Chapitre 9

Modèle client-serveur

9.1. Introduction

Nous avons vu dans les chapitres précédents l'essentiel des principes des protocoles réseau. Cet ouvrage ne prétend pas aborder l'ensemble des problèmes : nous avons fait des choix, de manière à présenter l'essentiel. Nous souhaitons conclure l'ouvrage sur une touche pratique, en présentant comment on peut, depuis un programme, utiliser les services du réseau. Ce chapitre aborde donc la programmation d'une communication dans l'environnement de travail de stations UNIX.

Il existe un grand nombre de produits réseau ou piles de protocoles qui respectent en général les principes du modèle de référence ISO/OSI [43]. Nous citons sur la figure 9.1. les plus célèbres.

Cet ensemble de produits réseau est conforme au modèle de référence OSI, mais ces produits ne peuvent travailler conjointement sans passerelles du fait qu'ils utilisent des protocoles différents et ou des versions différentes des mêmes protocoles.

— SNA : System Network Architecture, couvre l'ensemble des produits réseau d'IBM.

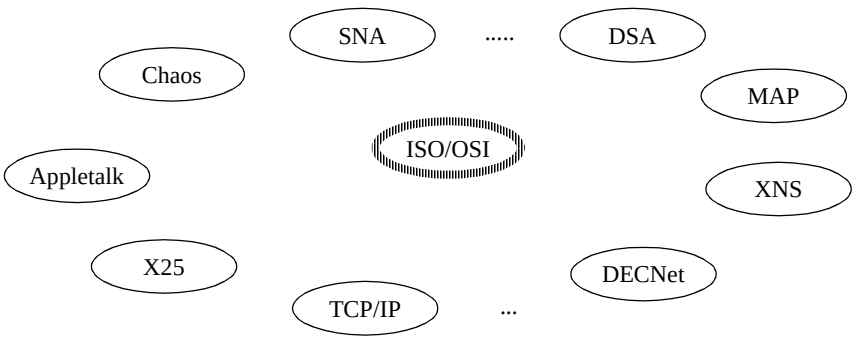

Figure 9.1. *Exemple de produits réseau ou piles de protocoles*

— DEC NET : architecture réseau de Digital Equipment, est en train de migrer d'une architecture (ensemble de protocoles) propriétaire aux protocoles OSI [15].

— TCP/IP : pile de protocole issue du réseau ARPA, normalisée par le DoD (Department of Defense américain). C'est le standard de fait pour interopérer. C'est la pile d'Internet [26].

— DSA : Distributed System Architecture de Bull.

— XNS : Xerox Network System, pile de protocole développée par Xerox et utilisée par de nombreux offreurs (3 + open entre autres dans LAN Manager).

— X25 : architecture des réseaux publics à commutation de paquets.

— Appletalk : réseau Apple.

— MAP : Manufacturing Automation Protocol, reprend pour l'essentiel les protocoles normalisés et y ajoute des services temps réels et des services pour la construction de systèmes industriels.

— IPC/VIP : de BANYAN System.

— IPX/SPX : de Novel Netware…

Il est important de comprendre qu'un système d'exploitation peut supporter plusieurs piles de protocoles et offrir à ses utilisateurs une interface idéalement commune pour chacune de ses piles par l'intermédiaires des primitives du système d'exploitation (cf. figure 9.2.). On appelle driver réseau ou activeur logiciel une pile de protocoles. En général, celles-ci sont vendues séparément du système d'exploitation. Par contre on ne peut pas faire parler :

— deux machines qui n'ont pas la même pile de protocoles,

— deux applications qui n'ont pas été interfacées (écrites) pour une pile de protocoles donnée.

L'interface de programmation réseau (API, *Application Programmation Interface*) n'est pas normalisée par l'ISO. La figure 9.2. laisse croire qu'une interface unique existe, mais ce n'est pas le cas actuellement. Des efforts sont en cours afin de définir l'interface application de manière unique pour un ensemble de constructeurs (groupe OSF, Open Software Foundation, par exemple). Nous décrivons dans ce chapitre l'interface socket d'UNIX. IBM pour sa part diffuse une interface appelée LU6.2 ou APPC, Application Pier to Pier Communication. Si la syntaxe diffère entre ces interfaces, les principes sont similaires.

9.2. Notion de client et de serveur

L'établissement d'une communication entre deux partenaires n'est pas symétrique. Lorsqu›on appelle quelqu'un au téléphone, il est nécessaire que celui-ci soit présent pendant la sonnerie et disposé à répondre à l'appel. Nous allons prendre l'exemple d'une transaction[1] chez un commerçant, car c'est un bon exemple de ce qui se fait dans les applications informatiques. Vous êtes le client et vous souhaitez acheter des timbres à la Poste. Pour que cette transaction ait lieu, il faut que soient

