TBXCAST

Un protocole de routage multicast explicite

Rapport de conception

Cyril BOULEAU
Hamze FARROUKH
Loïc LE HENAFF
Mickaël LECUYER
Jozef LEGENY
Benoît LUCET
Emmanuel THIERRY

Encadrants : Miklós MOLNÁR, Bernard COUSIN

Sommaire

1 Introduction

Le projet TBXcast a pour objectif la réalisation d'un protocole de routage multicast explicite. Ont été déterminés dès le départ les deux invariants suivants : le protocole se basera sur un protocole existant - Xcast - et son développement se fera sous le système d'exploitation NetBSD4. La phase de conception a été pour nous une période de raffinement de notre connaissance du protocole Xcast ainsi que l'aboutissement de la spécification des fonctionnalités propres à notre protocole TBXcast. En effet, en parallèle à la localisation précise des modifications à apporter à Xcast, nous avons établi - ou repris de l'année passée - des algorithmes et structures de données qui constituent le « cœur » de TBXcast, notre protocole de routage arborescent. Nous avons ainsi établi un canevas avec les objectifs précis pour chaque version du développement, chacune terminée par une phase de test-validation.

Après un rappel de notre projet, de son contexte et de son état d'avancement actuel, nous allons présenter les structures de données retenues pour l'implémentation de l'arbre et de la topologie du réseau. Les algorithmes de création, de parcours et de segmentation de cet arbre seront également présentés. Dans une seconde partie, nous nous intéresserons en détail au routage d'un paquet, avec les questionnements que cela a impliqué. Nous présenterons une solution et détaillerons le format des entêtes retenus.

La phase de conception nous a donc permis de clarifier avec rigueur les modifications à apporter à Xcast ainsi que les nouvelles fonctionnalités à implémenter dans TBXcast. Nous nous dirigeons donc vers la phase de réalisation avec un large bagage de connaissances et une vision plutôt claire du travail qui nous attend.

2 Rappels sur le projet

Cette partie a pour but de replacer le contexte de notre projet, ainsi que de faire l'état de son avancement actuel.

Le projet TBXcast a pour objectif de concevoir un protocole de routage multicast explicite arborescent. Lorsque dans un groupe d'utilisateurs une source émet vers différents destinataires, les paquets d'informations sont routés grâce à une représentation bien particulière de la topologie du réseau : un arbre de routage.

N'étant pas aisé d'implémenter un protocole de routage depuis le début, nous nous basons sur un protocole multicast explicite existant : le protocole Xcast. Ce protocole ne se sert pas d'un arbre pour effectuer le routage, et c'est sur ce point que nos ajouts seront significatifs.

Nous allons brièvement rappeler la structure de Xcast et de ses différents modules, dont nous nous devons d'avoir une excellente compréhension pour envisager y apporter nos modifications. Nous rappellerons ensuite les principales fonctionnalités de TBXcast.

2.1 Xcast

Comme détaillé dans les deux précédents rapports, Xcast est un protocole de routage multicast explicite plat : l'entête d'un paquet Xcast contient la liste des adresses IP des destinataires.

Malgré le fait qu'il comporte plusieurs avantages par rapport au multicast traditionnel, tel un allégement des tables de routage, une diminution du trafic sur le réseau, une bonne utilisation d'unicast et une gestion simple des groupes multicast, il présente aussi des inconvénients comme un temps de traitement potentiellement important des paquets par les routeurs. Il soulève également des questions liées à l'anonymat des destinataires et à la fragmentation des paquets IPv6.

Pour continuer, rappelons les différents modules de Xcast.

2.1.1 LibXcast

Grace à cette API de Xcast, les applications de la machine source vont pouvoir accéder aux fonctions de Xcast présentes dans le noyau du système d'exploitation.

Elle fournit des méthodes pour la gestion des groupes multicast (création, suppression, ajout des membres), des sockets et l'envoi des messages depuis une machine source Xcast.

2.1.2 Ping6x

C'est une application Ping simple qui utilise Xcast avec la version 6 d'IP.

Elle permet d'envoyer un paquet Xcast à une destination afin de savoir si elle est accessible.

Ping6x n'utilise ni l'API LibXcast, ni le code du module de Xcast. Le programme construit lui-même les paquets Xcast et les gère. Ping6x est toutefois lié au noyau de NetBSD pour la réception de paquets ICMPv6.

2.1.3 Driver[1] Xcast

C'est le code source du protocole qui est présent dans le noyau du système NetBSD et qui interagit avec ce dernier au travers d'une interface.

Ce module s'articule autour de la fonction principale *xcast6_branch* qui gère la réception des paquets Xcast et leur traitement. Elle peut soit les transmettre vers les couches applicatives supérieures

[1] Le terme « driver » fait référence au module de cœur de Xcast, qui est implémenté dans les routeurs. Voir rapport de spécifications fonctionnelles, page 3.

(*xcast6_launch*) ou bien les transmettre aux routeurs suivants en traitant chaque destinataire de la liste (*xcast6_forward* ou *xcast6_x2u_forward*).

2.2 TBXcast

Notre protocole TBXcast va retrouver une architecture similaire à celle de Xcast, c'est-à-dire qu'il sera constitué d'une API, d'un code de cœur ou « driver » destiné aux routeurs intermédiaires, et d'une application test simple : ping6tbx.

2.2.1 LibTBXcast

C'est la nouvelle API qui jouera le rôle d'interface entre les applications utilisateurs au niveau de la source et le module TBXcast présent dans le noyau.

Comme celle de Xcast, elle fournira des méthodes pour la gestion des groupes multicast, des sockets et l'envoi des messages.

Les fonctionnalités propres à TBXcast comme la gestion de la topologie, le calcul et le codage de l'arbre seront implémentées dans l'API LibTBXcast. Ces fonctionnalités sont très largement couvertes dans la partie 3 de ce rapport.

2.2.2 Application de test pour TBXCast

Nous avons envisagé une application de test pour notre protocole TBXcast, et sa description figure dans l'annexe 7.3.

2.2.3 Driver TBXcast

C'est le module TBXcast écrit en C qui sera incorporé au noyau du système NetBSD et qui va se baser sur le code du driver Xcast.

Les changements notables sont au niveau du traitement de l'arbre de routage par les routeurs intermédiaires, sujet couvert dans la partie 4 de ce rapport.

2.3 Développement en versions

Comme nous l'avions étudié dans le rapport de planification, l'aboutissement à un nouveau protocole de routage fonctionnel avec toutes les contraintes à prendre en compte ne peut pas se faire du jour au lendemain mais pas à pas.

Il faut donc développer cela en plusieurs versions fonctionnelles, dont chacune vient apporter de nouveaux changements par rapport à la précédente en testant au fur et à mesure.

Selon la planification initiale, nous avons prévu de faire huit versions au total. Nous nous contenterons cette année de développer TBXcast jusqu'à sa troisième version (voir l'annexe 1 pour le canevas détaillé des versions).

2.3.1 Remarque

Comme nous l'avions précisé dans le précédent rapport, notre objectif pour cette année est d'arriver à réaliser la version 3 de notre protocole.

Malgré tout, nous avons abordé la phase de conception sur la globalité du projet afin de toujours garder en tête les fonctionnalités complètes de TBXcast. Ainsi, nous pouvons assurer une cohérence et une meilleure modularité des premières versions vis-à-vis des dernières prévues.

La suite de ce rapport ne mentionne pas explicitement à quelle version fait référence telle ou telle fonctionnalité, mais le canevas fourni en annexe 1 permet de garder en tête une idée claire de l'avancement planifié.

3 L'arbre de routage

Le projet TBXcast se distingue des autres protocoles de routage multicast grâce à un arbre de routage. Ce dernier permet de définir le chemin emprunté par un paquet dans le réseau. Ainsi les routeurs n'ont plus besoin de rechercher le prochain routeur pour acheminer le paquet. Il en résulte un allégement de la charge des routeurs[2]. Nous allons voir dans cette partie comment coder et construire l'arbre de routage à partir d'une topologie existante.

3.1 Récupération de la topologie

Pour pouvoir construire l'arbre de routage, la source a besoin de connaître la topologie du réseau, c'est-à-dire la manière d'atteindre les destinataires.

Le réseau peut être représenté comme un graphe dont les sommets sont les routeurs et les arêtes les liens entre ces routeurs.

Pour que la source ait connaissance de ce graphe, nous allons, dans un premier temps, donner manuellement la topologie à la source dans un fichier stocké à la source. Un algorithme implémenté à la source permettra de récupérer la topologie contenue dans ce fichier pour construire une représentation matricielle du réseau. Cet algorithme sera implémenté de façon polymorphe afin de garder la possibilité de s'adapter aux versions futures du protocole.

La matrice construite est la matrice des liens du graphe : c'est une matrice de taille *n* x *n* où *n* est le nombre de routeurs dans le réseau.

