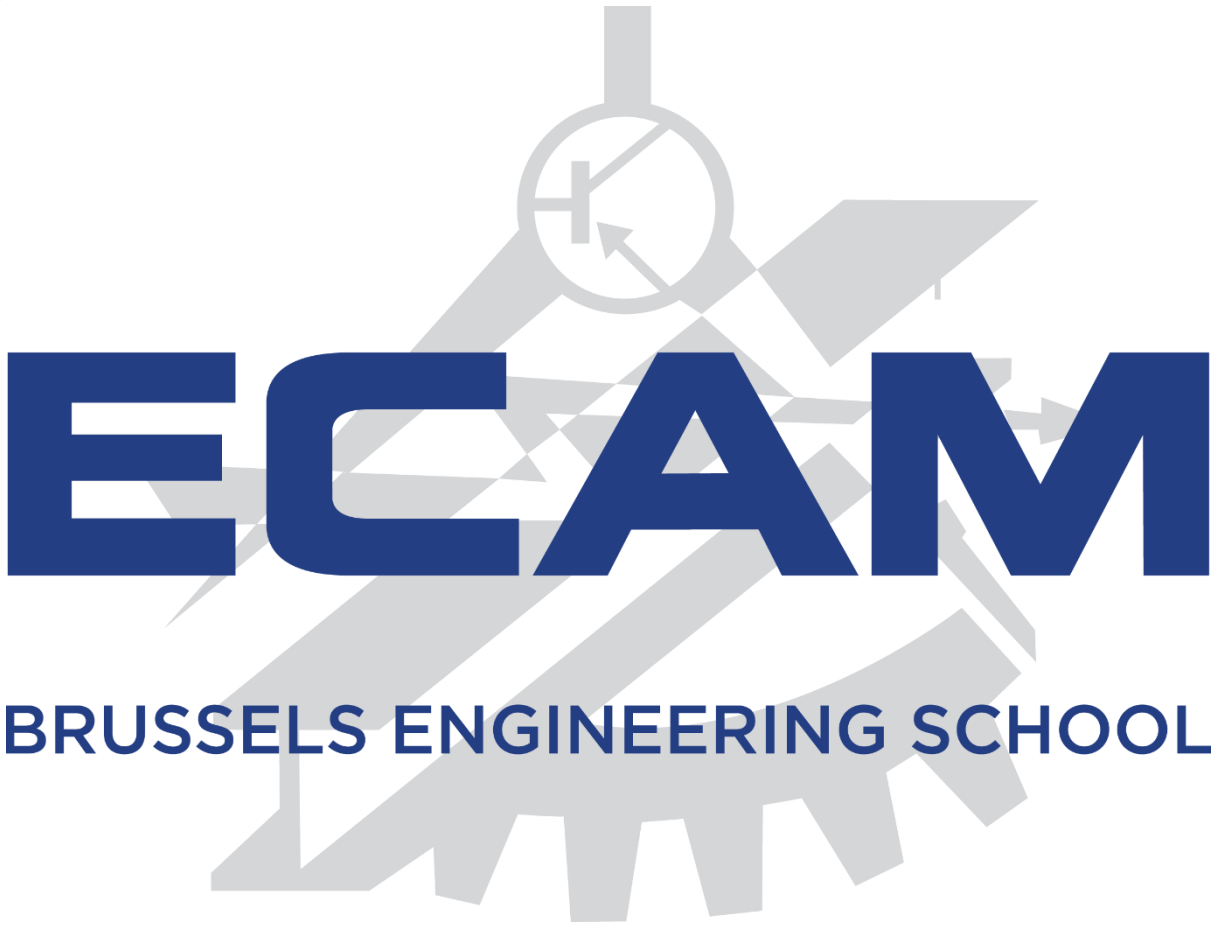


RAPPORT DE SCALABLE ARCHITECTURE

Mandelbrot

Gaëtan Herinne, Martin Sing, Thomas Vandermeersch

14/01/2022

Rapport de Scalable architecture

Mandelbrot

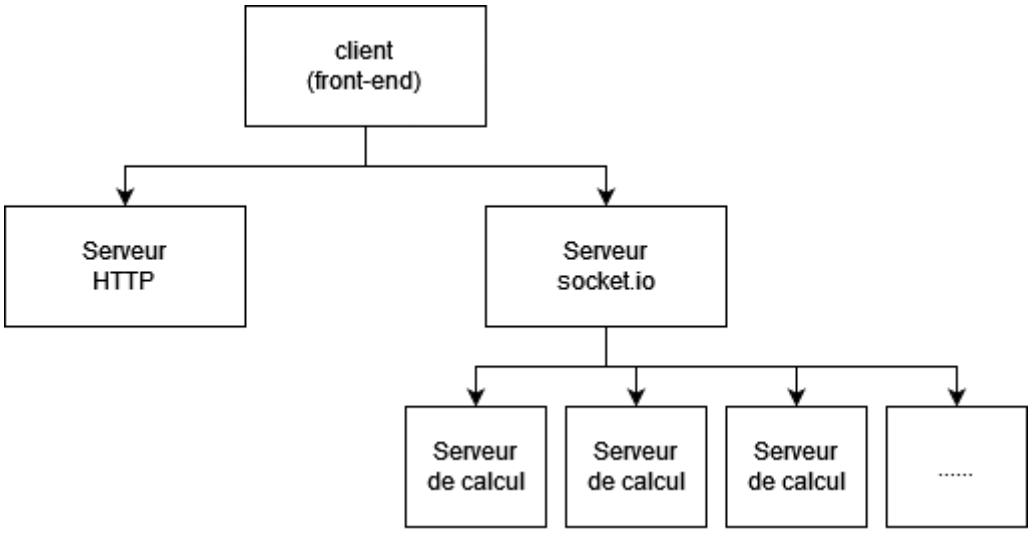
1 Introduction

Ce rapport présente le fonctionnement de notre projet pour le cours de “Scalabable architecture” donné en Master 2 informatique à l’ECAM. Ce projet vise à définir une stratégie de *load balancing* et à la mettre en application, avec l’objectif de déterminer si un point est situé dans le domaine de Mandelbrot et de dessiner cet ensemble.

2 Structure générale

La structure que nous avons adoptée est définie en trois parties principales. La première est la partie “client”, qui est composée de l’affichage directement en contact avec l'*end user*. La deuxième est la partie dédiée spécifiquement au *load balancing*. La troisième est la partie dont le rôle se limite au calcul.

Voici la structure présentée de manière schématique :



2.1 Chronologie

Concrètement, le parcours utilisateur se présente comme suit :

1) Le client se connecte au serveur HTTP ;
2) Le serveur HTTP renvoie une page HTML et un script (JS), permettant au client d’entrer un (ensemble) de nombre(s) complexe(s) afin de déterminer s’il(s) est(sont) compris dans l’ensemble de Mandelbrot ;
