

M4102C - Programmation système avancée
Implémentation d'un serveur web

Michaël Hauspie

Jean-Marie Place

Damien Riquet

2 février 2015

Table des matières

1	Description générale	2
1.1	Pré-requis	2
1.2	Consignes générales	3
2	Mise en place de l'environnement de travail	4
2.1	Contrôle de version	4
2.1.1	Mise en place de git	4
2.2	Organisation du dépôt	4
2.2.1	Premiers fichiers	4
2.2.2	Organisation des sources	5
2.2.3	Fichiers Makefile	5
2.2.4	Test du Makefile	7
3	Un premier serveur TCP	8
3.1	Le réseau en C	8
3.1.1	La socket serveur	8
3.1.2	Les sockets clientes	10
3.2	Mise en pratique	12
3.2.1	Test de votre programme	12
4	Premières améliorations	13
4.1	Options des sockets	13
4.1.1	Mise en pratique	14
4.2	Gestion du signal SIGPIPE	14
4.2.1	Mise en pratique	15
4.3	Gestion de plusieurs connexions simultanées	15
4.3.1	Mise en pratique	15
4.4	Processus zombies	16
4.4.1	Mise en pratique	17
5	Un premier serveur web	18
5.1	Récupération des lignes de la requête	18
5.1.1	Mise en pratique	18
5.2	Requête GET	19
5.2.1	Manipulations préliminaires	19
5.2.2	Première implémentation	19
5.2.3	Servir un seul contenu et générer une erreur 404	19
6	Mise au propre	20
6.1	Lecture des lignes envoyées par le client	20
6.2	Analyse de la requête	20
6.3	Analyse des entêtes	21
6.4	Réponse au client	21

Chapitre 1

Description générale

L'objectif du module est de continuer à vous faire découvrir les primitives système offertes par les noyaux UNIX de type POSIX. Afin d'appliquer ces connaissances à un exemple concret, vous allez développer votre propre serveur web. A la fin du module, votre serveur sera capable :

1. d'accepter des connexions en TCP sur IPv4 et sur IPv6 ;
2. de servir les requêtes GET du protocole HTTP ;
3. de servir plusieurs clients simultanément ;
4. de servir du contenu statique (html, css, js, images, son et vidéo) ;
5. de recevoir des commandes de configuration d'un autre processus via des outils de communications entre processus offerts par un noyau UNIX.

Pour les plus efficaces d'entre vous, le serveur pourra être amélioré, par exemple :

1. en ajoutant la gestion des `VirtualHost` comme dans Apache ;
2. en ajoutant le support de pages dynamiques écrites en langage C ;
3. en ajoutant le support de pages dynamiques à l'aide de la norme CGI ;
4. en ajoutant le support des requêtes POST, PUT, DELETE, HEAD, OPTIONS du protocole HTTP ;
5. ...

1.1 Pré-requis

Les pré-requis de ce module sont bien évidemment ceux du module M3101 et plus particulièrement :

1. savoir écrire un programme en langage C organisé en plusieurs fichiers et le compiler à l'aide de l'outil `make` ;
2. savoir manipuler les tableaux, les chaînes de caractères C, les structures et les pointeurs ;
3. connaître la différence entre les données allouées statiquement, les données allouées sur la pile et les données allouées dynamiquement à l'aide de la fonction `malloc` ;
4. savoir manipuler un fichier (au sens général) à l'aide des primitives dites de « bas niveau » (`open`, `read`, `write`, `close`) et connaître la signification d'un descripteur de fichier ;
5. savoir utiliser les tubes ;
6. savoir effectuer des redirection à l'aide de la fonction `dup` ;
7. savoir gérer des processus à l'aide de `fork`, `wait`, `waitpid` et les fonctions `exec*`.

Vous aurez également besoin de connaissances issues d'autres modules :

1. savoir utiliser l'outil de contrôle de version `git` ;
2. connaître un minimum de notion réseau : adresse IP (v4, v6), protocole TCP, port ;

1.2 Consignes générales

Vous devrez vous organiser par groupe de deux¹. L'évaluation du module sera faite sur la base du serveur web que vous aurez développé. Vous devrez nous rendre votre dépôt git (cf. chapitre 2 pour la mise en place) sous la forme d'une archive ou éventuellement d'un lien vers une plateforme d'hébergement de dépôt git nous permettant de cloner le dépôt (serveur git de l'IUT, github...). Votre note sera évaluée en fonction de la version finale de votre serveur, mais également de l'historique de développement. Vous suivrez donc **scrupuleusement** les indications des différents chapitres concernant les branches ou les tags à créer sur votre dépôt.

1. Si votre groupe contient un nombre impair d'étudiants, un groupe de trois est autorisé.

Chapitre 2

Mise en place de l'environnement de travail

Le programme que nous allons écrire dans ce module est plus complexe que ce que vous avez développé jusqu'à présent en C. Nous allons donc organiser l'environnement de travail pour simplifier la tâche.

2.1 Contrôle de version

Que vous développiez seul ou à plusieurs, s'imposer l'utilisation d'un système de contrôle de version est une bonne habitude.

Si vous êtes seul, vous ne perdez rien de votre travail passé. Vous pouvez tester des fonctionnalités dans de nouvelles branches et les intégrer une fois terminées ou les abandonner en revenant facilement à une version stable.

Si vous collaborez avec d'autres développeurs, en plus des avantages précédents, les systèmes de contrôle de version permettent de fusionner facilement le travail de tout le monde.

