POLYTECH NICE-SOPHIA

SI3 / 2012-2013

Erick Gallesio

Projet SI3

Un serveur Web modulaire

Le but de ce projet est d'écrire un serveur Web modulaire en C. Les principales caractéristiques de ce serveur, que nous appelerons `polyweb`, sont détaillées ci-dessous. Toutefois, avant de commencer à décrire le serveur à proprement parler, un certain nombre de principes sur les réseaux et le protocole HTTP vont être présentés. Ces principes sont ultra simplifiés pour rester compréhensibles par tout un chacun.
Les fichiers qui vous sont distribués permettent de construire et donc d'exécuter `polyweb`. Par contre la plupart des codes sources de ce programme ne vous sont pas livrés. Votre travail va consister à réécrire les différents composants du serveur et à reconstruire les sources de ces composants. Ainsi, vous aurez toujours un programme qui fonctionne en substituant petit à petit les composants qui vous ont été distribués, par ceux que vous avez réécrits.
Parmi les fichiers sources qui vous sont distribués, il y a un certain nombre de *headers* (des ".h") qui définissent l'interface des composants qui vous sont donnés. Ces fichiers sont des **contrats** et ne **doivent pas** être modifiés. Les modules que vous remplacerez doivent être **compatibles** avec l'interface décrite dans le *header* correspondant.

Réseaux: principes de base

Lorsqu'on veut travailler avec une machine distante (pour copier des fichiers, se connecter, parcourir une page web, jouer, lire son mail, ...), il faut en fait connaître 3 informations:

- le nom de la machine sur laquelle on veut se connecter (on l'appellera le serveur).
- le protocole que l'on va utiliser: HTTP pour le Web, IMAP ou POP3 pour le mail, SSH pour se connecter (ou exécuter une commande) sur la machine distante, ...
- le numéro du port que l'on va utiliser. La plupart du temps, le numéro de port est implicite (e.g. 80 pour faire de l'HTTP, 143 pour faire de l'IMAP, 993 si l'IMAP est sécurisé, 22 pour SSH, ...). Rien n'empêche de faire de l'HTTP sur un autre port que 80 ou du SSH sur le port 22. Nous verrons cela plus tard.

Bref, quand vous entrez par exemple sur votre navigateur l'adresse `http://wwww.google.fr` pour vous connecter à Google, le navigateur établit en fait une connexion sur le serveur nommé `www.google.fr` en utilisant le protocole HTTP. Comme vous n'avez rien dit sur le port utilisé, le navigateur utilisera le port par défaut (i.e. le port 80). En fait, vous auriez pu entrer l'URL `http://wwww.google.fr:80` pour expliciter le port de connexion.

Réseau en C

Il existe de nombreuses primitives pour faire du réseau en C. Malheureusement celles-ci, sont d'assez bas niveau. Il ne sera pas nécessaire de comprendre comment les choses marchent dans le détail. Les fichiers `src/network.c` et `include/network.h` permettent des s'abstraire des détails sordides. Trois primitives sont définies:

- `create_server`: permet de créer un serveur en attente sur le port qui a été passé en paramètre.
- `accept_connection`: permet d'attendre qu'un client arrive. Lorsqu'un client se présente, la fonction remplit une structure de données de type `struct client_info` qui contient des informations sur le client (son numéro IP et le nom de sa machine) ainsi qu'un descripteur de fichier permettant de discuter avec le client. Ce descripteur de fichier est ouvert en lecture et en écriture. Comme le protocole HTTP, utilisé par le web, est basé sur la notion de ligne, cette structure propose aussi deux champs `fin` et `fout` qui contiennent des `FILE *` permettant la lecture et l'écriture va et vers le client en mode "bufferisé".
- `shutdown_connection` permet de fermer la connexion vers le client.

Deux exemples de serveurs sont donné dans le répertoire `tests` (`simple-server` et `multi-server`):

- `simple-server.c` est un serveur simple pour illustrer comment se servir du fichier `network.c`. Ce programme lit une ligne de son client et lui renvoie cette ligne convertie en majuscules.
- `multi-server.c` est une petite modification du serveur précédent. Cette version lance un nouveau processus à chaque fois que l'on a un nouveau client. On peut donc avoir plusieurs clients simultanés. Pour chaque clien, les caractères saisis sont mis en majuscules (et seulement ceux qui sont saisis dans le client). Dans la fenêtre du serveur, on voit par contre toutes les lignes lues, quelque soit le client où elles on été saisies.

