Projet de programmation réseau
Groupes de discussion par arbre couvrant

Juliusz Chroboczek

14 novembre 2018

1 Introduction

> I think that I shall never see
> A graph more lovely than a tree.
>
> A tree whose crucial property
> Is loop-free connectivity.
>
> A tree which must be sure to span
> So packets can reach every LAN.
>
> First the Root must be selected.
> By ID it is elected.
>
> Least cost paths from Root are traced.
> In the tree these paths are placed.
>
> A mesh is made by folks like me
> Then bridges find a spanning tree.

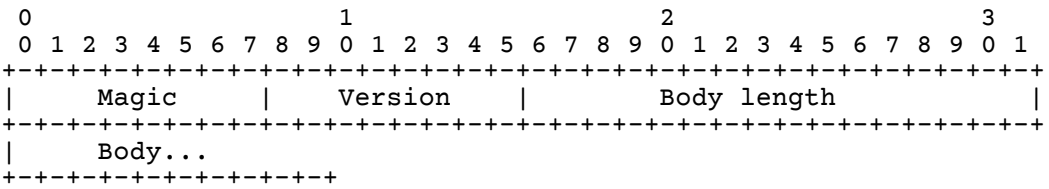
(Radia Perlman, An Algorithm for Distributed Computation of a Spanning Tree in an Extended LAN, *ACM SIGCOMM Computer Communication Review*, 1985.)

Le but de ce projet est d'implémenter un protocole de discussion de groupe, de style *IRC*, mais de façon complètement décentralisée : il n'y a pas de serveur central, les nœuds maintiennent un arbre couvrant et distribuent les messages au dessus d'un protocole fiable en suivant l'arbre couvrant.

2 Format des messages

Un message est encapsulé dans un datagramme UDP ayant une charge d'une taille inférieure ou égale à 1024 octets (entête de couche application inclus). Il consiste d'un entête de message suivi d'une suite de messages.

Le corps du datagramme (message de couche application) a le format suivant :



Les champs sont définis comme suit :

Magic : cet octet vaut 94. Tout message qui ne commence pas par un octet valant 94 sera ignoré par le récepteur.

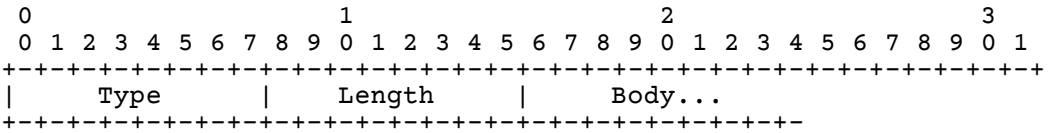
Version : cet octet vaut 0. Tout message qui ne contient pas un champ version valant 0 sera ignoré par le récepteur.

Body length : ce champ indique la longueur des données qui suivent (sans compter les champs *Magic*, *Version* et *Body length*). Si la charge du datagramme UDP est plus grande que *Body Length*+ 4, les données supplémentaires sont ignorées.

Body : ce champ contient le corps du message, une suite de TLV.

2.1 Format des TLV

Le corps d'un message consiste d'une suite de TLV, des triplets (*type*, *longueur*, *valeur*). À l'exception du TLV *Pad1*, chaque TLV a la forme suivante :



Les champs sont définis comme suit :

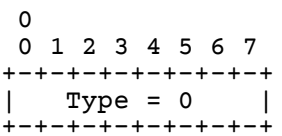
Type : le type du TLV, un entier.

Length : la longueur du corps, sans compter les champs *Type* et *Length*.

Body : une suite d'octets de longueur *Length*, dont l'interprétation dépend du TLV.

