Université de Liège
Faculté des Sciences Appliquées

Introduction to computer networking [INFO0010-1]

Projet 2 : Rapport

Mormont Romain, Servais Fabrice
3ème bachelier ingénieur civil, orientation ingénieur civil
Options *informatique* et *électricité et électronique*
s110940, s111093

Année académique 2013-2014

Table des matières

1 Software architecture **3**
1.1 Vue globale de la solution . . . 3
1.1.1 Initialisation du *Gateway* . . . 3
1.1.2 Récupération des connexions entrantes . . . 3
1.1.3 Gestion des requêtes . . . 3
1.2 Erreurs renvoyées par le *Gateway* . . . 3
1.3 Interaction avec le HTML . . . 5
1.3.1 Éléments HTML . . . 5
1.3.2 Parser . . . 5
1.3.3 Filtre . . . 6

2 Multi-thread coordination **6**

A Machine d'états **7**
A.1 Parsing des balises ouvrantes . . . 7
A.2 Parsing du contenu . . . 7

1 Software architecture

Le problème a été divisé en différents sous-problèmes que nous allons expliciter dans leur ordre temporel.

1.1 Vue globale de la solution

1.1.1 Initialisation du *Gateway*

Une instance de `Server` est créée au lancement de cette classe. Celle-ci crée une instance de `HTTPServer` et `ConfigurationServer` afin d'accepter les connexions entrantes vers le *Gateway* et la plate-forme de configuration. Elle récupère aussi le singleton `Displayer` servant à afficher des messages dans la console. Nous ne détaillerons pas la plate-forme de configuration.

1.1.2 Récupération des connexions entrantes

La classe `HTTPServer` récupère les connexions entrantes et lance un thread (si possible par la thread-pool) pour chacune d'entre-elles par l'intermédiaire de la classe `HTTPClientRequestThread` et ce, après l'acceptation de connexion du socket.

1.1.3 Gestion des requêtes

La classe `HTTPClientRequestThread` instancie un objet `HTTPRequest` en prenant le socket en argument. Celui-ci va récupérer la requête du navigateur et la parser pour en récupérer les différentes informations, notamment la méthode et le chemin.

On récupère alors l'adresse IP du *Gateway* qui va être utile lors du remplacement des liens, et l'on donne la requête récupérée à un nouvel objet de classe `GatewayRequestDecoder` qui va se charger de décoder la requête, c'est-à-dire récupérer l'adresse du site auquel l'utilisateur souhaite accéder et les éventuels paramètres (`forceRefresh`). On vérifie ensuite si l'adresse est correcte et que le format du fichier auquel on veut accéder est géré.

Pour le chemin donné, on regarde si la page est déjà contenue dans le cache et si celle-ci doit être rafraîchie dû au timeout ou à l'éventuel argument `forceRefresh`. Selon les cas, on va rechercher la page dans le cache ou sur le serveur distant. Dans ce cas-là, la méthode `getPageFromRemote(URL)` établit la connexion avec le serveur avec la classe `HttpURLConnection` et récupère le contenu de la page. Les liens de la page sont alors filtrés afin de les rediriger vers le *Gateway*. Les balises `img`, `link`, `frame` et `script` sont aussi filtrées afin de bypasser le *Gateway*.

On clone ensuite la page récupérée (afin de ne pas altérer la page originale contenue dans le cache) puis on filtre la page en fonction des mots-clés interdits. Cette étape est réalisée après la récupération de la page du cache ou du serveur distant. En effet, il est possible que la page ait été modifiée sur le serveur distant pendant le temps où celle-ci se trouvait dans le cache, amenant possiblement des mots-clés interdits et devant provoquer une censure, ce qui n'aurait alors pas été le cas, jusqu'au timeout dans le cache. Si la page doit être censurée, on renvoie une page informant d'utilisateur qu'il ne peut pas accéder à la page. Sinon, on envoie une réponse HTTP contenant la page voulue ou pas selon si on répond à une requête GET ou HEAD.

