Écrire un serveur HTTP

Maciej Zawadziński

HTTP est l'un des protocoles très répandus de la couche d'application, utilisés sur Internet actuellement . Grâce à sa simplicité et ses grandes fonctionnalités, il a été adapté à des applications diverses bien que le réseau internet soit la solution la plus populaire. Dans cet article, nous allons analyser les différentes questions, liées à la réalisation des serveurs de services réseau sur l'exemple d'une implémentation d'un simple serveur HTTP.

Sur l'auteur :

Maciej Zawadziński est étudiant à la Faculté de Mathématiques et d'informatique à l'Université de Wroclaw. Il est un programmeur enthousiaste des systèmes *BSD ; il est membre de l'Académie des systèmes alternatifs (*http://www.aaso.pl*).

Contact :
mzawadzinski@gmail.com

Le protocole HTTP est destiné à réaliser une communication suivant le modèle client-serveur. En tant que client (par exemple, via un navigateur Internet), vous envoyez une requête concernant une ressource définie (par exemple vous voulez connaître le contenu d'un site Web). Le serveur vous envoie en contre-partie des informations, le plus souvent le contenu du fichier demandé.

Observez un exemple d'une session HTTP. Vous vous connectez au serveur et vous envoyez la requête suivante :

```
GET /test/index.html HTTP/1.1
User-Agent: Mozilla/5.0 (X11; U;
   FreeBSD i386; en-US; rv:1.7.12)
Host: zawadzinski.pl:10000
```

Le serveur vous retourne alors une réponse du type :

```
HTTP/1.0 200 OK
Date: Sat, 25 Feb 2006 23:02:09 GMT
Server: SDJ_HTTPD/0.1
Content-Length: 93
Content-Type: text/html
<HTML><HEAD><TITLE>Écrire un serveur
   HTTP</TITLE></HEAD><BODY>
<P>Page de test
</BODY></HTML>
```

Aussi bien la requête que la réponse comprend trois parties. Selon qu'il s'agit d'un client ou d'un serveur HTTP, la première ligne contiendra toujours une requête de ressource ou le statut de sa progression. Ensuite, placez l'en-tête contenant des informations et des paramètres supplémentaires (sous forme `Nom-champ: valeur`). Vous interprétez le nom du champ sans respecter la casse et la valeur sous forme transmise. À la fin, vous ajoutez les données facultatives : le contenu (en anglais *body*).

Vous séparez aussi bien la ligne de début que les autres champs de l'en-tête à l'aide d'une séquence CRLF où CR si-

gnifie retour chariot (en anglais *carriage return*) et LF est le signe d'un saut de ligne (en anglais *line feed*). Le champ de l'en-tête peut avoir une valeur multiligne (par exemple, pour améliorer la lisibilité). Pour cette raison, vous interprétez toutes les lignes qui commencent par un espace ou un signe de tabulation comme la suite de la valeur du dernier champ. La fin de l'en-tête est signalée par une ligne vide.

Client

La première ligne de la requête comprend trois champs, séparés par un espace. Il s'agit respectivement de la méthode d'accès, du chemin des ressources et de la version utilisée par le protocole HTTP. Si vous employez le protocole HTTP/1.1, vous devez placer le champ `Host` (voir l'encadré : *HTTP/1.0 vs HTTP/1.1*) dans l'en-tête. Les autres champs sont facultatifs mais on précise en général les champs suivants : `User-Agent` (identifiant du navigateur), `Accept-Charset` (pages de code acceptées) `Accept-Encoding` (codage du fichier accepté, par exemple `gzip` pour la gestion de la compression de données) et d'autres, permettant de mieux adapter les serveurs avancés aux attentes d'un client donné.

Serveur

De même que pour le client, la ligne initiale contient trois champs, séparés par un espace. Cependant, vous commencez ici par la version du protocole HTTP et ensuite, vous placez le code de fin des opérations et le message associé. La Tableau 1 présente les codes les plus souvent rencontrés ainsi que leurs explications. Il est recommandé de transmettre dans l'en-tête les paramètres, comme `Content-type` (type de données retournées), `Content-lenght` (taille de données en octets) et Date (heure actuelle lors de la génération du résultat, précisée en GMT).