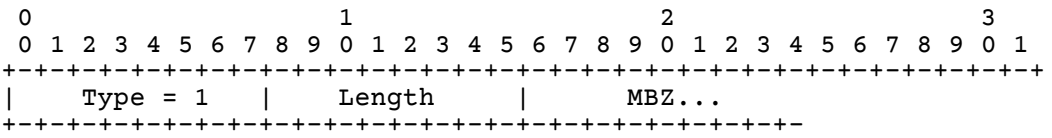
2.2 Détails des TLV

Pad1



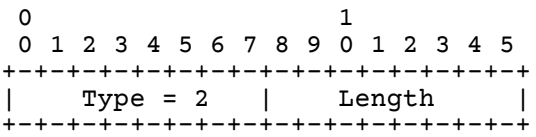
Ce TLV est ignoré à la réception.

PadN



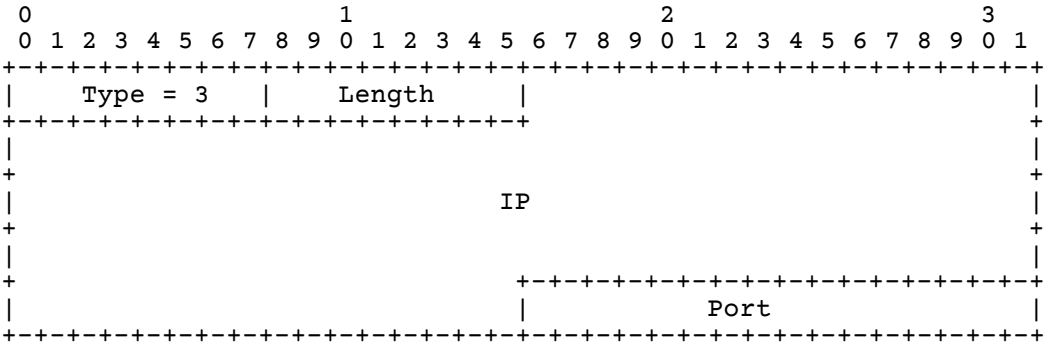
Ce TLV est ignoré à la réception. Le champ MBZ est une suite de zéros à l'émission, et il est ignoré à la réception.

Neighbour request



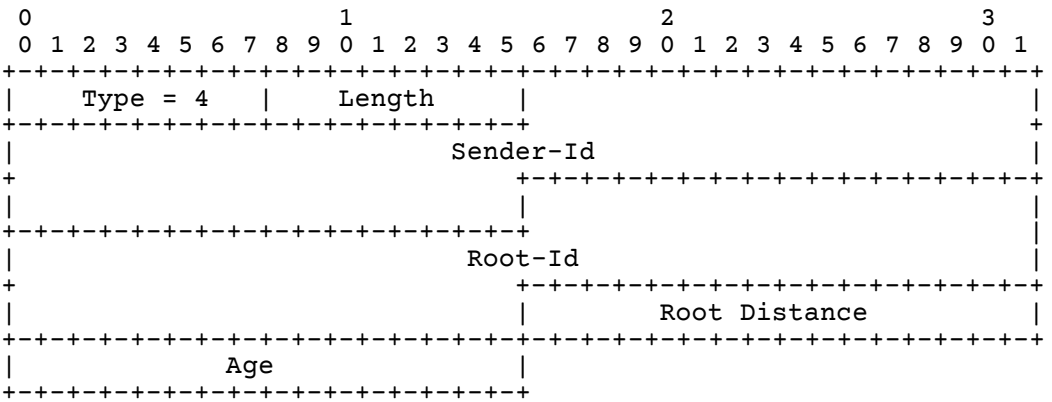
Ce TLV demande au récepteur d'envoyer un *Neighbour reply*

Neighbour reply



Ce TLV contient l'adresse d'un voisin vivant de l'émetteur. Il est envoyé en réponse à un *Neighbour request*. Les adresses IPv6 sont représentées telles quelles, les adresses IPv4 sont représentées sous forme *IPv4-Mapped* (dans le préfixe `::ffff:0:0/96`).