1. Une transaction est typiquement une communication applicative brève.

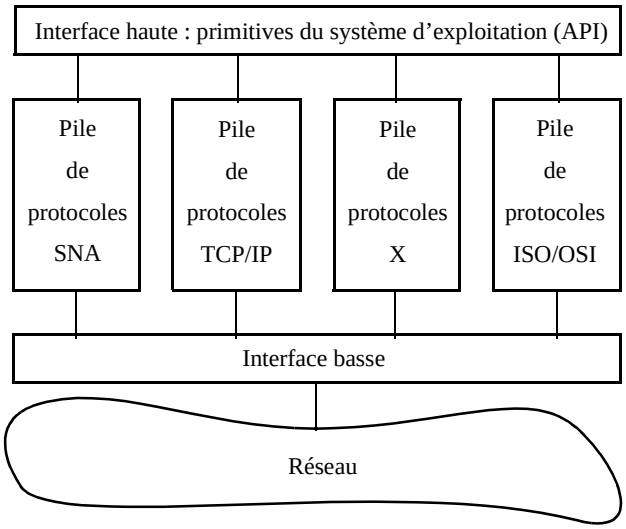

Figure 9.2. *Piles de protocoles dans une machine*

remplies un ensemble de conditions que nous allons énumérer. Cette transaction met en jeu deux partenaires : le client (vous-même) et un guichetier (le serveur). Le client doit connaître l'adresse de la Poste et probablement le bon guichet. La Poste doit être ouverte et le guichetier présent. Il est important de noter que le guichetier attend les clients. Il doit être présent avant eux pour que le service puisse être rendu. Il a d'ailleurs un certain nombre d'actions à faire préalablement à l'ouverture du guichet (ouverture de la caisse, recherche des timbres dans le coffre-fort, ouverture du bureau de Poste…). Cela correspond à une phase d'initialisation.

Lorsque le guichet est ouvert, les premiers clients peuvent se présenter. Le guichetier sert un client à la fois. La communication s'établit entre le premier client et le guichetier, à l'initiative du client. Le client soumet ses demandes dans une langue compréhensible par le guichetier. Le guichetier effectue le service demandé. Un dialogue peut être nécessaire à l'exécution correcte du service demandé. C'est-à-dire que l'application doit définir son propre protocole et que le serveur ne sait traiter qu'un ensemble prédéfini de fonctions exprimées selon une syntaxe rigoureuse.

Lorsque le service est terminé, le client (ou le guichetier) libère la communication. Le guichetier est alors libre de servir le client suivant.

Le serveur (guichetier) doit être présent, en attente d'un client. Un programme chargé d'offrir des services (FTP, mail…) comprend les étapes suivantes :

— étape a : initialisation, déclaration du service, mise en place du contexte…,

— étape b : attente du client,

— étape c : service d'un client.

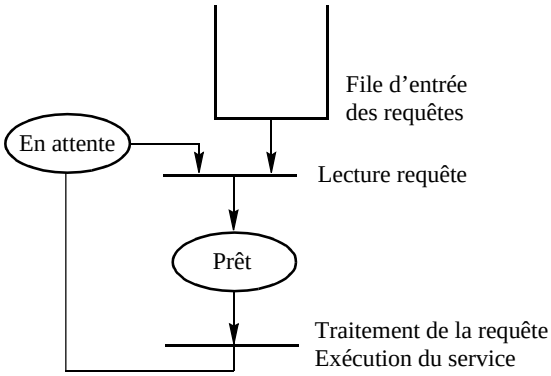

Figure 9.3. *Serveur itératif*

Un programme serveur boucle à l'infini sur les étapes b et c. Le comportement d'un client n'a pas de profil typique. La figure 9.3. montre, sous forme de réseau de Petri, le comportement d'un serveur de ce type. On dit que le serveur est itératif, car il traite une requête à la fois. Cela convient bien si le service est court (service de l'heure). Par contre, si le service est long (transfert de fichiers), cette approche est inacceptable. Un serveur itératif sert une requête à la fois, et les requêtes suivantes attendent dans la file d'attente.

Un serveur parallèle permet de traiter plusieurs requêtes simultanément. Une Poste avec plusieurs guichets identiques ouverts est typiquement un serveur parallèle. Sur la figure 9.4, nous schématisons ce service. Tous les clients arrivent dans une même file unique (ce n'est pas le cas en France). Un serveur libre prend le premier de la file. Ce type de service est plus équitable pour l'ensemble des clients, car chaque client

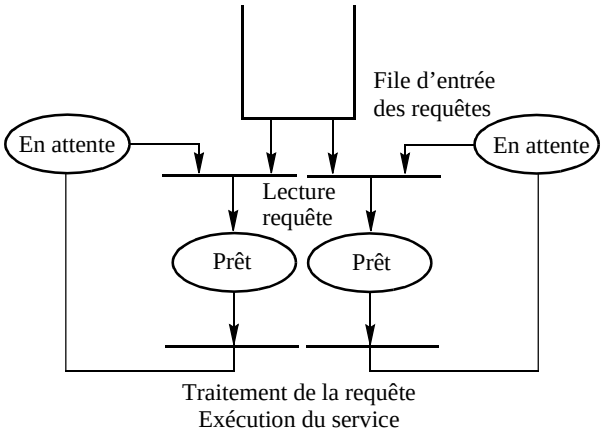

Figure 9.4. *Serveur parallèle*

attendra dans la file en fonction du temps de service de tous les clients précédents et non en fonction des clients d'un seul serveur.

En informatique, on sait créer dynamiquement des serveurs. Lorsqu'un client arrive, un serveur d'accueil va créer pour lui un nouveau serveur qui ne s'occupera que de lui et qui disparaîtra à la fin du service. L'intérêt de cette approche est que le serveur d'accueil est itératif sur un service court : il a pour seule fonction de prendre le client, de créer un processus pour le servir, puis de revenir attendre le client suivant. Il y a à tout instant autant de serveurs en parallèle que de clients plus un (le serveur d'accueil). Il est inutile de prévoir le nombre maximal de clients en parallèle. Nous reviendrons sur cet exemple à la fin de ce chapitre, en montrant comment est réalisé le service FTP vu dans le paragraphe 1.5.1. du premier chapitre.

