

Groupe de travail Réseau
Request For Comments : 817
Traduction Claude Brière de L'Isle

David D. Clark
MIT Laboratory for Computer Science
juillet 1982

Modularité et efficacité dans la mise en œuvre de protocole

1. Introduction

De nombreux développeurs de protocoles ont fait la découverte déplaisante que leurs paquetages ne fonctionnent pas tout à fait aussi vite qu'ils l'espéraient. Le blâme pour ce problème largement observé a été attribué à diverses causes, allant de détails de la conception du protocole à la structure sous-jacente du système d'exploitation de l'hôte. La présente RFC va exposer certaines des raisons les plus couramment rencontrées pour lesquelles les mises en œuvre de protocoles semblent fonctionner lentement.

L'expérience suggère que l'un des facteurs les plus importants dans la détermination des performances d'une mise en œuvre est la manière dont elle est modularisée et intégrée dans le système d'exploitation de l'hôte. Pour cette raison, il est utile de discuter la question de comment une mise en œuvre est structurée, en même temps qu'on va examiner comment elle va s'exécuter. En fait, la présente RFC va avancer que la modularité est le principal responsable dans les tentatives pour obtenir de bonnes performances, de sorte que le concepteur est en face du délicat et inévitable compromis entre bonne structure et bonnes performances. De plus, le seul facteur qui détermine le plus fortement comment ce conflit peut être résolu n'est pas le protocole mais le système d'exploitation.

2. Considérations d'efficacité

L'efficacité comporte de nombreux aspects. Un aspect est l'envoi des données au coût de transmission minimum, qui est un aspect critique des communications des transporteurs courants, sinon dans les communications de réseau de zone locale. Un autre aspect est l'envoi des données à haut débit, qui peut n'être pas possible du tout si le réseau est très lent, mais qui peut être la contrainte de conception centrale lorsque on tire parti d'un réseau local qui a une forte bande passante brute. La considération finale est de faire cela au coût minimum des ressources de calcul. Ce dernier point peut être nécessaire pour obtenir de hauts débits, mais dans le cas d'un réseau lent peut n'être important qu'en ce que les ressources utilisées, par exemple les cycles de cpu, sont coûteux ou par ailleurs nécessaires. Il vaut de mentionner que ces différents objectifs sont souvent en conflit ; par exemple il est souvent possible de faire un compromis entre l'efficacité d'utilisation de l'ordinateur et l'efficacité d'utilisation du réseau. Donc, il peut n'y avoir rien pour une mise en œuvre de protocole d'objet général réussie.

La plus simple mesure de performances est le débit, mesuré en bits par seconde. Il vaut la peine de faire quelques calculs simples afin d'avoir une idée de la magnitude des problèmes en cause. Supposons que les données qu'on envoie d'une machine à une autre soient en paquets de 576 octets, le maximum généralement acceptable pour la taille de paquet Internet. Permettant la redondance de l'en-tête, cette taille de paquet permet 4288 bits dans chaque paquet. Si un débit utile de 10 000 bits par seconde est désiré, un paquet qui porte des données doit alors quitter l'hôte envoyeur environ toutes les 430 millisecondes, un peu plus de deux par seconde. Ceci n'est évidemment pas difficile à réaliser. Cependant, si on souhaite réaliser un débit de 100 kilobits par seconde, le paquet doit quitter l'hôte toutes les 43 millisecondes, et pour réaliser un mégabit par seconde, ce qui n'est pas du tout déraisonnable sur un réseau local à haut débit, les paquets doivent être espacés de pas plus de 4,3 millisecondes.

Ces derniers chiffres sont un peu plus alarmant pour fixer ses vues. De nombreux systèmes d'exploitation prennent une fraction substantielle de milliseconde juste pour servir une interruption. Si le protocole a été structuré comme un processus, il est nécessaire de passer par une programmation de processus avant que le code de protocole puisse même commencer à fonctionner. Si un morceau quelconque du paquetage de protocole ou de ses données doit être cherché sur un disque, on peut s'attendre à des délais de temps réel de l'ordre de 30 à 100 millisecondes. Si le protocole doit entrer en compétition pour des ressources de cpu avec les autres processus du système, il peut être nécessaire d'attendre un temps de programmation avant que le protocole puisse tourner. De nombreux systèmes ont un délai de programmation de 100 millisecondes ou plus. En considérant ces sortes de nombres, il devient immédiatement clair que le protocole doit être ajusté au système d'exploitation de manière stricte et efficace si on veut atteindre un débit raisonnable.

Une conclusion évidente est immédiatement suggérée par même la plus simple analyse. Sauf dans des circonstances très particulières, lorsque de nombreux paquets doivent être traités tout de suite, le coût du traitement d'un paquet est dominé

par des facteurs, tels que la programmation de cpu, qui sont indépendants de la taille du paquet. Cela suggère deux règles générales auxquelles devrait obéir toute mise en œuvre. D'abord, envoyer les données dans de gros paquets. Il est évident que si le temps de traitement par paquet est une constante, le débit sera alors directement proportionnel à la taille du paquet. Ensuite, ne jamais envoyer un paquet non nécessaire. Les paquets non nécessaires utilisent autant de ressources qu'un paquet plein de données, mais n'effectuent aucune fonction utile. La RFC 813, "Stratégie de fenêtre et d'accusé de réception dans TCP", discute d'un aspect de la réduction du nombre de paquets envoyés par octet de données utile. Le présent document mentionnera d'autres approches du même problème. L'analyse ci-dessus suggère qu'il y a deux parties principales au problème de la réalisation de bonnes performances de protocole. La première a à voir avec la façon dont la mise en œuvre de protocole est intégrée dans le système d'exploitation de l'hôte. La seconde se rapporte à la façon dont le paquetage de protocole lui-même est organisé en interne. Le présent document va examiner tour à tour ces deux aspects.