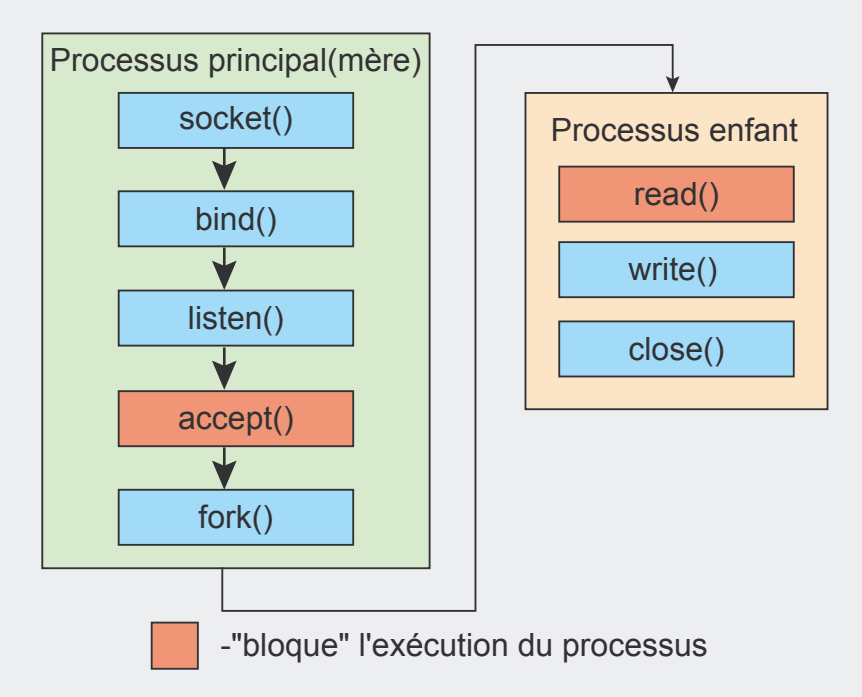

Figure 1. *Diagramme simplifié des appels système du serveur multi-processeurs HTTP*

Listing 1a. *Créer un socket d'écoute TCP/IPv4*

```
1  int init_socket4(int port) {
2      int lsock;
3      struct sockaddr_in saddr;

4      if( (lsock = socket(PF_INET, SOCK_STREAM, 0)) < 0 ) {
5          perror("socket()"); exit(-1);
6      }
7      memset(&saddr, 0, sizeof(saddr));
8      saddr.sin_family = PF_INET;
9      saddr.sin_port = htons(port);
10     saddr.sin_addr.s_addr = htonl(INADDR_ANY);

11     if( bind(lsock, (struct sockaddr*) &saddr, sizeof(saddr)) < 0 ) {
12         perror("bind()"); exit(-1);
13     }
14     if( listen(lsock, 1024) < 0 ) {
15         perror("listen()"); exit(-1);
16     }
17     return lsock;
18 }
```

Interface de socket

Vous vous imaginez à présent comment se déroule à peu près le processus de communication entre le client et le serveur HTTP. Avant d'implémenter votre serveur, regardez comment fonctionne une interface de socket, proposée par les noyaux de systèmes Unix (et d'autres).

Au jour d'aujourd'hui, la communication sur Internet se déroule principalement au moyen du protocole de couche transport(TCP) et réseau(IP). Actuellement, il s'agit toujours de IP, version 4 mais vous adapterez votre application pour qu'elle fonctionne aussi bien avec IPv4 que IPv6. La communication entre le serveur et le client se déroule par l'intermédiaire de socket. Chaque socket TCP/IP est doté d'une adresse et d'un numéro de port. La différence entre le client et le serveur est dans le fait que le serveur sélectionne le numéro de port déterminé à l'avance (admis ou déterminé par les standards) alors que le système d'exploitation du côté client attribue au socket le port et l'adresse (d'où découle la connexion) automatiquement. Toutes ces opérations s'effectuent lors de l'appel de la fonction `connect ()`, chargée de se connecter au serveur.