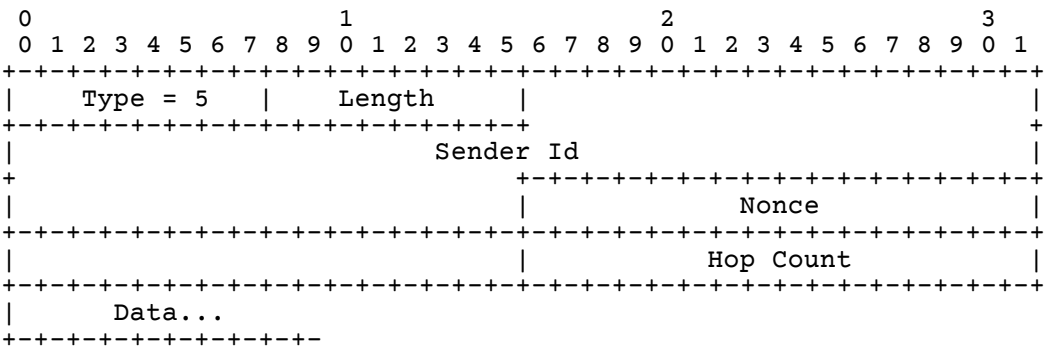
Hello



Ce TLV sert à la construction de l'arbre couvrant. Champs :

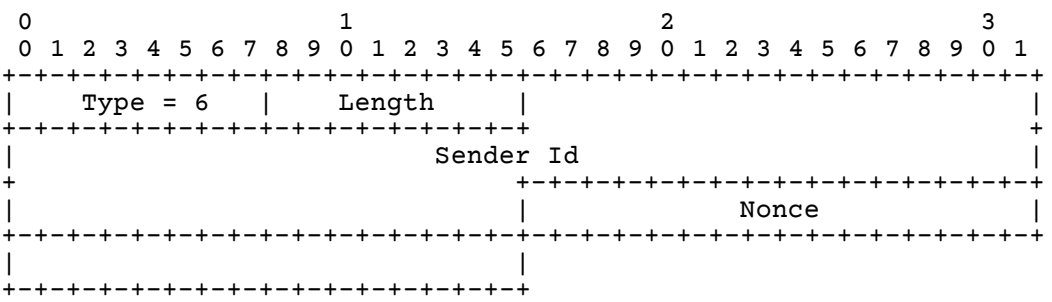
Sender-Id : l'*Id* de l'émetteur ;
Root-Id : l'*Id* de la racine de l'arbre couvrant ;
Root Distance : la distance entre l'émetteur et la racine, en nombre de sauts ;
Root Distance : une estimation de l'âge de ce *Hello*, en secondes.

Data



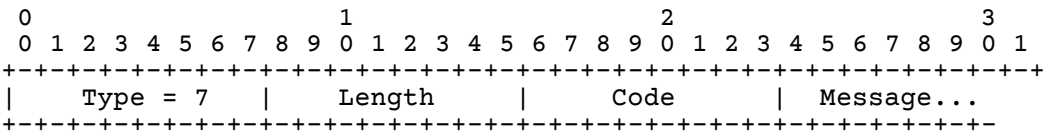
Ce TLV contient des données envoyé au groupe de discussion. Le champ *Sender Id* contient l'*Id* de l'originateur du message. Le champ *Nonce* contient une chaîne de 32 bits arbitraire choisie de façon à ce que la paire (*Sender Id*, *Nonce*) soit globalement unique ; on peut par exemple utiliser un compteur séquentiel, la tirer au hasard, ou y coder l'heure d'émission avec une granularité suffisante pour qu'elle soit unique (mais attention si vous utilisez des *threads*). Le champ *Data*, de longueur arbitraire, contient les données à afficher dans le groupe de discussion ; il est normalement de la forme « *nick* : *message* », où *nick* et *message* sont des chaînes de caractères codées en UTF-8, mais il peut en principe contenir des données arbitraires.