La case [i, j] de cette matrice peut prendre 3 valeurs différentes :

- **-1** s'il n'existe pas de lien entre le routeur i et le routeur j
- **0** si i = j
- **1** s'il y a un lien entre i et j

Si par exemple le réseau se présente sous la forme suivante :

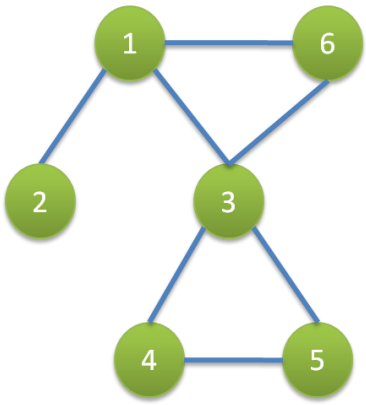


Figure 1 : Exemple de topologie

[2] Pour plus de détails, consulter le rapport de spécifications fonctionnelles

La matrice des liens associée sera alors :

X \ Y	1	2	3	4	5	6
1	0	1	1	-1	-1	1
2		0	-1	-1	-1	-1
3			0	1	1	1
4				0	1	-1
5					0	-1
6						0

Figure 2 : Matrice des liens associée à la Figure 1

A partir de la version 6 de notre protocole, la source devra être capable de récupérer la topologie par elle-même. On utilisera un protocole déjà existant pour remplir la matrice des liens : le protocole OSPF et on modifiera alors l'algorithme de construction de cette matrice pour qu'il utilise OSPF.

Le principe d'OSPF est que chaque routeur envoie des messages à ses voisins pour connaître tous les liens qu'il a avec eux. Ainsi, chaque routeur peut remplir une table contenant les liens qui existent entre lui et les routeurs voisins. Ensuite, en transmettant cette table aux autres routeurs, chaque routeur peut remplir la table des liens et connaître tous les liens existant entre tous les routeurs du réseau.

Le logiciel Quagga[3] permet de récupérer cette table à la source, ce qui nous permettra de remplir la matrice des liens.

Quagga nous affiche la table sous la forme suivante :

Link ID	ADV Router	Age	Seq#	Checksum
10.1.1.1	10.200.1.2	731	0x80000004	0x00C009

Figure 3 : Affichage donné par Quagga

Ici, cette ligne nous indique que la machine 10.1.1.1 est liée à la machine 10.200.1.2. On pourra donc rajouter un **1** dans la case correspondante de la matrice. Il faudra encore étudier plus profondément Quagga pour voir les possibilités qu'il nous offre de manière à ce que le remplissage de la matrice soit le plus rapide possible.

OSPF met à jour la table des liens à chaque changement de topologie, et les tables sont également rafraîchies toutes les trente minutes. Il faudra donc que notre algorithme de remplissage de la matrice en tienne compte.

Enfin, la version 7 de notre projet prévoit d'utiliser la QoS[4] pour paramétrer le routage. Les données concernant la QoS devront être récupérées en même temps que la topologie. La matrice changera alors et chaque case contiendra un ensemble de valeurs portant sur les liens : vitesse de propagation sur le lien, ou encore le taux de perte observé au niveau du lien.

De même que pour remplir la matrice des liens, la matrice des valeurs de la QoS sera remplie automatiquement à l'aide du protocole QOSPF qui est une version améliorée de OSPF. Il associe en effet chaque lien à des valeurs de QoS, et il nous restera donc plus qu'à remplir la matrice avec ces valeurs.

Après avoir vu comment on récupère la topologie, nous allons voir comment on se sert de cette topologie pour construire l'arbre de routage qui sera contenu dans le paquet TBXcast.

[3] www.quagga.net
[4] Quality Of Service : voir le rapport de spécifications pour plus de précisions

3.2 Création de l'arbre

Une fois la topologie du réseau récupérée, nous avons tout ce qu'il faut à notre disposition pour créer l'arbre de routage. L'arbre de routage sera conçu dans une fonction annexe. Chaque algorithme sera implémenté dans une nouvelle fonction. Seuls les paramètres d'entrées (topologie et destinataires) et le format de sortie seront communs aux différentes fonctions. Cela permettra à l'utilisateur de choisir l'algorithme de création de l'arbre qu'il désire.

L'arbre de routage sera amené à évoluer lorsque TBXcast intégrera la gestion de la QoS, dans sa version 7. Nous ne prévoyons pas d'atteindre cette version cette année, mais nous désirons concevoir TBXcast de manière à ce qu'il puisse être amélioré. Pour l'instant, l'arbre créé sera un arbre couvrant de plus court chemin[5]. La création de l'arbre de routage se placera dans le cadre de LibTBXcast.

Pour élaborer l'arbre de routage nous avons besoin, en entrée, de la topologie du réseau et de la liste des destinataires. La topologie du réseau devra faire état des routeurs TBXcast qui seront alors les nœuds de l'arbre. Les destinataires seront soit des feuilles, soit des nœuds dans le cas où un routeur est aussi récepteur d'un paquet TBXcast. En sortie, l'algorithme de création devra rendre un arbre couvrant de longueur minimale, ayant pour racine la machine source.

Pour commencer, nous nous baserons sur l'algorithme de Moore-Dijkstra. Ce dernier permet de résoudre le problème de la recherche d'un chemin de coût minimal entre un sommet donné et tous les autres sommets. Dans notre cas, le sommet donné sera la source, tandis que les autres sommets correspondront aux destinations et aux routeurs TBXcast. Nous proposons ci-dessous l'algorithme en pseudo-code.

[5] Arbre sous-ensemble d'un graphe qui connecte les sommets ensembles de manière à minimiser la taille d'un chemin depuis la racine vers une feuille quelconque

```
R : ensemble des routeurs TBXcast issus de la topologie
D : ensemble des destinataires du paquet TBXcast
X = R ∪ D
Γ : ensemble des arêtes

λ(i) : longueur minimale d'un chemin de la racine à i
pred(i) : prédécesseur du sommet i
S : ensemble des sommets de l'arbre couvrant des plus courts chemins
racine : source du paquet TBXcast

// Initialisation
S := {racine}
λ(racine) := 0
fin := faux

Pour i ∈ X-{racine}
     Si (racine, i) ∈ Γ
     Alors
          λ(i) = 1
          pred(i) = racine
     Sinon
          λ(i) = +∞
          pred(i) = -1
     Fin Si
Fin Pour

// Recherche des chemins de longueurs minimales
Tant que fin == faux faire
     Si pour chaque destinataire d ∈ D, λ(d) != +∞
     Alors fin := vrai
     Sinon
          Choisir i ∈ X-S tel que λ(i) = Min λ(j) pour j ∈ X-S
          S := S ∪ {i}

          Pour j ∈ Γ(i) ∩ (X-S) faire
               Si λ(j) > λ(i) + distance(i,j) alors
                    λ(j) := λ(i) + distance(i,j)
                    pred(j) := i
               Fin Si
          Fin Pour
     Fin Sinon
Fin Tant que
```

Suite à l'application de cet algorithme, nous disposerons d'un arbre couvrant de plus court chemin. Par contre, il se peut qu'il reste des sommets inutiles dans l'arbre. Nous proposons alors d'élaguer l'arbre en partant des feuilles, de manière à ne garder que les sommets significatifs pour le routage.

3.3 Codage de l'arbre

TBXcast comportera un arbre dans sa structure, ce qui le diffère de Xcast. Il faudra trouver la structure et ensuite l'intégrer dans le paquet. Dans cette partie, on va surtout présenter les différentes structures retenues pour cet arbre. Ces structures devront répondre à quelques critères bien précis :

- Compacité du code dans l'entête pour maximiser le payload[6]
- Minimisation du nombre de changements à apporter aux structures existantes dans Xcast
- Efficacité du traitement dans les routeurs intermédiaires.

Pour ce qui est de l'efficacité et de la minimisation, on va conserver le bitmap[7] introduit par Xcast avec une utilisation différente. Dorénavant, le bitmap servira uniquement à indiquer les destinataires dans l'arbre. Pour ce qui est de la compacité on va créer une structure le plus simple possible, mais elle sera obligatoirement plus volumineuse que la liste des destinataires présente dans Xcast.

On va maintenant confronter deux structures d'arbres complètement différentes. La première correspond à l'étude du groupe de l'année dernière sur la représentation de l'arbre, quant à l'autre, elle correspond à notre réflexion de cette année. Pour les deux représentations, on conservera le même exemple pour bien comprendre les différences. Sur cet exemple, les destinataires sont représentés par un petit point rouge. De plus, dans chaque représentation, on numérotera les nœuds selon un parcours préfixe[8].