1.2 Erreurs renvoyées par le *Gateway*

Voici une liste de différentes erreurs renvoyées par le *Gateway* ainsi que l'action qui les a déclenchées :

- **Erreur 400 - Bad Request** : La requête du client n'a pu être lue (dû à la variable `s` manquante dans l'URL, argument de `s` vide,...)
- **Erreur 500 - Internal Server Error** : Lorsqu'une erreur lors du parsage du code HTML a été rencontrée.

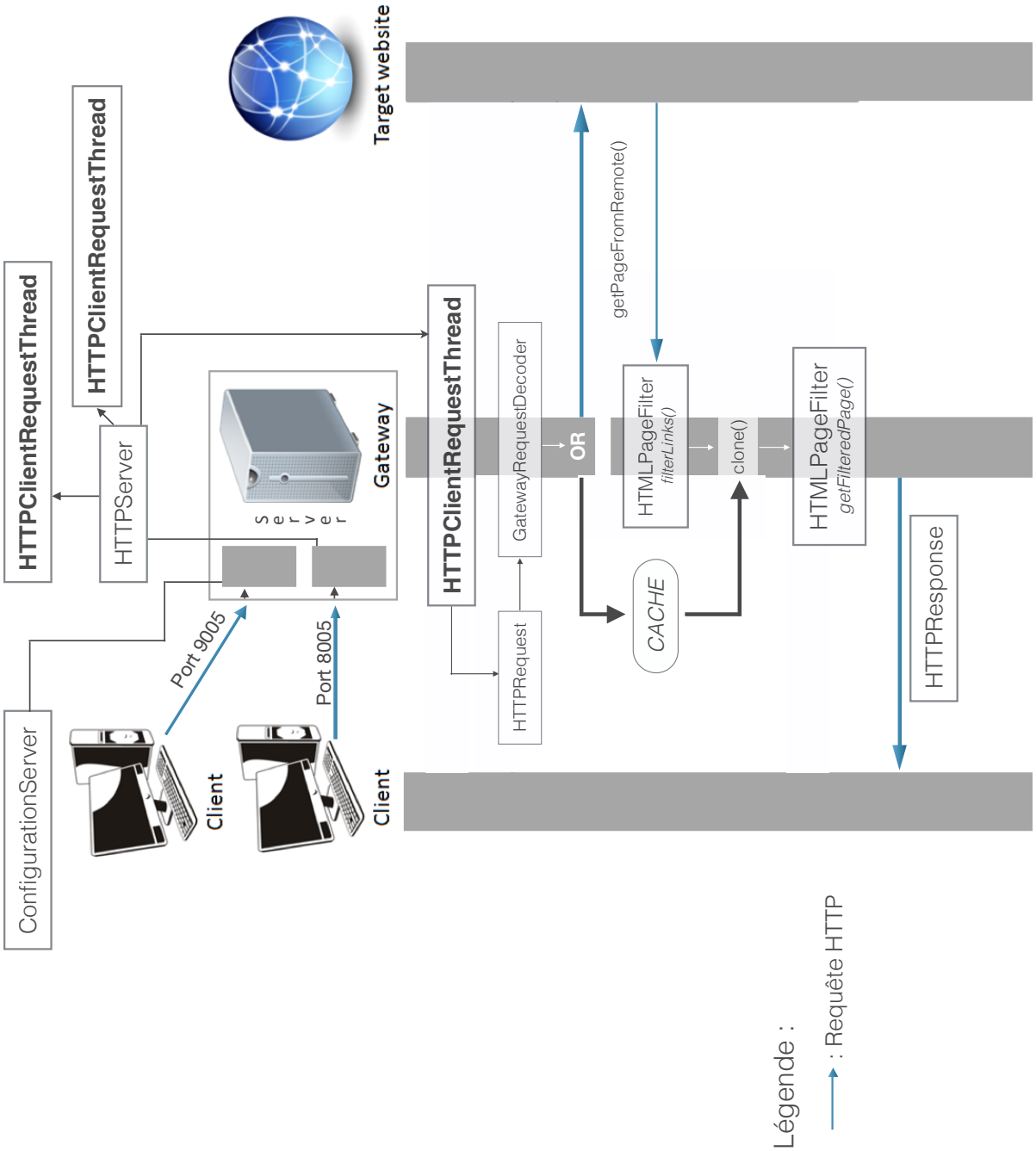

FIGURE 1 – Schéma général de l'implémentation

– **Erreur 501 - Not Implemented** : La requête que l'on reçoit n'est ni GET ni HEAD ou lorsque le format de la page demandée n'est pas prise en charge.
– **Erreur 502 - Bad Gateway** : Le serveur distant n'a pas pu être atteint.
– **Erreur 504 - Gateway Timeout** : Le serveur distant a été trop lent à joindre ou à répondre.

Lorsque le serveur distant répond et qu'il renvoie un code d'erreur HTTP, le *Gateway* transmet cette erreur au client.

1.3 Interaction avec le HTML

Nous avons voulu éviter, lors du traitement des liens et des mots clés, de devoir manipuler une longue chaîne de caractère contenant le code HTML. Ainsi faciliter ces manipulations, nous avons développé un package `html` contenant une série de classes utiles.

1.3.1 Éléments HTML

Voir package `html`

Tout d'abord, nous avons développé une classe `HTMLPage` qui contient notre représentation interne d'une page. Ensuite, nous avons développé une classe abstraite `HTMLElement` pour représenter un élément HTML et des classes héritant de cette dernière pour représenter les éléments concrets. Notre classification ainsi que les classes associées à chaque élément sont données dans la Table 1.

Élément	Forme	Classe
Balise ouvrante	`<tag_name attr="value" ...>`	`HTMLOpeningTag`
Balise fermante	`</tag_name>`	`HTMLClosingTag`
Balise commentaire	`<!-- ... -->`	`HTMLComment`
