TBXCAST

Un protocole de routage multicast explicite

Rapport de spécification fonctionnelle

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

Encadrants : Miklós MOLNÁR, Bernard COUSIN

Sommaire

1 Introduction

La spécification fonctionnelle de TBXcast a été abordée avec un angle bien particulier. En effet, plutôt que de décrire de façon « boîte noire » ses fonctionnalités, nous nous sommes basés sur l'existant - le protocole Xcast. La difficulté était donc de trouver un équilibre entre l'intime liaison de notre protocole avec Xcast, mais de néanmoins garder l'abstraction nécessaire à la description de ses fonctionnalités.

Etant donnée la nature à la fois logicielle et matérielle de notre projet, il est indispensable de posséder une plateforme de test et c'est pourquoi nous avons repris la salle d'expérimentation que le groupe de l'année dernière avait aménagé. Les modifications que nous y avons apportées ainsi que les démarches futures seront présentées dans la dernière partie.

Dans une première partie, nous décrirons le protocole Xcast dans son ensemble, puis nous détaillerons ses mécanismes de manière plus précise. C'est grâce à la bonne connaissance du code et du fonctionnement global de ce protocole que nous pourront faire la spécification de TBXcast, qui figure dans la deuxième partie.

2 Xcast6

2.1 Introduction

Comme nous l'avons vu dans la pré-étude, nous allons utiliser un protocole de routage existant : Xcast. Ce dernier nous servira de modèle pour développer et implémenter TBXcast. Dans cette optique, nous avons dû étudier le code de Xcast pour en comprendre son fonctionnement et mettre en valeur les points réellement utiles pour notre projet. Dans cette partie nous allons donc exposer le résultat de notre étude. Nous décrirons tout d'abord Xcast dans sa globalité et la façon dont il s'interface dans le noyau de NetBSD. Nous nous intéresserons ensuite aux fonctions de Xcast et de son API[1] (LibXcast) d'un point de vue externe, afin de définir le rôle de chaque fonction. Enfin, nous présenterons en détail quelques fonctions principales de Xcast. Cette dernière étude devrait nous aider à définir les zones du code à modifier pour implémenter TBXcast.

2.2 Vue d'ensemble

Rappelons tout d'abord le fonctionnement général de Xcast dans un environnement NetBSD avec ses avantages et ses inconvénients.
Xcast est un protocole de routage multicast explicite plat[2]. 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. Xcast présente également des inconvénients, comme un temps de traitement potentiellement important des paquets par les routeurs, et met en avant des problèmes liés à l'anonymat des destinataires et à la fragmentation du paquet IPv6.

Nous présentons dans la figure 1 une vue générale de l'environnement de Xcast. Cette représentation a évolué depuis la pré-étude, depuis que nous connaissons de manière plus détaillée le fonctionnement des différents modules.

Le schéma (Figure 1) représente une machine qui peut émettre, recevoir ou transmettre des données via le protocole Xcast. Nous avons déduit ce schéma à partir de nos études des différents modules qui le composent. Il n'est donc en aucun cas « officiel » et peut être amené à évoluer au fil de nos découvertes.

---

[1] API (Application Programming Interface) : fonctions mises à disposition de l'utilisateur.
[2] Voir le rapport de pré-étude pour plus de détails.

Commençons par le module de Xcast, qui se situe dans le noyau de NetBSD. La communication entre les deux est réalisée par l'intermédiaire d'une interface. Lors de la mise en place de Xcast, un pseudo-device[3] est créé. Ce dernier est lié à la carte réseau et permet une utilisation de l'interface par le module de Xcast.

Lors de la réception d'un paquet IPv6 de type Xcast, les données du paquet sont transmises depuis le pseudo-device à l'interface, grâce aux fonctions de NetBSD. L'interface envoie ensuite ces données au module de Xcast (typiquement, à la fonction principale xcast6_branch que l'on étudiera par la suite). Deux cas se présentent alors : si la machine doit réceptionner le paquet, le module de Xcast va transmettre les données par l'intermédiaire de fonctions NetBSD, jusqu'à l'application destinataire (à priori, on ne passe pas par l'interface). Si la machine doit réémettre le paquet, le module de Xcast fait les branchements nécessaires et réémet les paquets créés en les envoyant directement par une fonction de NetBSD (là non plus, on ne passe pas par l'interface pour envoyer les paquets).

Xcast fournit une API nommée LibXcast qui facilite les communications entre le module Xcast et le niveau applicatif. D'après notre étude et à notre grande surprise, LibXcast est totalement indépendant du noyau de Xcast. C'est à dire que lorsqu'une machine source veut envoyer un paquet Xcast en utilisant LibXcast, aucun appel des fonctions du module de Xcast n'est fait, comme on aurait pu s'y attendre. L'API crée elle-même le paquet et l'envoie au pseudo-device.

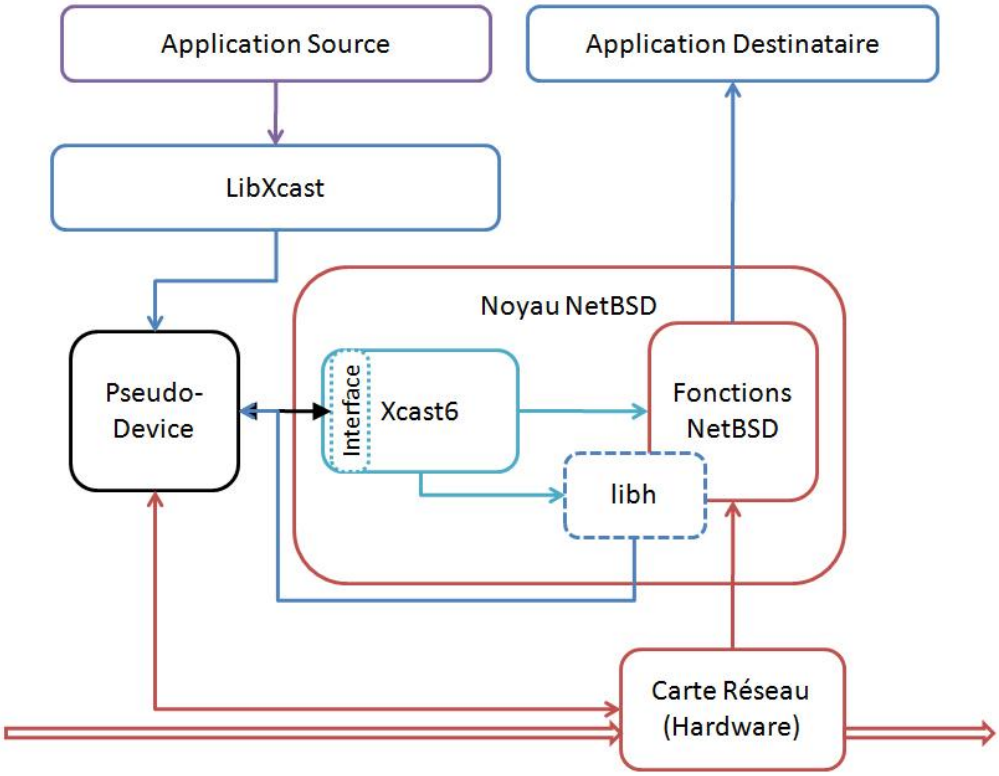


Figure 1: Vue générale du fonctionnement de Xcast

Nous avons également étudié une application test, Ping6x, qui permet d'envoyer un paquet Xcast à une destination afin de savoir si elle est accessible. Là aussi, à notre grande surprise, 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.

[3] Pseudo-device : Voir rapport de pré-étude (Interface du module Xcast)

Suite à cette confrontation des différents modules étudiés, nous constatons qu'aucun ne tire bénéfice des autres. Nous ne savons pas encore si cela est dû réellement à un choix de conception concret.

Dans les parties suivantes, nous allons voir de manière plus détaillée chacun de ces modules.

2.3 Fonctionnement Général

Dans cette partie, nous allons nous intéresser aux structures de données et aux fonctions de LibXcast, puis celles externes (interface) et internes au module de Xcast.

2.3.1 LibXcast

LibXcast représente l'API de Xcast. Comme on l'a remarqué dans la partie introductive, cette API est uniquement utilisée par l'application dans la machine source. Elle permet la gestion des groupes, des membres des groupes et l'envoi du message contenant ces deux ensembles.

Dans cette partie, la description des fonctions de cette API se fera d'un point de vue externe, c'est-à-dire que seules les valeurs de retour et les paramètres des fonctions seront présentés. On commencera par la description des fonctions du groupe, puis celle des membres et enfin de l'envoi de messages.

2.3.1.1 Fonction de groupe

2.3.1.1.1 La structure de données associée à un groupe

Un groupe est constitué de cinq attributs intégrés dans une structure « *xcast_grpentry* ». Cette structure est utilisée pour regrouper les différentes données qui définissent un groupe. Cette structure est chaînée car souvent, il n'y a pas assez de place dans une seule *grpentry* pour contenir la liste complète des destinataires de ce groupe. Cette structure est composée d'un groupid (numéro du *grpentry*), d'un pointeur vers un *grpentry* composé de la suite des destinataires, d'un attribut qui permet de savoir si le groupe est un sous-groupe ou non (par exemple, pour supprimer tout le groupe, on devra partir du premier groupe qui lui n'est pas un sous-groupe), du numéro du dernier groupe et enfin d'un *xcast_group* qui définit un groupe Xcast.

Cette structure (*xcast_group*) est constituée par l'identificateur unique de groupe, le type de famille (IPv6), le nombre des destinataires, le maximum de destinataires et enfin une union. Cette union est composée du cas IPv4 et du cas IPv6. Pour le cas IPv4, on ne construit pas réellement de groupe, mais on garde seulement l'adresse de la source (créateur du groupe). Pour l'IPv6, il contient plusieurs champs supplémentaires dont l'entête XCAST, le bitmap[4], la liste des adresses, le champ header IPv6 et UDP.

