe G=(V,E), D ∈ V, X→Y ∈ E avec la métrique initiale de X→Y=m et la métrique terminale de X→Y = M , tel que

$$KMS_{X\rightarrow Y}(D)=\{km_1, km_2, ..,km_n\},$$

A, B ∈ V dans un même cycle C ∈ $Cycle_{X\rightarrow Y}(D)$.

Si :

$$Path_{AB}(X\rightarrow Y=m) \in rSPT_D(X\rightarrow Y=m) \text{ et}$$

$$KM_{X\rightarrow Y}(A) > KM_{X\rightarrow Y}(B)$$

Alors, $KM_{X\rightarrow Y}(A)$ *n'est pas décisive pour C.*

Preuve

Cas 1: $KM_{X\rightarrow Y}(A) > KM_{X\rightarrow Y}(B)$

1. Étant donné $KM_{X\rightarrow Y}(A) > KM_{X\rightarrow Y}(B)$: $rSPT_D(X\rightarrow Y= KM_{X\rightarrow Y}(B))$ est un sous ensemble de $rSPT_D(X\rightarrow Y= KM_{X\rightarrow Y}(A))$ (cfr point 2, Théorème 4.2.2.2).
2. Si $Path_{BA}(X\rightarrow Y=KM_{X\rightarrow Y}(B)) \in rSPT_D(X\rightarrow Y= KM_{X\rightarrow Y}(B))$, étant donné (1), on a:
   a) $C=(Path_{AB}(X\rightarrow Y=m) \cup Path_{BA}(X\rightarrow Y=KM_{X\rightarrow Y}(B)) \in$ la fusion de $rSPT_D(X\rightarrow Y=m)$ et $rSPT_D(X\rightarrow Y= KM_{X\rightarrow Y}(B)$
   b) Étant donné a), $KM_{X\rightarrow Y}(A) > KM_{X\rightarrow Y}(B)$ et la définition 4.3.1.1, $KM_{X\rightarrow Y}(A)$ n'est pas décisive pour C
3. Si $Path_{BA}(X\rightarrow Y=KM_{X\rightarrow Y}(B)) \in rSPT_D(X\rightarrow Y= KM_{X\rightarrow Y}(B))$:
   a) Étant donné la définition 4.3.1.1, $Path_{BA}(X\rightarrow Y=KM_{X\rightarrow Y}(B)) \in rSPT_D(X\rightarrow Y= KM_{X\rightarrow Y}(A))$
   b) Étant donné a), on a que $C=(Path_{AB}(X\rightarrow Y=m) \cup Path_{BA}(X\rightarrow Y=KM_{X\rightarrow Y}(B)) \in$ la fusion de $rSPT_D(X\rightarrow Y=m)$ et $rSPT_D(X\rightarrow Y= KM_{X\rightarrow Y}(A)$, $KM_{X\rightarrow Y}(A)$ n'est pas décisive pour C

Cas 2: $KM_{X\rightarrow Y}(A) < KM_{X\rightarrow Y}(B)$

1. Étant donné $KM_{X\rightarrow Y}(A) < KM_{X\rightarrow Y}(B)$, $rSPT_D(X\rightarrow Y= KM_{X\rightarrow Y}(A))$ est un

sous ensemble de $rSPT_D(X \rightarrow Y = KM_{X \rightarrow Y}(B))$ (le point 2, Théorème 4.2.2.2).

2. Étant donné (1) et la définition 4.3.1.1, $Path_{BA}(X \rightarrow Y = KM_{X \rightarrow Y}(B)) \in rSPT_D(X \rightarrow Y = KM_{X \rightarrow Y}(A))$
3. Étant donné (2), $C = (Path_{AB}(X \rightarrow Y = m) \cup Path_{BA}(X \rightarrow Y = KM_{X \rightarrow Y}(B)) \in$ la fusion de $rSPT_D(X \rightarrow Y = m)$ et $rSPT_D(X \rightarrow Y = KM_{X \rightarrow Y}(A), KM_{X \rightarrow Y}(A)$
4. Étant donné la définition 4.4.2.1 et (3), $KM_{X \rightarrow Y}(A)$ n'est pas décisive pour C

La propriété est donc prouvée !

La deuxième propriété concerne la métrique clef décisive pour chaque macro-boucle. Elle exprime que s'il y a un cycle entre des nœuds cousins (deux nœuds qui n'on pas de relation de parenté dans le graphe), l'application de la plus grande métrique parmi les métriques correspondant aux nœuds de la boucle, moins 1, permettra d'éviter la boucles transitoire correspondant à ce cycle.

Propriété 4.3.2.2

Étant donné un graphe $G=(V,E)$, $D \in V$, $X \rightarrow Y \in E$ avec la métrique initiale de $X \rightarrow Y = m$ et la métrique terminale de $X \rightarrow Y = M$, tel que

$$KMS_{X \rightarrow Y}(D) = \{km_1, km_2, .., km_n\},$$

$$A, B \in V \text{ dans un même cycle } C \in Cycle_{X \rightarrow Y}(D),$$

$$Path_{AB}(X \rightarrow Y = m) \in rSPT_D(X \rightarrow Y = m) \text{ et } Path_{BA}(X \rightarrow Y = m) \in rSPT_D(X \rightarrow Y = m),$$

1. *Si $KM_{X \rightarrow Y}(A) < KM_{X \rightarrow Y}(B)$, $KM_{X \rightarrow Y}(A)$ n'est pas décisive pour C.*
2. *Si $KM_{X \rightarrow Y}(B) < KM_{X \rightarrow Y}(A)$, $KM_{X \rightarrow Y}(B)$ n'est pas décisive pour C.*

Cas 1: $KM_{X \rightarrow Y}(A) < KM_{X \rightarrow Y}(B)$

1. Étant donné $KM_{X \rightarrow Y}(A) < KM_{X \rightarrow Y}(B)$, $rSPT_D(X \rightarrow Y = KM_{X \rightarrow Y}(A))$ est un sous ensemble de $rSPT_D(X \rightarrow Y = KM_{X \rightarrow Y}(B))$ (le point 2, Théorème 4.2.2.2).
2. Étant donné la definition 4.3.1.1, (1) et $Path_{BA}(X \rightarrow Y = m) \in rSPT_D(X \rightarrow Y = m)$, on a que $Path_{BA}(X \rightarrow Y = m) \in rSPT_D(X \rightarrow Y = KM_{X \rightarrow Y}(A))$,

3. Étant donné (2), C= ($Path_{AB}$(X→Y=m) U $Path_{BA}$(X→Y=KM $_{X→Y}$ (B)) Є la fusion de rSPT $_D$(X→Y=m) et rSPT $_D$(X→Y= KM $_{X→Y}$ (A), KM $_{X→Y}$ (A)
4. Étant donné la definition 4.4.2.1 et (3), KM $_{X→Y}$ (A) n'est pas décisive pour C

Cas 1: KM $_{X→Y}$ (A) < KM $_{X→Y}$ (B): similaire au cas 1

Cette propriété est donc prouvée !

Théorème 4.3.2.1

Étant donné un graphe G=(V,E), D Є V, X→Y Є avec la métrique initiale de X→Y=m et la métrique terminale de X→Y = M, tel que

$$KMS_{X→Y}(D)=\{km_1, km_2, ..,km_n\},$$

il y a une seule métrique clef décisive pour un cycle C Є $Cycle_{X→Y}$(D).

Preuve

Supposons que A, B Є C .

1. Si $Path_{BA}$(X→Y=m) Є rSPT$_D$ (X→ Y =m), selon Propriété 4.3.2.1, KM $_{X→Y}$ (B) n'est pas décisive.
2. Si $Path_{AB}$(X→Y=m) Є rSPT$_D$ (X→ Y =m) selon Propriété 4.3.2.1, KM $_{X→Y}$ (A) n'est pas décisive.
3. Si $Path_{AB}$(X→Y=m) Є rSPT$_D$ (X→ Y =m) et $Path_{BA}$(X→Y=m) Є rSPT$_D$ (X→ Y =m)
   a) KM $_{X→Y}$ (A) < KM $_{X→Y}$ (B), selon Propriété 4.3.2.1, KM $_{X→Y}$ (A) n'est pas décisive pour C
   b) KM $_{X→Y}$ (B) < KM $_{X→Y}$ (A), selon Propriété 4.3.2.1, KM $_{X→Y}$ (B) n'est pas décisive pour C

Comme la partie 4.2, nous définissons la séquence des métriques clefs décisives.

Définition 4.3.1.2

Étant donné un graphe G=(V,E), D Є V, X→Y Є E avec la métrique initiale de X→Y=m et la métrique terminale de X→Y = M,, $Cycle_{X→Y}$ (D), l'ensemble des métriques clefs

décisives constitue la séquence des métriques clefs décisives

$$DMS_{X\rightarrow Y}(D)=\{dkm_1, dkm_2, ..,dkm_n\}.$$

Ensuite, nous définissons la séquence des métriques clefs décisives de reconfiguration.

Définition 4.3.1.3

Étant donné un graphe G=(V,E), D ∈ V, X→Y ∈ E avec la métrique initiale de X→Y=m et la métrique terminale de X→Y = M, $Cycle_{X\rightarrow Y}(D)$,

$$DMS_{X\rightarrow Y}(D)=\{dm_0, dm_1, ..,dm_{n+1}\},$$

la séquence de métriques clefs décisives de reconfiguration est

$$DRMS_{X\rightarrow Y}(D)=\{ drm_0 , drm_1 , drm_2, ..,drm_{n+1}\}$$

$$où\ drm_0 =dm_0 =m,\ et\ drm_{n+1}=dm_{n+1}=M\ et\ drm_i=dm_i -1\ pour\ i=1..n.$$

Et nous prouvons que l'application de cette séquence sur le lien est sans boucle.

Théorème 4.3.2.2

Étant donné un graphe G=(V,E), D ∈ V, X→Y ∈ E avec la métrique initiale de X→Y=m et la métrique terminale de X→Y = M, $Cycle_{X\rightarrow Y}(D)$. En appliquant progressivement $DRMS_{X\rightarrow Y}(D)$ sur le lien X→ Y, il n'y a pas de boucle transitoire dans le réseau représenté par G.

Preuve

Pour tout cycle C ∈ $Cycle_{X\rightarrow Y}(D)$ et la métrique clef décisive dm_i

1. Étant donné dm_i, la métrique clef décisive d'une boucle C ∈ $Cycle_{X\rightarrow Y}(D)$, $dm_i < drm_{i+1}$,
   C est dans la fusion de $rSPT_D(X\rightarrow Y= m)$ et $rSPT_D(X\rightarrow Y= drm_{i+1})$, car dm_i est la plus petite des métriques de $KMS_{X\rightarrow Y}(D)$ qui peut créer une boucle transitoire (par définition).
2. Selon le théorème 4.2.2.1, il n'y a pas de boucle dans la transition de la métrique

du lien X→ Y de $drm_i = dm_i - 1$ et dm_i, donc C n'est pas présent dans la fusion de $rSPT_D(X \rightarrow Y = drm_i)$ et $rSPT_D(X \rightarrow Y = dm_i)$

3. Étant donné (1) et (2), C n'est pas dans la fusion de $rSPT_D(X \rightarrow Y = drm_i)$ et $rSPT_D(X \rightarrow Y = dm_{i+1})$

Le théorème est donc prouvé !!

4.3.3 L'algorithme LE&DKM

À partir de deux propriétés au-dessus, nous pouvons construire un algorithme alternatif pour trouver une séquence de reconfiguration de métriques qui permet d'éviter les boucles transitoires. Cet algorithme contient quatre étapes principales :

i) Combiner les deux rSPTs pour calculer les métriques clefs (KMS) des nœuds dans chaque boucle.
ii) Énumérer les boucles dans la fusion de deux rSPTs en se basant sur l'algorithme de parcours en largeur d'abord (DFS: Depth First Search) [18].
iii) Chercher une DKM pour chaque boucle en se basant sur les deux propriétés (DKMS).
iv) Construire une DRMS à partir de DKMS

Nous allons montrer que l'application de la séquence produites par LE&DKM est sans boucle.