Contenu	Tout contenu ne se trouvant pas à l'intérieur d'une balise	`HTMLContent`
Attribut d'une balise ouvrante	`attr="value"`	`HTMLAttribute`

TABLE 1 – Classification des `HTMLElement`

Ces classes définissent toutes les méthodes permettant de consulter ou de modifier leurs différentes caractéristiques. Par exemple, la classe `HTMLContent` définit une méthode permettant de remplacer chaque caractère d'un mot donné par un autre caractère. Ou encore, la classe `HTMLPage` définit une méthode permettant de récupérer toutes les balises ouvrantes ayant le même nom.

Notre représentation interne d'une page est une liste d'objets `HTMLElement`.

1.3.2 Parser

Voir package `html.parser`

Afin de pouvoir exploiter les outils définis précédemment, il est nécessaire de convertir un code HTML en objets. Pour ce faire nous avons développé la classe `HTMLParser` permettant de pratiquer une analyse lexicale du code sur base d'une machine d'états finie.

La classe est composée, entre autres, de plusieurs méthodes chacune chargée de parser un élément HTML particulier (voir Table 1) :
– `parseOpeningTag` : balise ouvrante (machine d'états donnée dans la Figure 2)
– `parseClosingTag` : balise fermante
– `parseContent` : contenu (machine d'états donnée dans la Figure 3)

– `parseAttribute` : attribut (et sa valeur s'il y en a une) (machine d'états donnée dans la Figure 2)
– `parseComment` : commentaire

Ces méthodes sont coordonnées par les méthodes `parseTag` et `parseHTML`. La première, en fonction de la balise rencontrée, passe la main à `parseOpeningTag`, `parseClosingTag` ou `parseComment` et la seconde, en fonction du caractère, passe la main à `parseTag` ou `parseContent`.

A la fin de l'analyse, le parser a construit une liste de `HTMLElement` qui constitue notre représentation interne du code dans `HTMLPage`.

