Projet de programmation réseau
Dazibao par inondation non-fiable

Juliusz Chroboczek

2 mars 2020

1 Introduction

Le but de ce projet est d'implémenter un *dazibao* (« journal à grandes lettres »), semblable à un « mur » de réseau social, mais de façon complètement distribuée. Le protocole est basé sur un algorithme non-fiable d'inondation.

Pour simplifier le protocole, il n'y a pas de mécanisme d'expiration : la dernière donnée publiée par un participant qui a quitté le réseau reste publiée indéfiniment.

Ce protocole est inspiré du protocole *HNCP* conçu par Markus Stenberg et ses acolytes.

2 Structures de données

2.1 Arithmétique et ordres *modulo*

Le protocole manipule des *numéros de séquence*, des entiers de 16 bits interprétés *modulo* 2^{16}. Étant donné un numéro de séquence s et un entier naturel n, la somme de s et n est le numéro de séquence défini par

$$s \oplus n = (s + n) \bmod 2^{16}$$

ou, de façon équivalente,

$$s \oplus n = (s + n) \text{ and } 65535,$$

où « mod » est le reste de la division euclidienne (« `%` » en C) retournant un entier naturel, et « and » est l'opération de conjonction bit-à-bit (« `&` » en C).

Étant donnés deux numéros de séquence s et s', la relation $s \preccurlyeq s'$ est définie par

$$s \preccurlyeq s' \quad \text{lorsque} \quad ((s' - s) \bmod 2^{16}) < 32768$$

ou, de façon équivalente,

$$s \preccurlyeq s' \quad \text{lorsque} \quad ((s' - s) \text{ and } 32768) = 0.$$

Remarquez que « $\preccurlyeq$ » n'est pas une relation d'ordre — on dit parfois que c'est un *ordre cyclique*.

2.2 Données et *hashes*

Chaque nœud participant au protocole est identifié par son *Node Id*, une suite de 64 bits (8 octets) arbitraires et supposée globalement unique ; vous pouvez par exemple le tirer au hasard ou le dériver d'une adresse MAC.

Chaque nœud maintient une table de données publiées. C'est une suite de triplets (ι_i, s_i, d_i), où ι_i est l'*Id* du nœud qui a publié cette donnée, s_i est un numéro de séquence (un entier de 16 bits, interprété *modulo* 2^{16}) et d_i est la donnée elle-même, une suite de 0 à 192 octets interprétée comme une chaîne UTF-8 (sans sentinelle à la fin).

Un nœud publie toujours lui-même une donnée (possiblement la donnée de longueur 0) ; la table de données publiées contient donc toujours au moins une donnée.

Hashes *SHA-256* est une fonction de hachage cryptographique qui à toute suite d'octets associe un *hash* de 256 bits (32 octets). On notera h la fonction qui à une suite d'octets associe les 16 premiers octets de son *SHA-256*. En d'autres termes, pour calculer $\mathrm{h}(x)$, on calcule le *SHA-256* de x puis on tronque le résultat pour avoir une suite de 16 octets. Par exemple, soit x la suite de 6 octets représentant la suite de caractères ASCII « `szczaw` » (sans sentinelle à la fin) ; alors

$$\mathrm{h}(x) = \text{3960a2a8b9fa88c9d7c83969c4641093} \qquad \text{(hexadécimal).}$$

Étant donnée une donnée (ι_i, s_i, d_i), on définit le *hash du nœud i* par

$$h_i = \mathrm{h}(\iota_i \cdot s_i \cdot d_i)$$

où « · » est la concaténation des suites d'octets et s_i est codé en format gros-boutiste *(big-endian)*.

