Mini-serveur web en C

Réalisé par **Camille Gobert** et **Hugo Manet** Ce projet a été développé dans le cadre du cours *Systèmes et réseaux* de l'ENS (en 2017).

Description

Il s'agit tout simplement d'un petit serveur web, basé sur HTTP/1.1, capable de traiter de simples requêtes et d'y répondre. Il ne dipose que d'un pauvre support de la norme, et souffre de quelques bugs et lacunes ; mais il a également été concu afin que le code soit bien expliqué, évolutif, et de façon à nous permettre de manipuler plusieurs parties du système (fichiers et répertoires, réseaux et sockets, signaux, processus, tuyaux, etc).

Il ne faut donc pas trop lui en demander ; mais il est normalement apte à répondre aux simples requêtes HTTP. Essayez par exemple de visiter `localhost:4242/test.html` une fois lancé !

Version HTTP supportée : HTTP/1.1. **Méthodes HTTP supportées :** GET, HEAD.

Quelques champs de la requête étaient initialement supportés, mais leur support a du être retiré au dernier moment, car l'analyseur syntaxique qui se chargeait des requêtes des clients posait certains soucis, et un analyseur minimaliste a été réécrit en urgence... Le code en question se trouve dans le fichier `src/parse_header.c`.

Organisation du projet

Le projet est découpé en plusieurs répertoires et fichiers, suivant la logique suivante :

- `src` contient les fichiers sources.
- `build` contient les fichiers objets résultant de la compilation modulaire.
- `www` contient les fichiers pouvant être demandés au serveur.
- `misc` contient des données annexes, et notamment un graphe en Dot décrivant l'implémentation limitée de traitement des entêtes HTTP/1.1 que nous avions prévu de supporter.

Code source et modules

Le code source est découpé en plusieurs modules majeurs, que voici. De plus, le code de chacun se retrouve organisé en plusieurs parties logiques.

- `server` implémente notamment le serveur TCP et la manipulation des structures `Client` et `Server` : il se charge de toutes les opérations réseau du programme.
- `http` implémente le traitement des requêtes et des réponses HTTP, et la manipulation des divers types et structures associées (`HttpHeader`, `HttpContent`, `HttpMessage`, `HttpCode`, etc).
- `parse_header` implémente le traitement des entêtes HTTP, et plus particulièrement leur analyse syntaxique. Une grande partie d'un ancien code a été commentée car nous y avons trouvé des bugs majeurs et inattendus au moment du rendu...
- `file_cache` implémente le cache de fichier offrant une structure abstraite et performante pour

manipuler les données aptes à être demandées au serveur, et envoyées sur le réseau (structures `File`, `Folder` et `FileCache`). Il ne dispose cependant pas de politique de mise à jour dynamique, et demeure relativement statique à l'heure actuelle.
- `system` implémente des fonctions auxiliaires faisant appel à des fonctions systèmes. *Ce module étant également censé contenir des wrappers de fonctions systèmes ou standard utilisées dans tout le projet (telles que* `malloc()`, `read()`, *etc), notamment à des fins de propreté de code et de vérification systématique des valeurs de retour et des des erreurs potentielles, mais nous n'avons finalement pas eu le temps de le faire.*
- `toolbox` implémente fonctions utilitaires variées, utilisées par tous les autres modules : gestion des erreurs, affichage en couleur, opérations sur des chaînes de caractères, etc.
- `main` implémente simplement une initialisation et une terminaison propre du serveur, ainsi qu'une gestion du signal SIGINT.

Compilation

Notre serveur web dépend majoritairement de fonctions de la bilbiothèque standard C et Unix. De plus, il fait appel à deux programmes externes très répandus, mais devant dependant être installés. Il s'agit de `file` (pour deviner le type et l'encodage des fichiers) et de `gzip` (pour compresser certains fichiers).

Afin de compiler le programme, il suffit de se placer dans le répertoire racine (où se trouve le fichier `Makefile`), et d'utiliser la commande `make`. Par défaut, le compilateur utilisé est `clang`.

La commabde `make clean` permet de supprimer les fichiers objets et l'eéxcutable.

Utilisation

Une fois compilé, un exécutable `webserver` est produit à la racine du projet. Celui-ci peut alors être immédiatement lancé, et n'attend aucun argument sur la ligne de commande. Des informations plus ou moins utiles sont alors affichées, au lancement, et durant l'exécution du serveur.

Terminaison du programme et durée de sûreté liée à TCP

Lorsque le programme se termine via `exit()`, une fonction d'avant-terminaison est appelée, afin de correctement fermer les sockets ouvertes, de détruire certaines structures de données, etc. Celle-ci est également appelée lors de la réception d'un signal SIGINT (Ctrl + C).

Cependant, de par le fonctionnement du protocole TCP, le noyau réserve l'utilisation de la même adresse ou du même port pendant quelques secondes (voire minutes, sur certains systèmes). Il est possible de configurer ce comportement, qui est mis en palce pour assurer un déroulement de la fin de la connection TCP aussi bon que possible ; mais nous ne l'avons pas fait. Il se peut donc que le programme ne puisse pas être relancé immédiatement après avoir été quitté, sous peine d'avoir une erreur de type `bind() failed: Address already in use`.