Ack



Ce message acquitte la réception d'un message *Data*. Les champs *Sender Id* et *Nonce* sont recopiés depuis le message *Data* correspondant.

GoAway



Ce message sert à supprimer explicitement une relation de voisinage, ce qui évite de passer du temps à inonder vers un voisin qui n'existe plus. Lorsqu'il reçoit un TLV *GoAway*, le récepteur marque l'émetteur comme mort (s'il est statiquement configuré, voir ci-dessous), ou alors le supprime carrément de sa liste de voisins. Le champ *Code* indique la raison de la suppression, et il peut avoir les valeurs suivantes :

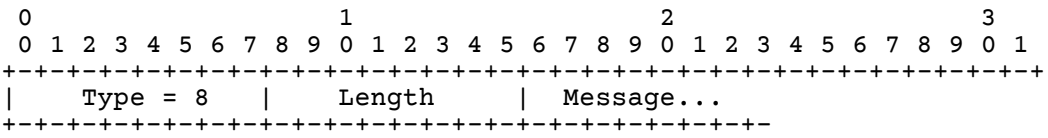
0 raison inconnue ;
1 l'émetteur quitte le réseau ;
2 le récepteur n'a pas envoyé un paquet depuis trop longtemps ou alors il n'a pas acquitté un TLV *Data* dans les temps ;
3 le récepteur a violé le protocole ;
4 le récepteur a trop de pairs associés.

Les valeurs inconnues du champ *Code* doivent être traitées comme la valeur 0.

Le champ *Message* est optionnel, et il contient un message lisible par un être humain, codé en UTF-8. Il est utilisé par mon implémentation pour vous aider lors du déboggage.

Attention, UDP est un protocole non-fiable, et par ailleurs un voisin peut très bien quitter le réseau sans l'indiquer à la couche application. L'utilisation du message *GoAway* est une optimisation utile, mais vous ne pouvez pas compter sur sa réception.

Warning



Ce TLV permet d'envoyer un message pour l'implémenteur ; mon implémentation l'envoie si la votre fait des choses bizarres mais techniquement légales. Le champ *Message* contient un message lisible par un être humain, codé en UTF-8, que vous pouvez simplement afficher.

3 Déroulement du protocole

Le protocole est un protocole pair-à-pair pur, dans lequel chaque pair est identifé par un *Id*, une suite globalement unique de 64 bits (8 octets). Au démarrage, votre implémentation choisira un *Id* globalement unique, par exemple en le tirant au hasard.

3.1 Découverte de voisins

Chaque nœud maintient une liste de relations de voisinage ou *liens*. Outre les informations décrites aux paragraphes suivants, chaque lien est équipé des informations suivantes :

- l'adresse IP (IPv4 ou IPv6) du voisin et le numéro de port UDP sur lequel il écoute ;
- un bit qui indique si le lien est statiquement configuré ou transient ;
- pour un lien statiquement configuré, un bit qui indique s'il est vivant (un voisin transient est toujours vivant) ;
- une date qui indique la date de dernière réception d'un paquet depuis ce voisin ;
- une date qui indique la date de dernière émission d'un paquet à ce voisin.

Chaque nœud est initialement configuré avec les adresses IP et numéros de port d'un petit nombre (typiquement 1 ou 2) de voisins statiquement configurés. Lorsqu'il reçoit un paquet avec un entête correct, et que le nombre total de voisins vivants est strictement inférieur à 6, le nœud ajoute l'émetteur du paquet à sa liste de liens s'il en est absent (et dans ce cas le marque comme transitoire), puis met à jour sa date de dernière réception. Si le nombre de voisins vivants est supérieur ou égal à 6, le récepteur tire au hasard un voisin *v* vivant, puis envoie à l'émetteur un TLV *Neighbour* avec *v* suivi d'un TLV *GoAway* de code 4.