Théorème 4.3.3.1

Étant donné un graphe G=(V,E), D ∈ V, X→Y ∈ avec la métrique initiale de X→Y=m et la métrique terminale de X→Y = M, l'algorithme LE&DKM produit une séquence des métriques de reconfiguration qui peut éviter toutes les boucles transitoires lors du changement de la métrique du lien X→Y de m à M.

Preuve

Selon la détection des boucles du chapitre 2, l'étape i) permet de détecter toutes les boucles possibles dans le réseau lors du changement de la métrique du lien X→Y de m à M.

L'idée de l'algorithme DFS [18] d'énumérer des boucles dans un graphe est que nous balayons tous les nœuds en fonction des arêtes du graphe et les stockons dans une liste LIFO (Last In First Out, ou STACK). Chaque entrée dans le STACK est assignée par une étiquette qui se compose d'un nœud et celui précédent sur son chemin. Un cycle sera détecté si nous balayons un nœud qui est déjà dans le STACK. Le cycle sera reconstruit en suivant les étiquettes des nœuds.

Les cycles seront stockés dans une liste pour être utilisé dans la phase suivante. DFS assure que nous pouvons trouver tous les cycles dans un graphe (l'étape ii)).
Le Théorème 4.3.2.1 assure que chaque boucle aura une seule métrique clef décisive.
Le théorème 4.2.2.2 assure qu'il n'y a pas de boucles transitoires lors de l'application de la séquence des métriques clefs décisives moins 1.
Ce théorème est donc prouvé !!
Dans la partie suivant, nous présenterons l'implémentation de LE&DKM en pseudo code.

4.3.4 L'implémentation de LE&DKM en pseudo code

Dans cette partie, nous présentons la procédure pour calculer RMS selon LE&DKM pour une destination ***dest***, un lien ***L*** avec la métrique initiale et terminale est ***m*** et ***M***.

```
Cycles énumérerCycles(Graph G, Link L avec métrique initiale, terminale m et M )
{
/* combiner deux rSPT */
Initial_rSPT=calculer_rSPT(dest, L, m);
Target_rSPT= calculer_rSPT (dest, L,M);
merge_RPST=combiner_rSPT(initialRSPT,targetRSPT);
/* énumère les boucles, l'algorithme DFS */

Set Cycles ={}; /* l'ensemble de cycles du graphe */
STACK LabelStack={}; /* le stack des étiquette */
/* insère un noeud quelconque dans le stack */
push(LabelStack, (AnyNode,Φ ));

Tant que LabelStack is not empty
faire
        {
        /* prendre un nœud du stack */
         varNode=pop(LabelStack);
         Pour chaque P is neighbor of varNode faire
                {
                si P is in LabelStack alors
                        {
                        /* constitute une boucle en suivant les étiquettes */
                        Set C= prendreBoucle(P, LabelStack);
                        /* ajoute cette boucle dans la liste */
                        ajouter (Cycles,C) ;
                        }
                Sinon   { /* insère ce noeud dans le stack */
                        Push(LabelStack,(P,varNode));
                        }
                }
        }
   retourne Cycles ;
}
```

Algorithme 4-5 : L'énumération des boucles selon l'algorithme DFS

```
/* construit RMS */

MetricSequence construireRMS (Graph G, Cycles, Link L Link L avec métrique
initiale, terminale m et M)
{
RMS={m};
BranchImpacted=Suivre une branche dont la racine est X dans l'arbre rSPTY (X→ Y =m);

Pour chaque  C in Cycles faire
        {
        Set Nodes={all nodes of C};
        Set Cousins={}; /* stocke les cousins */
        Set tempFamily={}; /* stocke une famille temporaire des noeuds */
        Tant que Nodes is not empty faire
                {
                /* examine un noeud quelconque */
                tempFamily={AnyNode of Nodes};
                /* regroupe une famille des noeuds étant donné un noeud quelconque:
                 C'est un groupe des noeuds dans un même chemin de la racine ver une feuille
                */
                regrouper_une_famille(tempFamily, C, BranchImpacted);

                /* choisit le père de la famille: le noeud le plus proche de la racine */

                fatherNode=pendrePère(tempFamily, BranchImpacted);

                /* insère ce noeud dans l'ensemble des cousins */

                ajouter(Cousins, (fatherNode, fatherNode.KeyMetric));

                /* supprime la branche visitée */

                supprimer(Nodes, tempFamily);
                }
        /* cherche la métrique clef décisive pour un cycle */
         Metric dkm=chercherKMMaximum(Cousins) ;
         ajouter(RMS, dkm-1) ;

        }

retourne RMS ;
}
```

Algorithme 4-6: La constitution de RMS pour une destination selon LE&DKM

4.3.5 Analyse de complexité

Supposons un graphe G=(V,E), N est le nombre des nœuds de G, N=|V|. Dans l'algorithme DFS, nous balayons toutes les arêtes du graphe de la fusion de deux rSPTs et reconstruisons un cycle lorsqu'il est détecté. La longueur maximale d'un cycle est N. La complexité de l'algorithme DFS dans ce cas est donc O(|E| N).

À chaque fois nous trouvons la métrique clef décisive d'un cycle, nous balayons toutes les métriques clefs correspondantes aux nœuds du cycle et vérifions leurs relations dans l'arbre $rSPT_Y$ ($X \rightarrow Y = m$). La hauteur maximale de cet arbre est N. La complexité de cette phase est donc $O(N^2)$. Celle de l'algorithme LE&DKM pour calculer la RMS pour une destination est $O(|E| N^3)$ et pour toutes les destinations est $O(|E| N^4)$.

Néanmoins, si le nombre d'arêtes du graphe est limité par O(N), la complexité de l'algorithme DFS est de $O(N^2)$. Celle de la recherche de la métrique clef décisive pour chaque cycle est de O(N). Le temps de calcul de DRMS pour une destination est de $O(N^2)$ et pour toutes destinations est de $O(N^3)$.

En comparant à l'algorithme LIF (pour une destination $O(N \log^2 N)$ et pour toutes destinations est de $O(N^2 \log^2 N)$, LIF est plus rapide que LE&DKM.

Pourtant, notons que le temps de calcul de LIF dépend de la longueur des KMS mais celui de LE&DKM dépend de la forme des boucles (micro boucle ou macro boucle).

Si les boucles sont de longueur-2 (micro-boucle), la complexité de DFS est de O(N) et la recherche de la métrique clef décisive pour un cycle est de logN. Le temps de calcul de DRMS pour une destination est de O(N logN) et pour toutes destinations est de $O(N^2$ logN), ce qui est plus rapide que LIF.

Nous continuerons à évaluer la performance de deux algorithmes dans le chapitre 6.

4.4 Conclusion

Dans ce chapitre, nous pouvons nous baser sur des reconfigurations progressives des poids, afin de résoudre le problème sans devoir modifier la spécification du protocole. Dans ce cas, pour reconfigurer la métrique d'un lien, il faut calculer une séquence de

métriques à appliquer sur le lien. L'application de ces métriques intermédiaires assure que l'ordre des mises à jour des FIBs soit respecter et donc évite les boucles.

LIF est utilisé pour optimiser la séquence des métriques de manière globale, tandis que LE&DKM distingue deux types de boucles : micro et macro boucles, et se base sur une résolution des boucles cycle par cycle.

LE&DKM ne produit pas une séquence optimisée et en théorie, LIF est plus rapide que LE&DKM.

Cependant, nous verrons au chapitre suivant qu'en pratique (sur des graphes représentatifs de réseaux OSPF réels), le temps de calcul de LE&DKM est moins long que pour LIF.

5 Implémentation en XORP

5.1 Introduction

XORP (eXtensible Open Router Platform) est la seule plateforme industrielle d' « open source » de routeur. XORP implémente les protocoles de routage d'IPv4 ainsi qu'IPv6. L'architecture de XORP facilite l'implémentation de nouvelles fonctionnalités, de nouveaux protocoles et il fournit une plateforme unifiée pour les configurer. XORP est largement utilisé par des instituts éducationnels et des communautés de développement [4]. Par rapport à Quagga[2], le code source de XORP est beaucoup plus documenté.

Dans ce chapitre, nous étudierons la faisabilité de deux algorithmes de reconfiguration des métrique (LIF et LE&DKM) en tant que leur fonctionnalité en XORP. Pour la première partie, nous présenterons l'architecture de XORP, l'architecture du processus OSPF_LOOPFREE. Ensuite, nous présenterons notre implémentation et des différentes variables des algorithmes en XORP. Nous testerons également l'effet des commandes de XORP relatives à OSPF qui pouvaient créer des boucles transitoires (e.g., « set interface cost » et « disable interface »).

5.2 Architecture de XORP

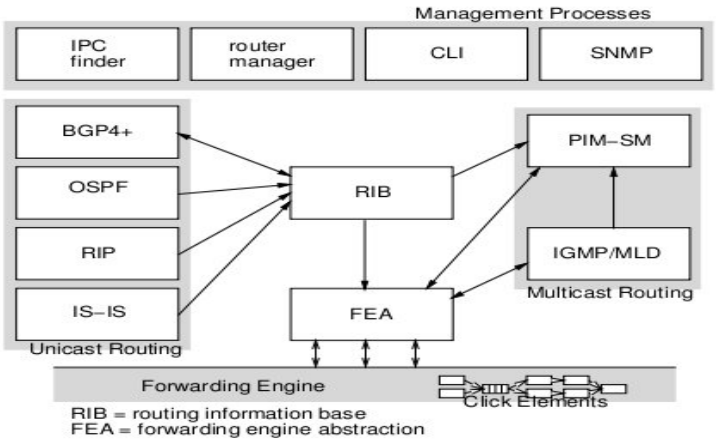

Figure 5-1: L'architecture de XORP [4]

XORP implémente le plan de contrôle de différents protocoles de routage dont OSPF. Le principe de XORP est de contrôler les protocoles de routage afin de construire la FIB de son hôte. Le plan de données est quant à lui géré entièrement par l'hôte de l'instance de XORP. En d'autres mots, XORP construit les tables de routage et les tables de forwarding, mais le forwarding est assuré par le système d'exploitation qui supporte l'instance de XORP (XORP ne traite aucun paquet en transit par le routeur)[4].

XORP est conçu selon le modèle multi processus (illustré à la Figure 5.1). Parmi ces processus, *Router Manager* (voir la partie 5.4.1 en plus détail) est responsable de gérer les informations de tous les composants du routeur (e.g., la configuration). Il surveille également le statut de tous les processus concernés [4].

CLI (Command Line Interface) permet à l'utilisateur d'accéder au routeur (via Router Manager) pour mettre à jour ou afficher les informations du routeur.
OSPF, IS-IS, RIP, etc. sont des processus indépendants qui jouent le rôle des protocoles de routage correspondants.
RIB (Routing Information Base) stocke les tables de routage du routeur. Il communique avec d'autres processus de routage (OSPF, BGP, RIP, etc) pour récupérer les routes et avec FEA (Forwarding Engine Abstraction) pour constituer les tables de forwarding utilisées pour le transfert de paquets sur le réseau.
IPC finder (Inter-Process Communication finder) est nécessaire pour la méthode de communication utilisée entre tous les composants de XORP. Chaque composant XORP enregistre sur *l'IPC finder*. *IPC finder* supporte la communication entre XRLs : il connaît la location des XRLs. Un processus XORP n'a pas donc besoin de connaître explicitement la location de tous les autres processus, ni la façon de communication avec eux. Le *Router Manager* incorpore un *finder*, *l'IPC finder* n'est nécessaire qu'au cas où le *Router Manager* n'est pas utilisé comme durant des tests [4].

5.3 Architecture de l'implémentation en XORP

Nous avons implémentés les deux algorithmes évoqués plus haut dans le module OSPF_LOOPFREE. Ce dernier est responsable de calculer et de stocker la RMS correspondant à chaque interface (lien) du routeur. Le graphe qui représente la topologie du routeur est construit à partir de la LSDB qui est obtenue grâce à un appel vers OSPF (en plus détail, voir la partie 5.4).