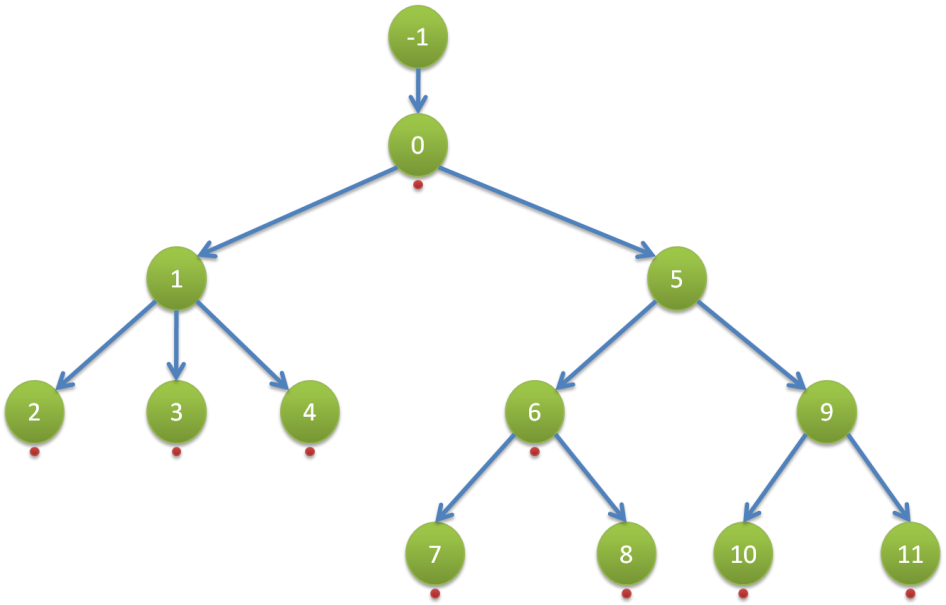


Figure 4 : Exemple d'arbre de diffusion

3.3.1 Représentation 1

Dans ce nouveau protocole, on codera les routeurs de branchement dans l'arbre de diffusion du paquet (plus les nœuds intermédiaires imposés par la QoS). On va s'inspirer d'un protocole de routage arborescent existant : ERM. Nous aurons donc trois listes pour stocker les informations : les adresses IPv6, les pères du nœud dans l'arbre et le bitmap des destinataires. A ces listes on va ajouter un offset pour représenter le nœud courant, ce qui facilitera grandement les calculs à effectuer dans les routeurs intermédiaires. Pour bien comprendre ce principe, on va l'appliquer à l'exemple ci-dessus.

Les nœuds et les feuilles sont codés par un entier présent sur le graphe. Cet entier représente l'indice dans les tableaux. On aura aussi besoin de ces entiers pour coder la liste des pères. Sur la figure ci-

[6] Le payload, ou charge utile, est la quantité d'information utile transportée par un paquet
[7] Le bitmap de Xcast est une suite de bits valant 0 ou 1 suivant si le destinataire correspondant a déjà été desservi
[8] Le parcours préfixe correspond à un parcours de l'arbre descendant gauche-droit

dessus, les points rouges représenteront les destinataires (un routeur de branchement peut aussi être un destinataire). On remarque aussi que le premier nœud qui correspond à la source aura l'indice -1 car il n'apportera aucune information pour la suite du routage. Néanmoins son adresse restera présente dans le champ source de l'entête IPv6. On aura donc les 3 listes suivantes.

Indice	0	1	2	3	4	5	6	7	8	9	10	11
Pointeur sur les pères	-1	0	1	1	1	0	5	6	6	5	9	9
Bitmap des Destinataires	1	0	1	1	1	0	1	1	1	0	1	1
Adresses IPv6	@	@	@	@	@	@	@	@	@	@	@	@

Figure 5 : Tableau représentant la structure de l'arbre dans l'entête (représentation 1)

A ce tableau, il faut rajouter aussi l'offset. L'offset prendra le numéro du nœud courant. Par exemple, si on se trouve au routeur numéro 5 alors l'offset aura le numéro 5.

3.3.2 Représentation 2

Cette solution requiert une structure particulière supplémentaire. Elle est composée de :

- **Un entier codé sur 7 bits** qui décrit la longueur d'un sous-arbre fils
- **128 bits réservés** pour l'adresse IPv6 correspondant au nœud
- **1 bit** pour le bitmap

La structure aura donc une taille maximale de 17 octets. La longueur d'un sous-arbre fils s'obtient en additionnant toutes les longueurs des fils plus 1. Lorsque c'est une feuille de l'arbre, la longueur est de 1. Il représente le nombre de nœuds dans le sous-arbre.

L'arbre sera représenté par une liste de la structure définie ci-dessus. Reprenons l'exemple de la Figure 4 :

Indice		0	1	2	3	4	5	6	7	8	9	10	11
Structure	Adresse	@	@	@	@	@	@	@	@	@	@	@	@
	Longueur	12	4	1	1	1	7	3	1	1	3	1	1
	Bitmap	1	0	1	1	1	0	1	1	1	0	1	1

Figure 6 : Topologie de la fig.4 selon la deuxième représentation

La solution 1 paraît être la solution collant le plus au code de Xcast. Elle reste compacte et moyennement efficace. Nous verrons dans la suite que la solution 2 permet une meilleure efficacité dans les routeurs intermédiaires. On va donc tester les différentes solutions pour savoir combien de temps représente l'intégration de la deuxième solution et donc savoir si elle est viable. On verra dans la partie *4.2.1* les différentes étapes et algorithmes de parcours de l'arbre de routage sur l'exemple présenté Figure 4.

3.4 Segmentation

Il peut arriver que l'arbre de routage soit trop grand pour qu'il puisse être stocké entièrement dans un seul paquet TBXcast. Nous allons devoir le diviser en plusieurs arbres. Cette division se nomme la segmentation et sera incluse dans TBXcast à partir de la version 4. Le code correspondant à la segmentation sera situé dans LibTBXcast. Tout comme la création de l'arbre, on proposera plusieurs algorithmes de segmentation de l'arbre. Chaque algorithme sera codé dans des fonctions différentes.

La taille de l'arbre de routage est directement proportionnelle à son nombre de nœuds. En effet, l'arbre de routage peut contenir un nombre maximal n_{max} de nœuds, déterminé par l'espace disponible dans l'entête du paquet. Si l'arbre contient plus de n_{max} nœuds, il devra alors être segmenté.

Nous devons ainsi transformer l'arbre en une forêt **F**, de manière à ce que tout arbre **a** $\in$ **F** contienne au maximum n_{max} nœuds.

Nous proposons un premier algorithme de segmentation. Nous allons tout d'abord associer une valeur v_n à chaque nœud de l'arbre d'origine, suivant la formule récursive suivante:

$$\mathbf{v_n} = \mathbf{1} + \sum \mathbf{v_f} \text{ avec } \mathbf{f} \text{ un fils de } \mathbf{n} \text{ dans l'arbre d'origine}$$

Suite à cette valuation, le nœud n ayant la plus grande valeur v, inférieure ou égale à n_{max} est choisi. Le sous-arbre de racine n est alors ajouté à la forêt F. On répète l'étape jusqu'à ce que l'arbre d'origine soit vide. Nous proposons ci-dessous l'algorithme en pseudo-code.

```
En entrée : un arbre couvrant T ayant plus de n_max noeuds
En sortie : une forêt F d'arbre ayant moins de n_max noeuds

F = vide
Tant que T != vide faire
     Pour chaque noeud n de T faire
          Calculer la valeur vn associée au noeud n
     Fin Pour

     n* = le noeud de T ayant la plus grande valeur inférieure
ou égale à n_max
     t* = le sous-arbre de T enraciné en n*
     Ajouter à t* un plus court chemin de la racine de T au
noeud n*
     Ajouter t* à F
     Enlever de T les membres du sous-arbre de t*
Fin Tant que
```

4 Routage d'un paquet TBXcast

Dans TBXcast, le routage d'un paquet présente deux aspects :

- La manière dont le paquet est acheminé vers le routeur suivant
- La manière dont un routeur analyse et traite l'arbre en vue de la réexpédition du paquet

Le premier point soulève la question de l'homogénéité du réseau : tous les routeurs supportent-ils TBXcast ? Que faut-il faire si ce n'est pas le cas ? Ce sont des questions sur lesquelles nous nous sommes penchés, et notre analyse est présentée ci-après en partie 4.1.

Le deuxième point relève du parcours de l'arbre de routage, et des algorithmes de parcours seront décrits en partie 4.2.

4.1 TBXcast dans un réseau non-homogène

Par « réseau non-homogène », nous entendons le non-support du protocole TBXcast par un ou plusieurs routeurs constituant ce réseau. Nous nous sommes penchés sur ce problème pour les raison suivantes :

- TBXcast étant un protocole expérimental, et ne risquant pas d'être largement utilisé à court terme, on se doit de considérer le cas où la plupart des routeurs du réseau ne sont pas équipés du driver TBXcast
- Comme nous l'avons vu dans la partie précédente, les nœuds de branchement explicités dans l'arbre supportent TBXcast, et on souhaiterait que le transport des informations entre ces nœuds soit transparent (on se base directement sur de l'unicast).