Ces structures sont stockées dans un tableau où les différentes entrées du tableau correspondent au numéro de la structure *grpentry* (ici, le groupid).

2.3.1.1.2 Les fonctions du groupe

Pour compléter cette structure, il existe différentes fonctions : celle de création, de suppression et de gestion du groupe.

Fonctions de création d'un groupe

Il existe quatre méthodes permettant de créer un groupe Xcast.

La première fonction est *XcastCreateGroup*. Elle prend en paramètres un entier *flags*, une structure *sockaddr* correspondant à l'adresse IP de la source sous le format NetBSD et un dernier paramètre représentant le

[4] Chaque bit du bitmap est associé à un destinataire et permet ainsi d'éviter la manipulation directe des adresses de destination (ce qui serait coûteux). Ainsi, lors d'un branchement, lorsqu'on veut enlever un destinataire, on positionne son bit associé à 0. On le laisse à 1 dans le cas contraire.

nombre maximum de membres qu'il peut contenir. Cette fonction permet de créer un groupe Xcast. Elle retourne l'id du groupe crée par la fonction Xcast6CreateGroup ou -1 en cas d'erreur.

La suivante est donc *Xcast6CreateGroup*. Elle prend les mêmes paramètres que la précédente et elle retourne l'id du groupe créé par la fonction *Xcast6CreateGroupSubr* en forçant la structure *sockaddr* au format IPv6. Ensuite *Xcast6CreateGroupSubr* prend un paramètre de plus que les précédentes : un entier représentant le socket[5]. Celle-ci remplit tous les champs de la structure *xcast_group* en fonction des flags passés en paramètre puis appelle la fonction *XcastAllocGroupEntry*.

La fonction *XcastAllocGroupEntry* va permettre d'allouer et de créer la structure *grpentry* associé au groupe. Elle prend en paramètre la structure *xcast_group* et un booléen permettant de savoir si le groupe créé est un sous groupe ou non.

La dernière fonction permettant la création d'un groupe Xcast est *Xcast6CreateSubgroup* qui créé un sous-groupe et le chaîne correctement en appelant la fonction *Xcast6CreateGroupSubr*.

On peut résumer cette présentation des fonctions de création d'un groupe à l'aide du graphique suivant qui décrit leur interrelation.

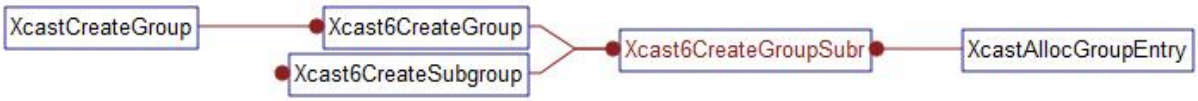


Figure 2 : Appel entre les différentes fonctions de création

Fonctions de suppression de groupe

Il existe deux fonctions permettant quant à elles de détruire un groupe, et qui se différencient par leurs paramètres. *XcastDeleteGroup* prend juste l'id du groupe en paramètre alors que *Xcast6DeleteSubgroup* prend aussi en paramètre l'entier correspondant au sous-groupe. Cette dernière supprime le sous-groupe dont l'identificateur est passé en paramètre à l'intérieur du chaînage du groupe groupid. La fonction supprime ce sous-groupe de la mémoire et du chaînage à l'aide de la fonction *XcastFreeGroupEntry*. De même, pour la première qui supprime le groupe ayant l'id passé en paramètre. *XcastDeleteGroup* renvoie 0 en cas de succès et -1 en cas d'échec (s'il ne trouve pas le sous-groupe dans le groupe passé en paramètre). Le graphique suivant montre les différents appels des fonctions de suppression.

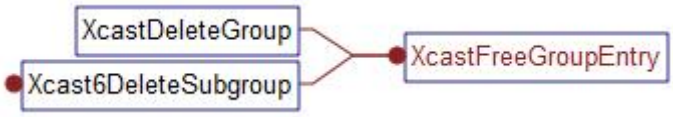


Figure 3 : Chemin entre les différentes fonctions de suppression

Xcast6DeleteSubgroup ne peut être appelée que si subgroupid est le dernier groupe du chaînage, sinon les sous-groupes suivants ne seront plus référencés.

Fonction « getter »

Il existe une fonction permettant de récupérer la structure d'un groupe en lui passant l'id de groupe recherché en paramètre. Elle se nomme *XcastGetGroupEntry*. Elle retourne NULL si le groupe n'est pas trouvé.

---

[5] Un socket est un point de communication entre deux ou plusieurs processus sur un réseau basé sur IP. C'est une interface entre le processus ou le thread d'une application et la partie TCP/IP du modèle OSI. (voir rapport de pré-étude) Un socket est identifié par un protocole (TCP, UDP ou raw IP), une adresse IP locale et un numéro de port local.

2.3.1.1.3 Les fonctions de gestion des membres

Structure de données associée à un membre

La structure d'un membre est composée d'un pointeur sur une structure *sockaddr*. Cette dernière sert à encapsuler l'adresse IP de destination et optionnellement le port UDP/TCP. L'autre composant de la classe est un *unsigned char* qui lui aussi est optionnel. Celui-ci sert au DiffServ Code Point[6].

Les fonctions des membres

Pour compléter cette structure de données, il existe quatre fonctions.

La première sert à ajouter un membre à un groupe: *XcastAddMember*. Cette fonction vérifie que le groupe existe bien, que les familles du groupe et du futur membre sont compatibles et que cette famille est IPv6. Si ces conditions sont remplies, elle effectue l'ajout du membre passé en paramètre dans le groupe passé également en paramètre par l'intermédiaire de la fonction *Xcast6AddMember*. Elle retourne un entier indiquant si l'ajout du membre s'est bien déroulé ou non.

La seconde, *XcastDeleteMember*, enlève le membre du groupe mais effectue les mêmes vérifications que la précédente avant d'appeler *Xcast6DeleteMember*. Elle retourne aussi un entier indiquant si le retrait s'est bien effectué ou non.

Les deux autres fonctions effectuent des opérations sur le bitmap. *XcastEnableMember* passe à 1 le bitmap correspondant au membre et au groupe passés en paramètre à l'aide de la fonction *Xcast6EnableMember*. *XcastDisableMember* le passe à 0 à l'aide de la fonction *Xcast6DisableMember*.

2.3.1.1.4 Les fonctions d'envoi de données

XcastSend vérifie que le groupe existe bien, que les familles du groupe et du membre sont compatibles et que cette famille est IPv6. Si ces conditions sont remplies, elle fait un appel à la fonction *Xcast6Send* qui va se charger d'envoyer effectivement le message notamment grâce à la primitive d'envoi *sendmsg*. Elle retourne un entier indiquant si l'envoi s'est bien passé ou non.

XcastSendMsg permet également l'envoi, mais prend en paramètre une structure *msghdr* contrairement à son homologue *XcastSend*. *msghdr* représente la structure d'un message envoyé à travers un socket. On n'a pas besoin de construire le message à envoyer comme on le faisait dans *XcastSend*. On appelle donc directement la primitive NetBSD *sendmsg* sans passer par une fonction intermédiaire.

2.3.2 Xcast

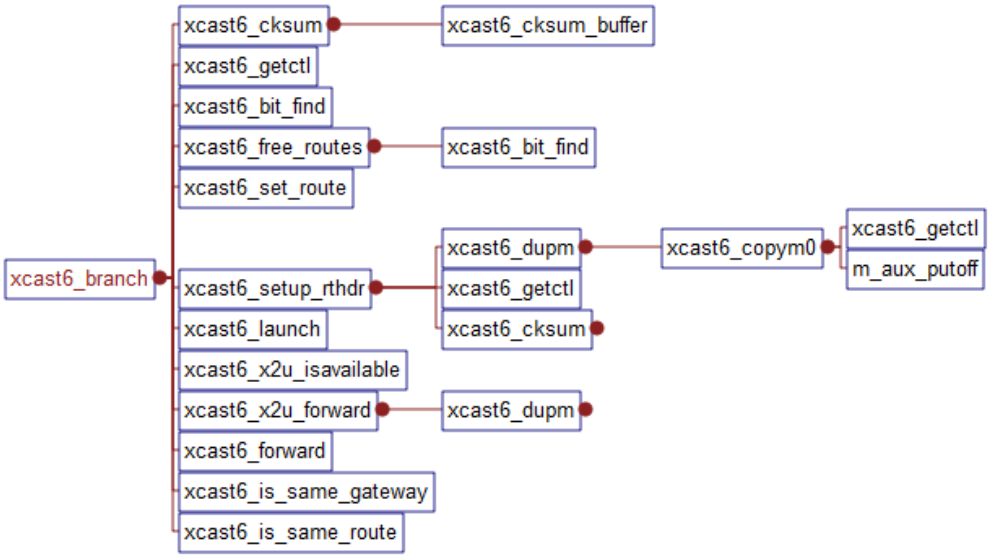
Le code source du protocole Xcast est situé dans deux fichiers distincts : l'interface qui s'occupe des entrées-sorties et les fonctions qui s'occupent du paquet Xcast. Tout d'abord, on a récupéré ces quatre fichiers (deux fichiers d'entête et deux fichiers sources) à partir de l'étude faite par le groupe de l'année passée. Dans ces fichiers, il a fallu alors décortiquer les fonctions pour comprendre leur fonctionnement et leur utilité. Cette étude nous a pris beaucoup de temps car :

- certaines fonctions de Xcast appellent des structures ou des fonctions de NetBSD qu'il a fallu à leur tour étudier
- de part sa nature, le code noyau est optimisé, et donc souvent difficile à lire
- la quantité de commentaires présents dans le code est très faible, et la documentation en ligne quasi-inexistante

L'année dernière, le groupe de projet TBXcast a utilisé un logiciel permettant de constituer les dépendances entre les fonctions et les structures contenues dans différents fichiers. Ce logiciel nous a donc permis de

[6] DSCP est un champ dans l'en-tête des paquets IP pour la classification des paquets

trouver les fichiers NetBSD qui étaient appelés par Xcast. De plus, il permet de voir la dépendance entre les fonctions d'un même fichier. On a donc obtenu le graphe des dépendances pour la fonction *xcast_branch* (voir graphique 1) qui prend une place importante dans le code Xcast.



