Enseirb-matmeca

Seconde année, Département électronique

Module PG208

Projet POO C++ : Serveur Web HTTP évolué

Auteurs :
MOUHAGIR Ayoub
FERIAT Abdessamad

Encadrant :
LEGAL Bertrand

18 Mai 2019

Table des matières

1 Introduction

L'objectif de ce projet est de concevoir un serveur HTTP évolué. Il doit permettre à tout utilisateur, qu'il soit client ou administrateur, de communiquer avec le serveur en envoyant des requêtes sous formes de page internet sur le poste où le serveur est lancé.

2 Cahier des charges

Notre serveur HTTP doit supporter les fonctionnalités suivantes :

- Traitement des requêtes de lecture de fichier.
- Traitement des requêtes de lecture de fichier (scripts).
- Traitement des requêtes de lecture du contenu d'un répertoire (listing du contenu des fichiers et répertoires contenus à une adresse).
- Traitement des erreurs (page non présente, pas de droit d'accès, etc.).
- Mémorisation des statiques d'utilisation.
- Génération d'une page d'information à la volée sur l'identité du serveur (nom des concepteurs, date et heure de compilation, etc.).
- Génération d'une page d'information à la volée sur les statistiques du serveur (combien de requêtes ont été reçues, traitées, etc.).
- Un gestionnaire de cache des fichiers récemment accédés : afin de réduire la consommation d'énergie, les accès disques, les calculs à réaliser tout en augmentant le débit du serveur, nous souhaitons mémoriser en mémoire RAM les pages récemment accédées afin d'évider de devoir les relire sur le disque dur ou le réseau à chaque requête.

3 Diagrammes du langage UML

3.1 Diagramme de cas d'utilisation

Le diagramme de cas d'utilisation nous donne une vision globale sur les fonctionnalités à implémenter afin de satisfaire les exigences du cahier des charges. Nous sommes partis sur un point de départ fourni par notre encadrant qui assure les fonctions de base suivantes :

- Création d'un serveur TCP écoutant sur le port 8080.
- Gestion des connexions réseaux entrantes.
- Un exemple de requête/réponse.

Après avoir testé et compris ce code fourni, ce qui est attendu de nous c'est d'étendre les capacités de l'outil dans le but de répondre au diagramme de cas d'utilisation suivant :