Lorsque vous écrivez un serveur, il faut créer un socket d'écoute qui attend de nouvelles connexions. Pour chaque

Listing 2. *Créer un socket d'écoute TCP/IPv6*

```
1  int init_socket6(int port) {
2      int lsock;
3      struct sockaddr_in6 saddr;

4      if( (lsock = socket(PF_INET6, SOCK_STREAM, 0)) < 0 ) {
5          perror("socket()"); exit(-1);
6      }
7      memset(&saddr, 0, sizeof(saddr));
8      saddr.sin6_family = PF_INET6;
9      saddr.sin6_port = htons(port);
10      saddr.sin6_addr = in6addr_any;

11      if( bind(lsock, (struct sockaddr*) &saddr, sizeof(saddr)) < 0 ) {
12          perror("bind()"); exit(-1);
13      }
14      if( listen(lsock, 1024) < 0 ) {
15          perror("listen()"); exit(-1);
16      }
17      return lsock;
18  }
```

Protocole HTTP/1.0 vs HTTP/1.1

La version actuelle du protocole HTTP est 1.1. Pourtant, la version 1.0 est employée pour de nombreuses applications simples. Pour cette raison, entre autres, il est demandé aux applications qui implémentent la version HTTP/1.1 de supporter également la version HTTP/1.0. Voici certains changements importants en HTTP/1.1 par rapport à la version 1.0 :

- *multihoming* – permet de supporter plusieurs domaines qui ont la même adresse IP attribuée (le champ Host est exigé dans les en-têtes de requêtes ; il contient soit le nom complet du domaine soit l'adresse IP du domaine auquel vous vous référez)
- *chunked-encoding* – envoi des données en *morceaux* ; avant de connaître la taille finale des données à envoyer (par exemple, générées par le script PHP), vous envoyez les parties actuelles et vous précédez chaque partie de sa taille
- *persistant connections* – support des requêtes dans le cadre d'une seule connexion au serveur (se connecter à chaque fois est peu efficace). De plus, le client peut envoyer les requêtes avant que le serveur retourne la réponse à la requête précédente.

Ordre des octets

Les architectures matérielles se divisent en général en deux catégories : *little endian* (elle stocke dans la mémoire l'octet le moins signifiant (de poids faible) au début, par exemple, l'architecture de type Intel x86 ou Alpha) et *big endian* (c'est l'octet le plus signifiant qui se trouve au début, par exemple, Sun Sparc et PowerPC)*. Afin d'unifier la façon d'envoyer les données entre les ordinateurs et de savoir *comment il faut lire*, on a introduit un ordre des octets du réseau qui est tout simplement un ordre utilisé dans les architectures *big endian*.
Les caractères individuels ne posent pas de problèmes mais les adresses IPv4 ont 4 octets et les numéros des ports TCP - 2 octets et ils doivent être enregistrés dans l'ordre des octets du réseau. Le noyau du système ne convertit pas ces valeurs automatiquement : il faut le faire soi-même.

Bibliographie

- R.Stevens *UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI*
- Unix Systems Programming
  http://www-users.cs.umn.edu/~bentlema/unix/
- HTTP Made Really Easy
  http://www.jmarshall.com/easy/http/
- Protocole T/TCP
  http://www.kohala.com/start/ttcp.html
- Documents RFC 1945 et RFC 2616

client qui se connecte, le noyau du système vous transmettra un nouveau socket, permettant dorénavant de communiquer avec le client donné.

Au travail !

La Figure 1 présente l'architecture la plus simple des serveurs réseaux. On le définit très souvent comme étant multi-processus car chaque requête est supportée par un nouveau processus fils et un processus de blocage car on utlisise des appels système. Ces appels arrêtent l'exécution du processus tant que les données d'entrée n'arrivent ou qu'une erreur ou une situation exceptionnelle ne survienne.

Regardez le Listing 1a, contenant la fonction, chargée de créer un socket d'écoute.

Lignes 4-6 : dans un premier temps, créez un socket de flux (variable `SOCK_STREAM`) pour le groupe d'adresses IPv4 (variable `PF_INET`). Le socket de flux signifie ici l'utilisation du protocole TCP pour la couche transport.