2.1.1 Mise en place de git

Commencez par créer un répertoire pour votre projet et faites en un dépôt git à l'aide des commandes ¹ :

```
$ mkdir pawnee && cd pawnee && git init
```

2.2 Organisation du dépôt

2.2.1 Premiers fichiers

Créez et ajoutez les fichiers suivants à la racine de votre dépôt

1. **README** : vous indiquerez dans ce fichier au fur et à mesure du développement les instructions nécessaires à qui voudra compiler, configurer et utiliser votre serveur ;
2. **AUTHORS** : vous indiquerez dans ce fichier les noms, prénoms et adresse mail des auteurs (vous) ;
3. **LICENSE** : ce fichier devra contenir le texte relatif à la licence d'utilisation de votre logiciel. Sauf volonté contraire de votre part, vous utiliserez la licence GPL.

1. Vous devez trouver un autre nom pour votre serveur web, un peu d'imagination.

2.2.2 Organisation des sources

Créez le répertoire **webserver**. Ce répertoire contiendra les sources du serveur web. Par la suite, nous ajouterons d'autres utilitaires dont les fichiers source seront situés dans d'autres répertoires.

2.2.3 Fichiers Makefile

Pour l'instant, nous allons créer deux fichiers **Makefile**, l'un à la racine du dépôt, l'autre dans le répertoire **webserver**. Le premier fichier permettra, à terme, de construire tous les programmes liés au serveur web : le serveur ainsi que les programmes de gestion.

Listing 2.1 – Fichier Makefile racine

```
# Fichier Makefile racine

# Cette variable contient la liste des sous répertoires
# pour lesquels il faudra construire un programme.
# Pour l'instant, seul le serveur web est dans la liste.
FOLDERS=webserver

# Indique à make les règles qui ne correspondent pas à la création
# d'un fichier
# make lancera toujours la construction de cette règle si elle est
# demandée, même si un fichier/répertoire du nom de la cible existe
# On indique ici la règle all ainsi que les répertoires
.PHONY: all $(FOLDERS)

# La règle qui sera exécutée si on lance make sans argument
all: $(FOLDERS)

# Cette règle va lancer make dans le répertoire webserver
# option -C de make
webserver:
    make -C webserver
```

Pour le serveur web, nous allons utiliser un makefile plus compliqué, mais qui nous rendra la tâche plus simple par la suite. Il fera usage des règles génériques, de l'inclusion de fichiers et de la génération automatique de dépendances.

Listing 2.2 – Fichier Makefile du serveur web

```

# Ces variables servent à préciser le compilateur que l'on veut utiliser
# ainsi que les paramètres de compilation
CC=gcc
LD=gcc
CFLAGS=-Wall -W -Werror
LDFLAGS=

# Le nom de l'exécutable à fabriquer
EXE=pawnee

# Les variables HEADERS, CFILES et OBJS vont contenir respectivement
# la listes des fichiers .h, .c et le nom des fichiers .o à fabriquer
# On utilise la directive particulière $(wildcard ...) qui permet
# de construire automatiquement une liste de fichiers
HEADERS=$(wildcard *.h)
CFILES=$(wildcard *.c)
# Cette construction de variable veut dire: remplacer la chaine ".c" par
# ".o" dans la variable CFILES
# Ceci nous permet de construire la liste des fichiers .o à fabriquer
OBJS=$(CFILES:.c=.o)

# Même utilisation que précédemment. On précise les règles qui
# ne fabriquent pas de fichier du nom de leur cible
.PHONY: all clean mrproper

# La règle par défaut déclenche la fabrication de l'exécutable
# par dépendance
all: $(EXE)

# Cette règle permet de construire l'exécutable. Elle dépend des fichiers
# .o et effectue l'édition de lien. Rien de nouveau ici
$(EXE): $(OBJS)
    $(LD) $^ $(LDFLAGS) -o $@

# Cette règle permet de construire automatiquement les règles
# de compilation pour chacun des fichiers .c
# l'option -MM de gcc analyse un fichier .c et
# affiche sur sa sortie standard une règle compatible
# make pour le compiler.
# Ceci permet d'ajouter automatiquement les fichiers .h aux dépendances
# des fichiers .o à construire. Ainsi, en modifiant un fichier .h
# tous les fichiers .c qui incluent ce fichier sont recompilés
# Après votre première compilation, regardez le contenu du fichier
# makefile.dep généré afin de comprendre exactement de quoi il retourne.
makefile.dep: $(CFILES) $(HEADERS)
    $(CC) -MM $(CFILES) > $@

# Cette règle efface le fichier de dépendances et les fichiers .o
clean:
    $(RM) $(OBJS) makefile.dep

# Cette règle effectue la précédente et efface en plus l'exécutable
mrproper: clean
    $(RM) $(EXE)

# On inclue le fichier de dépendance qui va contenir les règles
# de construction des fichiers .o

```

```
# S'il n'existe pas, make invoque automatiquement la règle
# qui l'a pour cible
include makefile.dep
```

2.2.4 Test du Makefile

Créez le fichier `webserver/main.c` suivant et compilez le à l'aide du makefile que nous venons d'écrire.

Listing 2.3 – Premier fichier source

```
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
    /* Arnold Robbins in the LJ of February '95, describing RCS */
    if (argc > 1 && strcmp(argv[1], "-advice") == 0) {
        printf("Don't Panic!\n");
        return 42;
    }
    printf("Need an advice?\n");
    return 0;
}
```

Le fichier compile et s'exécute correctement ? Vous avez terminé la première étape. Après avoir ajouté tous les fichiers à votre dépôt et avoir « commité » toutes les modifications, créez un « tag » `mise_en_place` à l'aide de la commande :

```
$ git tag mise_en_place
```

Cette commande a pour effet de donner un nom explicite à la référence du commit correspondant à la fin de ce chapitre. Ces tags serviront à l'évaluation, soyez stricts sur leur nommage et leur utilisation. N'attendez pas la fin du module pour demander de l'aide si vous ne parvenez pas à les mettre en place correctement.