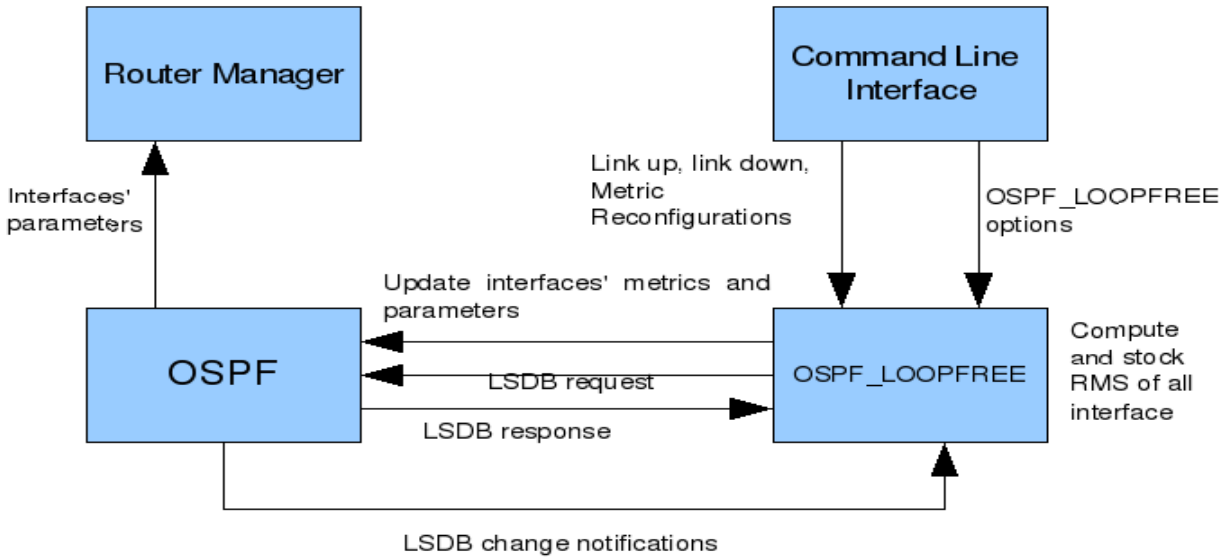

Figure 5-2: L'architecture de l'implémentation des algorithmes en XORP

Il y a quatre options à configurer OSPF_LOOPFREE via CLI: *graceful, lif, fast* et *interval.*

Si *graceful* est égale à *«true»*, les commandes «set interface cost» (la reconfiguration de métrique d'une interface) et «disable interface» (désactiver l'interface) sont traitées par le module OSPF_LOOPFREE. Lorsque l'administrateur ne mentionne aucune de ces options, les commandes sont traitées par les mécanismes standards d'OSPF.

Si *lif* est égale à *«true»,* LIF est utilisé pour optimiser RMS, sinon LE&DKM est utilisé.

Si *fast* est égale à *«true»,* le RMS est calculé avec une seule destination, à savoir le routeur qui est de l'autre côté du lien examiné. Sinon, toutes les destinations sont impliquées.

interval est l'intervalle (en ms) entre deux applications de métriques de la RMS, lors du changement de configuration.

Les valeurs initiales des paramètres d'OSPF_LOOPFREE (*graceful, lif, fast* et *interval*) peuvent être stockées dans un fichier « *template »* du processus. Elles peuvent être automatiquement lues depuis le *fichier de configuration* d'OSPF_LOOPFREE (voir la partie 5.4 pour le détail).

Les RMS sont recalculées quand l'option *graceful* est mise à vrai (true) et quand un changement de LSDB est détecté. Le module OSPF_LOOPFREE est automatiquement notifié de ce changement via un XRL (voir la partie 5.4 pour le détail).

5.4 Développement d'OSPF_LOOPFREE

XORP étant écrit en C++, nous avons choisi d'implémenter OSPF_LOOPFREE dans ce langage[1].
Les LSDB des zones d'OSPF sont indépendantes (chapitre 2), nous implémentons donc l'OSPF_LOOPFREE pour une seule zone.
Il y a deux types de lien (Point à Point, Multi-Access) sur les réseaux OSPF (chapitre 2). Le traitement des algorithmes pour les réseaux OSPF de type Multi-Access est plus compliqué, nous n'implémentons donc que les liens Point à Point entre les routeurs OSPF dans ce mémoire.
Cette section se compose de quatre grandes parties: (i) le développement du processus OSPF_LOOPFREE en XORP, (ii) la lecture de la LSDB pour construire le graphe, (iii) le calcul des RMS pour chaque interface du routeur et, (iv) l'application des séquences de métriques.

5.4.1 Le développement du processus OSPF_LOOPFREE en XORP [5][14]

1 Nous aurions pu écrire notre module dans un autre langage.

a) Le fonctionnement du processus *Router Manager (rtrmgr)* [5]

RouterManager est le seul processus qui est explicitement démarré durant le démarrage d'un routeur XORP. Un *IPC finder* est inclut dans ce processus, donc, nous n'avons pas besoin d'autres *IPC finders.*

La séquence des actions du processus est :

1. Le *rtrmgr* lit tous les fichiers « *template »* dans le répertoire « *template »* du routeur (xorp/etc/template). Normalement, il y a un fichier « *template »* pour chaque processus. Chaque fichier « *template »* décrit les fonctionnalités fournies par le processus correspondant en terme que tous les paramètres de configuration sont mis. Il décrit également les dépendances nécessaires avant le démarrage des processus. Après avoir lu les fichiers « *template »*, *rtrmgr* connaît tous les paramètres de configuration supportables sur ce routeur et les stocke dans un *arbre « template »*. L'arbre « *template »* est ensuite vérifié pour trouver les erreurs. Le *rtrmgr* quitte s'il y a une erreur durant la vérification des paramètres.
2. Ensuite, le *rtrmgr* lit le contenu du répertoire des XRLs (xorp/xrl) pour découvrir tous les XRLs supportés par les processus du routeur. Ces XRLs sont utilisés pour vérifier ceux correspondants dans l'arbre « *template »*. S'il y a une erreur durant la vérification des XRLs, le *rtrmgr* quitte.
3. Ensuite, le *rtrmgr* lit le fichier de *configuration* du routeur (ce qui est fourni dans la commande lors du démarrage du routeur). Toutes les options de configuration doivent correspondre aux fonctionnalités configurables décrites dans le fichier « *template »*. Cette configuration est stockée dans un *arbre de configuration* dont les nœuds sont marqués comme *« not existing »* : on ne communique pas encore avec les processus pour implémenter les fonctionnalités correspondantes à cette partie de configuration.
4. Ensuite, le *rtrmgr* traverse l'arbre de configuration pour découvrir la liste des processus nécessaires à démarrer afin de fournir des fonctionnalités demandées. Normalement, nous n'avons pas besoin de tous les logiciels disponibles pour une configuration spécifique.

5. Le *rtrmgr* traverse l'arbre *« template »* encore une fois pour découvrir l'ordre des processus afin de satisfaire toutes les dépendances.
6. Le *rtrmgr* fait fonctionner le premier processus dans la liste des processus à démarrer.
7. S'il n'y pas d'erreur, le *rtrmgr* traverse l'arbre de configuration afin de construire la liste des commandes nécessaires à configurer pour le processus fonctionné.
8. S'il n'y pas d'erreur dans l'étape précédente, le prochain processus sera démarré et configuré jusqu'à la fin de la liste des processus.
9. Le routeur maintenant est disponible pour les opérations interactives et des connexions de CLI qui est lancé par la commande *xorpsh*

b) Le fichier « *template »*

Le *Router Manager* lit le répertoire des fichiers *« template »* (xorp/etc/template) pour découvrir les options que le routeur supporte. Nous illustrons dans la figure 5.3 le fichier *« template »* du processus OSPF_LOOPFREE.

```
1: protocols {

2: ospf_loopfree {
3:          targetname: txt = "ospf_loopfree";

4:          fast:                bool=true ;
5:          lif:                 bool=true ;
6:          interval: u32 =20;
7:          graceful:bool=false;
              }
           }

8: protocols {
9: ospf_loopfree {

10:                %help: short "Avoid OSPF transient loops";
11:                %modinfo: provides ospf_loopfree;
12:                %modinfo: path "ospf_loopfree/xorp_ospf_loopfree";
13:                %modinfo: default_targetname "ospf_loopfree";
14:                targetname {
15:                   %user-hidden: "XRL target name";
16:                   %help:          short "XRL target name";
17:                   %set:;
                   }

18:                fast  {

                              %help: short "True: Tail-end destination; False: All Destinations";
                              %set:xrl "ospf_loopfree/ospf_loopfree/0.1/set_fast?fast:bool=$(@)";


                   }
21:                lif {
                              %help: short "True:Infocom's Algorithm. False: Adhoc Algorithm";
                              %set:xrl "ospf_loopfree/ospf_loopfree/0.1/set_lif?lif:bool=$(@)";
                   }
24:                interval {
                               %help: short "Miliseconds between increment insertions upon application";
                              %set:xrl "ospf_loopfree/ospf_loopfree/0.1/set_interval?interval:u32=$(@)";
                   }

27:                graceful {

                              %help: short "On/Off graceful mode";
                              %set:xrl "ospf_loopfree/ospf_loopfree/0.1/set_graceful?graceful:bool=$(@)";
                   }
```

Figure 5-3: Le fichier « template » du processus OSPF_LOOPFREE

Ce fichier *« template »* se compose de deux parties principales pour déclarer les paramètres et les XRLs du processus.

Dans les lignes de 2 à 7, nous déclarons les paramètres, leur type de données et leur valeur initiale. Dans la ligne 12, le chemin dans le répertoire racine vers le fichier d'exécution du processus est xorp/ospf_loopfree/xorp_ospf_loopfree. Dans la ligne 13, le nom du processus utilisé dans les appels via XRLs est *ospf_loopfree*. Dans les lignes de 18 à 27, nous déclarons la description et le XRL pour mette à jour la valeur de chaque paramètre du processus.

c) Le développement de l'interface des XRLs d'OSPF_LOOPFREE [14]

Nous déclarons dans cette partie l'interface des XRLs d'OSPF_LOOPFREE pour être enregistrée sur l'IPC finder du Router Manager. Ce fichier (ospf_loopfree.xif) réside dans le répertoire xorp/xrl/interfaces. Il contient la déclaration des procédures pour effectuer les fonctionnalités d'OSPF_LOOPFREE (voir l'annexe 3 pour le détail).

Ce fichier doit être compilé en quatre fichiers : ospf_loopfree_xif.cc, ospf_loopfree_xif.hh, ospf_loopfree_xif.lo et ospf_loopfree_xif.o en utilisant l'outil **clnt-gen** de XORP dans le répertoire xorp/xrl/scripts.

Sur Linux, nous pouvons taper la commande : **python clnt-gen ospf_loopfree.xif**
Le fichier ospf_loopfree_xif.cc contient le format des requêtes XRLs correspondantes aux procédures d'OSPF_LOOPFREE qu'un client utilise pour demander d'effectuer une tâche à ce processus. Par exemple, lorsque le processus OSPF veut notifier les changements de LSDB à OSPF_LOOPFREE, il utilise l'une de ces requêtes.
Ces quatre fichiers doivent être présents dans le répertoire xorp/xrl/interface.

Ensuite, nous créons un autre fichier ospf_loopfree.tgt dans le répertoire xorp/xrl/targets.

Ce fichier contient la déclaration des interfaces que nous souhaitons implémenter dans OSPF_LOOPFREE (y compris ospf_loopfree.xif, voir l'annexe 4 pour le détail).

Nous utilisons l'outil **tgt-gen** dans le répertoire xorp/xrl/scripts pour produire cinq fichiers : ospf_loopfree.xrls, ospf_loopfree_base.cc, ospf_loopfree_base.hh, ospf_loopfree_base.lo, ospf_loopfree_base.o.
Sur Linux, nous pouvons taper la commande : **python tgt-gen ospf_loopfree.tgt**

Le fichier ospf_loopfree.xrls contient la location des procédures d'OSPF_LOOPFREE. Par exemple, la location de la procédure pour mette à jour la valeur du paramètre *fast* d'OSPF_LOOPFREE est *finder://ospf_loopfree/ospf_loopfree/0.1/set_fast?fast:bool*.

