Chapitre 7

Les protocoles de transport

Le service fourni par IP n'étant pas fiable, il faut implanter par-dessus un protocole supplémentaire, en fonction de la qualité de service dont les applications ont besoin. Pour les échanges de données exigeant une grande fiabilité, le protocole de transport TCP (*Transmission Control Protocol*) est utilisé. Pour ceux qui ne nécessitent pas une telle fiabilité, un protocole de transport plus simple, appelé UDP (*User Datagram Protocol*) fournit un service de bout en bout en mode sans connexion. Avant de décrire les protocoles TCP et UDP, nous allons voir les notions qui leur sont communes.

Le protocole de transport, greffé au-dessus d'IP, est choisi en fonction de la qualité de service souhaitée par l'application. Celle-ci choisira l'un des deux protocoles disponibles : TCP, si elle souhaite une grande fiabilité de l'échange des données, ou UDP si elle ne souhaite pas être ralentie par la gestion des services assurant l'intégrité des données. Les deux diffèrent par bien des aspects mais utilisent des concepts communs, notamment les notions de ports, de sockets et de total de contrôle.

1 Notions utilisées dans les protocoles de transport

Un protocole de transport offre aux applications situées sur la machine de l'utilisateur une interface leur permettant d'utiliser les services offerts par le ou les réseaux physiques sous-jacents. Comme plusieurs applications sont susceptibles d'accéder au réseau, il faut les identifier sans ambiguïté ; on utilise pour cela la notion de *port*. Par ailleurs, les extrémités qui échangent des données sont repérées grâce à la notion de *socket*.

Un protocole de transport souhaite en outre s'assurer que l'en-tête contenant les informations nécessaires à la gestion du protocole n'a pas été altéré au cours de la transmission. Il emploie à cette fin des techniques de redondance appelées *total de contrôle* (ou *checksum*).

1.1 Notion de port

Les identifiants d'application sont indispensables, car plusieurs applications différentes peuvent s'exécuter simultanément sur la même machine. Un protocole de transport doit donc savoir pour qui il travaille ! L'identifiant d'application est appelé *numéro de port* – ou *port* – (ce qui n'a rien à voir avec les ports d'un commutateur). Selon les systèmes d'exploitation, le numéro de port peut correspondre au PID (*Process Identifier*), l'identifiant du processus qui s'exécute sur la machine.

Il existe des milliers de ports utilisables, puisque le numéro de port tient sur 16 bits. Une affectation officielle des ports a été définie par l'IANA (*Internet Assigned Numbers Authority*), afin d'aider à la configuration des réseaux. Parmi ceux-ci, les ports 0 à 1023 sont les ports bien connus ou réservés (*well known ports*), affectés aux processus système ou aux programmes exécutés par les serveurs. Les systèmes d'exploitation diffèrent dans leur affectation pour les identificateurs des autres processus. Ils choisissent *a priori* les grands numéros. Le tableau 7.1 cite quelques numéros de port bien connus.

Tableau 7.1

Quelques ports bien connus

Port	Service	Port	Service
20 et 21	FTP	989 et 990	FTPS
22	SSH		
23	Telnet		
25	SMTP		
53	DNS		
80	Web	443	HTTPS
110	POP3	995	POP3S
143	IMAP	993	IMAPS
161	SNMP		
179	BGP		
520	RIP		

1.2 Interface entre le protocole de transport et l'application

Les applications ayant le choix du protocole de transport (TCP ou UDP), le système d'exploitation doit utiliser un système de nommage qui affecte un identifiant unique, appelé *socket*[1], à tout processus local utilisant les services d'un protocole de transport. Le socket est constitué de la paire :

< adresse IP locale, numéro de port local >.

Pour identifier de manière unique l'échange de données avec le processus applicatif distant, le protocole de transport utilise un ensemble de cinq paramètres formé par le nom du protocole utilisé, le socket local et le socket distant :

< protocole, adresse IP locale, numéro de port local ; adresse IP distante, numéro de port distant >.

1.3 Total de contrôle ou checksum

Un total de contrôle est une information de redondance permettant de contrôler l'intégrité d'un bloc d'informations défini. Le calcul effectué est en général plus simple que celui décrit au chapitre 2 ; il s'agit, le plus souvent, d'une simple addition sans report des bits du bloc (un OU exclusif), suivie éventuellement d'une complémentation bit à bit du résultat précédent. Dans TCP ou dans UDP, le total de contrôle est un champ de 16 bits incorporé dans l'en-tête. Sa position dépend du protocole de transport utilisé.

2 Protocole TCP (*Transmission Control Protocol*)

TCP comble les carences d'IP lorsque les applications requièrent une grande fiabilité. Ce protocole de transport, lourd et complexe, met en œuvre la détection et la correction d'erreurs, gère le contrôle de flux et négocie les conditions du transfert des données entre les deux extrémités de la connexion.

L'entité gérée par le protocole TCP s'appelle le *segment*. Une fois le segment fabriqué, le module TCP sollicite le module IP pour le service de transmission, par l'intermédiaire de la primitive que nous avons vue au chapitre précédent : *Requête_émission (segment, adresse IP distante)*, dans laquelle les deux paramètres fournis sont le segment à émettre et l'adresse IP de destination. (Dans la pratique, plusieurs autres paramètres sont également fournis, mais nous nous intéressons ici au principe de fonctionnement.) À l'inverse, lorsque le module IP reçoit un datagramme destiné à la machine concernée et que celui-ci transporte un segment TCP, le module IP extrait le segment du datagramme et en signale l'arrivée au module TCP par la primitive : *Indication_réception (segment reçu, adresse IP source)*.

L'interface entre la couche TCP et la couche IP est très simple, celle entre la couche TCP et l'application utilisatrice est beaucoup plus complexe. Nous la détaillerons plus loin, après avoir vu le format du segment TCP et la vie d'une connexion.