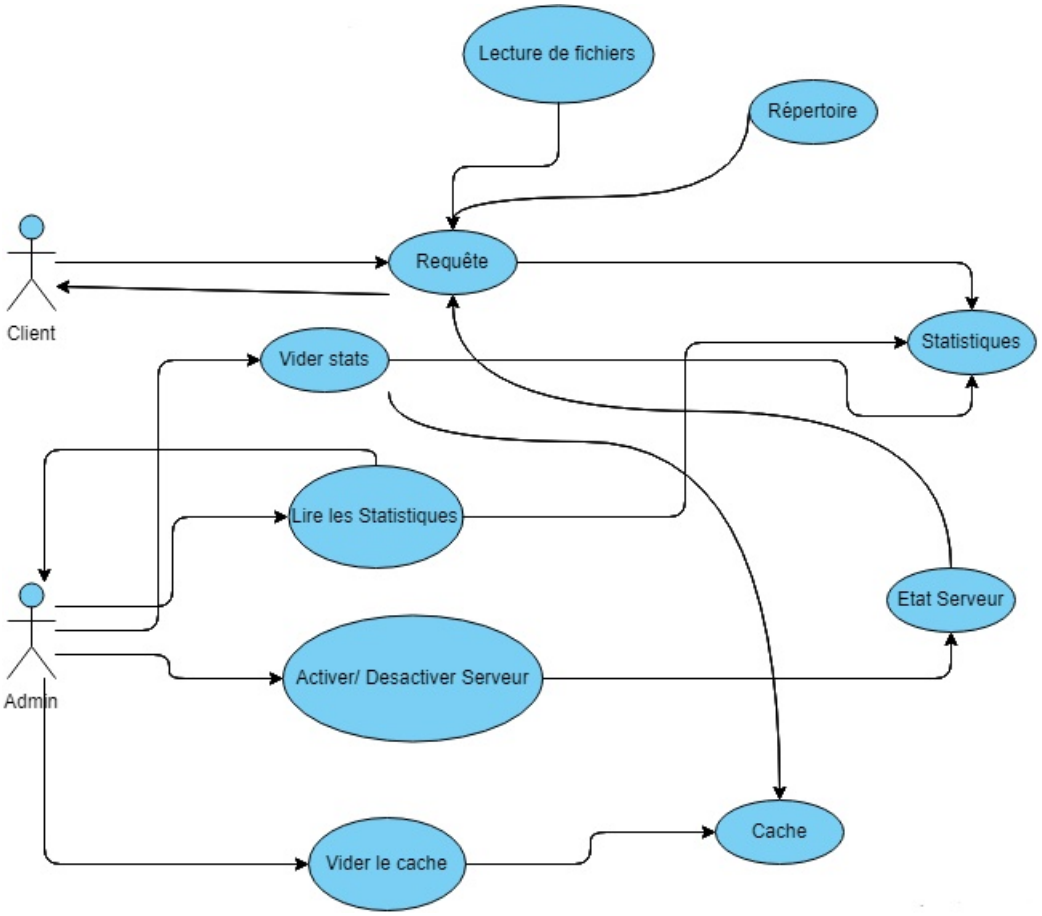


FIGURE 1 – Diagramme de cas d'utilisation

On remarque que l'outil doit s'adresser à deux type d'utilisateurs :

- Un utilisateur normal qui envoie, après que le serveur soit bien évidemment activé, des requêtes qui lui permettent d'accéder au contenu d'un fichier ou d'un répertoire. Il peut également demander d'afficher des pages d'informations à la volé.
- Un utilisateur Admin qui possède à sa disposition des fonctionnalités supplémentaires. Il peut surveiller le bon fonctionnement du serveur (Activation et désactivation du serveur, Effacement du contenu de la mémoire cache et les statistiques,..).

3.2 Diagramme de classe

Ce diagramme de classe rassemble l'ensemble des classes nécessaires pour aboutir la tâche, ainsi que les différentes relations entre celles-ci. Ce diagramme est représenté dans la figure 2.

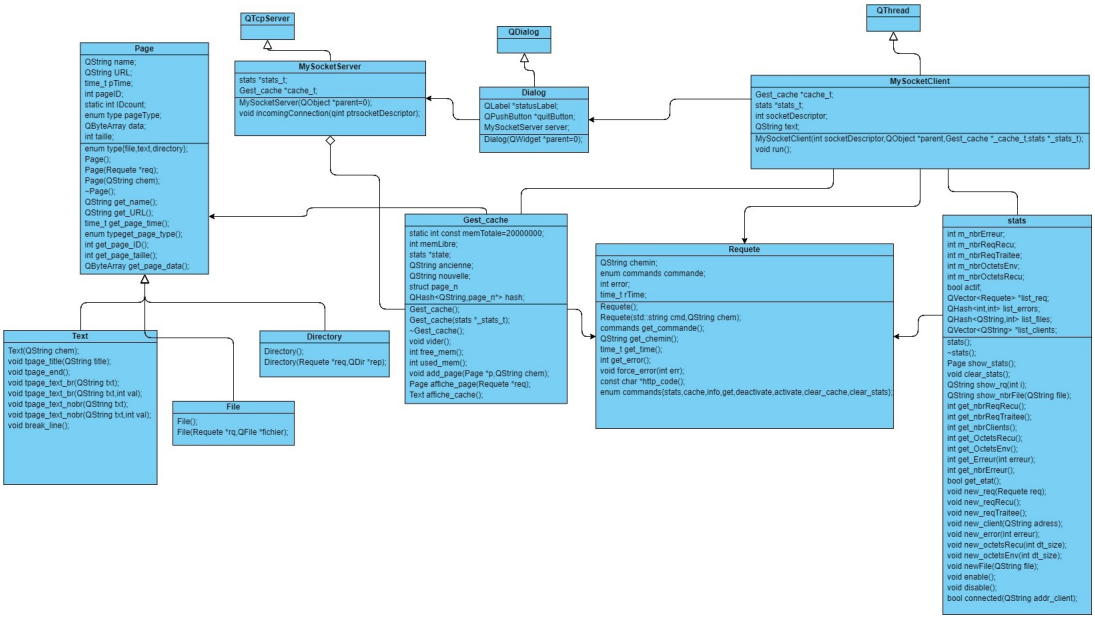


FIGURE 2 – Diagramme de classe

On présentera dans la suite chaque classe en détails.