Le fichier ospf_loopfree_base.hh contient la déclaration de la classe XrlOspfLoopfreeTargetBase dont les méthodes virtuelles correspondent aux procédures d'OSPF_LOOPFREE. Le code source de ces méthodes sera rempli lors du développement de la boucle principale du processus (la partie suivante).
Ces cinq fichiers doivent être présents dans le répertoire xorp/xrl/targets.

d) La boucle principale (Main loop) d'OSPF_LOOPFREE

Chaque processus possède une boucle qui s'appelle « event_loop » et un routeur virtuel pour traiter les requêtes XRL entrantes et sortantes du processus.
Cette boucle (event_loop) se situe dans la fonction « main » du fichier d'exécution du processus (xorp_ospf_loopfree).
La classe qui implémente les procédures d'OSPF_LOOPFREE doit aussi implémenter la classe standard du routeur virtuel (voir l'annexe 5,6 pour le détail).
En ce moment, le code source de cette classe doit être rempli pour traiter les fonctionnalités d'OSPF_LOOPFREE.

5.4.2 La notification des changements de LSDB depuis OSPF

Nous utilisons un appel procédural à distance via XRL dans le processus OSPF pour notifier des changements de LSDB au processus OSPF_LOOPFREE.

```
1:XrlOspfLoopfreeV0p1Client _xrl_ospf_loopfree_client;

/*-------------------------------------------------------------------*/
2:XrlIO<IPv4>::notifyLSDBchange(const IPv4&     router_id)
      {
      bool success=false;
5:    success=_xrl_ospf_loopfree_client.send_notify_lsdb_change
      ("ospf_loopfree",router_id,
      callback(this,&XrlIO::send_notify_lsdb_change_cb ));
      return success;

      }
/*-------------------------------------------------------------------------------------------*/

11: void XrlIO<A>::send_notify_lsdb_change_cb(const XrlError& xrl_error)
      {
      if (xrl_error.error_code()==OKAY) return;

      };
```

Figure 5-4: La notification des changements de LSDB à OSPF_LOOPFREE

Nous déclarons, d'abord, un client d'OSPF_LOOPFREE pour envoyer des requêtes XRLs vers ce processus (figure 5.4, la ligne 1). La procédure « notifyLSDBchange » est responsable d'envoyer l'appel via XRL avec l'ID du routeur vers la procédure send_notify_lsdb_change d'OSPF_LOOPFREE (la ligne 5 de la figure 5.4). Le résultat de cet appel va être retourné à la fonction « callback » d'OSPF : send_notify_lsdb_change_cb (la ligne 11 de la figure 5.4).

Dans notre implémentation, la procédure de notification a été utilisée dans le module de calcul de SPT d'OSPF parce que nous prenons en compte seulement les LSAs qui causent des changements de la LSDB.

5.4.3 La lecture de LSDB

OSPF fournit via une interface XRL des procédures qui permettent d'accéder à la LSDB du routeur local comme: Get_Area () ou Get_LSA(). On utilise Get_Area() pour récupérer les Area_IDs de toutes les zones dans le réseau OSPF. Get_LSA() retourne un LSA dont l'ID est fourni comme un paramètre. OSPF_LOOPFREE peut appeler ces XRLs et recevoir des résultats dans ses fonctions «callback» correspondantes.

Deux types de LSAs utilisés pour obtenir l'état complet d'une zone sont: Router-LSA et Network-LSA (voir l'annexe 1 pour le détail). À partir des Router-LSAs, nous pouvons trouver les ID des routeurs, les adresses et métriques de leurs interfaces. Network-LSA contient les informations de voisins des routeurs. Avec ces deux types de LSA nous pouvons construire un graphe représentant la topologie d'une zone.

Dans OSPF_LOOPFREE, la lecture de LSDB depuis OSPF est lancée lorsque l'option *graceful* est mise à «true», lorsqu'il y a une notification de changement de topologie du module OSPF ainsi que chaque fois l'application d'une séquence des métriques soit finie.

5.4.4 Le calcul des RMS

Nous pouvons calculer les RMS pour toutes les interfaces du routeur. Pour chaque interface, nous distinguons deux types de liens: point à point et point à multipoints (par exemple Ethernet). Notre implémentation supporte uniquement les liens point à point. Des études réalisées dans [1] ont montré que dans plus de 90% des cas de maintenance de liens, le RMS correspondant à la destination qui est à l'autre côté du lien est suffisant pour éviter les boucles transitoires pour toutes les destinations affectées, et ce dans les 5 réseaux d'ISP étudiés. Cela permet de ne pas stocker les informations sur les rSPT de toutes les destinations influencées par le changement. Seules celles correspondant au rSPT dont la racine est le routeur de l'autre côté du lien sont nécessaires (Chapitre 2). Les RMS de toutes les interfaces du routeur sont stockées en mémoire.

5.4.5 L'application des métriques

Deux commandes souvent utilisées lors d'opérations de maintenances sont «set interface cost» pour modifier la métrique de l'interface et «disable interface» pour désactiver l'interface avant de l'éteindre. La métrique de certains liens du réseau peut être modifiée pour répondre aux objectifs d'ingénierie de trafic du réseau (Traffic Engineering). L'idée de base étant de modifier les métriques afin de favoriser certains liens plutôt que d'autres. De la sorte, il est possible de diminuer ou d'augmenter le trafic de certains liens. L'utilisateur voudrait également éteindre un routeur pour le réparer ou mettre à niveau ses logiciels. Dans ce cas, toutes ses interfaces doivent être d'abord désactivées.

Ces commandes sont réalisées en modifiant le fichier ospfv2.tp dans CLI qui contient les modèles de celles de configuration d'OSPF IPv4. Cette modification leur permet d'être traitées par OSPF_LOOPFREE au lieu d'OSPF.

OSPF_LOOPFREE reçoit les informations d'une interface via un XRL; le RMS est disponible en mémoire. Il envoie ensuite des XRLs vers OSPF pour appliquer, étape par étape, chaque métrique de la séquence. Pendant ce temps, s'il y a une notification de changement de topologie d'autres routeurs, OSPF_LOOPFREE passe en mode *«fast convergence»* : la métrique terminale sera toute de suite appliquée.

5.5 Des difficultés

Dans cette section, nous présentons les difficultés et certain cas particuliers spécifiques à l'implémentation des algorithmes dans XORP.

Le premier cas apparaît pour l'option *fast* avec «toutes les destinations», nous ne prenons en compte que des nœuds influencés (Chapitre 2).

La vérification des boucles entre deux métriques clefs demande de calculer les deux rSPTs lors de l'optimisation des RMS (Chapitre 3). Si les RMS sont longues, cette étape est lourde pour le routeur. Nous avons utilisé une cache pour stocker les rSPTs. Mais une autre façon pour réduire le temps de vérification des boucles transitoires est que nous

utilisons un algorithme incrémental de l'arbre des plus courts chemins [8]. L'idée de cet algorithme est de mettre à jour les plus court chemins vers une destination pour des nœuds influencés. Bien qu'implémentée dans notre solution, cette technique n'est pas évaluée dans le présent document.

5.6 Conclusion

XORP est un routeur d'open-source qui est largement utilisé. Il a une architecture ouverte, ce qui facilite la modification par des développeurs. La communication entre des processus dans XORP est effectuée en utilisant XRLs. C'est une forme d'appels de procédures interprocessus. OSPF_LOOPFREE est construit comme un processus séparé visant à éviter les boucles transitoires causées par les deux commandes de reconfiguration de métrique comme « set interface cost » ou bien « disable interface ».
Notre implémentation montre qu'il est possible de mettre en œuvre des mécanismes d'évitement des boucles dans le protocole OSPF de XORP.

6 Évaluation des performances

6.1 Introduction

Dans ce chapitre, nous analyserons l'efficacité des algorithmes de reconfiguration de métrique décrits dans le chapitre 3 (LIF et LE&DKM), en nous basant sur des mesures de la performance de notre implémentation en XORP. Quatre topologies seront testées. Deux topologies sont des réseaux réels et les deux autres ont été générées avec IGEN [15]. Les temps de calcul de l'algorithme LIF et de l'algorithme LE&DKM seront évalués, ainsi que la longueur des séquence de reconfiguration qu'ils produisent.

6.2 Méthodologie de test

En général, les opérations de routeur comme la reconfiguration des métriques et la désactivation des interfaces se passent régulièrement durant la maintenance d'un routeur. Même si pendant cette phase, nous avons assez de temps de préparation, l'algorithme d'évitement des boucles transitoires pour ces opérations doit être raisonnablement rapide, afin d'être pratique. Il ne doit pas provoquer un nombre trop important de re-calcul de table de routage afin de préserver les performances du routeur. Les séquences de reconfiguration doivent donc être raisonnablement courtes. De plus, il doit être efficace pour pouvoir être appliqué à de grands réseaux.

Dans le cas des algorithmes de reconfiguration des métriques présentés dans le chapitre 3, pour vérifier le respect de ces exigences, il est nécessaire d'évaluer le temps de calcul des RMS et le nombre d'applications de métriques intermédiaires pour chaque lien. En fait, chaque fois qu'une nouvelle métrique est appliquée sur une interface, un LSA sera diffusé dans le réseau pour rafraîchir toutes les LSDBs des routeurs (Chapitre 1), ce qui cause le re-calcul des RMS. C'est pour cela que la longueur des RMS devrait être aussi courte que possible.

Comme analysé précédemment, cette dernière dépend des anneaux dans une topologie. Dans ces tests, nous allons donc examiner la distribution du temps de calcul et celle de la longueur des RMS sur des réseaux avec des nombres et des tailles différents d'anneaux (Chapitre 2).

Dans ce chapitre, nous évaluerons, d'abord, la performance de LIF et ensuite, nous la comparerons avec LE&DKM.

6.3 Cas de test

La première topologie, Abilene, est l'une des épines dorsales pour la recherche et l'éducation aux États-Unis [13]. Abilene est illustrée à la figure 6.1. Il se compose de 11 routeurs, 14 liens et 4 anneaux. La deuxième topologie étudiée est celle d'un réseau commercial national en Europe. Elle contient 53 nœuds et 196 liens. Nous l'appelons National_ISP.

Figure 6-1: Le réseau Abilene [13]

Deux autres topologies sont générées en utilisant IGEN [15]: ISP1 et ISP2. Chacune a 100 routeurs. IGEN permet de générer différents types de topologies avec un nombre quelconque de routeurs répartis dans les cinq continents. Les métriques des liens sont par

défaut les distances géographiques entre les routeurs. ISP1 et ISP2 contiennent 10 et 20 anneaux de coeurs, respectivement. Ces topologies sont illustrées dans les figures 6.2 et 6.3.