3. Protocole contre système d'exploitation

Il y a normalement trois façons raisonnables pour ajouter un protocole à un système d'exploitation. Le protocole peut être dans un processus qui est fourni par le système d'exploitation, ou il peut faire partie du noyau du système d'exploitation lui-même, ou il peut être mis dans un processeur de communications séparé ou le frontal d'une machine. Cette décision est fortement influencée par les détails de l'architecture du matériel et la conception du système d'exploitation ; chacune de ces trois approches a ses propres avantages et inconvénients.

Le "processus" est l'abstraction qu'utilisent la plupart des systèmes d'exploitation pour fournir l'environnement d'exécution pour les programmes utilisateurs. Un chemin très simple pour mettre en œuvre un protocole est d'obtenir un processus du système d'exploitation et de mettre en œuvre le protocole qui doit fonctionner dedans. Superficiellement, cette approche présente un certain nombre d'avantages. Comme des modifications au noyau ne sont pas nécessaires, la tâche peut être effectuée par quelqu'un qui n'est pas un expert de la structure du noyau. Comme il est souvent impossible de trouver quelqu'un qui soit expérimenté à la fois dans la structure du système d'exploitation et dans la structure du protocole, ce chemin, du point de vue de la gestion, est souvent extrêmement intéressant. Malheureusement, mettre un protocole dans un processus présente un certain nombre d'inconvénients, relatifs à la fois à la structure et aux performances. D'abord, comme on l'a exposé plus haut, la programmation de processus peut être une source significative de délais en temps réel. Il n'y a pas seulement le coût réel du passage au travers du programmeur, mais aussi le problème que le système d'exploitation peut n'avoir pas les bons outils de priorité pour amener rapidement le processus à s'exécuter chaque fois qu'il y a un travail à faire.

Structurellement, la difficulté de mettre un protocole dans un processus est que le protocole peut se trouver fournir des services, par exemple la prise en charge de flux de données, qui sont normalement obtenus en allant à des points d'entrée spéciaux du noyau. Selon les généralités du système d'exploitation, il peut être impossible de prendre un programme qui se lit d'habitude à travers un point d'entrée du noyau, et de le rediriger de façon à ce qu'il lise les données à partir d'un processus. L'exemple le plus extrême de ce problème survient lors de la mise en œuvre d'un serveur Telnet. Dans presque tous les systèmes, le manipulateur d'appareil pour les télétypes rattachés localement est situé à l'intérieur du noyau, et les programmes lisent et écrivent à partir de leur télétype en faisant des appels au noyau. Si le serveur Telnet est mis en œuvre dans un processus, il est alors nécessaire de prendre les flux de données fournis par le serveur Telnet et de les ramener à l'intérieur du noyau afin qu'ils simulent l'interface fournie par les télétypes locaux. Il est généralement le cas que des modifications particulières du noyau soient nécessaires pour réaliser cette structure, qui réduit quelque peu les avantages d'avoir retiré en premier lieu le protocole du noyau.

Il est donc clair qu'il y a des avantages à mettre le paquetage de protocole dans le noyau. Structurellement, il est raisonnable de voir le réseau comme un appareil, et les pilotes d'appareil sont traditionnellement contenus dans le noyau. Par hypothèse, on peut écarter les problèmes associés à la programmation de processus, au moins dans une certaine mesure, en plaçant le code à l'intérieur du noyau. Et il est évidemment plus facile de faire que les canaux du serveur Telnet simulent les canaux du télétype local si ils sont tous deux réalisés dans le même niveau dans le noyau.

Cependant, la mise en œuvre de protocoles dans le noyau a son propre ensemble de pièges. D'abord, les protocoles réseau ont une caractéristique qui n'est partagée par presque aucun autre appareil : ils exigent des actions assez complexes à effectuer par suite d'une fin de temporisation. Le problème de cette exigence est que le noyau n'a souvent pas de facilité par laquelle un programme puisse être mis à exécution par suite de l'événement de fin de temporisation. Ce qui est réellement nécessaire est bien sûr une sorte de processus particulier à l'intérieur du noyau. La plupart des systèmes n'ont pas ce mécanisme. Faute de quoi, le seul mécanisme d'exécution disponible est le lancement au moment de l'interruption.

Il n'y a pas d'inconvénient majeur à mettre en œuvre un protocole qui se lance au moment d'interruption. D'abord, les actions effectuées peuvent être assez complexes et consommatrices de temps, par rapport à la quantité maximum de temps pendant laquelle le système d'exploitation est prêt à se consacrer à servir une interruption. Des problèmes peuvent surgir si les interruptions sont masquées pendant trop longtemps. Cela est particulièrement mauvais lorsque le lancement est le

résultat d'une interruption d'horloge, ce qui peut impliquer que l'interruption d'horloge est masquée. Ensuite, l'environnement fourni par un manipulateur d'interruption est usuellement extrêmement primitif comparé à l'environnement d'un processus. Il y a habituellement diverses facilités système qui ne sont pas disponibles lors du fonctionnement d'un manipulateur d'interruption. La plus importante d'entre elles est la capacité à suspendre l'exécution pendant l'arrivée d'un événement ou message. C'est une règle cardinale de presque tous les systèmes d'exploitation connus qu'on ne doit pas invoquer le programmeur lors du fonctionnement avec un manipulateur d'interruption. Donc, le programmeur qui est forcé de mettre en œuvre tout ou partie de ce paquetage de protocole comme un manipulateur d'interruption doit être de la meilleure sorte d'expert dans le système d'exploitation impliqué et doit être prêt à des sessions de développement remplies d'obscurités et d'erreurs qui mettent en panne non seulement le paquetage de protocole mais tout le système d'exploitation.

