Serveur http évolué

# Introduction

## 1). Objectifs

L’objectif de ce projet était de mettre en place un serveur http codé en C++. Un serveur http ou serveur web est un logiciel qui une fois héberger permet le transfert de fichier de http, le http étant un protocole de communication client-serveur.

Lors du premier semestre, dans le cadre du cours de réseaux, nous avions déjà été amené à étudier des serveur web simple permettant le traitement de requêtes simples. Cependant pour ce projet il s’agira de concevoir et d’améliorer ce type de serveur afin de lui inclure de nouvelles fonctionnalités

## 2). Cahier des charges

Ce serveur http devra pouvoir permettre à tout client de lire une page HTML stocké sur le serveur.

Il devra aussi comporter les fonctionnalités suivantes :

-Traitements des requêtes du clients (GET ou POST).

-Traitements des erreurs.

-Génération d’une page affichant les statistiques du serveur.

-Ouverture d’une fenêtre (QWidget) affichant ces mêmes statistiques.

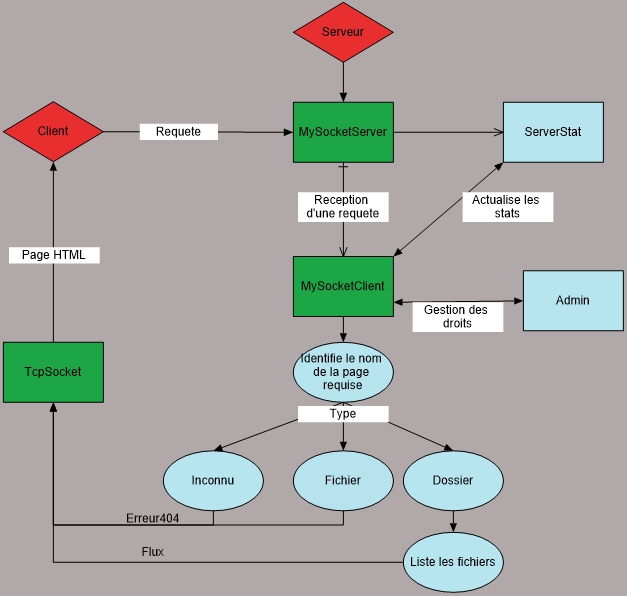
-Gestion du cache en mémoire RAM.

-Accès administrateur à une page Web permettant l’activation/désactivation du serveur ainsi que la gestion du cache.

La conception de ce serveur web et de ses fonctionnalités seront réalisés en langage C++ et en utilisant les classes proposées par la bibliothèque objets QT.

II/. Description de l’architecture et fonctionnement

1. Fonctionnement du serveur



Lorsqu’un client effectue une requête c’est la classe MySocketServer qui au travers de la classe dialog détecte cette requête. En effet lorsque le serveur détecte une demande de connexion en provenance d’un client la classe MySocketServer utilise une surcharge de la méthode incomingconnection() de la classe QTcpServer dont il hérite pour lancer le processus de traitement de la requête.

La classe MySocketServer génère une instance de MySocketClient qui s’occupera de traiter toutes les requêtes des différents clients :