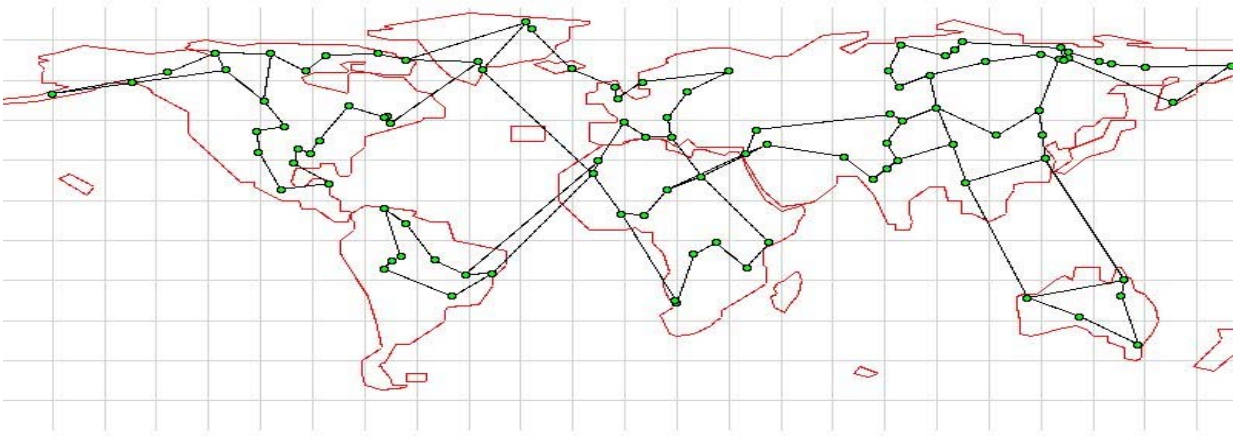

Figure 6-2: L'ISP 1

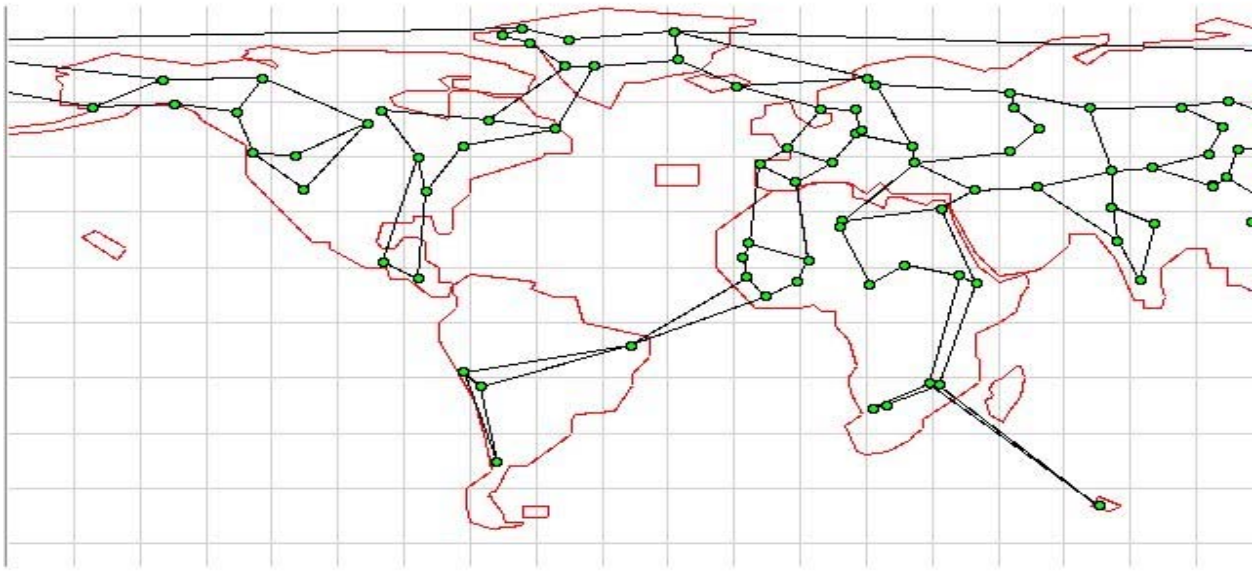

Figure 6-3: L'ISP 2

6.4 Évaluation des résultats

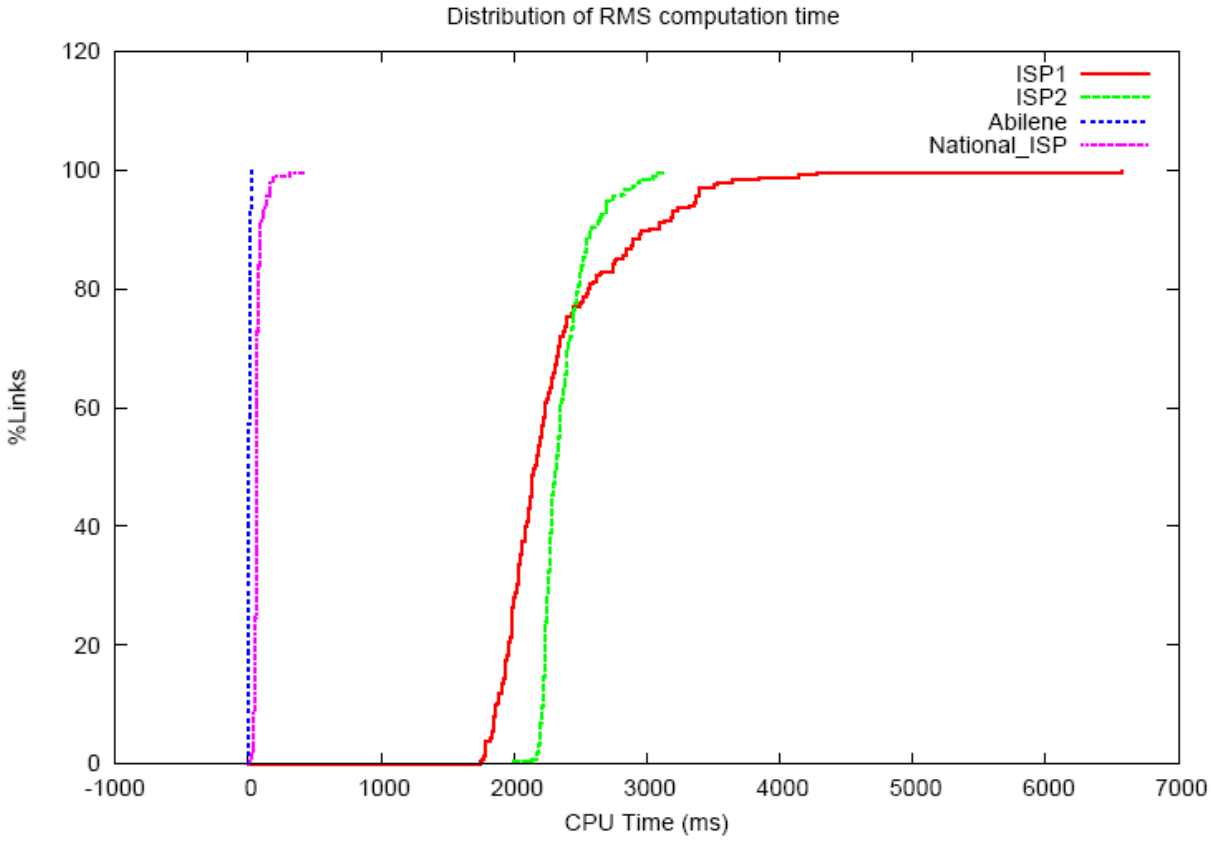

Figure 6-4: La distribution du temps de calcul des ORMS sur quatre topologies selon l'algorithme LIF

Dans cette partie, nous évaluons la performance de LIF des quatre topologies, sur deux aspects: le temps de calcul (figure 6.4) et le nombre des métriques de reconfiguration (figure 6.5). Pour chaque lien, nous considérons la modification de sa métrique à Max_Metric simulant ainsi une mise hors service du lien.

Dans la figure 6.4, l'axe « %Link » représente le pourcentage des liens et l'axe « CPU Time » représente le temps de calcul des ORMS pour les liens dans une topologie.
Dans la figure 6.5, « %CDF » représente le taux cumulé de la longueur des ORMS et l'axe « RMS length » représente la longueur des ORMS pour les liens dans une topologie.
Le temps de calcul des ORMS est effectué dans une machine portable IBM de type R60 dont la vitesse est de 1,66 GHz et la capacité de RAM est de 500 MB.

Il y a 70% de liens d'Abilene qui causent des boucles transitoires. En effet, seuls 30% des liens ont une longueur de RMS de 1 (dans la figure 6.5). Le temps de calcul de RMS pour chaque lien est de 30 millisecondes environ (avec 90% des cas, figure 6.4). La longueur de RMS est primordialement 2 (81%), ce qui veut dire que chaque lien a besoin en moyenne de deux applications de métriques pour éviter des boucles (figure 6.6) (Max_Metric y compris).

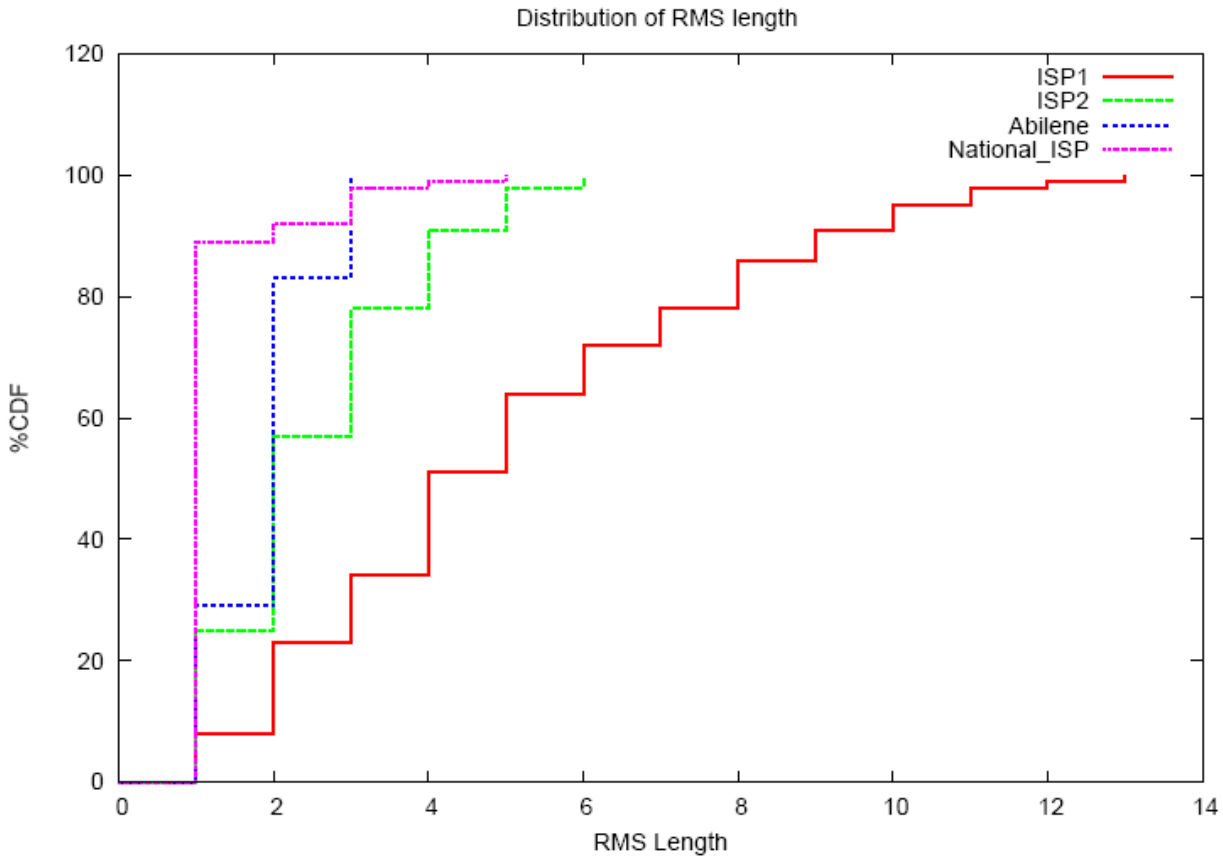

Figure 6-5: La distribution du temps de la longueur des ORMS sur quatre topologies selon l'algorithme LIF

National_ISP n'a que 31% des liens qui causent des boucles, ce qui peut être expliqué par le fait que le nombre de liens qui font partie d'anneaux est limité. Le temps de calcul de RMS pour chaque lien est 200 millisecondes (98%) qui est plus long que celui d'Abilene à cause du plus nombre de nœuds et de liens. Les liens qui ont besoin de métriques intermédiaires ont besoin de trois applications (98%) de métriques afin d'éviter des boucles transitoires possibles.

ISP1 et ISP2 sont beaucoup plus grands qu'Abilene et National_ISP. Pourtant le temps de calcul de RMS pour 90% des liens est moins de trois secondes. De plus, 90% des boucles transitoires sont évitées par moins de 10 applications de métrique intermédiaire. Notons que le nombre des anneaux de ISP2 (20) est beaucoup plus grand que celui de ISP1 (10), ce qui veut dire que la longueur moyenne d'un anneau de ISP1 est beaucoup plus petite que celle d'ISP2. Cela explique pourquoi il y a quelques RMS dont la longueur est plus grande que 10 dans ISP1 mais non dans ISP2. Ces RMS requièrent un long temps de calcul.