4.1.1 Une première approche

Pour réaliser notre protocole, nous nous basons sur Xcast qui lui-même implémente un pseudo-device (cf. rapport de spécification fonctionnelle). Pour rappel, le système redirige les données vers ce pseudo device si et seulement si l'adresse de destination du paquet IPv6 entrant est à « Xcast_All_Routers[9] ». Ayant décidé de se baser sur Xcast, nous devons nous soumettre à son architecture.

Ainsi, nous avions envisagé la solution suivante (Figure 7) qui consiste à simplement mettre l'adresse du routeur suivant significatif **supportant TBXcast** dans le champ de destination de l'entête du paquet, et nous reposer sur les tables de routage des routeurs intermédiaires pour le bon acheminement du paquet.

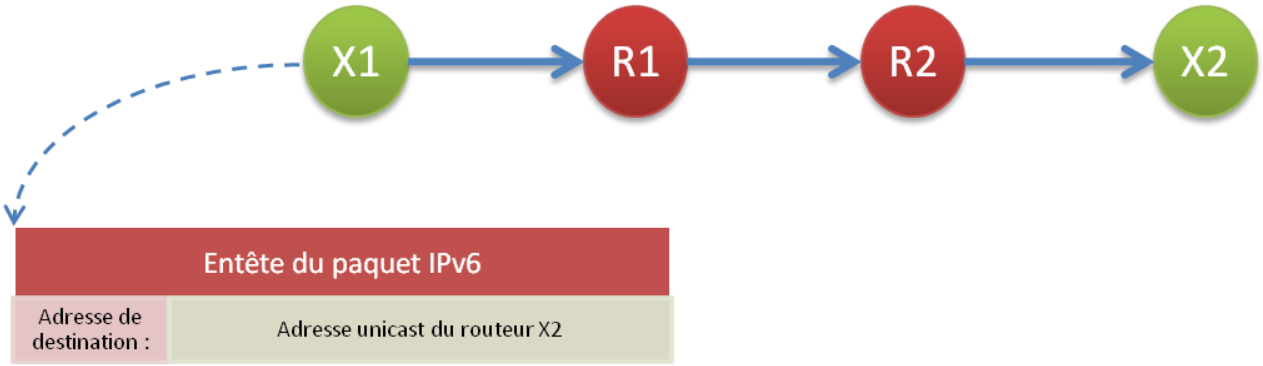


Figure 7 : une première (fausse) solution

[9] Xcast_All_Routers fait référence à une adresse de groupe de type multicast spécial, à laquelle appartiennent tous les routeurs du réseau supportant le protocole Xcast. Nous reprendrons ce principe avec l'adresse TBXcast_All_Routers.

Cette solution est donc (presque) fausse aux vues de ce que l'on a précisé : afin que le paquet soit pris en considération par le driver TBXcast, il faut laisser « TBXcast_All_Routers » dans le champ adresse de destination du paquet IPv6. Le paquet serait bien arrivé au bon routeur (X2 sur la Figure 7), mais il aurait été détruit car le protocole correspondant à l'extension d'entête (en l'occurrence TBXcast) n'aurait pas été reconnu par l'interface réseau standard.

Il existe néanmoins une alternative dans laquelle une adresse IP peut être spécifiée dans le champ destination de l'entête IPv6. C'est possible dans le cas où très peu de routeurs TBXcast sont présents par réseau, voir un unique routeur dans un sous-réseau donné. Dans ce cas-là, le routeur TBXcast agirait comme un « serveur » vis-à-vis des autres routeurs, dans le sens ou ces derniers connaîtraient l'adresse spéciale de l'unique routeur TBXcast (cette solution est détaillée au point 11.5 du RFC5058). Même si cette approche ne nous semble pas prioritaire, il est bon de savoir qu'elle existe pour la suite.

De manière générale, ne voulant pas nous restreindre au cas d'un réseau homogène comme justifié en introduction, nous avons envisagé une nouvelle solution.

4.1.2 Le tunneling

Le tunneling est une notion qui nous est plutôt familière. En effet, lorsque nous avons étudié en détail les fonctionnalités de Xcast (RFC5058), nous avons vu la possibilité de traverser des routeurs ne supportant pas Xcast : c'est le tunneling IPv6.

En premier lieu, nous nous sommes dit que le tunneling ne serait pas nécessaire car dans TBXcast tous les routeurs de branchement sont connus à l'avance dans l'arbre, contrairement à Xcast où les routeurs suivants ne le sont pas. Au vu du problème précédent, nous nous sommes intéressés au tunneling de manière plus précise et nous allons donc le présenter ici tel qu'il est implémenté dans Xcast.

4.1.3 Packet tunneling IPv6

Le protocole IPv6 offre, par nature, une possibilité de tunneling. Il consiste à encapsuler un paquet IPv6 (entête plus charge utile) dans le champ de données d'un autre paquet IPv6. L'entête du paquet **encapsulé** est appelé « inner IP header», et son nouvel entête « outer IP header ».

Tant que le paquet n'a pas atteint la destination correspondant à l'adresse de destination spécifiée dans le nouvel entête, les routeurs intermédiaires vont le transférer jusqu'au routeur désiré qui est la sortie du tunnel. Seul ce routeur pourra alors **décapsuler** ce paquet et le traiter comme s'il venait de le recevoir avec son entête original. (voir Figure 8)

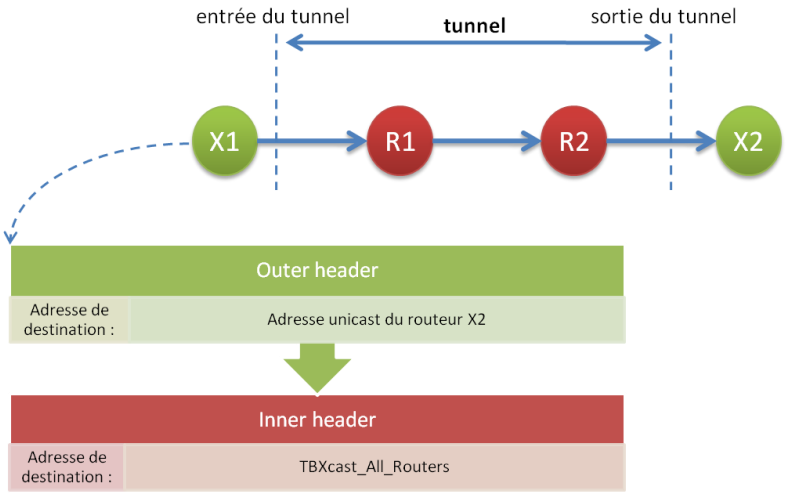


Figure 8 : encapsulation et tunneling dans IPv6

Le tunneling dans IPv6 et plus spécifiquement dans Xcast a été étudié dans la salle d'expérimentation, et cette analyse nous a aidé à élaborer cette partie du rapport. Se référer à l'annexe 2 pour les détails.

4.1.4 Tunneling dans TBXcast

Compte tenu de la nature de TBXcast, qui est un protocole de routage **explicite**, on peut supposer qu'il existera en général un grand nombre de routeurs intermédiaires entre deux nœuds de branchement significatifs. En effet, les protocoles de routage explicite trouvent leur intérêt lorsque les destinataires sont clairsemés.

Ainsi, le tunneling est tout à fait adapté à la circulation des paquets entre les routeurs intermédiaires, qui de plus supportent tous l'unicast. C'est pourquoi nous avons décidé d'utiliser l'encapsulation IPv6 pour **chaque transfert** et de ne pas le limiter aux seules zones non-homogènes du réseau.

4.2 Analyse de l'entête et de l'arbre par un routeur

Dans les routeurs de branchement décrits dans l'arbre de routage, les paquets TBXcast doivent subir un traitement spécifique qui sera effectué par le driver TBXcast après l'arrivée du paquet dans le routeur.

Le traitement du paquet va être similaire à celui effectué dans Xcast, le principe étant de faire circuler le paquet vers les bons routeurs pour qu'il arrive aux destinations voulues.

Le paquet reçu contient l'arbre de routage, on peut donc trouver les fils du routeur courant dans cet arbre. Pour chacun de ces fils, il faut différencier deux cas :

- Le fils est une feuille de l'arbre
- Le fils est un nœud de branchement

Si c'est un nœud de branchement, on va envoyer un nouveau paquet TBXcast contenant les mêmes données que le paquet reçu et en remplaçant l'arbre de routage par le nouveau sous-arbre qui aura pour racine le nœud fils auquel on envoie le paquet. Ce paquet sera ensuite encapsulé dans un paquet IPv6 comme expliqué dans la partie 4.1 et sera envoyé à l'adresse du nœud par unicast.