À chaque fois qu'il émet un paquet à un voisin, le nœud met à jour la date de dernière émission. Si aucun paquet n'a été émis à un voisin depuis 20 secondes, le nœud envoie un paquet vide à ce voisin (entête et zéro TLVs).

Si la date de dernière réception d'un paquet depuis un voisin est plus de 65 secondes dans le passé, le voisin est probablement mort. S'il est transitoire, il est éliminé de la liste de voisins ; s'il est statiquement configuré, il est marqué comme non-vivant.

Si le nombre de voisins vivants est strictement inférieur à 3, le nœud envoie à l'un de ses voisins vivants une demande de nouveau voisin. Lorsqu'il reçoit une réponse à sa demande, et le nombre de voisins vivants est encore inférieur ou égal à 6, le nœud envoie un paquet vide au nouveau voisin. Un nœud n'envoie pas plus d'une demande de nouveau voisin toutes les 5 secondes.

Lorsqu'il reçoit une demande de nouveau voisin, le nœud choisit au hasard un de ses voisins vivants autre que l'émetteur de la demande, et répond par un TLV *Neighbour*.

3.2 Construction de l'arbre couvrant

Le but de l'algorithme de construction de l'arbre couvrant est de construire un arbre couvrant du graphe de voisinage dont la racine est le nœud de plus petit *Id*.

En outre des informations décrites au paragraphe suivant, chaque relation de voisinage vivante est équippée des informations suivantes :

- son *état*, qui peut valoir *forwarding*, *backup*, *pre-forwarding* ou *pre-backup* ;
- si l'état est *pre-forwarding* ou *pre-backup*, la date à laquelle le lien est passé dans cet état ;
- l'infomation du *Hello* le plus jeune reçu sur ce lien, qui contient l'*Id* de l'émetteur du *Hello*, l'*Id* de la racine, et la date à laquelle ce *Hello* a été produit par la racine. On appelle *âge* du *Hello* la différence entre l'heure courante et la date d'origine du *Hello* ;
- un bit qui indique si le nœud est *désigné* pour ce lien.

Le but de l'algorithme de construction de l'arbre couvrant est de construire un arbre couvrant du graphe de voisinages dont les arêtes sont dans l'état *forwarding*, et dont les arêtes descendantes ont le bit désigné mis à 1.

3.2.1 État initial

Lorsqu'un lien est créé, il est dans l'état *pre-forwarding* et son bit désigné est à 1.

3.2.2 Élection de la racine

Le nœud commence par choisir la racine de l'arbre à laquelle il va participer et calculer la distance à laquelle il se trouve de la racine. Pour cela, il parcourt tous les liens, et calcule le plus petit (au sens de `memcmp`) *Id* racine *d* parmi les *Hellos* associés aux liens.

Si *d* n'existe pas (aucun lien n'a de *Hello* associé), ou si *d* est strictement plus grand que l'*Id* du nœud courant (au sens de `memcmp`), alors le nœud s'élit lui-même racine, et la distance à la racine est 0. Sinon, le nœud d'*Id d* est choisi comme racine, et la distance du nœud est la plus petite distance associée à un *Hello* d'*Id* racine *d* plus 1.

3.2.3 Élection des nœuds désignés

Une fois choisie la racine, le nœud décide s'il est désigné pour chacun de ses liens. Pour chaque lien, il consulte le *Hello* associé :

- s'il n'y a pas de *Hello* associé au lien, alors le nœud est désigné pour ce lien (le bit *désigné* est mis à 1) ;
- si l'*Id* racine du *Hello* associé au lien est différent de l'*Id* élu racine à l'étape précédente, alors le nœud est désigné pour ce lien ;
- sinon, l'*Id* racine du *Hello* associé au lien est égal à l'*Id* élu racine, et on compare la distance à la racine du nœud à la distance du *Hello* associé au lien ; si la distance est strictement inférieure, le nœud est désigné pour ce lien ; si elle est strictement supérieure, alors le nœud n'est pas désigné ;
- sinon, les *Id* racines et les distances sont égales, et on compare l'*Id* du nœud avec l'*Id* du voisin ayant servi le *Hello* (pas l'*Id* racine !) ; si l'*Id* du nœud courant est strictement inférieur, alors le nœud est désigné pour le lien ; s'il est supérieur ou égal, alors le nœud n'est pas désigné.