Graphique 1 : Dépendances de la fonction xcast_branch par rapport aux autres fonctions Xcast

D'après le graphique, on peut vérifier que cette fonction est la fonction principale dans Xcast car elle en appelle la plupart des autres fonctions. On va décrire ici la plupart de ces fonctions appelées.

2.3.2.1 Les fonctions Xcast

La fonction *xcast6_branch* prend en paramètre une structure mbuf (structure NetBSD contenant les données à transmettre sur le réseau), une structure contenant l'entête du paquet Xcast. Elle renvoie -1 à la fin mais peut sortir avant en renvoyant par contre des constantes indiquant un type d'erreur prédéfini (*EINVAL*, *ENOBUFS*, *ENETUNREACH*). On peut dire que c'est la fonction principale de Xcast. On la verra donc plus en détails dans la prochaine partie.

La fonction *xcast6_cksum* calcule le checksum[7] du paquet Xcast (uniquement sur l'entête apparemment). Elle appelle la fonction *xcast6_cksum_buffer* pour ce calcul et le retourne par la suite.

La fonction *xcast6_bit_find* prend une structure contenant un tableau d'entiers non signés sur 8 bits chacun. La recherche est effectuée sur ce tableau pour les entrées dont le numéro est compris entre les deux entiers passés en paramètre. On retourne la position du premier bit à 1 trouvé dans ce tableau. Si on en trouve aucun, on retourne alors -1.

La fonction *xcast6_free_routes* est appelée lorsque des adresses du paquet Xcast sont invalides (par exemple, adresse multicast, locale ou loopback[8]). La fonction supprime toutes les destinations définies par le bitmap (c'est à dire, pour chaque bit à 1 du bitmap, on supprime la route correspondante). La suppression de la route est faite par la fonction *rtfree*. On met également à 0 le bit correspondant dans le bitmap.

[7] Le checksum, ou somme de contrôle, est produit par un algorithme qui est appliqué sur les données à transmettre. Grâce à l'ajout du checksum aux données à transmettre dans le paquet (redondances), et après la réception du paquet, le récepteur peut détecter une corruption du message.

[8] Une adresse loopback (127.0.0.1) est une adresse permettant à une machine de s'envoyer à elle-même des messages.

La fonction *xcast6_getctl* retourne un pointeur sur une structure *xcast6_ctl_t* contenant les options du paquet Xcast (nombre de destinations, port …) à partir d'un des champs de la structure mbuf passée en paramètre.

La fonction *xcast6_set_route* remplit la structure *xcast6_routetab_t* à partir de la structure *in6_addr* (représentant une adresse IPv6) nécessaire au routage d'un paquet Xcast6.

La fonction *xcast6_setup_rthdrp* prend un entête Xcast en paramètre (*xcast6_rthdr_t* ox6*) et rend ce header dans un mbuf m. Ce n'est pas une simple recopie, on recalcule également les champs optionnels Xcast. Si le paramètre *need_copy* est à 0, on fait une simple affectation de l'adresse de m dans un pointeur sur une structure *mbuf*. On recopie également l'adresse de ox6 dans un pointeur vers une structure *xcast6_rthdr_t*. Si le paramètre *need_copy* est à 1, on fait une copie en profondeur du *mbuf* passé en paramètre (*xcast6_dupm*). Elle retourne le *mbuf* créé.

La fonction *xcast6_forward* envoie un paquet xcast6. On trouve principalement dans cette méthode, la gestion du tunneling. La fonction *xcast6_x2u_forward* transforme un paquet Xcast6 en paquet unicast. Elles retournent 0 si tout s'est bien passé, -1 sinon.

La fonction *xcast6_launch* passe le paquet Xcast6 à l'application. Cette fonction sera détaillée dans la prochaine partie.

Les fonctions *xcast6_dupm* et *xcast6_copym0* sont juste des fonctions de recopie. *Xcast6_copym0* recopie par rapport à un paramètre : 1 pour une recopie en profondeur, 0 pour une simple recopie des pointeurs. *Xcast6_dupm* rappelle cette dernière avec le paramètre 1. Pour ce qui est des structures spécifiques à Xcast, les plupart ont été décrites dans les fonctions de Xcast précédemment énoncées.

Les fonctions présentes sur le graphique, et non décrites ici, nous retournent juste des booléens servant à donner des indications pour les options Xcast supplémentaires.

2.4 Au cœur du code

Après avoir vu les fonctions externes, nous allons entrer dans le cœur du module Xcast. Dans cette partie, nous identifierons les fonctions critiques de Xcast et verrons leur fonctionnement en profondeur. Cette étude nous permettra d'en comprendre les rouages et de pouvoir identifier précisément les parties de code à modifier pour l'implémentation de TBXcast.

2.4.1 xcast6_branch

Comme nous l'avons vu précédemment, xcast6_branch est la fonction principale de Xcast. Nous allons voir quelles sont les différentes étapes effectuées sur un paquet IPv6 lors de sa réception et en vue de son traitement (réémission ou passage à l'application destinataire). En complément de cette description fonctionnelle, nous fournirons le diagramme d'activité des grandes étapes de *xcast6_branch* en annexe.

Le traitement du paquet commence tout d'abord par une série de vérifications. On contrôle tout d'abord que le paquet IPv6 est bien de type Xcast et que sa version correspond à la version Xcast6. On récupère ensuite principalement, le bitmap et la liste des adresses des destinataires contenues dans le paquet. On procède également au calcul du checksum sur l'entête du paquet pour vérifier si le paquet est correct.

La suite concerne la gestion de l'anonymat, dans le cas où cette option est activée dans le paquet Xcast. Normalement, les adresses des destinataires sont toujours conservées dans le paquet, seul le bitmap est modifié lors d'un branchement. Par contre, si l'anonymat est activé, les adresses ayant un bit associé à 0 seront masquées pour la suite du traitement, et remplacées par des 0.

La prochaine étape concerne la vérification de chaque destination. Une boucle est appliquée sur le bitmap tant qu'un bit à 1 est trouvé. Une vérification est alors faite sur la validité de chaque adresse de destination.

Si une adresse multicast, une adresse locale ou une adresse loopback est trouvée, le paquet est détruit. La réémission échoue. En effet, la liste des destinations ne doit contenir que des adresses unicast. Par ailleurs, pour chaque destination, un test est également réalisé pour savoir si la route est valide. On regarde dans la table de routage si les destinations sont accessibles. Dans le cas contraire, la destination est supprimée (le bit associé du bitmap est positionné à 0).

Nous arrivons enfin au traitement du paquet en lui-même, c'est à dire aux branchements à effectuer et à la transmission des copies du paquet reçu. Afin de faciliter les explications, rappelons brièvement le principe. Lorsqu'un paquet arrive sur un routeur R, la liste des destinations va devoir être analysée. Supposons qu'une partie E1 des destinations doive passer par un routeur R1, et qu'une autre partie E2 doive passer par un routeur R2 pour arriver à bon port. R1 et R2 sont des next-hop[9] pour les paquets arrivés en R. Deux copies du paquet vont devoir alors être créés à partir du paquet d'origine: un qui contiendra les destinations E1 et ayant pour next-hop R1 et un autre qui contiendra les destinations E2 et ayant pour next-hop R2. Rappelons également qu'en réalité, toutes les destinations sont présentes dans chaque paquet. Seul le bitmap détermine les destinations à prendre en compte dans le paquet.

L'objectif est donc ici de grouper les destinataires qui ont une prochaine destination commune et de construire ainsi les copies du paquet à transmettre.

Pour cela, dans le code Xcast, une boucle A est appliquée tant qu'il y a des adresses de destination à traiter. C'est-à-dire qu'on s'arrête lorsque chaque destination est attribuée à un paquet. Il peut très bien y avoir un seul passage dans la boucle si toutes les destinations ont le même next-hop. Développons alors ce qu'il se passe lors d'une itération.

On sélectionne tout d'abord le premier bit à 1, d'index i, que l'on trouve dans le bitmap *bmrest* du paquet d'origine. En vue de créer un nouveau paquet Xcast, un nouveau bitmap *bm* est créé. Le bit i est alors mis dans *bm* et enlevé de *bmrest*. Plusieurs cas sont alors envisageables. Dans chacun de ces cas, la méthode *xcast6_setup_rthdr* est appelée pour construire l'entête du nouveau paquet Xcast à partir de l'entête du paquet d'origine et du nouveau bitmap *bm*.

Cas d'un loopback :

Le prochain next-hop est la machine locale. Cela signifie que le paquet est déjà arrivé à destination. La fonction *xcast6_launch* est alors appelée pour passer les données du paquet à l'application. Il est bon de noter également que X2U[10] n'est pas utilisé dans ce cas-là car la machine sait interpréter les paquets Xcast.

Cas où il ne reste qu'un seul destinataire à traiter dans le paquet Xcast d'origine :

Il ne reste plus qu'un seul bit à 1 dans le bitmap d'origine *bmrest*. Si X2U est disponible, le paquet Xcast est transformé en paquet unicast puis envoyé en appelant la méthode *xcast6_x2u_forward*. Si X2U n'est pas disponible, le paquet est envoyé par la méthode *xcast6_forward*.