Lignes 7-10 : mettez à zéro et remplissez la structure d'adresses des données. Le premier champ (`sin_family`) définit le groupe d'adresses utilisé. Dans les autres champs, indiquez le numéro du port et l'adresse que vous voulez attribuer au socket. Les fonctions `htons()` et `htonl()` convertissent le paramètre de 2 et de 4 octets en ordre des octets du réseau. (utilisez les fonctions `ntohs()` et `ntohl()` pour convertir dans le sens inverse) (voir l'encadré : *Ordre des octets*). Une constante spéciale `INADDR_ANY` signifie que vous ne vous limitez pas à une seule adresse ; vous voulez recevoir les connexions de toutes les interfaces disponibles du système.

Si vous voulez toutefois attendre uniquement la connexion de votre ordinateur (donc sur l'interface « virtuelle » *loopback*), vous modifierez la ligne 12 en : `saddr.sin_addr.s_addr = inet_addr("127.0.0.1");`

La fonction `inet_addr()` sert à convertir la forme lisible à l'homme (en anglais *human readable*) en valeur binaire dans l'ordre des octets du réseau. Afin d'utiliser cette fonction, il est nécessaire d'inclure le fichier d'en-tête `arpa/inet.h`.

Lignes 11-13 : ensuite, liez l'adresse et le port au socket à l'aide de la fonction `bind()`. Elle prend en premier argument le numéro du descripteur, retourné auparavant par `socket()` ; les deux autres paramètres comprennent : la structure

d'adresses créée et sa taille. Puisque la fonction `bind()` est adaptée à travailler avec des groupes d'adresses différents, il faut lui préciser dans l'argument une structure d'adresses universelle pour laquelle vous effectuerez le typage.

Lignes 14-16 : paramétrez le socket en mode attente de connexions. Le premier argument transmis à la fonction `listen()` comprend, comme ci-dessus, le descripteur du socket alors que le second est le nombre maximal de connexions qui peuvent attendre dans la queue pour être ensuite acceptées par le serveur.

Si votre noyau supporte le protocole IPv6, vous pouvez alors utiliser dans votre programme la fonction du Listing1b ; cette fonction est chargée de créer un socket d'écoute TCP/IPv6. De plus, il sera toujours possible de supporter les requêtes de clients qui emploient IP, v4. (ce n'est pas toujours par défaut, par exemple dans le système FreeBSD, il est nécessaire de paramétrer sysctl net.inet6.ip6.v6only=0, pour que les sockets IPv6 supportent la connexion avec Ipv4) puisque la nouvelle version du protocole est retro-compatible ; elle mappe les anciennes adresses de 32 bits sur 128 bits.

La fonction `init_socket6()` ressemble à la fonction précédente, à cette différence près que vous utilisez ici les constantes et les structures pour le protocole IPv6. Il est en revanche impossible d'utiliser la fonction `inet_addr()` car elle ne fonctionne pas correctement pour les adresses IPv6. Si vous voulez lier une adresse concrètement au socket (par exemple l'adresse de l'interface loopback *::1*), vous utiliserez une fonction plus universelle pour remplir le champ de la structure d'adresses :

```
inet_pton(PF_INET6, "::1",
   &saddr.sin6_addr);
```

Voici la fin des modifications élémentaires nécessaires pour que votre serveur supporte les deux protocoles.

Listing 2. *Chargement de la configuration et boucle principale du programme*

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include <errno.h>
6  #include <signal.h>
7  #include <fcntl.h>
8  #include <dirent.h>
9  #include <time.h>
10  #include <sys/types.h>
11  #include <sys/socket.h>
12  #include <sys/stat.h>
13  #include <sys/param.h>
14  #include <sys/time.h>
15  #include <netinet/in.h>

16  char *Getenv(const char *name) {
17      char *val = getenv(name);
18      if( !val ) {
19          fprintf(stderr,"Enviromental variable %s is not set!\n", name);
20          exit(-1);
21      } else return val;
22  }

23  int main() {
24      int csock, lsock, pid;
25      lsock = init_socket4(atoi(Getenv("SERVER_PORT")));