3.2.4 Choix du lien ascendant

Un nœud qui ne s'est pas élu racine choisit à tout moment exactement un lien « ascendant » qui a les propriétés suivantes :

- le voisin est vivant ;
- le nœud n'est pas désigné pour ce lien ;
- la distance à la racine du *Hello* associé est minimale parmi tous les liens qui satisfont les contraintes ci-dessus (*i.e.* elle est égale à la distance calculée ci-dessus moins 1).

Bien sûr, le lien ascendant est recalculé à chaque fois que l'ensemble des voisins vivants change, lorsque l'ensemble des liens désignés change, lorsque les distances à la racine des liens changent. Remarquez aussi que la description ci-dessous est non-déterministe — il y a un choix arbitraire à faire parmi les chemins de distance minimale (et il est bon de le faire de façon à privilégier la stabilité).

3.2.5 État des liens

Après avoir calculé l'ensemble des liens désignés, le pair recalcule l'état des liens. Pour chaque lien qui est soit le lien ascendant soit un lien désigné, il essaie de passer le lien dans l'état *forwarding* :

- si le lien est dans l'état *forwarding* ou *pre-backup*, son état est immédiatement changé en *forwarding* ;
- si le lien est dans l'état *pre-forwarding*, il n'y a rien à faire ;
- si le lien est dans l'état *backup*, alors il est mis dans l'état *pre-forwarding*, et la date actuelle est notée.

Pour tous les autres liens (ni ascendant ni racine), le nœud essaie de les passer à l'état *backup* :

- si le lien est dans l'état *backup* ou *pre-forwarding*, le lien est immédiatement changé en *backup* ;
- si le lien est dans l'état *pre-backup*, il n'y a rien à faire ;
- si le lien est dans l'état *forwarding*, il est passé dans l'état *pre-backup*, et la date actuelle est notée.

Par ailleurs, périodiquement le nœud parcourt la liste des liens dans l'état *pre-forwarding* et *pre-backup*, et les transitionne dans l'état *forwarding* ou *backup* respectivement si la date de dernière transition est plus de 10 *s* dans le passé.

3.2.6 Émission de *Hellos*

Toutes les 8 secondes environ, un nœud envoie un TLV *Hello* sur tous les liens où il est désigné (le bit *désigné* est mis à 1). Les paramètres du *Hello* sont les suivants :

- l'*Id* de l'émetteur est l'*Id* du nœud ;
- l'*Id* de la racine est l'*Id* du nœud élu racine ;
- la distance à la racine est la distance à la racine calculée ci-dessus ;
- l'âge du *Hello* est 0 si le nœud s'est élu lui-même racine ; sinon, c'est la différence entre l'heure courante et la date d'origine du *Hello* associé au lien ascendant, arrondie vers le haut (ce qui implique qu'il vaut au moins 1).

Remarquez que le *Hello* n'est pas envoyé en réaction à la réception d'un *Hello* sur le lien ascendant — le *Hello* reçu est stocké dans une structure de données, et l'émission est découplée de la réception. Cette technique (classique) évite d'avoir un taux de pertes qui croît avec le diamètre du réseau.

Remarquez aussi qu'aucun *Hello* n'est envoyé sur un lien où le pair n'est pas désigné.