Chapitre 3

Un premier serveur TCP

Dans cette deuxième étape, nous allons développer un serveur TCP chargé de répéter à un client qui s’y connecte tout ce que ce dernier lui envoie.

3.1 Le réseau en C

L’utilisation du réseau en C passe par l’utilisation de « sockets ». Une socket est un fichier¹ qui permet de fournir une abstraction aux protocoles réseaux, et en particulier celui qui nous intéresse, TCP.

Dans le cas du développement d’un serveur TCP, nous allons distinguer deux types de sockets :

1. la socket dite « serveur » : c’est elle qui nous permettra d’accepter des connexions ;
2. les sockets dites « clientes » : ces sockets permettent au serveur de communiquer avec les clients, c’est à dire de leur envoyer des données. Du point de vue de votre programme, cette socket est vue, et utilisée, à l’aide d’un descripteur de fichier. Vous pourrez donc utiliser les primitives `read` et `write` pour, respectivement, recevoir des données du client et envoyer des données au client. Comme vous pouvez le voir, une socket s’utilise comme n’importe quel autre fichier (fichier régulier, tube, ...) et vous savez donc déjà comment l’utiliser... à condition d’avoir le descripteur.

3.1.1 La socket serveur

La socket serveur est celle qui permet à votre processus de demander au système d’exploitation (et en particulier à sa pile réseau) d’accepter les connexions en TCP sur un port donné et de le notifier. Pour cela, le système a besoin de connaître :

1. le protocole que le serveur veut utiliser : dans notre cas TCP ;
2. le port sur lequel il veut écouter.

Création de la socket La première étape se fait à la création de la socket. Cette création se fait à l’aide de la primitive :

```
int socket(int domain, int type, int protocol);
```

Cette fonction crée une socket et retourne un descripteur vers la socket créée ou -1 en cas d’erreur. Les trois paramètres de la fonction permettent de préciser :

- le domaine de la socket. On précise ici si l’on souhaite une socket utilisant IPv4 (`AF_INET`) ou IPv6 (`AF_INET6`) par exemple ;
- le type de la socket. C’est ici que l’on précisera si l’on veut utiliser TCP (`SOCK_STREAM`), UDP (`SOCK_DGRAM`) ou encore avoir un accès complet au protocole réseau (`SOCK_RAW`)²

1. rappelez vous, tout est fichier sous unix

2. Pour ce dernier cas, il faut être `!root!` sur la machine

- le protocole spécifique à utiliser pour le type et le domaine souhaité. Généralement, un seul protocole est disponible et ce paramètre peut-être 0. Ce sera le cas pour TCP sur IPv4 et IPv6, TCP est le seul protocole permettant de fournir une socket de type `SOCK_STREAM` sur IPv4 ou IPv6.

Créer une socket serveur revient donc à utiliser le code suivant :

Listing 3.1 – Création de la socket serveur en IPv4

```
int socket_serveur;

socket_serveur = socket(AF_INET, SOCK_STREAM, 0);
if (socket_serveur == -1)
{
    perror("socket_serveur");
    /* traitement de l'erreur */
}
/* Utilisation de la socket serveur */
```

Configuration de la socket La deuxième étape se fait à l'aide de la primitive `bind`. Elle permet d'attacher la socket à une interface réseau et à un port particulier. Attention, seule une socket à la fois peut être attachée à une même adresse et un même port. Le prototype de la primitive `bind` est le suivant :

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Ici, les choses se compliquent un peu. Le premier paramètre est le descripteur de la socket que l'on veut attacher. C'est la valeur retournée par la fonction `socket` utilisée précédemment.

Le deuxième et le troisième paramètre permettent de donner les paramètres d'attachement : la/les interfaces à utiliser et le port sur lequel le serveur va accepter des connexions. Le deuxième paramètre est donc un pointeur vers une structure qui contient ces informations et le deuxième est la taille de cette structure (en octets).

En réalité, le type `struct sockaddr` n'est pas utilisable tel quel. Chaque type de socket possède sa propre version de cette structure. Ceci est nécessaire car chaque type de socket n'utilise pas nécessairement les mêmes informations, comme, par exemple, des adresses de taille différentes. En effet, IPv4 utilise des adresses de 4 octets et IPv6 de 16 octets.

Pour IPv4, le type à utiliser est `struct sockaddr_in`. Cette structure contient les champs suivants :

Listing 3.2 – Structure `sockaddr_in` (extrait de la page de man `ip(7)`)

```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;    /* internet address */
};

/* Internet address. */
struct in_addr {
    uint32_t        s_addr;     /* address in network byte order */
};
```

On retrouve bien dans cette structure le port (`sin_port`) et l'adresse de l'interface sur laquelle on veut accepter des connexions (`sin_addr`). Le champ `sin_family` doit être affecté à la valeur `AF_INET`.

La page de manuel indique que le champ `sin_port` doit être donné dans le sens « réseau » des octets (*network byte order*). Un port TCP est sur deux octets et les protocoles réseaux utilisent

généralement³ un encodage de type « *big endian* ». Il faudra donc affecter ce champ avec un entier au format big endian. Heureusement pour nous, la librairie système nous fournit une fonction permettant de faire la conversion, quelque soit l'architecture de la machine qui exécute le code. Cette fonction est `htons` : *Host to Network Short*.

