1Groupe de travail Réseau
Request for Comments : 1144
Traduction Claude Brière de L'Isle

V. Jacobson/1/
LBL
février 1990

Compression des en-têtes TCP/IP pour les liaisons de série à basse vitesse

Statut de ce mémoire
La présente RFC est la proposition du choix d'un protocole pour la communauté de l'Internet et elle appelle à des discussions et suggestions pour son amélioration. Elle décrit une méthode de compression des en-têtes des datagrammes TCP/IP pour améliorer les performances sur les liaisons de série à faible vitesse. La motivation, la mise en œuvre et les performances de la méthode sont décrites. Le code C pour un exemple de mise en œuvre est donné pour servir de référence. La distribution du présent mémoire n'est soumise à aucune restriction.

Note : Les versions ASCII et Postscript du présent document sont toutes deux disponibles. Dans la version ASCII manquent évidemment toutes les figures et les informations codées par des variations typographiques (italiques, gras, etc.). Comme ces informations étaient, selon l'auteur, une partie essentielle du document, la version ASCII est au mieux incomplète et au pire, trompeuse. Quiconque prétend travailler avec ce protocole est vivement encouragé à se procurer la version Postscript de cette RFC.

Note 1. Ce travail a été soutenu en partie par le Ministère U.S. de l'énergie sous le contrat numéro DE-AC03-76SF00098.----------

Table des matières

1. Introduction

À mesure que s'installent dans les foyers des ordinateurs de plus en plus puissants, il y a un intérêt croissant pour l'extension de la connexion de ces ordinateurs à l'Internet. Malheureusement, cette extension révèle des problèmes complexes de tramage de niveau liaison, d'allocation d'adresses, d'acheminement, d'authentification et de performances. Au

moment de la rédaction de ce document sont menés d'actives recherches dans tous ces domaines. Le présent mémoire décrit une méthode qui a été utilisée pour améliorer les performances de TCP/IP sur des liaisons série à basse vitesse (300 à 19 200 bit/s).

La compression proposée ici est dans son esprit similaire à celle du protocole Thinwire-II décrit en [5]. Cependant, ce protocole compresse plus efficacement (l'en-tête compressé moyen est de 3 octets à comparer aux 13 de Thinwire-II) et est à la fois efficace et simple à mettre en œuvre (la mise en œuvre Unix est de 250 lignes de langage C et il faut, en moyenne, 90 µs (170 instructions) pour un MC68020 à 20 MHz pour compresser ou décompresser un paquet).

Cette compression est spécifique des datagrammes TCP/IP.

Note 2 Le lien avec TCP est plus profond qu'il n'y paraît. En plus de la compression 'connaissant' le format des en-têtes TCP et IP, certaines caractéristiques de TCP ont été utilisées pour simplifier le protocole de compression. En particulier, la livraison fiable de TCP et le modèle de conversation en flux d'octets ont été utilisés pour éliminer le besoin de toute espèce de dialogue de correction d'erreur dans le protocole (voir la Section 4).

L'auteur a étudié la compression de datagrammes UDP/IP mais il s'est révélé qu'ils sont trop peu fréquents pour qu'il vaille la peine de s'en occuper et qu'il y a trop peu de cohérence d'un datagramme à l'autre pour une bonne compression (par exemple, les interrogations de serveur de nom) ou que les en-têtes de protocole de couche supérieure dépassent de loin le coût de l'en-tête UDP/IP (par exemple, RPC/NFS de Sun). Compresser séparément la portion IP et la portion TCP du datagramme a aussi été étudié mais rejeté car cela augmente la taille moyenne d'en-tête compressé de 50 % et double la taille du code de compression et de décompression.

2. Le problème

Les services Internet auxquels on souhaite accéder par une ligne IP de série à partir de chez soi vont de la connexion interactive de type 'terminal' (par exemple, telnet, rlogin, xterm) au transfert de données en vrac (par exemple, ftp, smtp, nntp). La compression d'en-tête est motivée par le besoin d'une bonne réponse interactive. C'est-à-dire que l'efficacité de ligne d'un protocole est le ratio <données sur en-tête + données> dans un datagramme. Si un transfert efficace de données brutes est le seul objectif, il est toujours possible de faire le datagramme assez grand pour approcher une efficacité de 100 %.

Les études sur les facteurs humains [15] ont montré que la réponse interactive est perçue comme 'mauvaise' lorsque le retour de niveau inférieur (l'écho de caractère) prend plus de 100 à 200 ms. Les en-têtes de protocole interagissent de trois façons avec ce seuil :

(1) Si la ligne est trop lente, il peut être impossible de faire tenir à la fois les en-têtes et les données dans une fenêtre de 200 ms : La frappe d'un caractère résulte en l'envoi d'un paquet TCP/IP de 41 octets et en la réception d'un écho de 41 octets. La vitesse de ligne doit être d'au moins 4000 bit/s pour traiter ces 82 octets en 200 ms.

(2) Même avec une ligne assez rapide pour traiter l'écho de frappe mis en paquets (4800 bit/s ou plus) il peut y avoir une interaction indésirable entre le trafic interactif et celui des données en vrac. Pour une efficacité de ligne raisonnable, la taille du paquet de données en vrac doit faire 10 à 20 fois la taille d'en-tête. C'est-à-dire que l'unité maximum de transmission (ou MTU) de ligne devrait être de 500 à 1000 octets pour 40 octets d'en-tête TCP/IP. Même avec une mise en file d'attente par type de service pour donner la priorité au trafic interactif, un paquet telnet doit attendre que se finisse tout paquet de données en vrac en cours. En supposant le transfert des données dans seulement une direction, cette attente est en moyenne de la moitié de la MTU ou 500 ms pour une MTU de 1024 octets à 9600 bit/s.

(3) Tout support de communication a un taux maximum de signalisation, la limite de Shannon. Sur la base d'une étude de l'AT&T [2], la limite de Shannon pour une ligne téléphonique normale est d'environ 22 000 bit/s. Comme un modem bidirectionnel à 9600 bit/s fonctionne déjà à 80 % de la limite, les fabricants de modem commencent à offrir des schémas d'allocation asymétriques pour augmenter la bande passante effective : comme une ligne a rarement des quantités équivalentes de données qui s'écoulent simultanément dans les deux directions, il est possible de donner à une extrémité de la ligne plus de 11 000 bit/s soit par un multiplexage par division dans le temps d'une ligne unidirectionnelle (par exemple, le Trailblazer de Telebit) soit en offrant un 'canal inverse') basse vitesse (par exemple, le HST de USR Courier) /3/. Dans l'un et l'autre cas, le modem essaye de façon dynamique de deviner quelle extrémité de la conversation a besoin de la plus forte bande passante en supposant qu'une extrémité de la conversation est un humain (c'est-à-dire que sa demande est limitée à < 300 bit/s par la vitesse de frappe). Le facteur de multiplication par quarante de la bande passante due aux en-têtes de protocole va déjouer cette heuristique d'allocation et être cause de l'échec de ces modems.

Note 3 Voir l'excellente discussion sur les capacités de liaison téléphonique deux fils dans [1], chap. 11. En particulier, on comprend généralement que les capacités des modems à 'annulation d'écho' (comme ceux qui se conforment à la Recommandation UIT-T V.32) : l'annulation d'écho peut offrir à chaque côté d'une ligne deux fils la pleine bande passante de la ligne mais, le signal du locuteur distant s'ajoutant au bruit local, pas la totalité de la capacité de ligne. La limite de 22 kbit/s de Shannon est une limite matérielle du taux de données à travers une connexion téléphonique deux fils.

D'après ce qui précède, il est clair qu'un objectif de conception de la compression devrait être de limiter la demande de bande passante du trafic de frappe et d'accusé de réception à au plus 300 bit/s. Une vitesse de frappe maximum typique est d'environ cinq caractères par seconde /4/ qui laisse un budget de 30 - 5 = 25 caractères pour les en-têtes ou cinq octets d'en-tête par caractère frappé /5/. Cinq octets d'en-tête résolvent directement les problèmes (1) et (3) et indirectement le problème (2) : une taille de paquet de 100 à 200 octets va aisément amortir le coût de cinq octets d'en-tête et offrir à l'usager 95 à 98 % de la bande passante de la ligne pour les données. Ces paquets courts signifient peu d'interférence entre le trafic interactif et celui de données en vrac (voir au paragraphe 5.2).

Note 4 Voir [13]. Des salves de frappe ou des touches multi-caractères telles que des touches de curseur peuvent excéder ce taux moyen d'un facteur de deux à quatre. Cependant la demande de bande passante reste approximativement constante car l'algorithme de TCP Nagle [8] agrège le trafic avec un temps d'arrivée < 200 ms et le taux amélioré d'en-tête à données compense l'accélération des données.

Note 5 Une analyse similaire conduit essentiellement à la même limite de taille d'en-tête pour les paquets d'accusé de réception de transfert de données en vrac. En supposant que la MTU a été choisie pour une base de transfert de fichiers 'non obstructive' (c'est-à-dire, choisie de telle sorte que le temps de paquet soit de 200 à 400 ms --- voir à la Section 5) il peut y avoir au plus 5 paquets de données par seconde dans la direction à 'forte bande passante'. Une mise en œuvre TCP raisonnable va accuser réception au plus de tous les autres paquets de donnés de sorte qu'à 5 octets par accusé de réception la bande passante du canal inverse est de 2,5 * 5 = 12,5 octet/s.

Un objectif de conception est que le protocole de compression ne se fonde que sur des informations dont on ait la garantie qu'elles sont connues des deux extrémités d'une même liaison en série. En considérant la topologie montrée à la figure 1 où les hôtes en communication A et B sont sur des réseaux de zone locale distincts (les lignes noires en gras) et où les réseaux sont connectés par deux liaisons de série (les lignes ouvertes entre les passerelles C--D et E--F) /6/. Une possibilité de compression serait de convertir chaque conversation TCP/IP en une conversation sémantiquement équivalente dans un protocole ayant de plus petits en-têtes, par exemple, un appel X.25. Mais, à cause des hasards de l'acheminement ou de l'existence de chemins multiples, il est entièrement possible qu'une partie du trafic A--B suive le chemin A-C-D-B et qu'une partie suive le chemin A-E-F-B. De même, il est possible que le trafic A->B s'écoule suivant A-C-D-B et que le trafic B->A s'écoule selon B-F-E-A. Aucune passerelle ne peut compter voir tous les paquets d'une conversation TCP particulière et un algorithme de compression qui fonctionne pour une telle topologie ne peut pas être lié à la syntaxe de la connexion TCP.

Note 6 Noter que bien que les points d'extrémité TCP soient A et B, dans cet exemple la compression/décompression doit être faite à la passerelle de liaison série, c'est-à-dire, entre C et D et entre E et F. Comme A et B utilisent IP, ils ne peuvent pas savoir que leur chemin de communication inclut une liaison série à basse vitesse. Il est une exigence claire que la compression ne casse pas le modèle IP, c'est-à-dire, que la fonction de compression soit entre les systèmes intermédiaires et pas juste entre les systèmes d'extrémité.

Une liaison physique traitée comme deux liaisons unidirectionnelles indépendantes (une dans chaque direction) impose des exigences minimales sur la topologie, l'acheminement et le traitement en parallèle. Les extrémités de chaque liaison unidirectionnelle ont seulement à se mettre d'accord sur le ou les paquets les plus récents envoyés sur cette liaison. Donc, bien que tout schéma de compression implique un état partagé, cet état est spatialement et temporellement local et se conforme au principe de Dave Clark de partage de sort [4] : les deux extrémités peuvent seulement être en désaccord sur l'état si la ligne qui les connecte devient inopérante, auquel cas le désaccord n'a plus d'importance.

3. Algorithme de compression

3.1 Idée de base

La Figure 2 montre un en-tête typique (et la longueur minimum) de datagramme TCP/IP /7/. La taille de l'en-tête est de 40 octets : 20 octets de IP et 20 de TCP. Malheureusement, comme les protocoles TCP et IP n'ont pas été conçus par un comité, tous ces champs d'en-tête ont un objet utile et il n'est pas possible de simplement en omettre certains au nom de l'efficacité.

Note 7 Les protocoles TCP et IP et les en-têtes de protocole sont décrits dans [10] et [11].

Cependant, TCP établit les connexions et, normalement, des dizaines ou des centaines de paquets sont échangés sur chaque connexion. Combien d'information par paquet va vraisemblablement rester constant sur la vie d'une connexion ? La moitié --- le champ ombré sur la Figure 3. Donc, si envoyeur et receveur gardent trace des connexions actives /8/ et si le receveur garde une copie de l'en-tête provenant du dernier paquet qu'il a vu provenant de chaque connexion, l'envoyeur obtient un facteur de compression de deux en envoyant seulement un petit (≤ 8 bits) identifiant de connexion avec les 20 octets qui changent et en laissant le receveur remplir les 20 octets fixes provenant de l'en-tête économisé.

Note 8 Le tuplet de 96 bit <adresse de source, adresse de destination, accès de source, accès de destination> identifie de façon univoque une connexion TCP.

On peut grappiller quelques octets de plus en notant que tout protocole raisonnable de tramage de niveau liaison va dire au receveur la longueur d'un message reçu de sorte que la longueur totale (les octets 2 et 3) est redondante. Mais alors la somme de contrôle d'en-tête (octets 10 et 11) qui protège les sauts individuels du traitement d'un en-tête IP corrompu est essentiellement la seule partie de l'en-tête IP qui est envoyée. Il semble plutôt stupide de protéger la transmission d'informations qui ne seront pas transmises. Aussi, le receveur peut vérifier la somme de contrôle d'en-tête lorsque l'en-tête est en fait envoyé (c'est-à-dire, dans un datagramme non compressé) mais, pour les datagrammes compressés, les régénérer en local au même moment que le reste de l'en-tête IP est régénéré /9/.

Note 9 La somme de contrôle d'en-tête IP n'est pas une somme de contrôle de bout en bout dans le sens de [14] : la mise à jour de la durée de vie force la somme de contrôle IP à être recalculée à chaque bond. L'auteur a eu la désagréable expérience personnelle des conséquences de la violation de l'argument de bout en bout dans [14] et ce protocole fait attention à passer la somme de contrôle TC de bout en bout non modifiée. Voir la Section 4.