3.2.7 Réception et expiration de *Hello*

Lorsqu'il reçoit un *Hello* sur un lien, un nœud se comporte comme suit :

- si l'âge du *Hello* reçu est supérieur à l'heure courante moins la date d'origine du *Hello* associé au lien, le *Hello* est ignoré ;
- sinon, les données du *Hello* sont stockées dans la structure de données correspondant au lien (la date d'origine est calculée comme l'heure courante moins l'âge contenu dans le TLV).

De même, lorsque l'âge du *Hello* associé à un lien dépasse $MaxAge = 30\,s$, le *Hello* associé au lien est effacé (comme si on n'avait jamais reçu de *Hello* sur le lien).

Après avoir mis à jour les structures de données, le nœud élit de nouveau une racine et détermine l'ensemble des liens pour lesquels il est désigné, et recalcule l'état des liens, comme décrit ci-dessus.

3.3 Transfert de données

Lorsqu'il est à l'origine d'une donnée, un nœud la transfère de manière fiable (voir ci-dessous) à tous les voisins qui sont dans l'état *forwarding* avec le champ *Nonce* ayant une valeur qui n'a pas été utilisée récemment et un champ *Hop Count* égal à 32.

Lorsqu'il reçoit une donnée depuis un voisin, le nœud se comporte comme-suit :

- si le *Hop Count* est égal à 0, alors il n'y a plus rien à faire (la donnée est détruite) ;
- la donnée est acquittée à l'aide d'un TLV *Ack* contenant un *Sender-Id* et *Nonce* copiés depuis le TLV *Data* ;
- si l'émetteur est dans l'état *backup* ou *pre-forwarding*, il n'y a plus rien à faire (la donnée acquittée est détruite) ;
- la donnée est affichée dans l'interface utilisateur ;
- si le *Hop Count* est égal à 1, il n'y a plus rien à faire ;
- si l'émetteur n'est pas dans l'état *forwarding* ou *pre-backup*, il n'y a plus rien à faire ;
- sinon (le *Hop Count* est strictement supérieur à 1 et l'émetteur est dans l'état *forwarding* ou *pre-backup*), alors le *Hop Count* est décrémenté, et la donnée est envoyée de façon fiable à tous les voisins qui sont dans l'état *forwarding* ou *pre-backup* à l'exception du voisin dont la donnée a été reçue.

Émission fiable L'émission fiable consiste à envoyer la donnée répétitivement à un voisin, avec *backoff* exponentiel, jusqu'à recevoir un acquittement. Si un acquittement n'est pas reçu au bout d'un certain nombre d'essais, le voisin est éliminé de la liste de voisins.

Plus précisément, pour chaque donnée et chaque voisin auquel la donnée est envoyée, le nœud crée une structure de données qui contient les données suivantes :

- la donnée et le voisin ;
- le nombre de fois n que la donnée a été envoyée.

Le nœud émet alors la donnée, et incrémente n. Si un acquittement arrive, alors la structue de données est détruite ; sinon, le nœud attend 2^n secondes, et recommence. Lorsque n atteint 5, la structure de données et détruite, et le voisin est soit détruit (s'il était transitoire) soit marqué mort (s'il était statiquement configuré), puis l'arbre couvrant est reconstruit.

4 Sous-ensembles, optimisations et extensions

Toute extension au sujet sera acceptée avec enthousiasme et évaluée avec sympathie. A contrario, une implémentation d'un sous-ensemble du sujet sera évaluée avec indulgence du moment qu'elle interopère avec mon implémentation. *Attention :* une implémentation qui n'interopère pas avec la mienne sera notée sévèrement, aussi jolie soit-elle.

4.1 Sous-ensembles

A priori, le protocole décrit ci-dessus peut sembler très riche : il consiste de trois à quatre sous-protocoles (découverte de voisins, élection de la racine et construction de l'arbre couvrant, transfert fiable). Cependant, il est possible d'implémenter un sous-ensemble du protocole et rester interopérable avec mon implémentation.