Enfin, le champ `sin_addr` permet de préciser l'adresse de l'interface réseau sur laquelle on veut accepter des connexions. Si l'on souhaite accepter des connexions depuis toutes les interfaces de la machine (ce qui est souvent le cas), on peut utiliser la valeur particulière `INADDR_ANY`.

Le listing suivant donne un exemple d'utilisation de la fonction `bind` pour IPv4. Le descripteur `socket_serveur` aura été créé au préalable par le code du listing précédent.

Listing 3.3 – Attachement de la socket serveur sur toutes les interfaces

```
struct sockaddr_in saddr;
saddr.sin_family = AF_INET; /* Socket ipv4 */
saddr.sin_port = htons(8080); /* Port d'écoute */
saddr.sin_addr.s_addr = INADDR_ANY; /* écoute sur toutes les interfaces */

if (bind(socket_serveur, (struct sockaddr *)&saddr, sizeof(saddr)) == -1)
{
    perror("bind socket_serveur");
    /* traitement de l'erreur */
}
```

Vous remarquerez que nous devons effectuer un « cast » de l'adresse de la structure en `(struct sockaddr *)`. En effet, le C n'est pas un langage objet, la notion de « classe de base » n'existe pas et est simulée par l'emploi du type `struct sockaddr`. Le cast doit donc forcément être explicite.

Démarrer l'attente de connexions La dernière étape de préparation de la socket serveur consiste à signaler au système d'exploitation que le processus souhaite accepter des connexions (écouter) sur la socket que nous venons de configurer. Cette dernière étape se fait avec la fonction :

```
int listen(int sockfd, int backlog);
```

Le premier paramètre est toujours le descripteur de la socket serveur. Le deuxième paramètre est la taille de la file d'attente de connexions. Ceci indique au système le nombre de connexions qu'il peut mettre en attente d'acceptation de la connexion par le serveur. Cela ne correspond pas au nombre de connexions simultanées que peut gérer le serveur, mais uniquement au nombre de connexions non encore acceptées que le système accepte de mettre en attente avant de les rejeter. Le choix de cette valeur va dépendre de la charge du serveur, de sa capacité à gérer plusieurs clients simultanément ou d'autres paramètres. Pour notre utilisation, nous pouvons le fixer arbitrairement à 10.

Listing 3.4 – Lancer l'attente de connexion

```
if (listen(socket_serveur, 10) == -1)
{
    perror("listen socket_serveur");
    /* traitement d'erreur */
}
```

3.1.2 Les sockets clientes

Après avoir créé la socket serveur, le programme peut accepter des connexions. C'est en acceptant des connexions qu'il obtient des descripteurs vers les sockets des clients. Pour accepter une connexion, le serveur utilise la primitive :

3. C'est le cas pour TCP en particulier

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Le premier paramètre est toujours notre socket serveur. Le deuxième et le troisième paramètre permettent d'obtenir des informations sur le client qui se connecte. Ce sont des pointeurs vers une structure et un entier qui seront remplies par la fonction **accept** pour donner l'adresse IP du client par exemple. Dans un premier temps, nous pourrions passer **NULL** à la fonction pour ces deux paramètres.

La fonction retourne un descripteur qui correspond à la socket permettant de dialoguer avec le client ou -1 en cas d'erreur. Cette fonction est **bloquante**, c'est à dire que le programme serveur restera bloqué sur cette fonction tant qu'aucun client ne tentera de se connecter au serveur.

Listing 3.5 – Accepter une connexion

```
int socket_client;
socket_client = accept(socket_serveur, NULL, NULL);
if (socket_client == -1)
{
    perror("accept");
    /* traitement d'erreur */
}
/* On peut maintenant dialoguer avec le client */
const char *message_bienvenue = "Bonjour, bienvenue sur mon serveur\n";
write(socket_client, message_bienvenue, strlen(message_bienvenue));
```

3.2 Mise en pratique

Créez les fichiers `socket.c` et `socket.h`. Le fichier `socket.h` devra être le suivant :

Listing 3.6 – Fichier `socket.h`

```
#ifndef __SOCKET_H__
#define __SOCKET_H__
/** Crée une socket serveur qui écoute sur toute les interfaces IPv4
    de la machine sur le port passé en paramètre. La socket retournée
    doit pouvoir être utilisée directement par un appel à accept.

    La fonction retourne -1 en cas d'erreur ou le descripteur de la
    socket créée. */

int creer_serveur(int port);
#endif
```

Le fichier `socket.c` doit implémenter la fonction `creer_serveur` à l'aide des primitives `socket`, `bind` et `listen`.

Modifiez ensuite votre fichier `main.c` pour qu'il crée un serveur et accepte indéfiniment des connexions sur le port 8080.

À chaque connexion, le serveur envoie un **long (au moins 10 lignes)** message de bienvenue (soyez créatifs!) au client puis, tant que le client n'est pas déconnecté, lui répète les messages envoyés par le client.

3.2.1 Test de votre programme

Pour tester votre programme, vous pouvez utiliser les commandes `nc` ou `telnet`. Répondez aux questions suivantes dans un fichier `reponses.txt` qui devra être à la racine de votre dépôt :

1. Quittez votre serveur (`Ctrl+C`) et relancez le. Que se passe t'il ?
2. Ajouter un petit délai avant l'envoi du message de bienvenue (1 seconde). Puis, exécutez la commande suivante : `nc -z 127.0.0.1 8080`. Que se passe t'il ?
3. Exécutez la commande suivante : `telnet ::1 8080`. Que se passe t'il ? Pourquoi ?
4. Lancez deux clients simultanément. Que se passe t'il ? Pourquoi ?