1.3.3 Filtre

Voir package `html.filter`

Le filtrage des liens nous a posé pas mal de problèmes. En effet, étant donné la permissivité du langage HTML, les liens peuvent être relatifs ou absolus, avec ou sans protocole,... De plus, il se peut que le contenu de la balise `href` ne soit pas un lien vers une page web mais un lien vers une adresse mail (`mailto:...`) ou vers une fonction javascript (`javascript:...`). Enfin, l'adresse de référence des liens relatifs peut être différente de la racine du serveur selon que la balise `<base ...>` ait été déclarée ou non à l'intérieur des balises `head`.

Étant donnée la complexité du problème, il nous a semblé judicieux de créer une classe dédiée à sa résolution : la classe `LinkFilter`. En plus de reconstruire une url absolue sur base de l'adresse d'une page web et d'un lien, cette classe permet aussi d'encoder une url et de construire le lien complet dirigeant vers le *Gateway*. Cette classe est utilisée dans la classe `HTMLPageFilter`.

Cette dernière classe permet aussi de filtrer la page selon les critères précisés dans l'énoncé. Les différents états sont repris dans l'énumération `PageGatewayStatus`.

2 Multi-thread coordination

L'accès au cache est *thread-safe*. En effet, le cache est implémenté avec la classe `ConcurrentHashMap` qui est *thread-safe*. De plus, nous avons ajouté l'attribut `synchronized` aux méthodes de `Cache` agissant avec le cache. Si deux threads tentent d'accéder à une même page se trouvant dans le cache, ces deux threads recevront la même page, sans incohérence.

Le seul moment où le cache est modifié est lors l'insertion ou la mise à jour d'une page lorsqu'un utilisateur demande cette page. Supposons que deux utilisateurs souhaitent accéder à la même page ne se trouvant pas dans le cache. Les deux threads demanderont tous les deux au serveur distant après la page, filtreront les liens et attributs cités au point 1.1.3, et tenteront de l'insérer dans le cache. Un premier y parviendra avant l'autre, et ce deuxième ajout aura pour effet d'écraser la précédente. Cependant, ceci n'est pas un problème puisque l'on peut supposer que la page n'a pas changé entre les moments où les utilisateurs ont demandé la page, celle-ci est alors dans le même état pour chacun des threads.

On peut noter aussi que le clonage de l'élément contenu dans le cache ajoute un niveau de sécurité supplémentaire car, lorsque l'on modifie la page, le cache n'est pas modifié, cela n'a donc pas d'incidence sur les autres threads.

A Machine d'états

A.1 Parsing des balises ouvrantes

La représentation graphique est donnée à la Figure 2.