9.3. UNIX et les réseaux

Le système d'exploitation UNIX, développé dans les années 1970 par ATT, a d'emblée été conçu pour communiquer. Aux mécanismes de base, que sont les *pipes* pour la communication entre tâches et le courrier électronique pour la communication entre humains, ont rapidement été ajoutés des moyens de communication en réseau. Dans un premier temps, il s'agissait d'utiliser les liaisons téléphoniques à travers des modems. Puis, au fur et à mesure que sont apparus les concepts réseau, développés dans cet ouvrage, les solutions ont été intégrées dans UNIX. Ce travail fut réalisé en parallèle par ATT pour aboutir à UNIX system V et par l'Université de Berkeley avec UNIX BSD. Nous nous attachons à décrire dans ce chapitre essentiellement les services offerts dans UNIX (principalement dans la version BSD 4.3) et les interfaces utilisateurs.

Les principaux protocoles évoqués précédemment ont été implantés sous UNIX. Cependant ils ne sont pas tous systématiquement présents, il faut acheter l'implémentation dans la plupart des cas. Il en va tout autrement de TCP/IP inclus gratuitement aux système UNIX modernes. Les protocoles SLIP et PPP sont accessibles librement et gratuitement sur de nombreux sites INTERNET. Certains vendeurs les intègrent à leur livraison de système. Il y a aussi toute une batterie de protocoles adaptés à des voies physiques telles que : satellites, lignes téléphoniques, radio…

	Sur connexion	Hors connexion
Serveur itératif	Rare (ex : commande date)	Usuel
Serveur parallèle	Usuel	Rare (ex : TFTP)

Figure 9.5. *Modes d'utilisation des connexions*

9.3.1. *TCP/IP dans UNIX*

Les protocoles de transport utilisés sous UNIX sont TCP (T*ransport Control Protocol*) et UDP (*User Datagram Protocol*), et le protocole IP (*Internetwork Protocol*) pour la couche réseau. Ils ont été intégrés au sein d'UNIX dès le début des années 1980. Nous avons étudié ces protocoles au chapitre 8 et la figure 8.2. montre les entités protocolaires qui sont installées dans le système UNIX.

Le nommage des canaux de communication entre processus en réseau est plus complexe qu'au niveau d'une machine UNIX, et l'on ne peut pas se contenter des adresses réseau vues au paragraphe 6.2.1. A l'intérieur du système d'exploitation comme UNIX, un programme peut facilement communiquer avec un autre à l'aide de mécanismes systèmes internes (*pipe*, mémoire partagée[1],...) ou à l'aide de fichiers. Dans une application réseau, il n'existe pas un seul système de fichiers mais plusieurs, totalement indépendants. De plus les mécanismes internes aux systèmes ne peuvent plus servir car il y a plusieurs systèmes autonomes en jeu. Les concepteurs d'UNIX ont souhaité homogénéiser les accès aux différents objets, qu'ils soient locaux ou en réseau, à travers le concept de *socket*. Ce concept permet de désigner un canal de communication à travers un service réseau entre deux processus. Il doit donc décrire le service réseau utilisé (pile de protocoles TCP/IP, X25, mais aussi système de fichier...), les adresses (réseau IP, X25 aux deux extrémités [locale et distante], ou fichier) et les processus aux deux extrémités. C'est l'association de ces 5 éléments qui est appelée socket. Nous ne décrirons ici que les paramètres réseau pour TCP/IP :

<protocole,
adresse[2] locale,
"porte source", process local,
adresse[1] distante,
"porte destination", process distant>

Cette association permet de désigner de manière unique la voie de communication utilisée par l'application. Selon le type de service réseau utilisé :

— la taille des adresses varie :
  - Internet TCP/IP, 32 bits,
  - XNS, 10 octets,
  - ISO, taille 20 octets,
  - X25, taille variable ;

— dans certains protocoles la notion de paquet n'existe pas (cas de TCP/IP) ;

— il faut accepter des protocoles différents : TCP/IP, X25, XNS, ISO TP4...

Les notions de « porte source » et « porte[3] destination » associées aux adresses IP source et destination permettent de définir de manière unique la communication par

1. Dans deux machines distinctes, les espaces mémoire sont différents. Aussi, la notion d'adresse mémoire ou de pointeur n'a plus de sens entre deux applications en réseau.

2. Selon le protocole utilisé, l'adresse sera de nature différente : par exemple une adresse X25 demande 14 demi-octets et est de taille variable.

TCP ou UDP entre un couple de processus. Elle permet donc d'identifier un SAP de niveau transport. Nous avons décrit au début de ce chapitre la notion de client/serveur et nous avons montré que, si le serveur se met en attente d'une requête quelconque, le client, lui, doit connaître le serveur appelé. Pour cela, il faut que le client connaisse à l'avance la porte du serveur auquel il s'adresse. En effet, toute machine multitâche peut héberger plusieurs serveurs. Plusieurs procédés sont utilisés au sein d'UNIX pour résoudre ce problème. La première solution consiste à réserver a priori les portes 1 à 1 023 pour des services généraux[1]. Ceux-ci sont décrits dans le fichier /etc/services du système. Ainsi, le service FTP, *File Transfert Protocol*, possède la porte 21. Un client qui souhaite utiliser le service FTP d'un site distant doit seulement connaître l'adresse Internet de ce site. Il sait par convention que le serveur FTP écoute sur le port 21 et que celui-ci utilise le protocole TCP.