Cela laisse 16 octets d'informations d'en-tête à envoyer. Tous ces octets vont vraisemblablement changer durant la vie de la conversation mais ils ne changent pas tous en même temps. Par exemple, durant un transfert de données FTP, seul l'identifiant de paquet, le numéro de séquence et la somme de contrôle changent dans la direction envoyeur-->receveur, et seul l'identifiant de paquet, l'accusé de réception, la somme de contrôle et, éventuellement, la fenêtre, changent dans la direction receveur-->envoyeur. Avec une copie du dernier paquet envoyé pour chaque connexion, l'envoyeur peut se représenter quels champs changent dans le paquet en cours puis envoyer un gabarit binaire qui indique ce qui a changé suivi par les champs changés /10/.

Note 10 C'est approximativement Thinwire-I dans [5]. Une légère modification est de faire un codage du delta où l'envoyeur retranche le paquet précédent du paquet en cours (en traitant chaque paquet comme une matrice d'entiers de 16 bits) puis envoie un gabarit binaire de 20 bits qui indique les différences non zéro suivies de ces différences. Si des conversations distinctes sont séparées, c'est un schéma de compression très efficace (par exemple, normalement des en-têtes de 12-16 octets) qui n'implique pas que le compresseur connaisse les détails de la structure du paquet. Des variations sur ce thème ont été utilisées, avec succès, pendant un certain nombre d'années (par exemple, le protocole de liaison en série du routeur Proteon [3]).

Si l'envoyeur n'envoie que les champs qui diffèrent, le schéma ci-dessus diminue la taille moyenne de l'en-tête jusqu'à environ dix octets. Cependant, il vaut la peine de regarder comment changent les champs : l'identifiant de paquet vient normalement d'un compteur qui est incrémenté de un pour chaque paquet envoyé. C'est-à-dire, la différence entre les identifiants du paquet en cours et celui du précédent paquet devrait être un petit entier positif, normalement <256 (un octet) et fréquemment = 1. Pour les paquets qui proviennent du côté envoyeur d'un transfert de données, le numéro de séquence dans le paquet en cours sera le numéro de séquence du précédent paquet plus la quantité de données dans le précédent paquet (en supposant que les paquets sont arrivés dans l'ordre). Comme les paquets IP peuvent faire au plus 64 k, le changement du numéro de séquence doit être < 2^16 (deux octets). Ainsi, si les différences dans les champs qui changent sont envoyées plutôt que les champs eux-mêmes, ce sont trois ou quatre octets par paquet qui peuvent être économisés.

Cela nous amène à la cible d'un en-tête de cinq octets. Reconnaître un couple de cas particuliers nous amènera des en-têtes de trois octets pour les deux cas les plus courants --- le trafic de frappe interactive et le transfert de données en vrac --- mais le schéma de base de compression est le codage différentiel développé ci-dessus. Étant donné que cet exercice intellectuel suggère qu'il est possible d'obtenir des en-têtes de cinq octets, il semble raisonnable d'étoffer les détails manquants et de mettre quelque chose en œuvre.

3.2 Détails sordides

3.2.1 Vue d'ensemble

La Figure 4 montre un diagramme du logiciel de compression. Le système de réseautage appelle un pilote de sortie SLIP avec un paquet IP à envoyer sur la ligne de série. Le paquet passe à travers un compresseur qui vérifie si le protocole est TCP. Les paquets non TCP et les paquets TCP 'non compressibles' (décrits plus loin) sont juste marqués de TYPE_IP et

passés à un trameur. Les paquets TCP compressibles sont recherchés dans une matrice d'en-têtes de paquet. Si une connexion correspondante est trouvée, le paquet entrant est compressé, l'en-tête (non compressé) du paquet est copié dans la matrice, et un paquet de type COMPRESSED_TCP est envoyé au trameur. Si aucune correspondance n'est trouvée, la plus ancienne entrée dans la matrice est éliminée, l'en-tête du paquet est copié à sa place, et un paquet de type UNCOMPRESSED_TCP est envoyé au trameur. (Un paquet UNCOMPRESSED_TCP est identique au paquet IP original excepté le champ Protocole IP qui est remplacé par un numéro de connexion --- un indice dans la matrice des en-têtes de paquet par connexion sauvegardés. C'est comme cela que l'envoyeur (re)synchronise le receveur et l'alimente avec le premier paquet, non compressé, d'une séquence de paquets compressés.)

Le trameur est chargé de communiquer les données, le type et les limites du paquet (afin que le décompresseur puisse savoir combien d'octets sont sortis du compresseur). Comme la compression est un codage différentiel, le trameur ne doit pas réordonner les paquets (c'est rarement un problème sur une seule ligne de série). Il doit aussi fournir une bonne détection d'erreur et, si les numéros de connexion sont compressés, il doit fournir une indication d'erreur au décompresseur (voir la Section 4) /11/.

Note 11. Le tramage de niveau liaison sort du domaine d'application du présent document. Tout tramage qui assure les facilités énumérées dans ce paragraphe devrait être adéquat pour le protocole de compression. Cependant, l'auteur encourage les mises en œuvre potentielles à voir dans [9] une proposition de norme, le tramage SLIP.

Le décompresseur fait un 'changement' sur le type des paquets entrants : pour le TYPE_IP, le paquet est simplement passé. Pour UNCOMPRESSED_TCP, le numéro de connexion est extrait du champ Protocole IP et IPPROTO_TCP est restauré, puis le numéro de connexion est utilisé comme un indice dans la matrice des en-têtes TCP/IP sauvegardés du receveur et l'en-tête du paquet entrant est copié dans l'emplacement indexé. Pour COMPRESSED_TCP, le numéro de connexion est utilisé comme un indice matriciel pour obtenir l'en-tête TCP/IP du dernier paquet provenant de cette connexion, les informations dans le paquet compressé sont utilisées pour mettre à jour cet en-tête, puis un nouveau paquet est construit qui contient l'en-tête qui est maintenant l'en-tête en cours provenant de la matrice, enchaîné avec les données provenant du paquet compressé.

Noter que la communication est unidirectionnelle --- aucun flux d'informations dans la direction décompresseur à compresseur. En particulier, cela implique que le décompresseur s'appuie sur les retransmissions TCP pour corriger l'état sauvegardé dans le cas d'erreurs en ligne (voir la Section 4).

3.2.2 Format de paquet compressé

La Figure 5 montre le format d'un paquet TCP/IP compressé. Il y a un gabarit de changement qui identifie quels sont les champs dont il est attendu qu'ils changent en fonction des paquets réellement changés, un numéro de connexion afin que le receveur puisse localiser la copie sauvegardée du dernier paquet pour cette connexion TCP, la somme de contrôle TCP non modifiée afin que la vérification d'intégrité des données de bout en bout soit encore valide, puis pour chaque bit établi dans le gabarit de changement, la quantité de champs changés associée. (Les champs facultatifs, contrôlés par le gabarit, sont entourés de lignes en pointillé dans la figure.) Dans tous les cas, le bit est établi si le champ associe est présent et à zéro si le champ est absent /12/.

Note 12. Le bit 'P' sur la figure est différent des autres : c'est une copie du bit 'PUSH' provenant de l'en-tête TCP. 'PUSH' est un curieux anachronisme considéré comme indispensable par certains membres de la communauté Internet. Comme PUSH peut changer (et il le fait) dans n'importe quel datagramme, un schéma de compression préservant les informations doit le passer explicitement.

Comme les deltas dans le numéro de séquence, etc., sont habituellement petits, en particulier si les lignes directrices de réglage de la section 5 sont respectées, tous les nombres sont codés dans un schéma de longueur variable qui, en pratique, traite la plus grande partie du trafic avec huit bits : un changement de un à 255 est représenté sur un octet. Zéro est improbable (un changement de zéro n'est jamais envoyé) de sorte qu'un octet de zéro signale une extension : les deux octets suivants sont respectivement le MSB et le LSB d'une valeur de 16 bits. Les nombres supérieurs à 16 bits forcent l'envoi d'un paquet non compressé. Par exemple, le décimal 15 est codé par l'hexadécimal 0f, 255 par ff, 65 534 par 00 ff fe, et zéro par 00 00 00. Ce schéma met en paquet et décode très efficacement : le cas usuel pour aussi bien le codage que le décodage exécute trois instructions sur un MC680x0.

Les nombres envoyés pour le numéro de séquence et les accusés de réception TCP sont la différence /13/ entre la valeur en cours et la valeur dans le paquet précédent (un paquet non compressé est envoyé si la différence est négative ou supérieure à 64 k). Le nombre envoyé pour la fenêtre est aussi la différence entre la valeur en cours et la valeur précédente. Cependant, des changements positifs ou négatifs sont admis car la fenêtre est un champ de 16 bits. Le pointeur urgent du paquet est envoyé si URG est établi (un paquet non compressé est envoyé si le pointeur urgent change mais URG n'est pas établi). Pour l'identifiant de paquet, le nombre envoyé est la différence entre la valeur en cours et la précédente. Cependant,

à la différence du reste des champs compressés, le changement supposé lorsque I est à zéro est un, pas zéro.
Note 13 Toutes les différences sont calculées à l'aide d'une arithmétique de compléments à deux.

Il y a deux importants cas particuliers :

(1) Le numéro de séquence et l'accusé de réception changent tous deux de la quantité de données dans le dernier paquet ; il n'y a pas de changement de la fenêtre ou de URG.

(2) Le numéro de séquence change de la quantité de données dans le dernier paquet, aucun accusé de réception ou fenêtre ou URG ne change.

(1) est le cas du trafic terminal avec écho. (2) est le côté envoyeur de trafic terminal sans écho ou un transfert de données unidirectionnel. Certaines combinaisons des bits S, A, W et U du gabarit de changement sont utilisés pour signaler ces cas particuliers. 'U' (données urgentes) est rare, de sorte que des combinaisons improbables sont S W U (utilisé pour le cas 1) et S A W U (utilisé pour le cas 2). Pour éviter des ambiguïtés, un paquet non compressé est envoyé si les changements réels dans un paquet sont S * W U.

Comme la connexion 'active' change rarement (par exemple, un usager va frapper pendant plusieurs minutes dans une fenêtre telnet avant de passer à une fenêtre différente) le bit C permet d'élider le numéro de connexion. Si C est à zéro, la connexion est supposée être la même que lors du dernier paquet compressé ou non compressé. Si C est à un, le numéro de connexion est dans l'octet qui suit immédiatement le gabarit de changement /14/.

Note 14 Le numéro de connexion est limité à un octet, c'est-à-dire, 256 connexions TCP simultanément actives. En presque deux ans de fonctionnement, l'auteur n'a jamais vu un cas où plus de seize états de connexion auraient été utiles (même dans un cas où la liaison SLIP était utilisée comme passerelle derrière un multiplexeur terminal très actif à 64 accès). Donc, cela ne semble pas être une restriction très significative et permet d'utiliser le champ Protocole dans les paquets UNCOMPRESSED_TCP pour le numéro de connexion, ce qui simplifie le traitement de ces paquets.

De ce qui précède, il est probablement évident que le trafic terminal compressé ressemble habituellement à (en hexadécimal) : 0B c c d, où le 0B indique le cas (1), c c est la somme de contrôle TC de deux octets et d est le caractère tapé. Les commandes de vi ou emacs, ou les paquets dans la direction du transfert des données d'un 'put' ou 'get' FTP ressemblent à 0F c c d ... , et les accusés de réception pour ce FTP ressemblent à 04 c c a, où a est la quantité de données dont il est accusé réception /15/.

Note 15 Il est aussi évident que le gabarit de changement change peu fréquemment et pourrait souvent être élidé. En fait, on peut faire légèrement mieux en sauvegardant le dernier paquet compressé (il peut être au plus de 16 octets de sorte que ce n'est pas beaucoup d'état supplémentaire) et en vérifiant pour voir si il y en a (excepté la somme de contrôle TC) qui ont changé. Si non, envoyer un type de paquet qui signifie 'TCP compressé, le même que la dernière fois' et un paquet contenant seulement la somme de contrôle et les données. Mais, comme l'amélioration est au plus de 25 %, l'ajout de complexité et d'état ne semble pas justifié. Voir l'Appendice C.

3.2.3 Traitement au compresseur

Le compresseur est invoqué avec le paquet IP à traiter et la structure d'état de compression pour la ligne série sortante. Il retourne un paquet prêt pour le tramage final et le 'type' de niveau liaison de ce paquet.

Comme on l'a noté au paragraphe précédent, le compresseur convertit chaque paquet d'entrée en un paquet TYPE_IP, UNCOMPRESSED_TCP ou COMPRESSED_TCP. Un paquet TYPE_IP est une copie non modifiée /16/ du paquet d'entrée et son traitement ne change en aucune façon l'état du compresseur.

Note 16 Il n'est nécessaire (ou souhaitable) de dupliquer réellement le paquet d'entrée pour aucun des trois types de résultat. Noter que le compresseur ne peut pas augmenter la taille d'un datagramme. Comme le montre le code donné à l'Appendice A, le protocole peut être mis en œuvre de telle sorte que toutes les modifications d'en-tête soient faites 'en place'.

Un paquet UNCOMPRESSED_TCP est identique au paquet d'entrée excepté le champ Protocole IP (octet 9) qui est changé de '6' (protocole TCP) en un numéro de connexion. De plus, l'intervalle d'état associé au numéro de connexion est mis à jour avec une copie des en-têtes IP et TCP du paquet d'entrée et le numéro de connexion est enregistré comme la dernière connexion envoyée sur cette ligne série (pour la compression C décrite ci-dessous).

Un paquet COMPRESSED_TCP contient les données, s'il en est, provenant du paquet d'origine mais les en-têtes IP et TCP

sont complètement remplacés par un nouvel en-tête compressé. L'intervalle d'état de connexion et la dernière connexion envoyés sont mis à jour par le paquet d'entrée exactement comme pour un paquet UNCOMPRESSED_TCP.

La procédure de décision du compresseur est :

- Si le paquet n'est pas au protocole TCP, l'envoyer comme TYPE_IP.