1. Nous avons conservé ce terme anglais car aucun équivalent français n'est utilisé.

2.1 Dialogue de bout en bout

À la demande des applications qui ont besoin d'échanger des données de manière fiable, TCP ouvre une connexion et gère un dialogue. TCP n'est implanté que sur les machines des utilisateurs, c'est-à-dire qu'il n'est pas géré par les systèmes intermédiaires (ponts, commutateurs et autres routeurs du réseau). Il est défini dans la RFC 793 et s'occupe de gérer et de fiabiliser les échanges, alors qu'IP est responsable de la traversée des différents réseaux.

TCP permet à deux utilisateurs d'avoir une vision de bout en bout de leurs échanges, quels que soient l'interconnexion de réseaux sous-jacente et le chemin par lequel IP a fait passer les données. Pour apporter la fiabilité dont les utilisateurs ont besoin, TCP gère un *contexte* de l'échange (l'ensemble des paramètres choisis et négociés par les deux utilisateurs pour leur dialogue, ainsi que l'ensemble des paramètres temporels liés à l'état du dialogue). Le contexte est mémorisé dans le module TCP des deux interlocuteurs. Ce protocole fonctionnant en mode connecté, les deux modules TCP doivent être opérationnels simultanément. En outre, il faut que l'une des extrémités ait sollicité l'autre et que cette dernière ait répondu positivement. On parle de la *vie* de la connexion pour décrire tous les événements qui s'y produisent.

2.2 Fonctionnalités de TCP

TCP est capable de détecter les datagrammes perdus ou dupliqués et de les remettre dans l'ordre où ils ont été émis. Ce service repose sur la numérotation et l'acquittement des données et utilise une fenêtre d'anticipation. Remarquons dès à présent que la numérotation des données dans TCP s'effectue octet par octet, alors que les protocoles définis dans le modèle OSI numérotent les unités de données du niveau concerné.

TCP considère les données transportées comme un *flot* non structuré d'octets. Une connexion étant *a priori* bidirectionnelle, les deux flots de données sont traités indépendamment l'un de l'autre. Le flot géré par chaque module concerne les octets de données compris dans une zone délimitée nommée *fenêtre*, dont la taille est définie par un entier de 16 bits, ce qui la limite *a priori* à 65 535 octets. Chaque module TCP annonce la taille de son tampon de réception à l'ouverture de la connexion. Il est convenu que l'émetteur n'envoie pas plus de données que le récepteur ne peut en accepter. La taille de la fenêtre varie en fonction de la nature du réseau et surtout de la bande passante estimée à partir des mesures de performances qui sont effectuées régulièrement (voir section 2.4).

Grâce aux mesures effectuées, différents temporisateurs d'attente maximale d'acquittement de bout en bout sont dimensionnés de manière dynamique, à partir de la connaissance acquise sur le fonctionnement du réseau. En effet, le délai de traversée du réseau change d'une connexion à l'autre ; il peut même changer pendant la vie d'une connexion puisque IP ne garantit rien, pas même l'ordre dans lequel arrivent les données. Par ailleurs, TCP gère un flot de données urgentes, non soumises au contrôle de flux.

Remarque

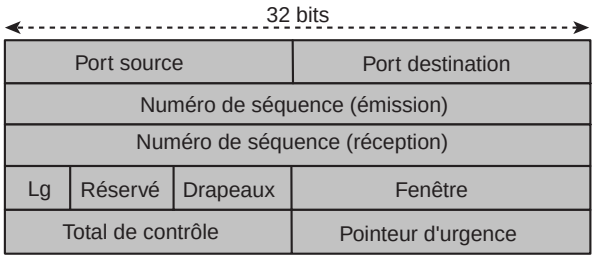
Par bien des aspects, le protocole de transport TCP ressemble au protocole de liaison de données du modèle OSI, puisque tous les deux gèrent le séquencement des données, le contrôle de flux, la détection et la récupération des erreurs. Il existe toutefois des différences importantes : tout d'abord, les entités de liaison communiquent directement sur le support physique, alors que les entités de transport communiquent à travers un ou plusieurs réseaux interconnectés. De ce fait, l'entité de transport doit spécifier l'adresse du destinataire. En outre, une entité de transport peut gérer un nombre important et variable de connexions, alors que l'entité de liaison n'en gère le plus souvent qu'une seule. Enfin, les données transmises dans l'interconnexion de réseaux peuvent « tourner en rond » ou disparaître pour réapparaître un peu plus tard. Elles peuvent donc être reçues dans un ordre qui n'était pas celui de leur émission, ce qui ne peut pas se produire sur un support physique unique.

2.3 Format du segment TCP

Il faut remarquer qu'il n'y a qu'un seul format de segment TCP, illustré à la figure 7.1, bien que le protocole soit complexe. Le segment contient un en-tête de 20 octets (sauf options) et un champ de données.

Figure 7.1

Format du segment TCP (en-tête sans option).



Signification des différents champs

- *Port Source* (16 bits). Numéro du port utilisé par l'application en cours sur la machine source.
- *Port Destination* (16 bits). Numéro du port relatif à l'application en cours sur la machine de destination.
- *Numéro d'ordre* (32 bits). La signification de ce numéro est à interpréter selon la valeur du drapeau SYN (*Synchronize*). Lorsque le bit SYN est à 0, le numéro d'ordre est celui du premier octet de données du segment en cours (par rapport à tous les octets du flot de données transportées). Lorsqu'il est à 1, le numéro d'ordre est le *numéro initial*, celui du premier octet du flux de données qui sera transmis (*Initial Sequence Number*). Celui-ci est tiré au sort, plutôt que de commencer systématiquement à 0[2].
- *Numéro d'accusé de réception* (32 bits). Numéro d'ordre du dernier octet reçu par le récepteur (par rapport à tous les octets du flot de données reçues).
- *Longueur en-tête* (4 bits). Il permet de repérer le début des données dans le segment. Ce décalage est essentiel, car il est possible que l'en-tête contienne un champ d'options

2. Le numéro de séquence initial est géré par une horloge interne, propre à la machine. Par exemple, si cette horloge est à 200 MHz, les numéros changent toutes les cinq microsecondes. Comme ce numéro est codé sur 32 bits, il y 4 milliards de numéros. Un même numéro ressort toutes les cinq heures et demie environ !

de taille variable. Un en-tête sans option contient 20 octets, donc le champ longueur contient la valeur 5, l'unité étant le mot de 32 bits (soit 4 octets).