Pour travailler avec ce serveur (c'est à dire pour être un client de ce serveur), il suffit de lancer le programme dans un terminal et de lancer la commande suivante[1] d'un **autre** terminal[2]:

```
$ netcat localhost 1234
```

puisque le server tourne sur le port 1234 par défaut.

HTTP: principes de base

Le protocole HTTP est un protocole relativement simple utilisé pour "servir" des pages Web. Nous n'allons pas voir ce protocole en détails ici et nous nous bornerons aux méthodes GET et POST qui sont les plus utilisées (la plupart des *vrais* serveurs Web n'implémentent en général que trois ou quatre des sept méthodes définies par le protocole HTTP).
Une requête HTML est formée d'une suite de lignes dont la dernière est une ligne vide. La première ligne contient la commande à exécuter (pour accéder à une page, on utilise la requête GET). Ensuite, se trouvent des lignes précisant la requête (le type de fichier que l'on sait traiter, les langues connues, le nom de navigateur, ...). Voici un exemple de requête pour accéder à la page principale du site `http://www.polytech.unice.fr`

```
GET / HTTP/1.0
Host: www.polytech.unice.fr
User-agent: Fait a la main v0.0
          <== ligne vide: on a fini de discuter
```

Ici , on déclare que l'on veut accéder la page principale (elle s'appelle "/") et que l'on utilise la version 1.0 du protocole HTTP. La ligne `Host:` indique que l'on veut se connecter sur le serveur `www.polytech.unice.fr` (ce qui est déjà le cas, mais un serveur Web peut servir plusieurs domaines) et que notre browser Web s'appelle "Fait a la main v0.0"). Les deux dernières lignes ne sont, en principe, pas obligatoires, mais certains serveurs Web les exigent; il est donc plus prudent de les mettre systématiquement.
Essayons maintenant une connexion sur le serveur Web de l'école. Pour cela, nous allons encore utiliser la commande `netcat`. Un exemple de session est donné ci-dessous.

[1] la commande `netcat` (aussi appelée `nc`) est un une sorte de `cat` sur le réseau (d'où son nom ;-).
[2] Par convention, la machine locale est toujours accessible par le nom `localhost`.

```
balico$ netcat www.polytech.unice.fr 80
GET / HTTP/1.0
Host: www.polytech.unice.fr
User-agent: Fait a la main v0.0
                        <= ligne vide = on a fini notre requête
HTTP/1.1 200 OK         <= À partir de là, entête de la réponse
Date: Wed, 02 Jan 2008 20:26:21 GMT
Server: Zope/(Zope 2.7.8-final, python 2.3.5, linux2) ZServer/1.1
Content-Length: 34913
Content-Language: fr
Expires: Sat, 1 Jan 2000 00:00:00 GMT
Pragma: no-cache
Content-Type: text/html;charset=utf-8
Connection: close
                        <= ligne vide: la réponse proprement dite
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
                "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
        <head>
                <title>Test Page for Apache Installation</title>
        </head>
.........  pendant longtemps
```

La réponse est formée de deux parties (séparées là encore par une ligne vide). La première partie (l'entête) est une suite de lignes qui indiquent comment les choses se sont passées (200 dit que tout va bien, 404 que la page n'existe pas ...) puis des informations diverses comme le nom du serveur, la taille du résultat (ici 34913 octets) et le type MIME du document (ici on dit que la réponse et du "text/html"). La seconde partie (le corps) est le code HTML proprement dit. Quand le serveur à répondu, il ferme la connexion.

Un embryon de serveur web vous est donné dans le fichier `tests/serveur-web.c`.

Nous avons vu comment se connecter avec la commande `netcat` à un serveur, mais cette commande permet aussi de se faire passer pour un serveur grâce à son option `-l` (pour *listen*). Par exemple, nous pouvons ouvrir un terminal et exécuter la commande:

```
balico$ netcat -l 1234
```

pour définir un serveur sur la machine locale qui se met en attente sur le port 1234. Dans un autre terminal, nous pouvons ensuite exécuter la commande:

```
balico$ netcat localhost 1234
```

pour créer un client qui discute sur la machine locale sur le port 1234.