Dans tous les autres cas :

On va parcourir une nouvelle fois le bitmap *bmrest* et déterminer quelles sont les destinations d'index j qui ont le même next-hop que i. Pour chaque destination trouvée, on met à jour le bitmap *bmrest* (le bit associé est mis à 0) et le nouveau bitmap *bm* (le bit associé est mis à 1). Suite à ce parcours, nous avons tous les éléments nécessaires pour réaliser un branchement (l'entête a été calculé et nous avons le bitmap contenant les destinations ayant leur prochain next-hop en commun). Si le bitmap *bm* ne contient qu'une

[9] Le next-hop est la prochaine destination d'un paquet. Par exemple, si un paquet Xcast arrive sur un routeur R1 et s'il doit passer par un routeur R2 pour être acheminé, le routeur R2 est le next-hop.

[10] Xcast To Unicast (X2U) permet de transformer un paquet Xcast en paquet unicast lorsqu'il ne reste plus qu'une seule destination dans le paquet Xcast.

seule destination après ce parcours, le paquet est envoyé par *xcast6_x2u_forward*. Sinon, il est envoyé par *xcast6_forward*.

Cette boucle A est répétée tant qu'il reste des destinations non traitées, c'est à dire, tant qu'il reste des bits à 1 dans le bitmap d'origine *bmrest*.

2.4.2 xcast6_launch

Cette méthode est appelée pour transmettre le paquet Xcast à l'application. D'après ce que nous avons réussi à identifier, une boucle est effectuée sur les entêtes du paquet IPv6. Le paquet est ainsi parcouru grâce au chaînage des entêtes. Pour chaque itération, on envoie les données aux couches hautes du système grâce à la fonction *pr_input*. Lorsque tous les entêtes ont été traités, *pr_input* renvoie une valeur spéciale pour signaler que la fin des entêtes du paquet IPv6 a été atteinte.

2.4.3 xcast6_forward

Cette méthode est appelée par *xcast6_branch* pour envoyer un paquet Xcast. On peut distinguer deux parties dans cette fonction. Tout d'abord, le TTL[11] du paquet est géré. S'il est différent de 0, il est simplement décrémenté et mis à jour dans l'entête du paquet. Sinon, le paquet est détruit et un paquet ICMPv6 est renvoyé.

Dans un second temps (et si le TTL n'est pas nul), on va s'occuper de l'encapsulation du paquet pour mettre en œuvre le tunneling[12]. Comme nous l'avons vu dans la pré-étude, cela consiste en la création d'un entête optionnel Hop-By-Hop permettant aux routeurs non Xcast de traiter le paquet en unicast.

Enfin, le paquet est envoyé par la méthode *nd6_output* (méthode que nous n'avons pas encore pu identifier). Si des erreurs interviennent, elles sont comptabilisées pour la création de statistique sur les envois non réussis.

2.4.4 xcast6_x2u_forward

Cette méthode est appelée par *xcast6_branch* lorsqu'il n'y a qu'un seul destinataire à gérer dans le paquet Xcast reçu. Le but de cette fonction est de transformer un paquet Xcast en paquet unicast, puis de l'envoyer. Pour cela, on va adapter l'entête du paquet Xcast en entête IPv6 classique.
Comme pour *xcast6_forward* et de manière analogue, on va gérer le TTL. Ensuite, les différents champs de l'entête du paquet IPv6 vont être renseignés à partir de l'entête du paquet Xcast et le checksum sera calculé sur l'entête créé. La suite est analogue à *xcast6_forward*: on envoie le paquet unicast avec *nd6_output* et on récupère les erreurs en vue d'un traitement statistique.

2.5 Conclusion

Grâce à cette étude détaillée de Xcast, nous avons établi une base de connaissances pour notre projet TBXcast. Nous utiliserons les fonctions décrites en modifiant certaines parties pour implémenter les fonctionnalités de notre protocole de routage.

Intéressons-nous maintenant aux fonctionnalités de TBXcast.

---

[11] Le Time To Live (TTL) correspond au nombre de sauts autorisés pour le paquet. Par exemple, si le TTL vaut 3, le paquet ne sera autorisé qu'à traverser 3 routeurs. A chaque next-hop, le TTL est décrémenté. S'il arrive à 0, le paquet est détruit et un paquet ICMPv6 est renvoyé vers la source du paquet détruit.

[12] Le tunneling est mis en place lorsqu'un réseau n'est pas homogène, en l'occurrence, si tous les routeurs ne supportent pas Xcast. Cette solution permet de traverser les routeurs non-Xcast en tant que simple paquet unicast. Voir le rapport de pré-étude pour une explication plus détaillée.

3 TBXcast

3.1 Description générale

Le protocole TBXcast se divise en trois grands blocs fonctionnels :

- L'émission par la machine source
- Le traitement par les routeurs
- La réception par le destinataire

Etant fortement basé sur le code de Xcast, il reprendra un schéma semblable :

- Une API qui permet la communication entre le niveau applicatif et le noyau du système (LibTBX)
- Un code pour le module TBXcast de noyau à l'intention des routeurs intermédiaires

Malgré tout, les ajouts concernant les traitements algorithmiques sur l'arbre sont une caractéristique essentielle qui est propre à TBXcast. Nous allons les présenter brièvement dans cette introduction et les voir en détail dans les parties suivantes.

3.1.1 Création de l'arbre

En récupérant la topologie du réseau, la source sera capable de créer un arbre représentant les routes pour acheminer les données vers les destinataires.

3.1.2 Segmentation de l'arbre

Lorsque l'arbre est trop volumineux pour que l'entête puisse le contenir, il devra être segmenté en plusieurs arbres dont la structure sera fonction de la liste des destinataires associés à cet arbre.

3.1.3 Analyse par un routeur

Lorsqu'un routeur lit l'arbre contenu dans l'entête du paquet TBXcast, il doit être capable de le comprendre et de se situer dans celui-ci. Ainsi, il peut élaguer les parties ne le concernant pas et envoyer au routeur suivant un arbre réduit.

3.2 Description fonctionnelle

Dans cette partie, nous allons voir plus en détail les différentes fonctionnalités de notre protocole. Pour cela nous allons décomposer ces fonctionnalités en deux groupes : les fonctionnalités propres à la source du message et les fonctionnalités des routeurs intermédiaires et des destinataires.

3.2.1 Fonctionnalités de l'émetteur

Pour l'émetteur, nous allons nous appuyer essentiellement sur les fonctionnalités de l'API du module de Xcast, LibXcast, car elle offre les fonctionnalités de l'émetteur d'un message Xcast. Nous allons donc reprendre les principales fonctions de LibXcast et nous allons ajouter de nouvelles fonctionnalités propres à notre protocole : gestion de la topologie, codage de l'arbre, calcul de l'arbre, fragmentation du paquet et segmentation de l'arbre.

3.2.1.1 Les fonctionnalités de LibXcast

LibXcast apporte quatre fonctionnalités principales permettant à la source de gérer les groupes Xcast et l'envoi des messages : la gestion de groupe, la gestion des membres, la gestion du socket et l'envoi de messages.

Concrètement, nous n'aurons pas besoin de modifier le fonctionnement de base de ces fonctionnalités car les deux protocoles reposent sur les mêmes principes de gestion de groupe : découpage en sous-groupes, nombre maximum de membres dans un sous-groupe pour éviter la surcharge de l'entête, création d'un nouveau sous-groupe lorsque l'on dépasse le nombre maximum de membre. La source tient à jour ces informations et peut ajouter ou supprimer des membres à la demande de ceux-ci.

De même, la gestion des sockets et l'envoi des messages se feront de la même façon que pour Xcast.

Les changements que nous allons devoir apporter à cette API ne vont être que des ajouts de nouvelles fonctionnalités de TBXcast. Nous devrons malgré tout modifier la gestion des membres et celle des groupes pour rajouter la gestion de l'arbre.

3.2.1.2 Gestion de la topologie

Pour calculer l'arbre de routage, la source doit connaitre la topologie du réseau, c'est-à-dire les liens entre les routeurs et la façon d'atteindre les destinataires. Pour cela, la machine source stockera les adresses IP de tous les routeurs concernés ainsi que les liens entre ces routeurs et différentes informations concernant ces liens (par exemple le délai, le débit, le taux de perte) qui seront utiles par la suite.

La source doit maintenir à jour ces informations car un changement dans la topologie peut modifier l'arbre de routage (lorsqu'un routeur est déconnecté ou si une des valeurs stockées change par exemple). Si l'arbre de routage n'est pas mis à jour certains destinataires pourraient ne pas être atteints.

Dans les premières versions de notre protocole, ces informations seront des données fixes et l'arbre de routage sera calculé en conséquence. Puis nous mettrons en place la mise à jour automatique de cette topologie par le protocole OSPF qui est habituellement utilisé dans les protocoles de routage unicast.

Ces calculs seront effectués en dehors des fonctions de l'API mais seront ensuite utilisés par celle-ci.

3.2.1.3 Codage de l'arbre

Pour que les routeurs puissent faire circuler le paquet TBXcast, il faut qu'ils aient accès à l'arbre de routage. Pour cela, il faut stocker cet arbre dans l'entête du paquet. Dans le protocole Xcast, l'entête contient la liste des destinataires. Dans notre projet, on mettra l'arbre de routage à la place de cette liste, il faudra changer la structure d'un groupe et remplacer l'entête Xcast par notre entête TBXcast contenant l'arbre.

L'arbre de routage représente les connections entres les différentes machines. La racine de cet arbre est la source du message et les feuilles de l'arbre, les destinataires. Les nœuds de l'arbre représentent les routeurs intermédiaires de branchement qui peuvent aussi être des destinataires.