Pour calculer le *hash du réseau*, on trie la liste de données par ordre croissant des *Id* ; on obtient la suite $((\iota_0, s_0, d_0), (\iota_1, s_1, d_1), \ldots (\iota_{n-1}, s_{n-1}, d_{n-1}))$, où $\iota_k < \iota_{k+1}$. Si h_i est le *hash* du nœud de *Id* ι_i (défini ci-dessus), alors le *hash du réseau* est

$$H = \mathrm{h}(h_0 \cdot h_1 \cdots h_{n-1}),$$

i.e. le *hash* de la concaténation de tous les *hashes* de nœuds triés par ordre croissant d'*Id*.

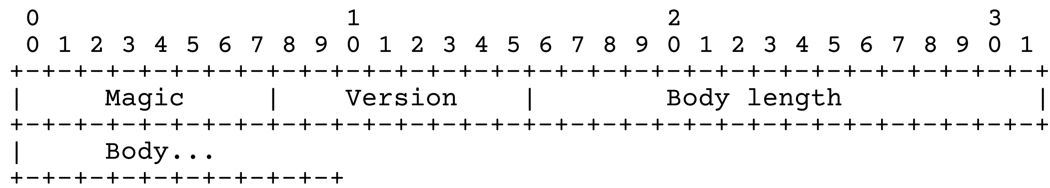
3 Syntaxe du protocole

Dans ce paragraphe, je décris la syntaxe des messages manipulés par le protocole. Le comportement d'un pair est décrit au paragraphe 4.

3.1 Format des paquets

Le protocole est encapsulé dans UDP. Chaque datagramme UDP contient un *paquet*, qui contient lui-même une suite de TLV. Tous les champs représentant des entiers sont codés en format gros-boutiste *(big-endian)*.

Un paquet 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 paquet suivi d'une suite de TLV. Le corps du datagramme a le format suivant :



Les champs sont définis comme suit :

Magic : cet octet vaut 95. Tout paquet qui ne commence pas par un octet valant 95 sera ignoré par le récepteur.

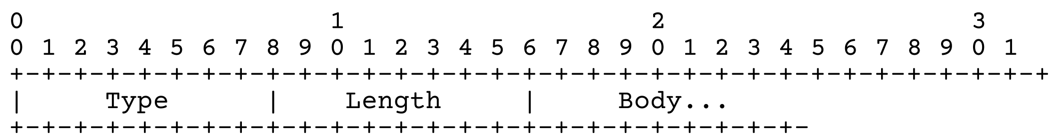
Version : cet octet vaut 1. Tout paquet qui ne contient pas un champ *Version* valant 1 sera ignoré par le récepteur.

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 paquet, une suite de TLV.

3.2 Format des TLV

Le corps d'un paquet 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 :

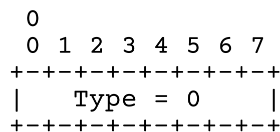
Type : le type du TLV, un entier.

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.

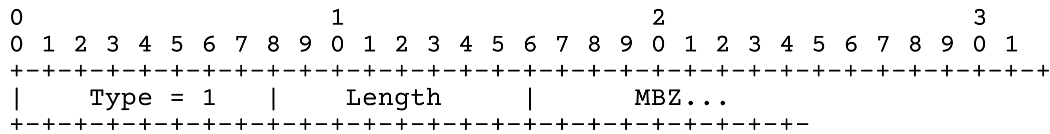
3.3 Détails des TLV

Pad1



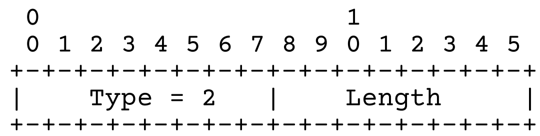
Ce TLV est ignoré à la réception.