Un problème final du traitement au moment d'interruption est que le programmeur du système n'a pas le contrôle du pourcentage du temps système utilisé par le manipulateur du protocole. Si un grand nombre de paquets arrivent, provenant d'un hôte étranger qui fonctionne soit mal, soit trop vite, tout le temps peut se passer dans le manipulateur d'interruption, tuant effectivement le système.

Il y a d'autres problèmes associés avec la mise des protocoles dans le noyau d'un système d'exploitation. Le plus simple problème souvent rencontré est que l'espace d'adresse du noyau est simplement trop petit pour contenir la quantité de code en question. C'est une sorte de problème assez artificielle, mais c'est néanmoins un problème sévère dans de nombreuses machines. C'est une expérience extraordinairement déplaisante que de faire une mise en œuvre en sachant que pour chaque octet du nouveau dispositif entré on doit trouver un autre octet du vieux dispositif à éliminer. Il ne sert à rien d'espérer une mise en œuvre efficace et générale avec ce genre de contraintes. Un autre problème est que le paquetage de protocole, une fois qu'il a été soigneusement introduit dans le système d'exploitation, peut devoir être refait chaque fois que change le système d'exploitation. Si le protocole et le système d'exploitation ne sont pas entretenus par le même groupe, cela rend la maintenance du paquetage de protocole un casse tête perpétuel.

La troisième option de mise en œuvre de protocole est de prendre le paquetage de protocole et de le déplacer entièrement hors de la machine, sur un processeur séparé dédié à ce genre de tâche. Une telle machine est souvent décrite comme un processus de communications ou un processeur frontal. Il y a plusieurs avantages à cette approche. Premièrement, le système d'exploitation sur le processeur de communications peut être fait sur mesure pour précisément cette sorte de tâche. Cela rend beaucoup plus facile la tâche de mise en œuvre. Ensuite, on n'a pas besoin de refaire le travail pour chaque machine à laquelle on doit ajouter le protocole. Il est possible de réutiliser la même machine frontale sur différents ordinateurs hôtes. Comme la tâche n'a pas besoin d'être effectuée aussi souvent, on peut espérer qu'on fera plus attention à la faire bien. Avec une mise en œuvre soignée dans un environnement qui est optimisé pour ce type de tâche, le paquetage résultant devrait se révéler très efficace. Malheureusement, il y a aussi des problèmes avec cette approche. Il y a bien sûr un problème financier qui est associé à l'achat d'un ordinateur supplémentaire. Dans de nombreux cas, ce n'est pas un problème du tout car ce coût est négligeable comparé à ce que coûterait le programmeur pour faire le travail du cœur de chaîne lui-même. Plus fondamental, l'approche du processeur de communications ne règle complètement aucun des problèmes soulevés précédemment. La raison en est que le processeur de communications, comme c'est une machine séparée, doit être rattaché au cœur de chaîne par un mécanisme. Quel que soit ce mécanisme, du code est nécessaire dans le cœur de chaîne pour le traiter. On peut objecter que le programme pour traiter le processeur de communications est plus simple que le programme pour mettre en œuvre le paquetage de protocole tout entier. Même si il en est ainsi, le paquetage d'interface de processeur de communications est quand même un protocole par nature, avec tous les mêmes problèmes structurels. Donc, on est quand même confronté à toutes les questions soulevées ci-dessus. En plus de ces problèmes, il y en a quelques autres, des problèmes plus subtils associés à la mise en œuvre en externe d'un protocole. On reviendra à ces problèmes plus tard.

Il y a un moyen de rattacher un processeur de communications à un hôte cœur de chaîne qui écarte tous les problèmes de mise en œuvre de cœur de chaîne, qui est d'utiliser une interface préexistante sur la machine hôte comme l'accès par lequel est rattaché un processeur de communications. Cette stratégie est souvent utilisée en désespoir de cause lorsque le logiciel sur l'ordinateur hôte est si intraitable qu'il ne peut être changé d'aucune façon. Malheureusement, il est presque inévitablement le cas que toutes les interfaces disponibles sont totalement inutilisables à cette fin, de sorte que les résultats sont au mieux insatisfaisants. La façon la plus courante dont se produit cette forme de rattachement est lorsque une connexion réseau est utilisée pour simuler des télétypes locaux. Dans ce cas, le processeur frontal peut être rattaché au cœur de chaîne en fournissant simplement un certain nombre de câbles en sortie du processeur de cœur de chaîne, correspondant chacun à une connexion, qui sont branchés sur les accès du télétype sur l'ordinateur cœur de chaîne. (À cause de l'apparence de la configuration physique qui résulte de cet arrangement, Michael Padlipsky a décrit cela comme l'approche de la "trayeuse" de la mise en réseau d'ordinateurs.) Cette stratégie résout le problème immédiat de la fourniture d'un accès distant à un hôte, mais elle est extrêmement inflexible. Les canaux qui sont fournis à l'hôte sont contraints par le logiciel de l'hôte à un seul objet, la connexion à distance. Il est impossible de les utiliser pour aucun autre objet, comme le transfert de fichiers ou l'envoi de messagerie, de sorte que l'hôte est intégré dans l'environnement réseau d'une manière extrêmement limitée et inflexible. Si c'est le mieux qui puisse être fait, cela devrait alors être toléré. Autrement, les mises en œuvre devraient être fortement encouragées à emprunter une approche plus souple.

