Projet HTTP - partie 2

Henri de Boutray - Alexis Bertrand - Guillaume Bruchon

Mars 2016

Table des matières

1 Méthode générale pour l'analyse syntaxique

Afin de faire un parseur comprenant la syntaxe d'un message HTTP, il a fallu implémenter la grammaire ABNF. Ensuite, nous avons fait le choix de stoquer le résutat du traitement dans une structure de donnée qui donnerait un accès facile aux données une fois ce traitement fait. (Et qui permettrait de ne pas avoir à parser de nouveau à chaque fois qu'une donnée est recherchée). Pour ce, nous avons choisi d'utiliser une map : une structure qui fera correspondre à chaque field-name, le header-field entier. Puisque nous avons la main sur cette structure, cela nous permet une gestion plus efficace des champs. Un point particulier a été considéré pour la start-line, qui ne se compose pas comme le reste du header.

Le code étant (en partie) commenté, pour toute information non contenue dans ce document, se référer aux fichiers sources directement.

2 Choix d'implémentation

2.1 Le partage des taches

Le parseur stricto sensus ne se compose donc que de lui même, mais la map est nécessaire pour stoquer le résultat du parseur. Il y a donc eu deux partie promordiales, suivies d'utilitaires et de fonctions de test. L'interaction entre la map et le parseur se fait de la manière suivante :

Une map a trois éléments obligatoires, method, request-target et HTTP-version. Le créateur de la map prend donc ces trois éléments en paramètres. Pour obtenir ces trois éléments, toutes les règles de la grammaire necessaires pour cette partie du projet ont été codées en utilisant des fonctions imitant le comportement des expressions régulières (les readers, cf partie suivante).

Ces fonctions lisent le symbole suivant dans le buffer et renvoient ce qui à été lu ainsi qu'un code d'état. Le buffer avance que si la lecture est un succès.

Une fois la map ainsi créée, le parseur parcours le fichier header-field par header-field en ajoutant à chaque fois les field-name et header-field à la map jusqu'à trouver un double CRLF.

À ce point, il ne reste plus qu'à récupérer les symboles désirés dans la map pour avoir des tests fonctionnels. Pour cela, il suffit de demander à la map les différents éléments étants inclus, en notant que field-value n'étant pas inclu dans la map à part entière (redandant avec le header-field) et qu'il faut donc le recalculer à partir du header-field à chaque demande (un traitement très léger est utilisé sans readers)

2.2 Le parseur

Le Parseur est divisé en deux parties :

- Une fonction principale qui parse un message HTTP entièrement et stock la start-line et les header-fields dans une "map"
- Une api de test (fonction parse()) ainsi qu'un main pour les tests unitaires

Dans tous le projet nous utilisons notre propre structure de donnée pour stocker une string : le type StringL :

```
typedef struct {
    char* s;
    unsigned int len;
} StringL;
```

Il nous permet de stocker une chaîne de caractère sous forme d'un pointeur et d'une longueur.

La fonction parse_return parse_HTTP_message(StringL* buff); prends en paramètre un buffer contenant le message HTTP et retourne une structure contenant une possible map chargée (voir section suivante) et un code d'erreur. Son fonctionnement est simple, elle utilise des fonctions de lecture

élémentaires appelés readers pour parcourir l'entête, et vérifier la syntaxe. De plus elle stock au fur et à mesure les champs d'intérêt dans la map.

Les fonctions readers : Le coeur du parseur est la fonction get_reader() qui prends en paramètre un buffer et un symbol (défini dans un enum) et retourne un objet qui permet de lire/consommer ce symbol à l'emplacement actuel du buffer. Il existe autant de symbols qu'il y a de règles dans la grammaire ABNF, et les readers sont générés lors de l'exécution selon ces règles dans la fonction get_reader(). Le but de cette approche est de composer des éléments de syntaxe en fonction d'autres éléments de syntaxe déja défini via des opérateurs logiques.

voici des exemples d'opérateurs de base :

- R or(R1,R1) créé un reader qui lis la même chose que R1 ou R2 (qui sont eux même des readers)
- R concat(R1,R2) concaténation de la lecture de R1 puis R2
- R kleene(R1) fermeture itérative de R1
- ...

voici des exemples de générateurs de readers de base :

- R letter(char) créé un reader qui lis un certain caractère
- R charIn(char[]) créé un reader qui lis un des caractères dans le tableau (c'est une string) non nécessaire mais pratique (on peux y remplacer par des letter() et or() mais fastidieux
- ...