9.3.2. *Les sockets*

La philosophie d'UNIX est de rendre la communication, une fois établie entre deux processus, identique aux opérations de lecture et écriture sur un fichier. La figure 9.8. rappelle les outils de communication du système UNIX. La différence entre les opérations du client et du serveur réside dans la phase d'ouverture. Dans le premier paragraphe, nous soulignons la dissymétrie du modèle client-serveur. Celle-ci va apparaître dans les opérations que vont effectuer les deux partenaires (cf. figure 9.6.). Le rôle des partenaires est inclus dans le code des deux programmes.

La communication peut être réalisée à l'aide d'un service sur connexion ou hors connexion. Dans un service sur connexion, le lien étant établi avec un partenaire unique, les échanges pourront être exactement similaires aux accès à un fichier. Dans un service hors connexion, chaque message reçu par le serveur peut provenir d'un client différent. Dans le service sur connexion, il faut ouvrir la connexion (cf. figure 9.6.), alors que cette opération n'est pas réalisée dans le service hors connexion (cf. figure 9.7.).

Le nommage et la désignation des objets sont plus délicats en réseau qu'au sein d'un système, où il suffit de passer à une procédure ou à un processus fils un descripteur de fichier pour que ce dernier puisse l'utiliser sans risque. Il ne peut en être de même en réseau. Le processus qui hérite du descripteur d'un lien de communication a besoin de vérifier que son correspondant est bien celui qu'il attend. Par exemple, il

---

3. Dans le système UNIX on parle de « port » et non de porte. Le concept est le même. Nous utilisons le terme porte car son acception commune correspond exactement à la définition d'un point d'accès à l'intérieur d'un immeuble où sont abrités plusieurs services (ou clients) ayant la même adresse. Par exemple le Bureau des missions, porte 34, à Télécom Bretagne, rue de la Châtaigneraie, BP 78, 35512 Cesson.

1. Les portes (port) 0 à 1 023 sont réservées aux services préassignés qui doivent être lancés pour des raisons de sécurité par l'administrateur et sont normalisé par l'IANA (IETF Internet Advisory Network Association). Vous en trouverez une description dans le fichier /etc/services. Les portes supérieures à 6 000 sont aussi utilisées pour des services publics (par exemple Xwindow).

souhaitera connaître le nom du processus correspondant pour des raisons de sécurité ou de choix de protocole.

La définition de la voie de communication contient donc le quintuplé suivant :

<protocole, adresse locale, process local, adresse distante, process distant>

Outre qu'il nous faut plus de paramètres pour établir un lien « réseau » qu'avec un fichier, les paramètres d'adresses vont avoir des structures différentes selon le protocole utilisé. Une adresse Internet tient sur 4 octets ; avec XNS il faut 10 octets, avec X25 il en faut 7 et avec l'ISO la taille est 20 octets.

L'accès à un fichier dans UNIX est conçu comme un flux d'octets. Seul le protocole TCP respecte ce principe. UDP, XNS, ISO sont orientés vers le transfert de blocs de données.

Dans la suite de cet exposé, nous allons décrire l'interface réseaux UNIX à l'aide des primitives du langage C dans lequel ce système est écrit. Chaque langage et chaque système d'exploitation ayant une philosophie propre, les services de communication mis à la disposition des utilisateurs seront fortement orientés par l'esprit du système d'exploitation et du langage utilisés. Les normes ne traitent pas de l'interface de programmation, ou API (*Application Program Interface*). Cette lacune est en train d'être comblée par les groupes de constructeurs tels que l'OSF (Open Software Foundation) afin d'améliorer la portabilité des applications. Actuellement chaque système, chaque langage fournit son propre système d'interfaces. Ainsi, C et ADA n'ont pas la même vision du service réseau.

Il existe deux interfaces UNIX pour programmer la communication en réseau. La première et la plus ancienne que nous utiliserons ici est connue sous le nom de socket BSD. Elle est simple à utiliser et reste la plus fréquemment rencontrée dans les programmes développés pour Internet (en fait la pile TCP-UDP/IP). La seconde, plus récente, appelée TLI, *Transport Level Interface*, a été introduite à partir de 1986. Elle permet un accès aussi bien aux services de transport Internet que OSI, XNS... Elle est donc plus universelle. Elle est maintenant systématiquement employée pour la programmation des systèmes d'exploitation. Son jeu de primitive inclut celui des socket BSD mais est plus riche à la fois en nombre de primitives et en fonctionnalités. Dans ce chapitre, nous présentons les interface socket BSD d'UNIX pour le langage C.

9.3.2.1. *Primitive socket()*

Les deux partenaires débutent toute activité réseau par l'invocation de la primitive socket (). Celle-ci a trois arguments :

— famille du protocole et/ou niveau d'accès : Internet (AF_INET ou PF_INET) avec pour TCP/IP ou UDP/IP, mais ce peut aussi être X25 (AF_X25), XNS, liaison…,

— type dans la famille : datagrammes ou connexion (flux de donnée ou stream), soit SOCK_STREAM pour TCP et SOCK_DGRM pour UDP, des types existent pour les autres familles,

— la version à utiliser. La valeur zéro indique le protocole standard mais une valeur différente permet d'invoquer par exemple une version à tester.