4. Mise en couche de protocole

L'exposé précédent a suggéré qu'il y a une décision à prendre quant au lieu où un protocole devrait être mis en œuvre. En fait, la décision est beaucoup plus compliquée que cela, car le but n'est pas de mettre en œuvre un seul protocole, mais de mettre en œuvre toute une famille de couches de protocoles, en commençant par un pilote d'appareil ou un pilote de réseau local comme base, puis IP et TCP, et en atteignant finalement le protocole spécifique d'application, comme Telnet, FTP et SMTP au sommet. Il est clair que la plus profonde de ces couches est quelque part au sein du noyau, car le pilote de l'appareil physique pour le réseau est presque inévitablement situé là. Il est également clair que les couches supérieures de ce paquetage, qui fournissent à l'utilisateur sa capacité à effectuer la fonction de connexion à distance ou d'envoi de messages, ne sont pas entièrement contenues dans le noyau. Donc, la question n'est pas de savoir si la famille de protocoles doit être à l'intérieur ou à l'extérieur du noyau, mais comment elle devra être coupée en deux entre la partie intérieure et la partie extérieure.

Comme les protocoles viennent gentiment en couches, une proposition évidente est qu'une des interfaces de couche devrait être le point auquel les composants intérieurs et extérieurs sont découpés. La plupart des systèmes ont été mis en œuvre de cette façon, et beaucoup ont été rendus assez efficaces. Un endroit évident pour faire le découpage est l'interface supérieure de TCP. Comme TCP fournit un système d'octets bidirectionnel, qui est assez similaire à la facilité d'entrée/sortie fournie par la plupart des systèmes d'exploitation, il est possible de faire que l'interface à TCP simule l'interface aux autres appareils existants. Sauf en matière d'ouverture de connexion, et pour traiter de défaillances particulières, le logiciel qui utilise TCP n'a pas besoin de savoir que c'est une connexion réseau plutôt qu'un flux local d'entrée/sortie qui assure la fonction de communications. Cette approche met bien TCP à l'intérieur du noyau, ce qui soulève tous les problèmes évoqués plus haut. Cela soulève aussi le problème que l'interface à la couche IP peut, si le programmeur n'y veille pas, devenir excessivement enterrée à l'intérieur du noyau. Il faut se rappeler que des choses autres que TCP sont supposées fonctionner par dessus IP. L'interface IP doit être rendue accessible, même si TCP se tient par dessus à l'intérieur du noyau.

Un autre endroit évident pour le découpage est au-dessus de Telnet. L'avantage de découper au dessus de Telnet est que cela résout le problème d'avoir des canaux de connexion distante qui émulent des canaux de télétype locaux. L'inconvénient de mettre Telnet dans le noyau est que la quantité de code qui doit maintenant y être incluse devient remarquablement grande. Dans certaines mises en œuvre anciennes, la taille du paquetage réseau, lorsque inclus les protocoles au niveau de Telnet, rivalise en taille avec le reste du superviseur. Cela conduit au vague sentiment que tout cela ne va pas.

Toute tentative de découpe à travers une frontière de couche inférieure, par exemple entre internet et TCP, révèle un problème fondamental. La couche TCP, aussi bien que la couche IP, effectue une fonction de démultiplexage sur les datagrammes entrants. Tant que l'en-tête TCP n'a pas été examiné, il n'est pas possible de savoir quel utilisateur est la destination ultime du paquet. Donc, si TCP, comme entité, est déplacé à l'extérieur du noyau, il est nécessaire de créer un processus séparé appelé le processus TCP, qui effectue la fonction de multiplexage TCP, et probablement aussi tout le reste du traitement TCP. Cela signifie que les données entrantes destinées à un processus utilisateur n'impliquent pas qu'une programmation du processus utilisateur, mais de programmer d'abord le processus TCP.

Cela suggère une autre stratégie de structuration qui découpe à travers les protocoles, non pas selon les limites de couches établies, mais selon des frontières fonctionnelles en rapport avec le démultiplexage. Dans cette approche, certaines parties de IP et certaines parties de TCP sont placées dans le noyau. La quantité de code placée là est suffisante pour que, lorsque arrive un datagramme entrant, il soit possible de savoir à quel processus ce datagramme est finalement destiné. Le datagramme est alors acheminé directement au processus final, où les traitements IP et TCP supplémentaires sont effectués sur lui. Cela retire du noyau toute exigence d'actions fondées sur la temporisation, car elles peuvent être faites par le processus fourni par l'utilisateur. Cette structure présente l'avantage supplémentaire de réduire la quantité de code requise dans le noyau, de sorte qu'elle convient pour les systèmes où l'espace du noyau est compté. La RFC 814, intitulée "Noms, adresses, accès, et chemins" expose plus en détails cette stratégie de découpage assez orthogonale.

On trouvera un exposé se rapportant à la mise en couches et au multiplexage de protocoles dans Cohen et Postel [1].