3) Le client entre sa requête et la confirme. Un script s’exécute alors côté client pour ouvrir une connexion de type *socket*. Ensuite :
    a. Si le client entre un seul point, ce point est envoyé par la connexion *socket*.

b. S'il entre plusieurs points, ces points sont envoyés un par un au travers de la connexion *socket*.

4) Le serveur *socket.io* reçoit ce(s) point(s) et le(s) redirige vers des serveurs de calculs ;
5) Les serveurs de calculs renvoient la réponse au serveur socket.io ;
6) Le serveur socket.io transfère cette réponse au client.

2.2 Caractéristiques générales des différents acteurs

Comme évoqué précédemment, notre projet suit une structure organisée autour de trois principaux éléments : le *front end*, le *load balancing* et le *back end*. Vous trouverez ci-après la présentation détaillée de ces différents éléments.

2.2.1 Client

- Navigateur Web (avec moteur Javascript)

2.2.2 Serveur HTTP

- Serveur Express (Node.JS)
- 2 routes
    - **./**
        - Renvoie un fichier HTML permettant au client d'accéder à une interface afin de vérifier si un point appartient au domaine de Mandelbrot ;
        - Renvoie un script (JS) permettant d'ouvrir un *socket* pour la requête.
    - **./multiple**
        - Renvoie un fichier HTML permettant au client de générer un *canva* d'une zone de l'ensemble de Mandelbrot ;
        - Renvoie un script (JS) permettant d'ouvrir un *socket* pour les requêtes (une requête par point).

