Un serveur HTTP minimaliste

Objectifs du DM :

1. Découvrir le protocole HTTP
2. Utiliser des sockets en C
3. Programmer un serveur HTTP basique

1 À la découverte du protocole HTTP

HTTP (`HyperText Transfer Protocol`) est un des principaux protocoles du web, destiné à l'échange de documents reliés entre eux par des liens *hypertextes*. Il s'agit d'un protocole de la couche application du modèle OSI, et basé sur les protocoles plus bas niveau TCP et IP. Le client se connecte au serveur, lui envoie une requête (en général il demande le contenu d'un fichier html), éventuellement accompagnée de données brutes (le contenu d'un formulaire, par exemple) ; le serveur lui répond en lui renvoyant d'abord des en-têtes de réponse avant de lui envoyer le contenu du fichier. Il s'agit d'un protocole sans état : si le client effectue plusieurs requêtes successives auprès d'un même serveur, le serveur n'a pas de moyen de s'en apercevoir. Ainsi, lors de l'accès à une page protégée, les identifiants sont transmis à chaque requête, même si l'utilisateur final ne les entre qu'une fois.

1.1 La requête

Une requête HTTP est constituée de 4 sections :
- Une ligne de requête, précisant le service désiré, une ressource (un fichier) ainsi que la version HTTP utilisée, généralement HTTP/1.1.
- Des *headers*, un par ligne, qui servent à apporter des informations facultatives au serveur : les langues acceptées par le client, le type du navigateur, des cookies, les types de compression acceptées, etc.
- Une ligne vide, pour signaler la fin des headers.
- Un corps de requête optionnel, contenant par exemple le formulaire.

HTTP/1.1 définit 8 types de requêtes différents :
- GET : demande au serveur de lui envoyer la ressource précisée.
- HEAD : identique à GET, sauf que seuls les en-têtes seront transmis.
- POST : identique à GET, sauf que des informations supplémentaires seront transmises dans le corps de la requête.
- PUT : envoie une ressource au serveur.
- DELETE : efface une ressource sur un serveur.
- TRACE : le serveur renvoie au client sa propre requête.
- OPTIONS : permet au client de connaître les méthodes supportées par le serveur.
- CONNECT : transforme la requête en tunnel TCP/IP (pour permettre les communications HTTPS).

Voici un exemple de requête :

```
GET /index.php HTTP/1.1
Host: www.example.com
```

1.2 La réponse

La réponse d'un serveur est également constituée de 4 sections :
- La première précise à nouveau le protocole utilisé, ainsi qu'un code de retour (200 = OK, 404 = NOT FOUND, etc.).
- Des *headers*, avec le même rôle que pour la requête.

– Une ligne vide, pour signaler la fin des headers.
– Le corps de la réponse.

Voici un exemple de réponse :

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix)  (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Etag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Content-Length: 438
Connection: close
Content-Type: text/html; charset=UTF-8
```

1.3 Jouons avec `telnet`

`telnet` est un petit programme permettant d'établir des connexions TCP/IP, en précisant uniquement l'adresse du serveur et le port choisi. Ainsi, `telnet graal.ens-lyon.fr 80` se connecte en TCP au serveur web de GRAAL. Si vous voulez effectuer une requête HTTP, il faut donc taper à laGET mgallet/index.php HTTP/1.1 main les différentes lignes données en 1.1. Essayez la requête suivante en vous connectant à GRAAL.ens-lyon.fr :

```
GET /~mgallet/dm.php? HTTP/1.1
User-Agent:Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_5_6; fr-fr)
Accept:application/xml,application/xhtml+xml,text/html;
Accept-Language:fr-fr
Accept-Encoding:gzip, deflate
Connection:keep-alive
Host:graal.ens-lyon.fr
```

2 Premiers sockets

Naturellement, pour créer un serveur web, il faut être capable de faire passer des informations par le réseau. Certes, nous pourrions aller faire le tri dans les paquets TCP ou même IP qui transitent par les interfaces réseaux, mais il est nettement plus simple d'utiliser une couche d'abstraction très classique : les *sockets*. Les sockets permettent de considérer les connexions réseaux comme de simples fichiers, suivant ainsi le principe UNIX selon lequel tout est fichier. Ainsi, pour lire les données envoyées par le client ou pour lui envoyer des données, nous lisons ou écrivons le fichier correspondant au socket. Les fonctions usuelles (`fwrite, fprintf, fgets`, ...) sont alors couramment utilisées.

D'abord, il créer un nouveau socket, en utilisant la bien-nommée fonction`socket` :

```
#include <sys/socket.h>
int listen_fd;
int one = 1;
listen_fd = socket(PF_INET, SOCK_STREAM, 0);
setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR, & one, sizeof(int));
```

Ensuite, il est nécessaire de lui préciser sur quel port et sur quelles adresses écouter :

```
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
struct sockaddr_in addr;
memset(& addr, 0, sizeof(struct sockaddr_in));
```

```
addr.sin_family = AF_INET;
addr.sin_port   = htons(port);   /* port: 2 octets en ordre de réseau */
addr.sin_addr.s_addr = htonl(INADDR_ANY);
bind(listen_fd, (const struct sockaddr*) &addr, sizeof(sockaddr_in));
```

Enfin, on peut demander au socket de se mettre à l'écoute des connexions entrantes, en lui précisant la taille de la file d'attente des connexions :

```
listen(listen_fd, 15);
```

À cette étape, le socket écoute les connexions réseaux entrantes. Avec la fonction `accept`, on prend la première connexion de la file d'attente, on crée un nouveau socket pour elle et on peut alors accéder à la connexion sous forme d'un fichier en lecture/écriture. Cette fonction va attendre tant qu'il n'y a pas de connexion en attente.