PadN



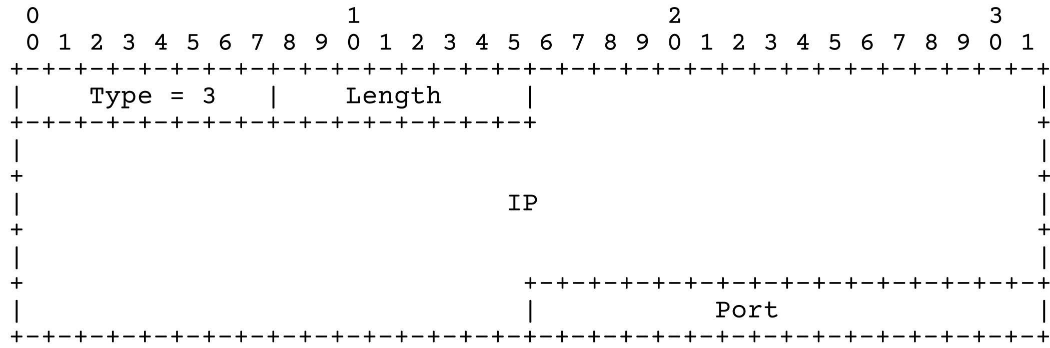
Ce TLV est ignoré à la réception. Le champ MBZ est une suite de zéros dont la longueur est donnée par le champ *Length*.

Neighbour request



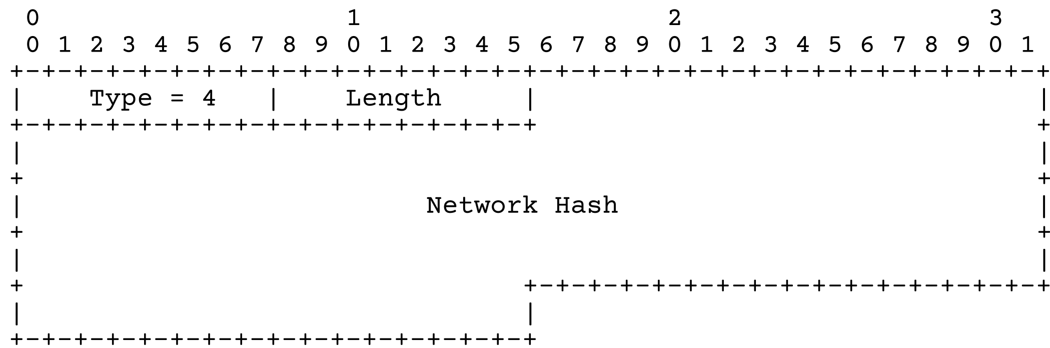
Ce TLV demande au récepteur d'envoyer un TLV *Neighbour*. Il n'a pas de contenu (sa longueur vaut 0).

Neighbour



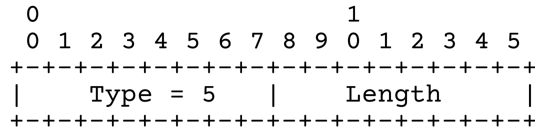
Ce TLV contient l'adresse d'un voisin vivant de l'émetteur. Il est envoyé en réponse à un TLV *Neighbour Request*. Les adresses IPv6 sont représentées telles quelles, les adresses IPv4 sont représentées sous forme *IPv4-Mapped* (dans le préfixe `::ffff:0:0/96`).

Network Hash



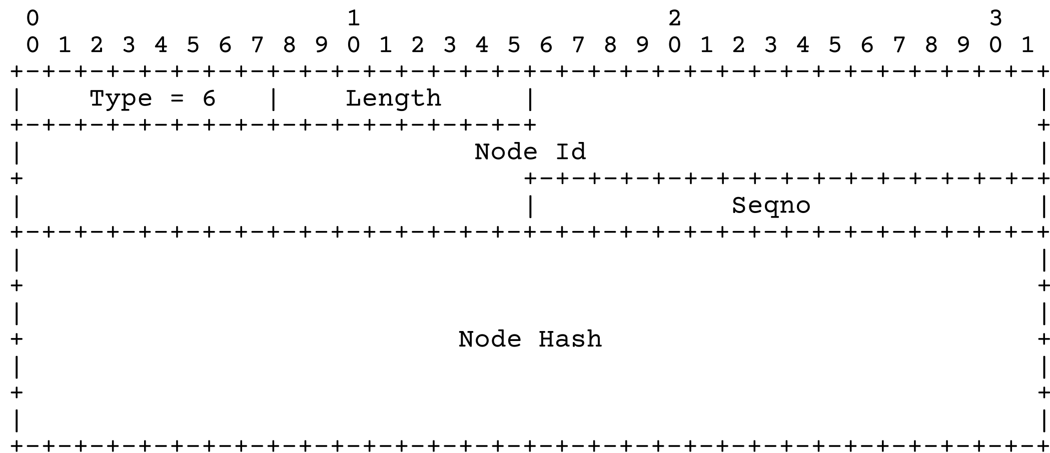
Ce TLV indique l'idée que se fait l'émetteur de l'état actuel du réseau. Il contient le *Network Hash* du réseau calculé par l'émetteur comme indiqué au paragraphe 2.2 ci-dessus.