- **Trait_Requête :** Elle permet de gérer les requêtes de l'utilisateur. Une requête sera caractérisée par son chemin et la commande exécutée. Ensuite, la classe se chargera de mémoriser les erreurs et de générer la réponse HTTP correspondante à la requête.
  Des méthodes sont créées pour accéder à toutes ces informations.
- **Page :** Elle permet de mémoriser les données de la ressource à laquelle on accède. Ainsi que sa taille et son type.
  Les classes **File**, **Directory** et **Text** héritent de la classe Page. Ils permettent de construire la page en fonction de la donnée demandée.
- **Stats :** Elle permet de mettre à jour toutes les statistiques d'utilisation et de les afficher. Elle est également chargée de mémoriser l'état du serveur et permettra de le changer.
- **Gest_cache :** Elle permet de mettre en cache des pages qu'on accède régulièrement. C'est dans cette classe qu'on renvoie la page correspondante à la requête. En plus, elle se charge de la mise en mémoire la première et la dernière page consultée. Ainsi que la page précédente et la page suivante d'une page donnée.

3.3 Diagramme de séquence

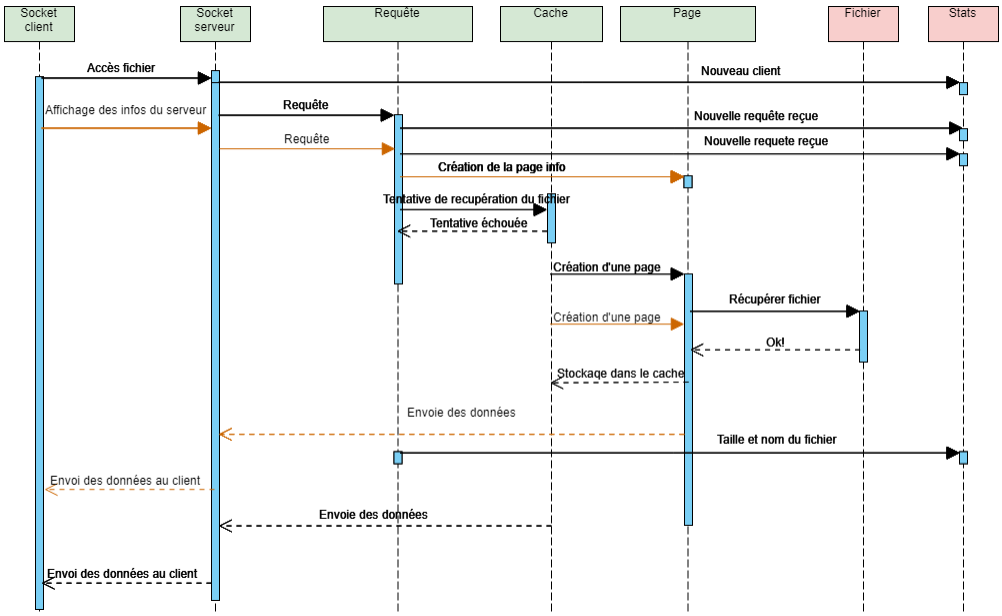


FIGURE 3 – Diagramme de séquence

Après avoir envoyé une demande au serveur par le client, une **Requête** est créée. Ensuite, cette requête sera utilisée pour demander au cache d'afficher la page correspondante. Si la page en question n'est pas en cache, elle est créée et stockée. Selon l'information demandée, un fichier correspondant est ouvert (même au cas des erreurs là où on envoie le code HTML de l'erreur) et son contenu est envoyé sur une page HTML au client. Enfin, durant cette opération, on met à jour toutes les statistiques.