- Si le paquet est un fragment IP (c'est-à-dire, soit le champ Décalage de fragment est non zéro, soit le bit Fragments à venir est établi) l'envoyer comme TYPE_IP./17/

Note 17 Seul le premier fragment contient l'en-tête TCP de sorte que la vérification du décalage de fragment est nécessaire. Le premier fragment peut contenir un en-tête TCP complet et donc pourrait être compressé. Cependant la vérification que c'est un en-tête TCP complet ajoute pas mal de code et, étant donné les arguments de [6], il semble raisonnable d'envoyer tous les fragments IP non compressés.

Si un des bits de contrôle TCP, SYN, FIN ou RST, est établi ou si le bit ACK est à zéro, considérer le paquet comme non compressible et l'envoyer comme TYPE_IP /18/.

Note 18 Le test ACK est redondant car une mise en œuvre conforme à la norme doit établir ACK dans tous les paquets sauf le paquet SYN initial. Cependant, la vérification ne coûte rien et évite de transformer un paquet bogué en paquet valide. Les paquets SYN ne sont pas compressés parce que seule la moitié d'entre eux contient un champ ACK valide et ils contiennent habituellement une option TCP (taille maximale de segment) que les paquets suivants n'ont pas. Donc, le paquet suivant va être envoyé non compressé parce que la longueur d'en-tête TCP a changé et que l'envoi du SYN comme UNCOMPRESSED_TCP au lieu de TYPE_IP n'apporterait rien. La décision de ne pas compresser les paquets FIN est discutable. En faisant le décompte du dispositif dans l'Appendice B.1, il y a un bit libre dans l'en-tête qui pourrait être utilisé pour communiquer le fanion FIN. Cependant, comme les connexions tendent à durer pendant de nombreux paquets, il a semblé déraisonnable de dédier un bit entier à un fanion qui ne va apparaître qu'une fois dans toute la durée de vie de la connexion.

Si un paquet est retenu par les vérifications ci-dessus, il sera envoyé soit comme UNCOMPRESSED_TCP, soit comme COMPRESSED_TCP :

- Si aucun état de connexion ne peut être trouvé qui corresponde aux adresses IP de source et destination et accès TCP du paquet, un état est réclamé (qui devrait probablement être le plus ancien utilisé) et un paquet UNCOMPRESSED_TCP est envoyé.

- Si un état de connexion est trouvé, l'en-tête de paquet qu'il contient est confronté au paquet en cours pour s'assurer qu'il n'y a pas de changement inattendu. (Par exemple, que tous les champs ombrés de la Figure 3 sont les mêmes). Les champs Protocole IP, Décalage de fragment, Fragments à venir, SYN, FIN et RST ont été vérifiés ci-dessus et les adresses et accès de source et destination ont été vérifiés au titre de la localisation de l'état. De sorte que les champs restants sont Version du protocole, Longueur d'en-tête, Type de service, Ne pas fragmenter, Durée de vie, Décalage des données, Options IP (s'il en est) et Options TCP (s'il en est). Si un de ces champs diffère entre les deux en-têtes, un paquet UNCOMPRESSED_TCP est envoyé.

Si tous les champs 'inchangés' correspondent, on fait une tentative de compression du paquet en cours :

- Si le fanion URG est établi, le champ Données urgentes est codé (noter qu'il peut être zéro) et le bit U est établi dans le gabarit de changement. Malheureusement, si URG est à zéro, le champ Données urgentes doit être confronté à celui du paquet précédent et, si il change, un paquet UNCOMPRESSED_TCP est envoyé. ('Données urgentes' ne devrait pas changer lorsque URG est à zéro mais [11] ne l'exige pas.)

- La différence entre le champ Fenêtre des paquets en cours et précédent est calculée et, si elle est différente de zéro, est codée et le bit W est établi dans le gabarit de changement.

- La différence entre les champs Acc est calculée. Si le résultat est inférieur à zéro ou supérieur à 2^16 - 1, un paquet UNCOMPRESSED_TCP est envoyé /19/. Autrement, si le résultat est non zéro, il est codé et le bit A est établi dans le gabarit de changement.

Note 19 les deux tests peuvent être combinés en un seul des 16 bits de poids fort de la différence qui est non zéro.

- La différence entre les champs Numéro de séquence est calculée. Si le résultat est moins de zéro ou plus de 2^16 - 1, un paquet UNCOMPRESSED_TCP est envoyé /20/. Autrement, si le résultat est non zéro, il est codé et le bit S est établi dans le gabarit de changement.

Note 20 Un changement de numéro de séquence négatif indique probablement une retransmission. Comme cela peut être dû à l'abandon d'un paquet par le décompresseur, un paquet non compressé est envoyé pour resynchroniser le décompresseur (voir la Section 4).

Une fois que les changement U, W, A et S ont été déterminés, les cas particuliers de codages peuvent être vérifiés :

- Si U, S et W sont établis, les changements correspondent à un des cas particuliers de codage. Envoyer un paquet UNCOMPRESSED_TCP.

- Si seul S est établi, vérifier que le changement est égal à la quantité de données d'usager dans le dernier paquet. C'est-à-dire, soustraire les longueurs d'en-tête TCP et IP du champ Longueur totale du dernier paquet et comparer le résultat au changement de S. Si ils sont les mêmes, régler le gabarit de changement à SAWU (le cas particulier pour 'transfert de données unidirectionnel') et éliminer le changement de numéro de séquence codé (le décompresseur peut le reconstruire car il connaît la longueur totale et la longueur d'en-tête du dernier paquet).

- Si seuls S et A sont établis, vérifier si les deux ont changé de la même quantité et si cette quantité est celle des données d'usager dans le dernier paquet. S'il en est ainsi, régler le gabarit de changement à SWU (le cas particulier pour le trafic 'écho interactif') et éliminer les changements codés.

- Si rien n'a changé, vérifier si ce paquet n'a pas de données d'usager (auquel cas, il est probablement un accusé de réception dupliqué ou une sonde de fenêtre) ou si le précédent paquet contenait des données d'usager (ce qui signifie que ce paquet est une retransmission sur une connexion sans traitement en parallèle). Dans l'un et l'autre de ces cas, envoyer un paquet UNCOMPRESSED_TCP.

Finalement, l'en-tête TCP/IP sur le paquet sortant est remplacé par un en-tête compressé:

- Le changement dans l'identifiant de paquet est calculé et, si il n'est pas un /21/, la différence est codée (noter qu'il peut être zéro ou négatif) et le bit I est établi dans le gabarit de changement.

Note 21 Noter que ici, l'essai est par rapport à un, et non zéro. L'identifiant de paquet est normalement incrémenté de un pour chaque paquet envoyé de sorte qu'un changement de zéro est très improbable. Un changement de un est probable : il survient durant toute période où le système d'origine a une activité sur une seule connexion.

- Si le bit PUSH est établi dans le datagramme d'origine, le bit P est établi dans le gabarit de changement.

- Les en-têtes IP et TCP du paquet sont copiés dans l'intervalle d'état de connexion.

- Les en-têtes IP et TCP du paquet sont éliminés et un nouvel en-tête est ajouté devant le paquet, consistant en (en ordre inverse) :
- les changements codés accumulés,
- la somme de contrôle TC (si le nouvel en-tête va être construit 'en place', la somme de contrôle peut avoir été réécrite et devra être tirée de la copie de l'en-tête dans l'état de connexion ou sauvegardée temporairement avant que l'en-tête original soit éliminé),
- le numéro de connexion (si il est différent du dernier envoyé sur cette ligne série). Cela signifie aussi que la dernière connexion envoyée de la ligne doit être réglée au numéro de connexion et que le bit C est établi dans le gabarit de changement,
- le gabarit de changement.

À ce point, le paquet TCP compressé est passé au trameur pour transmission.

3.2.4 Traitement au décompresseur

À cause du modèle de communication unidirectionnel, le traitement au décompresseur est beaucoup plus simple qu'au compresseur --- toutes les décisions ont été prises et le décompresseur fait simplement ce que le compresseur a dit de faire.

Le décompresseur est invoqué avec le paquet entrant /22/, la longueur et le type du paquet et la structure d'état de compression pour la liaison série entrante. Un paquet IP (éventuellement reconstruit) sera retourné.

Note 22 On suppose que le tramage de niveau liaison a été retiré par ce point et le paquet et la longueur n'incluent pas les octets de type ou de tramage.

Le décompresseur peut recevoir quatre types de paquet : les trois générés par le compresseur et un pseudo-paquet TYPE_ERROR généré lorsque le trameur récepteur détecte une erreur /23/. La première étape est une 'commutation' du type de paquet:

Note 23 Aucune donnée n'a besoin d'être associée au paquet TYPE_ERROR. Il existe de telle sorte que le trameur receveur peut dire au décompresseur qu'il peut y avoir un trou dans le flux de données. Le décompresseur utilise cela comme un signal que les paquets devraient être rejetés jusqu'à ce qu'il en arrive un avec un numéro de

connexion explicite (le bit C établi). Voir à la dernière partie du paragraphe 4.1 un exposé sur les raisons qui rendent cela nécessaire.

- Si le paquet est TYPE_ERROR ou d'un type non reconnu, un fanion 'rejet' est établi dans l'état pour forcer les paquets COMPRESSED_TCP à être éliminés jusqu'à ce qu'en arrive un avec le bit C établi ou un paquet UNCOMPRESSED_TCP. Rien n'est retourné (un paquet nul).

- Si le paquet est TYPE_IP, une copie non modifiée en est retournée et l'état n'est pas modifié.

- Si le paquet est UNCOMPRESSED_TCP, l'indice d'état provenant du champ Protocole IP est vérifié /24/. Si il est illégal, le fanion rejet est établi et rien n'est retourné. Autrement, le fanion rejet est mis à zéro, l'indice est copié sur le champ Dernière connexion reçue de l'état, une copie du paquet entrant est faite /25/, le numéro de protocole TCP est restauré dans le champ Protocole IP, l'en-tête du paquet est copié dans l'intervalle d'état indiqué, puis la copie du paquet est retournée.

Note 24. Les indices d'état suivent la convention du langage C et fonctionnent de 0 à N - 1, où $0 < N \leq 256$ est le nombre d'intervalles d'état disponibles.

Note 25. Comme avec le compresseur, le code peut être structuré de telle façon qu'aucune copie ne soit faite et que toutes les modifications soient faites en place. Cependant, comme le paquet résultant peut être plus grand que le paquet en entrée, 128 octets d'espace libre doivent être laissés devant la mémoire tampon de paquet d'entrée pour laisser de la place pour ajouter l'en-tête TCP/IP.

Si le paquet n'a pas été traité ci-dessus, c'est un COMPRESSED_TCP et un nouvel en-tête TCP/IP doit être synthétisé à partir des informations dans le paquet plus l'en-tête du dernier paquet dans l'intervalle d'état. D'abord, le numéro de connexion explicite ou implicite est utilisé pour localiser l'intervalle d'état :

- Si le bit C bit est établi dans le gabarit de changement, l'indice d'état est vérifié. Si il est illégal, le fanion rejet est établi et rien n'est retourné. Autrement, la dernière connexion reçue est réglée à l'indice d'état du paquet et le fanion rejet est mis à zéro.

- Si le bit C est à zéro et si le fanion rejet est établi, le paquet est ignoré et rien n'est retourné.

À ce point, la dernière connexion reçue est l'indice de l'intervalle d'état approprié et le ou les premiers octets du paquet compressé (le gabarit de changement et, éventuellement, l'indice de connexion) ont été consommés. Comme l'en-tête TCP/IP dans l'intervalle d'état doit finir par refléter le paquet nouvellement arrivé, il est plus simple d'appliquer les changements du paquet à cet en-tête puis de construire le paquet résultant à partir de cet en-tête enchaîné avec les données provenant du paquet d'entrée. (Dans la description qui suit, 'en-tête sauvegardé' est utilisé comme abréviation pour 'en-tête TCP/IP sauvegardé dans l'intervalle d'état'.)

- Les deux prochains octets dans le paquet entrant sont la somme de contrôle TC. Ils sont copiés dans l'en-tête sauvegardé.

- Si le bit P est établi dans le gabarit de changement, le bit TCP PUSH est établi dans l'en-tête sauvegardé. Autrement, le bit PUSH est mis à zéro.

- Si les quatre bits de moindre poids (S, A, W et U) du gabarit de changement sont tous établis (le cas particulier 'données unidirectionnelles') la quantité de données d'usager dans le dernier paquet est calculée en soustrayant les longueurs d'en-tête TCP et IP de la longueur IP totale dans l'en-tête sauvegardé. Cette quantité est alors ajoutée au numéro de séquence TCP dans l'en-tête sauvegardé.

- Si S, W et U sont établis et si A est à zéro (le cas particulier du 'trafic terminal') la quantité de données d'usager dans le dernier paquet est calculée et ajoutée aux deux champs Numéro de séquence TCP et Acc dans l'en-tête sauvegardé.

- Autrement, les bits du gabarit de changement sont interprétés individuellement dans l'ordre où le compresseur les a rangés :

    - Si le bit U est établi, le bit URG TCP est établi dans l'en-tête sauvegardé et le ou les octets suivants du paquet entrant sont décodés et introduits dans le pointeur Urgent TCP. Si le bit U est à zéro, le bit TCP URG est mis à zéro.

    - Si le bit W est établi, le ou les prochains octets du paquet entrant sont décodés et ajoutés au champ Fenêtre TCP de l'en-tête sauvegardé.

    - Si le bit A est établi, le ou les prochains octets du paquet entrant sont décodés et ajoutés au champ Acc TCP de

l'en-tête sauvegardé.

- Si le bit S est établi, le ou les prochains octets du paquet entrant sont décodés et ajoutés au champ Numéro de séquence TCP de l'en-tête sauvegardé.

- Si le bit I est établi dans le gabarit de changement, le ou les prochains octets du paquet entrant sont décodés et ajoutés au champ Identifiant IP du paquet sauvegardé. Autrement, un est ajouté à l'identifiant IP.

À ce point, toutes les informations d'en-tête provenant du paquet entrant ont été consommées et seules restent les données. La longueur des données restantes est ajoutée à la longueur des en-têtes IP et TCP sauvegardés et le résultat est mis dans le champ Longueur totale IP sauvegardé. L'en-tête IP sauvegardé est maintenant à jour de sorte que sa somme de contrôle est recalculée et mémorisée dans le champ Somme de contrôle IP. Finalement, un datagramme de résultat consistant en l'en-tête sauvegardé enchaîné avec les données entrantes restantes est construit et retourné.