Ici, nous avons utilisé une seule machine mais, dans le cas général, nous pourrions utiliser deux machines distinctes. Une fois que le client et le serveur sont lancés, nous avons 2 commandes `cat` qui sont connectées l'une avec l'autre à travers le réseau. Cela veut dire que si l'on tape des caractères dans un terminal, ils apparaissent dans le second (et vice-versa bien-sûr). En quelque sorte, nous venons donc de nous construire un programme de *chat* très basique.

Quelques exemples d'utilisation de netcat

1. En utilisant `netcat` en mode serveur dans un terminal sur le port 1234, on peut visulaiser la requête HTTP qui est envoyée par un navigateur au serveur. Ainsi, en utilisant l'URL `http://localhost:1234`, on voit les caractères qui sont envoyés au serveur dans la fenêtre `netcat`. Noter au passage ce qui change si on entre l'URL `http://localhost:1234/x/y/z`.

2. Avec la même configuration, on peut se substituer au serveur et répondre à la main (dans le terminal d'où vous a été lancé `netcat`) à la requête du navigateur et lui renvoyer une ligne de texte de notre choix. Pour la réponse, on donnera comme première ligne la chaîne "HTTP/1.0 200 OK" (pour dire que tout va bien). Le mimetype pour le texte sera "text/plain":

```
HTTP/1.0 200 OK tout va bien
Content-type: text/plain
Connection: close
   <= ligne vide: la réponse suit
Ma réponse est: peut-être....
```

Voila, vous commencez à comprendre pourquoi la commande `netcat` a la réputation de *couteau suisse*... Elle pourra vous être très utile pour la mise au point de votre projet.

Principe de fonctionnement

Le serveur web que l'on va construire a une architecture assez spécifique. A la base, le serveur sait faire peu de choses, mais il sait gérer une liste de modules qui vont enrichir son comportement. Ces modules peuvent faire partie du serveur lui même, ou être chargés dynamiquement (voir plus loin).
Lorsqu'une URL est demandée au serveur celui-ci prend le premier module de la liste et lui demande de la traiter. Chaque module définit une fonction de traitement (un "*handler*") qui pourra être appelée avec les informations sur la requête courante. Si ce module sait traiter l'URL, son handler construit la réponse qu'il envoie au client et renvoie, par convention, la valeur 1. Si le handler ne sait pas traiter la requête, il renvoie 0 et le serveur délègue le traitement de cette requête au module suivant.
A la base, le serveur implémente un module dont le handler indique que l'URL demandée est introuvable quelque soit l'URL qui lui est passée (la célèbre erreur 404 de HTTP). Ensuite, on empile sur ce module un module qui sait traiter les fichiers. Son handler renvoie le fichier au client s'il arrive à le lire (le type MIME du fichier est trouvé dans une table globale du serveur en fonction du suffixe du fichier). Avec ces deux modules le serveur est donc capable de servir des fichiers, et indique bien une erreur si le fichier correspondant à l'URL n'existe pas sur le serveur.
Le serveur implémente aussi un module dont le handler sait traiter les répertoire. Ce module renvoie la liste des éléments du répertoires comme des liens sur lesquels on peut cliquer. Enfin, le dernier module pré-défini du serveur traite les répertoires qui contiennent un fichier dont le nom est `index.html`. Si l'URL désigne un répertoire et qu'un tel fichier existe, le handler demande son affichage, sinon il délègue le traitement au module inférieur (qui est le module de traitement des répertoires). Ce mode de fonctionnement est résumé dans la figure 1.

Chargement dynamique de code

Sous Unix, il est possible de charger dynamiquement (c'est à dire à l'éxecution) un fichier C qui a déjà été compilé au préalable. Cette technique va nous permettre d'enrichir le serveur web sans avoir à le recompiler! Sous GNU/Linux, le chargement dynamique nécessite l'utilisation de la bibliothèque `dl` et des fonctions `dlopen`, `dlsym`, ... Ces fonctions ne sont pas détaillées ici car on peut trouver un exemple d'utilisation complet de ces fonctions dans la page de manuel `dlopen(3)`. Attention aux options de compilation, qui sont spécifiques (mais décrites dans le manuel). Par ailleurs, un exemple vous est aussi fourni dans le fichier `simple-module.c` dans le répertoire `ext`[3].