6.5 Comparaison de la performance des deux algorithmes

La comparaison de performance des algorithmes LIF et LE&DKM de deux topologies ISP1 et ISP2 est montrée les figures 6.6, 6.7, 6.8 et 6.9.

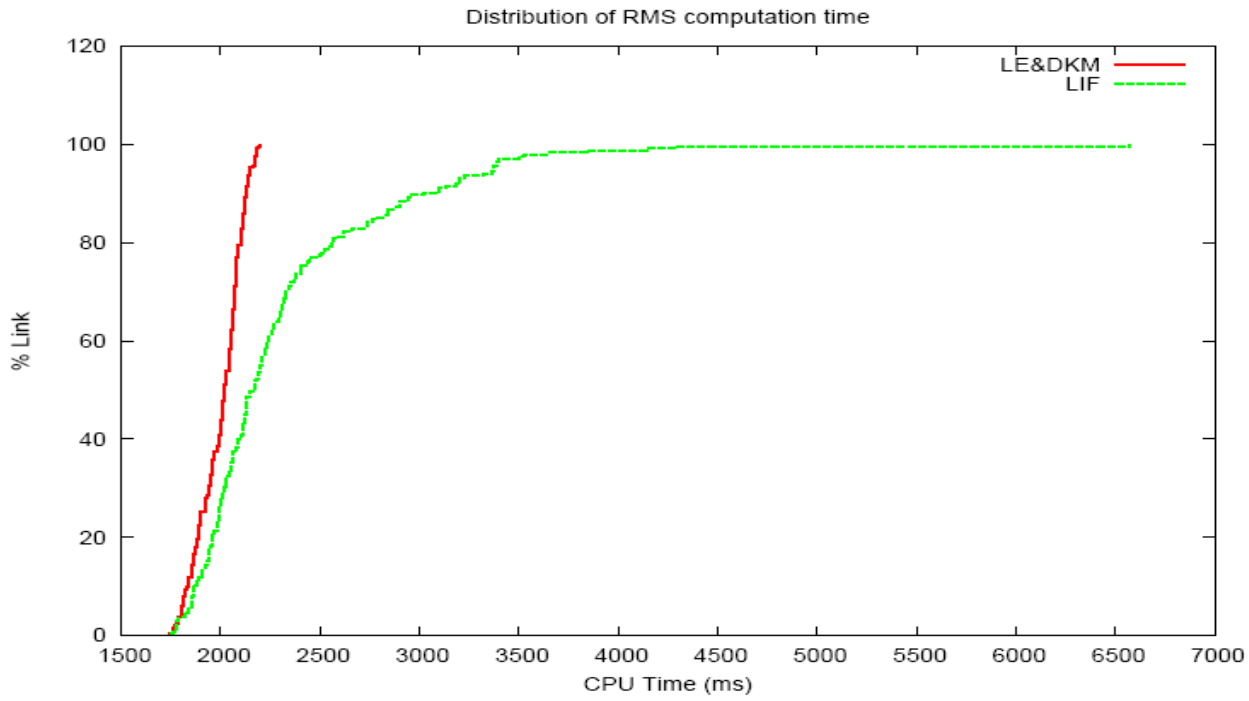

Figure 6-6: La comparaison entre LIF et LE&DKM sur le temps de calcul des ORMS (ISP1)

Dans la figure 6.6 et 6.7, l'axe « %Link » représente le pourcentage des liens et l'axe

« CPU Time » représente le temps de calcul des ORMS pour les liens d'ISP1 et d'ISP2, respectivement.

Dans la figure 6.8 et 6.9, « %CDF » représente le taux cumulé de la longueur des ORMS et l'axe « RMS length » représente la longueur des ORMS pour les liens d'ISP1 et d'ISP2, respectivement..

Le temps de calcul des ORMS est aussi effectué dans une machine portable IBM de type R60 dont la vitesse est de 1,66 GHz et la capacité de RAM est de 500 MB.

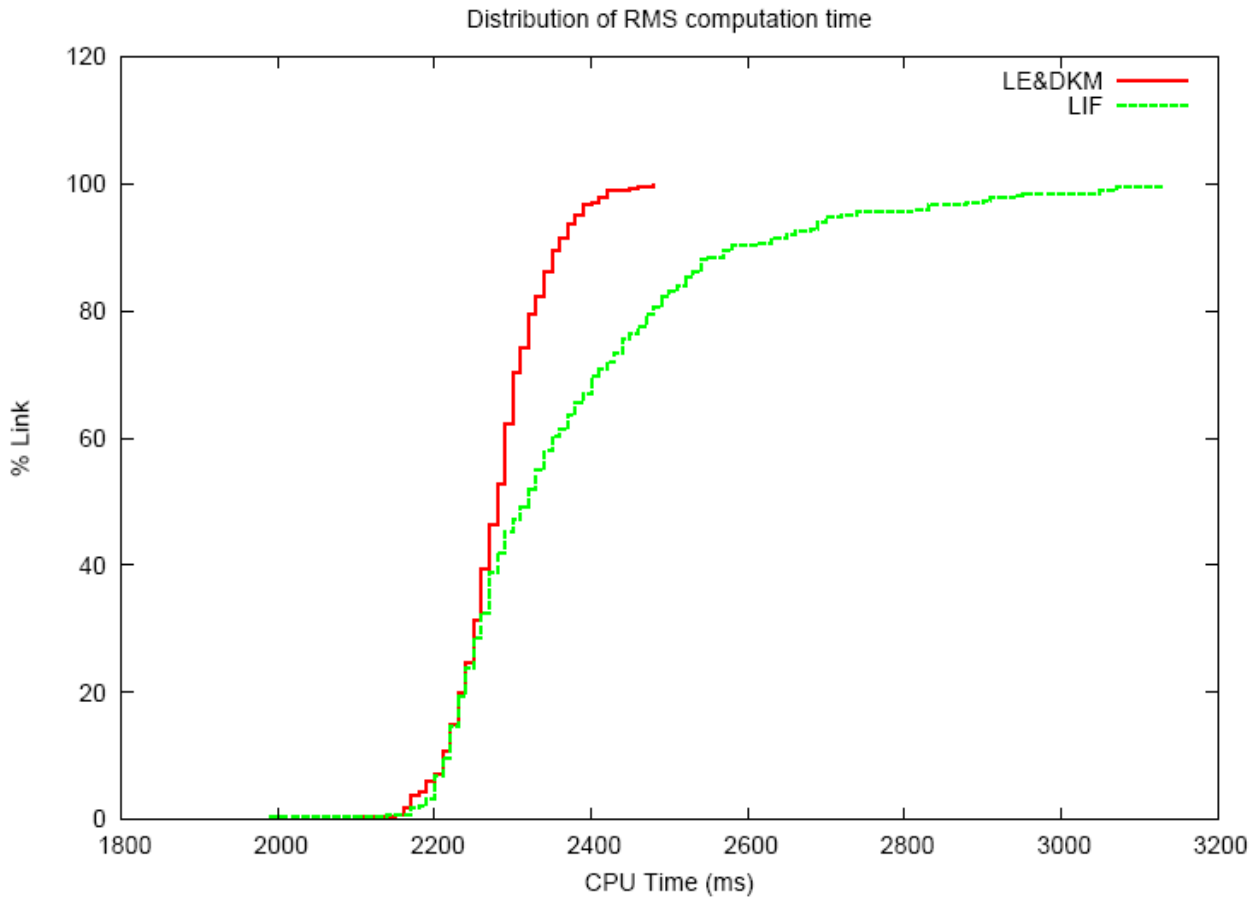

Figure 6-7: La comparaison entre LIF et LE&DKM du temps de calcul des ORMS sur ISP2

Concernant le temps de calcul, notons que dans le chapitre 4, nous avons vu que théoriquement LIF est plus rapide que LE&DKM. Néanmoins, dans les figures 6.6 et 6.7, LIF est plus lent que LE&DKM: sur ISP1, le temps de calcul de 100% des ORMS selon LE&DKM est de moins de trois secondes alors que ce taux en fonction de LIF est

de 80%. De même, sur ISP2, 80% des ORMS selon LIF prends deux secondes de calcul par rapport à 100% de LE&DKM.

Dans les topologies ISP1 et ISP2, le temps de calcul de 20% ORMS selon LIF est variable et assez long. Cela s'explique par les longues RMS à minimiser dans LIF. Il y a donc des cas particuliers pour cet algorithme. Comme analysé précédemment (la partie 6.4), ce sont des longs anneaux dans une topologie.

La distribution du temps de calcul des ORMS est par contre mise à échelle. La raison est que LE& DKM ne dépend pas de la longueur des RMS et la plupart des boucles sont de type «micro» (comme analysé dans le chapitre 3). La recherche du nombre des micros boucles dans la fusion de deux rSPTs peut nous aider à trouver d'autres solutions pour ce problème.

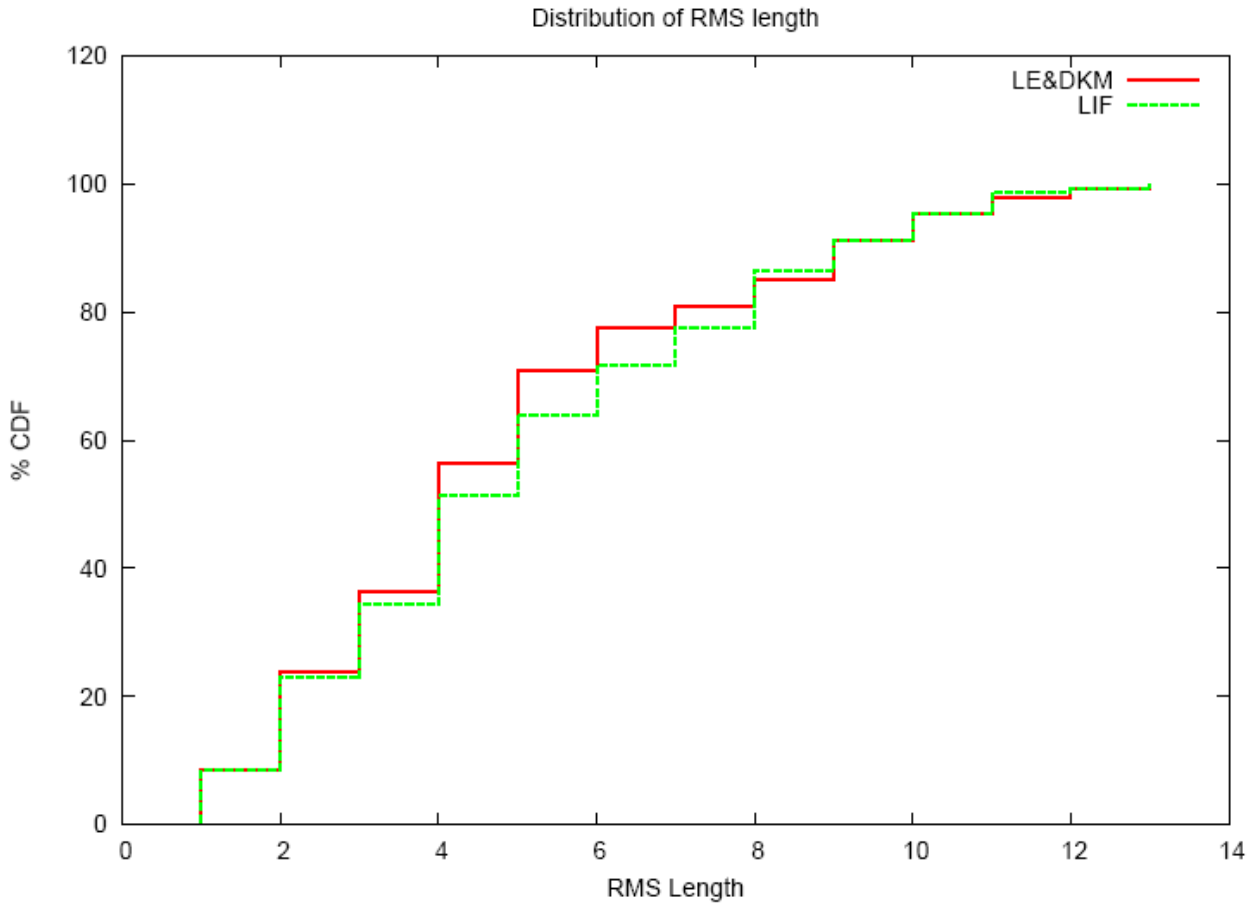

Figure 6-8: La comparaison entre LIF et LE&DKM de la longueur des ORMS sur ISP1

Dans les cas ISP1 et ISP2 (figure 6.8, 6.9), la longueur des ORMS selon LE&DKM est clairement moins optimisée que celle de LIF mais la différence entre les deux algorithmes LIF et LE&DKM est négligeable. En effet, le décalage sur ISP1 et ISP2 est respectivement de 5% et 1%.