4. Traitement des erreurs

4.1 Détection des erreurs

D'après l'expérience de l'auteur, les connexions téléphoniques sont particulièrement enclines aux erreurs de données. Ces erreurs interagissent avec la compression de deux façons différentes :

Il y a d'abord l'effet local d'une erreur dans un paquet compressé. Toute détection d'erreur se fonde sur la redondance et la compression a écrasé la plupart des redondances dans les en-têtes TCP et IP. En d'autres termes, le décompresseur va heureusement transformer du bruit de ligne aléatoire en paquet TCP/IP parfaitement valide /26/.

Note 26 Modulo la somme de contrôle TC.

On pourrait s'appuyer sur la somme de contrôle TC pour détecter les paquets compressés corrompus mais, malheureusement, certaines erreurs assez probables ne seront pas détectées. Par exemple, la somme de contrôle TC va souvent ne pas détecter deux erreurs d'un seul bit séparées par 16 bits. Pour un modem V.32 qui signale à 2400 bauds avec 4 bits/baud, toute ligne qui dure plus de 400 µs. va corrompre 16 bits. Selon [2], des trous dans les lignes téléphoniques résidentielles allant jusqu'à 2 ms sont probables.

La façon correcte de traiter ce problème est d'assurer la détection d'erreur au niveau du tramage. Comme le tramage (au moins en théorie) peut être ajusté aux caractéristiques d'une liaison particulière, la détection peut être aussi légère ou lourde qu'il est approprié pour cette liaison /27/. Comme la détection d'erreur de paquet est faite au niveau du tramage, le décompresseur suppose simplement qu'il va avoir une indication que le paquet en cours a été reçu avec des erreurs. (Le décompresseur ignore toujours (élimine) un paquet avec des erreurs. Cependant, l'indication est nécessaire pour empêcher l'erreur d'être propagée --- voir ci-dessous.)

Note 27 Alors que la détection d'erreur appropriée dépend de la liaison, le CRC du CCITT utilisé dans [9] tient un excellent équilibre entre facilité de calcul et détection d'erreur robuste pour une grande variété de liaisons, en particulier aux relativement petites tailles de paquet nécessaires pour une bonne réponse interactive. Donc, pour les besoins de l'interopérabilité, le tramage de [9] devrait être utilisé sauf si il y a une raison vraiment impérieuse pour faire autrement.

La politique 'd'élimination des paquets erronés' donne lieu à la seconde interaction d'erreurs et de compression. Considérons la conversation suivante :

original	**envoyé**	**reçu**	**reconstruit**
1: A	1: A	1: A	1: A
2: BC	1, BC	1, BC	2: BC
4: DE	2, DE	---	---
6: F	2, F	2, F	4: F
7: GH	1, GH	1, GH	5: GH

(Chaque entrée ci-dessus a la forme 'numéro de séquence de début : données envoyées' ou 'changement de numéro de séquence ? données envoyées'.) La première chose envoyée est un paquet non compressé, suivi par quatre paquets compressés. Le troisième paquet prend une erreur et est éliminé. Pour reconstruire le quatrième paquet, le receveur applique le changement de numéro de séquence provenant d'un paquet compressé entrant au numéro de séquence du dernier paquet reçu correctement, le paquet deux, et génère un numéro de séquence incorrect pour le paquet quatre. Après l'erreur, tous les numéros de séquence des paquets reconstruits seront erronés, avec un décalage de la quantité de données du paquet manquant /28/.

Note 28 C'est un exemple d'un problème général avec les codages différentiels ou delta connus sous le nom de 'DC à perte'.

Sans quelque sorte de vérification, l'erreur précédente va avoir pour résultat que le receveur va perdre de façon invisible deux octets provenant du milieu du transfert (comme le décompresseur régénère les numéros de séquence, les paquets qui contiennent F et GH arrivent au TCP du receveur avec exactement le numéro de séquences qu'ils auraient eu si le paquet DE n'avait jamais existé). Bien que certaines conversations TCP puissent survivre à des données manquantes /29/, ce n'est pas une pratique à encourager. Heureusement, la somme de contrôle TC, comme elle est une simple somme du contenu du paquet incluant les numéros de séquence, détecte 100 % de ces erreurs. Par exemple, la somme de contrôle calculée du receveur pour les deux derniers paquets ci-dessus diffère toujours de deux de la somme de contrôle du paquet.

Note 29 De nombreux gestionnaires de système prétendent que les trous dans un flux NNTP sont plus précieux que les données.

Malheureusement, il existe un moyen pour que la protection de la somme de contrôle TC décrite ci-dessus échoue si les changements dans un paquet compressé entrant sont appliqués à la mauvaise conversation : considérons deux conversations actives C1 et C2 et un paquet provenant de C1 suivi par deux paquets provenant de C2. Comme le numéro de connexion ne change pas, il est omis du second paquet C2. Mais, si le premier paquet C2 est reçu avec une erreur de CRC, le second paquet C2 sera à tort considéré comme le prochain paquet dans C1. Comme la somme de contrôle de C2 est un nombre aléatoire par rapport aux numéros de séquence de C2, il y a au moins une probabilité de 2^-16 que ce paquet soit accepté par le receveur TCP C1 /30/. Pour empêcher cela, après une indication d'erreur de CRC provenant du trameur, le receveur élimine les paquets jusqu'à ce qu'il reçoive soit un paquet COMPRESSED_TCP avec le bit C établi, soit un paquet UNCOMPRESSED_TCP. C'est-à-dire que les paquets sont éliminés jusqu'à ce que le receveur obtienne un numéro de connexion explicite.

Note 30 Avec le pire cas de trafic, cette probabilité se traduit en une erreur indétectée toutes les trois heures sur une ligne à 9600 bauds avec un taux d'erreur de 30 %).

Pour résumer ce paragraphe, il y a deux différents types d'erreurs : la corruption par paquet et la perte de synchronisation par conversation. Le premier type est détecté chez le décompresseur à partir d'une erreur de CRC de niveau liaison, la seconde chez le receveur TCP à partir d'une somme de contrôle TC (garantie) invalide. La combinaison de ces deux mécanismes indépendants assure que les paquets erronés sont éliminés.

4.2 Récupération d'erreur

Le paragraphe précédent notait qu'après une erreur de CRC le décompresseur va introduire des erreurs de somme de contrôle TC dans chaque paquet non compressé. Bien que les erreurs de somme de contrôle empêchent la corruption du flux de données, la conversation TCP ne sera pas très utile jusqu'à ce que le décompresseur génère à nouveau des paquets valides. Comment peut on forcer cela à arriver ?

Le décompresseur génère des paquets invalides parce que son état (le "dernier en-tête de paquet" sauvegardé) est en désaccord avec l'état du compresseur. Un paquet UNCOMPRESSED_TCP va corriger l'état du décompresseur. Donc la récupération d'erreur va contribuer à forcer un paquet non compressé à sortir du compresseur chaque fois que le décompresseur est (ou pourrait être) trompé.

La première idée qui vient est de tirer parti de la liaison de communication bidirectionnelle et de faire que le décompresseur envoie quelque chose au compresseur pour lui demander un paquet non compressé. Cela est clairement indésirable car cela contraint la topologie plus que le minimum suggéré à la Section 2 et exige qu'une grande quantité de protocole soit ajoutée aussi bien au décompresseur qu'au compresseur. En réfléchissant un peu, on se convaincra que cette alternative est non seulement indésirable, mais qu'en plus, elle ne marche pas : les paquets compressés sont petits et il est probable qu'une ligne sera si complètement oblitérée que le décompresseur n'obtiendra rien du tout. Donc, les paquets sont reconstruits incorrectement (à cause du paquet compressé manquant) mais seuls les points d'extrémité TCP savent que les paquets sont incorrects, le décompresseur ne le sait pas.

Mais les points d'extrémité TCP connaissent l'erreur et TCP est un protocole fiable conçu pour fonctionner sur des supports non fiables. Cela signifie que les points d'extrémité doivent finalement prendre des mesures de récupération d'erreur et il y a là un déclencheur évident pour que le compresseur resynchronise le décompresseur : l'envoi de paquets non compressés chaque fois que TCP fait de la récupération d'erreur.

Mais comment le compresseur reconnaît il la récupération d'erreur TCP ? Considérons le schéma de transfert de données TCP de la Figure 6. Le décompresseur dans l'embarras est dans la moitié transmission (transfert de données) de la conversation TCP. Le TCP receveur élimine les paquets plutôt que d'en accuser réception à cause des erreurs de somme de contrôle) le TCP envoyeur arrive finalement en fin de temporisation et retransmet un paquet, et le compresseur du chemin de transmission trouve que la différence entre le numéro de séquence dans le paquet retransmis et le numéro de séquence dans le dernier paquet vu est soit négatif (si il y a plusieurs paquets en transit) soit zéro (un paquet en transit). Le premier cas est détecté dans l'étape de compression qui calcule les différences de numéro de séquence. Le second cas est détecté dans l'étape qui vérifie les cas particuliers de codage mais a besoin d'une vérification supplémentaire : il est très courant

qu'une conversation interactive envoie un paquet d'accusé de réception sans données suivi par un paquet de données. Le paquet d'accusé de réception et le paquet de données vont avoir le même numéro de séquence bien que le paquet de données ne soit pas une retransmission. Pour prévenir l'envoi de paquets non compressés non nécessaires, la longueur du paquet précédent devrait être vérifiée et, si il contient des données, un changement de numéro de séquence de zéro doit indiquer une retransmission.

Un décompresseur dans l"embarras dans l'autre moitié (accusé de réception) de la conversation est tout aussi facile à détecter (Figure 7) : le TCP envoyeur élimine les accusés de réception (parce qu'ils contiennent des erreurs de somme de contrôle) arrive finalement en fin de temporisation, puis retransmet un paquet. Le TCP receveur obtient donc un paquet dupliqué et doit générer un accusé de réception pour le prochain numéro de séquence attendu [11, p. 69]. Cet accusé de réception sera un duplicata du dernier accusé de réception généré par le receveur de sorte que le compresseur du chemin inverse ne va trouver aucun accusé de réception, numéro de séquence, fenêtre ou changement urgent. Si cela arrive pour un paquet qui ne contient pas de données, le compresseur suppose que c'est un duplicata d'accusé de réception envoyé en réponse à une retransmission et envoie un paquet UNCOMPRESSED_TCP /31/.

Note 31 Le paquet pourrait être une sonde de fenêtre zéro plutôt qu'une retransmission d'accusé de réception mais les sondes de fenêtre devraient être peu fréquentes et il n'y a pas de dommage à les envoyer non compressées.

5. Paramètres et réglages configurables

5.1 Configuration de compression

Deux paramètres de configuration sont associés à la compression d'en-tête : si les paquets compressés devraient ou non être envoyés sur une ligne particulière et, si oui, combien d'intervalles de temps (en-têtes de paquet sauvegardés) réserver. Il y a aussi un paramètre de configuration de niveau liaison, la taille de paquet maximale ou la MTU, et un paramètre de configuration d'extrémité frontale, la compression des données, qui interagit avec la compression d'en-tête. La configuration de la compression est discutée dans ce paragraphe. La MTU et la compression des données sont discutées dans les deux paragraphes suivants.

Certains hôtes (par exemple, les PC d'extrémité inférieure) peuvent n'avoir pas assez de ressources de processeur ou de mémoire pour mettre en œuvre cette compression. Il y a aussi de rares caractéristiques de liaison ou d'application qui rendent la compression d'en-tête inutile ou indésirable. Et il y a aussi de nombreuses liaisons SLIP existantes qui n'utilisent pas actuellement ce style de compression d'en-tête. Pour les besoins de l'interopérabilité, les pilotes IP de ligne série qui permettent la compression d'en-tête devraient inclure une sorte de fanion configurable par l'utilisateur pour désactiver la compression (voir l'appendice B.2) /32/.

Note 32 Le protocole PPP décrit dans [9] permet aux points d'extrémité de négocier la compression de sorte qu'il n'y ait pas de problème d'interopérabilité. Cependant, il devrait quand même y avoir une disposition pour que le gestionnaire de système de chaque extrémité contrôle si la compression est négociée ou non. Et évidement, la compression devrait par défaut être désactivée jusqu'à ce que son activation soit négociée.

Si la compression est activée, le compresseur doit être sûr de ne jamais envoyer d'identifiant de connexion (indice d'état) qui sera éliminé par le décompresseur. Par exemple, un trou noir est créé si le décompresseur a seize intervalles et le compresseur en utilise vingt /33/. Aussi, si trop peu d'intervalles sont autorisés au compresseur, l'allocateur LRU va échouer et la plupart des paquets seront envoyés comme UNCOMPRESSED_TCP. Avec trop d'intervalles, la mémoire est gaspillée.

Note 33 Strictement parlant, il n'y a pas de raison pour que l'identifiant de connexion doive être traité comme un indice de tableau. Si l'état du décompresseur a été conservé dans un tableau haché ou autre structure associative, l'identifiant de connexion va être une clé, et non un indice, et les performances avec trop peu d'intervalles au décompresseur vont seulement être énormément dégradées plutôt que de toutes échouer. Cependant, une structure associative est substantiellement plus coûteuse en code et en temps de cpu et, étant donné le faible coût par intervalle (128 octets de mémoire) il semble raisonnable de concevoir des dispositifs d'intervalles au décompresseur et une communication (éventuellement implicite) de la taille du dispositif.

En expérimentant avec différentes tailles pendant l'année dernière, l'auteur est arrivé à la conclusion que huit intervalles vont échouer (c'est-à-dire que la dégradation des performances est perceptible) lorsque de nombreuses fenêtres sur une station à fenêtres multiples sont utilisées simultanément ou si la station de travail est utilisée comme passerelle pour trois machines ou plus. Seize intervalles n'ont jamais été pris en défaut. (Cela peut être simplement parce que une ligne à 9600 bit/s qui s'étale sur plus de 16 voies est déjà tellement surchargée que la dégradation supplémentaire sur le round-robbin des intervalles est négligeable.)