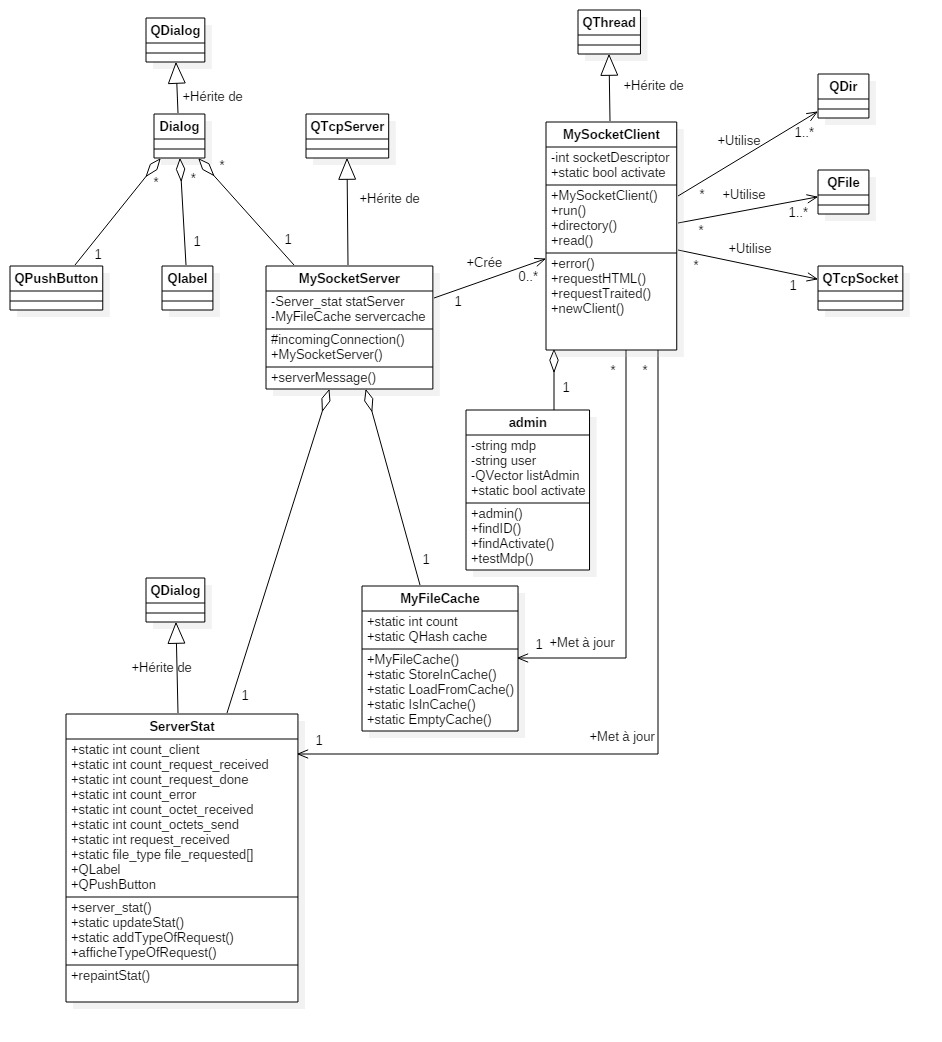
-La classe MySocketClient stocke la requête envoyée par le client, il identifie alors le type de requête (GET ou POST) ainsi que le nom du fichier requis ;

-Pour une requête GET si le fichier n’est pas stocké dans le serveur alors le serveur renverra une page « erreur 404 » ;

-Si le fichier est le nom d’une page HTML alors le serveur renverra cette page au client au travers d’une « TcpSocket » ;

- Si la requête concerne un dossier alors le serveur renverra une page HTML listant tous les fichiers contenus dans ce dossier. Si le client clique sur le nom d’un fichier il sera redirigé vers la page en question.

La classe MySocketClient gère aussi le droit d’administrateur des clients au travers de la classe admin. Elle met aussi à jour les différentes statistiques du serveur au travers de la classe server\_stat.

1. Architecture du serveur

*Figure 1 : Diagramme UML*

Ce Diagramme UML décrit l’architecture du serveur. Elle représente les différentes classes utilisées par le serveur ainsi que leurs liens entre elles :

-La classe dialog hérite de la classe QDialog de QTcreator et comporte un objet de la classe Qbutton, Qlabel et MySocketServer.

-La classe MySocketServer hérite de la classe QTcpServer et comporte un objet de la classe Server\_Stat, MyFileCache en tant qu’attribut privée. Elle comporte une méthode incomingConnection() qui gère les connexions des différents clients et qui crée une instance de la classe MySocketClient.

-La classe MySocketClient hérite de la classe QThread et comporte un objet de la classe admin en tant qu’attribut publique. Elle utilise les méthodes run() qui gère les requêtes des clients, directory() qui gère les requêtes concernant les dossiers. Cette classe utilise aussi les signaux error(), requestHTML(), RequestTraited(), newClient() et newStat(). Elle est aussi chargée de mettre à jour les objets de la classe MyFileCache et server\_stat.