Il faudra coder cet arbre de façon à ce qu'il prenne le moins de place possible dans l'entête afin d'éviter une surcharge. L'arbre sera représenté par un tableau contenant les adresses des nœuds de l'arbre. Chaque case du tableau contiendra également la position du nœud père dans le tableau. Cela permet un parcours aisé de l'arbre et également de réduire la taille de l'entête (à chaque nœud ne sont associées que deux valeurs). On devra aussi modifier le bitmap contenu dans l'entête pour qu'il prenne en compte les modifications du tableau des adresses des nœuds de l'arbre.

Toujours dans l'idée de limiter la surcharge de l'entête, on ne codera que les nœuds de l'arbre qui suffisent à faire circuler le paquet, c'est-à-dire les nœuds destinataires, les nœuds de branchement et certains nœuds spécifiques par lesquels on voudrait faire passer le paquet (voir partie suivante).

Concrètement, le codage de l'arbre sera effectué lors de la création du groupe et à chaque fois qu'un calcul de l'arbre sera effectué (également dans la partie suivante).

3.2.1.4 Calcul de l'arbre

Pour la mise en place de notre protocole, il faut calculer l'arbre de routage qui permet de joindre la source aux destinataires. Après avoir récupéré les informations concernant la topologie du réseau, nous pouvons former un graphe contenant l'ensemble des destinataires, la source et plusieurs autres routeurs. L'arbre que nous devons calculer est un arbre couvrant qui permet de relier la source aux destinataires.

Afin d'optimiser le transport des paquets, cet arbre couvrant qui relie les nœuds spécifiques doit être de coût minimal. Ce problème est trop complexe pour être résolu, nous allons donc utiliser des algorithmes existants qui se rapprochent au mieux de la solution. Nous pourrons obtenir un arbre ayant un coût faible et qui sera suffisant pour respecter les exigences de l'utilisateur en terme de qualité de service.

Dans un premier temps, nous nous baserons sur les chemins les plus courts (en nombre de routeurs traversés) pour calculer cet arbre. Ensuite nous essaierons d'optimiser cet arbre en fonction de la QoS[13]. Nous nous baserons sur des valeurs comme le taux de perte de données, le délai de transport ou le coût de transport et nous calculerons l'arbre minimisant le mieux possible l'une de ces valeurs. C'est dans ce cadre que l'on aura besoin de garder certains nœuds spécifiques dans l'arbre, si on veut forcer le paquet à passer par un chemin spécifique. Les valeurs de la QoS seront récupérées au moment de la gestion de la topologie. Nous essaierons de construire un algorithme flexible qui permettra à l'utilisateur de choisir plusieurs paramètres qu'il veut minimiser selon son application (certaines applications privilégient la rapidité alors que d'autres préfèrent ne perdre aucune donnée).

Par rapport à l'application existante, il faudra rajouter le calcul de l'arbre à plusieurs reprises : lors de la création d'un groupe, lors de l'ajout ou de la suppression d'un membre ainsi qu'à chaque fois que la topologie est amenée à changer (déconnection de routeurs, modifications des valeurs de la QoS etc.).

3.2.1.5 Fragmentation des paquets

Le problème de la fragmentation des paquets se pose lorsque l'on doit envoyer un message assez long. Un paquet IP a une taille limitée. Ce paquet est rempli par l'entête du message et par le contenu du message en lui-même. Si on enlève, à la taille du paquet, la taille de l'entête, il reste la taille maximum du message : la MTU (maximum transmission unit).

Si la taille du message que l'on veut envoyer dépasse la MTU, il faut alors fragmenter le message en plusieurs paquets IP.

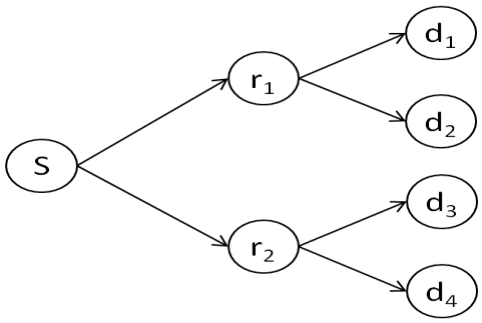
Ce principe est déjà implémenté avec le protocole IP, il va falloir cependant modifier la fonction de calcul de la taille de l'entête car celle-ci dépend de la taille de l'arbre qui y est stocké.

3.2.1.6 Segmentation de l'arbre

Malgré nos efforts pour réduire la taille de l'entête au maximum, il est inévitable que cette taille devienne trop importante lorsque le groupe contiendra beaucoup de membres. Ce problème a déjà été traité dans le protocole Xcast avec la création de sous-groupes reliés au premier groupe principal. Cependant, dans notre version, nous ne pouvons pas nous contenter de scinder les groupes arbitrairement. En effet notre protocole se base sur le codage de l'arbre de routage et si on effectue une mauvaise répartition des membres du groupe, on devra recoder plusieurs fois les mêmes nœuds de branchement dans les différents sous-groupes.

[13] La QoS (Quality of Service) définit la prise en compte supplémentaire de paramètres spécifiques aux communications réseaux. On peut citer le délai (temps de transmission), la gigue (variation du délai), le taux d'erreur etc. Ces données permettent d'optimiser les ressources d'un réseau en offrant aux utilisateurs des caractéristiques différentes pour chaque application.

Prenons un exemple simple (S représente la source, les nœuds r_i les routeurs de branchement et d_i les destinataires :



Graphique 2 : Exemple de segmentation de l'arbre

On peut voir que si on met d_1 et d_3 dans un sous-groupe et d_2 et d_4 dans un autre sous-groupe il faudra alors coder r_1 et r_2 dans les deux sous-arbres qui en découlent : ce n'est pas satisfaisant.

Le but de la segmentation est de faire en sorte qu'il y ait le moins possible de nœuds qui se répètent dans les différents sous-arbres. Ici, on créera un sous-groupe avec d_1 et d_2 et un autre avec d_3 et d_4.

3.2.1.7 Conclusion : diagramme d'activité

Nous avons donc vu les différentes fonctionnalités qui seront disponibles pour l'émetteur d'un message TBXcast. Elles reprennent principalement les fonctionnalités de Xcast sur la gestion des groupes et des membres auxquelles on ajoute la gestion de l'arbre de routage. Cette gestion de l'arbre représentera donc une partie importante dans notre projet.

Nous pouvons résumer toutes ces fonctionnalités sur le diagramme d'activité suivant, qui montre les relations entre toutes les fonctionnalités de l'émetteur.

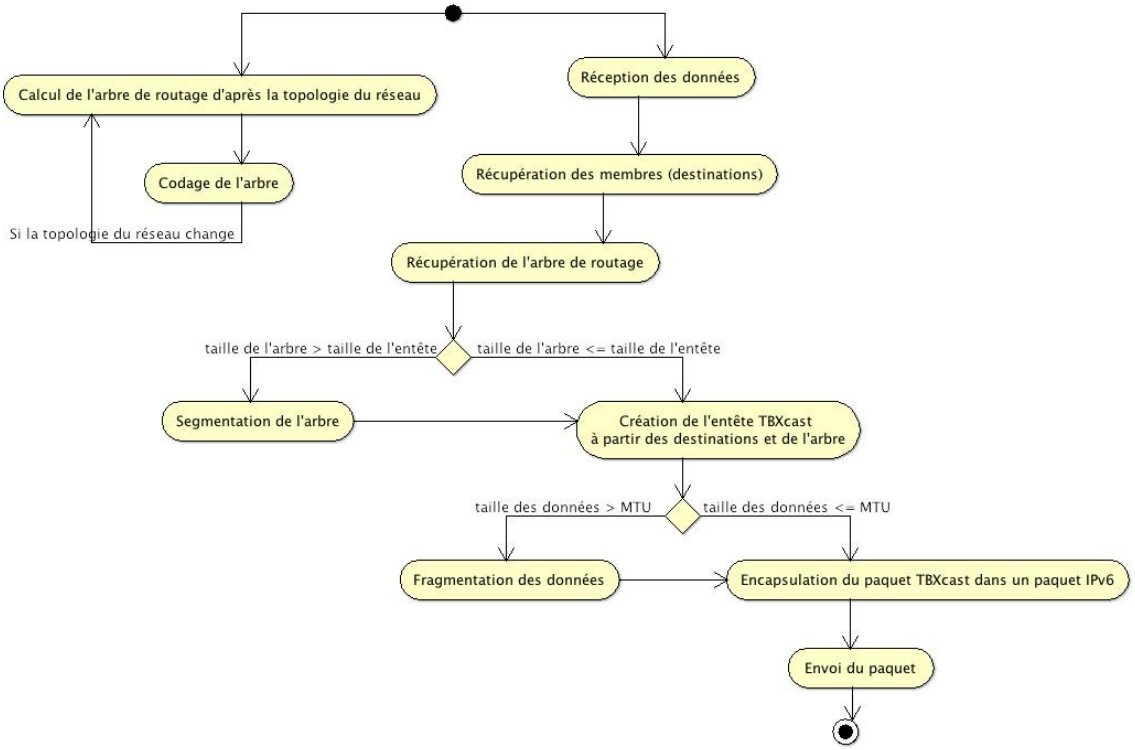


Diagramme 1 : Diagramme d'activité de l'émetteur

Après avoir vu comment fonctionne l'émetteur, voyons maintenant le fonctionnement des routeurs intermédiaires et des destinataires des paquets.

3.2.2 Les fonctionnalités des routeurs intermédiaires et des destinations sous TBXcast

Dans cette partie, nous allons traiter les fonctionnalités des routeurs intermédiaires et des machines destinations. Ceux-ci sont gérés principalement par Xcast6. Nous allons voir que même si la gestion de l'entête se fait par la même fonction (*Xcast_branch*) pour les routeurs intermédiaires et les récepteurs, les données qui en résultent ne sont pas les mêmes.