Quand votre programme fonctionne, assurez vous d'avoir bien commité les dernières modifications et créez le tag `serveur_tcp_simple`. Cela marque la fin de ce chapitre.

Chapitre 4

Premières améliorations

En répondant aux questions du chapitre précédent, vous avez du identifier plusieurs problèmes :

1. une fois votre serveur arrêté, vous devez patienter quelques temps avant de pouvoir le redémarrer (l'appel à `bind` échoue).
2. votre serveur « plante » s'il envoie des données à un client déconnecté ;
3. votre serveur n'accepte pas de connexion sur une adresse IPv6 ;
4. votre serveur ne peut gérer qu'une connexion à la fois ;

Nous allons maintenant régler ces problèmes un par un, à l'exception de l'IPv6 qui sera traité plus tard.

4.1 Options des sockets

Quand vous arrêtez votre serveur par un `Ctrl+C` alors que des clients étaient connectés à celui-ci, une nouvelle tentative de démarrage se solde par un échec de l'appel à la fonction `bind` avec le message : « Address already in use ». La raison à ce message est que, par défaut, même quand le processus qui a créé la socket est terminé, le système maintient la socket au cas où des paquets se seraient perdus et devraient être retransmis.

Ceci assure la fiabilité à TCP si un paquet s'est perdu après que le processus est arrêté, mais empêche de redémarrer un serveur pendant une durée de 30 à 120 secondes en fonction de la configuration des piles TCP/IP. Or, quand on développe un serveur on est souvent amené à vouloir le redémarrer (par exemple, changer son fichier de configuration et le redémarrer). Il existe une option permettant de modifier ce comportement de façon à ce qu'un processus qui demande de lier une socket sur une adresse encore utilisée puisse le faire, à la condition que le processus qui a créé la première socket soit terminé.

La fonction qui permet de modifier des options relatives au comportement des sockets est :

```
int setsockopt(int socket, int level, int option_name,
               const void *option_value, socklen_t option_len);
```

Cette fonction prend 5 paramètres :

1. le descripteur de la socket que l'on souhaite modifier ;
2. le niveau auquel s'applique l'option. Elle peut s'appliquer au niveau socket `SOL_SOCKET` ou à d'autre niveau de protocole comme `IPPROTO_TCP` par exemple. Dans le cas qui nous intéresse, c'est une option au niveau socket et on passera donc `SOL_SOCKET` comme valeur pour le paramètre `level` ;
3. le nom de l'option. L'option qui nous intéresse est `SO_REUSEADDR` qui permet à un processus de faire un `bind` sur une même adresse/port (sauf si un processus est déjà en `listen` dessus) ;

4. la valeur de l'option. Comme la fonction `setsockopt` est générique (plusieurs options possibles), ce paramètre est un `void *`. Dans la plupart des cas, on utilisera un entier (et donc un pointeur vers un entier valide);
5. la taille de l'option. Cette est également nécessaire pour apporter de la généricité à la fonction. Comme dans notre cas, on va passer un pointeur de type `int*` pour le paramètre `option_value`, on utilisera `sizeof(int)` pour ce paramètre.

Nous voulons donc activer l'option `SO_REUSEADDR`, il faudra donc passer comme valeur de l'option un pointeur vers un entier qui contient un nombre différent de 0.

L'appel à la fonction `setsockopt` doit être fait **avant** l'appel à `bind`.

Listing 4.1 – Activation de l'option `SO_REUSEADDR`

```
int optval = 1;

if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(int)) == -1)
    perror("Can not set SO_REUSEADDR option");
```

4.1.1 Mise en pratique

Modifiez votre fonction `creer_serveur` pour activer l'option `SO_REUSEADDR` sur la socket que vous créez.

Essayez à nouveau de lancer votre serveur, immédiatement après l'avoir terminé par `Ctrl+C`. Il doit normalement redémarrer sans erreur.

Une fois la modification effectuée, testée et commitée, créez un tag `REUSEADDR` sur votre dépôt git.

4.2 Gestion du signal `SIGPIPE`

Lorsque votre programme envoie un message de bienvenue suffisamment long et que le client se déconnecte pendant la transmission de ce message, votre serveur s'arrête. L'explication peut se trouver dans la page de manuel de la fonction `write` dont voici un extrait :

```
[...]
RETURN VALUE
[...]
EPIPE  fd is connected to a pipe or socket whose reading end
       is closed. When this happens the writing process will
       also receive a SIGPIPE signal. (Thus, the write return
       value is seen only if the program catches, blocks or
       ignores this signal.)
```

En clair, lorsque votre programme effectue un appel à la fonction `write` alors que le fichier correspondant au descripteur fermé, le processus reçoit un signal.

Un signal est un événement envoyé par le système à un processus. Un processus qui reçoit un signal peut réagir de différentes façon :

1. l'ignorer ;
2. s'arrêter ;
3. interrompre le programme en cours, exécuter un traitement spécifique et revenir au programme.

Par défaut, la plupart des signaux entraînent l'arrêt du processus. C'est la raison de l'arrêt de votre serveur. Comme on ne peut pas savoir à l'avance quand un client va se déconnecter, il est impératif d'ignorer ce signal.