-La classe admin stocke en attribut les identifiants des clients ainsi que la variable activate qui est défini en statique. Elle utilise les méthodes findId() qui collecte les identifiants des clients, findActivate qui vérifie l’activation/désactivation du serveur et testMdp() qui vérifie si les mots de passe sont justes.

-La classe MyFileCache stocke en attribut les fichiers sauvegardés dans le cache ainsi que leur nombre. Elle utilise les méthodes StoreInCache() qui stocke un fichier dans le cache, LoadFromcache() qui permet de charger un fichier plus rapidement depuis le cache, IsInCache() qui vérifie si un fichier est déjà dans le cache et EmptyCache() qui vide le cache.

-La classe server\_stat stocke en attribut toutes les données statistiques du serveur. Elle utilise les méthodes updateStat() qui met à jour les données de statistique et addTypeOfrequest.

III/. Description des fonctionnalités

1). Structures de base (Traitements des requêtes)

Lors de la récupération du code source d’initiale du serveur les fonctionnalités de base du serveur étaient déjà implémentées. En effet lorsque nous avons commencé le projet le serveur était déjà capable de traiter une requête GET d’un client afin de renvoyer une page HTML. Cependant pour que cette fonctionnalité soit réellement fonctionnelle nous avons dû rajouter l’envoi de l’entête HTML « HTTP/1.1 200 ».

Le traitement des requêtes est réalisé par la classe MySocketClient qui gère toutes les interactions avec les différents clients internet :

-On récupère le lien de communication avec le client ;

-On récupère la requête et sa taille avec la méthode readLine() ;

-On traite ensuite la chaine de caractère reçu afin de ne récupérer la commande de la requête, le nom du fichier ou du dossier demandé. On peut ensuite déterminer le chemin du fichier à partir du dossier « public\_html » ;

-Si le fichier existe on envoi la page HTML au client qui peut donc y accéder depuis son navigateur.

Ce code de base ne permet que de traiter les demandes GET de pages HTML par exemple il est impossible de traiter la demande d’un dossier ou téléchargement.

2). Gestion des dossiers

Grâce aux types QFile et QDir proposés par la bibliothèque Qt il est possible déterminer si le client demande un fichier ou un dossier.

Pour traiter la requête du client dans le cas d’un dossier nous avons créés un fonction directory() :

Tout d’abord la classe MySocketClient ouvre le dossier puis créée une QFileInfoList contenant le nom et le chemin des fichiers contenus dans le dossier. La fonction directory() prend ensuite en argument cette liste de fichier puis génère une page HTML. Cette page HTML affiche des liens cliquables renvoyons les pages HTML contenu dans le dossier. Pour générer la page chaque ligne du code HTML est écrite dans le fichier à partir de la fonction directory(), pour cela on a utilisé la classe QTextStream proposé par QT.

3). Statistiques du serveur

a. Page HTML de statistiques

Une des fonctionnalités rajoutées au serveur est l’utilisation de statistiques d’utilisation. Le client et les administrateurs doivent pouvoir avoir accès aux différentes statistiques d’utilisation du serveur :

-Le nombre de requêtes reçues par le serveur ;

-Le nombre de requêtes traitées ;

-Le nombre de clients connectés ;

-Le nombre d’erreurs détectées ;

-Le nombre d’octets reçus ;

-Le nombre d’octets envoyés ;

Toutes ces statistiques sont gérées par la classe server\_stat qui contient en attribut toutes les valeurs de statistiques. Tout ces attributs sont déclarés en static afin qu’ils soient mis à jour pour tous les clients. C’est la méthode updateStat() qui permet de mettre à jour et d’incrémenter ces attributs, cette fonction prend en argument un type énumération qui permet de choisir quel attribut de statistique incrémenter (NEWCLIENT, NEWREQUEST…). Il suffit d’appeler cette fonction au moment où une des statistiques varie pour la mettre à jour, par exemple au moment MySocketClient lance une connexion client (méthode run()) on fait appelle à la fonction updateStat() avec l’argument NEWCLIENT pour incrémenter le nombre de clients connectés.