Chaque intervalle doit être assez grand pour contenir une longueur maximum d'en-tête TCP/IP de 128 octets /34/ de sorte que 16 intervalles occupent 2 kB de mémoire. Avec les cartes de RAM à 4 Mbit qu'on a de nos jours, 2 kB semble une si petite mémoire que l'auteur recommande les règles de configuration suivantes :

Note 34 La longueur d'en-tête maximum, fixée par la conception du protocole, est de 64 octets de IP et 64 octets de TCP.

(1) Si le protocole de tramage ne permet pas la négociation, le compresseur et le décompresseur devraient fournir seize intervalles, de zéro à quinze.

(2) Si le protocole de tramage permet la négociation, tout nombre d'intervalles mutuellement acceptable entre 1 et 256 devrait être négociable /35/. Si le nombre d'intervalles n'est pas négocié, ou jusqu'à ce qu'il soit négocié, les deux côtés devraient en supposer seize.

Note 35 Ne permettre qu'un seul intervalle peut rendre le code du compresseur plus complexe. Les mises en œuvre devraient éviter si possible d'offrir un seul intervalle et les mises en œuvre de compresseur peuvent désactiver la compression si seulement un intervalle est négocié.

(3) Si on a un contrôle complet de toutes les machines des deux côtés sur chaque liaison et si aucune d'elles ne sera jamais utilisée pour parler à des machines hors de son contrôle, on est libre de les configurer comme on veut, ignorant ce qui est dit ci-dessus. Cependant, quand votre petit monde fermé sur lequel vous régnez en maître va s'effondrer (comme il semble qu'ils finissent toujours par le faire) sachez bien qu'une grande communauté vocale Internet pas particulièrement tolérante va se faire un plaisir de vous montrer du doigt en disant à quiconque veut bien l'entendre que vous avez mal configuré vos systèmes et que vous n'êtes pas interopérable.

5.2 Choix d'une unité de transmission maximum

De la discussion de la Section 2, il semble désirable de limiter la taille maximum de paquet (la MTU) sur toute ligne où il pourrait y avoir du trafic interactif et plusieurs connexions actives (pour conserver une bonne réponse interactive entre les différentes connexions en compétition pour la ligne). La question évidente est "de combien cela réduit il le débit ?" Cela ne le réduit pas.

La Figure 8 montre comment le débit de données d'usager /36/ s'adapte à la MTU avec (ligne pleine) et sans (ligne en tirets) compression d'en-tête.

Note 36 L'axe vertical est en pourcentage de la vitesse de ligne. Par exemple, '95' signifie que 95 % de la bande passante de la ligne va aux données d'usager ou, en d'autres termes, que l'usager va voir un taux de transfert de données de 9120 bit/s sur une ligne à 9600 bit/s. Quatre octets d'encapsulation de niveau liaison (trameur) en plus de l'en-tête TCP/IP ou compressé ont été inclus lors du calcul du débit relatif. Les temps de paquet de 200 ms ont été calculés en supposant une ligne asynchrone utilisant 10 bits par caractère (8 bits de données, 1 de début, 1 d'arrêt, pas de parité).

Les lignes en pointillé montrent que la MTU correspond à un temps de paquet de 200 ms à 2400, 9600 et 19 200 bit/s. Noter qu'avec la compression d'en-tête même une ligne à 2400 bit/s peut être réactive et donc avoir un débit raisonnable (83%) /37/.

Note 37 Cependant, le MSS TCP de 40 octets requis pour une ligne à 2400 bit/s peut faire subir des contraintes sévères à votre mise en œuvre TCP.

La Figure 9 montre comment l'efficacité de ligne s'adapte à des vitesses de ligne croissantes, en supposant qu'une MTU de 200 ms est toujours choisie /38/.

Note 38 Pour une ligne asynchrone normale, une MTU de 200 ms est simplement 0,02 fois la vitesse de ligne en bits par seconde.

Le point d'inflexion de la courbe de performances est aux alentours de 2400 bit/s. En dessous, l'efficacité est sensible aux petits changements de vitesse (ou de MTU car les deux sont en relation linéaire) et la bonne efficacité se fait aux dépens d'une bonne réactivité. Au-dessus de 2400 bit/s, la courbe est plate et l'efficacité est relativement indépendante de la vitesse ou de la MTU. En d'autres termes, il est possible d'avoir à la fois une bonne réactivité et une forte efficacité de ligne.

À titre d'illustration, noter que pour une ligne à 9600 bit/s avec compression d'en-tête il n'y a pratiquement aucun avantage à augmenter la MTU au delà de 200 octets : si la MTU est augmentée à 576, le délai moyen augmente de 188 % tandis que le débit ne s'améliore que de 3 % (de 96 à 99 %).

5.3 Interaction avec la compression des données

Depuis le début des années 1980, des algorithmes rapides, efficaces de compression de données tels que Lempel-Ziv [7] et des programmes qui les incorporent, tels que le programme de compression fourni par Berkeley Unix, sont devenus largement disponibles. Lorsque on utilise des lignes à bas débit ou à longue portée, il est devenu de pratique courante de compresser les données avant de les envoyer. Pour les connexions à numérotation, cette compression est souvent faite dans les modems, indépendamment des hôtes communicants. Certaines questions intéressantes semblent être :

(1) Étant donné un bon compresseur de données, y a t-il besoin d'un compression d'en-tête ?
(2) La compression d'en-tête interagit-elle avec la compression des données ?
(3) Les données devraient elles être compressées avant ou après la compression d'en-tête ? /39/

Note 39 Les réponses, pour ceux qui souhaitent sauter le reste de ce paragraphe, sont respectivement 'oui', 'non' et 'l'un ou l'autre'.

Pour investiguer sur (1), la compression Lempel-Ziv a été faite sur une trace de 446 paquets TCP/IP pris du côté usager d'une conversation telnet normale. Comme les paquets résultaient d'une frappe, presque tous ne contenaient qu'un octet de données plus 40 octets d'en-tête. C'est-à-dire que l'essai a essentiellement mesuré la compression L-Z des en-têtes TCP/IP. Le taux de compression (le rapport entre les données non compressées et compressées) était de 2,6. En d'autres termes, l'en-tête moyen a été réduit de 40 à 16 octets. Bien que ceci soit une bonne compression, c'est loin des 5 octets d'en-tête nécessaires pour une bonne réponse interactive et loin des 3 octets d'en-tête (un taux de compression de 13,3) que la compression d'en-tête donnait sur la même trace de paquets.

Les seconde et troisième questions sont plus complexes. Pour investiguer sur elles, plusieurs traces de paquets provenant de transferts de fichiers FTP ont été analysées /40/ avec et sans compression d'en-tête et avec et sans compression L-Z.

Note 40 Le volume de données provenant du côté usager d'une connexion telnet est trop petit pour bénéficier de la compression de données et être affecté par les délais qu'ajoutent (nécessairement) la plupart des algorithmes de compression. Les statistiques et le volume du côté ordinateur d'une connexion telnet sont similaires à un FTP (ASCII) de sorte que ces résultats devraient s'appliquer à l'un et l'autre.

La compression L-Z a été essayée à deux endroits du flux de données sortant (Figure 10) :
(1) juste avant que les données aient été traitées par TCP pour encapsulation (simulant la compression faite au niveau 'application') et
(2) après l'encapsulation des données (simulant la compression faite dans le modem). Le Tableau 1 résume les résultats pour un fichier texte ASXII de 78 776 octets (l'entrée manuelle de Unix csh.1) /41/ transféré en utilisant les lignes directrices du paragraphe précédent (256 octets de MTU ou 216 octets de MSS ; un total de 368 paquets). Les taux de compression pour les dix essais suivants sont données (en lisant de gauche à droite et de haut en bas) :

Note 41 Les dix expériences décrites ont été faites chacune sur dix fichiers ASCII (quatre longs messages électroniques, trois fichiers source Unix C et trois entrées manuelles Unix). Les résultats ont été remarquablement similaires pour les différents fichiers et les conclusions générales obtenues ci-dessous s'appliquent à tous les fichiers.

- fichier de données (pas de compression ou d'encapsulation)
- données -> compresseur L--Z
- données -> encapsulation TCP/IP
- données-> L--Z -> TCP/IP
- données -> TCP/IP -> L--Z
- données -> L--Z -> TCP/IP -> L--Z
- données -> TCP/IP -> Compression d'en-tête.
- données -> L--Z -> TCP/IP -> Compression d'en-tête.
- données -> TCP/IP -> Compression d'en-tête. -> L--Z
- données -> L--Z -> TCP/IP -> Compression d'en-tête. -> L--Z

	Pas de compression de données	L--Z sur les données	L--Z sur la ligne	L--Z sur les deux
Données brutes	1,00	2,44	----	----
+ encapsulation TCP	0,83	2,03	1,97	1,58
w/Compression d'en-tête	0,98	2,39	2,26	1,66

Tableau 1 : Taux de compression de fichier texte ASCII

La première colonne du tableau 1 dit que les données subissent une expansion de 19 % ('compressées' de 0,83) lorsque encapsulées dans TCP/IP et de 2 % lorsque encapsulées dans l'en-tête compressé TCP/IP /42/.

Note 42 C'est ce à quoi on peut s'attendre d'après les tailles relatives d'en-tête : 256/216 pour TCP/IP et 219/216 pour la compression d'en-tête.

La première rangée dit que la compression L--Z est assez efficace sur ces données, les réduisant à moins de la moitié de leur taille d'origine. La colonne quatre illustre le fait bien connu que c'est une faute de compresser par L--Z des données déjà compressées. Les informations intéressantes sont dans les lignes deux et trois des colonnes deux et trois. Ces colonnes disent que les avantages de la compression de données dépassent les coûts de l'encapsulation, même pour TCP/IP direct. Elles disent aussi qu'il est légèrement mieux de compresser les données avant de les encapsuler plutôt que de compresser au niveau tramage/modem. Les différences sont cependant petites --- 3 % et 6 %, respectivement, pour les encapsulations de TCP/IP et d'en-tête compressé /43/.

Note 43 Les différences sont dues aux schémas d'octets très différents des datagrammes TCP/IP et de texte ASCII. Tout schéma de compression avec un modèle de source de Markov sous-jacent, tel que Lempel-Ziv, va faire moins bien lorsque des sources radicalement différentes sont entremêlées. Si les proportions relatives des deux sources sont changées, c'est-à-dire, si la MTU est augmentée, la différence de performances entre les deux localisations de compresseur diminuent. Cependant, le taux de diminution est très lent --- augmenter la MTU de 400 % (de 256 à 1024) change seulement la différence entre les choix L--Z de données et de modem de 2,5 % à 1,3 %.

Le Tableau 2 montre la même expérience pour un fichier binaire de 122 880 octets (l'exécutable Sun-3 ps). Bien que les données brutes ne se compressent pas aussi bien, le résultat est qualitativement le même que pour les données ASCII. Le seul changement significatif est dans la ligne deux : il est meilleur d'environ 3 % de compresser les données dans le modem plutôt qu'à la source si on fait l'encapsulation TCP/IP (apparemment, les fichiers binaires Sun et les en-têtes TCP/IP ont des statistiques similaires). Cependant, avec la compression d'en-tête (rangée trois) le résultat était similaire à celui des données ASCII --- c'est plus mauvais d'environ 3 % de compresser au modem plutôt qu'à la source /44/.

Note 44 Il y a d'autres bonnes raisons de compresser à la source : beaucoup moins de paquets doivent être encapsulés et beaucoup moins de caractères doivent être envoyés au modem. L'auteur soupçonne que la solution des 'données compressées dans le modem' devrait être évitée sauf lorsque on est en face d'un système d'exploitation intraitable, propriété d'un fabricant.

	Pas de compression de données	L--Z sur les données	L--Z sur la ligne	L--Z sur les deux
Données brutes	1,00	1,72	----	----
+ encapsulation TCP	0,83	1,43	1,48	1,21
w/Compression d'en-tête	0,98	1,69	1,64	1,28

Tableau 2 : Taux de compression de fichiers binaires

6 Mesure des performances

Un objectif de mise en œuvre de code de compression était d'arriver à quelque chose d'assez simple pour fonctionner aux vitesses du RNIS (64 kbit/s) sur une station de travail normale de 1989. 64 kbit/s est un octet toutes les 122 µs de sorte que 120 µs a été choisi (arbitrairement) comme temps cible de compression/décompression /45/.

Note 45 Le choix du temps n'a pas été complètement arbitraire : la décompression est souvent faite durant le temps de caractère 'fanion' inter-trame de sorte que, sur les systèmes où la décompression est faite au même niveau de priorité que l'interruption d'entrée de la ligne de série, des temps beaucoup plus longs qu'un temps de caractère résulteraient en dépassements de temps chez le receveur. Et avec la moyenne actuelle de trames de cinq octets (sur le réseau, y compris l'en-tête compressé et le tramage) une compression/décompression qui prend le temps d'un octet peut utiliser au plus 20 % du temps disponible. Cela semble un budget confortable.

Machine	**Temps moyen de traitement par paquet (µs)**	
	Compresseur	**Décompresseur**
Sparcstation-1	24	18
Sun 4/260	46	20
Sun 3/60	90	90
Sun 3/50	130	150
HP9000/370	42	33
HP9000/360	68	70
DEC 3100	27	25
Vax 780	430	300
Vax 750	800	500
CCI Tahoe	110	140

Tableau 3 : Temporisations de code de compression

Au titre du développement du code de compression, un exécuteur piloté par une trace a été développé. Il était initialement utilisé pour comparer différents choix de protocoles de compression puis ensuite pour essayer le code sur différentes architectures d'ordinateurs et faire des essais de régression après des "améliorations" de performances. Une petite modification de ce programme d'essai a donné un outil de mesure utile /46/.

Note 46 Le programme d'essai et le programme de temporisateur sont tous deux inclus dans le paquetage à capacité ftp décrit dans l'Appendice A comme tester.c et timer.c de fichiers.

Le Tableau 3 montre le résultat de l'étalonnage du code de compression sur toutes les machines que l'auteur a eu à sa

disposition (les temps ont été mesurés en utilisant des traces de trafic mixtes telnet/ftp). À l'exception des architectures de Vax, qui souffrent (a) d'avoir les octets dans le mauvais ordre et (b) d'un compilateur défectueux (Unix pcc), toutes les machines ont essentiellement satisfait à l'objectif de 120 µs.