```
int client;
FILE *stream;
char buff[21];
client = accept(listen_fd, NULL, 0);
stream = fdopen(client, "r+");
setlinebuf(stream);
```

Question 2.1. *En partant du code ci-dessus, écrivez un petit programme C qui écoute un port donné (par exemple le port 1234) et qui affiche à l'écran les données reçues. Après avoir reçu des données, il enverra un message au client. Il pourra n'accepter qu'une connexion à la fois.*

Pour tester votre code, vous utiliserez `telnet` à partir d'un autre terminal.

3 Un petit serveur web

3.1 Une première version fonctionnelle

À partir des quelques informations précédentes, et en utilisant les TP précédents, nous avons tous les outils à notre disposition pour créer un véritable serveur web. Ainsi, nous devons créer un logiciel qui écoute les connexions entrantes, analyse les en-têtes, puis envoie le fichier demandé. Cependant, s'il y a beaucoup de clients, un traitement séquentiel risque de prendre trop de temps, et des connexions seront perdues. Les serveurs web actuels travaillent différemment, avec plusieurs threads. Un thread principal écoute les connexions entrantes, et dès qu'une connexiont `client` est accepté, crée un nouveau thread en lui transmettant l'identifiant de cette connexion. Ainsi, plusieurs connexions peuvent être traitées en parallèle.

Question 3.1. *Modifiez votre programme précédent pour qu'il puisse accepter plusieurs connexions en parallèle.*

À partir de maintenant, nous allons supposer que les connexions suivront scrupuleusement le protocole TCP. Cependant, nous allons nous limiter aux méthodes GET et HEAD, ainsi qu'aux codes d'erreur 200 (OK) et 404 (NOT FOUND).

Question 3.2. *Complétez votre programme pour qu'il accepte une requête de la forme GET /chemin HTTP/1.1, qui sera interprétée comme une demande de téléchargement du fichier se trouvant à la position chemin par rapport au répertoire où est lancé le serveur. Pour l'en-tête de réponse, vous pouvez vous contenter du code d'erreur HTTP et de la taille du fichier (`Content-Length`). Après avoir répondu à la requête (mode `close`) ou quand le client clôt le socket (mode `keep-alive`), le thread ferme la connexion et quitte*

3.2 Quelques extensions

Question 3.3. *Que sont les types MIME ? Implémentez quelques uns des plus courants dans votre serveur.*

Pour vérifier que le serveur fonctionne correctement, il est d'usage d'enregistrer les informations de connexion dans un fichier de journalisation.

Question 3.4. *Implémentez une fonction de journalisation, en faisant attention à ce qu'un seul thread puisse y écrire à la fois.*

Dans la section précédente, le serveur crée un nouveau thread à chaque connexion. Cependant, cette méthode ne convient pas, car elle est trop lente. Les serveurs web actuels créent *a priori* un ensemble de N threads, et dès qu'il y a une connexion entrante, attribuent la connexion au premier thread inoccupé. Après avoir traité la requête, le thread repasse en attente. S'il n'y a pas assez de threads, le serveur peut en créer un nouveau, ou attendre qu'un thread soit libéré.

Question 3.5. *En utilisant les conditions vues dans un TP précédent, modifiez votre serveur pour qu'il ne crée ses threads qu'au lancement.*

Cependant, si la ressource bloquante est la connexion réseau des clients et non le processeur (ou le disque dur) du serveur, les différents threads risquent d'être sous-utilisés. La fonction `pselect` permet justement à un seul thread de surveiller différents flux ouverts (ici les sockets réseaux), afin d'utiliser un seul thread pour plusieurs connexions réseaux, le thread passant à la connexion suivante pendant les transferts de données.

Question 3.6. *Modifiez votre programme afin d'utiliser* `pselect` *pour la gestion de différentes connexions par un même thread. L'ensemble des sockets réseaux ouverts peuvent êtr spécifique à un thread ou partagés par tous les threads.*

3.3 S'il vous reste du temps

Naturellement, le serveur reste très basique, mais voici quelques idées pour l'améliorer, s'il vous reste du temps :

Question 3.7. *Ce n'est pas pratique de devoir recompiler le serveur si on veut modifier légèrement la configuration. Vous pourriez ajouter le support d'un fichier de configuration, donnant par exemple le dossier contenant les pages html, le port à écouter, les documents en cas d'erreur, etc.*

Question 3.8. *Comment se passe l'envoi de fichiers vers le serveur ? Essayez d'implémenter cette fonction.*

Question 3.9. *Comment est gérée l'authentification HTTP par le serveur ? Quel est l'inconvénient du principe sans-état de HTTP ? Implémentez une authentification basique des utilisateurs.*