Network State Request



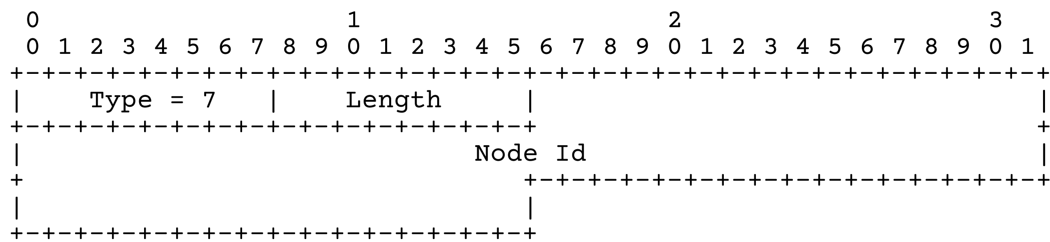
Ce TLV demande au récepteur d'envoyer une série de TLV *Node Hash*, un pour chaque donnée qu'il connaît (pas forcément dans le même paquet). Ce TLV n'a pas de contenu (sa longueur vaut 0).

Node Hash



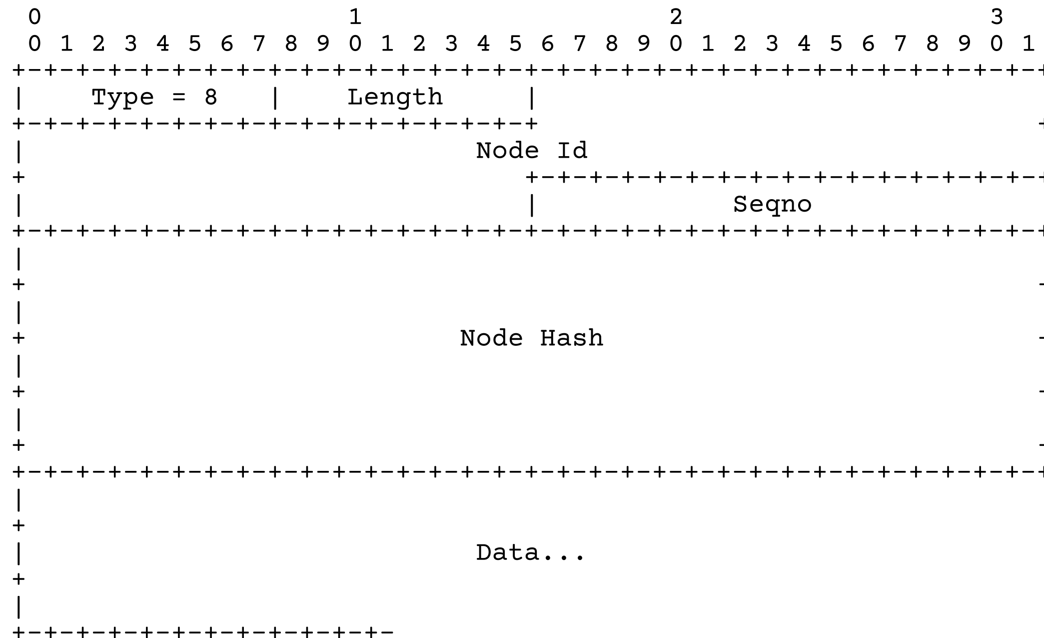
Ce TLV est envoyé en réponse à un TLV *Network State Request*. Il contient l'*Id* d'un nœud, le numéro de séquence actuellement publié par ce dernier, et son *node hash* comme défini au paragraphe 2.2 ci-dessus.