Si le fils est une feuille, il faudra alors transformer le paquet TBXcast en un paquet unicast pour que le fils puisse le recevoir sans traitement supplémentaire. Cette action sera effectuée par une fonction *tbx2u* qui est déjà implémentée dans Xcast sous le nom de *_x2u_forward* et que nous réutiliserons.

Lorsqu'un routeur intermédiaire reçoit un paquet et que ce routeur est aussi destinataire, il faut alors remonter les données contenues dans le paquet vers les applications avant le traitement sur l'arbre. On saura qu'on à affaire à un destinataire car son bit dans le bitmap sera à 1.

Dans la suite, nous allons voir plus en détails comment cela se déroule exactement dans les routeurs : comment on trouve les nœuds fils et comment on change l'arbre dans le paquet.

4.2.1 Parcours de l'arbre de routage

Pour voir comment se déroule le parcours de l'arbre de routage dans les routeurs, nous allons devoir reprendre les différentes solutions de représentation de l'arbre étudiées dans la partie 3.3. Pour chaque solution, nous allons étudier comment on trouve les fils du nœud courant et comment on détecte si un fils est une feuille ou non. Nous verrons aussi comment on remplace le nouvel arbre dans le paquet envoyé.

4.2.1.1 Représentation 1

Pour cette solution, on rappelle que l'arbre est représenté par un offset représentant le nœud courant, un tableau sur les pères, un bitmap des destinataires et un tableau des adresses IP ; chacun de ces champs étant représenté dans l'entête du paquet TBXcast (voir partie 5, récapitulatif du format des entêtes).

Le driver TBXcast va tout d'abord regarder le champ offset ; l'adresse IP de ce nœud doit correspondre au routeur traitant le paquet. Nous allons ensuite parcourir le tableau des pères afin de trouver les fils du nœud courant. Si le numéro contenu dans la case **i** de ce tableau est égal à l'offset, alors le nœud **i** est un fils du nœud courant. On commence le parcours à partir de l'offset car ses fils sont forcément situés après, puis on l'arrête lorsque l'on a atteint le nombre de nœuds.

Pour chacun de ces nœuds fils ainsi trouvés, il faut maintenant regarder si celui-ci est une feuille ou non. Pour vérifier cela, nous allons à nouveau parcourir le tableau des pères à partir de l'indice du fils. Si une case contient le numéro du nœud fils que l'on est en train de traiter, alors ce nœud fils est un nœud de branchement, sinon aucun nœud ne l'a comme père et c'est donc une feuille de l'arbre de routage.

On peut maintenant différencier les deux traitements à effectuer. Si le nœud fils que l'on traite est une feuille, on va alors envoyer un paquet unicast à ce nœud. Si le nœud fils en cours de traitement est un nœud de branchement, il faut alors lui envoyer un paquet TBXcast qui est la copie du paquet reçu. On change alors le champ offset de ce nouveau paquet pour fixer sa valeur au numéro du nœud auquel on envoie le paquet. Ainsi le prochain routeur saura trouver le début de l'arbre qu'il doit analyser.

Le traitement effectué dans le routeur peut être résumé par l'algorithme suivant :

```
pour i de offset à nombre de nœuds faire
     si père[i] = offset alors
          feuille <- true
          pour j de i à nombres de nœuds faire
               si père[j] = i alors
                    feuille <-false
                    break
               fsi
          fpour
     si feuille = true alors tbx2u(adresse[i])
     sinon     copier paquet reçu
          mettre offset du nouveau paquet à i
          envoyer(adresse[i])
     fsi
fpour
```

Si on regarde la complexité de cet algorithme en comptant le nombre d'accès dans le tableau, on remarque que cette complexité est en $O(n^2)$ où n est le nombre de cases dans le tableau.

On peut dérouler l'algorithme pour l'exemple étudié dans la partie 3.3, si on se situe dans le nœud **5** par exemple :

Le routeur reçoit un paquet contenant les différentes listes. Ces listes seront toujours les mêmes pour les différents routeurs. L'offset situé dans l'entête est à 5, on va donc regarder le tableau des pères à partir de la case 5. On trouve que les nœuds 6 et 9 ont le nœud 5 comme père, ce sont donc des fils de ce nœud. On va donc envoyer un paquet à chacun de ces nœuds.

Pour le nœud 6, on peut voir que le nœud 7 l'a comme père, ce n'est donc pas une feuille. On va alors lui envoyer un paquet TBXcast qui est la copie du paquet reçu avec le champ offset mis à 6. Il en est de même pour le nœud 9.

4.2.1.2 Représentation 2

Pour cette deuxième solution, le traitement va être différent. On rappelle que cette fois-ci l'arbre est représenté par un tableau contenant une structure pour chaque nœud. Cette structure est composée de la longueur du sous-arbre du nœud, d'un bit indiquant s'il est destinataire et de son adresse IP.

La différence majeure est que cette fois le routeur ne reçoit un paquet qu'avec son sous-arbre dans l'entête. Si on prend l'exemple donné dans la partie 3.3, le nœud 5 va recevoir un paquet avec les nœuds 5 à 11 et le nœud 1, un paquet avec les nœuds 1 à 4.

Pour trouver les fils du nœud courant, le routeur va parcourir la structure donnée selon la méthode suivante : il se place d'abord au début du tableau où se trouve le nœud courant, puis il regarde la case suivante dans laquelle se trouve le premier fils du nœud courant, selon le parcours descendant gauche-droite que l'on a effectué pour construire le tableau. Ensuite, on va ajouter à l'indice du tableau (qui est à 1 pour le moment) la longueur de l'arbre associé au premier fils. Ainsi on va sauter tout l'arbre qui dérive du premier fils pour tomber directement sur le deuxième fils du nœud courant. On va procéder comme cela pour trouver tous les fils jusqu'à ce que l'indice du tableau soit plus grand que la longueur de l'arbre associée au premier nœud (c'est-à-dire lorsqu'on sort du tableau).

Pour chaque fils ainsi trouvé, on va pouvoir envoyer un nouveau paquet contenant les mêmes données que le paquet reçu. Si le fils est une feuille (longueur associée égale à 1), on envoie un paquet unicast et si le fils est un nœud de branchement, on envoie un paquet TBXcast. L'arbre contenu dans le nouveau paquet va être une copie de n cases du tableau reçu en partant de l'indice du nœud fils où n est la longueur de l'arbre associée au nœud fils. Ainsi le routeur suivant ne va recevoir que le sous-arbre dans lequel il sera la racine.

Le traitement effectué dans le routeur peut être résumé par l'algorithme suivant (on appelle **tab** le tableau des structures) :

```
i <- 1
tant que i < tab[0].longueur faire
     si tab[i].longueur = 1 alors
          tbx2u(tab[i].adresse)
     sinon     construire nouveau paquet TBXcast avec données
               du paquet reçu
          copie de tab[i].longueur case du tableau dans le
nouveau paquet
          envoi(tab[i].adresse)
     fsi
     i = i+tab[i].longueur
ftq
```

La complexité de ce nouvel algorithme en comptant également le nombre d'accès dans le tableau est cette fois-ci en *O(n)*.

On peut dérouler l'algorithme sur l'exemple étudié, lorsqu'un paquet arrive au nœud 5.

Le routeur représenté par le nœud 5 recevra l'arbre suivant :

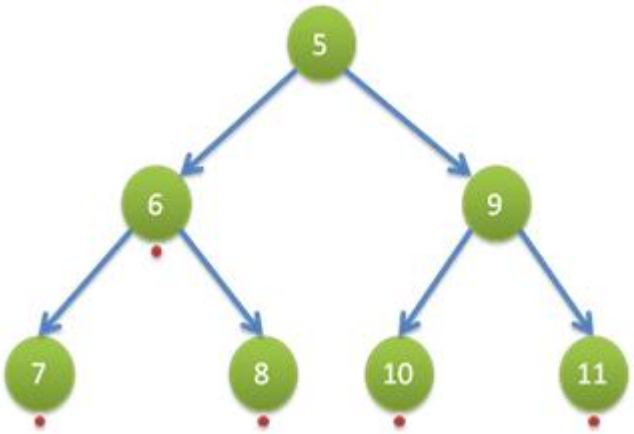


Figure 9 : Arbre reçu par le nœud 5

Cet arbre sera alors représenté par le tableau suivant :

Indice		0	1	2	3	4	5	6
Nœud correspondant		5	6	7	8	9	10	11
Structure	Adresse	@	@	@	@	@	@	@
	Longueur	7	3	1	1	3	1	3
	Bitmap	0	1	1	1	0	1	1