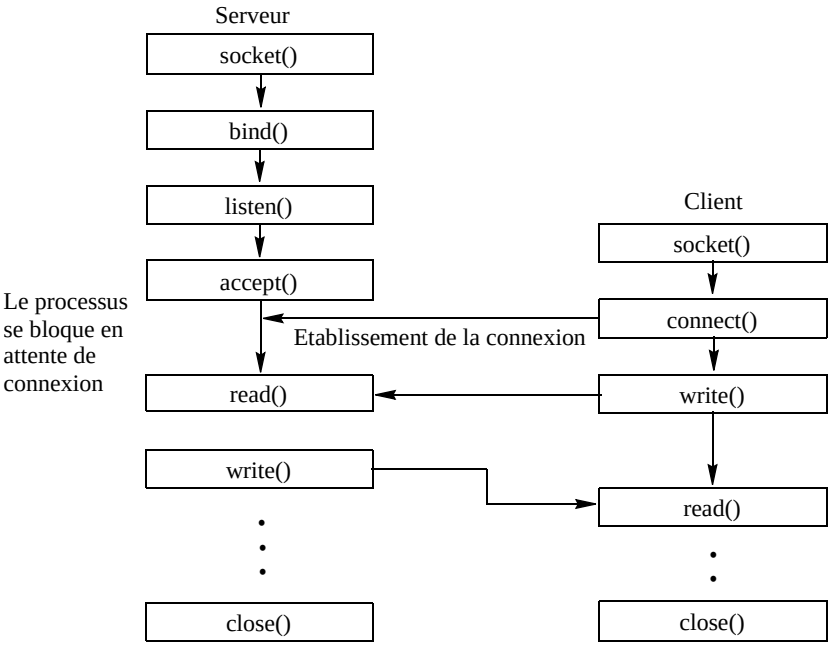

Figure 9.6. *Etablissement d'une connexion entre deux processus. Séquence de primitives à invoquer*

La procédure *socket*() rend une valeur entière qui permet de pointer sur une structure interne au noyau UNIX, de la même manière que, lors d'une ouverture de fichier, l'opération d'ouverture donne un pointeur sur le descripteur de fichier. Comme pour les ouvertures de fichier, la primitive *socket*() fournit un entier, sd, qui désigne le descripteur de socket qui n'est autre que le quintuplé précédemment décrit, mais où seuls sont initialisés le premier champ (type du protocole), l'adresse locale et le port local (cf. figures 9.6. et 9.8.). Ce descripteur est rangé dans les tables du système. Il est logique que chaque partenaire doive créer son propre descripteur puisqu'il ne saurait y avoir de mémoire commune au réseau.

Achever de remplir le descripteur dépend du rôle joué par le programme concerné et du protocole utilisé. Le serveur ignore bien sûr le nom et l'adresse du processus qui va l'appeler. Par contre, le client connaît l'ensemble de ces informations. Cela conduit donc à des primitives différentes. D'autre part, un échange hors connexion n'a pas besoin de maintenir autant d'informations qu'un service sur connexion.

9.3.2.2. *Primitive bind()*

Dans notre descripteur de socket, sd, le champ porte source est initialement mis à une valeur choisie par le système (>1024). La primitive *bind*() permet au processus propriétaire de spécifier son numéro de porte. Si *bind*() n'est pas invoqué, UNIX

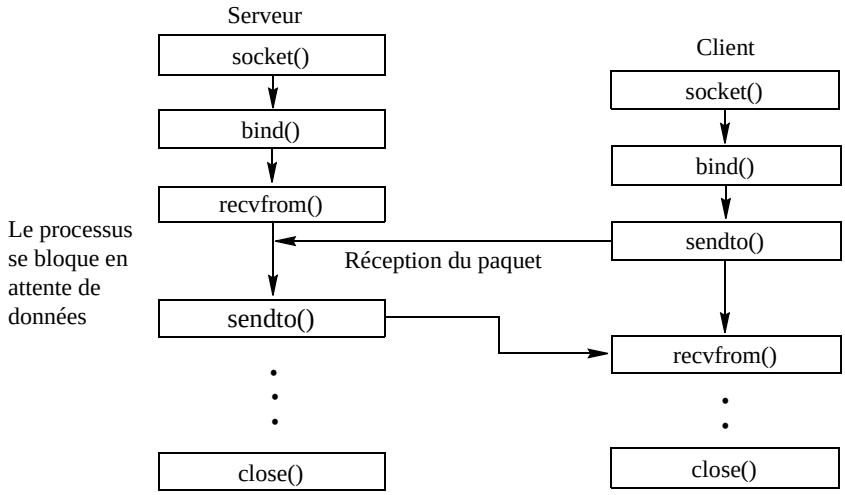

Figure 9.7. *Etablissement d'un échange hors connexion entre deux processus. Séquence de primitives à invoquer*

	Socket	Messages	FIFO	Pipe
Serveur création	socket() bind()	msget()	mknod	pipe()
Client création	socket() bind()	msgget()	open()	
Attente de connexion (serveur)	listen() accept()	semctl	shmctl	read()
Communication sur connexion	read() write()	msgrcv() msgsnd()	read() write()	read() write()
Datagramme	recvfrom() sendto()			
Terminaison	close()	msgcntl()	close()	close()

Figure 9.8. *Comparaison de trois méthodes de communication*

établit automatiquement le lien entre la porte attribuée au processus et le nom du processus sous UNIX. De ce fait, le programme client n'a pas besoin d'invoquer cette

primitive. Un mécanisme du système est chargé de l'attribution des portes de manière unique aux différents processus.