7 Remerciements

L'auteur exprime sa reconnaissance aux membres de l'équipe d'ingénierie de l'Internet (IETF), présidée par Phill Gross, qui a prodigué ses encouragements et a relu avec soin ce travail. Plusieurs patients beta-testeurs, en particulier Sam Leffler et Craig Leres, ont traqué et corrigé les problèmes de la mise en œuvre initiale. Cynthia Livingston et Craig Partridge ont relu avec soin et largement amélioré une séquence interminable de projets partiels du présent document. Et enfin, et pas le moindre, la "Telebit modem corporation", en particulier Mike Ballard, a encouragé ce travail depuis sa conception et a été le champion incontesté de la ligne de série et de l'accès téléphonique IP.

8 Références

[1] Bingham, J. A. C. "Theory and Practice of Modem Design". John Wiley & Sons, 1988.

[2] Carey, M. B., Chan, H.-T., Descloux, A., Ingle, J. F., et Park, K. I. "1982/83 end office connection study: Analog voice and voiceband data transmission performance characterization of the public switched network". Bell System Technical Journal 63, 9, novembre 1984.

[3] Chiappa, N., 1988. "communication privée".

[4] Clark, D. D. "The design philosophy of the DARPA Internet protocols". Dans Proceedings of SIGCOMM '88 (Stanford, CA, août 1988), ACM.

[5] D. Farber, G. Delp et T. Conte, "Protocole pour la connexion d'ordinateurs individuels à l'Internet par paire fine", RFC0914, septembre 1984. *(Historique)*

[6] Kent, C. A., et Mogul, J. "Fragmentation considered harmful". Dans Proceedings of SIGCOMM '87 (août 1987), ACM.

[7] Lempel, A., et Ziv, J. "Compression of individual sequences via variable-rate encoding". IEEE Transactions on Information Theory IT-24, 5 (juin 1978).

[8] J. Nagle, "Contrôle de l'encombrement dans l'inter-réseau IP/TCP", RFC0896, janvier 1984.

[9] D. Perkins, "Protocole point à point : proposition d'une transmission multi protocole de datagrammes sur liaisons en point à point", RFC1134, novembre 1989. *(Obsolète, voir* RFC1661)

[10] J. Postel, éd., "Protocole Internet - Spécification du protocole du programme Internet", RFC0791, STD 5, septembre 1981.

[11] J. Postel (éd.), "Protocole de commande de transmission – Spécification du protocole du programme Internet DARPA", RFC0793, STD 7, septembre 1981.

[12] J. Romkey, "Non norme pour la transmission des datagrammes IP sur des lignes en série : SLIP", RFC1055, STD 47, juin 1988.

[13] Salthouse, T. A. "The skill of typing". Scientific American 250, 2 (février 1984), pages 128--135.

[14] Saltzer, J. H., Reed, D. P., et Clark, D. D. "End-to-end arguments in system design". ACM Transactions on Computer Systems 2, 4 (novembre 1984).

[15] Shneiderman, B. "Designing the User Interface". Addison-Wesley, 1987.

Annexe A Exemple de mise en œuvre

Ce qui suit est un échantillon de mise en œuvre du protocole décrit dans le présent document.

Comme de nombreuses personnes qui pourraient avoir besoin de traiter ce code sont familiarisées avec le noyau Berkeley Unix et son style de codage (connu affectueusement sous le nom de forme normale de noyau) ce code a été fait dans ce style. Il utilise les 'sous-programmes' Berkeley (en fait, les macros et/ou expansions d'assembleur en ligne) pour convertir de/vers l'ordre des octets du réseau et copier/comparer les chaînes d'octets. Ces sous-programmes sont brièvement décrits au paragraphe A.5 pour ceux à qui ils ne sont pas familiers.

Ce code a fonctionné sur toutes les machines énumérées dans le tableau de la Section 6. Donc, l'auteur espère qu'il n'y a pas de problème d'ordre des octets ou d'alignement (bien qu'il y ait des hypothèses a priori sur l'alignement qui soient valides pour Berkeley Unix mais pourraient n'être pas vraies pour d'autres mises en œuvre IP --- voir les commentaires qui mentionnent l'alignement dans sl_compress_tcp et sl_decompress_tcp).

Il y a eu des tentatives pour rendre ce code efficace. Malheureusement, cela peut en avoir rendu certaines portions incompréhensibles. L'auteur présente ses excuses pour la frustration que cela peut engendrer. (En toute honnêteté mon style de code C est connu pour être obscur et les revendications 'd'efficacité' sont simplement une excuse commode).

Cet échantillon de code et une mise en œuvre complète Berkeley Unix sont disponibles en langage machine via ftp anonyme sur l'hôte Internet < ftp.ee.lbl.gov > (128.3.254.68) fichier cslip.tar.Z. C'est un fichier Unix tar compressé. Il doit être transféré par ftp en mode binaire.

Tout le code du présent appendice est couvert par le droit de reproduction suivant :

"Copyright (c) 1989 Regents of the University of California. Tous droits réservés.

La redistribution et l'utilisation en formes source et binaire sont permises pourvu que la notice de droits de reproduction ci-dessus et le présent paragraphe soient dupliqués dans les mêmes formes et que tout document, matériel publicitaire, et autres matériels se rapportant à une telle distribution et usage reconnaisse que le logiciel a été développé par l'Université de Californie à Berkeley. Le nom de l'Université ne peut pas être utilisé pour soutenir ou promouvoir des produits dérivés de ce logiciel sans permission écrite spécifique préalable .
Le présent logiciel est fourni "EN L'ÉTAT" et sans aucune garantie expresse ou implicite, y compris sans limitation, que les garanties impliquées de possibilité de commercialisation ou d'adéquation à un objet particulier."

A.1 Définitions et données d'état

```
#define MAX_STATES 16                   /* doit être > 2 et < 255 */
#define MAX_HDR 128                     /* longueur maximum d'en-tête TCP+IP (par définition du protocole) */

        /* type de paquets */
#define TYPE_IP 0x40
#define TYPE_UNCOMPRESSED_TCP 0x70
#define TYPE_COMPRESSED_TCP 0x80
#define TYPE_ERROR 0x00                 /* ce n'est pas un type qui apparaît toujours sur la ligne. Le trameur récepteur
                                        /* l'utilise pour dire au décompresseur qu'il y a une erreur de transmission du
                                        /* paquet. */

        /* Bits dans le premier octet du paquet compressé */
        /* Bits fanions pour ce qui a changé dans un paquet */
#define NEW_C  0x40
#define NEW_I  0x20
#define TCP_PUSH_BIT 0x10
#define NEW_S  0x08
#define NEW_A  0x04
#define NEW_W  0x02
#define NEW_U  0x01

        /* valeurs réservées, cas particuliers des précédentes */
#define SPECIAL_I (NEW_S|NEW_W|NEW_U)         /* trafic interactif en écho */
#define SPECIAL_D (NEW_S|NEW_A|NEW_W|NEW_U)   /* données unidirectionnelles */
```

```
#define SPECIALS_MASK (NEW_S|NEW_A|NEW_W|NEW_U)

/* Données "d'état" pour chaque conversation tcp active sur la ligne. C'est basiquement une copie de l'en-tête IP/TCP
    entier du dernier paquet avec un petit identifiant qu'utilisent les extrémité d'émission et de réception de la ligne pour
    localiser l'en-tête sauvegardé. */
struct cstate {
    struct cstate *cs_next;             /* prochain cstate le plus récemment utilisé (seulement émission) */
    u_short cs_hlen;                    /* taille de l'en-tête (seulement réception) */
    u_char cs_id;                       /* numéro de connexion associé à cet état */
    u_char cs_filler;
    union {
        char hdr[MAX_HDR];
        struct ip csu_ip;               /* en-tête ip/tcp provenant du paquet le plus récent */
    } slcs_u;
};
#define cs_ip slcs_u.csu_ip
#define cs_hdr slcs_u.csu_hdr

/* toutes les données d'état pour une ligne série (on a besoin d'elles pour chaque ligne). */
struct slcompress {
    struct cstate *last_cs;             /* tstate le plus récemment utilisé */
    u_char last_recv;                   /* dernier identifiant de connexion reçu */
    u_char last_xmit;                   /* dernier identifiant de connexion envoyé */
    u_short flags;
    struct cstate tstate[MAX_STATES];   /* états de connexion en émission */
    struct cstate rstate[MAX_STATES];   /* états de connexion en réception */
};

        /* valeurs de fanion */
#define SLF_TOSS 1                      /* élimination des trames reçues à cause d'erreurs d'entrée */

/* Les macros suivantes sont utilisées pour coder et décoder des nombres. Elles supposent toutes que 'cp' pointe sur une
    mémoire tampon où le prochain octet codé (décodé) est à mémoriser (restituer). Comme les sous-programmes de
    décodage font de l'arithmétique, ils doivent convertir de/vers l'ordre des octets du réseau. */

/* ENCODE code un nombre qui est connu comme non zéro. ENCODEZ vérifie les zéros (zéro doit être codé en forme
    longue à 3 octets). */
#define ENCODE(n) { \
    if ((u_short)(n) >= 256) { \
        *cp++ = 0; \
        cp[1] = (n); \
        cp[0] = (n) >> 8; \
        cp += 2; \
    } else { \
        *cp++ = (n); \
    } \
}
#define ENCODEZ(n) { \
    if ((u_short)(n) >= 256 || (u_short)(n) == 0) { \
        *cp++ = 0; \
        cp[1] = (n); \
        cp[0] = (n) >> 8; \
        cp += 2; \
    } else { \
        *cp++ = (n); \
    } \
}

/* DECODEL prend le changement (compressé) à l'octet cp et l'ajoute à la valeur actuelle du champ 'f' du paquet (qui doit
    être un entier (long) de 4 octets dans l'ordre des octets du réseau). DECODES fait la même chose pour un entier (court)
    de 2 octets. DECODEU prend le changement à cp et et le fourre dans le champ f (court). 'cp' est mis à jour pour pointer
    sur le prochain champ d'en-tête compressé. */
#define DECODEL(f) { \
```

```
 if (*cp == 0) {\
     (f) = htonl(ntohl(f) + ((cp[1] << 8) | cp[2])); \
     cp += 3; \
 } else { \
     (f) = htonl(ntohl(f) + (u_long)*cp++); \
 } \
}
#define DECODES(f) { \
    if (*cp == 0) {\
        (f) = htons(ntohs(f) + ((cp[1] << 8) | cp[2])); \
        cp += 3; \
    } else { \
        (f) = htons(ntohs(f) + (u_long)*cp++); \
    } \
}
#define DECODEU(f) { \
    if (*cp == 0) {\
        (f) = htons((cp[1] << 8) | cp[2]); \
        cp += 3; \
    } else { \
        (f) = htons((u_long)*cp++); \
    } \
}
```

A.2 Compression

Ce sous-programe paraît intimidant mais il ne l'est pas vraiment. Le code se partage en quatre sections de taille approximativement égale : le premier quart donne une liste liée en cercle des connexions TCP "actives" les plus récemment utilisées /47/. Le second représente les changements de séquence/accusé de réception/fenêtre/urgent et construit la gros du paquet compressé. Le troisième traite les cas particuliers de codage. Le dernier quart fait le codage d'identifiant de paquet et d'identifiant de connexion et remplace l'en-tête original du paquet par l'en-tête compressé.

Note 47 Les deux opérations les plus courantes sur la liste des connexions sont un 'find' *(trouver)* qui se termine à la première entrée (un nouveau paquet pour la connexion le plus récemment utilisée) et déplace la dernière entrée de la liste en tête de liste (le premier paquet d'une nouvelle connexion). Un liste circulaire traite efficacement ces deux opérations.

L'arguments de ce sous-programme est un pointeur sur un paquet à compresser, un pointeur sur les données d'état de compression pour la ligne série, et un fanion qui active ou désactive la compression d'identifiant de connexion (C bit).

La compression est faite 'en place' de sorte que si un paquet compressé est créé, l'adresse de départ et la longueur du paquet entrant (les champs off et len de m) seront mis à jour pour refléter le reste de l'en-tête original et son remplacement par l'en-tête compressé. Si un paquet compressé ou non compressé est créé, l'état de compression est mis à jour. Ce sous programme retourne le type de paquet pour le trameur d'émission (TYPE_IP, TYPE_UNCOMPRESSED_TCP ou TYPE_COMPRESSED_TCP).

Parce que l'arithmétique de 16 et 32 bits est faite sur divers champs d'en-tête, le paquet IP entrant doit être aligné de façon appropriée (par exemple, sur un SPARC, l'en-tête IP est aligné sur une limite de 32 bits). Des changements substantiels devraient être apportés au code ci-dessous si ce n'était pas vrai (et il serait probablement moins coûteux de copier les octets de l'en-tête entrant sur un support correctement aligné que de faire ces changements).