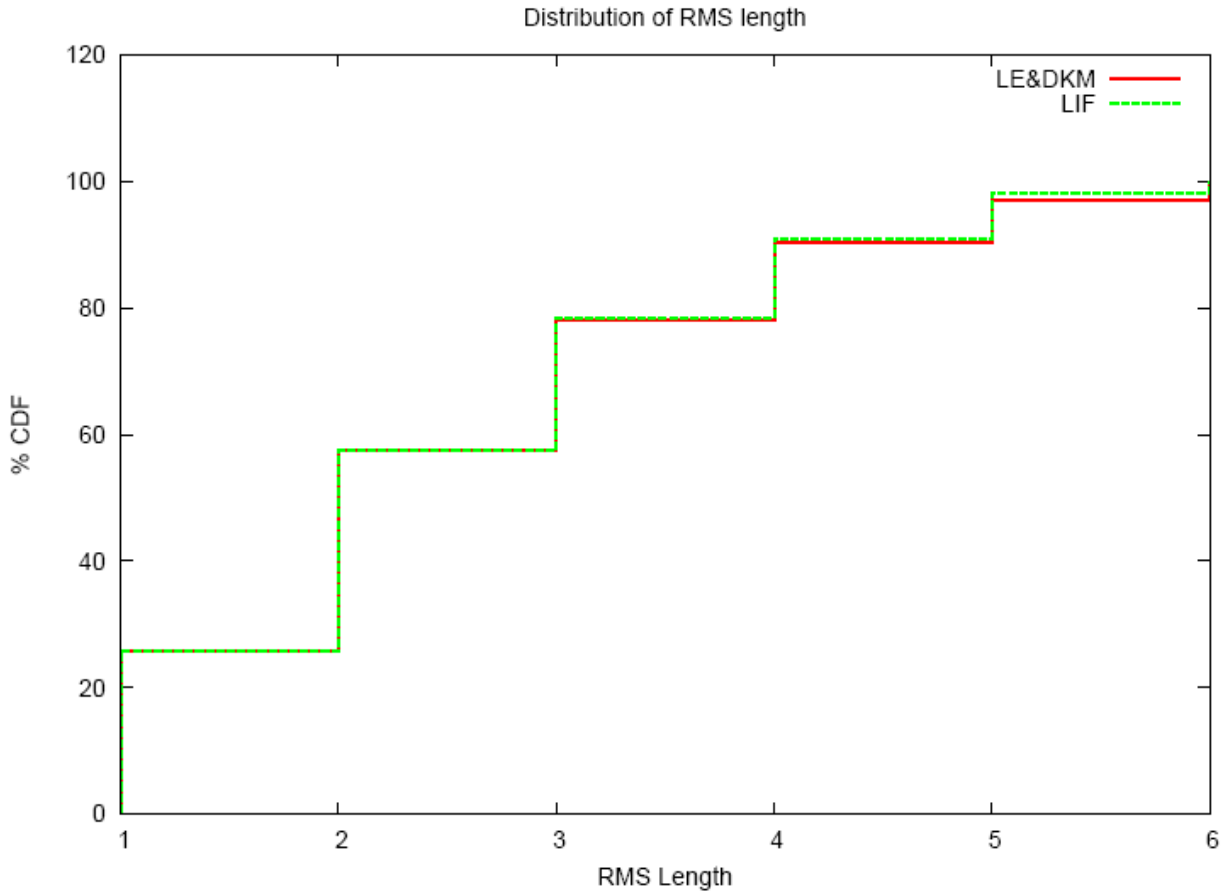

Figure 6-9: La comparaison entre LIF et LE&DKM de la longueur des ORMS sur ISP2

6.6 Conclusion

L'efficacité de LIF a été montrée dans ce chapitre avec des réseaux qui contiennent un nombre différent d'anneaux: deux réseaux réels : l'Abilene et un réseau commercial national, et deux autres réseaux générés par IGEN. Deux aspects ont été étudiés : le

temps de calcul et la longueur de la séquence de reconfiguration. Le temps de calcul des ORMS est variable mais de dizaine secondes pour les topologies de centaine routeurs. La longueur des ORMS est aussi court: moins de dix pour la plupart des cas. Nous voyons qu'en pratique, LE&DKM est plus rapide que LIF, et il ne produit pas de séquences plus longue que LIF (la différence est de 5%). Il s'avère donc judicieux d'utiliser LE&DKM comme fonctionnalité de routeur. Tandis que, pour un calcul « off-line » de RMS, où l'efficacité est moins importante que pour une fonctionnalité interne de routeur, LIF est à recommander car il produit des séquences optimales.

Conclusion

Dans ce mémoire, nous avons étudié des mécanismes d'évitement de boucles transitoires durant la convergence de OSPF, un protocole de routage à état de lien largement utilisé dans les réseaux de transit de trafic Internet.

Le protocole OSPF établit la connectivité entre les routeurs au sein d'un réseau de transit. Il se base sur l'établissement de la connaissance du graphe du réseau (les routeurs et les liens physiques entre ces routeurs), et sur l'application de l'algorithme de calcul des chemins les plus courts sur ce graphe.

Lorsqu'un opérateur effectue une maintenance sur un lien ou sur un routeur, le graphe change, et donc les routeurs doivent adapter le transfert des paquets IP aux nouveaux chemins les plus courts dans le graphe modifié. Cependant, les routeurs ne réagissent pas en même temps au changement, et peuvent être transitoirement incohérents dans le transfert des paquets. Ces incohérences peuvent mener à des boucles transitoires, qui saturent les liens sur lesquelles elles se produisent et produisent des pertes de paquets.

Nous avons étudié en détail une technique de la littérature permettant d'éviter ces boucles, qui consiste à l'augmentation progressive du poids du lien qui subit la maintenance, et qui assure que chaque étape de la progression est sans boucle.

Nous avons proposé une méthode alternative pour le calcul des séquences de poids à appliquer. Notre méthode est plus complexe mais elle s'exécute plus rapidement.

Nous avons implémenté les deux techniques dans XORP, une plateforme de routage open-source, afin, d'une part, d'en évaluer la faisabilité d'implémentation dans un routeur réel, et, d'autre part, d'en évaluer les performances.

Cette implémentation nous a permis de vérifier que la technique alternative est en effet plus rapide. Nous avons aussi constaté que les séquences produites peuvent être plus longues que celles produites par l'algorithme décrit dans la littérature. Notre travail a donc permis d'ouvrir le champ des solutions possibles pour l'utilisation de cette approche. Nous avons en effet offert une alternative légère mais non optimale, à une technique optimale, mais lourde en terme de temps de calcul.

Bibliographie

[1] Pierre François, *Improving the convergence of IP Routing Protocols*. l'Université catholique de Louvain ,Octobre,2008

[2] *Quagga Routing Suite***,** http://www.quagga.net/.

[3] Pacôme Massol, *Initiation au routage*, 3ème partie, Année universitaire 2006-2007, http://www.pmassol.net/lm/zebraospf.html

[4] XORP Team, *XORP Design Overview*, 2008, http://www.xorp.org/releases/1.5/docs/design_arch/design_arch.pdf

[5] XORP Team, *XORP Router Manager Process*, 2008, http://www.xorp.org/releases/1.5/docs/rtrmgr/rtrmgr.pdf

[6] Aman Shaikh, Rohit Dube, Anujan Varma, *Avoiding Instability during Graceful Shutdown of Multiple OSPF Routers*. http://www.research.att.com/~ashaikh/papers/ospf-ibb-ton06.pdf

[7] Wikimedia, *Dijkstra's algorithm*, 2008, http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

[8] Paolo Narváez Sycamore Networks, Chelmsford, MAKai-Yeung Siu Massachusetts Institute of Technology, CambridgeHong-Yi Tzeng Amber Networks, Santa Clara, CA, *New dynamic algorithms for shortest path tree computation*, 2000, http://infoscience.epfl.ch/record/117069

[9] H3C Technologies Co., Limited, *OSPF Introduction*, 2004-2008, http://www.h3c.com/portal/Products___Solutions/Technology/IP_Routing/OSPF/200702/201240_57_0.htm

[10] Rhys Haden , *OSPF*, 1996-2008,http://www.rhyshaden.com/ospf.htm

[11] Olivier Bonaventure, *Link failures, cours de Réseaux: configuration et gestion*.

[12] Gianluca Iannaccone, Chen-nee Chuah, Richard Mortier, Supratik Bhattacharyya, Christophe Diot, *Analysis of link failures in an IP backbone*, 2002, http://www.imconf.net/imw-2002/imw2002-papers/202.pdf

[13] Qwest, *Qwest and Internet2*, 2008, http://www.qwest.com/about/qwest/internet2/map.html

[14] XORP Team, *An Introduction to Writing a XORP Process*, 2008, http://www.xorp.org/releases/1.5/docs/xorpdev_101/xorpdev_101.pdf

[15] Bruno Quotin, Topology generation through network design heuristics.,2006,http://www.info.ucl.ac.be/~bqu/igen/

[16] Zifei Zhong , Ram Keralapura , Srihari Nelakuditi , Yinzhe Yu , Junling Wang , Chen-Nee Chuah , and Sanghwan Lee, *Avoiding Transient Loops through Interface-Specific Forwarding.*,2005,http://www.springerlink.com/content/jw8pu45deqxfwrlx/

[17] Wikimedia, Binary search algorithm.,2008,http://en.wikipedia.org/wiki/Binary_search

[18] Wikimedia, Depth-first search.,2008,http://en.wikipedia.org/wiki/Depth-first_search

[19]Wikimedia, *Routing Information Protocol*, 2008, http://en.wikipedia.org/wiki/Routing_Information_Protocol

[20] Wikimedia, *IS-IS,* 2008, http://en.wikipedia.org/wiki/IS-IS

[21] Dax Networks, *IP Forwarding Information Base (FIB)*, 2003, http://www.daxnetworks.com/Technology/TechDost/TD-102605.pdf

[22] Yoshihiro Ohba, *Issues on Loop Prevention in MPLS Networks,* 1999, http://www.comsoc.org/ci/private/1999/dec/pdf/Ohba.pdf

[23] Saikat Ray, Roch Guerin1, and Rute Sofia, *Distributed Path Computation without Transient Loops:An Intermediate Variables Approach.* http://mor306-6.seas.upenn.edu/mnlab/papers/Ray_DIV_TR_version.pdf

[24] Urs Hengartner, Sue Moon, Richard Mortier, Christophe Diot. *Detection and Analysis of*
Routing Loops in Packet Traces. http://an.kaist.ac.kr/~sbmoon/paper/intl-conf/2002-imw-routing-loop.pdf
[25] Srinivas Vutukury, Garcia-Luna-Aceves, *A Simple Approximation to Minimum-Delay Routing,* in *Proceedings of ACM SIGCOMM*, Cambridge, MA, September 1999.
http://ccrg.soe.ucsc.edu/publications/vutukury.sigcomm99.pdf
[26] J. J. Garcia-Lunes-Aceves, *Loop-Free Routing Using Diffusing Computations, IEEE/ACM Trans. Networking*, 1:130–141, February 1993.
http://www.ecse.rpi.edu/Homepages/shivkuma/teaching/sp2001/readings/garcia-luna-aceves.pdf
[27] Wikimedia, *Routing Protocol,* http://en.wikipedia.org/wiki/Routing_protocol
[28] MicrosoftTechnet*, Dynamic Routing Protocols*, http://technet.microsoft.com/en-us/library/cc758398.aspx
[29] RFC 2328, *RFC2328 - OSPF Version 2*, http://www.faqs.org/rfcs/rfc2328.html