L'usage de la primitive *bind* est essentiel pour un serveur qui est connu à l'extérieur par son numéro de porte. Cette procédure a trois paramètres :

— sd, désigne le descripteur de socket obtenu par *socket*(),

— un pointeur sur une structure appelée « *sockaddr* » locale, qui contient :

• l'adresse du site,

• la porte (numéro de port) du processus.

— la longueur de la structure *sockaddr*.

Nous avons vu que les adresses avaient des longueurs différentes, il faut donc la fournir. La structure *sockaddr*, aussi appelée demi-socket, est utilisée systématiquement. Dans les versions récentes d'UNIX (BSD 4.4), la longueur du champ adresse est intégrée dans la structure *sockaddr*. C'est pourquoi nous l'ignorerons dans la suite de cet exposé.

Le paramètre important que l'on va retrouver dans toutes les primitives suivantes est la structure *sockaddr* locale ou distante. La forme exacte de cette structure dépend du protocole utilisé. Le lecteur devra se reporter au fichier <sys/socket.h> de son système pour trouver les définitions exactes. La définition précise nous entraînerait dans des développements longs et nécessite une connaissance approfondie du langage C et d'UNIX.

Bind va être utilisé dans deux situations :

1 - Le serveur enregistre dans le descripteur son adresse, qui est par ailleurs connue de tous. Ainsi, le système pourra lui envoyer les messages qui portent son numéro de porte (port ou SAP de niveau transport). Un serveur doit effectuer ce lien quel que soit le type du protocole sur connexion (figure 9.6.) ou hors connexion (figure 9.7.) avant de pouvoir recevoir un message quelconque. Cet enregistrement est nécessaire : en effet le processus serveur est connu d'UNIX sous un numéro de processus arbitraire (dépendant du moment de lancement de la tâche). Cette primitive permet de dire sur quel SAP ce processus est connu du monde externe. Les portes des services publics sont décrites dans le fichier «/etc/services».

2 - Un client peut s'enregistrer aussi s'il a besoin de recevoir une réponse du serveur sur une porte particulière (en général ce n'est pas le cas). Et donc en général il n'y a pas besoin d'invoquer *bind()* dans un client. Néanmoins, dans certains services tels que les RPC, *Remote Procedure Call*, le client obtient un numéro de la porte d'un processus particulier (*portmapper*) : il a donc besoin d'affecter son numéro de porte.

9.3.2.3. *Primitive connect()*

La primitive *connect*() établit un service sur connexion et a comme la primitive *bind*() les paramètres :

— sd, désigne le descripteur de socket ;

— un pointeur sur la structure « *sockaddr* » contenant :

• l'adresse du site distant,

• la porte (numéro de port) du processus distant.

Ainsi, les deux paramètres qualifiés de distants dans le descripteur socket peuvent être remplis. La primitive *connect()* ne nécessite pas que soit fourni le nom du processus local qui est créé dynamiquement par le système. Quant à l'adresse locale, elle est bien sûr connue du système. *Connect*() effectue automatiquement le travail de la fonction *bind()*. La primitive *connect()* est bien sûr nécessaire pour établir une communication dans un protocole de type sur connexion. Elle est en général invoquée par le client.

Dans un service en mode connecté, cette primitive rend un résultat après l'établissement effectif de la connexion ou l'échec. Elle provoque donc l'exécution du protocole définit sur la figure 8.15.

Elle peut aussi être utilisée pour un service hors connexion afin d'enregistrer le nom du serveur dont on souhaite recevoir les messages. La réponse à la primitive dans ce cas est immédiate et n'informe pas de l'existence du correspondant.

9.3.2.4. *La primitive listen()*

La primitive *listen*() permet à un serveur travaillant en mode sur connexion d'indiquer au noyau le nombre de connexions qu'il peut accepter simultanément pour un descripteur de socket. *Listen*() crée une file d'attente de taille limitée dans le système pour notre processus.

9.3.2.5. *La primitive accept()*

La primitive *accept()* prend la première requête d'ouverture de connexion qui vient du réseau, crée un nouveau descripteur de *socket()*, remplit les informations porte source et adresse source de la requête d'ouverture. Si aucune requête n'est arrivée, le processus appelant est mis en attente de l'arrivée d'une requête. La primitive *accept*() a les paramètres habituels :

— sd, désigne le descripteur du socket ;

— un pointeur sur une structure *sockaddr* comprenant :

  • l'adresse du site distant,

  • la porte (le numéro de port) du process distant.

Ces valeurs ne sont pas remplies à l'appel de la primitive, mais à l'arrivée du message de demande d'ouverture. Elle correspond aux étapes « *passive open* » jusqu'à l'ouverture réussie de la figure 8.15. Au retour la structure *sockaddr* contient l'adresse du site et le nom du processus appelant. La primitive crée un nouveau descripteur de *socket* identique à celui référencé par sd (contenant les mêmes informations) de manière à ce que le serveur puisse, en créant un fils, traiter les requêtes venant sur la connexion en utilisant le descripteur créé et à ce que le processus père puisse venir se mettre en attente d'une nouvelle demande d'ouverture (serveur concurrent) avec le descripteur initial.