En ce qui concerne l'affichage des statistiques et l'état du cache, une requête est créée et transmise au cache. Pour les statistiques, c'est la classe **Stats** qui s'occupe de la création de la page à la volé puis elle la transmet au cache. Contrairement à la page de l'état du cache qui est créée par le **Cache** lui-même. Ces deux pages ne seront pas mises en cache comme leur contenu change au fur et à mesure.
Des méthodes sont créées pour accéder à toutes ces informations.

4 Traitement d'une requête

Cette classe nous permettra de faire le lien entre les demandes du client et le reste du programme. Une requête sera caractérisée par son chemin et la commande exécutée. Ensuite, la classe se chargera de mémoriser les erreurs et de générer la réponse HTTP correspondante à la requête.

En outre, L'objet se charge de différentier les requêtes envoyées par le type de la donnée demandé, soit évidemment un fichier/dossier ou l'affichage d'une page d'information ainsi qu'une tache de maintenance comme le vidage du cache et les statistiques. Cela nous permettra ensuite de traiter chaque requête différemment.

Les attributs de la classe sont les suivants :

- **chemin :**C'est une chaîne de caractère **QString** qui représente l'emplacement du fichier ou le lien auquel on accède. Ce chemin est relatif à l'emplacement depuis lequel le programme est éxecuté. Un exemple de chemin serait : *./public_html/image.jpg*, cela correspond à l'image *image.jpg* contenu dans le dossier *public_html* situé à la racine du serveur.
  Pour l'affichage des pages à la volée (statistiques par exemple) le chemin serait le suivant : */private/stats.html*, ceci n'est pas un fichier ni un dossier, il représente un lien qui nous permettra dans la suite de distinguer la commande et afficher la page correspondante.
- **commande :**Il représente le type de l'information demandée par le client. C'est un type énuméré qui prend par exemple selon le chemin récupéré :**get** quand l'utilisateur essaye d'accéder un fichier ou un dossier, **info** pour l'affichage de la page d'informations, **stats** pour l'affichage des statistiques ainsi de suite.
- **Error :** Il représente un code d'erreur HTTP qui sont définis par défaut dans le protocole HTTP. Il nous permet de créer l'entête HTTP correspondante à la requête du client. Par exemple, pour l'erreur 404 qui signifie que l'information demandé n'existe pas, l'entête correspondante est : *HTTP/1.1 404 Not Found*. Or, si la requête n'a pas rencontré des problème la réponse est *HTTP/1.1 200 OK*. Cela est géré par la méthode **http_code()**.

Une requête est crée dans *MySocketClient.cpp* à chaque fois qu'un client envoie une demande en prenant en paramètres la méthode (En général **GET**) et le chemin de la ressource. En plus, Le constructeur met la valeur d'erreur en 200.

5 Création d'une page

La classe Page contient les informations nécessaires pour la création d'une page html, qui servira comme réponse du serveur aux requêtes du client. Trois classes dérivent de la classe Page :

- **File :** Permet l'ouverture d'un fichier. Son constructeur prend en argument l'adresse de la requête ainsi que celle du fichier Qfile. Après avoir récupéré la taille et le type de la page, on ouvre le fichier. Une erreur 500 est soulevée en cas d'échec d'ouverture du fichier. Finalement le contenu du fichier est affiché sur la page.
- **Directory :** Permet l'affichage du contenu d'un répertoire. Son constructeur prend l'adresse de la requête ainsi que celle d'une répertoire Qdir. La classe détermine le type de la page, puis écrit l'entête html. Ensuite, elle parcourt le contenu du répertoire et crée un lien pour chaque fichier ou répertoire trouvé. Il faut aussi rajouter un '/' au lien d'un répertoire pour pouvoir y accéder.
- **Text :** Permet de générer une page à la volée. Elle contient des méthodes pour générer l'entête et la fin de la page html, pour écrire une ligne de texte avec ou sans retour à la ligne.

6 Statistiques

La classe Statistiques permet l'accès et le stockage des statistiques d'utilisation du serveur HTTP. Elle permet aussi l'affichage des statistiques sous forme d'une page

de type text.

Nous utilisons des objets QHash pour stocker la liste des erreurs et la liste des fichiers, et des objets QVector pour la liste des clients et des requêtes. Ce choix est justifié par le fait qu'on a pas besoin de stocker le nombre d'occurrences des requêtes et des clients.