Deux fonctions permettent de contrôler le comportement d'un processus à la réception d'un signal : `signal` et `sigaction`. Pour les réglage fin de comportement (et en particulier l'exécution d'un traitement spécifique), la fonction à utiliser est `sigaction`¹

Néanmoins, dans notre cas, comme on souhaite uniquement ignorer le signal, l'utilisation de `signal` est possible. Le prototype de la fonction `signal` est le suivant :

```
sighandler_t signal(int signum, sighandler_t handler);
```

Le premier paramètre correspond au numéro de signal (`SIGPIPE` pour nous) et le deuxième au comportement que le processus doit adopter à la réception du signal : `SIG_IGN` pour ignorer, `SIG_DFL` pour restaurer le comportement par défaut.

Listing 4.2 – Ignorer `SIGPIPE`

```
if (signal(SIGPIPE, SIG_IGN) == SIG_ERR)
{
    perror("signal");
}
```

4.2.1 Mise en pratique

Ajoutez une fonction

```
void initialiser_signaux(void);
```

qui s'assure que votre processus ignorera à présent le signal `SIGPIPE`.

Testez à nouveau l'utilisation de `nc` avec l'option `-z` pour confirmer que votre serveur ne s'arrête plus.

Une fois la modification effectuée, testée et commitée, créez un tag `SIGPIPE` sur votre dépôt git.

4.3 Gestion de plusieurs connexions simultanées

Pour gérer plusieurs connexion simultanément, le serveur doit pouvoir, en parallèle, :

1. attendre des connexions ;
2. lire les données envoyées par les clients connectés.

Le véritable problème réside dans le fait que les appels à `accept` et à `read` sont **bloquants**. Pour résoudre ce problème, une solution serait de créer un nouveau processus à chaque connexion d'un nouveau client.

Nous allons mettre en œuvre cette solution et voir si elle peut convenir.

4.3.1 Mise en pratique

Modifiez votre serveur pour qu'il crée un nouveau processus à chaque nouvelle connexion. Le nouveau processus devra être créé après l'appel à `accept` et ce dernier effectuera la boucle de traitement du client. Le processus père devra fermer le descripteur de la socket du client après avoir créé le processus fils.

Testez maintenant votre serveur avec plusieurs connexions simultanées, faites un commit des modifications et créez le tag `fork1`.

1. cf page de manuel de `signal` : « The `sigaction` function provides a more comprehensive and reliable mechanism for controlling signals; new applications should use `sigaction()` rather than `signal()`. »

4.4 Processus zombies

Après avoir testé quelques connexions, affichez la liste des processus. Vous devez normalement voir apparaître autant de processus zombies que de connexions acceptées puis terminées.

Pour faire disparaître ces processus zombies, il faudra faire un appel à `wait` dans le processus père. Cependant, il faudrait qu'il le fasse entre chaque connexion. Une solution serait d'utiliser `waitpid` avec l'option `WNOHANG` mais on peut faire mieux.

A la mort d'un de ces fils, le processus père reçoit un signal `SIGCHLD`. Par défaut, le comportement du processus est d'ignorer ce signal. Il est possible de changer ce comportement par défaut pour, cette fois, effectuer un traitement particulier. Dans ce traitement particulier, on pourra effectuer l'appel à `waitpid`.

Pour cela, on utilise la fonction `sigaction`. Cette fonction permet d'attribuer un comportement spécifique à la réception d'un signal. Le prototype de la fonction est le suivant :

```
int sigaction(int sig, const struct sigaction *act,
              struct sigaction *oact);
```

Le premier paramètre est le signal pour lequel on veut effectuer un traitement particulier. Le deuxième paramètre est un pointeur vers une structure qui décrit le traitement. Le dernier paramètre permet de sauvegarder le comportement précédent dans le but de le restaurer plus tard si nécessaire.

Le comportement est décrit dans une structure qui contient trois champs² :

- `sa_handler` : Ce champ est un pointeur de fonction. Un pointeur de fonction est simplement un pointeur dans lequel on peut stocker l'adresse d'une fonction. Le pointeur peut alors être utilisé comme un appel de fonction ;
- `sa_mask` : un ensemble de signaux qui doivent être bloqués pendant le traitement du signal. Dans notre cas, nous utiliserons un ensemble vide qui peut être construit à l'aide de la fonction `sigemptyset` ;
- `sa_flags` : qui permet de régler des options relative au traitement du signal. On utilisera l'option `SA_RESTART` pour redémarrer automatiquement les fonctions qui auraient du être interrompue par la réception d'un signal (comme la fonction `accept`).

Le code suivant permet d'installer un traitement (d'appeler une fonction) à la réception du signal `SIGCHLD`.

Listing 4.3 – Utilisation de `sigaction`

```
void traitement_signal(int sig)
{
    printf("Signal %d reçu\n", sig);
}

void initialiser_signaux(void)
{
    struct sigaction sa;

    sa.sa_handler = traitement_signal;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    if (sigaction(SIGCHLD, &sa, NULL) == -1)
    {
        perror("sigaction(SIGCHLD)");
    }
}
```

2. En réalité quatre, mais le détail du quatrième champ sort du cadre de ce document. Vous pouvez consulter la page de manuel pour en savoir plus.

4.4.1 Mise en pratique

Modifiez votre programme pour qu'il effectue les appels nécessaires à la fonction `waitpid` à la réception du signal `SIGCHLD`.

Vérifiez que vous n'avez plus de processus zombies après les fermetures de connexions.

Après le commit de vos modifications, créez le tag `SIGCHLD`.

Chapitre 5

Un premier serveur web

Nous allons maintenant commencer à implémenter le protocole HTTP. Ce protocole est décrit de manière exhaustive dans les RFC 7230 à 7237. Pour des raisons de simplicité et de temps, nous n'allons gérer qu'une partie de ce protocole. Dans un premier temps, nous allons implémenter la réponse à une requête `GET` en ignorant les éventuelles entêtes envoyées par le client (le navigateur).