Node State Request



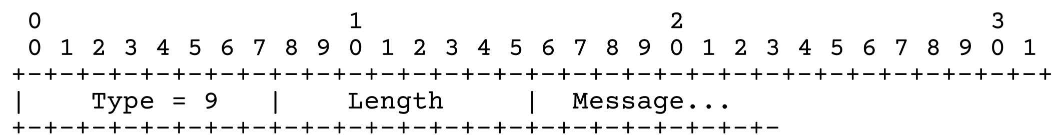
Ce TLV demande au récepteur d'envoyer un TLV *Node State* décrivant l'état du nœud indiqué par le champ *Node Id*.

Node State



Ce TLV est envoyé en réponse à un TLV *Node State Request*. En plus des données contenues dans un TLV *Node Hash*, il contient la donnée associée au nœud dans son intégralité. La donnée ne doit pas dépasser 192 octets. Elle est normalement interprétée comme une chaîne codée en UTF-8 (sans sentinelle à la fin), mais l'inondation est agnostique au format des données : votre pair ne doit pas refuser d'inonder des données qui n'ont pas le bon format (ce qui implique notamment qu'il n'est pas correct d'utiliser le type `string` de Python 3 pour les stocker).

Warning



Ce TLV permet d'envoyer un message pour l'implémenteur ; mon pair l'envoie si votre pair fait des choses bizarres ; il n'est pas nécessaire que votre pair en envoie. Le champ *Message* contient un message lisible par un être humain, codé en UTF-8 (sans sentinelle à la fin).

Autres TLV Un TLV ayant un type inconnu est ignoré lors de la réception (mais les autres TLV contenus dans le même paquet sont traités normalement), ce qui permet d'étendre le protocole sans changer de numéro de version.

4 Déroulement du protocole

Ce protocole contient deux sous-protocoles : le protocole de maintenance de la liste de voisins (paragraphe 4.2) et le protocole d'inondation (paragraphe 4.3).

4.1 Structures de données conceptuelles

Chaque pair est initialisé avec son *Id*, une suite de 8 octets (64 bits) globalement unique ; vous pouvez par exemple la tirer au hasard. Cette valeur est immutable (une fois choisie, elle ne change plus).

Chaque pair a à tout moment une donnée qu'il publie (qui peut être la donnée de longueur nulle) et son numéro de séquence (que vous pouvez initialiser par exemple à 0). À chaque fois que le pair change la valeur de la donnée qu'il publie (par exemple parce que l'utilisateur a tapé un nouveau message dans l'interface utilisateur), il incrémente son numéro de séquence.

En outre, chaque pair maintient deux structures de données :

- la table de voisins, qui est indexée par adresses de *socket* (des paires $(IP, Port)$), et dont chaque entrée contient un booléen indiquant si le pair est permanent (configuré au lancement) ou transitoire, et la date de dernière réception d'un paquet de la part de ce pair ;
- la table des données publiées par les autres pairs, qui est indexée par *Id* de nœud ι (il y a donc au plus une donnée par nœud), et dont chaque entrée contient une donnée (s, d) comme décrit au paragraphe 2.2 ci-dessus.

Les structures de données décrites ci-dessus sont conceptuelles : vous pouvez utiliser des structures de données équivalentes, et vous pouvez stocker des données supplémentaires dans votre pair. (Par exemple, vous pouvez décider de stocker la donnée publiée par le pair dans la table des données publiées.)