Bien entendu, ces fonctions servent seulement à construire des readers, la lecture se fait dans un second temps avec ces readers.

Conceptuellement, un reader est une fonction. Plus particulièrement une closure dans d'autres languages, c'est à dire un fonction défini dans un certain contexte qui perdure entre les différents appels. Pour simuler ce comportement, nous séparons le contexte du code de la fonction, et une structure "reader" est là pour assembler ces deux composantes.

```
typedef struct {
    void* ctxt; //contexte de la closure
    read_return (*fun)(void*); //code de la closure
} reader;
```

Le contexte est une structure qui est propre à chaque closure (d'où le void*) lors de l'appel de la closure, le contexte est passé en paramètre. toutes les closures se trouvent dans le fichier src/parseur/-base_readers.c

Utilisation d'un GC :

Durant le développement, nous nous somme rendu compte qu'il n'y avait pas de moyen simple de free() les contextes car on ne sait jamais si un contexte va être réutilisé, et des problèmes de double free() se posent, ce qui complexifie nettement le code. Une solution est de stocker en global toutes les références des contextes et de tout free d'un coup à la fin du parsing, ce qui reviens à implémenter un mini GC, nous avons plutôt choisis un vrai GC pour une question de simplicité. Mais cela risque de changer pour la solution précédente qui n'est pas non plus très difficile à mettre en oeuvre.

http ://www.hboehm.info/gc/

2.3 Map : Stockage des informations trouvées par le parseur

2.3.1 Structure

L'objectif de la MAP est de pouvoir stocker les différents éléments trouvés dans la requête http par le parseur. Une requête est toujours composée de champs « obligatoire » comme tout ceux de la start-line mais aussi d'autres champs variables. Il faut donc bien sûr avoir une structure dynamique qui pourra accueillir tous les éléments pour pouvoir ultérieurement faire des recherches à l'intérieur.

On a alors défini 2 structures :

— MapStruct : qui sera la base de la map, avec les champs obligatoire de la start-line et un pointeur vers la liste chainée qui contient les autres champs.

```
typedef struct {
    StringL methode;
    StringL request_target;
    StringL http_version;
    Field* fields;
} mapStruct;
```

— Field : maillon de la chaine qui contient le field-name ainsi que le field-value pour chaque header-field détecté par le passeur.

```
typedef struct {
    StringL field_name;
    StringL header_field;
    Field* suivant;
} field;
```

Exemple :

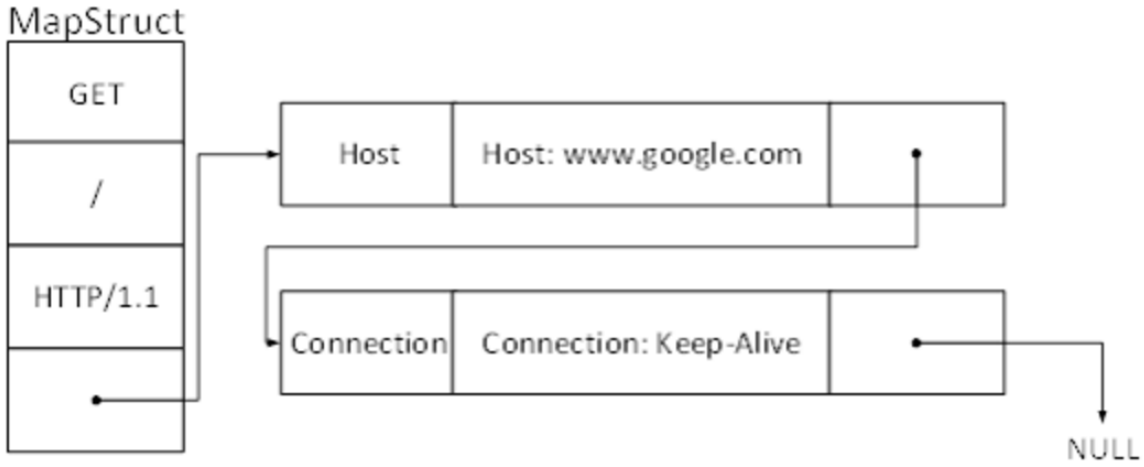

FIGURE 1 – exemple de map

2.3.2 Ajout d'éléments

Lors de la création de la MAP, on remplit en même temps les éléments de la start-line. Ensuite, à chaque nouvel header-field lu par le parseur, on l'insère dans la liste chainée en tête (pour des raisons de performances).

