Web Scrapper

Nathanael Demacon
3ème année AL - Promo 2019
École Supérieur du Génie Informatique
Paris, France
nathanael.dmc@outlook.fr

Océane Denis
3ème année AL - Promo 2019
École Supérieur du Génie Informatique
Paris, France
oceane.marie.d@gmail.com

***Abstract*—Ce projet a pour but de créer un *crawler* performant consommant le moins de mémoire vive possible. Il a été initié par l'École Supérieur du Génie Informatique en tant qu'épreuve dans le but de nous évaluer sur nos compétences du langage C.**

I. Introduction

Dans l'optique de résoudre les problèmes techniques et algorithmique de ce projet, il est nécéssaire d'adopter une vision purement théorique. Ce document a pour but de présenter les diverses problèmatiques recontrés ainsi que leur solution trouvée durant nos recherches. Ce document ne présentera en aucun cas la manière dont le programme sera utilisé d'un point de vu utilisateur (voir document d'utilisation).

Pour appel, ceci est un projet d'études et ne constitue donc pas un projet de recherche en cas réel et n'est qu'un simple exercice de compréhension du langage C.

II. Fichier de configuration

Coeur du logiciel, le fichier de configuration va permettre de configurer le comportement de celui-ci en listant des *Actions* et des *Tâches*.

Nous allons traiter ce fichier de configuration comme un langage de programmation. La syntaxe étant déjà établie par les coordinateurs du projet.

Pour celà nous devons énumérer les différentes étapes de conception d'un langage de programmation *basique*:

1) Analyse syntaxique
2) *Parsing*

A. Analyse syntaxique

Cette première étape consiste en la découpe des différentes composantes du langage en *tokens*. On l'appelle aussi la tokénisation.

Pour se faire nous devons définir chaque caractère ou chaîne de caractères à un identificateur précis. Par exemple dans le format du fichier de configuration, la chaîne de caractères -¿ équivaut à un séparateur entre le nom de l'option et sa valeur. Le caractère ¯ quant à lui signifie une déclaration d'*Action*.

B. Parsing

Aussi appellé *Analyse sémantique*, le parsing est l'étape qui va transformer notre liste de *Tokens* en un format utilisable, fréquemment un arbre syntaxique, *Abstract Syntax Tree* en anglais.

Dans notre cas nous n'allons pas utiliser d'arbre mais générer la structure *Config* (voir partie *Structures de données*).

III. Récupération des données HTTP

Pour récupérer les informations de chaque sites, et donc pouvoir récupérer les *URLs* en référence, il nous faut envoyer une requête via le protocol HTTP. Nous utiliserons la bibliothèque *CURL*, communément appellée *libcurl*.

Une fois les données récupérées, une recherche des *URLs* sera effectuée sur le *buffer*. Ces *URLs* seront utilisées pour parcourir récursivement les autres sites.

IV. Multi-threading

Rapidité, point fort de notre application, fonctionne sur plusieurs *threads*. Nous utiliserons la biliothèque native *pthread* (soit *POSIX Thread*).

Étant donné l'implication du nombre de requêtes *I/O* envers les fichiers recevants chaque pages téléchargés, il est important de prévenir la concurrence envers différents threads sur un même fichier. Pour éviter un tel problème, l'utilisation des *mutex* (verrou virtuel) sera omniprésent.

A. Scheduler

Pour faire fonctionner les *Tâches* en parallèle et de lancer leurs *Actions* à un temps donné, il nous faudra créer différents threads.

Dans le cas actuel, un thread par *Tâche* pour contrôler la temporisation des *Actions* a été mise en place. Ensuite un thread par *Action* dans l'intervale donnée.

V. Structures de données

A. Vecteur

Comme définits dans le langage C, la gestion de tableaux est fastidieuse, notamment dûe à l'allocation *manuelle* de mémoire. Une solution pour palier à ce problème serait d'avoir une structure qui s'agrandirait sans intervention du dévelopeur par rapport à la mémoire. Hors, c'est exactement pour celà qu'a été créé la structure *Vector* (fr. *Vecteur*).

```
typedef struct vector_t Vector;
struct vector_t
{
   size_t length;
   size_t capacity;

   void **values;
};
```

Étant donné le manque de programmation orienté object dans le langage C, nous allons devoir passer par des fonctions pour contrôler cette structure. Toute les fonctions relatives aux Vecteurs sont nommés commençants par *vector_*.

B. Map / Dictionnaire

Les maps, aussi appelés dictionnaires, sont prévues pour recevoir n'importe quelle données qui sera symbolisée par une clé. La clé faisant office d'identification pour facilement récupérer la donnée stockée.

```
typedef struct map_node_t MapNode;
struct map_node_t
{
   char *key;
   void *value;

   MapNode *prev;
   MapNode *next;
};

typedef struct map_t Map;
struct map_t
{
   MapNode *first;
};
```

La structure est basée sur le principe de *liste doublement chainée*, ce qui en fait un type facilement parcourable et modifiable. Cette méthode permet aussi de limiter l'utilisation de la mémoire (contrairement à la structure *Vecteur*).

C. Configuration

Pour stocker le fichier de configuration nous devons d'abbord énumérer ses différentes parties ainsi que leur composantes:

- Actions
  - Nom (chaîne) [obligatoire]
  - URL (chaîne) [obligatoire]
  - Options [optionel]
    - max-depth (entier)
    - versioning (chaîne)
    - types (liste)
    - max-buffer (entier)
- Tâches
  - Nom (chaîne)
  - secondes (entier)
  - minutes (entier)
  - heures (entier)
  - actions (liste)

La structure de donnée *Map* convient parfaitement à la propriétée *Options* ainsi qu'au stockage des *Actions* et des *Tâches*, ces derniers identifiés par un nom (une clé et une valeur). Pour la propriétée *Options*, il faudra aussi séparer les chaînes de caractère, les entiers et les listes. Pour se faire nous allons utiliser une *union* ainsi qu'un *enum*.

```
enum config_option_type_t
{
   CONFIG_OPTION_STRING,
   CONFIG_OPTION_ARRAY,
};
typedef enum config_option_type_t
    ConfigOptionType;

typedef union config_option_value_t
    ConfigOptionValue;
union config_option_value_t {
   char *str;
   Vector *arr;
};

typedef struct config_option_t
    ConfigOption;
struct config_option_t
{
   ConfigOptionType type;
   ConfigOptionValue value;
};

typedef struct config_action_t
    ConfigAction;
struct config_action_t
{
   char *url;

   Map *options; // Map<ConfigOption>
};

typedef struct config_task_t ConfigTask;
struct config_task_t
{
   int hours;
   int minutes;
   int seconds;

   Vector *actions; // Vector<char *>
};
```

Le vecteur des actions pour les *Tâches* pourra être utiliser avec le dictionnaire des *Actions* présent dans la structure *Config* présente ci-dessous.

```
typedef struct config_t Config;
struct config_t
{
   Map *actions; // Map<ConfigAction>
   Map *tasks; // Map<ConfigTask>
};
```

VI. Références

1) *Page Github* - github.com/quantumsheep/webscrapper-esgi.
2) *CURL* - curl.haxx.se.
3) *libcurl* - curl.haxx.se/libcurl.
4) *POSIX Thread* - pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html.