4.2 Maintenance de la liste de voisins

Lorsqu'un nœud reçoit un paquet, il commence par vérifier l'entête ; s'il est incorrect, le paquet est ignoré. Sinon, il recherche l'émetteur dans sa table de voisins. Si l'émetteur n'y est pas présent, et si la table de voisins contient déjà au moins 15 entrées, alors le paquet est ignoré (ainsi que tous les TLV qu'il contient). Si l'émetteur n'est pas présent, mais que la table de voisins contient strictement moins de 15 entrées, alors le nœud ajoute l'émetteur à sa table de voisins et le marque comme transitoire. Si l'émetteur a été ajouté à la table de voisins, ou s'il y était déjà présent, le récepteur met à jour la date de dernière réception de paquet contenue dans l'entrée de la table de voisins.

Toutes les 20 secondes environ, le nœud parcourt sa table de voisins. Si un voisin transitoire n'a pas émis de paquet depuis 70 secondes, il est éliminé de la table de voisins (un pair permanent n'est jamais éliminé).

Si la table contient moins de 5 voisins, alors le nœud envoie occasionnellement (par exemple toutes les 20 secondes) un TLV *Neighbour Request* à un voisin tiré au hasard.

Lorsqu'un nœud reçoit un TLV *Neighbour Request*, il tire au hasard une entrée de sa table de voisins et envoie un TLV *Neighbour* contenant l'adresse de cette dernière à l'émetteur du *Neighbour Request*.

Lorsqu'un nœud reçoit un TLV *Neighbour*, il envoie un TLV *Network Hash* à l'adresse contenue dans le TLV *Neighbour* (mais n'ajoute pas le nouveau voisin à sa table de voisins — il ne le fera que lorsqu'il recevra un paquet correct de la part de ce dernier).

4.3 Inondation

Chaque pair envoie toutes les 20 secondes environ (mais voir le paragraphe *Trickle* ci-dessous) un TLV *Network Hash* reflétant son idée de l'état du réseau.

Lorsqu'un nœud A reçoit un TLV *Network Hash* émis par un pair B, il compare le *network hash* émis par B à celui qu'il a calculé. Si les deux *hashes* sont identiques, il n'y a rien à faire. Dans le cas contraire (les deux *hashes* diffèrent), A envoie un TLV *Network State Request* à B.

Lorsqu'un nœud B reçoit un TLV *Network State Request*, il répond par une série de TLV *Node Hash*, un pour chacun des pairs dont il connaît l'état. Ces TLV ne sont pas forcément envoyés dans le même paquet : chaque TLV est contenu entièrement dans un seul paquet, mais la série de TLV peut être fragmentée parmi plusieurs paquets.

Lorsqu'un nœud A reçoit de B un TLV *Node Hash* décrivant le nœud d'*Id* ι avec le *hash* h, il consulte sa table de données publiées. S'il n'y a pas d'entrée pour ι, ou si une donnée pour ι existe, il envoie un TLV *Node State Request* à l'émetteur ; si par contre les *hashes* sont identiques, il n'y a rien à faire (mais voyez le paragraphe *Extensions* ci-dessous).

Lorsque A reçoit un TLV *Node State* pour le nœud d'*Id* ι, numéro de séquence s, *hash* h et donnée d, il compare les *hashes* comme dans le cas d'un TLV *Node Hash*. Si les *hashes* sont identiques, il n'y a rien à faire. Sinon, soient ι', s' et d' les données contenues dans la table de données, et h' le *hash* correspondant. Il y a deux cas principaux :

- si ι est l'*Id* du récepteur A, alors :
  - si $s \succcurlyeq s'$, alors A affecte $s \oplus 1$ (*modulo* 2^{16}) au numéro de séquence contenu dans sa table de données ;
  - sinon, $s \prec s'$, et il n'y a rien à faire ;
- sinon, ι est l'*Id* d'un autre nœud, et dans ce cas :
  - s'il n'y a pas d'entrée correspondant à ι dans la table de A, ou si $s \succ s'$, alors A stocke la donnée (ι, s, d) dans sa table de données (en éliminant l'anciennce donnée (ι', s', d') le cas échéant) ;
  - sinon, $s \preccurlyeq s'$, et il n'y a rien à faire.