2.2.3 Serveur socket.io

- Utilisation de la bibliothèque **socket.io** ;
- Permet d'ouvrir une connexion permanente avec le client ;
- Sert d'intermédiaire entre le client et les serveurs de calculs. C'est ici que se fait le *load balancing*.

2.3 Serveurs de calculs

- Serveurs Express (Node.JS) ;
- Notre solution comprend plusieurs serveurs de calcul permettant de répartir la charge ;
- Chaque serveur a une route unique :
    - **./inMandelbrot/ :real/ :imag/ :itt**
        - Les paramètres **real** et **imag** correspondent à la partie réelle et imaginaire du nombre complexe ;
        - Le paramètre **itt**, correspond au nombre d'itérations maximales qu'il faut utiliser dans l'algorithme.

3 Stratégie de LoadBalancing et justifications

Lors de nos recherches, nous avons appris que le *load balancing* se fait généralement grâce à un *reverse proxy*. Il redirige les demandes vers différents serveurs. La décision de redirection se fait selon des algorithmes prenant généralement en compte le **nombre de connexions ouvertes** sur un serveur et sa **vitesse**.

Dans notre cas cependant, ces algorithmes ne semblent pas optimaux. En effet, le temps de réponse dans notre problème dépend fortement du nombre complexe envoyé. Ainsi, comme nous considérons que les vitesses des différents serveurs sont identiques, il n'est pas toujours optimal d'envoyer la requête sur le serveur ayant le moins de connexions. Par exemple, il est tout à fait possible que nous ayons :

- Un serveur avec 1 connexion, qui calcule un nombre complexe avec 10000 itérations ; et
- Un serveur avec 3 connexions de chacune 10 itérations.

Le second serveur sera donc plus rapide. Malheureusement, nous ne pouvons pas prévoir cela à l'avance. C'est pourquoi nous avons implémenté notre propre *load balancer*.

3.1 Stratégie de load balancing

1. Le serveur de *socket* reçoit une requête et place celle-ci dans une queue ;
2. Un fichier JSON spécifie les serveurs disponibles (IP, Port, Statut) ;
3. Si un serveur est dans l'état "disponible", le *load balancer* effectue une requête HTTP [GET] vers celui-ci avec le premier élément de la queue ;
4. Le serveur de calcul exécute l'algorithme et répond au *load balancer*.

3.2 Justification de l'utilisation de Node

NodeJS permet d'exécuter du code Javascript côté serveur. Dans notre cas, l'avantage du Javascript est qu'il s'agit d'un langage asynchrone. Cela implique que, lorsque le *load balancer* va envoyer une requête GET à l'un des serveurs de calculs, le programme ne va pas être bloqué par l'attente de la réponse. Javascript à l'avantage d'être non bloquant.

3.3 Justification de l'utilisation des sockets

Le *socket* permet de garder la connexion entre le client et le serveur ouverte. Ainsi, si le serveur met beaucoup de temps à répondre, cela évite d'avoir une erreur de type "*Request Timeout*" coté client.

4 Conclusion

Notre projet a permis avec succès d'implémenter une stratégie de *load balancing*. Pour ce faire, nous avons utilisé une structure en trois parties, ainsi que le langage Javascript et NodeJS pour réaliser le *load balancer*. Cela permet à un utilisateur de vérifier si un nombre complexe fait partie de l'ensemble de Mandelbrot, ou de le dessiner dans un intervalle grâce à différents serveurs de calculs.

5 Annexes

5.1 Bibliothèques utilisées

- socket.io
  - Quoi ?
    - Envoie dans les deux sens
    - Garder une connexion ouverte
  - Pourquoi ?
    - Permet de garder la connexion ouverte le temps que toutes les requêtes soient exécutées
  - Documentation
    - https://socket.io/
- express (NodeJS)
  - Quoi ?
    - Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.
  - Pourquoi ?
    -
  - Documentation
    - https://expressjs.com/
- fs
  - Quoi ?
    - Bibliothèque qui permet de lire des fichiers
  - Pourquoi ?
    - Permet de lire le fichier JSON
- http
  - Quoi ?
    - Bibliothèque permettant de se connecter aux serveurs
  - Pourquoi ?
    - Faire des requêtes aux serveurs de calcul