Figure 10 : Tableau reçu par le nœud 5

La première case de ce tableau correspond au nœud 5, sa longueur associée est de 7. On regarde la case 1 du tableau correspondant au nœud 6, c'est un fils du nœud 5. Sa longueur associée est de 3, il faut donc lui envoyer un paquet TBXcast. On construit un nouveau paquet qui contiendra les mêmes données que le paquet reçu et dont l'arbre va être la copie des 3 cases du tableau à partir de la case 1 (les cases 1, 2 et 3 correspondant aux nœuds 6, 7 et 8 de l'arbre). Cela aura pour effet d'envoyer l'arbre suivant au nœud 6 (c'est le sous-arbre du nœud 6 de longueur 3) :

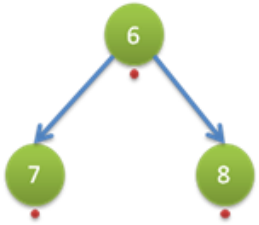


Figure 11 : Arbre envoyé au nœud 6

Indice		0	1	2
Nœud correspondant		6	7	8
Structure	Adresse	@	@	@
	Longueur	3	1	1
	Bitmap	1	1	1

Figure 12 : Tableau envoyé au nœud 6

La longueur située à la case 1 est 3, on va donc trouver le fils suivant du nœud courant à la case 4 (1+3). Cette case correspond au nœud 9 de l'arbre, sa longueur associée est de 3, il va donc subir le même traitement que le nœud précédent en envoyant un nouveau paquet TBXcast contenant les nœuds 9, 10 et 11.

On va trouver le prochain fils à la case 7 (4+3), or cette case sort du tableau car 7 est la longueur associée à la case 0 du tableau. Le parcours va donc pouvoir s'arrêter là.

4.2.2 Bilan

Si on compare les deux algorithmes, on peut voir que le deuxième est plus rapide au niveau du parcours du tableau car on passe directement d'un fils à l'autre en connaissant leurs positions et on évite ainsi de regarder chaque case du tableau. De plus on peut détecter directement qu'un nœud est une feuille sans avoir à re-parcourir le tableau comme on le fait avec la première représentation ce qui se traduit par une complexité moins importante. La deuxième représentation est donc la plus efficace

des deux. Cependant elle va entraîner plus de changements dans le code de Xcast car le traitement effectué et les structures utilisées sont très différentes de celles utilisées dans Xcast. Il faudra vérifier l'efficacité de la deuxième représentation par rapport à la première pour pouvoir effectuer un choix.

5 Entêtes

Un protocole de routage est toujours identifié par des entêtes spécifiques, car ce sont ces entêtes qui contiennent les informations le caractérisant. Nous avons à présent suffisamment d'éléments pour concevoir de manière précise les entêtes nécessaires à TBXcast :

- Le tunneling, via l'encapsulation des paquets IPv6, sera utilisé dans tous les cas
    - Cela implique deux entêtes IPv6 : **outer** header (entête de tunnel) et **inner** header (entête original du paquet)
- Nous connaissons la structure du paquet TBXcast : on se base sur Xcast et on y remplace la liste des destinataires à plat par notre arbre de routage

La Figure 11 montre l'agencement des différents entêtes, la taille et la valeur de leurs différents champs. On peut noter les éléments suivants :

- Dans le paquet TBXcast, le champ « X » n'est plus justifié (voir rapport de spécification pour les détails) : en effet, on utilise naturellement l'unicast (grâce au tunneling) et la transformation d'un paquet TBXcast en paquet unicast est implicite. Dans l'esprit d'un minimum de modifications, nous le laisserons malgré tout tel quel
- « Routing Tree » désigne bien sûr notre arbre de routage. C'est ici que la structure d'arbre définie dans la partie 3.3 sera placée.
- « Payload » désigne la charge utile, c'est-à-dire l'information que le destinataire du paquet va exploiter

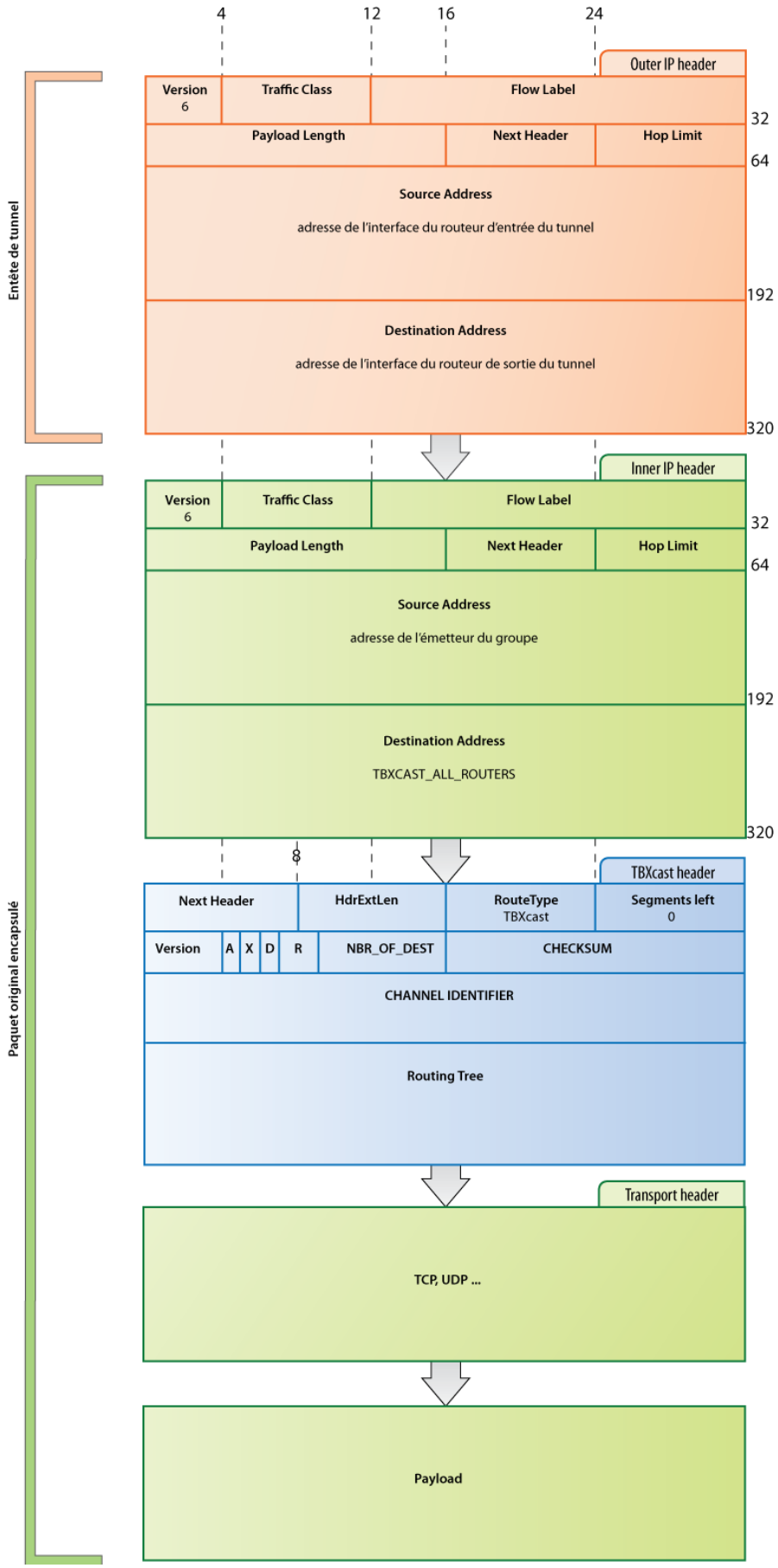


Figure 11 : Enchaînement et format des entêtes

6 Conclusion

Cette phase de conception nous a permis de raffiner notre connaissance de Xcast et d'établir de nouvelles structures de données et algorithmes spécifiques pour TBXcast. Nous avons repris de l'équipe de l'année dernière la structure représentant la topologie. Pour la représentation de l'arbre de routage, nous avons confronté la solution de l'année dernière avec notre nouvelle solution, proposant une structure particulière pour chaque nœud de l'arbre. En tirant parti du tunneling, nous allons pouvoir proposer un routage unicast et transparent entre les nœuds explicitement codés dans l'arbre de routage. Avec le support précieux qu'est notre salle d'expérimentation, nous sommes prêts à nous lancer dans la réalisation de la première version de TBXcast. La planification initiale nous servira de repère dans un premier temps, mais sera sûrement amenée à changer au vu de la nature « imprévisible » du développement qui nous attend. Afin d'être le plus rigoureux possible, nous attacherons une grande importance aux tests unitaires et aux validations de nos différents versions.

7 Annexes

7.1 Canevas des versions

Rappelons que chaque version possède ses propres phases de **développement**, **test** et **validation**. Malgré tout, la **conception commune** initiale implique que notre stratégie de développement ne puisse pas être qualifiée d'en « spirale ».