Attention, tous les ordres sont *modulo* 2^{16} (paragraphe 2.1).

4.4 Gestion des erreurs

L'évaluation prendra en compte la robustesse de votre implémentation, non seulement face aux erreurs de communication mais aussi fase à des pairs boggués ou malicieux. En particulier, il est essentiel de gérer correctement le cas où un paquet annonce une longueur supérieure à la taille du datagramme qui le contient moins quatre octets, et le cas d'un TLV qui annonce une longueur qui le ferait déborder du paquet. (Dans les deux cas, il est raisonnable d'ignorer le paquet ou le TLV, peut-être après avoir émis un TLV *Warning*.)

5 Extensions

Je m'attends à ce que vous implémentiez des extensions au sujet minimal ci-dessus, et vos extensions seront évaluées avec intérêt et bienveillance. Cependant, même étendue, votre implémentation devra rester interopérable avec le sujet minimal défini ci-dessus — toutes les extensions devront rester compatibles. Si vous avez besoin d'un numéro de TLV pour votre extension, annoncez votre besoin sur la liste de diffusion et nous vous en affecterons un.

Ci-dessous quelques idées d'extensions au sujet.

Trickle* et ordonnancement des *Network Hash Les performances de la synchronisation dépendent de l'intervalle d'envoi des TLV *Network Hash*. Si cet intervalle est trop long, la synchronisation sera lente ; s'il est trop court, le protocole sera bruyant. Le protocole décrit ci-dessus suggère d'utiliser un intervalle constant de 20 secondes, ce qui me semble un bon compromis.

Une meilleure approche consiste à varier l'intervalle dynamiquement. *Trickle*, défini dans la RFC 6206, est un exemple d'algorithme qui permet de varier l'intervalle dynamiquement. Je vous propose d'utiliser une instance de *Trickle* pour chaque voisin, et d'utiliser les paramètres $I_{\mathrm{min}} = 2\,\mathrm{s}$, $I_{\mathrm{max}} = 20\,\mathrm{s}$ et $k = 1$.

Une variante de *Trickle*, due à Markus Stenberg, qui est moins bavarde lorsque la synchronisation échoue (par exemple du fait d'un lien asymétrique), consiste à ne reinitialiser à I_{min} l'intervalle de *Trickle* que lorsqu'une donnée a changé et non à chaque fois qu'on détecte une incohérence ; en d'autres termes, de n'appliquer le point 6 de la section 4.2 de la RFC 6206 non à chaque fois qu'on entend un TLV *Network Hash* incohérent, mais seulement lorsqu'une donnée a effectivement été modifiée.

Il y a sûrement d'autres algorithmes possibles. Faites attention, cependant, à ne pas créer de boucle de rétroaction — il ne faut surtout pas réagir à un TLV *Network Hash* en envoyant immédiatement un TLV *Network Hash*.

Vérification de la cohérence des *Node State* Un TLV *Node State* contient toutes les données nécessaires pour recalculer le *node hash* ainsi que le *hash* lui-même. Le protocole ci-dessus suggère de faire confiance aux TLV *Node State* reçus ; une autre approche, plus robuste mais plus gourmande en temps CPU, consiste à recalculer le *hash* et ignorer les TLV incohérents (peut-être après avoir envoyé un TLV *Warning*). Une approche intermédiaire consiste à ne recalculer et vérifier le *hash* que s'il diffère du *hash* contenu dans la table de données.

Calcul des *hashes* Le protocole est conçu pour limiter le nombre de *hashes* qu'il faut calculer : lorsqu'une donnée change, il suffit de recalculer le *hash* de celle-ci et le *hash* du réseau — il n'est pas nécessaire de recalculer tous les *hashes*, ou, pire encore, de les recalculer à chaque envoi et réception de paquet.