5.1 Récupération des lignes de la requête

Le protocole HTTP est essentiellement basé sur une séquence de lignes, séparées par le caractère `'\r'`¹ suivi du caractère `'\n'`² (notés `CRLF` dans la RFC).

Bien que le protocole TCP offre à l'application une vue en flux des données, il est tout de même basé sur un protocole paquet (IP). De ce fait, il se peut qu'une ligne se retrouve « coupée » et soit récupérée par l'application par deux appels successifs à la primitive `read` (voire plus pour une ligne très longue). Avant de traiter une ligne, votre serveur doit donc s'assurer qu'elle est complète. Cette opération n'est pas très difficile à mettre en place, mais la librairie C nous offre une primitive qui le fait déjà : `fgets`.

La fonction `fgets`, ainsi que toutes les fonctions `f*` (`fread`, `fwrite`, `fprintf`...) n'utilise pas directement un descripteur mais une structure de type `FILE *`. Cette structure contient le descripteur correspondant au fichier, mais également des tampons d'entrée et de sortie.

Pour obtenir un pointeur vers une structure `FILE *` à partir d'un descripteur de fichier, on utilise la fonction `fdopen`.

```
FILE *fdopen(int fd, const char *mode);
```

Comme le descripteur que l'on va utiliser correspond à la socket cliente, le mode devra être `"w+"'`³.

Le pointeur retourné pourra donc être utilisé avec les fonctions `fgets`, `fread`, `fwrite` et même `fprintf` pour un envoi de chaîne formatée au client.

5.1.1 Mise en pratique

Modifiez votre serveur pour utiliser les primitives `fgets` et `fprintf` pour, respectivement, lire les données envoyées par le client et lui retransmettre, en faisant précéder ces dernières par la chaîne `"<Pawnee> "`, dans laquelle vous remplacerez `Pawnee` par le nom de votre serveur.

Après modification et commit, créez le tag `fdopen` avant de passer à la suite.

1. CR : *carriage return*

2. LF : *line feed*

3. n'oubliez pas de lire la page de manuel.

5.2 Requête GET

5.2.1 Manipulations préliminaires

1. Modifiez votre programme pour qu'il ne transmette plus rien au client et qu'il affiche sur sa sortie standard les données envoyées par le client ;
2. Lancez la commande `$ curl http://localhost:8080/`
3. Quel est le message envoyé par le client web ?
4. À l'aide de la RFC, trouvez le *nom* donné aux trois constituant de la première ligne de la requête envoyée par le client.

5.2.2 Première implémentation

Le premier message envoyé par un navigateur est donc de la forme :

```
GET / HTTP/1.1
User-Agent: curl/7.40.0
Host: localhost:8080
Accept: */*
```

Sur ces quatre lignes, seules deux sont obligatoires pour le protocole HTTP dans sa version 1.1 : la première (la requête à proprement parlé) et la troisième.

Nous allons modifier le serveur pour qu'il réponde correctement à une requête simple.

1. Modifiez votre serveur pour qu'il analyse la première ligne que lui envoie le client. Il doit vérifier que cette ligne commence par le mot **GET** et qu'elle contient exactement 3 mots ;
2. vérifiez que le troisième mot est bien de la forme **HTTP/M.m** ou **M** et **m** sont deux chiffres. Vérifiez que **M** vaut 1 et que **m** vaut 0 ou 1 ;
3. ignorez les lignes envoyées par le client tant qu'il ne s'agit pas d'une ligne vide (et donc que la chaîne de la requête soit `"\r\n"` ou `"\n"`) ;
4. si la requête ne respecte pas un des critères précédents, transmettez la réponse suivante (chaque ligne doit se terminer par `"\r\n"`) :

```
HTTP/1.1 400 Bad Request
Connection: close
Content-Length: 17
```

```
400 Bad request
```

Vous pouvez tester cette partie de votre code en utilisant `nc` ou `telnet`.

5. si la requête respecte les critères, transmettez votre message de bienvenue. Vous devrez adapter le message précédent en changeant le code HTTP de `"HTTP/1.1 400 Bad Request"` à `"HTTP/1.1 200 OK"` et la taille du contenu.

Testez votre programme avec `curl` et un navigateur graphique. Après avoir effectué un commit de vos modifications, créez le tag `simple_get`.

5.2.3 Servir un seul contenu et générer une erreur 404

1. Modifiez votre serveur pour qu'il ne réponde qu'à une requête sur l'URL `'/'`. Dans ce cas, il doit retourner votre message de bienvenue comme dans la section précédente.
2. Si l'URL est différente, votre serveur doit retourner une erreur 404. Inspirez vous de l'erreur 400 précédente pour générer correctement l'erreur 404.

Après avoir testé et commité, créez le tag `url_or_404`.

Chapitre 6

Mise au propre

Il est fort probable que pour répondre aux demandes du chapitre précédent vous ayez tout implémenté quasiment dans une fonction unique (voire même votre main...). Afin de vous faciliter la tâche, voici quelques indications pour nettoyer un peu tout ça.

Si vous avez déjà fait des choses similaires, bravo, vous pouvez directement passer à la suite après avoir bien lu ce chapitre et vérifié que votre code est organisé aussi clairement que ce qui vous est proposé dans ce chapitre. Attention, ce chapitre donne des précisions sur le protocole HTTP (tirées directement de la RFC) qu'il est important d'avoir comprises.