ANNEXE

Annexe 1 : Un exemple de la LSDB dans un routeur

OSPF link state database, Area 0.0.0.0
Router-LSA:
LS age 531 Options 0x2 DC: 0 EA: 0 N/P: 0 MC: 0 E: 1 LS type 0x1 Link State ID 192.168.2.1 Advertising Router 192.168.2.1 LS sequence number 0x80000004 LS checksum 0x2a5c length 60
bit Nt false
bit V false
bit E false
bit B false
Type 2 Transit network IP address of Designated router 192.168.4.1 Routers interface address 192.168.4.1 Metric 10
Type 2 Transit network IP address of Designated router 192.168.2.1 Routers interface address 192.168.2.1 Metric 10
Type 2 Transit network IP address of Designated router 192.168.1.1 Routers interface address 192.168.1.2 Metric 50
Router-LSA:
LS age 568 Options 0x2 DC: 0 EA: 0 N/P: 0 MC: 0 E: 1 LS type 0x1 Link State ID 192.168.3.1 Advertising Router 192.168.3.1 LS sequence number 0x80000002 LS checksum 0xb0ca length 48
bit Nt false
bit V false
bit E false
bit B false
Type 2 Transit network IP address of Designated router 192.168.3.1 Routers interface address 192.168.3.1 Metric 10
Type 2 Transit network IP address of Designated router 192.168.1.1 Routers interface address 192.168.1.1 Metric 50
Network-LSA:
LS age 633 Options 0x2 DC: 0 EA: 0 N/P: 0 MC: 0 E: 1 LS type 0x2 Link State ID 192.168.1.1 Advertising Router 192.168.3.1 LS sequence number 0x80000001 LS checksum 0x9b0c length 32
Network Mask 0xffffff00
Attached Router 192.168.3.1
Attached Router 192.168.2.1
Network-LSA:
LS age 569 Options 0x2 DC: 0 EA: 0 N/P: 0 MC: 0 E: 1 LS type 0x2 Link State ID 192.168.2.1 Advertising Router 192.168.2.1 LS sequence number 0x80000001 LS checksum 0x9c0b length 32
Network Mask 0xffffff00
Attached Router 192.168.2.1
Attached Router 192.168.2.2
Router-LSA:
LS age 483 Options 0x2 DC: 0 EA: 0 N/P: 0 MC: 0 E: 1 LS type 0x1 Link State ID 192.168.2.2 Advertising Router 192.168.2.2 LS sequence number 0x80000004 LS checksum 0x6e33 length 60
bit Nt false
bit V false
bit E false
bit B false
Type 2 Transit network IP address of Designated router 192.168.6.2 Routers interface address 192.168.6.2 Metric 10
Type 2 Transit network IP address of Designated router 192.168.2.1 Routers interface address 192.168.2.2 Metric 10
Type 2 Transit network IP address of Designated router 192.168.3.1 Routers interface address 192.168.3.2 Metric 10
Network-LSA:
LS age 568 Options 0x2 DC: 0 EA: 0 N/P: 0 MC: 0 E: 1 LS type 0x2 Link State ID 192.168.3.1 Advertising Router 192.168.3.1 LS sequence number 0x80000001 LS checksum 0x9311 length 32
Network Mask 0xffffff00
Attached Router 192.168.3.1
Attached Router 192.168.2.2
Network-LSA:
LS age 531 Options 0x2 DC: 0 EA: 0 N/P: 0 MC: 0 E: 1 LS type 0x2 Link State ID 192.168.4.1 Advertising Router 192.168.2.1 LS sequence number 0x80000001 LS checksum 0x9f04 length 32
Network Mask 0xffffff00
Attached Router 192.168.2.1
Attached Router 192.168.5.1

Router-LSA:
LS age 482 Options 0x2 DC: 0 EA: 0 N/P: 0 MC: 0 E: 1 LS type 0x1 Link State ID 192.168.5.1 Advertising Router 192.168.5.1 LS sequence number 0x80000003 LS checksum 0xa6ec length 48
bit Nt false
bit V false
bit E false
bit B false
Type 2 Transit network IP address of Designated router 192.168.5.1 Routers interface address 192.168.5.1 Metric 10
Type 2 Transit network IP address of Designated router 192.168.4.1 Routers interface address 192.168.4.2 Metric 10
Network-LSA:
LS age 482 Options 0x2 DC: 0 EA: 0 N/P: 0 MC: 0 E: 1 LS type 0x2 Link State ID 192.168.5.1 Advertising Router 192.168.5.1 LS sequence number 0x80000001 LS checksum 0xa7f3 length 32
Network Mask 0xffffff00
Attached Router 192.168.5.1
Attached Router 192.168.6.1
Router-LSA:
LS age 478 Options 0x2 DC: 0 EA: 0 N/P: 0 MC: 0 E: 1 LS type 0x1 Link State ID 192.168.6.1 Advertising Router 192.168.6.1 LS sequence number 0x80000002 LS checksum 0xfc90 length 48
bit Nt false
bit V false
bit E false
bit B false
Type 2 Transit network IP address of Designated router 192.168.5.1 Routers interface address 192.168.5.2 Metric 10
Type 2 Transit network IP address of Designated router 192.168.6.2 Routers interface address 192.168.6.1 Metric 10
Network-LSA:
LS age 483 Options 0x2 DC: 0 EA: 0 N/P: 0 MC: 0 E: 1 LS type 0x2 Link State ID 192.168.6.2 Advertising Router 192.168.2.2 LS sequence number 0x80000001 LS checksum 0x900d length 32
Network Mask 0xffffff00
Attached Router 192.168.2.2
Attached Router 192.168.6.1

Router-LSA:
LS age 531 Options 0x2 DC: 0 EA: 0 N/P: 0 MC: 0 E: 1 LS type 0x1 Link State ID 192.168.2.1 Advertising Router 192.168.2.1 LS sequence number 0x80000004 LS checksum 0x2a5c length 60
bit Nt false
bit V false
bit E false
bit B false
Type 2 Transit network IP address of Designated router 192.168.4.1 Routers interface address 192.168.4.1 Metric 10
Type 2 Transit network IP address of Designated router 192.168.2.1 Routers interface address 192.168.2.1 Metric 10
Type 2 Transit network IP address of Designated router 192.168.1.1 Routers interface address 192.168.1.2 Metric 50

Explications : **Le routeur dont l'ID est 192.168.2.1 a trois interfaces 192.168.4.1, 192.168.2.1, 192.168.1.1 avec la métrique 10,10 et 50, respectivement.**

Network-LSA:
LS age 483 Options 0x2 DC: 0 EA: 0 N/P: 0 MC: 0 E: 1 LS type 0x2 Link State ID 192.168.6.2 Advertising Router 192.168.2.2 LS sequence number 0x80000001 LS checksum 0x900d length 32
Network Mask 0xffffff00
Attached Router 192.168.2.2
Attached Router 192.168.6.1

Explications : L'interface 192.168.6.2 du routeur dont l'ID est 192.168.2.2 se connecte au routeur dont l'ID est 192.168.6.1

Annexe 2 : Le fichier de configuration du processus OSPF_LOOPFREE

```
/*XORP Configuration File, v1.0*/
/*Router A */
protocols {
    ospf4 {
        router-id: 192.168.3.1
        rfc1583-compatibility: false
        ip-router-alert: false
        area 0.0.0.0 {
            area-type: "normal"
            interface eth1 {
                link-type: "broadcast"
                vif eth1 {
                    address 192.168.3.1 {
                        priority: 128
                        hello-interval: 10
                        router-dead-interval: 40
                        interface-cost: 10
                        retransmit-interval: 5
                        transit-delay: 1
                        passive: false
                        disable: false
                    }
                }
            }
            interface eth0 {
                link-type: "broadcast"
                vif eth0 {
                    address 192.168.1.1 {
                        priority: 128
                        hello-interval: 10
                        router-dead-interval: 40
                        interface-cost: 50
                        retransmit-interval: 5
                        transit-delay: 1
                        passive: false
                        disable: false
                    }
                }
            }
        }
    }
}
```

```
interfaces {
    restore-original-config-on-shutdown: false
    interface eth0 {
        disable: false
        discard: false
        description: "LAN 1 interface"
        vif eth0 {
            disable: false
            address 192.168.1.1 {
                prefix-length: 24
                broadcast: 192.168.1.255
                disable: false
            }
        }
    }
    interface eth1 {
        disable: false
        discard: false
        description: "Link Interface"
        vif eth1 {
            disable: false
            address 192.168.3.1 {
                prefix-length: 24
                broadcast: 192.168.3.255
                disable: false
            }
        }
    }
}
```

Annexe 3 : L'interface des XRLs d'OSPF_LOOPFREE (ospf_loopfree.xif)

```
/* $XORP: xorp/xrl/interfaces/ospf_loopfree.xif $ */

/*
 * OSPF_LOOPFREE XRL interface.
 */


interface ospf_loopfree/0.1 {

        /**
         * Enable/disable/start/stop OSPF Loopfree.
         *
         * @param enable if true, then enable OSPF LOOPFREE, otherwise
         * disable it.
         */
         get_interface_rms ? router_id:ipv4 & addr:ipv4
         notify_LSDB_change ? router_id:ipv4
         set_peer_state ? router_id:ipv4 & addr:ipv4  & ifname:txt  & vifname:txt & area:ipv4 &
enable:bool
         set_fast ? fast:bool
         set_lif ? lif : bool
         set_interval ? interval: u32
         set_graceful ? graceful:bool
         set_kms ? router_id:ipv4 \
             & addr: ipv4 \
             & fast: bool \
             & lif: bool \
             & interval: u32
          set_interface_cost ?router_id:ipv4 & addr:ipv4 & ifname:txt & vifname:txt & area:ipv4
& cost:u32


}
```

Annexe 4 : Le fichier « target » d'OSPF_LOOPFREE

```
/* $XORP: xorp/xrl/targets/ospf_loopfree.tgt $ */

#include "common.xif"
#include "finder_event_observer.xif"
#include "policy_backend.xif"
#include "ospf_loopfree.xif"

target ospf_loopfree implements common/0.1,                          \
                        finder_event_observer/0.1,           \
                        policy_backend/0.1,                  \
                        ospf_loopfree/0.1
```

Annexe 5 : La boucle principale d'OSPF_LOOPFREE (le fichier xorp_ospf_loopfree)

```
static void
ospf_loopfree_main(const string& finder_hostname, uint16_t finder_port)
{
    EventLoop eventloop;

    //
    XrlOspfLoopFreeNode xrl_ospf_loopfree_node(
            eventloop,
            "ospf_loopfree",
            finder_hostname,
            finder_port,
            "finder",
            "fea",
            "rib");
    wait_until_xrl_router_is_ready(eventloop,
                                   xrl_ospf_loopfree_node.xrl_router());

    // startup
    xrl_ospf_loopfree_node.startup();

    //
    // Main loop
    //
    while (!xrl_ospf_loopfree_node.is_done()) {
            eventloop.run();
    }
}
```

Annexe 6 : Un extrait de la déclaration de la classe XrlOspfLoopfreeNode pour traiter les fonctionnalités d'OSPF_LOOPFREE

```
class XrlOspfLoopfreeNode :
                                    public XrlStdRouter,
                                   public XrlOspfLoopfreeTargetBase{
public:
   XrlOspfLoopfreeNode(EventLoop&         eventloop,
                                const string&       class_name,
                                const string&       finder_hostname,
                                uint16_t finder_port,
                                const string&       finder_target,
                                const string&       fea_target,
                                const string&       rib_target);
   ~XrlOspfLoopfreeNode();

   /**
    * Startup the node operation.
    */
   bool    startup();

   /**
    * Shutdown the node operation
    */
   bool    shutdown();

   /**
    * Get a reference to the XrlRouter instance.
    *
    * @return a reference to the XrlRouter (@ref XrlRouter) instance.
    */
   XrlRouter&      xrl_router() { return *this; }

     // XRL target methods

    XrlCmdError ospf_loopfree_0_1_notify_lsdb_change(const IPv4&router_id);

/*.....  .....  .....  .....*/

}
```