26      if(chroot(Getenv("SERVER_ROOT"))< 0) {
27          perror("chroot()"); exit(-1);
28      }
29      if(setgid(atoi(Getenv("HTTPD_UID"))) < 0) {
30          perror("setgid()"); exit(-1);
31      }
32      if(setuid(atoi(Getenv("HTTPD_GID"))) < 0) {
33          perror("setuid()"); exit(-1);
34      }
35      daemon(0, 0);
36      signal(SIGCHLD, SIG_IGN);

37      while( 1 ) {
38          if( (csock=accept(lsock, 0, 0)) < 0 )
39              exit(-1);
40          if((pid=fork()) < 0)
41              exit(-1);
42          if(pid==0) {
43              close(lsock);
44              serve(csock);
45          } else close(csock);
46      }
47      return 0;
48  }
```

Lancement du serveur et sécurité

Vous lancez la plupart des services réseaux avec les droits root. Il en est ainsi parce que dans le cas contraire vous ne pouvez pas attribuer les ports, dont les numéros sont inférieurs à 1024, au socket. De plus, si vous disposez des droits administrateur, vous êtes capables de modifier le répertoire racine / (en anglais *root directory*) et permettre ainsi l'accès aux fichiers en dehors de ce répertoire (vous n'avez plus de problèmes liés aux chemins d'accès contenant *"../"*). Pour des raisons de sécurité, il est toutefois recommandé d'exécuter le moins de code possible avec les droits de l'administrateur ; si le service est compromis, les personnes non autorisées peuvent prendre le contrôle de l'ordinateur.

Regardez maintenant le Listing 2 qui lit la configuration et la boucle principale du programme.

Lignes 1-15 : ajoutez des fichiers d'entête, contenant les prototypes de fonctions et les constantes, nécessaires dans les autres parties de l'application.

Lignes 16-22 : définissez la fonction d'enveloppeur (en anglais *wrapper*) ; elle charge la variable d'environnement demandée : si elle échoue, elle affiche une erreur et termine le programme. Vous l'emploierez pour lire les paramètres de configuration du serveur.

Ligne 25 : initialisez le socket d'écoute

à l'aide de la fonction définie auparavant `init_socket4()` ou `init_socket6()`. Vous précisez dans son argument le numéro du port, obtenu grâce à la variable d'environnement `SERVER_PORT`. Convertissez-la en un entier à l'aide de la fonction `atoi()`.

Lignes 26-34 : modifiez le répertoire racine et les droits : modifiez le UID et le GID du processus grâce aux variables HTTPD_UID et HTTPD_GID.

Lignes 35-36 : appelez la fonction `daemon()` ; elle fait en sorte que le processus quitte le terminal et commence à s'exécuter en tâche de fond. Déclarez ensuite que vous voulez ignorer le signal `SIGCHLD` : ce signal est fourni au processus mère lorsque l'un de ses enfants s'arrête de fonctionner. Vous n'aurez toutefois pas besoin de cette information. Si vous ne le déclarez pas clairement, les processus terminés passeraient en mode *zombie*, en attendant que vous effectuiez une des fonctions du groupe `wait()` afin de récupérer le code de retour (en anglais *return code*).

Lignes 37-46 : boucle principale du programme : il faut attendre infiniment les connexions entrantes. La fonction `accept()` bloque le processus tant que quelqu'un ne se connecte pas. Lorsqu'une connexion arrive, vous créerez un nouveau processus à l'aide de la fonction `fork()`. Elle retourne la valeur 0 au nouveau processus et elle transmet l'identifiant du processus fils (en anglais *pid - process id*) au processus mère. Arrêtez l'exécution du socket, inutile dans ce contexte. L'objectif du nouveau processus consiste à gérer la requête du client : réalisez-le dans la fonction `serve()`.

Avant d'expliquer comment gérer la requête du client, nous définirons plusieurs fonctions auxiliaires. Regardez le Listing 3.

Lignes 2-18 : la fonction `send_headers()` crée et envoie une ligne initiale avec le statut d'exécution de la requête et l'en-tête HTTP ; cet en-tête contient le champ `Date` et `Content-length` ainsi que `Content-Type` (les deux derniers sont facultatifs). Créez la date au format adéquat pour HTTP dans les lignes 9-11 (il s'agit plus concrètement du format déterminé par le document RFC 1123).

Lignes 19-23 : la fonction `error_page()` génère la réponse contenant le code de l'erreur ainsi que le message et les envoie au client.

Lignes 24-29 : la fonction `send_resource()` envoie le nombre d'octets demandé du descripteur, transmis au client. Vous verrez dans un instant que cette implémentation est assez naïve et qu'il est possible de réaliser ce point d'une manière plus efficace.

Lignes 30-40 : la dernière fonction génère le contenu de la page, contenant la liste de fichiers du répertoire. Vous l'envoyez au client avec le code adéquat de la requête réussie ainsi que l'en-tête HTTP.

Le Listing 4 présente la dernière partie de votre serveur HTTP ; il contient la fonction de traitement de requête du client.

Lignes 8-13 : vous y lisez les données du client en paramétrant à l'avance la durée maximale de l'attente. Ceci permettra d'éviter une attaque potentielle de type DoS (en anglais Denial of Service) ; cette attaque utiliserait le fait que `read()` bloque l'exécution du processus tant qu'aucune donnée n'arrivent pas. Ouvrir un grand nombre de connexions sans envoyer de données pourrait entraîner la création de milliers de processus qui épuiseraient les ressources de système. Si vous paramétrez l'option `SO_RCVTIMEO`, la fonction `read()` retournera la valeur 0 après une durée déterminée si aucune donnée n'arrive de la part du client. Vous pouvez paramétrer cette durée à 10 secondes.

Malheureusement, cette solution ne vous protégera pas complètement contre les attaques de type DoS. Vous devriez ajouter aussi une série d'autres conditions de contrôle, comme le nombre de proces-

Listing 3. *Fonctions auxiliaires*