3.2.2.1 Analyse des entêtes Xcast/TBXcast

Afin d'optimiser le temps de traitement et de faciliter les calculs des routeurs Xcast, le protocole TBXcast va pouvoir fonctionner d'une autre manière, plus efficace. Tandis que Xcast manipule une liste de destinataires encapsulée dans l'entête du paquet à envoyer, TBXcast va manipuler les mêmes données légèrement étendues et les présenter de manière différente. Les destinataires sont les feuilles d'un arbre de routage déjà calculé et codé par la source. Cet arbre peut représenter des nœuds de branchement ou encore des nœuds spécifiques se trouvant entre la source et les destinataires en question, comme nous venons de le voir.

3.2.2.2 Les routeurs intermédiaires

Le routeur va recevoir des paquets de données TBXcast, contenant chacun une ou plusieurs destinations. Son rôle est de traiter ce paquet de données pour savoir comment l'acheminer pour qu'il arrive à ses destinataires. Les routeurs Xcast doivent faire de nombreux calculs sur la liste des destinataires encapsulée dans l'entête Xcast. De plus, ils doivent aussi rechercher les chemins vers les différents routeurs pour acheminer les paquets vers les destinataires. L'idée de TBXcast consiste principalement à faciliter la tâche des routeurs en leur donnant un arbre dans lequel se trouvent les prochains nœuds auxquels il faut adresser le paquet Xcast. Cet arbre est généré par la source comme on l'a vu précédemment. Celle-ci gérant aussi les changements de topologie du réseau, les routeurs intermédiaires n'auront donc aucun calcul à faire au niveau de la structure de l'arbre. Le seul calcul que devront effectuer ces routeurs TBXcast sera le partage des paquets vers les différentes destinations.

Dans un premier temps, le rôle du routeur va donc être d'extraire l'arbre de routage de l'entête des paquets. Le routeur va tout d'abord regarder s'il appartient aux destinations (dans le cas où le routeur serait une machine cible). Si c'est le cas, il va alors effectuer deux traitements distincts : l'acheminement du paquet et le traitement pour passer les données du paquet à l'application (on approfondira ce traitement plus loin dans ce rapport). Si ce n'est pas le cas, il effectuera seulement l'acheminement du paquet.

Cet acheminement s'effectue en plusieurs étapes. Le routeur va alors récupérer les différents fils de la racine de l'arbre courant et élaguer celui-ci le nombre de fois qu'il a de fils. Chaque sous arbre ainsi créé aura donc comme racine un des fils de l'arbre courant. Le routeur va alors créer de nouveaux entêtes pour chaque sous-arbre et dupliquer les données contenues dans le paquet autant de fois qu'il y a d'entêtes. Les paquets nouvellement créés vont être alors envoyés vers la racine de leur arbre respectif dans un paquet TBXcast. Il y a une exception à la création du paquet TBXcast : si le nombre de destinataires du nouvel arbre est égal à 1 alors le routeur intermédiaire créera un paquet unicast IPv6. Cette exception étant aussi proposée par Xcast, il ne sera peut-être pas nécessaire de la modifier. Ce traitement sera effectué par tous les routeurs intermédiaires.

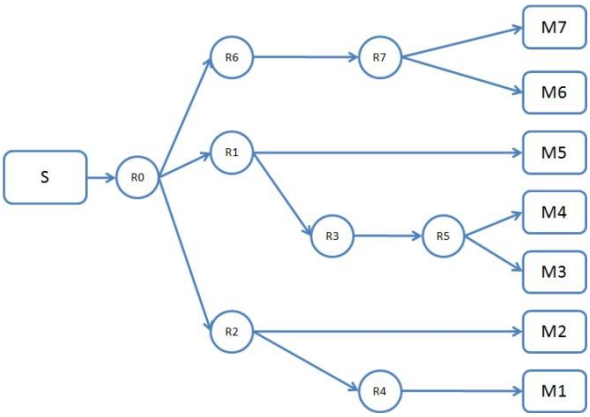


Figure 4 : Exemple d'arbre de routage pour S

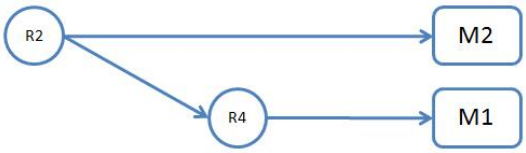


Figure 5 : arbre de routage pour R2

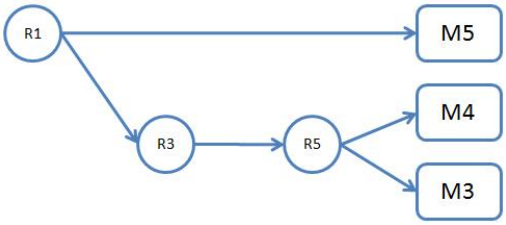


Figure 6 : arbre de routage pour R1

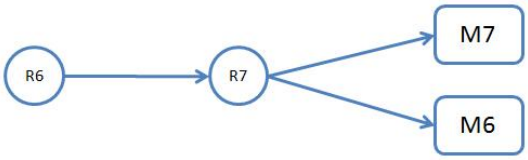


Figure 7 : arbre de routage pour R6

La figure 4 est un exemple d'arbre de routage que l'on peut rencontrer avec le protocole TBXcast. Les Ri représentent des routeurs TBXcast et les Mj des destinataires. Les trois autres figures l'accompagnant montrent la segmentation de l'arbre. Par exemple sur la figure 6, on remarque que le routeur R1 doit segmenter l'arbre en deux et envoyer un paquet vers M5 (destinataire) en unicast puis vers R3 en TBXcast, et ainsi de suite pour le reste de l'arbre.

3.2.2.3 Les machines destination

A ce niveau, la machine va pouvoir récupérer les données reçues et les envoyer vers les couches supérieures. Elle va tout d'abord décoder l'entête et récupérer ces données. Grâce aux informations contenues dans l'entête, la machine va pouvoir savoir si le paquet reçu fait partie d'un ensemble de fragments du paquet et en déduire combien il en reste à recevoir pour récupérer toutes les données. Lorsqu'il a tout reçu, il peut alors rassembler les différents fragments de données et les transmettre à l'application.

3.2.2.4 Conclusion

Le rôle d'un routeur intermédiaire apparaît vraiment comme très proche du rôle d'une machine destinataire. La différence entre TBXcast et Xcast, du point de vue du code, se fera sûrement au niveau de la structure de l'entête où il faudra insérer l'arbre. De plus, il faudra avoir des fonctions permettant de tester l'arbre et de récupérer les sous-arbres correspondants. Le graphique suivant nous présente le diagramme d'activité d'un routeur intermédiaire et d'une destination.

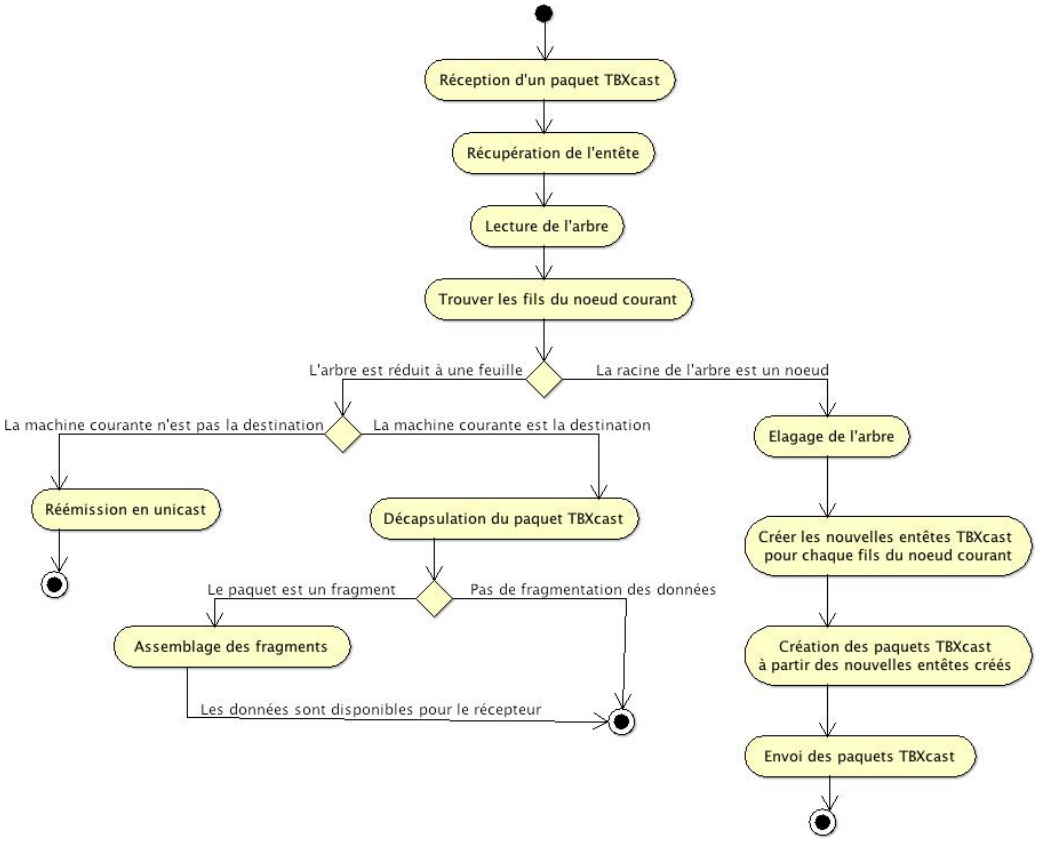


Diagramme 2 : le diagramme d'activité des routeurs intermédiaires et de destinations

3.3 Les différentes versions

Le groupe de projet TBXcast de l'année passée avait choisi de scinder le développement de TBXcast en plusieurs parties, en rajoutant à chaque fois de nouvelles fonctionnalités au protocole.

Nous allons continuer dans ce sens cette année : en effet, même si notre approche est différente car davantage conditionnée par Xcast, il nous semble judicieux de modifier ce code Xcast ou d'insérer de nouvelles fonctionnalités de manière progressive. De plus, des phases de test suivront chacune des étapes de développement.