Pour mettre à jour les statistiques d'utilisations du serveur, On suit l'enchaînement suivant :

- On récupère la requête après avoir établi la connexion avec le client et récupérer son adresse.
- On stocke le nombre d'octets envoyé par la requête du client.
- On récupère le nombre d'octets envoyés par la réponse du serveur en récupérant la taille de la page envoyée au client.
- Si la requête correspond à la lecture d'un fichier, on récupère le nom du fichier après avoir accéder au fichier.

Par ailleurs, une méthode `show_stats()` permet l'écriture d'une page html contenant tous les données de statistiques stockés en mémoire. Cette méthode écrit ligne par ligne le contenu de la page à l'aide des méthodes de la classe **Text**.

7 Gestionnaire de cache

Cette classe nous permet de garder en mémoire des pages qu'on accède régulièrement. De plus, c'est dans cette objet qu'on renvoie la page correspondante à la requête. Un cache est créé lors de l'initialisation du serveur et aucun autre exemplaire n'est créé dans la suite.

Le stockage des pages en mémoire se fait à l'aide de la classe **QHash**. Celle-ci permet d'associer et mémoriser une clé à une valeur. Dans notre cas, la clé correspond au chemin de la requête et la valeur contenue est sous forme d'une structure de données qui contient un pointeur vers la page demandée et deux chaînes nommées **suivante** et **derniere** qui correspondent au chemin de la page suivante et précédente. Ce qui nous permettra d'afficher les pages en ordre chronologique de leur création. La classe contient également le chemin de la page la toute première page créée (la plus ancienne) et la plus récente afin de savoir quelle page à supprimer pour libérer l'espace dans le cache et aussi quelle page page précède celle qu'on vient d'ajouter.

Ajout d'une page dans le cache : add_page(Page*,QString)

Pour la mise en cache d'une page, on utilise la méthode **add_page**(Page*,QString) qui prend en argument le pointeur de la page et son chemin. La méthode stocke la page dans le **QHash** après avoir attribué la donnée **nouvelle** comme le chemin de la page qui la précède et celle qui vient juste après tout en modifiant les attributs **nouvelle** et **ancienne** en conséquence (la page ajoutée sera déclaré comme la plus récente ou la plus ancienne si c'est la première dans le cache).

Affichage d'une page : affiche_page(Requete*)

L'affichage de la page sera en fonction de la commande de la classe **Requete** ainsi que le type d'erreur. A chaque fois qu'on reçoit une requête, on vérifie d'abord si la page correspondante existe dans le cache. Si oui on la renvoie directement, si non on l'ajoute au cache puis on la renvoie. Pour le cas des fichiers/dossiers, on crée la page correspondante à l'aide des classes **File**/**Directory** et pour des pages HTML on utilise la classe **Text**.

8 Difficultés et améliorations

Durant toute la période de la réalisation de ce projet, nous avons rencontré plusieurs difficultés dont on peut citer :

- L'utilisation des bibliothèques de **Qt** n'était pas évidente, vu que cette bibliothèque est immense. Il fallait faire des recherches et bien lire la documentation pour bien profiter de ce qu'offre cette bibliothèque comme classes et méthodes.
- Un autre problème concerne la façon avec laquelle on stocke nos pages dans la mémoire cache et de pouvoir les repérer ensuite.
  La première approche consisterait de définir un identifiant à chaque page au moment de sa création en utilisant un entier qu'on incrémente. Cette solution a marché, mais il faut prendre des précautions au niveau du son repérage comme les pages sont plutôt caractérisées par leurs chemins présentés dans les requêtes.
  Pour résoudre ce problème, on a utilisé la classe **QHash** qui permet d'associer et mémoriser une clé à une valeur en prenant le chemin (URL) comme clé et la page correspondante comme valeur. Avec ça, on n'a plus le problème d'ambiguïté comme les pages sont bien associées à leurs chemins.