```
1  #define CRLF "\r\n"

2  void send_headers(int sock, const char *status,
3          int c_len, const char *c_type)
4  {
5      char header[512];
6      time_t t;
7      int n = snprintf(header, sizeof(header),
8              "%s HTTP/1.0" CRLF, status);
9      time(&t);
10      n += strftime(header+n,sizeof(header)-n,
11              "Date: %a, %d %b %Y %H:%M:%S GMT" CRLF,gmtime(&t));
12      if(c_len) n += snprintf(header+n,sizeof(header)-n,
13              "Content-length: %d" CRLF, c_len);
14      if(c_type) n += snprintf(header+n,sizeof(header)-n,
15              "Content-type: %s" CRLF, c_type);
16      n += snprintf(header+n,sizeof(header)-n, CRLF);
17      write(sock,header,n);
18  }

19  void error_page(int sock, const char *status) {
20      send_headers(sock,status,strlen(status),"text/html");
21      write(sock,status,strlen(status));
22      exit(0);
23  }

24  void send_resource(int sock, int fd, int size) {
25      char buf[32*1024];
26      int n;
27      while( (n=read(fd,buf,sizeof(buf))) > 0 )
28          if(write(sock,buf,n) <= 0) break;
29  }

30  void send_filelist(int sock, DIR *dirp, const char *path) {
31      struct dirent *cdir;
32      char line[2048];
33      send_headers(sock, "200 OK", 0, "text/html");
34      while( (cdir=readdir(dirp)) ) {
35          snprintf(line,sizeof(line),"<BR><A href='%s%s'>%s</A>",
36                  cdir->d_name,cdir->d_type==DT_DIR?"/":"",
37                  cdir->d_name);
38          write(sock,line,strlen(line));
39      }
40  }
```

Listing 4. *Gestion de la requête du client*