Noter que le paquet sortant sera aligné de façon arbitraire (par exemple, il pourrait facilement commencer sur une limite d'octet impaire).

```
u_char
sl_compress_tcp(m, comp, compress_cid)
    struct mbuf *m;
    struct slcompress *comp;
    int compress_cid;
{
    register struct cstate *cs = comp->last_cs->cs_next;
    register struct ip *ip = mtod(m, struct ip *);
```

```
    register u_int hlen = ip->ip_hl;
    register struct tcphdr *oth;          /* dernier en-tête TCP */
    register struct tcphdr *th;           /* en-tête TCP en cours */

    register u_int deltaS, deltaA;        /* temporaires d'utilité générale */
    register u_int changes = 0;           /* gabarit de changement */
    u_char new_seq[16];                   /* changements de dernier en actuel */
    register u_char *cp = new_seq;
```

/* Vérifier si c'est un fragment IP ou si le paquet TCP n'est pas 'compressible' (c'est-à-dire, si ACK n'est pas établi ou si quelque autre bit de contrôle n'est pas établi). (On suppose que l'appelant s'est déjà assuré que le paquet est IP proto TCP). */

```
if ((ip->ip_off & htons(0x3fff)) || m->m_len < 40)
      return (TYPE_IP);

th = (struct tcphdr *) & ((int *) ip)[hlen];
if ((th->th_flags & (TH_SYN | TH_FIN | TH_RST | TH_ACK)) != TH_ACK)
      return (TYPE_IP);
```

/* Le paquet est compressible – on va envoyer un paquet COMPRESSED_TCP, ou paquet UNCOMPRESSED_TCP. Dans les deux cas on doit localiser (ou créer) l'état de connexion. Cas particulier de la connexion utilisée le plus récemment car il est très probable de l'utiliser à nouveau et on n'a pas à faire de réarrangement si elle est utilisée. */

```
if (ip->ip_src.s_addr != cs->cs_ip.ip_src.s_addr ||
   ip->ip_dst.s_addr != cs->cs_ip.ip_dst.s_addr ||
   *(int *) th != ((int *) &cs->cs_ip)[cs->cs_ip.ip_hl]) {
```

/* Elle n'était pas la première – la chercher. Les états sont conservés dans une liste liée en cercle avec last_cs pointant sur la fin de la liste. La liste est conservée en ordre lru en passant un état en tête de liste chaque fois qu'il est référencé. Comme la liste est courte et, empiriquement, que la connexion qu'on veut est presque toujours près de la tête, on localise les états via une recherche linéaire. Si on ne trouve pas un état pour le datagramme, le plus vieil état est (ré)utilisé. */

```
      register struct cstate *lcs;
      register struct cstate *lastcs = comp->last_cs;

      do {
          lcs = cs;
          cs = cs->cs_next;
          if (ip->ip_src.s_addr == cs->cs_ip.ip_src.s_addr
              && ip->ip_dst.s_addr == cs->cs_ip.ip_dst.s_addr
              && *(int *) th == ((int *) &cs->cs_ip)[cs->cs_ip.ip_hl])
               goto found;

      } while (cs != lastcs);
```

/* On ne l'a pas trouvé -- réutiliser le plus ancien cstate. Envoyer un paquet non compressé qui dit à l'autre côté quel numéro de connexion on utilise pour cette conversation. Noter que comme la liste d'états est circulaire, le plus ancien état pointe sur le plus récent et on a seulement besoin de régler last_cs à mettre à jour la liaison lru. */

```
      comp->last_cs = lcs;
      hlen += th->th_off;
      hlen <<= 2;
      goto uncompressed;

found:                                /* On l'a trouvé – le passer en tête de la liste des connexions. */
      if (lastcs == cs)
          comp->last_cs = lcs;
      else {
          lcs->cs_next = cs->cs_next;
          cs->cs_next = lastcs->cs_next;
          lastcs->cs_next = cs;
      }
   }
```

/* S'assurer que seul ce qu'on s'attend à changer a changé. La première ligne du 'if' vérifie la version de protocole IP, la longueur d'en-tête et le type de service. La seconde ligne vérifie le bit "Ne pas fragmenter". La troisième ligne vérifie la

durée de vie et le protocole (la vérification du protocole est inutile mais ne coûte rien). La quatrième ligne vérifie la longueur d'en-tête TCP. La cinquième ligne vérifie les options IP, si il en est. La sixième ligne vérifie les options TCP, s'il en est. Si une de ces choses diffère entre le datagramme précédent et celui en cours, on envoie le datagramme en cours 'non compressé'. */

```
    oth = (struct tcphdr *) & ((int *) &cs->cs_ip)[hlen];
    deltaS = hlen;
    hlen += th->th_off;
    hlen <<= 2;

    if (((u_short *) ip)[0] != ((u_short *) &cs->cs_ip)[0] ||
        ((u_short *) ip)[3] != ((u_short *) &cs->cs_ip)[3] ||
        ((u_short *) ip)[4] != ((u_short *) &cs->cs_ip)[4] ||
        th->th_off != oth->th_off ||
        (deltaS > 5 && BCMP(ip + 1, &cs->cs_ip + 1, (deltaS - 5) << 2)) ||
        (th->th_off > 5 && BCMP(th + 1, oth + 1, (th->th_off - 5) << 2)))
        goto uncompressed;
```

/* Représente quels champs changeants ont changé. Le receveur attend les changements dans l'ordre : urgent, fenêtre, accusé de réception, séquence. */

```
    if (th->th_flags & TH_URG) {
        deltaS = ntohs(th->th_urp);
        ENCODEZ(deltaS);
        changes |= NEW_U;
    } else if (th->th_urp != oth->th_urp)
```

/* Zut ! URG n'est pas établi mais urp a changé – une mise en œuvre intelligente ne devrait jamais faire cela mais la RFC793 n'interdit pas ce changement, de sorte qu'on doit faire avec. */

```
        goto uncompressed;

    if (deltaS = (u_short) (ntohs(th->th_win) - ntohs(oth->th_win))) {
        ENCODE(deltaS);
        changes |= NEW_W;
    }
    if (deltaA = ntohl(th->th_ack) - ntohl(oth->th_ack)) {
        if (deltaA > 0xffff)
            goto uncompressed;
        ENCODE(deltaA);
        changes |= NEW_A;
    }
    if (deltaS = ntohl(th->th_seq) - ntohl(oth->th_seq)) {
        if (deltaS > 0xffff)
            goto uncompressed;
        ENCODE(deltaS);
        changes |= NEW_S;
    }
```

/* Cherche les cas particuliers de codage. */

```
    switch (changes) {

cas 0 :
```

/* Rien n'a changé. Si ce paquet contient des données et si le dernier n'en contenait pas, c'est probablement un paquet de données suivant un accusé de réception (normal sur une connexion interactive) et on l'envoie compressé. Autrement, c'est probablement une retransmission, un accusé de réception retransmis ou une sonde de fenêtre. L'envoyer non compressé au cas où l'autre côté aurait manqué la version compressée. */

```
        if (ip->ip_len != cs->cs_ip.ip_len &&
            ntohs(cs->cs_ip.ip_len) == hlen)
            break;

/* (passé au travers) */

cas SPECIAL_I :
cas SPECIAL_D :
```

/* Les changements réels correspondent à un de nos cas particuliers de codage – envoyer un paquet non compressé. */

```
        goto uncompressed;
```

```
cas NEW_S | NEW_A :
        if (deltaS == deltaA &&
           deltaS == ntohs(cs->cs_ip.ip_len) - hlen) {       /* cas particulier de l'écho de trafic terminal */
            changes = SPECIAL_I;
            cp = new_seq;
        }
        break;

cas NEW_S :
        if (deltaS == ntohs(cs->cs_ip.ip_len) - hlen) {       /* cas particulier de transfert de données */
            changes = SPECIAL_D;
            cp = new_seq;
        }
        break;
    }
    deltaS = ntohs(ip->ip_id) - ntohs(cs->cs_ip.ip_id);
    if (deltaS != 1) {
        ENCODEZ(deltaS);
        changes |= NEW_I;
    }
    if (th->th_flags & TH_PUSH)
        changes |= TCP_PUSH_BIT;
/* Saisir la somme de contrôle avant qu'elle soit réécrite ci-dessous. Puis mettre à jour l'état avec cet en-tête de paquet. */
    deltaA = ntohs(th->th_sum);
    BCOPY(ip, &cs->cs_ip, hlen);

/* On veut utiliser le paquet original comme paquet compressé. (cp - new_seq) est le nombre d'octets nécessaire pour les
    numéros de séquence compressés. On a besoin en plus d'un octet pour le gabarit de changement, d'un autre pour
    l'identifiant de connexion et de deux pour la somme de contrôle tcp. Ainsi, (cp - new_seq) + 4 octets d'en-tête sont
    nécessaires. hlen est le nombre d'octets du paquet original à lancer, aussi soustraire les deux pour obtenir la nouvelle
    taille de paquet. */
    deltaS = cp - new_seq;
    cp = (u_char *) ip;
    if (compress_cid == 0 || comp->last_xmit != cs->cs_id) {
        comp->last_xmit = cs->cs_id;
        hlen -= deltaS + 4;
        cp += hlen;
        *cp++ = changes | NEW_C;
        *cp++ = cs->cs_id;
    } else {
        hlen -= deltaS + 3;
        cp += hlen;
        *cp++ = changes;
    }
    m->m_len -= hlen;
    m->m_off += hlen;
    *cp++ = deltaA >> 8;
    *cp++ = deltaA;
    BCOPY(new_seq, cp, deltaS);
    return (TYPE_COMPRESSED_TCP);

non compressé :
/* Mettre à jour l'état de connexion cs et envoyer le paquet non compressé ('non compressé' signifie un paquet ip/tcp
    normale mais avec 'l'identifiant de conversation' qu'on espère utiliser sur les futurs paquets compressés dans le champ
    Protocole). */
    BCOPY(ip, &cs->cs_ip, hlen);
    ip->ip_p = cs->cs_id;
    comp->last_xmit = cs->cs_id;
    return (TYPE_UNCOMPRESSED_TCP);
}
```

A.3 Décompression

Ce sous-programme décompresse un paquet reçu. Il est invoqué avec un pointeur sur le paquet, la longueur et le type du paquet, et un pointeur sur la structure d'état de compression pour la ligne série entrante. Il retourne un pointeur sur le paquet résultant ou zéro si il y avait des erreurs dans le paquet entrant. Si le paquet est COMPRESSED_TCP ou UNCOMPRESSED_TCP, l'état de compression sera mis à jour.

Le nouveau paquet sera construit en place. Cela signifie qu'il doit y avoir 128 octets d'espace libre devant bufp pour laisser de la place pour les en-têtes IP et TCP reconstruits. Le paquet reconstruit sera verrouillé sur une limite de 32 bits.

```
u_char *
sl_uncompress_tcp(bufp, len, type, comp)
    u_char *bufp;
    int len;
    u_int type;
    struct slcompress *comp;
{
    register u_char *cp;
    register u_int hlen, changes;
    register struct tcphdr *th;
    register struct cstate *cs;
    register struct ip *ip;
    switch (type) {

cas TYPE_ERROR :
    default:
        goto bad;

cas TYPE_IP :
        return (bufp);

cas TYPE_UNCOMPRESSED_TCP :
/* Localiser l'état sauvegardé pour cette connexion. Si l'indice d'état est légal, mettre le fanion 'discard' à zéro. */
        ip = (struct ip *) bufp;
        if (ip->ip_p >= MAX_STATES)
            goto bad;

        cs = &comp->rstate[comp->last_recv = ip->ip_p];
        comp->flags &= ~SLF_TOSS;
/* Restaurer le champ Protocole IP puis sauvegarder une copie de cet en-tête de paquet. (La somme de contrôle est mise à
    zéro dans la copie de sorte qu'on n'a pas à le faire à chaque fois qu'on traite un paquet compressé. */
        ip->ip_p = IPPROTO_TCP;
        hlen = ip->ip_hl;
        hlen += ((struct tcphdr *) & ((int *) ip)[hlen])->th_off;
        hlen <<= 2;
        BCOPY(ip, &cs->cs_ip, hlen);
        cs->cs_ip.ip_sum = 0;
        cs->cs_hlen = hlen;
        return (bufp);

cas TYPE_COMPRESSED_TCP :
        break;
    }                                       /* On a un paquet compressé. */
    cp = bufp;
    changes = *cp++;
    if (changes & NEW_C) {
/* S'assurer que l'indice d'état est dans la gamme, puis attraper l'état. Si on a un bon indice d'état, mettre le fanion 'discard'
    à zéro. */
        if (*cp >= MAX_STATES)
            goto bad;

        comp->flags &= ~SLF_TOSS;
```

```
        comp->last_recv = *cp++;
    } else {
/* Ce paquet a un indice d'état implicite. Si on a eu une erreur de ligne depuis la dernière fois qu'on a eu un indice d'état
     explicite, on doit rejeter le paquet. */
        if (comp->flags & SLF_TOSS)
            return ((u_char *) 0);
    }
/* Trouver l'état, puis remplir la somme de contrôle TC et le bit PUSH. */
    cs = &comp->rstate[comp->last_recv];
    hlen = cs->cs_ip.ip_hl << 2;
    th = (struct tcphdr *) & ((u_char *) &cs->cs_ip)[hlen];
    th->th_sum = htons((*cp << 8) | cp[1]);
    cp += 2;
    if (changes & TCP_PUSH_BIT)
        th->th_flags |= TH_PUSH;
    else
        th->th_flags &= ~TH_PUSH;

/* Réparer les champs ack, seq, urg et win de l'état sur la base du gabarit de changement. */
    switch (changes & SPECIALS_MASK) {
    cas SPECIAL_I :
        {
        register u_int i = ntohs(cs->cs_ip.ip_len) - cs->cs_hlen;
        th->th_ack = htonl(ntohl(th->th_ack) + i);
        th->th_seq = htonl(ntohl(th->th_seq) + i);
        }
        break;

cas SPECIAL_D :
        th->th_seq = htonl(ntohl(th->th_seq) + ntohs(cs->cs_ip.ip_len) - cs->cs_hlen);
          break;

par défaut :
        if (changes & NEW_U) {
            th->th_flags |= TH_URG;
            DECODEU(th->th_urp)
        } else
            th->th_flags &= ~TH_URG;
        if (changes & NEW_W)
            DECODES(th->th_win)
        if (changes & NEW_A)
            DECODEL(th->th_ack)
        if (changes & NEW_S)
            DECODEL(th->th_seq)
        break;
    }                                       /* Mettre à jour l'identifiant IP */
    if (changes & NEW_I)
        DECODES(cs->cs_ip.ip_id)
    else
        cs->cs_ip.ip_id = htons(ntohs(cs->cs_ip.ip_id) + 1);

/* À ce moment, cp pointe sur le premier octet de données dans le paquet. Si on n'est pas verrouillé sur une limite de 4
     octets, copier les données de telle sorte que les en-têtes IP et TCP soient alignés. Puis sauvegarder cp de la longueur de
     l'en-tête TCP/IP pour faire de la place pour l'en-tête reconstruit (on suppose que le paquet qu'on a en main a assez
     d'espace pour ajouter 128 octets d'en-tête). Ajuster la longueur pour tenir compte du nouvel en-tête et remplir le champ
     Longueur totale IP. */
    len -= (cp - bufp);
    if (len < 0)
/* on doit avoir abandonné certains caractères (le crc devrait le détecter mais pas le vieux tramage slip) */
        goto bad;

    if ((int) cp & 3) {
        if (len > 0)
```

```
            OVBCOPY(cp, (int) cp & ~3, len);
        cp = (u_char *) ((int) cp & ~3);
    }
    cp -= cs->cs_hlen;
    len += cs->cs_hlen;
    cs->cs_ip.ip_len = htons(len);
    BCOPY(&cs->cs_ip, cp, cs->cs_hlen);

                                        * recalcule la somme de contrôle d'en-tête IP */
    {
        register u_short *bp = (u_short *) cp;
        for (changes = 0; hlen > 0; hlen -= 2)
            changes += *bp++;
        changes = (changes & 0xffff) + (changes >> 16);
        changes = (changes & 0xffff) + (changes >> 16);
        ((struct ip *) cp)->ip_sum = ~changes;
    }
    return (cp);

bad:
    comp->flags |= SLF_TOSS;
    return ((u_char *) 0);
}
```