Les procédures proposées ci-dessus pour réagir à un TLV *Node Hash* et *Node State* inconsistants sont les plus simples que j'ai pu concevoir. Cependant, on peut faire mieux. Par exemple, si le numéro de séquence du voisin a été incrémenté, il est inutile de comparer les *hashes* — on enverra forcément un TLV *Node State Request*. Si notre numéro de séquence est supérieur à celui du voisin,

par contre, il peut être intéressant d'envoyer immédiatement un *Node State* non-solicité ; attention cependant aux boucles de rétroaction.

Agrégation Le protocole permet d'agréger plusieurs TLV dans un seul paquet. Votre implémentation agrège-t-elle les TLV de façon optimale ? Qu'est-ce que ça veut dire ? (Clairement, il y a une tension entre latence et efficacité.)

Adresses multiples Si votre implémentation tourne sur un hôte qui a plusieurs adresses (par exemple parce qu'il a plusieurs interfaces), communique-t-elle toujours avec un voisin donné en utilisant la même adresse ? Votre pair risque-t-il de devenir voisin de lui-même ? Est-ce grave ? Que se passe-t-il si c'est l'hôte du pair voisin qui a plusieurs adresses ?

Adresses locale au lien Votre pair gère-t-il correctement les pairs qui utilisent des adresses IPv6 locales au lien ? (On peut soit ignorer les paquets qui proviennent d'adresses locales au lien, soit retenir le numéro d'interface entrante quelque part. Il faut éviter d'envoyer des adresses locales au lien dans les TLV *Neighbour*, sauf peut-être aux nœuds voisins à la couche réseau.)

Découverte multicast Un TLV *Network Hash* peut être envoyé en multicast, ce qui permet de découvrir automatiquement les voisins. Je vous propose d'utiliser le port 1212 et le groupe multicast `ff12::4eeb:8d51:534e:e69b` (je l'ai tiré au hasard, il est donc globalement unique). Si vous arrivez à faire fonctionner le multicast au-delà du lien local, racontez-moi comment vous avez fait.

Contrôle de flots Lorsque votre pair envoie une suite de TLV *Node State*, limite-t-il son débit à quelque chose que le réseau peut raisonnablement accepter, ou envoie-t-il tous les *Node State* aussi vite qu'il peut ? Quel est le bon débit à utiliser ? (Il faudra probablement prendre le nombre total de pairs du réseau en compte.)

Traversée de Firewalls, de NAT En IPv6, le protocole PCP permet de traverser les *firewalls*, mais il est rarement implémenté. En IPv4, PCP ou NAT-PMP permettent de traverser les NAT, mais c'est NAT-PMP qui est généralement implémenté. N'implémentez en aucun cas uPNP, c'est un protocole mal conçu qu'il faut laisser mourir en paix. Regardez aussi les techniques de type STUN.

6 Modalités de rendu

Vous nous soumettrez une archive `.tar.gz` contenant au moins :

- le code source ;
- un fichier texte nommé `README` expliquant comment compiler et exécuter votre code (les *scripts shell* et les *makefiles* seront les bienvenus) ;

– un rapport de quelques pages au format PDF indiquant notamment comment est structuré votre programme, les choix d'implémentation que vous aurez faits, les différences entre votre solution et les algorithmes décrits dans ce document, les extensions que vous aurez implémentées, et toute information qui pourra nous aider lors de l'évaluation de votre projet (ou simplement qui est de nature à nous intéresser).

L'archive devra s'appeler *nom1-nom2*.tar.gz, et s'extraire dans un sous-répertoire *nom1-nom2* du répertoire courant. Par exemple, si vous vous appelez *Francis Crick* et *James Watson*, votre archive devra porter le nom `crick-watson.tar.gz` et son extraction devra créer un répertoire `crick-watson` contenant tous les fichiers que vous nous soumettrez.