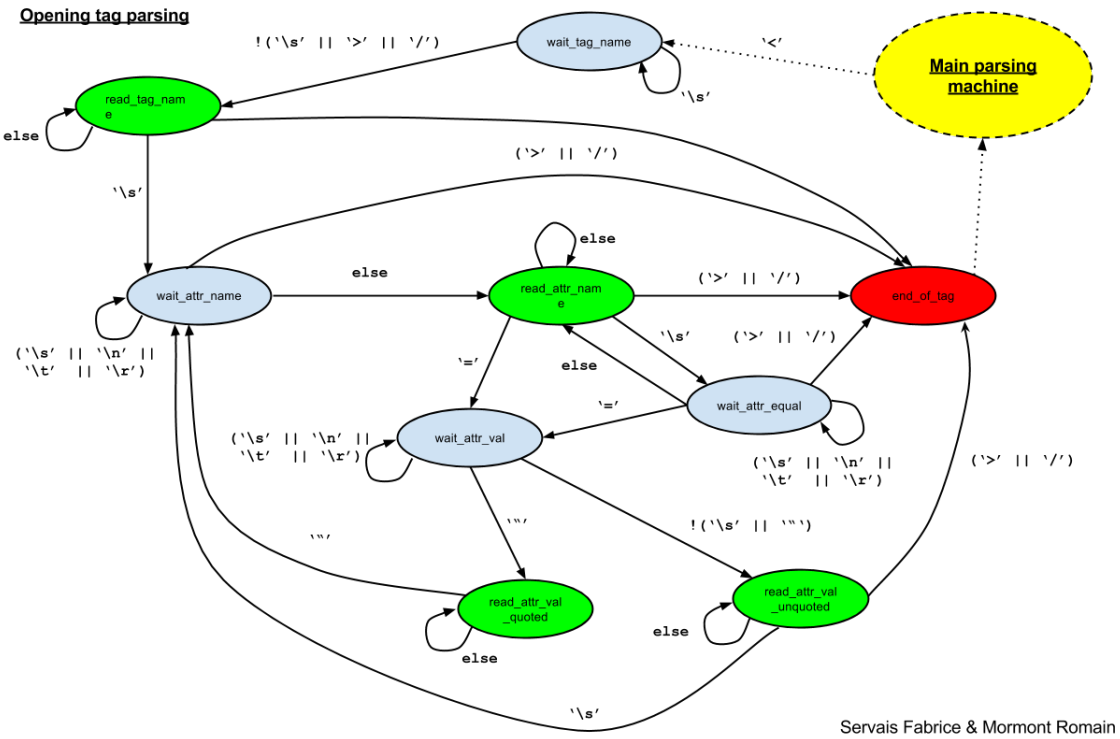

FIGURE 2 – Parsing des balises ouvrantes

Les différents états :
- `wait_tag_name` : en attente du début du nom de la balise
- `read_tag_name` : lecture du nom de la balise
- `wait_attr_name` : en attente d'un nom d'attribut
- `read_attr_name` : lecture du nom d'un attribut
- `wait_attr_equal` : en attente du `=` suivant l'attribut
- `wait_attr_val` : en attente de la valeur de l'attribut
- `read_attr_val_unquoted` : lecture d'un attribut sans apostrophe ou guillemet
- `read_attr_val_quoted` : lecture d'un attribut avec apostrophe ou guillemet

Ce modèle d'état ne prend pas en compte les cas où le nom d'attribut est entouré de guillemets (peut arriver pour certaines balises `!DOCTYPE`). Ce cas a néanmoins été pris en compte dans le code.

A.2 Parsing du contenu

La représentation est donnée à la Figure 3.

Les différents états :
- `read_content` : lecture du contenu
- `check_end` : dans le cas où le contenu est un code javascript (placé dans des balises `script`), il se peut que le chevron fermant ne soit pas le début de la balise fermante `</script>`. Il faut s'assurer que c'est bien le cas avant d'achever le parsing du contenu.

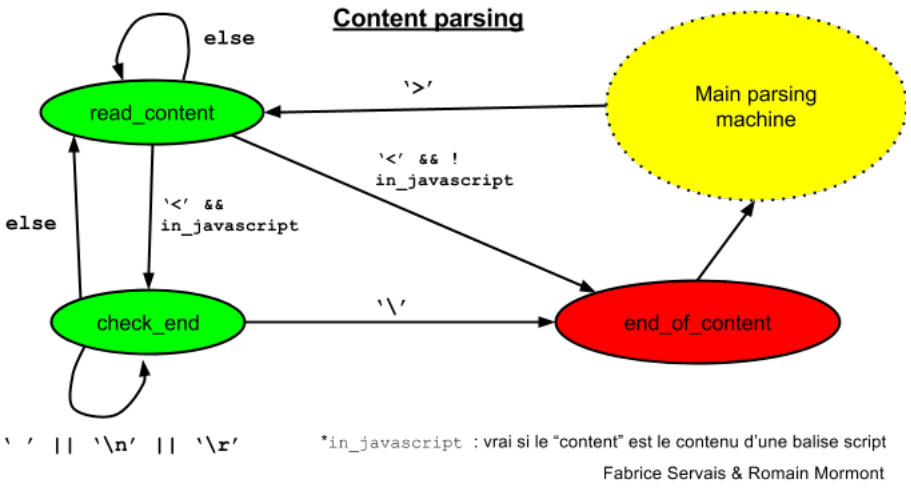

FIGURE 3 – Parsing d'un contenu