Remarque : On peut ajouter plusieurs fois les mêmes champs, mais si pour certains ce n'est pas possible dans le RFC. C'est le serveur qui, plus tard, jettera le paquet s'il n'est pas bon. Ici on se contente juste de vérifier la syntaxe.

2.3.3 Recherche d'élément

Pour la recherche d'élément dans cette MAP, il faut tenir compte de nombreux éléments :

— Si c'est un champ dans la première ligne :

    - Si c'est un champ dans la start-line (methode, request-target, HTTP-version) : On renvoie directement le StringL correspondant
    - Si c'est la start-line elle-même ou la request-line, il faut d'abord reconstituer la ligne (car on ne la stocke pas entière), puis on peut la retourner
- Si c'est un élément « générique » (header-field, field-name, field-value) : On parcourt toute la liste chainée et avec on retourne tous les éléments concernés
- Si on a une requête directement sur un field, on parcourt toute la liste afin de pouvoir chercher le champ qui ont été enregistré avec ce field-name :
    - Si la demande se termine par -header, on sait qu'il faudra tout le header-field
    - Sinon c'est qu'il faudra uniquement le field-value

La fonction retourne le nombre d'élément trouvé et donc soumis à la fonction de callback().

2.4 Le système de test

Nous avons effectué deux types de tests durant le développement :

- Les tests visuels, on vérifie les résultats à la main. Ces derniers sont faciles à mettre en oeuvre et pratique pour le débug mais non viables sur le long terme
- Les tests automatiques, les résultats sont vérifiés automatiquement par comparaison avec un résultat attendu mise en place plus difficile mais indispensable pour surveiller la constance du comportement du code au cours de son évolution

Au cours du développement les tests visuels sont convertis en tests autimatiques une fois le comportement acquis. Il y a 2 types de tests automatiques :

- Les tests de fonctions outils : ce sont des tests destinés à vérifier le comportement de fonctions variées et importantes du programme comme :
    - l'API du type StringL
    - l'API de la map
    - les les readers générés par get_reader() Ici tous les tests consistent en l'appel d'une fonction spécialisée pour tester un symbole avec un buffer d'entrée et une expectation de lecture
- Les tests généraux du parseur : Ce sont les tests Concrets avec de véritables requêtes stockés dans des fichiers. Pour cela nous utilisons l'API de test demandé ainsi qu'un script shell pour faciliter l'écriture des tests (on peut créer des tests visuels et automatiques) Pour une raison obscure provenant sûrement du shell certains tests automatiques ne fonctionnent pas (ils ont été enlevés) et la vérification a été faite à la main

Les tests peuvent êtres lancés depuis src/ avec make test

3 Exemples d'exécution

Voici plusieurs exemples d'exécution, qui ont étés réalisés avec les requettes donées en exemple sur Chamilo :

- Pour récupèrer un élément de la première ligne

```
bin/mainTest.e tests/get1 start-line
GET / HTTP/1.1
```

- Pour récupèrer un field-value

```
bin/mainTest.e tests/get7 User-Agent
Mozilla/5.0 (X11; Linux x86_64; rv:44.0) Gecko/20100101 Firefox/44.0
```

- Ou pour récupèrer un header-field entier

```
bin/mainTest.e tests/get7 User-Agent-header
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:44.0) Gecko/20100101 Firefox/44.0
```

- Pour récupèrer la liste des field-name

```
    bin/mainTest.e tests/get4 field-name
Connection
Strange-Header
Accept-Language
Accept
User-Agent
Host
```

On peut voir que même les champs "étranges" sont aussi parsés

4 Pour faire vos propres tests ...

Afin de réaliser vos propres tests, il suffit de remplacer dans le dossier src notre main.c par le votre, en ajoutant :

```
#include "api_test.h"
```

Afin de compiler vous pouvez tout simplement exécuter make sans argument dans le même dossier. L'exécutable sera alors crée dans le dossier bin.