Notre serveur est bien fonctionnel et il répond à la quasi-totalité des exigences du cahier des charges. Il y'avait d'autres choses que nous aurions aimé ajouter pour l'améliorer encore plus, mais malheureusement nous n'avions pas assez du temps pour les mettre en place. Parmi ces améliorations, on cite :

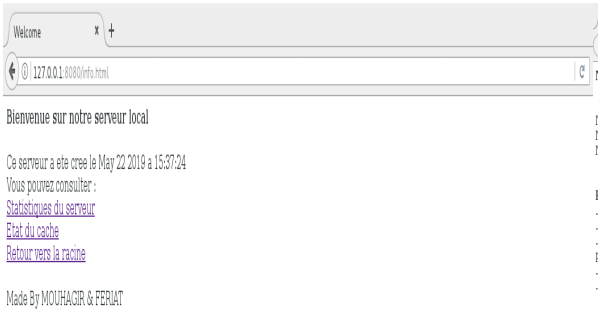
- Afin de bien protéger la page private et donner à l'admin plus de confidentialité pour les actions de maintenance, on pourra lui demander d'entrer un mot de passe à chaque fois il voulait faire l'une de ses actions de maintenance (vider le cache ou les statistiques, ...).
- Travailler sur le côté esthétique de nos pages pour un affichage plus agréable.
- Ajouter plus d'options pour la maintenance de notre serveur ainsi que développer une application graphique avec Qt pour cet effet.

9 Conclusion

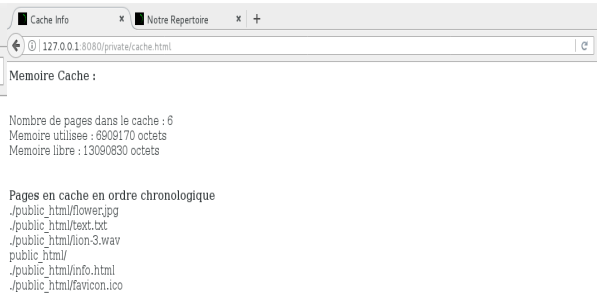
Ce projet nous a permis d'approfondir nos connaissance vis à vis le langage C++ et le fonctionnement d'un serveur HTTP. Bien qu'il soit tout à fait possible de le réaliser en un langage autre que le C++, la programmation objet est la manière la plus adéquate à ce genre de projet, car d'une part, la conception du projet en objet est plus

claire, rigoureuse et organisée. D'autres part, l'intérêt de l'objet est l'encapsulation des données, ce qui donne une grande maîtrise pour accéder à ces objets et les manipuler. Par ailleurs, les notions d'héritage et de polymorphisme permettent d'abstraire ces données. Le langage C++ fournit tous les outils pour faire ce qu'on vient de citer facilement et sans risque. D'où la raison de son utilisation.

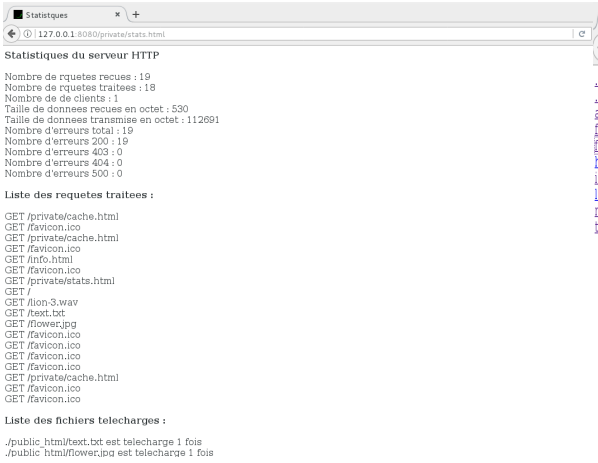
Voici quelques exemples d'utilisation en envoyant des différentes requêtes :



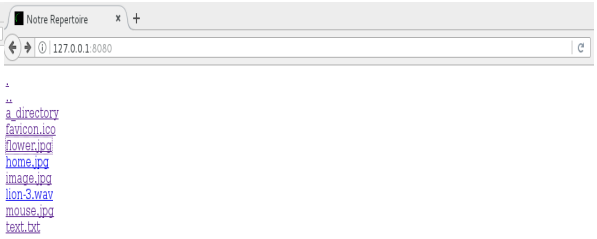
(a) Requête : */info.html*



(b) Requête : */private/cache.html*



(c) Requête : */private/stats.html*



(d) Requête : lister le contenu d'un dossier