Une fois les statistiques mises à jour, comme pour la gestion des dossier, la méthode statHTML() de la classe server\_stat va générer un fichier HTML qui va afficher les différentes statistiques du serveur.

Pour cette classe de statistiques un « log » a aussi été implémenté afin de garder une trace des différentes requêtes reçues par le serveur. Ce « log » est affiché sur la page HTML à l’aide d’une fenêtre déroulante afin d’éviter d’encombrer la page.

//Expliquer fonction addtyperequest

b. Widget de statistiques

De la même manière que pour la page HTML on cherche aussi à afficher les statistiques du serveur sur une interface graphique (Widget) en temps réel. Ces statistiques sont les mêmes que pour la page HTML donc le widget affiche les mêmes valeurs stockées dans les attributs de la classe server\_stat que la page HTML. Aucune nouvelle méthode n’a donc été créée pour mettre à jour les données statistiques de cette fonctionnalité.

Pour afficher un widget la classe server\_stat hérite de la classe QDialog qui permet d’utiliser QWidget pour créer des interfaces graphiques. Les données seront affichées avec QLabel et un bouton est créée pour quitter et fermer le serveur.

Pour que les données soient affichées en temps réel la méthode repaintstat de la classe server\_stat réactualise le texte du Qwidget affichant les données statistiques.

4). Gestion Cache

// CACHE

5). Gestion de la page administrateurs

a. Identification

Sur le serveur se trouve une page HTML « configuration » qui n’est accessible qu’aux administrateurs du serveurs possédant le bon identifiant ainsi que le bon mot de passe. Cette page de configuration doit permettre à un administrateur de pouvoir activer ou désactiver le serveur et de vider le cache. Lorsqu’un client tente de se connecter à la page de configuration il est automatiquement redirigé vers une page d’identification où il devra rentrer un identifiant et un mot de passe. Lorsque le client clique sur le bouton de validation un formulaire HTML contenant l’identifiant et le mot de passe est envoyé au serveur par une requête POST. La vérification des identifiants est géré par la classe admin qui validera ou non la connexion du client. La récupération de formulaire HTML est faite dans la classe MySocketClient où il est stocké dans un tableau QVector. Ce QVector est ensuité traité par la méthode findId() de la classe admin.

La méthode findId() fonctionne comme suit :

-Le tableau pris en argument est composé des lignes du formulaires HTML ;

-La ligne contenant le mot de passe et l’identifiant est construite comme ceci : « usr=*identifiant*&id=*motdepasse* » ;

-La méthode cherche dans ce tableau le mot « usr » pour déterminer quelle est la ligne contenant le mot de passe et l’identifiant ;

-On cherche ensuite les caractères « = » et « & » pour trouver l’emplacement du mot de passe et de l’identifiant ;

-Ces identifiants sont ensuite stockés pour être testé.

Lorsque le client valide sont mot de passe le serveur le vérifie avec la méthode testMdp() de la classe admin. Cette méthode renvoi 1 si le mot de passe est bon et 0 sinon. Dans le cas où le mot de passe est correct le client accédera à la page de configuration sinon il sera renvoyé vers la page d’identification.

b. Page de configuration

Une fois connecté à la page de configuration le client a deux possibilités : Activer/désactiver le serveur ou vider le cache. Pour réaliser ces deux actions deux boutons sont présents sur la page de configuration qui envoient un formulaire par requête POST au serveur. Le serveur vérifie ensuite si le bouton a été actionné de la même façon que pour les mots de passe : La méthode findActivate() de la classe admin récupère un tableau contenant les lignes de la requête et vérifie si il contient le mot « activate ». Si c’est le cas la méthode change la valeur de l’attribut static « activate » entre True et False c’est cet attribut qui est testé à chaque fois que le serveur doit renvoyer une page HTML à un client : Si l’attribut est à True alors le client peut accéder à la page s’il est à False non.

Pour le cache c’est la même méthode qui gère si le client à appuyer sur le bouton « vider le cache ». Si le tableau contenant le formulaire POST contient le mot « vider+le+cache » alors la méthode EmptyCache de la classe MyFileCache vide le cache.