L'implémentation la plus primitive que je puisse imaginer est celle d'un client de chat, feuille de l'arbre couvrant. Un nœud feuille ne participe pas à la découverte de voisins, il s'associe simplement à un seul voisin statique auquel il envoie périodiquement des paquets vide. Il émet les messages de données dont il est l'originateur (et ignore les acquittements), et acquitte les messages qu'il reçoit. Il n'implémente pas les autres parties du protocole : comme il n'envoie jamais de *Hello*, il n'est jamais élu racine, et comme il n'a qu'un seul voisin, son unique lien est toujours ascendant et donc dans l'état

Je vous encourage à commencer votre travail avec une implémentation feuille, et de l'étendre progressivement aux autres parties du protocole.

4.2 Optimisations

Le protocole décrit ci-dessus peut être optimisé de plusieurs façons. Tout d'abord, le format de paquets permet d'envoyer plusieurs TLV dans un même paquet, ce qui permet d'éviter les entêtes de couches inférieures ; l'implémenteur malin agrégera les TLV dans un tampon (ou une autre structure de données), et les enverra au bon moment pour maximiser la taille des paquets sans introduire trop de délai.

À plusieurs endroits, il est possible d'envoyer les données à des moments différents de ceux décrits dans le sujet. Le cas le plus visible est celui d'un *Hello*, qui pourrait être envoyé immédiatement après un changement des paramètes du nœuds ; attention cependant à ne pas générer trop de traffic durant la reconvergence du protocole.

Le choix des structures de données est laissé à votre choix. Par exemple, il faut une structure de données pour conserver les données émises de façon fiable pour lesquelles on n'a pas encore reçu d'acquittement ; un tas binaire est probablement un bon choix.

4.3 Extensions

Il s'agit d'un projet de réseau, ne passez donc pas trop de temps à faire une jolie interface graphique — un programme s'exécutant dans le terminal ou implémenté comme une passerelle IRC m'ira très bien.

Le protocole suppose que tous les liens sont symétriques. En présence de NAT ou de *firewalls*, il se peut qu'un lien soit asymétrique. Cette situation peut être détectée lorsqu'on continue à recevoir des *Hello* sur un lien où le nœud local se croit désigné. On peut alors passer le lien à un nouvel état *Error* qui l'empêche d'être désigné.

Une extension naturelle est de permettre le transfert d'images. Pour cela, vous pourrez déclarer qu'un TLV *Data* ayant une charge dont le premier octet est strictement inférieur à 20 contient des données binaires, et que ce premier octet indique la nature de ces données.

Tel qu'il est défini, le protocole ne permet pas d'envoyer des messages de plus de 1 ko environ. Pour envoyer des données plus volumineuses, il faudra concevoir et implémenter un mécanisme de fragmentation à la couche application. Attention aux interactions entre fragmentation et fiabilité — vaut-il mieux acquitter le message complet, ou acquitter chaque fragment ?

Il existe de nombreuses extensions à l'algorithme de l'arbre couvrant (RSTP, MSTP, etc.). Je serai intéressé de voir leurs implémentations, surtout si elles interopèrent avec le protocole de base.

5 Modalités de soumission

Vous me soumettrez les sources de votre programme, compilables et exécutables sous Linux Debian, ainsi qu'un rapport sous format PDF qui m'indiquera notamment les parties du sujet et les extensions que vous aurez implémentées.

Vous me soumettrez votre solution par courrier électronique à mon adresse. Le courrier que vous m'enverrez devra impérativement avoir le sujet « `Projet reseaux Cachan` » avec une archive `.tar.gz` en attachement. L'archive devra s'appeler *prénom-nom*.tar.gz, et s'extraire dans un sous-répertoire *prénom-nom* du répertoire courant. Par exemple, si vous vous appelez *Janis Joplin*, votre archive devra porter le nom `janis-joplin.tar.gz` et son extraction devra créer un répertoire `janis-joplin` contenant tous les fichiers que vous me soumettrez.