5. Casser les barrières

En fait, la mise en œuvre devrait être sensible à la possibilité de stratégies de découpage encore plus particulières qui divisent les diverses couches de protocole entre le noyau et le ou les processus utilisateurs. Le résultat de la stratégie proposée ci-dessus était que cette partie de TCP devrait s'exécuter dans le processus de l'utilisateur. En d'autres termes, au lieu d'avoir un processus TCP pour le système, il y a un processus TCP par connexion. Avec cette architecture, il n'est plus nécessaire d'imaginer que tous les TCP soient identiques. Un TCP pourrait être optimisé pour les applications à haut débit,

telles que de transfert de fichiers. Un autre TCP pourrait être optimisé pour les petites applications à faibles délais comme Telnet. En fait, il serait possible de produire un TCP qui serait en quelque sorte intégré avec le Telnet ou FTP par dessus lui. Une telle intégration est extrêmement importante, car elle peut conduire à un type d'efficacité que les structures plus traditionnelles sont incapables de produire. Le présent article a souligné plus haut qu'une des règles importantes pour réaliser l'efficacité était d'envoyer le nombre minimum de paquets pour une certaine quantité de données. L'idée de mise en couche de protocole interagit très fortement (et de façon négative) avec cet objectif, parce que des couches indépendantes ont des idées indépendantes sur le moment où les paquets doivent être envoyés, et sauf si ces couches peuvent être amenées plus ou moins à coopérer, les paquets supplémentaires vont s'écouler. Le meilleur exemple de cela est le fonctionnement du serveur Telnet en mode d'écho distant d'un caractère à la fois par dessus TCP. Lorsque un paquet contenant un caractère arrive à l'hôte serveur, chaque couche a une réponse différente à ce paquet. TCP a une obligation d'accuser réception du paquet. L'un ou l'autre du serveur Telnet ou de la couche application au dessus a l'obligation de faire écho au caractère reçu dans le paquet. Si le caractère est une séquence de contrôle Telnet, Telnet a alors des actions supplémentaires qu'il doit effectuer en réponse au paquet. Le résultat de cela, dans la plupart des mises en œuvre, est que plusieurs paquets sont renvoyés en réponse à celui qui arrive. Combiner tous ces messages de retour en un seul paquet est important pour plusieurs raisons. D'abord, cela réduit bien sûr le nombre de paquets envoyés sur le réseau, ce qui réduit directement les charges supportées dans de nombreuses structures de tarifs des transporteurs. Ensuite, cela réduit le nombre d'actions de programmation qui vont survenir au sein des deux hôtes, ce qui, comme on l'a exposé ci-dessus, est extrêmement important pour améliorer le débit.

La façon de réaliser cet objectif de partage de paquets est de casser la barrière entre les couches des protocoles, d'une manière très restreinte et prudente, afin qu'une quantité limitée d'informations puisse franchir la barrière pour permettre à une couche d'optimiser son comportement par rapport aux désirs des couches au dessus et en dessous d'elle. Par exemple, cela représenterait une amélioration si TCP, lorsque il reçoit un paquet, pouvait demander à la couche supérieure si il vaudrait ou non la peine de faire une pause de quelques millisecondes avant d'envoyer un accusé de réception afin de voir si la couche supérieure aurait quelques données sortantes à envoyer. Attendre un peu avant d'envoyer l'accusé de réception produit précisément la bonne sorte d'optimisation si le client de TCP est le serveur Telnet. Cependant, s'attarder avant d'envoyer un accusé de réception est absolument inacceptable si TCP est utilisé pour un transfert de fichier, car dans le transfert de fichier il n'y a presque jamais de données qui s'écoulent dans la direction inverse, et le retard d'envoi de l'accusé de réception se traduit probablement directement en un retard de l'obtention des prochains paquets. Donc, TCP doit en savoir un peu sur les couches au dessus de lui pour ajuster en tant que de besoin ses performances.

Il serait possible d'imaginer un TCP d'usage général qui serait équipé de toutes sortes de mécanismes spéciaux par lesquels il interrogerait la couche supérieure et modifierait son comportement en conséquence. Dans les structures suggérées ci-dessus, dans lesquelles il n'y a pas un mais plusieurs TCP, le TCP peut simplement être modifié de telle sorte qu'il produise le comportement correct de façon naturelle. Cette structure présente l'inconvénient qu'il y aurait plusieurs mises en œuvre de TCP qui existeraient sur une seule machine, ce qui peut signifier plus de casses têtes de maintenance si il y a un problème et que TCP doit être changé. Cependant, il est aussi probable que chaque TCP sera substantiellement plus simple que le TCP d'utilisation générale qui devrait autrement être construit. Il y a quelques projets expérimentaux qui sont actuellement en cours et qui suggèrent que cette approche pourrait rendre substantiellement plus facile la conception d'un TCP, ou de presque toutes les autres couches, de sorte que l'effort total impliqué par la construction d'un paquetage complet serait moindre si on suivait cette approche. Celle-ci n'est en aucune façon généralement acceptée, mais mérite considération.

La conclusion générale à tirer de cette sorte de considérations est qu'une frontière de couche a à la fois des avantages et des inconvénients. Une frontière de couche visible, avec une interface bien spécifiée, assure une forme d'isolation entre deux couches ce qui permet d'en changer une avec l'assurance que l'autre ne va pas cesser de fonctionner pour autant. Cependant, une frontière de couche ferme conduit presque inévitablement à un fonctionnement inefficace. On peut facilement le voir par analogie avec d'autres aspects des systèmes d'exploitation. Considérons, par exemple, les systèmes de fichiers. Un système d'exploitation typique fournit un système de fichiers, qui est une représentation très abstraite d'un disque. L'interface est très formalisée, et supposée être très stable. Cela rend très facile aux simples utilisateurs d'avoir accès aux disques sans avoir à écrire de grosses quantités de logiciel. L'existence d'un système de fichiers est clairement bénéfique. D'un autre côté, il est clair que l'interface restreinte à un système de fichiers conduit presque inévitablement à l'inefficacité. Si l'interface est organisée comme une lecture/écriture séquentielle d'octets, il y aura alors des gens qui souhaiteront faire des transferts à haut débit qui ne pourront pas réaliser leur souhait. Si l'interface est à mémoire virtuelle, d'autres utilisateurs vont alors regretter la nécessité de construire une interface de flux d'octets par dessus le fichier transposé en mémoire. L'inefficacité la plus critiquable résulte lorsque un paquetage très sophistiqué, comme un paquetage de gestion de base de données, doit être construit par dessus un système d'exploitation existant. Presque inévitablement, la mise en œuvre du système de base de données tentera de rejeter le système de fichiers et d'obtenir l'accès direct aux disques. Ils ont sacrifié la modularité à l'efficacité.