Configuration

Le serveur dipose de quelques degrés de liberté. Outre les diverses valeurs définies dans le préprocesseur C, il existe une structure de type `ServParameters` (définie dans `src/server.h`), ayant pour but de regrouper les paramètres importants, qui auraient par exemple pu être décrits dans un fichier chargé au lancement du serveur, si nous avions eu plus de temps.

Ces paramètres se voient attribués des valeurs par défaut. En particulier :

- Le serveur se connecte à n'importe quelle adresse locale (`INADDR_ANY`) sur le port `4242`. Il est donc possible de s'y connecter à l'adresse `localhost:4242` (à l'aide de `telnet` ou d'un navigateur web par exemple).
- Le répertoire racine dans lequel les fichiers sont cherchés est `./www`. Un petit site et quelques fichiers de test y sont déjà placés.

TCP sur port 80

Par convention, les serveurs HTTP utilisent naturellement le port 80. Sur la plupart des machines Unix, les ports de numéro inférieur à 1024 sont réservés, et requierent que le programme dipose d'une autorisation particulière pour être utilisés. Le serveur étant très peu avancé, nous ne lui avons pas donné de tels droits à l'heure actuelle (mais cela se change rapidement).

Choix techniques

Notre serveur s'articule autour de trois sous-programmes principaux :

- Un serveur TCP.
- Un traitement des requêtes et réponses HTTP.
- Un cache de fichiers.

La majorité des choix techniques sont expliqués dans le code, qui contient de nombreux commentaires. Ci-dessous sont résumés les points majeurs :

- Le serveur utilise des sockets TCP sur IPv4 (il était prévu de passer à IPv6, et il s'agit d'une amélioration simple et rapide à mettre en place).
- De nombreuses structures et types personnalisés ont été crées afin de donner du sens à un code devenant important. Ils sont souvent décrits dans les fichiers entête C.
- Les clients forment une liste doublement chaînées, facilitant insertion, suppression, et parcours de l'ensemble des clients du serveur. Ils sont tous indépendants, possèdent leurs propres statut, buffers, et leurs propres structures de messages HTTP (requête et réponse).
- Le serveur maintient une liste des clients. En fonction de leurs statuts, il construit une liste cohérente de sockets à écouter en lecture ou en écriture, afin de rendre ces opérations non-bloquantes, via la fonction `poll()`.
- Chaque client stocke au plus une requête lui étant adressées ; une fois bien reçue, celle-ci est analysée, traitée, et une réponse adaptée est produite, puis envoyée. Il est alors de nouveau prêt à en recevoir.
- Bien que différents paradigmes aient été discuttés, une fois lancé, le serveur n'est constitué que d'un unique processus. Certaines parties du code ont toutefois été développées avec l'idée

d'utiliser des threads (un pour tous les clients, un par groupe de clients, etc), et ce notamment pour analyser les requêtes HTTP en arrière-plan, pendant que le serveur continue de lire et d'écrire sur le socket via le noyau en avant-plan (étant donné qu'il s'agit d'opérations relativement indépendantes).
- Un système de cache de fichiers est disponible. Au lancement du serveur, il est construit suivant le contenu du répertoire racine (`www` par défaut). Nous n'avons pas eu le temps de lui développer de politique particulière, mais c'était l'une des idées initiales.
- Les structures abstraites de fichiers et dossier en cache contiennent des informations utiles, qui permettent notamment de limiter (1) les calculs et (2) les accès au disque lors du traitement de requêtes HTTP, en proposant des informations pré-calculées et stockées en mémoire, telles que le type d'un fichier, ou son contenu déjà compressé via `gzip`. Les opérations coûteuses sont ainsi réduites, et préférablement réalisées au démarrage du serveur uniquement.

Difficultés rencontrées

La principale difficulté de ce projet fût le temps requis pour combiner un code C propre et réfléchi et une implémentation apte à détecter ou éviter diverses erreurs, à supporter un sous-ensemble cohérent d'HTTP/1.1 (ce qui est en pratique un échec, mais fut instructif !), et à offrir un serveur avec un minimum de performances.

Le travail en binôme ne fut par non plus très efficace, et assez inégal.

Enfin, la dernière difficulté rencontrée concerne le choix du langage. Je (Camille) tenais à utiliser un langage bas niveau tel que C, afin d'être proche du système que nous allions manipuler, et de diposer de nombreuses fonctions, options, etc, toutes relativement bien documentées. Hugo proposait C++, mais ne connaissant pas (bien) ce langage, j'ai préféré me concentrer sur l'utilisation plus poussée du C que sur la découverte hasardeuse du C++, toujours dans l'optique de produire un code peut-être incomplet, mais propre et bien compris.

Il est toutefois indéniable que le C impose certaines lourdeurs, et ne rende pas le développement très rapide ; il est par exemple particulièrement désaréable de manipuler des chaîne sde caractères, des structures de données de base, ou encore des expressions régulières. Ces raisons participent probablement au fait que ntore implémentation d'HTTP soit plus que limitée.