- *Réservé* (6 bits). Champ inutilisé (comme dans tous les formats normalisés, il reste une petite place, prévue en cas d'évolutions à venir ou en cas de bogue à corriger, par exemple).
- *Drapeaux* ou *flags* (6 bits). Ces bits sont à considérer individuellement :
  - *URG (Urgent)*. Si ce drapeau est à 1, le segment transporte des données urgentes dont la place est indiquée par le champ Pointeur d'urgence (voir ci-après).
  - *ACK (Acknowledgement)*. Si ce drapeau est à 1, le segment transporte un accusé de réception.
  - *PSH (Push)*. Si ce drapeau est à 1, le module TCP récepteur ne doit pas attendre que son tampon de réception soit plein pour délivrer les données à l'application. Au contraire, il doit délivrer le segment immédiatement, quel que soit l'état de son tampon (méthode Push).
  - *RST (Reset)*. Si ce drapeau est à 1, la connexion est interrompue.
  - *SYN (Synchronize)*. Si ce drapeau est à 1, les numéros d'ordre sont synchronisés (il s'agit de l'ouverture de connexion).
  - *FIN (Final)*. Si ce drapeau est à 1, la connexion se termine normalement.
- *Fenêtre* (16 bits). Champ permettant de connaître le nombre d'octets que le récepteur est capable de recevoir sans accusé de réception.
- *Total de contrôle* ou *checksum* (16 bits). Le total de contrôle est réalisé en faisant la somme des champs de données et de l'en-tête. Il est calculé par le module TCP émetteur et permet au module TCP récepteur de vérifier l'intégrité du segment reçu.
- *Pointeur d'urgence* (16 bits). Indique le rang à partir duquel l'information est une donnée urgente.
- *Options* (taille variable). Options diverses, les plus fréquentes étant :
  - *MSS* (*Maximum Segment Size*). Elle sert à déterminer la taille maximale du segment que le module TCP accepte de recevoir. Au moment de l'établissement d'une connexion, le module émetteur annonce sa taille de MSS.

Exemple

Pour une application qui s'exécute sur un réseau Ethernet dans un environnement TCP/IP, le paramètre MSS pourra être 1 460 octets, soit 1 500 – taille maximale du champ de données d'une trame Ethernet – moins 40 octets, c'est-à-dire deux en-têtes de 20 octets (la taille normale de l'en-tête du datagramme IP sans option et de celle du segment TCP).

  - *Timestamp* (*estampille temporelle*). Sert à calculer la durée d'un aller et retour (RTT, *Round Trip Time*).
  - *Wscale* (*Window Scale* ou facteur d'échelle). Sert à augmenter la taille de la fenêtre au-delà des 16 bits du champ Fenêtre normal. Si la valeur proposée est *n*, alors la taille maximale de la fenêtre est de $65\,535*2^n$.
- *Remplissage.* Les options utilisent un nombre quelconque d'octets, or les segments TCP sont toujours alignés sur une taille multiple entier de 4 octets. Si l'en-tête sans option compte 5 mots de 32 bits, les options peuvent avoir une taille quelconque. Si besoin, on remplit l'espace qui suit les options avec des zéros pour aligner la taille du segment à une longueur multiple de 32 bits.
- *Données.* Ce champ transporte les données normales et éventuellement les données urgentes du segment.

2.4 CONNEXION TCP

TCP est un protocole qui fonctionne en mode client/serveur : l'un des utilisateurs est le serveur offrant des services, l'autre est le client qui utilise les services proposés par le serveur.

Ouverture de la connexion TCP

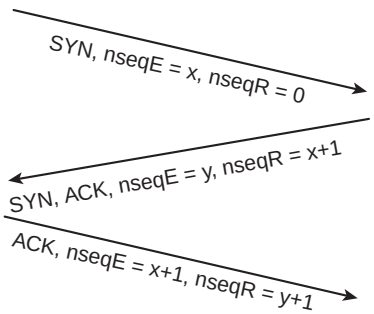
Le serveur doit être initialisé le premier ; on dit qu'il exécute une *ouverture passive*. Dès qu'il est opérationnel, il attend les demandes des clients, qui peuvent alors faire une *ouverture active*. Pour ouvrir une connexion, les clients doivent connaître le numéro de port de l'application distante. En général, les serveurs utilisent des numéros de ports bien connus, mais il est possible d'implanter une application sur un numéro de port quelconque. Il faut alors prévenir les clients pour qu'ils sachent le numéro de port à utiliser.

Le client ouvre la connexion en envoyant un premier segment, parfois appelé *séquence de synchronisation*. Ce segment contient en particulier le numéro de séquence initial (le numéro du premier octet émis) et le drapeau SYN est mis à 1. Le serveur répond par un acquittement comprenant le numéro du premier octet attendu (celui qu'il a reçu + 1) et son propre numéro de séquence initial (pour référencer les octets de données du serveur vers le client). Dans ce segment de réponse, le serveur a positionné les drapeaux SYN et ACK à 1. Enfin, le client acquitte la réponse du serveur en envoyant un numéro d'acquittement égal au numéro de séquence envoyé par le serveur + 1. Dans ce troisième message, seul le drapeau ACK est mis à 1.

L'ensemble des trois segments correspond à l'ouverture de la connexion, illustrée à la figure 7.2. Ce mécanisme d'ouverture est appelé *three-way-handshake* (établissement en trois phases). Après la dernière phase, le transfert des données peut commencer.