Le même conflit apparaît dans le réseautage, sous une forme assez extrême. Le concept de protocole est toujours inconnu de la plupart des programmeurs débutants et les effraie. L'idée qu'ils pourraient avoir à mettre en œuvre un protocole, ou même une partie d'un protocole, au titre d'un paquetage d'application, est une pensée qui fait peur. Et il y a donc une

grande pression pour cacher la fonction du réseau derrière une barrière très solide. D'un autre côté, la sorte d'inefficacité qui en résulte est particulièrement indésirable, car elle aboutit, entre autres choses, à augmenter le coût des ressources de communications utilisées pour réaliser le but de l'application. Dans les cas où on doit payer ses coûts de communications, ils se révèlent habituellement être le coût dominant au sein du système. Donc, faire un travail de paquetage excessivement bon pour les protocoles d'une façon non flexible a un impact direct sur l'augmentation du coût de la ressource critique au sein du système. C'est un dilemme qui ne sera probablement résolu que lorsque les programmeurs deviendront un peu moins effrayés par les protocoles, et qu'ils seront d'accord pour introduire une certaine quantité de structure de protocole dans leurs programmes d'application, tout comme les programmes d'application introduisent aujourd'hui des parties de systèmes de gestion de base de données dans la structure de leurs programmes d'application.

Un exemple extrême de mise du paquetage de protocole derrière une barrière de couche ferme survient lorsque le paquetage de protocole est relégué dans un processeur frontal. Dans ce cas, l'interface du protocole est un autre protocole. Il est difficile d'imaginer comment construire une étroite coopération entre les couches lorsque elle sont aussi séparées. De façon réaliste, un des prix qui doivent être associés à une mise en œuvre aussi modularisée physiquement est que les performances vont en souffrir. Bien sûr, un processeur séparé pour les protocoles pourrait être intégré très étroitement dans l'architecture du cœur de chaîne, avec des signaux de coordination inter processeur, une mémoire partagée, et des dispositifs similaires. Une telle modularité physique pourrait très bien fonctionner, mais il y a peu d'expériences documentées sur cette architecture étroitement couplée à la prise en charge de protocoles.

6. Efficacité du traitement du protocole

Jusqu'à présent, le présent document a examiné comment un paquetage de protocole devrait être découpé en modules, et comment des modules devraient être répartis entre des machines autonomes, le noyau du système d'exploitation, et un ou plusieurs processus utilisateurs. Il est maintenant temps d'examiner l'autre moitié de la question de l'efficacité, qui est ce qu'on peut faire pour accélérer l'exécution de ces programmes qui en fait mettent en œuvre les protocoles. On va faire des observations spécifiques sur TCP et IP, puis conclure par quelques généralités.

IP est un protocole simple, en particulier par rapport au traitement des paquets normaux, de sorte qu'il devrait être facile de le faire fonctionner de façon efficace. Le seul domaine d'une certaine complexité par rapport au traitement réel des paquets est celui de la fragmentation et du réassemblage. Le lecteur se référera à la RFC 815, intitulée "Algorithmes de réassemblage des paquets IP", pour les considérations spécifiques de ce point.

La plupart des coûts dans la couche IP viennent des fonctions de recherches dans les tableaux, par opposition aux fonctions de traitement des paquets. Un paquet sortant exige que deux fonctions de traduction soient effectuées. L'adresse Internet doit être traduite en un routeur cible, et une adresse de routeur doit être traduite en un numéro de réseau local (si l'hôte est rattaché à plus d'un réseau). Il est facile de construire une mise en œuvre simple de ces fonctions de recherche dans les tableaux qui en fait fonctionnent très mal. Le programmeur devrait garder à l'esprit qu'il peut y avoir plus d'un millier de numéros de réseau dans une configuration normale. La recherche linéaire d'un tableau d'un millier d'entrées sur chaque paquet est extrêmement inappropriée. En fait, il peut valoir la peine de demander à TCP de mettre en antémémoire un conseil pour chaque connexion, qui peut être repassé à IP chaque fois qu'un paquet est envoyé, pour essayer d'éviter la surcharge d'une recherche dans le tableau.

TCP est un protocole plus complexe, qui présente beaucoup plus d'opportunités pour que les choses se passent mal. Il y a un domaine dont il est généralement accepté qu'il cause une surcharge notable et substantielle au titre du traitement TCP. C'est le calcul de la somme de contrôle. Il serait bien que ce coût puisse être dans une certaine mesure évité, mais l'idée d'une somme de contrôle de bout en bout est absolument centrale pour le fonctionnement de TCP. Aucune mise en œuvre d'hôte ne devrait envisager d'omettre la validation d'une somme de contrôle sur les données entrantes.