Dans le mode de service sur connexion, les primitives *accept*() et *connect*() se correspondent pour compléter le contenu des deux descripteurs de sockets.

	Serveur sur connexion	Client	Serveur hors connexion	Client
protocole	socket()			
adresse locale	bind()	connect()	bind()	bind()
process local				
adresse distante	accept()	connect()	recvfrom()	sendto()
process distant				

Figure 9.9. *Rôle des primitives et instant où les champs de la socket sont remplis*

9.3.2.6. *Primitive de transfert sur connexion*

Dans un service sur connexion, les partenaires de la connexion utilisent les primitives de lecture *read*() et écriture *write*(), standard du langage C, pour travailler sur les fichiers, pour recevoir ou émettre.

9.3.2.7. *Primitives sentdo et recvfrom*

Dans un service hors connexion, la notion de flux associée au fichiers sous UNIX ne peut être maintenue. Plusieurs primitives permettent d'émettre et de recevoir des messages (cf. figure 9.9.). Ces primitives peuvent aussi être utilisées avec les connexions ; elles permettent entre autres d'accéder aux services de données hors bande ou express.

L'émission est effectuée par *sendto()*. La réception est effectuée par *recvfrom()*. Elles possèdent chacune les paramètres suivants :

— sd désigne le descripteur de socket ;

— un pointeur sur la chaîne d'octet à transmettre (resp. à émettre) ;

— la longueur de la chaîne à transmettre (resp. à recevoir) ;

— un ensemble de drapeaux qui peuvent être :

• envoi ou réception de données express aussi appelé hors-bande,

• prendre une copie des données reçues sans que celles-ci soient retirées du tampon à l'arrivée,

• ne pas effectuer de routage ;

— un pointeur sur une structure *sockaddr*, qui dans le cas d'une émission contient l'adresse du site et le nom du processus destinataire. Dans le cas d'une réception, cette structure permet de sélectionner l'origine du message que l'on souhaite recevoir.

9.3.2.8. *Primitive close()*

Quel que soit le mode utilisé (sur ou hors connexion), la primitive *close(sd)* libère les ressources attribuées : la connexion, les tampons, le descripteur de socket.

9.3.2.9. *Primitives de gestion des adresses et des structures*

UNIX possède un grand nombre de primitives de services qui permettent de simplifier l'usage du réseau et en particulier les phases initiales. L'ouvrage [10] en fournit une description détaillée. Les exercices de ce chapitre vous donnent un exemple de programmation. On trouve des procédures pour établir la correspondance entre les noms logiques des stations et leur adresse réseau. Celles-ci sont bien sûr rangées dans des fichiers systèmes. De même, on trouve des procédures capables de traduire les adresses réseau de leur forme décimale dans la forme hexadécimale et réciproquement.

9.3.2.9.1. Traduction des noms *gethostbyname(name)*

Cette fonction permet de connaître l'adresse de la machine. Sous UNIX cette interrogation consulte le fichier `/etc/host` local, puis le serveur NIS (sorte de pages jaunes informatisées) et enfin le serveur de noms si celui-ci est disponible. Cette dernière possibilité permet d'avoir l'adresse de n'importe quelle machine dans le monde. La déclaration de la structure est faite dans le fichier d'include `netdb.h`.

9.3.2.9.2. La fonction *gethostbyaddr(addr, len, type)*

Cette fonction rend le nom symbolique correspondant à une adresse. Elle remplit une structure de type *hostent* à partir d'une adresse fournie en paramètre.

9.3.2.9.3. La fonction *gethostname(name, namelen)*

Elle permet de connaître le nom de la machine locale sur laquelle le programme s'exécute.

9.3.2.9.4. Fonctions sur les serveurs et le réseau

Comme pour les adresses, il existe aussi des commandes pour trouver :

— le réseau sur lequel on travaille : *getnetbyname(name)* et *getnetbyaddr (netaddr, addrtype)*,

— le protocole utilisé *getprotobyname(name)* et *getprotobynumber(number)* ,

— le service : *getservbyname(name, proto)* et *getservbyport(port, proto)*.

9.4. Exemple du serveur FTP

Le serveur, ici FTP, attend une connexion TCP (cf. figure 9.10.). Le client connaît le service FTP et le nom du site qu'il veut atteindre. Sur la figure 9.11., le processus 1 231 sur buro1 exécute la commande « ftp RSM » ce qui provoque une demande d'ouverture de connexion avec le serveur FTP sur RSM.

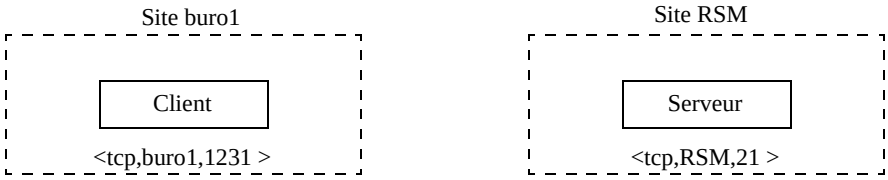

Figure 9.10. *Etat initial, le serveur FTP est en attente d'une requête*

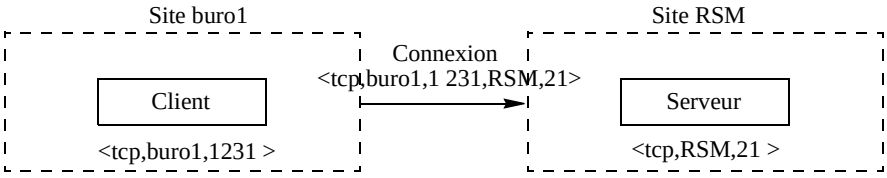

Figure 9.11. *Le client émet une requête de connexion avec le serveur*

A la réception, la primitive *accept*() du serveur crée une nouvelle association. Le serveur dérive un fils pour effectuer le traitement. Le fils est une copie exacte du père : il connaît donc les deux associations (cf. figure 9.11.).