```
1  void serve(int sock) {
2      char buf[2048], path[1024];
3      char method[16], version[16];
4      struct stat st;
5      struct timeval tv;
6      int fd;
7      DIR *dir;
8      tv.tv_sec = 10;
9      tv.tv_usec = 0;
10      if(setsockopt(sock,SOL_SOCKET,SO_RCVTIMEO,&tv,sizeof(tv)) < 0)
11          error_page(sock, "500 Internal Server Error");
12      if(read(sock,buf,sizeof(buf)-1) <= 0)
13          exit(0);
14      if(sscanf(buf, "%15[^ \r\n\t] %1023[^ \r\n\t] %15[^ \r\n\t]",
15                  method,path,version) < 3 || path[0] != '/')
16          error_page(sock, "400 Bad Request");
17      if(strcmp("GET",method) != 0)
18          error_page(sock, "501 Method Not Implemented");
19      if(lstat(path, &st) < 0) {
20          if(errno == EACCES)
21              error_page(sock, "403 Forbidden");
22          else error_page(sock, "404 Not Found");
23      }
24      if(S_ISDIR(st.st_mode) && (dir=opendir(path))) {
25          send_filelist(sock,dir,path);
26          exit(0);
27      }
28      if(S_ISREG(st.st_mode) && (fd=open(path,O_RDONLY)) >= 0) {
29          send_resource(sock,fd,st.st_size);
30          exit(0);
31      }
32      error_page(sock, "403 Forbidden");
33  }
```

Listing 5. *Équivalent de send_resource() en utilisant mmap()*

```
1  void send_resource2(int sock, int fd, int size) {
2      void *ptr;
3      if( !(ptr=mmap(0, size, PROT_READ, MAP_PRIVATE, fd, 0)) )
4          error_page(sock, "500 Internal Server Error");
5      write(sock, ptr, size);
6  }
```

Listing 6. *Équivalent de send_resource() en utilisant sendfile()*

```
1  #if defined(__linux__)
2      #include <sys/sendfile.h>
3      #define Sendfile(in, out, offset, size) \
4          sendfile(in,out,offset,size);
5  #elif  defined(__FreeBSD__)
6      #include <sys/uio.h>
7      #define Sendfile(in, out, offset, size) \
8          sendfile(in,out,offset,size,NULL,NULL,0);
9  #else
10      #error "define Sendfile() macro according to your system!"
11  #endif
12  void send_resource3(int sock, int fd, int size) {
13      send_headers(sock, "200 OK", size, NULL);
14      Sendfile(fd, sock, 0, size);
15  }
```

sus exécutés et de connexions effectuées. Ceci dépasse toutefois le cadre de cet article.

Lignes 14-16 : chargez la méthode d'accès et le chemin de la ressource ainsi que la version du protocole HTTP, utilisée par le client. Vérifiez en même temps si la première ligne de la requête comprend bien trois champs séparés d'un espace.

Le paramètre `%15[^ \r\n\t]` est interprété comme une demande de chargement de 15 caractères différents de l'espace, du retour chariot (`'\r'`), du saut de ligne (`'\n'`) et d'une tabulation (`'\t'`). Le caractère ^ demande d'interpréter la classe de caractères comme « tout sauf ».

Lignes 17-18 : votre serveur ne supportera que la méthode GET : lorsque le client utilise une autre méthode, vous enverrez une information appropriée.

Lignes 19-32 : chargez les informations concernant le chemin (ressource) indiqué. S'il existe, vous envoyez la ressource adéquate selon que c'est un fichier ou un répertoire. Si vous êtes incapables de lire le chemin ou si l'objet indiqué est différent d'un fichier ordinaire (en anglais *regular file*) ou d'un répertoire (par exemple fichiers d'outils, queue fifo et d'autres), vous retournez une erreur (accès refusé).

Avant de lancer le serveur, n'oubliez pas de paramétrer les variables appropriées. Il est recommandé de créer un simple script, permettant de lancer le service, par exemple :