Diverses astuces habiles ont été utilisées pour essayer de minimiser le coût du calcul de la somme de contrôle. Si il est possible d'ajouter des instructions micro codées supplémentaires à la machine, une instruction de somme de contrôle est le candidat le plus évident. Comme le calcul de la somme de contrôle implique de prendre chaque octet du segment et de l'examiner, il est possible de combiner l'opération de calcul de la somme de contrôle avec l'opération de copie du segment d'une localisation à l'autre. Comme un certain nombre de copies des données sont probablement déjà requises au titre de la structure de traitement, cette sorte de partage serait concevable si elle ne causait pas trop de soucis à la modularité du programme. Finalement, le calcul de la somme de contrôle semble être un domaine où un peu d'attention aux détails de l'algorithme utilisé peut faire une différence drastique dans le débit du programme. Le système Multics fournit un des meilleurs cas d'études de cela, car Multics est aussi mal organisé pour effectuer cette fonction que n'importe quelle autre machine qui met en œuvre TCP. Multics est une machine à mots de 36 bits, avec quatre octets de 9 bits par mot. L'octet de huit bits d'un segment TCP est déposé empaqueté dans la mémoire, en ignorant les limites de mots. Cela signifie que lorsque il est nécessaire de prendre les données comme un ensemble d'unités de 16 bits pour les besoins de l'addition pour calculer les sommes de contrôle, d'horribles masquages et glissements sont exigés pour chaque valeur de 16 bits. Une

ancienne version d'un programme utilisant cette stratégie demandait 6 millisecondes pour faire la somme de contrôle d'un segment de 576 octets. Évidemment, à ce point, le calcul de somme de contrôle devient le goulet d'étranglement central du débit. Un recodage plus soigneux de cet algorithme a réduit le temps de traitement de la somme de contrôle à moins d'une milliseconde. La stratégie utilisée était extrêmement sale. Elle impliquait d'ajouter les mots choisis avec soin du domaine dans lequel reposent les données, sachant que pour ces mots particuliers, les valeurs de 16 bits étaient correctement alignées à l'intérieur des mots. C'est seulement après que l'addition a été faite que les diverses sommes sont déplacées, et finalement ajoutées pour produire la somme de contrôle finale. Cette sorte de programmation hautement spécialisée n'est probablement pas acceptable si elle est utilisée n'importe où au sein d'un système d'exploitation. Elle est clairement appropriée pour une fonction très localisée qui peut être clairement identifiée dans un goulet d'étranglement extrême des performances.

Un autre domaine du traitement de TCP qui peut causer des problèmes de performances est la surcharge de l'examen de tous les fanions et options possibles qui surviennent dans chaque paquet entrant. Un article de Bunch et Day [2], affirme que la surcharge du traitement d'en-tête de paquet est en fait un important facteur de limitation du calcul de débit. Toutes les expériences de mesures ne tendent pas à valider ce résultat. Cependant, quelle que soit la mesure dans laquelle c'est vrai, il y a une stratégie évidente que devraient utiliser les mises en œuvre pour la conception de leur programme. Elles devraient construire leur programme pour optimiser le cas attendu. Il est facile, en particulier lors de la première conception d'un programme, de porter une égale attention à tous les résultats possibles de chaque essai. Cependant, en pratique, ceci a peu de chances d'arriver. Un TCP devrait être construit avec l'hypothèse que le prochain paquet à arriver n'aura absolument rien de particulier, et sera le suivant qui est attendu dans l'espace de la séquence. Un ou deux essais sont suffisants pour déterminer que l'ensemble attendu de fanions de contrôle est établi. (Le fanion ACK devrait être établi; le fanion Push peut être établi ou non. Aucun autre fanion ne devrait être établi.) Un essai est suffisant pour déterminer que le numéro de séquence du paquet entrant est supérieur d'un au dernier numéro de séquence reçu. Dans presque tous les cas, ce sera le résultat réel. Là encore, en utilisant le système Multics comme exemple, l'échec à optimiser le cas de réception du numéro de séquence attendu a un effet détectable sur les performances du système. Un problème particulier survient lorsque un certain nombre de paquets arrivent à la fois. TCP a tenté de traiter tous ces paquets avant d'alerter l'utilisateur. Il en résulte que au moment où le dernier paquet est arrivé, il y avait une liste de paquets à la file avec plusieurs éléments dessus. Lorsque un nouveau paquet est arrivé, il cherche sur la liste pour trouver la place où le paquet devrait être inséré. Évidemment, la recherche sur la liste devrait se faire du plus fort numéro de séquence au plus faible numéro de séquence, parce qu'on s'attend à recevoir un paquet qui vient après ceux déjà reçus. Par erreur, la recherche sur la liste était faite d'avant en arrière, commençant par les paquets avec le plus faible numéro de séquence. Le temps passé à chercher à reculer sur cette liste était facilement détectable dans les mesures.

D'autres structures de données peuvent être organisées pour optimiser l'action qui est normalement effectuée sur elles. Par exemple, la file d'attente de retransmission est en fait très rarement utilisée pour la retransmission, de sorte qu'elle ne devrait pas être organisée pour optimiser cette action. En fait, elle devrait être organisée pour optimiser l'élimination des choses qui en proviennent lorsque leur accusé de réception arrive. Dans de nombreux cas, le plus facile pour le faire est de ne pas sauvegarder du tout le paquet, mais de le reconstruire seulement si il a besoin d'être retransmis, en commençant par les données comme elles ont été mises à l'origine dans la mémoire tampon par l'utilisateur.