Figure 7.2

Ouverture d'une connexion TCP.



Fermeture de la connexion TCP

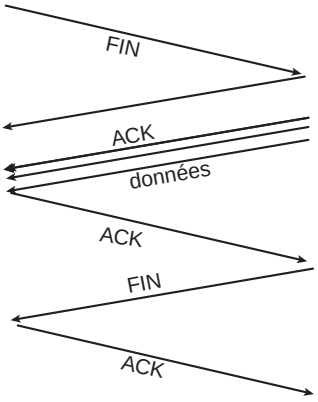
Une connexion TCP étant bidirectionnelle, les flots de données circulent dans les deux sens et chaque sens est géré par une extrémité. Les flots doivent donc être arrêtés indépendamment l'un de l'autre. De ce fait, si trois segments sont échangés pour établir une connexion, il en faut quatre pour qu'elle s'achève correctement, comme le montre la figure 7.3.

Pour indiquer qu'il a terminé l'envoi des données, un des modules TCP envoie un segment avec le drapeau FIN mis à 1. Ce segment doit être acquitté par le module distant. La connexion n'est vraiment fermée que lorsque les deux modules ont procédé à cet échange. La fermeture définitive n'intervient qu'après expiration d'un temporisateur.

Remarquons qu'une connexion peut être brutalement fermée à la suite d'un incident (par exemple lorsque l'utilisateur ferme inopinément son application). Le module TCP qui arrête brutalement la connexion émet un segment avec le drapeau RST mis à 1. Ce segment contient éventuellement les derniers octets en attente et aucun acquittement n'en est attendu. Le module TCP distant qui reçoit un segment avec le bit RST à 1 transmet les éventuelles dernières données à l'application et lui signale la fin anormale de la connexion.

Figure 7.3

Fermeture d'une connexion TCP.



Automate de gestion d'une connexion TCP

On voit que l'ouverture et la fermeture d'une connexion dépendent de nombreux événements et peuvent conduire à beaucoup de situations différentes. Une représentation commode des étapes de l'établissement et de la libération d'une connexion TCP se fait en utilisant un automate d'états finis à onze états (voir tableau 7.2). Dans chaque état, seuls certains événements sont autorisés et une action précise est entreprise. Dans le cas où l'événement n'est pas autorisé, on renvoie un signal d'erreur. La figure 7.4 donne les transitions d'états de l'automate d'une connexion TCP.