La suite présente un découpage possible du développement de TBXcast, et reprend donc les fonctionnalités et concepts présentés tout au long de cette partie.

Ce découpage aboutit à une version finale de TBXcast qui est opérationnelle et autonome. Il serait ambitieux de vouloir parvenir à une telle version pendant cette année de projet 2008-2009, mais il nous a semblé important de garder une vision sur ce que TBXcast devra être à terme.

Version 0

Renommer toutes les fonctions, constantes et macros Xcast.

Effectuer également les changements pertinents dans l'en-tête du paquet Xcast (route type, version etc.).

On pourra également ajouter des fonctions d'affichage pour suivre le comportement du programme avec plus de précision.

Version 1

Réaliser l'envoi de paquets TBXcast simplifiés à travers les routeurs : on donne un **arbre simplifié** formé d'une racine et de N nœuds correspondant aux N destinataires.

C'est fondamentalement la technologie Xcast qui agit, mais cette version permet d'introduire les « vrais » paquets TBXcast (envoi et réception de N paquets unicast).

Version 2

Les routeurs intermédiaires sont capables de traiter les paquets TBXcast et les destinataires sont capables de les recevoir.

L'arbre est donné à la source.

Dans cette version, les routeurs intermédiaires (de branchement) savent réellement traiter l'arbre non-simplifié donné en entrée.

Version3

La source sait construire l'arbre à partir d'une topologie fournie en entrée.

Version 4

Gestion de **la segmentation de l'arbre** et fragmentation des paquets de manière cohérente.

Version 5

Gestion des groupes avec xcgroup.

Version 6

Récupération de la topologie grâce au protocole OSPF (entre autres).

On a ici un protocole autonome qui sait récupérer la topologie, construire et segmenter l'arbre, et transmettre les informations aux routeurs afin que les paquets soient traités et reçus correctement.

Version 7

Gestion de la QoS. Prise en compte simultanée de différents indicateurs : délai, gigue, perte d'information etc.

Extension de l'algorithme de création de l'arbre pour s'adapter suivant ces différents paramètres et ainsi fournir un arbre adapté à des contraintes données.

3.4 Conclusion

La construction de l'arbre et les modifications des membres ne sont pas compliquées, comme nous venons de le voir dans cette partie, mais vont le devenir lorsque l'on va devoir agir sur le code. Il faudra modifier structures et code de Xcast pour obtenir notre protocole. Le développement de TBXcast peut devenir très compliqué si on ne teste pas suffisamment lors de notre passage de Xcast vers TBXcast. Pour y remédier, on a donc mis en place une salle d'expérimentation et on va créer une application classique de test : ping6tbx.

4 Expérimentation

Pour tester TBXcast on aura principalement besoin de deux éléments : une plateforme et une application.

4.1 Salle

Le développement de TBXcast devra se faire pas-à-pas et demandera de nombreux tests pour valider chaque modification. Le développement et les tests se feront donc de manière concourante. Nous travaillerons sur une plateforme d'expérimentation, qui nous permettra de valider les différentes versions et sous-versions.

4.1.1 Base de développement

L'équipe de l'année précédente avait choisi de développer sur un système NetBSD 3.1. Les structures de NetBSD sont très claires et très propres, ce qui facilite le développement.

Comme indiqué lors de la phase de pré-étude, nous avons décidé de profiter du travail de l'année dernière et de rester sous NetBSD pour continuer le développement. Mais nous avons choisi de mettre à jour vers la version 4.0. Xcast possède une version finale pour 3.1. Pour NetBSD 4.0, c'est une version de développement de Xcast qui est proposée.

Nous avons repris la plateforme réseau de l'année précédente, et l'avons mise à jour, tant du côté système que du côté matériel. Nous avions à notre disposition les postes de l'année précédente et de nouveaux postes, il a donc fallu migrer de NetBSD 3.1 et Xcast sur les anciens postes vers les nouvelles machines, puis vers NetBSD 4.0 avec Xcast.

Les premiers tests se sont faits avec deux postes directement reliés ; une fois la migration terminée, nous avons simulé un réseau plus conséquent avec trois clients et un routeur. Ce test a également permis de se familiariser avec le switch[14] mis à notre disposition.

4.1.2 Architecture réseau

Nous aurons 6 à 7 postes pour nos expérimentations. A ces postes s'ajoute une machine sous Linux pour l'accès à internet. Nous prévoyons un réseau IPv4 pour pouvoir contrôler nos postes par le réseau et être sûrs d'avoir un lien constant même si le réseau de test était défectueux. Ce réseau monopolisera la carte réseau intégrée de chaque poste, mais nous avons 12 cartes supplémentaires pour le réseau TBXcast. Notre réseau d'expérimentation est constitué de plusieurs sous-réseaux en IPv6, reliés par le routage.

Le switch qui nous est fourni est configurable, il permet la création de VLANs (Virtual Local Area Network, ou réseaux virtuels). Nous pouvons donc modifier la topologie du réseau en quelques commandes et sans brancher ou débrancher les nombreuses connexions. Avec une répartition adéquate des cartes réseaux sur les postes, nous sommes capables de couvrir un large éventail de topologies différentes.

Ainsi, sur une architecture à 7 postes avec 2 cartes réseaux supplémentaires, nous répartirons les cartes de la manière suivante :

- Poste 1 : 3 interfaces + 1 interface standard
- Poste 2 : 3 interfaces + 1 interface standard
- Poste 3 : 3 interfaces + 1 interface standard
- Poste 4 : 2 interfaces + 1 interface standard
- Poste 5 : 1 interface + 1 interface standard
- Poste 6 : 1 interface + 1 interface standard
- Poste 7 : 1 interface + 1 interface standard

Voici des exemples de topologies possibles :

[14] Equipement qui relie plusieurs interfaces dans un réseau informatique.

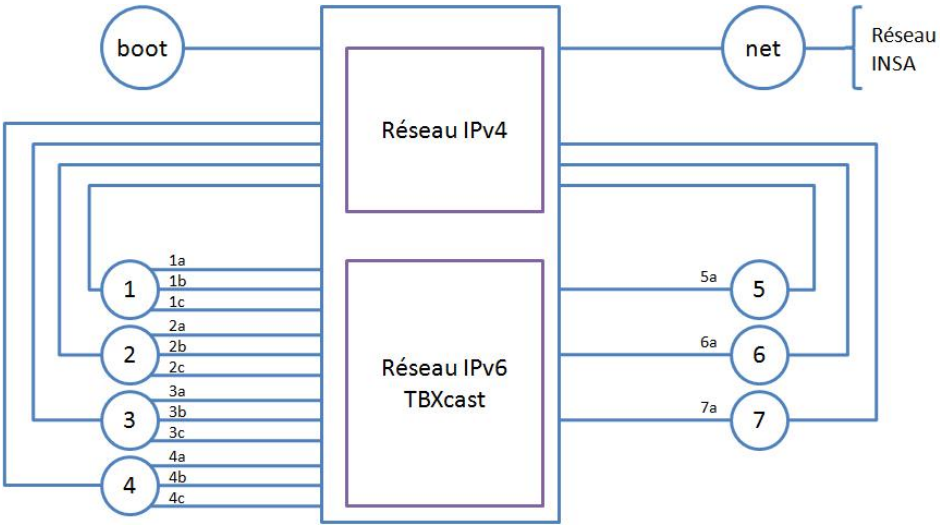


Figure 8 : Topologie de la salle d'expérimentation

On configure deux réseaux sur le switch. Un réseau IPv4 qui ne changera pas. Sur celui-ci, tous les postes sont connectés pour bénéficier d'une connexion constante utile pour lancer des scripts. Et un réseau IPv6 qui est en réalité une liste de sous-réseaux. C'est le réseau d'expérimentation dont voici les topologies.

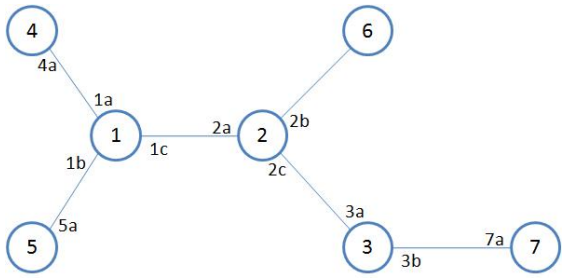


Figure 9 : Exemple de topologie

Topologie avec trois routeurs en forme d'arbre. Un paquet qui traverse l'arbre doit passer par de nombreux postes et peut avoir trois destinataires différents.

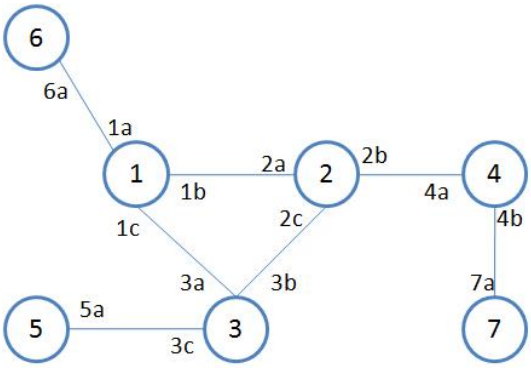


Figure 10 : Exemple de topologie

Topologie à quatre routeurs avec des routeurs en triangle au centre. Une topologie intéressante pour laisser un premier choix de routage au protocole.