Il y a un autre point, au moins aussi important que d'optimiser le cas courant, qui est d'éviter de copier des données plus souvent que nécessaire. Un autre résultat du TCP Multics se révélera éclairant ici. Multics prend entre deux et trois millisecondes au sein de la couche TCP pour traiter un paquet entrant, selon sa taille. Pour un paquet de 576 octets, les trois millisecondes sont utilisées approximativement comme suit : une milliseconde est utilisée à calculer la somme de contrôle ; six cents microsecondes sont passées à copier les données. (Les données sont copiées deux fois, à 3 millisecondes la copie.) Une de ces opérations de copie pourrait correctement être incluse au titre du coût de la somme de contrôle, car elle est faite pour obtenir les données sur une frontière connue de mot pour optimiser l'algorithme de somme de contrôle. Cependant, la copie effectuée aussi en même temps un autre transfert nécessaire. Le traitement de l'en-tête et le reséquençage du paquet prend 0,7 milliseconde. Le reste du temps est utilisé en divers traitements, comme de retirer les paquets de la file d'attente de retransmission qui sont acquittés par ce paquet. La copie des données est la seconde opération la plus coûteuse après la somme de contrôle des données. Certaines mises en œuvre, souvent à cause d'une modularité excessivement en couches, finissent par copier les données un grand nombre de fois. D'autres mises en œuvre finissent par copier les données parce qu'il n'y a pas de mémoire partagée entre les processus, et que les données doivent être déplacées d'un processus à l'autre via une opération du noyau. Si les quantités traitées par cette activité ne sont pas strictement contrôlées, elles vont rapidement devenir le goulet majeur d'étranglement des performances.

7. Conclusions

Le présent document a traité deux aspects de l'obtention des performances à partir d'une mise en œuvre de protocole, la façon dont le protocole est mis en couches et intégré dans le système d'exploitation, et la façon dont le traitement détaillé du paquet est optimisé. Ce serait bien si l'un ou l'autre de ces coûts pouvait complètement dominer, de sorte que toute

l'attention puisse se concentrer sur lui. Malheureusement, ce n'est pas le cas. Selon la sorte particulière de trafic qu'on obtient, par exemple, si on a des paquets Telnet d'un octet ou un transfert de fichier avec des paquets de taille maximum à la vitesse maximum, on peut s'attendre à voir l'un ou l'autre coût devenir le goulet d'étranglement majeur du débit. La plupart des développeurs qui ont étudié leurs programmes pour tenter de comprendre où passe le temps sont arrivés à la conclusion désagréable qu'il se répartit également entre toutes les parties de leur programme. À l'exception possible du traitement de la somme de contrôle, très peu de gens ont jamais trouvé que leurs problèmes de performances soient dus à un seul horrible goulet d'étranglement qu'on pourrait réparer par un seul coup de programmation inventive. Les performances sont plutôt quelque chose qui a été amélioré par un réglage pénible du programme entier.

La plupart des discussions sur les protocoles commencent par introduire le concept de mise en couche, qui tend à suggérer que la mise en couche est fondamentalement une idée merveilleuse qui devrait faire partie de tous les aspects des protocoles. En fait, la mise en couche n'est pas qu'une bénédiction. Il est clair qu'une interface de couche est nécessaire chaque fois que plus d'un client d'une couche particulière doit être autorisé à utiliser la même couche. Mais une interface, précisément parce qu'elle est fixée, conduit inévitablement à un manque de compréhension complète de ce qu'une couche particulière souhaite obtenir d'une autre. Cela doit conduire à l'inefficacité. De plus, la mise en couche est un piège potentiel en ce qu'on est tenté de penser qu'une limite de couche, qui est un artifice de la procédure de spécification, est en fait la frontière appropriée à utiliser pour modulariser la mise en œuvre. Là encore, dans certains cas, une couche dans l'architecture doit correspondre à une couche dans la mise en œuvre, précisément de telle sorte que plusieurs clients puissent avoir accès à cette couche d'une manière raisonnablement directe. Dans d'autres cas, un réarrangement astucieux des frontières du module mis en œuvre pour correspondre à diverses fonctions, comme le démultiplexage des paquets entrants, ou l'envoi de paquets sortants asynchrones, peut conduire à des améliorations de performances inattendues comparées à des stratégies de mise en œuvre plus traditionnelles. Finalement, de bonnes performances sont quelque chose qu'il est difficile de redistribuer dans un programme existant. Comme les performances sont influencées, non seulement par les bons détails, mais par la structure du gros œuvre, il peut se trouver qu'afin d'obtenir une amélioration substantielle de performances, il soit nécessaire de refaire complètement le programme depuis le début. C'est un gros désappointement pour les programmeurs, spécialement ceux qui font une mise en œuvre de protocole pour la première fois. Les programmeurs qui sont un peu inexpérimentés et ne sont pas familiers des protocoles sont suffisamment concernés par l'obtention d'un programme logiquement correct pour n'avoir pas la capacité de penser en même temps aux performances de la structure qu'ils construisent. C'est seulement après qu'ils ont réalisé un programme logiquement correct qu'ils découvrent qu'ils l'ont fait d'une façon qui empêche de réelles performances. Évidemment, il est plus difficile de concevoir un programme en pensant dès le départ à la fois à la correction logique et aux performances. Avec le temps, lorsque le groupe des développeurs en saura plus sur les structures appropriées à utiliser pour construire des protocoles, il sera possible de travailler sur un projet de mise en œuvre en ayant plus confiance que la structure est rationnelle, que le programme va fonctionner, et qu'il va fonctionner bien. Ceux qui parmi nous mettent en œuvre des protocoles ont le privilège d'être à l'avant garde de ce processus d'apprentissage. Il ne devrait pas être surprenant que parfois nos programmes souffrent de l'incertitude que nous leur faisons porter.

Références

- [1] Cohen and Postel, "On Protocol Multiplexing", Sixth Data Communications Symposium, ACM/IEEE, novembre 1979.
- [2] Bunch and Day, "Control Structure Overhead in TCP", Trends and Applications: Computer Networking, NBS Symposium, mai 1980.