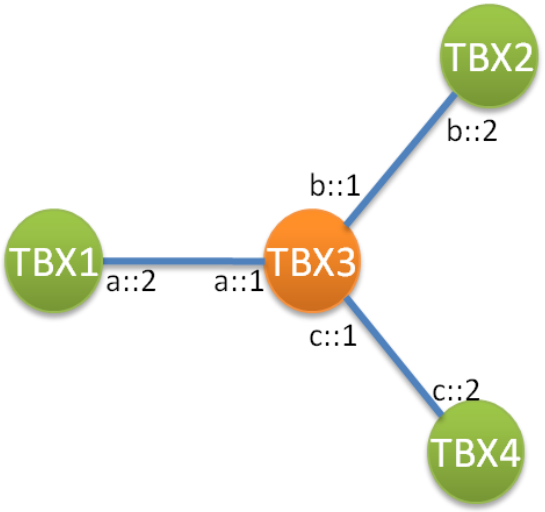
Version 0	Identifier les fichiers à modifier. Changer les noms de fonctions, des variables et des constantes en les préfixant différemment (tbx_). Modifier la valeur des identifiants (identifiant paquet, adresse multicast globale…) **Objectif :** la compilation se déroule bien, et Xcast marche en parallèle (on aura donc dupliqué le pseudo-device)
Version 1	Construction de la structure de données simplifiée de l'arbre. Modifier la structure de l'entête. Légers changements dans le traitement des paquets. Ajouter le tunneling à chaque renvoi. Se familiariser d'avantage avec les zones de code à modifier (traitement des routeurs, codage des entêtes sous NetBSD) **Objectif :** Possibilité d'acheminer un paquet à travers un réseau contenant des routeurs non TBXcast.
Version 2	Implémentation de la structure d'arbre finale dans le code. Il faut prévoir un moyen pour fournir l'arbre à la source. Modifier la fonction de traitement des paquets dans les routeurs intermédiaires pour tenir compte de l'arbre – implémentation de l'algorithme de parcours de l'arbre. **Objectif :** Les routeurs doivent être capables de transférer les paquets en parcourant l'arbre figurant dans l'entête.
Version 3	Création de l'arbre. Implémentation de l'algorithme de création de l'arbre dans LibTBXcast à partir de la représentation de la topologie. Il est nécessaire de passer la structure représentant la topologie à LibTBXcast. Cette représentation doit être conforme au format qui sera utilisé plus tard. **Objectif :** La source est capable de fabriquer l'arbre à partir de la topologie et la liste des destinataires.
Version 4	Implémentation de l'algorithme de segmentation de l'arbre dans LibTBXcast. **Objectif :** La source doit être capable d'optimiser la charge sur les arbres lesquelles elle envoie.
Version 5	Ajout de la gestion des groupes dans LibTBXcast. Réutilisation et modification des méthodes existantes dans le code Xcast. **Objectif :** La source doit être capable de créer un groupe et le maintenir, c'est-à-dire le modifier en cas d'un nouvel utilisateur ou la déconnexion d'un ancien.
Version 6	Récupération de la topologie avec OSPF. On doit être capable d'utiliser OSPF pour récupérer une représentation de la topologie. On va remplacer la topologie fournie par des méthodes capables de la créer et de la maintenir à jour. **Objectif :** LibXcast n'aura plus besoin d'une topologie fournie, mais la récupère automatiquement via une interface à définir. TBXcast est maintenant entièrement fonctionnel.
Version 7	Gestion de QoS, extension de l'algorithme de création de l'arbre. Il faut augmenter la représentation de la topologie par les valeurs de QoS. Implémentation des algorithmes modifiés de calcul de l'arbre. **Objectif :** La source doit être capable de fournir un arbre respectant les contraintes de QoS particulières.

7.2 Rapport d'analyse du tunneling de Xcast

L'enjeu est de tester la gestion du tunneling par Xcast pour permettre d'identifier le code à modifier/éliminer, et aussi mieux comprendre l'interaction entre le protocole IPv6 et le protocole Xcast. Un premier test sommaire a été fait avec tcpdump pour vérifier que Xcast gérait bien ce tunneling. Ici, on poursuit l'étude avec Wireshark pour analyser cette fonctionnalité plus en profondeur.

7.2.1 Protocole de test

Pour le test du tunneling, on a une configuration de base du type suivant :

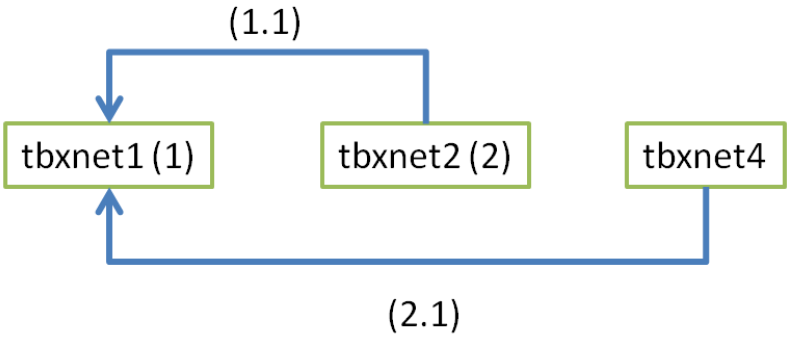


Soit un routeur central (tbxnet3) non compatible Xcast, une source (tbxnet1) compatible Xcast et des destinations (tbxnet2 et tbxnet4) compatibles Xcast. On fait un ping6x sur tbxnet1 et on scanne avec Wireshark sur tbxnet1, tbxnet2, et tbxnet4 les interfaces correspondantes aux adresses respectives a::2, b::2 et c::2.

7.2.2 Résultats

Pour chaque ping, un paquet icmpv6 est envoyé de a::2 à b::2, lequel répond directement au ping, et transmet un paquet icmpv6 à c::2. c::2 répond ensuite directement à a::2.

On obtient donc :



7.2.3 Analyse

Pour comprendre comment le protocole réussit à « tunneler » le paquet, on peut analyser un paquet capturé par Wireshark :

```
Frame 2 (190 bytes on wire, 190 bytes captured)
    Arrival Time: Feb  2, 2009 22:58:21.037784000
    [Time delta from previous captured frame: 0.125859000 seconds]
    [Time delta from previous displayed frame: 0.125859000 seconds]
    [Time since reference or first frame: 0.125859000 seconds]
    Frame Number: 2
    Frame Length: 190 bytes
    Capture Length: 190 bytes
    [Frame is marked: False]
    [Protocols in frame: eth:ipv6:ipv6:icmpv6:data]
Ethernet  II,  Src:  D-Link_71:16:6f  (00:1b:11:71:16:6f),  Dst:  D-Link_71:0c:3a
(00:1b:11:71:0c:3a)
    Destination: D-Link_71:0c:3a (00:1b:11:71:0c:3a)
        Address: D-Link_71:0c:3a (00:1b:11:71:0c:3a)
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
        .... ..0. .... .... .... .... = LG bit: Globally unique address (factory
default)
    Source: D-Link_71:16:6f (00:1b:11:71:16:6f)
        Address: D-Link_71:16:6f (00:1b:11:71:16:6f)
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
        .... ..0. .... .... .... .... = LG bit: Globally unique address (factory
default)
    Type: IPv6 (0x86dd)
Internet Protocol Version 6
    0110 .... = Version: 6
        [0110 .... = This field makes the filter "ip.version == 6" possible: 6]
    .... 0000 0000 .... .... .... .... .... = Traffic class: 0x00000000
    .... .... .... 0000 0000 0000 0000 0000 = Flowlabel: 0x00000000
    Payload length: 136
    Next header: IPv6 hop-by-hop option (0x00)
    Hop limit: 64
    Source: a::2 (a::2)
    Destination: b::2 (b::2)
    Hop-by-Hop Option
        Next header: IPv6 (0x29)
        Length: 0 (8 bytes)
Internet Protocol Version 6
    0110 .... = Version: 6
        [0110 .... = This field makes the filter "ip.version == 6" possible: 6]
    .... 0000 0000 .... .... .... .... .... = Traffic class: 0x00000000
    .... .... .... 0000 0000 0000 0000 0000 = Flowlabel: 0x00000000
    Payload length: 88
    Next header: IPv6 hop-by-hop option (0x00)
    Hop limit: 64
    Source: a::2 (a::2)
    Destination: ff05::10 (ff05::10)
    Hop-by-Hop Option
        Next header: IPv6 routing (0x2b)
        Length: 0 (8 bytes)
    Routing Header, Type : Unkown (17)
        Next header: ICMPv6 (0x3a)
        Length: 7 (64 bytes)
        Type: Unknown (17)
        Left Segments: 0
Internet Control Message Protocol v6
    Type: 128 (Echo request)
    Code: 0
    Checksum: 0xdef6 [correct]
    ID: 0x1645
    Sequence: 0x0000
    Data (8 bytes)
0000  8d 7a 87 49 76 93 00 00                           .z.Iv...
        Data: 8D7A874976930000
```

Alors, on peut notamment constater une encapsulation des paquets dans les headers du premier paquet, on a une référence vers un second paquet ipv6, lequel contient enfin le header icmpv6. Le premier paquet a pour source a::2 (tbxnet1) et pour destination b::2 (tbxnet2). Le second paquet est un paquet de a::2 (tbxnet1) vers ff05::10 (adresse xcast).