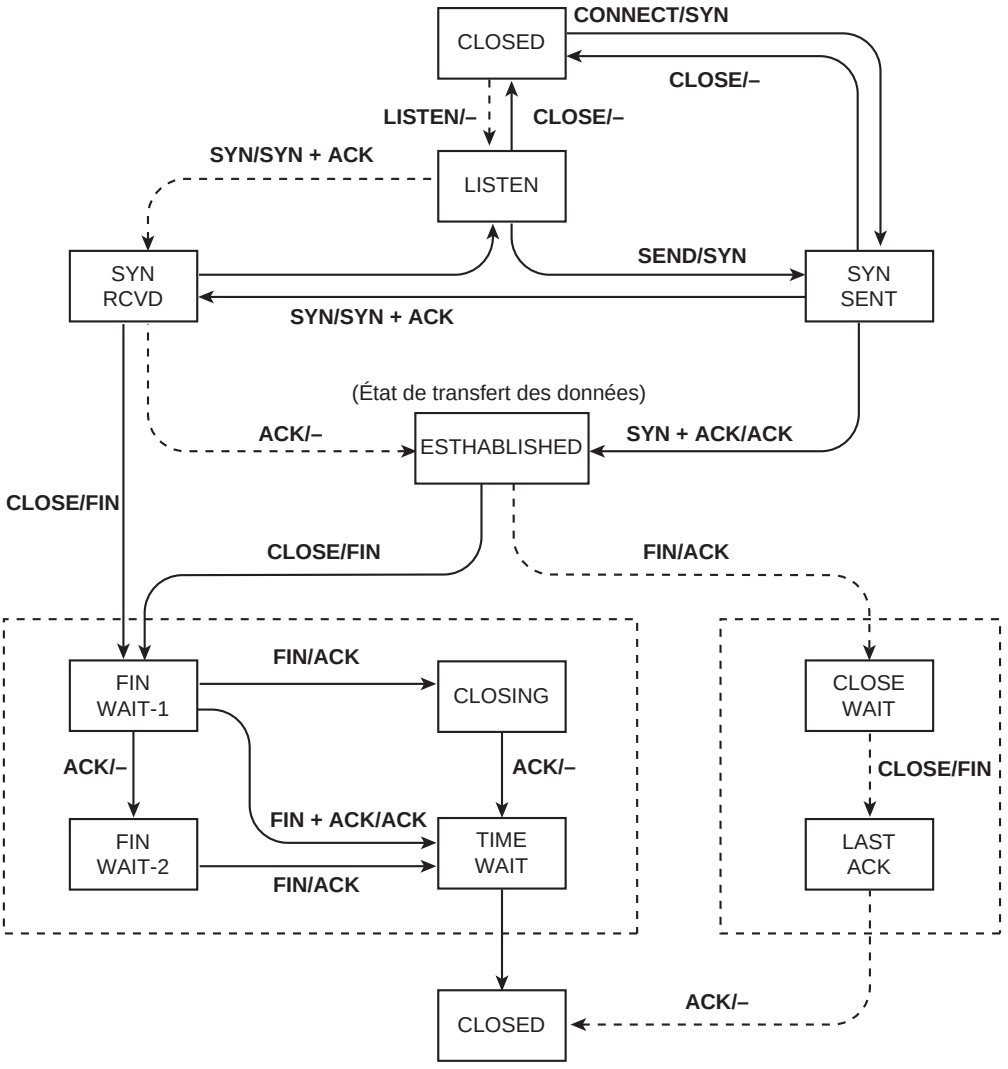
Tableau 7.2

Les états d'une connexion TCP

État	Description
CLOSED / FERMÉ	Aucune connexion active ou en attente
LISTEN / ÉCOUTE	Attente d'appel entrant par le serveur
SYN RCVD / SYN RECU	Attente d'ACK après l'arrivée de requête de connexion
SYN SENT / SYN ENVOYÉ	Ouverture de connexion déjà commencée par l'application
ESTABLISHED / ÉTABLIE	État normal de transfert de données
FIN WAIT1 / FIN ATTENTE 1	L'application a terminé
FIN WAIT2 / FIN ATTENTE 2	L'autre extrémité est d'accord pour libérer la connexion
TIME WAIT /TEMPORISATION	Attente de la fin d'émission des segments
CLOSING / FERMETURE	Tentative de fermeture de connexion
CLOSE WAIT / ATTENTE DE FERMETURE	L'autre extrémité a initialisé une fermeture de connexion
LASTACK / DERNIER ACK	Attente de la fin d'émission des segments

Figure 7.4

Diagramme de transitions d'états dans TCP.



Exemple

Pour obtenir des informations, en particulier sur les connexions TCP gérées par une machine, il faut taper la commande : `netstat  s  p tcp`.

Le tableau 7.3 montre les différents états des connexions TCP de la machine.

Tableau 7.3

Connexions TCP et leurs états

Protocole	Adresse locale	Adresse distante	État
TCP	ma_machine:1039	Pluton… : netbios	*ESTABLISHED*
TCP	ma_machine:1040	Uranus… : 993	*CLOSE-WAIT*
TCP	ma_machine:1026	207.46.19.30 : http	*ESTABLISHED*
TCP	ma_machine:1051	193.51.224.15 : http	*ESTABLISHED*
TCP	ma_machine:1055	Mercure… : https	*TIME-WAIT*

Transfert des données

Un des concepts les plus importants et les plus complexes de TCP est lié à sa façon de gérer les temporisations et les retransmissions. Comme d'autres protocoles fiables, TCP suppose

que le destinataire émet des accusés de réception chaque fois qu'il reçoit de nouvelles données valides. Il arme une temporisation chaque fois qu'il émet un segment, puis il attend de recevoir l'accusé de réception correspondant. Si le délai d'attente d'acquittement est atteint avant que les données du segment ne soient acquittées (on dit que la temporisation expire), TCP suppose que ce segment a été détruit ou perdu et le retransmet.

TCP est destiné à être utilisé dans un réseau quelconque, un petit réseau local ou dans Internet. Un segment qui transite d'un ordinateur à l'autre peut traverser un seul réseau à faible temps de transit ou traverser un ensemble de réseaux et de routeurs intermédiaires. Il est impossible, *a priori*, de connaître la rapidité avec laquelle les accusés de réception seront reçus par la source. De plus, le délai dépend du trafic ; il peut ainsi subir des variations considérables d'un moment à l'autre. Le temps d'aller et retour ou RTT (*Round Trip Time*) mesure le temps mis pour traverser le réseau. TCP doit donc prendre en compte à la fois les variations des délais de retransmission pour les différentes destinations et la grande dispersion des délais pour une destination donnée.

Exemple

L'utilitaire `ping` permet de savoir si une machine est opérationnelle et fournit une estimation du délai aller et retour. Dans l'exemple qui suit, la première machine est dans le même réseau local que celle de l'émetteur (adresse privée), la seconde est en Australie !

```
ping 192.168.0.2
délai approximatif des boucles en millisecondes
minimum = 3 ms, maximum = 11 ms, moyenne = 6 ms
ping 203.50.4.178
délai approximatif des boucles en millisecondes
minimum = 355 ms, maximum = 361 ms, moyenne = 358 ms
```

Algorithme de Jacobson Le module TCP surveille le comportement de chaque connexion et en déduit des valeurs de temporisation raisonnables. Il s'adapte aux variations de délais en modifiant les valeurs de temporisation. Le choix de la durée d'une temporisation est difficile, car la mesure du temps aller et retour dans le réseau est complexe. En outre, même si cette valeur est connue, il est difficile de fixer une temporisation : trop courte, elle provoque d'inutiles retransmissions ; trop longue, elle fait chuter les performances. L'*algorithme de Jacobson* est un algorithme dynamique qui ajuste la valeur de cette temporisation en fonction de mesures prises à intervalles réguliers dans le réseau.

On recueille des informations – comme la date à laquelle un segment TCP est émis et celle à laquelle l'accusé de réception correspondant lui parvient – pour calculer le RTT, qui est actualisé à chaque transmission. Le module TCP retient une estimation pondérée de RTT qui s'appuie à la fois sur le présent (*Nouveau_RTT*) et le passé (*RTT_estimé*). Le calcul de cette estimation tient compte d'un facteur de pondération a compris entre 0 et 1, donné par la formule :

$$RTT\_estimé = a*RTT\_estimé + (1 - a)*Nouveau\_RTT.$$

Le choix d'une valeur de a proche de 1 rend la valeur estimée de RTT insensible aux variations brèves, contrairement au choix d'une valeur de a proche de 0. Le plus souvent, on prend $a = 7/8$. TCP calcule finalement une valeur de temporisation à partir de la valeur *RTT_estimé* :

$$Temporisation = b*RTT\_estimé \text{ (avec } b > 1).$$

La valeur de temporisation doit être proche de la valeur *RTT_estimé* pour détecter les pertes de segment aussi vite que possible : si b vaut 1, tout retard provoque des retransmissions inutiles ; la valeur $b = 2$ n'est pas optimale lorsque la variance change. Jacobson proposa un algorithme qui rend b variable, par un calcul de probabilité donnant la valeur du délai de retour d'un accusé de réception : plus la variance croît et plus b est élevé et réciproquement. L'algorithme demande de conserver la trace d'une autre variable,