```
#!/bin/sh
export SERVER_ROOT="/usr/local/www"
export SERVER_PORT="8000"
export HTTPD_UID="80"
export HTTPD_GID="80"
/path/to/executable/sdj_httpd
```

Votre serveur ne supporte toujours pas de nombreux points importants, y compris les scripts CGI, les types MIME, les caractères spéciaux dans l'URL (y compris l'espace qui est transmis en tant que %20) et avant tout, les autres méthodes d'accès élémentaires (POST, HEAD) ainsi que les extensions ajoutées au HTTP/1.1. Ajouter certains d'entre eux constituera un exercice très utile avant de commencer à créer vos propres serveurs d'applications réseau.

Question d'efficacité

Mis à part la sécurité et la stabilité du travail de l'application, son efficacité est

Tableau 1. *Les codes choisis de fin des opérations, retournés par le serveur HTTP*

Code de fin et message	Sens
200 OK	Opération réussie
302 Moved Temporarily	Ressource transférée (l'en-tête doit contenir le champ Location avec le nouveau chemin de la ressource)
400 Bad Request	Requête incorrecte
403 Forbidden	L'accès à la ressource est interdit
404 Not Found	Ressource non trouvée
500 Internal Server Error	Une erreur est survenue du côté du serveur
501 Method Not Implemented	Le serveur ne supporte pas la méthode transmise dans la requête

un point très important. Retournons à la fonction `send_resource()` du Listing 3. Vous y chargez progressivement le fichier et l'envoyez au client via le socket. Cette méthode peut s'avérer peu efficace avec de grands fichiers parce que le noyau copie inutilement les données entre l'espace de l'utilisateur et l'espace du noyau. Il est possible de l'éviter à l'aide de l'appel système `mmap()` ; il mappe le fichier dans la mémoire et il est plus optimisé en ce qui concerne les opérations de lecture/écriture. Malheureusement, il limite la taille du fichier : la taille maximale actuellement autorisée sur les plates-formes de 32 bits s'élève à 2 GB.

Listing 5 présente l'équivalent de la fonction `send_resource()` en utilisant la fonction `mmap()`.

Une autre solution consiste à employer la fonction `sendfile()`. Malheureusement, elle n'est que partiellement portable, par exemple, sous Linux, elle prend d'autres arguments que dans FreeBSD. Il est toutefois possible d'écrire une macro adéquate pour garantir une compilation et un fonctionnement correct dans les deux systèmes mentionnés. Le Listing 6 présente cette solution.

Un autre point lié à l'efficacité du serveur est un nombre de paquets envoyés. À chaque appel de la fonction `write()`, un paquet est envoyé bien qu'il soit possible de placer les données, par exemple l'entête et le contenu de la page, dans un seul paquet. De plus, ces différences apparemment minimes, peuvent être significatives vu la spécification du protocole TCP. Il est possible de résoudre ce problème de deux manières. La première d'entre elles consiste à utiliser la fonction `writev()` ; elle prend en argument une liste de vecteurs (espace de mémoire) à envoyer. Une solution alternative consiste à paramétrer l'option du socket `TCP_NOCORK` sous Linux (ou respectivement `TCP_NOPUSH` sous *BSD) ; cette option serait chargée de bloquer l'envoi de données lors de l'appel de la fonction `write()` (et d'autres), tant que cette option n'est pas mise à zéro pour permettre l'envoi de données. Les données seront alors adaptées à la taille maximale du paquet.

Le protocole T/TCP est un autre point à observer ; il permet de réduire l'établissement d'une connexion en trois étapes. Il peut être très utile dans le protocole HTTP et d'autres protocoles où existent de nombreuses connexions de courte durée. Pour ne pas alourdir cet article, nous n'expliciterons pas d'avantage ces questions.

Conclusion

Vous avez vu les principes généraux de la structure d'un logiciel réseau côté serveur ainsi que certains problèmes qu'il est possible de rencontrer lors de sa réalisation. L'architecture présentée est l'une des nombreuses conceptions employées dans les serveurs de services réseau. Je pense qu'il faut en citer au moins plusieurs. Vous pouvez créer un serveur à un processus à l'aide des fonctions `select()` ou `poll()`. Elles ont cependant un défaut : elles sont peu efficaces lors d'un grand nombre de connexions . Pour cette raison, divers systèmes d'exploitation ont ajouté leurs propres fonctions pour supporter efficacement l'entrée/sortie multiple. Sous Linux, il s'agit de `epoll()` et dans FreeBSD, c'est `kqueue()`. En pratique, on emploie des combinaisons différentes des techniques mentionnées. Par exemple Apache lance au démarrage plusieurs processus et partage les connexions entrantes. Le sujet de la structure du logiciel réseau est tellement large que de nombreux ouvrages ont été consacrés uniquement à cette question. Je vous recommande vivement de consulter au moins l'un d'entre eux. ■