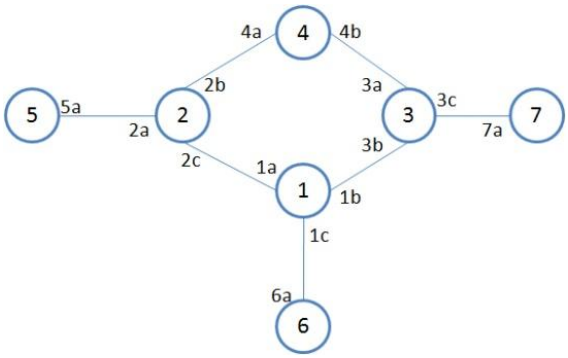


Figure 11 : Exemple de topologie

Topologie à quatre routeurs formant un carré. Une seconde topologie pour tester les choix de routage.

4.1.3 Mécanismes d'automatisation

Pour accélérer les expérimentations, nous élaborerons des scripts pour automatiser le déploiement du code et contrôler plus facilement les postes.

Plusieurs possibilités s'offrent à nous : netboot, contrôle par ssh des postes.

Les axes de travail tournent autour de :

- la centralisation des programmes et du système, pour qu'à tout moment chaque poste soit exactement sur la même base de système et la même version de TBXcast.
- les tests automatisés par ssh, pour que sans devoir lancer les programmes de test et de monitoring sur chaque poste, on puisse avoir une vision globale du comportement de TBXcast.
- la configuration automatique de la topologie, de manière à ce qu'avec le simple plan de la topologie, le réseau puisse se configurer sans intervention.

Ces mécanismes pourraient faciliter sensiblement les tests du protocole et nous permettre de nous concentrer sur l'essentiel, le développement.

Pour permettre un travail dans les meilleures conditions, des efforts indispensables sont à développer sur la salle d'expérimentation. Les machines de test sont un outil très important du développement de TBXcast. Les applications de test le sont tout autant.

4.2 Application de test : ping

Afin de vérifier le fonctionnement du protocole TBXcast lors du développement, il nous faudra une application de test. Les protocoles IPv4, IPv6 et Xcast possèdent tous une version d'une application de test simple : ping. Comme on base le développement de TBXcast sur le code de Xcast, il est utile d'étudier l'application ping6x (ping pour Xcast6) afin de mieux diriger la conception de notre propre application de test : ping6tbx.

4.2.1 Étude de ping6x

4.2.1.1 Fonctionnement

Le principe de Ping est l'envoi d'un paquet de contrôle ICMPv6 à un poste (ou plusieurs dans le cas de ping6x) et attendre une réponse de celui-ci en mesurant éventuellement le temps qu'il faut au paquet pour faire un aller-retour. Les messages ICMP utilisés sont *ECHO REQUEST* et *ECHO REPLY*. Le poste émetteur envoie un message *ECHO REQUEST* au poste cible. Par défaut celui ci doit répondre avec un message *ECHO*

REQUEST. Le contenu du paquet *ECHO REQUEST* est usuellement la suite des lettres de l'alphabet. Dans tous les cas le paquet *ECHO REPLY* doit contenir toutes les données du paquet auquel il correspond. On se sert de cette propriété pour mesurer le délai.

4.2.1.2 Étude du code

Ping6x prend en paramètre une liste des options et une suite d'adresses auxquelles il faut envoyer le message de contrôle. Xcast définit une structure qui lui permet d'envoyer les messages ICMP multicast. Cette structure contient une adresse spéciale, celle du destinataire qui va envoyer la réponse ECHO REPLY. Par défaut celle-ci est le poste avec la première adresse dans la liste mais on peut éventuellement le spécifier en rajoutant son index en paramètre.

De plus on remarque que ping6x n'utilise pas LibXcast. Il crée un socket qui le lie au pseudo-device de Xcast et par sa propre fonction, crée le paquet de contrôle ICMP et l'envoie plusieurs fois. Les éventuelles réponses sont capturées et imprimées. Chaque paquet contient aussi le PID du processus Ping qui l'a créé afin de permettre à plusieurs instances de Ping d'être exécutées simultanément sur la même machine.

4.2.2 Conception de ping6tbxLite

Ping6x envoie des messages ICMPv6, de ce fait il ne peut pas se servir de LibXcast, car celui-ci ne les traite pas. Ceci, et le fait que le code de ping6x soit compliqué nous amène à une décision difficile. Va-t-on se baser sur ping6x ?

Comme on aura besoin d'une application de test très simple qui pourra servir à capturer les bugs dans le code que l'on va modifier il sera préférable de faire une nouvelle application : ping6tbxLite. Dans un premier temps cette application pourra se contenter d'envoyer un paquet à un poste à travers TBXcast. La réception du paquet sera détectée par un sniffer. Ensuite, on rajoutera l'utilisation de LibTBXcast, en ayant préalablement testé libXcast avec une application semblable. On pourra éventuellement créer une simple application serveur qui capturera ces paquets et enverra une réponse.

5 Conclusion

Ce rapport de spécification fonctionnelle pose les bases de notre projet en définissant le fonctionnement de TBXcast par rapport à Xcast. De plus, celui-ci dévoile les points à modifier dans le code Xcast et les structures à rajouter.

Nous aurons donc à implémenter les fonctionnalités de l'arbre et à modifier celles de Xcast déjà présentes. Il faudra penser dans un avenir proche à la structure utilisée pour stocker l'arbre dans l'entête IPv6 pour qu'elle soit simple et efficace. Comme on l'a vu dans la partie Version (partie 3.3), nous avancerons pas à pas dans le projet en espérant pouvoir aller le plus loin possible dans les versions de TBXcast.

Avant tout chose, il est important de réaliser une planification précise des tâches à effectuer. Cette planification nous permettra de nous organiser avant la phase de conception, et de visualiser les versions qui sont à notre portée.

6 Annexe : Diagramme d'activité de xcast6_branch

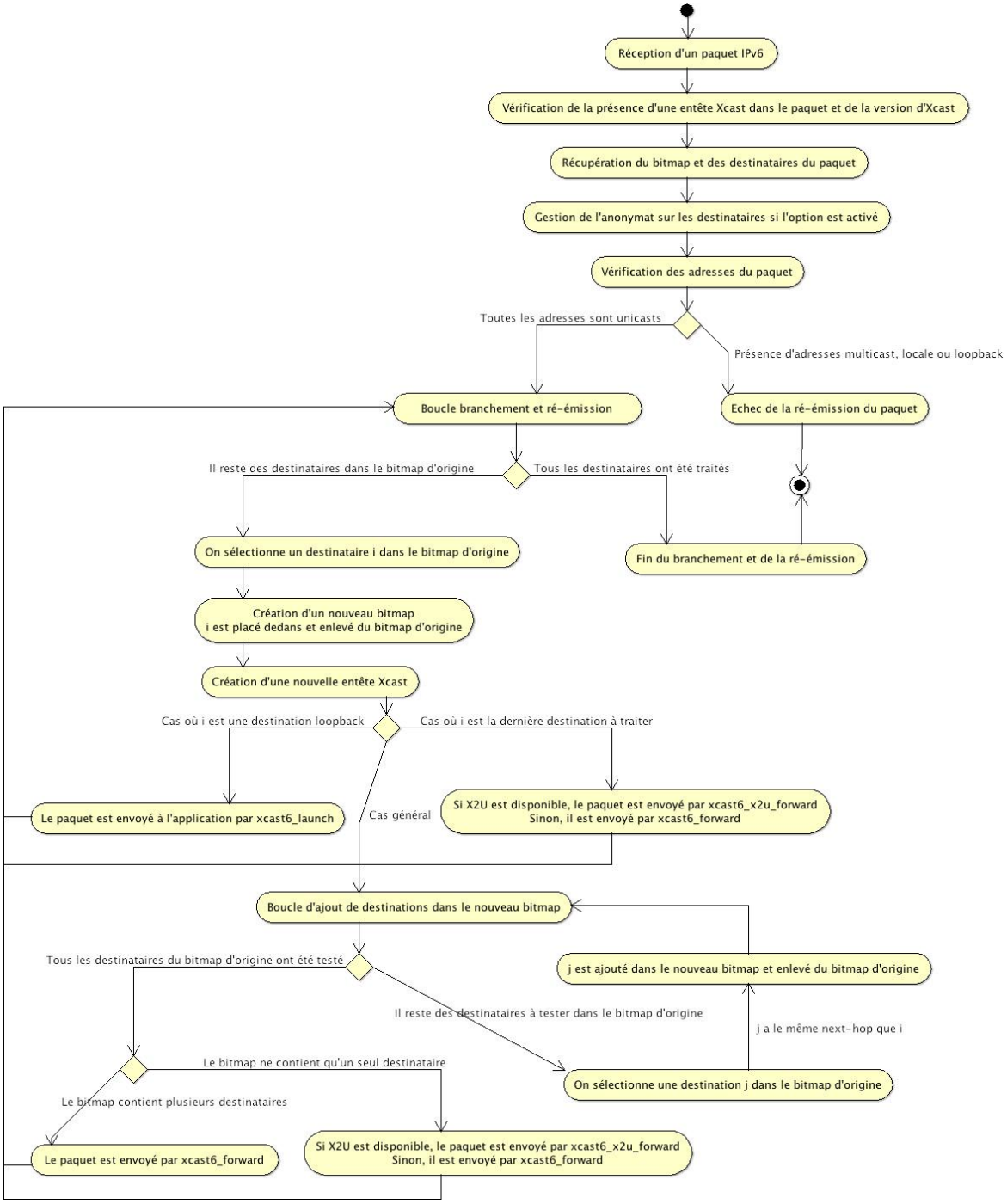


Diagramme 3: Diagramme d'activité de la fonction xcast6_branch

Ce diagramme représente la suite des opérations principales effectuées par la fonction xcast6_branch. Nous retrouvons les étapes liées à la réception du paquet et à la vérification de son intégrité dans un premier temps. Puis dans un second temps, le diagramme montre les différentes possibilités de branchement d'un paquet. A noter que nous avons relevé ici les grandes lignes de la fonction xcast6_branch afin d'en dégager son fonctionnement principal. Le diagramme n'est donc pas exhaustif.