l'écart *D*. Chaque fois qu'un accusé de réception arrive, on calcule la valeur absolue de l'écart entre la valeur estimée et la valeur observée : |*RTT_estimé – Nouveau_RTT*| et *D* est alors donné par la formule :

$$D = (1 - a)^*|RTT\_estimé - Nouveau\_RTT|.$$

Dans cette formule, *a* peut avoir la même valeur que celle utilisée pour pondérer RTT car la valeur de *D* qui en résulte est suffisamment proche du résultat donné par le calcul de probabilité. On peut ainsi utiliser des opérations simples pour calculer *D*, afin d'obtenir rapidement le résultat du calcul. La plupart des implantations utilisent cet algorithme et prennent comme valeur de temporisation :

$$Temporisation = RTT\_estimé + 4^*D.$$

Algorithme de Karn Un autre problème a été identifié avec le mode d'acquittement de TCP : l'*ambiguïté des accusés de réception*. En effet, TCP utilise une technique d'acquittement cumulatif des octets reçus, dans laquelle l'accusé de réception concerne les données elles-mêmes et non pas le segment qui a servi à les acheminer. L'exemple donné ci-après illustre ce phénomène.

L'algorithme de Karn lève l'ambiguïté des accusés de réception. Lors du calcul de la valeur estimée du RTT, on ignore les mesures qui correspondent à des segments retransmis, mais on utilise une stratégie d'augmentation des temporisations (*timer back off strategy*). On conserve la valeur de temporisation obtenue, tant qu'une nouvelle mesure n'a pas été faite. Cela revient donc à calculer une valeur de temporisation initiale à l'aide de la formule précédente. À chaque expiration de la temporisation, TCP augmente la valeur de la temporisation (celle-ci est toutefois bornée pour éviter qu'elle devienne trop longue).

$$Nouvelle\ temporisation = g^*Temporisation\ (généralement\ avec\ g = 2).$$

Exemple d'ambiguïté des acquittements de *TCP*

TCP fabrique un segment, le donne à IP qui l'encapsule dans un datagramme puis l'envoie dans le réseau. Si la temporisation expire, TCP retransmet le même segment qui sera donc encapsulé dans un autre datagramme. Comme les deux segments transportent exactement les mêmes données, le module TCP émetteur n'a aucun moyen de savoir, à réception d'un acquittement, si celui-ci correspond au segment initial ou à celui qui a été retransmis : les accusés de TCP sont donc ambigus.
En effet, associer l'accusé de réception à la transmission initiale peut provoquer une forte augmentation du RTT en cas de perte de datagrammes IP : si un accusé de réception arrive après une ou plusieurs retransmissions et que TCP se contente de mesurer le RTT à partir du segment initial, il calcule un nouveau RTT à partir d'un échantillon particulièrement grand, donc il va augmenter sensiblement la temporisation du segment suivant. Si un accusé de réception arrive après une ou plusieurs retransmissions, le RTT suivant sera encore plus grand et ainsi de suite !

Autres temporisateurs TCP utilise trois autres temporisateurs : *de persistance*, *de limitation d'attente* et *de fermeture de connexion*. Le premier permet d'éviter la situation de blocage mutuel suivante : le récepteur envoie un accusé de réception avec une fenêtre nulle pour demander à l'émetteur de patienter. Un peu plus tard, le récepteur met la fenêtre à jour mais le segment est perdu : émetteur et récepteur s'attendent mutuellement. Quand le temporisateur de persistance expire, l'émetteur envoie un message « sonde » (*probe*) au récepteur. La réponse à ce message donne la taille de la fenêtre actuelle. Si elle est toujours nulle, l'émetteur réarme son temporisateur de persistance ; sinon, il peut envoyer des données.

Quand une connexion est inactive depuis un certain temps, le temporisateur de limitation d'attente expire et permet à une extrémité de vérifier si l'autre est toujours présente. On ferme la connexion s'il n'y a pas de réponse. Ce mécanisme n'est pas implanté partout car il augmente la charge du réseau et peut conduire à fermer une connexion active à cause d'une coupure passagère. D'un autre côté, il peut être intéressant de fermer une connexion inactive pour récupérer des ressources : pour toute connexion gérée, le module TCP en mémorise tous les paramètres et donc monopolise des ressources pour la gestion de cette connexion.

Le dernier temporisateur gère l'état *TIME WAIT* pendant la fermeture de connexion. Il vaut deux fois la durée de vie maximale d'un segment, ce qui assure que tous les segments d'une connexion ont disparu quand on la ferme.

Algorithmes de Clark et Nagle Un autre mode de fonctionnement, appelé *syndrome de la fenêtre stupide* (*silly window syndrom*), peut contribuer à faire s'effondrer les performances de TCP. Ce cas se rencontre lorsque l'application donne de grands blocs de données à l'entité TCP émettrice, alors que l'entité réceptrice traite les données octet par octet (une application interactive par exemple). Au départ, le tampon mémoire du récepteur est plein et l'émetteur le sait car il a reçu une indication de fenêtre de taille 0. L'application lit un caractère du flux ; le module TCP du récepteur envoie une indication d'actualisation de fenêtre pour qu'on lui envoie l'octet suivant. L'émetteur se croit obligé d'envoyer un octet et le tampon est à nouveau plein. Le récepteur acquitte le segment d'un octet mais positionne la fenêtre à 0 et ainsi de suite…

La solution de Clark consiste à empêcher le récepteur d'envoyer une indication d'actualisation de fenêtre pour un seul octet. On oblige le récepteur à attendre qu'il y ait suffisamment d'espace disponible avant de l'annoncer. En pratique, il n'enverra pas d'indication d'actualisation de fenêtre tant qu'il n'a pas atteint la taille maximale du segment. Le récepteur a obtenu cette valeur à l'établissement de la connexion, ou lorsque son tampon mémoire est à moitié vide (on prend généralement le minimum de ces deux valeurs). L'émetteur peut aider en n'envoyant pas de petits segments. Pour cela, il s'appuie sur les estimations qu'il établit à partir des indications d'actualisation de fenêtre déjà reçues. L'algorithme de Nagle propose d'envoyer le premier octet seul et d'accumuler les autres octets dans un tampon, tant que le premier octet n'est pas acquitté, puis d'envoyer dans un seul segment toutes les données accumulées et ainsi de suite. Cet algorithme est largement employé dans les implantations TCP mais dans certains cas, il est préférable de le désactiver. Les algorithmes de Clark et de Nagle sont complémentaires et utilisables simultanément.