7.2.4 Bilan

De cette manière, on peut expliquer le comportement de Xcast. Xcast envoie un paquet au premier destinataire ou bien au premier routeur. Une fois le paquet reçu, il est analysé directement par la couche IPv6, qui constate qu'il contient une adresse interne, un nouveau paquet est envoyé à l'adresse Xcast, ce qui permet à Xcast de traiter le paquet. Xcast n'a plus qu'à continuer le traitement en envoyant le paquet au prochain hôte.

On notera par ailleurs que ce fonctionnement n'est pas spécifique au tunneling. Pour faire parvenir le paquet au prochain routeur, Xcast doit absolument donner l'adresse de ce routeur en outer address, et pour faire parvenir le paquet à Xcast, il faut absolument donner l'adresse de Xcast en inner address. Ce comportement est alors simplement utilisé pour gérer le tunneling.

7.3 Application de test pour TBXcast

Afin de tester TBXcast il est nécessaire de vérifier plusieurs choses. Tout d'abord on doit être capable d'envoyer un paquet à un groupe d'utilisateurs. Ceci implique la possibilité de création de groupe. De plus, pour envoyer un paquet, il nous faut connaître l'arbre de routage que l'on va insérer dans celui-ci. L'arbre de routage va être calculé à partir de la topologie et la liste des destinataires (groupe d'utilisateurs).

Ces fonctionnalités appartiennent toutes à LibTBXcast, il est donc judicieux que notre application de test utilise cette bibliothèque.

Tout d'abord on a regardé l'application de test de Xcast, car c'est sur le modèle de ce protocole que l'on se base. Xcast utilise un programme nommé ping6x, un dérivé de ping6 (ping sur IPv6). Après l'étude du fonctionnement et du code de ping6x, nous avons décidé de ne pas utiliser ping6x comme base de notre application de test. Afin d'expliquer ce choix, on va expliquer le fonctionnement des applications ping (et en particulier ping6x).

7.3.1 Fonctionnement général de ping

Le but d'une application ping est d'envoyer un paquet à une destination (à plusieurs dans le cas de ping6x) et d'attendre une réponse venant de celle ci. De plus, on mesure le temps de propagation des paquets pour obtenir quelques caractéristiques de liaison entre les deux postes.

Ping est conçu de telle façon qu'il fonctionne sans que l'on ait besoin de lancer une application serveur sur le poste destinataire. En effet, il est très important que chaque machine réponde aux requêtes ping à tout moment. Il n'est donc pas possible d'utiliser les paquets UDP ou TCP car ceux ci remontent toujours vers une application pour un traitement ultérieur. On utilise donc un autre type de paquets - les messages ICMP.

On va en particulier utiliser un sous type de messages ICMP : les messages *ECHO*. La machine invoquant le ping va envoyer un message *ECHO REQUEST* au destinataire. Sur la réception de ce dernier, le poste en question va fabriquer un message *ECHO REPLY* et le renvoyer. Ce renvoi est géré entièrement par le système et ne peut pas être altéré. Toute machine est capable de répondre à des messages *ECHO REQUEST*.

Xcast introduit un autre concept. Étant un protocole multicast il envoie le message ICMP à plusieurs destinataires, cependant l'un d'entre eux est désigné comme «spécial» et seulement celui ci enverra une réponse du type *ECHO REPLY*.

A l'envoi, on utilise donc un paquet IPv6 dans lequel on encapsule le paquet Xcast. Au lieu d'utiliser un protocole de transport de type UDP (ou TCP) on y encapsule un message ICMP. Quant à la destination, elle renvoie un paquet IPv6 dans lequel elle encapsule un message ICMP.

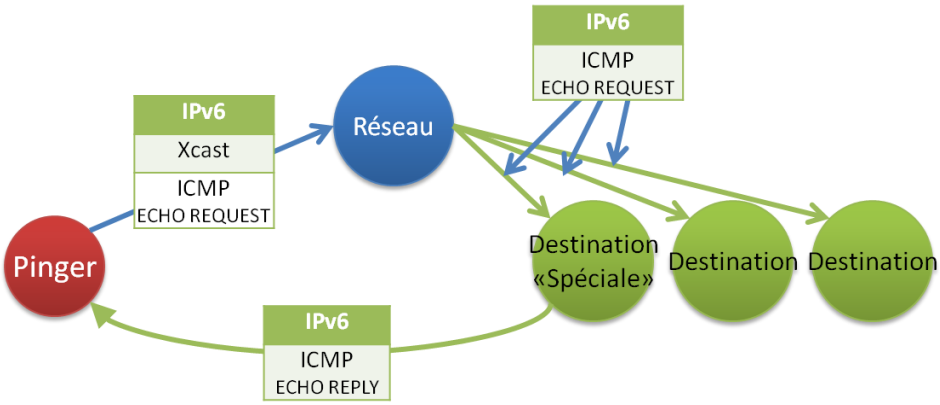


Figure 12 : Description de ping6x

7.3.2 Conséquences pour TBXcast

Ce fonctionnement bien pratique présente cependant un grand problème : la bibliothèque LibXcast ne traite pas des messages ICMP. Dans Xcast, le paquet contenant le message est entièrement construit au sein de ping6x. Étant donné les contraintes au niveau temporel et prenant en compte la complexité du code de ping6x il préférable que notre équipe se focalise sur la création de protocole et non sur le codage d'une application test.

Pour simplifier notre tâche et afin de pouvoir tester la bibliothèque LibTBXcast, nous avons décidé de concevoir une application qui envoie des messages UDP via LibTBXcast. On ne sera donc pas capables d'obtenir une réponse, mais on pourra détecter la réception des paquets grâce à tcpdump[10] et observer leur contenu avec WireShark[11]. Sur le long terme il est envisageable de créer une application serveur qui, à la réception d'un paquet, enverrait une réponse.

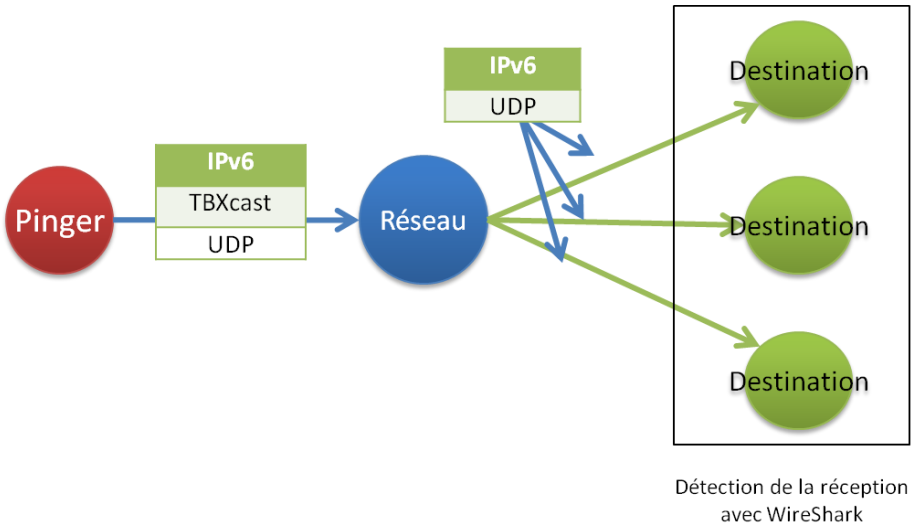


Figure 13 : Description de ping6tbx

On a décidé d'appeler notre application de test ping6tbx mais il est important de comprendre que le fonctionnement de celle ci n'a rien à voir avec le fonctionnement des applications ping classiques.

[10] Tcpdum est un sniffer de paquets. Il permet d'obtenir les détails du trafic visible depuis une interface réseau
[11] Wireshark est également un sniffer de paquets. Voir www.wireshark.org

7.4 Table des illustrations

7.5 Bibliographie

- RFC 2460 : Internet Protocol, Version6 (IPv6) Specification
- http://www.ietf.org/rfc/rfc2460.txt
- RFC 2473 : Generic Packet Tunneling in IPv6 Specification
- http://www.ietf.org/rfc/rfc2473.txt
- RFC 5058 : Explicit Multicast (Xcast) Concepts and Options
- http://tools.ietf.org/html/rfc5058
- Rapports de l'équipe TBXcast 2007-2008