[3]Ce répertoire contient les modules d'extension qui peuvent être chargés dynamiquement. Les module dynamiques compilés sont suffixés par `.so`.

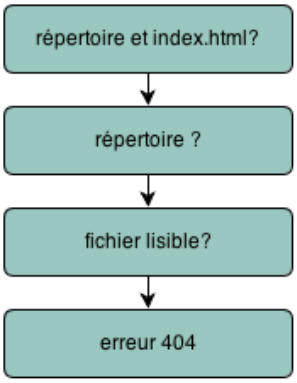

Figure 1: Organisation des modules pré-définis

Modules supplémentaires

Les quatre modules de la figure 1 sont pré-définis, et il correspondent aux modules classiques d'un serveur web. Polyweb, propose d'autres modules dont les fonctions sont brièvement décrites ici:

- `cgi.so`: ce module permet d'implementer la **Common Gateway Interface**, généralement abrégée CGI.
- `mytraceuri.so`: ce module permet d'afficher une trace pour chaque URI demandée au serveur.
- `prettify.so`: ce module permet d'embellir un fichier lorsqu'il est chargé par un navigateur Web. Cela peut servir pour *"fontifier"* des programmes ou afficher en HTML des fichiers Markdown.
- `ssi.so`: Ce module permet d'implémenter une version simplifiée du mécanisme de Server Side Includes.

Une documentation complète de ces modules est accessible dans `doc/doc-extensions.html`

Fichier de configuration

Lorsque Polyweb démarre, il cherche un fichier de configurations qui permet de décrire le comportement du serveur. Par défaut, ce fichier s'appelle `polywebrc` et il est cherché dans le répertoire courant (mais il peut aussi être passé en paramètre). Pour ce projet, le plus simple est de se placer dans le répertoire `src` pour lancer le serveur, puisque la configuration se fait de façon relative par rapport à ce répertoire. Le fichier de configuration permet de spécifier:

- le répertoire racine du serveur
- le port sur lequel le serveur tourne
- les types MIME reconnus
- le répertoire où se trouvent les extensions
- les extensions à charger (attention l'ordre est important: le dernier module chargé sera le premier à être exécuté).

Un exemple commenté de fichier de configuration est présent dans le répertoire `src`.

Déroulement du projet

Ce sujet et les fichiers qui l'accompagnent, doivent être considérés comme une **spécification**, c'est-à-dire que votre programme doit implémenter exactement ce qui demandé (en particulier l'ajout inconsidéré de fonctionnalités sera sanctionné). A titre d'exemple et pour fixer les idées, voici le nombre de lignes sources pour les parties qui ne vous sont pas livrées:

```
145 ext/cgi.c
 27 ext/mytraceuri.c
 91 ext/prettify.c
145 ext/ssi.c
105 src/config.c
245 src/handler.c
160 src/http.c
 71 src/mimetype.c
125 src/misc.c
```

soit moins de 1150 lignes (commentées). Il n'y a a pas de raison pour que vous ayez des choses fondamentalement différentes. Si c'est beaucoup plus gros, c'est que vous vous y êtes probablement mal pris. N'essayez pas bien sûr de concentrer votre code, au risque de le rendre illisible, pour tenir dans ces chiffres.
L'évaluation de votre travail prendra en compte principalement la qualité (et non pas la quantité) du code que vous nous rendrez.

Un module original

Le travail principal de ce projet consiste à réécrire les fichiers `.c` donnés plus haut. Comme cela laisse peu de place à la créativité, vous devrez aussi implémenter **un** module personnel original que vous présenterez au moment de votre soutenance. Pour ceux qui manquent d'idées, voici une liste de suggestions, que vous pourrez adapter:

- un module d'affichage d'images qui renvoie une image retaillée si celle-ci est trop grosse
- un module qui utilise des expression régulières pour changer à la volée le texte contenu dans un fichier avnat de l'afficher. Par exemple, remplacer les chaînes "Linux" par "GNU/Linux" pour les puristes.
- Un module qui fabrique des fichiers de *log* du genre de ceux du serveur Apache
- Un module qui ajoute systématiquement un `?markdown` au fichiers suffixés par `.md` et `?fontify` aux fichiers C, C++, Java, Python, ...
- ...

Soyez inventifs.