L'objectif est que l'émetteur n'envoie pas de petits segments et que le récepteur n'en réclame pas en émettant des indications d'actualisation de fenêtre avec des valeurs trop faibles. En effet, le TCP récepteur, tout comme le TCP émetteur, peut accumuler les données. Pour cela, il bloque les primitives de lecture de l'application, tant qu'il n'a pas de données en nombre suffisant à fournir. Cette façon d'opérer diminue le nombre d'appels à TCP et diminue la surcharge globale (elle augmente légèrement le temps de réponse mais ce n'est pas très gênant pour les applications interactives).

Contrôle de congestion par TCP TCP manipule dynamiquement la taille de la fenêtre, pour ne pas injecter de nouveau segment tant qu'il en reste un ancien dans le réseau. On part de l'hypothèse que l'expiration d'un temporisateur sur Internet ne peut provenir que de la congestion d'une partie du réseau, la perte due à une erreur de transmission étant considérée comme un événement rare. TCP surveille donc en permanence les temporisateurs pour détecter toute congestion potentielle. Les capacités d'émission et de réception constituent deux sources possibles de problèmes et doivent être traitées séparément. Pour

cela, chaque émetteur gère deux fenêtres : celle qui est accordée par le récepteur et la *fenêtre de congestion*. Le nombre d'octets envoyés est égal au minimum des deux fenêtres ; celle qui est réellement utilisée correspond au minimum de ce qui convient à l'émetteur et au récepteur.

À l'initialisation de la connexion, l'émetteur prend une fenêtre de congestion correspondant à la taille maximale du segment utilisé et envoie un segment de taille maximale. Si le segment est acquitté avant expiration d'une temporisation, il augmente la taille de la fenêtre de congestion (la nouvelle fenêtre est de taille double par rapport à la précédente), puis l'émetteur envoie deux segments. À chaque accusé de réception de segment, il augmente la fenêtre de congestion en incrémentant d'une unité la taille maximale du segment.

Quand la fenêtre de congestion atteint *n* segments et si tous les segments ont été acquittés dans les temps impartis, l'émetteur augmente la taille de la fenêtre de congestion du nombre d'octets correspondant aux *n* segments. Ainsi, pour chaque émission couronnée de succès, la taille de la fenêtre de congestion va-t-elle doubler : elle croît exponentiellement, jusqu'à expiration de la temporisation ou si on atteint la taille maximale de la fenêtre de réception.

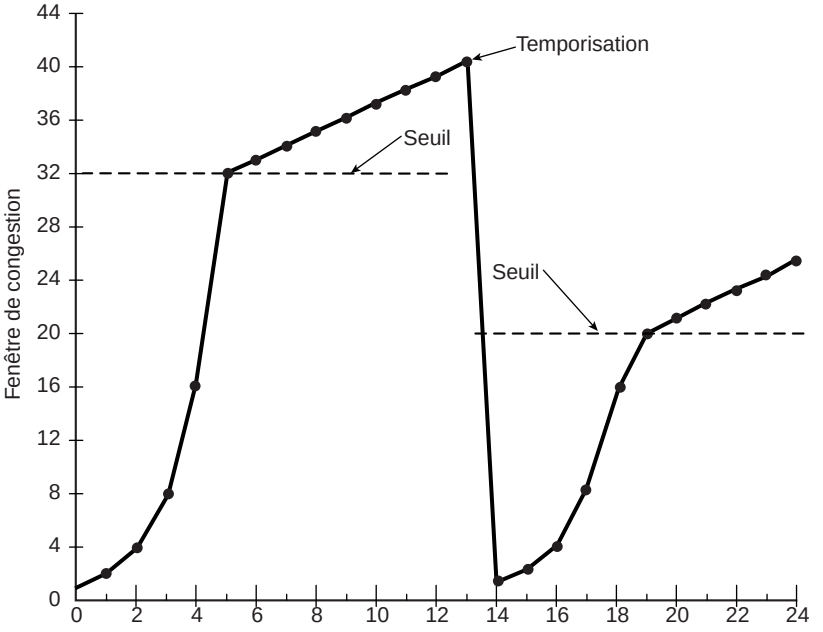
Exemple

Si on a envoyé des segments de 1 024, 2 048, 4 096 octets correctement mais qu'un segment de 8 192 octets provoque une expiration de temporisation, on évitera la congestion en dimensionnant la fenêtre à 4 096 octets et on n'enverra pas de segments de taille supérieure à cette valeur, quelle que soit la taille de la fenêtre de réception. Cet algorithme est appelé *algorithme de démarrage lent* (qui n'est en fait pas lent du tout puisqu'il a une croissance exponentielle !).

Un troisième paramètre, le seuil d'évitement de congestion (*threshold*), a pour valeur initiale 64 Ko. À expiration de la temporisation, le seuil est pris égal à la moitié de la fenêtre de congestion courante, et TCP réinitialise la fenêtre de congestion à la taille maximale de segment. Le démarrage lent est utilisé pour tester les possibilités d'absorption du récepteur ; on arrête la croissance exponentielle lorsque le seuil d'évitement de congestion est atteint. En cas de transmission avec succès, on augmente linéairement la taille de la fenêtre de congestion (c'est-à-dire qu'on augmente d'un segment la taille de la fenêtre au lieu d'un segment par segment acquitté), comme le montre la figure 7.5.

Figure 7.5

Évolution de la fenêtre de congestion.