A.4 Initialisation

Ce sous-programme initialise la structure d'état pour les deux moitiés émission et réception de certaines lignes série. Il doit être invoqué chaque fois que la ligne est activée.

```
void
sl_compress_init(comp)
    struct slcompress *comp;
{
    register u_int i;
    register struct cstate *tstate = comp->tstate;

/* Nettoyer tout ce qui reste de la dernière utilisation de la ligne. */
    bzero((char *) comp, sizeof(*comp));
/* Relier les état d'émission en une liste circulaire. */
    for (i = MAX_STATES - 1; i > 0; --i) {
        tstate[i].cs_id = i;
        tstate[i].cs_next = &tstate[i - 1];
    }
    tstate[0].cs_next = &tstate[MAX_STATES - 1];
    tstate[0].cs_id = 0;
    comp->last_cs = &tstate[0];
/* S'assurer qu'on ne fait pas accidentellement de compression de CID (on suppose MAX_STATES < 255). */
    comp->last_recv = 255;
    comp->last_xmit = 255;
}
```

A.5 Dépendance à Unix Berkeley

Note : Ce qui suit n'intéresse que ceux qui essayent de monter l'échantillon de code sur un système qui n'est pas dérivé de 4BSD (Berkeley Unix).

Le code utilise les fichiers normaux d'en-tête Berkeley Unix (de /usr/include/netinet) pour les définitions de la structure des en-têtes IP et TCP. Les étiquettes de structure tendent à suivre étroitement les RFC de protocole et devraient être évidentes même si on n'a pas accès à un système 4BSD /48/.

Note 48 Au cas où ils ne seraient pas évidents, les fichiers d'en-tête (et tout le code de réseautage Berkeley) peut être

collecté par ftp anonyme à l'hôte < ucbarpa.berkeley.edu>, fichiers pub/4.3/tcp.tar et pub/4.3/inet.tar.

La macro BCOPY(src, dst, amt) est invoquée pour copier les octets amt de source à destination. Dans BSD, elle se traduit en une invocation de bcopy. Si on a le malheur de fonctionner avec un System-V Unix, elle peut être traduite en une invocation de memcpy. La macro OVBCOPY(src, dst, amt) est utilisée pour copier lorsque source et destination se chevauchent (c'est-à-dire, lorsque on fait la copie d'alignement sur une limite de quatre octets). Dans le noyau BSD, elle traduit en une invocation de ovbcopy. Comme AT&T a saboté la définition de memcpy, cela devrait probablement se traduire en une copie en boucle dans le System-V.

La macro BCMP(src, dst, amt) est invoquée pour comparer l'égalité des octets amt de source et de destination. Dans BSD, elle se traduit en une invocation de bcmp. En System-V, elle peut être traduite en une invocation de memcmp ou on peut écrire un sous-programme pour faire la comparaison. Le sous programme devrait retourner zéro si tous les octets de source et destination sont égaux et non zéro par ailleurs.

Le sous-programme ntohl(dat) convertit les données longues (4 octets) de l'ordre des octets du réseau en ordre des octets de l'hôte. Sur un cpu raisonnable, ce peut être la macro no-op : #define ntohl(dat) (dat)

Sur un Vax ou IBM PC (ou tout ce qui a l'ordre des octets d'Intel) on devra définir une macro ou un sous programme pour réarranger les octets.

Le sous-programme ntohs(dat) est comme ntohl mais convertit les courts (2 octets) au lieu des longs. Les sous-programmes htonl(dat) et htons(dat) font la transformation inverse (de l'ordre des octets de l'hôte à celui du réseau) pour les longs et les courts.

Une structure mbuf est utilisée dans l'invocation de sl_compress_tcp parce que ce sous-programme a besoin de modifier l'adresse de départ et la longueur si le paquet entrant est compressé. En BSD, un mbuf est la structure de gestion de mémoire tampon du noyau. Si on a d'autres systèmes, la définition suivante devrait être suffisante :

```
struct mbuf {
        u_char  *m_off;                 /* pointeur sur le début des données */
        int     m_len;                  /* longueur des données */
};

   #define mtod(m, t) ((t)(m->m_off))
```

Annexe B Compatibilité avec les erreurs du passé

Lorsque elle est combinée avec le protocole moderne de ligne série PPP [9], l'utilisation de la compression d'en-tête est automatique et invisible à l'usager. Malheureusement, de nombreux sites ont des usagers existants du SLIP décrit dans [12] qui ne permettent pas que différents types de protocoles distinguent les paquets à en-tête compressé des paquets IP ou les numéros de version ou un échange d'option qui pourrait être utilisé pour négocier automatiquement la compression d'en-tête.

L'auteur a utilisé les trucs suivants pour permettre que les en-têtes compressés SLIP interopèrent avec les serveurs et clients existants. Noter que ce sont des trucs pour la compatibilité avec les erreurs du passé et qu'ils pourraient choquer toutes les personnes bien pendantes. Ils ne sont proposés que pour apaiser la douleur d'avoir à faire fonctionner SLIP pendant que les usagers attendent patiemment que leurs fabricants sortent PPP.

B.1 Vivre sans octet de "type" de tramage

Les nombres bizarres de type de paquet du paragraphe A.1 ont été choisis pour permettre qu'un `type de paquet' soit envoyé sur des lignes où il est indésirable ou impossible d'ajouter un octet de type explicite. Noter que le premier octet d'un paquet IP contient toujours '4' (la version du protocole IP) dans les quatre bits de poids fort. Et que le bit de poids fort du premier octet de l'en-tête compressé est ignoré. En utilisant le type de paquets du paragraphe A.1, le type peut être codé dans les bits de poids fort du paquet sortant en utilisant le code

```
p->dat[0] |= sl_compress_tcp(p, comp);
```

et décodé du côté receveur par

```
si (p->dat[0] & 0x80)
```

```
        type = TYPE_COMPRESSED_TCP;
    autrement si (p->dat[0] ≥ 0x70) {
        type = TYPE_UNCOMPRESSED_TCP;
        p->dat[0] &=~ 0x30;
    } autrement
        type = TYPE_IP;
    status = sl_uncompress_tcp(p, type, comp);
```

B.2 Serveurs SLIP rétro compatibles

Le SLIP décrit dans [12] ne comporte aucun mécanisme qui puisse être utilisé pour négocier automatiquement la compression d'en-tête. Il serait bien de permettre aux usagers de SLIP d'utiliser la compression d'en-tête mais, lorsque des usagers des deux variantes de SLIP partagent un serveur commun, il serait ennuyeux et difficile de configurer manuellement les deux extrémités de chaque connexion pour permettre la compression. La procédure suivante peut être utilisée pour éviter la configuration manuelle.

Comme il y a deux types de clients à numérotage (ceux qui mettent en œuvre la compression et ceux qui ne le font pas) mais un seul serveur pour les deux types, il est clair que le serveur va reconfigurer pour chaque nouvelle session client mais que les clients changent rarement, sinon jamais, de configuration. Si la configuration manuelle doit être faite, elle devrait l'être sur le côté qui change le moins souvent --- c'est-à-dire, le client. Ceci suggère que le serveur devrait en quelque sorte apprendre du client s'il utilise la compression d'en-tête. En supposant la symétrie (c'est-à-dire, si la compression est utilisée, elle devrait l'être dans les deux directions) le serveur peut utiliser la réception d'un paquet compressé d'un client pour indiquer qu'il peut envoyer des paquets compressés à ce client. Cela conduit à l'algorithme suivant :

Il y a deux bits par ligne pour contrôler la compression d'en-tête : admis et activé. Si activé est établi, des paquets compressés sont envoyés, si il est à zéro, ils ne sont pas compressés. Si admis est établi, des paquets compressés peuvent être reçus et, si un paquet UNCOMPRESSED_TCP arrive lorsque activé est à zéro, activé sera mis à un /49/. Si un paquet compressé arrive lorsque admis est à zéro, il sera ignoré.

Note 49 Comme le tramage de [12] ne comporte pas de détection d'erreur, on devrait faire attention à ne pas faire de 'faux départ' de compression sur le serveur. La cohérence du paquet UNCOMPRESSED_TCP devrait être vérifiée (par exemple, l'exactitude de la somme de contrôle IP) avant d'activer la compression. L'arrivée de paquets COMPRESSED_TCP ne devrait pas être utilisée pour activer la compression.

Les clients sont configurés avec les deux bits établis (admis est toujours établi si activé l'est) et le serveur commence chaque session avec admis établi et activé à zéro. Le premier paquet compressé provenant du client (qui doit être un paquet UNCOMPRESSED_TCP) active la compression pour le serveur.

Annexe C Compression plus agressive

Comme on l'a noté au paragraphe 3.2.2, il existe des schémas facilement détectables dans le flux d'en-têtes compressés, qui indiquent que plus de compression pourrait être faite. Est-ce que cela en vaut la peine ?

Le datagramme compressé moyen a seulement sept bits d'en-tête /50/.

Note 50 Les essais fonctionnent avec plusieurs millions de paquets provenant d'une charge de trafic mixte (c'est-à-dire, des statistiques conservées sur le trafic d'un an de mon domicile à mon bureau) montrent que 80 % des paquets utilisent un des deux cas particuliers de codage et, donc, le seul en-tête est le gabarit de changement.

Le tramage doit être d'au moins un bit (pour coder le 'type') et sera plus probablement de deux ou trois octets. Dans la plupart des cas intéressants, il y aura au moins un octet de données. Finalement, la vérification de bout en bout --- la somme de contrôle TC --- doit être passée non modifiée /51/.

Note 51 Si quelqu'un essaye de vous vendre un schéma qui compresse la somme de contrôle TC 'dites non'. Quelques pauvres fous devront subir la triste expérience qui révèle que l'argument de bout en bout est une vérité d'évangile. Pire, comme le fou déjoue votre vérification d'erreur de bout en bout, vous pourrez payer le prix de votre éducation et ce ne sera pas le plus sage. Quel profit y a t-il à gagner deux octets au prix de la paix de l'esprit ?

Le tramage, les données et la somme de contrôle vont rester même si l'en-tête est complètement compressé de sorte que le changement de taille moyenne de paquet sera, au mieux, de quatre octets à trois octets et un bit --- en gros, une amélioration du délai de 25 % /52/.

Note 52 Noter encore qu'on doit se préoccuper du délai interactif de cet argument : les performances de transfert de données brutes seront dominées par le temps d'envoi des données et la différence entre trois et quatre octets d'en-tête sur un datagramme contenant des dizaines ou des centaines d'octets de données est, en pratique, indifférent.

Bien que cela puisse sembler significatif, sur une ligne à 2400 bit/s, cela signifie que les réponses d'écho de frappe prennent 25 ms plutôt que 29 ms. À l'étape actuelle de l'évolution humaine, cette différence n'est pas détectable.

Cependant, l'auteur admet timidement de pervertir ce schéma de compression pour un cas très particulier de problème d'acquisition de données : on avait un paquetage d'instrumentation et de contrôle flottant à 200 kV, communiquant avec le rez-de-chaussée via un système de télémétrie. Pour de nombreuses raisons (communication multiplexée, traitement en parallèle, récupération d'erreur, disponibilité de mises en œuvre éprouvées, etc.) il était pratique de parler avec le paquetage en utilisant TCP/IP. Cependant, comme la principale utilisation de la liaison de télémétrie était l'acquisition de données, elle avait été conçue avec une capacité de canal montant <0,5 % de celle de la liaison descendante. Pour tenir les budgets de délai d'application, les paquets de données étaient de 100 octets et, comme TCP accuse réception de tout autre paquet, la bande passante relative de la liaison montante pour les accusés de réception est a/200 où 'a' est la taille totale des paquets d'accusé de réception. En utilisant ce schéma dans cet article, le plus petit ack ferait quatre octets ce qui impliquerait une bande passante de liaison montante de 2 % de celle de la liaison descendante. Ce n'était pas possible, de sorte qu'on a utilisé le schéma décrit dans la note 15 : si le premier bit de la trame est un, cela signifie 'même en-tête compressé que la dernière fois'. Autrement, les deux prochains bits donnent un des types décrits au paragraphe 3.2. Comme la liaison avait une excellente correction d'erreur directe et que le trafic faisait seulement un bond, la somme de contrôle TC était compressée (quelle honte !) du type de paquet 'même en-tête' /53/ de sorte que la taille de l'en-tête total pour ces paquets était un bit.

Note 53 La somme de contrôle était régénérée dans le décompresseur et, bien sûr, la logique de 'rejet' était rendu considérablement plus agressive pour empêcher la propagation d'erreur.

Sur plusieurs mois de fonctionnement, plus de 99 % des en-têtes TCP/P de 40 octets ont été compressés à un bit /54/.

Note 54 On a entendu la suggestion que les besoins du 'temps réel' exigent l'abandon de TCP/IP en faveur d'un protocole 'léger' avec de plus petits en-têtes. Il est difficile d'envisager un protocole qui ait en moyenne moins d'un bit d'en-tête par paquet.

D Considérations pour la sécurité

Les considérations de sécurité ne sont pas traitées dans le présent mémoire.

E Adresse de l'auteur

Van Jacobson
Real Time Systems Group
Mail Stop 46A
Lawrence Berkeley Laboratory
Berkeley, CA 94720
mél : van@helios.ee.lbl.gov