6.1 Lecture des lignes envoyées par le client

Afin de faciliter la gestion d'erreur, nous allons écrire une fonction qui lit les données en provenance du client et quitte le processus si le client se déconnecte ou si une erreur survient.

La fonction à écrire est la suivante :

```
char *fgets_or_exit(char *buffer, int size, FILE *stream)
```

La fonction reprends le prototype de la fonction `fgets`. Elle fera donc appel à `fgets`. Si `fgets` détecte que le client est déconnecté, votre fonction devra quitter le processus par un appel à `exit`. Vous vous servirez désormais exclusivement de cette fonction pour lire les données du client. Vous n'aurez donc plus besoin de tester son retour, elle s'occupera de quitter le processus si nécessaire.

6.2 Analyse de la requête

Écrivez une fonction qui permet d'analyser la première ligne de la requête. D'après la RFC, cette première ligne est de la forme :

```
request-line = method SP request-target SP HTTP-version CRLF
```

- `method` doit être `GET` dans notre cas;
- `request-target` correspond à l'url;
- `SP` représente **un seul et unique** espace;
- `HTTP-version` correspond dans notre cas à `HTTP/1.0` ou `HTTP/1.1`;
- `CRLF` est la succession du caractère `'\r'` et du caractère `'\n'` ou uniquement le caractère `'\n'`.

On propose d'utiliser le prototype suivant pour la fonction :

```
int parse_http_request(const char *request_line, http_request *request);
```

Le premier paramètre est la première ligne envoyée par le client. Le deuxième paramètre est un pointeur vers une structure que nous allons décrire par la suite et qui contiendra les informations

relative à la requête. Elle sera donc modifiée par la fonction. La fonction doit retourner 0 si la requête est invalide et 1 si le format de la requête est correct.

La structure `http_request` est définie ainsi :

```
enum http_method {
    HTTP_GET,
    HTTP_UNSUPPORTED,
};

typedef struct
{
    enum http_method method;
    int major_version;
    int minor_version;
    char *url;
} http_request;
```

Vous pouvez remarquer que la structure utilise un type énuméré pour stocker un identifiant de la méthode employé. Ainsi, pour une évolution future du serveur, il suffira de rajouter des valeurs possibles¹ dans le type énuméré et de modifier la fonction d'analyse.

6.3 Analyse des entêtes

Une nouvelle lecture de la RFC nous apprend qu'un message HTTP doit être de la forme :

Listing 6.1 – Message HTTP

```
HTTP-message = start-line
               *( header-field CRLF )
               CRLF
               [ message-body ]

avec ,

start-line    = request-line / status-line
```

Nous nous sommes déjà occupé de la `start-line`² avec la fonction précédente. Il s'agit maintenant de s'occuper de la succession de `header-field`, les entêtes.

Dans un premier temps, notre analyse des entêtes vont simplement consister à les ignorer.

Écrivez une fonction qui lit les lignes envoyées par le client tant que celles ci ne sont pas `"\r\n"` ou `"\n"`. Le prototype d'une telle fonction peut être :

```
void skip_headers(FILE *client);
```

Au retour de cette fonction, vous êtes prêts à envoyer la réponse attendue par le client.

6.4 Réponse au client

La réponse au client étant un message HTTP, elle doit répondre au format donné par le listing 6.1. La différence avec la requête du client réside dans la première ligne, qui n'est plus une `request-line` mais une `status-line`. C'est la première ligne qui indiquera si le serveur répond avec succès à la requête. Elle doit être de la forme :

```
status-line = HTTP-version SP status-code SP reason-phrase CRLF
```

1. Pour les autres méthodes HTTP comme POST par exemple

2. La `start-line` est une `request-line` lorsque que le message vient du client et une `status-line` quand le message est émis par le serveur

- HTTP-version doit être du même format que précédemment ;
- status-code est un code à 3 chiffres indiquant comment le serveur réagit à la requête. Voici quelques codes HTTP avec la reason-phrase conseillée par la RFC :
 - 200 : OK
 - 400 : Bad Request
 - 403 : Forbidden
 - 404 : Not Found
 - 405 : Method Not Allowed
 - 500 : Internal Server Error
 - 501 : Not Implemented
 - 505 : HTTP Version Not Supported

Écrivez la fonction

```
void send_status(FILE *client, int code, const char *reason_phrase);
```

qui transmet au client une ligne de status correspondant aux paramètres passés à la fonction.

Pour compléter la réponse, après la ligne de status, le serveur doit envoyer un certain nombre d'entêtes. En particulier le serveur doit indiquer la taille du contenu qu'il transmet à l'aide de l'entête `Content-Length`. La valeur de l'entête doit être le nombre d'octet du message qui suivra (`message-body` dans le listing 6.1).

Écrivez la fonction

```
void send_response(FILE *client, int code, const char *reason_phrase,  
                  const char *message_body);
```

Cette fonction doit transmettre une réponse complète au client et doit donc utiliser la fonction `send_status` pour transmettre la première ligne.

Les réponses du serveur du chapitre précédent peuvent alors être envoyées à l'aide des deux appels suivants (en supposant que `motd` pointe vers une chaîne de caractères contenant le message de bienvenue) :

```
if (bad_request)  
    send_response(client, 400, "Bad Request", "Bad request\r\n");  
else if (request.method == HTTP_UNSUPPORTED)  
    send_response(client, 405, "Method Not Allowed", "Method Not Allowed\r\n");  
else if (strcmp(request.url, "/") == 0)  
    send_response(client, 200, "OK", motd);  
else  
    send_response(client, 404, "Not Found", "Not Found\r\n");
```

Une fois le nettoyage terminé et commité, créez le tag `url_or_404_clean`.