2.5 Interface entre TCP et l'application

Les processus applicatifs communiquent en utilisant des sockets TCP. La programmation d'une application client/serveur se fait donc en manipulant les sockets.

Côté serveur, il faut d'abord créer le socket et le paramétrer (primitive *Bind*), en lui associant le numéro de port correspondant. Il faut ensuite le placer dans un état d'attente du client (primitive *Listen*).

Côté client, il faut créer le socket et établir la connexion (primitive *Connect*) qui sera acceptée par le serveur (primitive *Accept*). Le transfert des données pourra commencer, en utilisant des primitives *Read* et *Write (*ou *Sendto* et *Receivefrom*).

La fermeture est liée à l'utilisation de la primitive *Close* côté client (fermeture active) ou côté serveur (fermeture passive).

3 Protocole UDP (*User Datagram Protocol*)

UDP est un protocole de transport sans connexion qui permet l'émission de messages sans l'établissement préalable d'une connexion. C'est un protocole non fiable, beaucoup plus simple que TCP, car il n'ajoute aucune valeur ajoutée par rapport aux services offerts par IP. L'utilisateur n'est donc pas assuré de l'arrivée des données dans l'ordre où il les a émises, pas plus qu'il ne peut être sûr que certaines données ne seront ni perdues, ni dupliquées, puisqu'UDP ne dispose pas des mécanismes de contrôle pour vérifier tout cela. De ce fait, il n'introduit pas de délais supplémentaires dans la transmission des données entre l'émetteur et le récepteur. C'est la raison pour laquelle il est utilisé par les applications qui ne recherchent pas une grande fiabilité des données ou qui ne veulent pas assumer la lourdeur de gestion des mécanismes mis en jeu dans le mode connecté.

3.1 Service minimal

UDP est efficace pour le transfert des serveurs vers les clients avec des débits élevés, en délivrant les données sous forme de datagrammes de petite taille et sans accusé de réception. Ce type de service est utile pour les applications en temps réel, telles que les émissions en flux continu d'informations audio et vidéo. En effet, pour celles-ci, la perte d'une partie des données n'a pas grande importance. Les jeux en réseau, le *streaming* (procédé permettant de lire des fichiers audio ou vidéo avant même que celui-ci soit totalement téléchargé, grâce à une mise en mémoire tampon) utilisent aussi UDP. D'autres applications de type questions-réponses, comptant de petites quantités de données, peuvent également utiliser UDP. De ce fait, l'erreur ou la perte d'un datagramme sont gérées directement par l'application elle-même, le plus souvent à l'aide d'un mécanisme de temporisation. Au-dessus d'UDP, on trouve en particulier : le service d'annuaire DNS (*Domain Name System*), la transmission des informations de gestion de réseaux SNMP (*Simple Network Management Protocol*) ou d'informations de routage RIP (*Routing Information Protocol*).

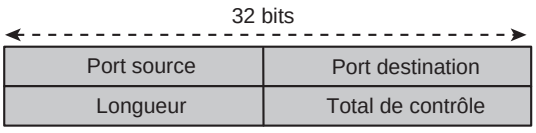
Notons que nous avons déjà vu (chapitre 6, exercice 15), un exemple utilisant le protocole UDP avec la commande *traceroute* sous Unix : celle-ci génère des datagrammes UDP, placés dans des datagrammes IP avec des durées de vie délibérément trop courtes.

3.2 Format du datagramme UDP

Les messages UDP sont généralement appelés *datagrammes UDP*. Ils contiennent deux parties, un en-tête et des données encapsulées dans les datagrammes IP, comme les segments TCP. Le format est illustré dans la figure 7.6.

Figure 7.6

Format du datagramme UDP.



L'en-tête très simple compte quatre champs :

- *Port source* (16 bits). Il s'agit du numéro de port correspondant à l'application émettrice du paquet. Ce champ représente une adresse de réponse pour le destinataire.
- *Port destination* (16 bits). Contient le port correspondant à l'application de la machine à laquelle on s'adresse. Les ports source et destination ont évidemment la même signification que pour TCP.
- *Longueur* (16 bits). Précise la longueur totale du datagramme UDP, exprimée en octets. La longueur maximale des données transportées dans le datagramme UDP est de : $2^{16} – 4*16$, soit 65 472 octets.
- *Total de contrôle* ou *checksum* (16 bits). Bloc de contrôle d'erreur destiné à contrôler l'intégrité de l'en-tête du datagramme UDP, comme dans TCP.

3.3 Interface entre UDP et l'application

Les processus applicatifs utilisent des sockets UDP. Leur manipulation est très simple puisque le protocole n'est pas en mode connecté : il n'y a pas de procédure de connexion et donc pas de fermeture non plus. Comme pour TCP, du côté du serveur, il faut d'abord créer le socket et le paramétrer par la primitive *Bind*, en lui associant le numéro de port correspondant. Puis il faut le placer dans un état d'attente des données du client (primitive *Listen*). Côté client, il faut créer le socket. Le transfert des données peut commencer directement en utilisant des primitives *Read* et *Write* (ou *Sendto* et *Receivefrom*).

Résumé

Deux protocoles de transport sont utilisés dans l'architecture TCP/IP. Le premier, TCP, est un protocole complet, destiné à pallier toutes les défaillances de l'interconnexion de réseaux. C'est un protocole en mode connecté qui met en œuvre une détection et une correction d'erreurs, un contrôle de séquence, de flux et de congestion. Il est de ce fait complexe et lourd à gérer. TCP est indispensable pour toutes les applications qui transfèrent de grandes quantités d'informations et qui ont besoin de fiabilité dans les échanges de données.

UDP, lui, utilise le protocole IP pour acheminer un message d'un ordinateur à un autre, sans aucune valeur ajoutée (pas de connexion, pas de contrôle d'erreur, de contrôle de flux ni de contrôle de séquence) par rapport aux services rendus par IP. Il convient aux applications de type requête/réponse simples ou ayant des contraintes temporelles fortes.