Le serveur initial (père) :

— ferme la seconde association,

— retourne en attente de nouvelles requêtes.

Le serveur dérivé (fils) :

— ferme la première association,

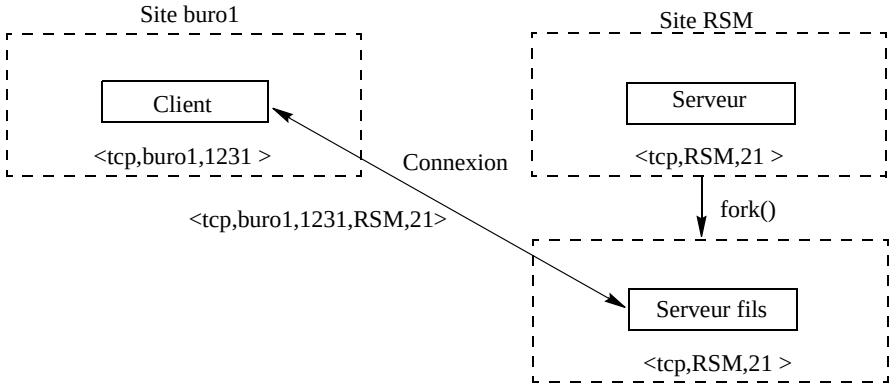

Figure 9.12. *Le serveur crée un processus fils auquel il attribue la connexion avec le client*

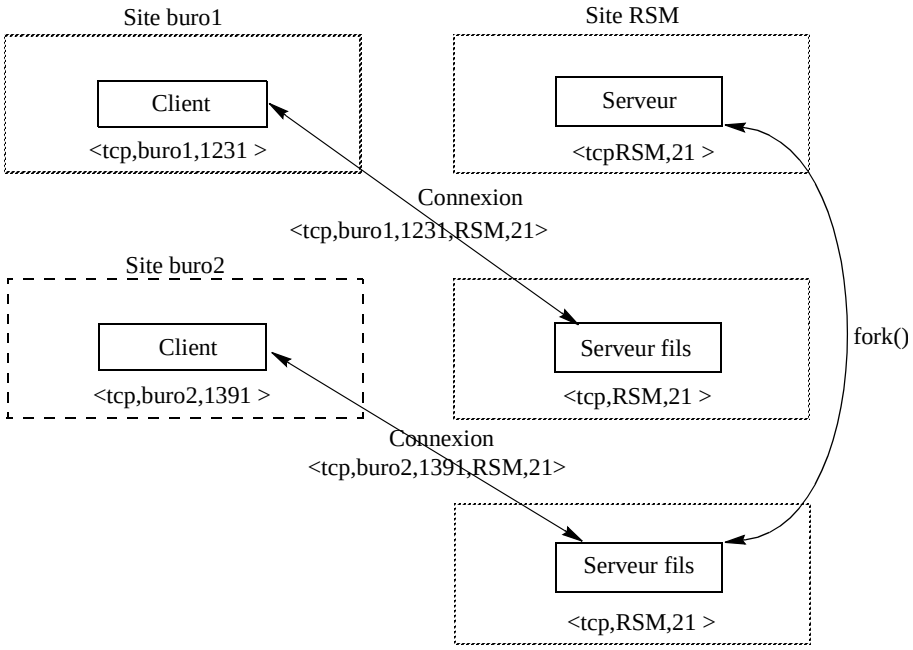

Figure 9.13. *Un nouveau client demande un service FTP sur RSM*

— fait la commande *exec* sur le programme FTP (ce qui a pour effet d'exécuter le programme FTP au lieu du programme serveur).

Ainsi, le serveur initial est toujours prêt à recevoir de nouvelles requêtes.

Le module TCP différencie les deux processus par les ports et les adresses des machines origines, comme le montre la figure 9.13.

<process source, process destinataire>

<1231,21>

<1391,21>

9.5. Conclusion

La création d'une application réseau est toujours basée sur le modèle client-serveur. L'agencement des primitives est presque immuable. Aussi, lorsque vous avez à écrire un tel programme, il est astucieux de partir d'un modèle déjà écrit.

Le schéma typique client-serveur n'est pas symétrique. Un des deux processus sera serveur de l'autre client. Chacun des deux sait le rôle qu'il doit jouer.

Il y a deux types de services :

— sur connexions <=>, le flot de données sera très similaire aux accès fichiers. Donc il faut ouvrir le flot avant usage ;

— hors connexion (datagramme)<=> échanges de bloc de données, il n'y a pas de séquence d'ouverture.

Un serveur est un processus permanent. Il attend une demande de service sur un mécanisme de communication. Un serveur a deux états :

— en attente d'une requête,

— service d'une requête.