INFO0010-4: Introduction to computer Networking

Seconde partie du projet

Olivier Moitroux Thomas Vieslet

Année académique 2017-2018

1 Introduction

Le but de ce projet est de développer un jeu Mastermind dont l'entièreté de la gestion se fait dans un serveur simulé par un programme implémenté par nos soins en Java. Ce serveur doit utiliser le protocole http et notre code doit suivre une série de conditions et critères. La manière dont notre code est organisé pour répondre à ces attentes est décrite ci-dessous dans la section Software architecture.

2 Software architecture

Pour construire notre code, nous avons démarré de notre classe Serveur du projet 1. En effet, la classe que nous avons construite ici n'est pas bien différente de celle dans le projet 1. La différence notable réside dans le fait que ici le WebServeur gère les connexions avec le concept de pool.C'est à dire que seulement un certain nombre de thread peuvent co-existersur le serveur. Un client supplémentaire devra attendre pour se connecter qu'un des thread de la pool soit libéré. Étant donné qu'un thread ne s'occupe que de consulter une requête et d'y répondre, un client ne peut monopoliser un thread indéfiniment.

Ensuite, comme dans le projet 1, le WebServeur instancie un objet de classe Worker qui gérer la coordination de la majeure partie des opérations. La première tâche de cette classe est d'envoyer la page web correspondante à l'utilisateur qui se connecte. Ceci est réalisé à l'aide de la fonction *send()* de la classe *httpManager* et ayant obtenu les données de la page avec la méthode *get_page_array()*. Cette page Web grâce à l'environement *noScript* et d'autres mécanismes du html fonctionne autant quand le browser à la capacité de lire *javaScript* que sans. Dans le cas où le browser ne peut lire *javaScript*, un menu déroulant remplace les bulles de l'autre cas pour permettre le choix de la combinaison au joueur. Sans *javascript*, au lieu d'avoir une page qui change dynamiquement en fonction des réponses du serveur, on a une page statique qui est envoyée par le serveur à chaque fois que l'utilisateur clique sur envoyer. Cette page statique est naturellement la version updatée de la page avec les bonne couleurs aux bons endroits.

Comme dit précédemment, l'envoi des pages et de toutes les réponses aux requêtes du client se fait par la fonction *send()* qui se trouve dans la classe *httpManager* (celle-si, comme son nom le laisse présager, s'occupe globalement de la gestion des requêtes http). L'envoi de ces réponses se fait utilisant un procédé appelé le *chunked encoding* qui consiste à envoyer l'information par bloque en spécifiant tout d'abord à chaque envoi la taille du bloc. Cette fonction *send()* est appelée dans la méthode *run()* du *Worker*.

Donc quel mécanisme, quel chemin est pris lorsque l'utilisateur fait une requête au serveur ? Le *Worker*, dans sa méthode *run()*, instancie un objet de type *httpManager* et dans cette classe se trouve une méthode qui s'occupe de la réception de requêtes : *receive().receive()* lit le buffer et puis sépare les différentes parties de la requête pour en extraire les informations (comme la méthode utilisée, la version d http et évidement le contenu). On détecte également la présence d'un cookie si il y en a un. Avec ces information, un objet de type *httpRequest* est instancié. Cet objet est une classe représentant une requête et contenant toutes les informations relatives à celle-ci. En réalité, cette classe pourrait s'apparenter à une structure de donnée.

Le *Worker* vérifie alors si le client possèdre déjà un cookie. Si c'est le cas, la fonction *retrieve_pending_game* est appelée. Celle si accède à une hashmap contenant tout les couples cookies/game existant. Une partie est donc stockée sous la forme d'un couple cookie/game où game est un objet de type *mastermind*. La fonction *retrieve_pending_game()* renvoie donc la partie correspondante au cookie.

Par contre, si la requête ne comporte pas de cookie. Il faut lui en créer un. Du coup, un nouveau cookie aléatoire mais différent de ceux déjà existant est créé dans la méthode *new_cookie()*. Ce cookie est ensuite envoyé à l'utilisateur via une requête en utilisant la méthode :*set_cookie_for_user()*.

Ensuite, toujours dans *run()*, différentes opérations sont réalisées en fonction de la méthode de requête.Dans le cas de la méthode *GET*, on check si l'utilisateur fait une requête particulière via *get_user_submit()*. Si c'est la cas, les couleurs envoyées par l'utilisateur sont lues et traitées via la fonction *compute_flags()* qui est contenue dans la classe *Mastermind*[1]. Le résultat est ensuite renvoyé à l'utilisateur. Si ce n'est pas le cas, c'est très certainement que l'utilisateur vient de se connecter et donc qu'il faut lui envoyer la page web *play.html* c'est donc ce qui est fait en utilisant le procédé que nous avons expliqué précédemment.

Pour le cas de la méthode *POST*, la même démarche est suivie mais sans le cas où il n'y a pas de requête car ce n'est pas possible. En effet, une demande de page est toujours réalisée avec une requête de type *GET*.

Finalement, la connexion est fermée et la partie est sauvée en l'état dans la map contenant les paire cookie/game afin de pouvoir recommencer la partie à la prochaine requête de l'utilisateur.

Évidement, toute une série de systèmes de gestion d'erreurs est mise en place pour traité des différents problèmes pouvant survenir entre les entités de ce programme.

3 Multi-thread coordination

Tout d'abord, il est bon de noter que la principale différence avec le projet 1 est que le concept de pool est ici intégré. Comme dit précédemment, ce concept impose un nombre maximal de threads tournant sur le serveur. Ceci permet à ce dernier de ne pas être trop surchargé.

Comment avons nous développé ce concept avec l'utilisation de cookies ? Une threadPool est créée avec un nombre maximum de threads rentré en argument. Ensuite dans une boucle infinie, le serveur écoute sur le port 8015 et attends une connexion. A chaque javaSocket accepté, un thread est attribué à partir de la pool (via l'instanciation d'un objet de classe *Worker*) à condition que le nombre de threads tournants sur le serveur ne soit pas le nombre maximal de thread. Si une demande de connexion supplémentaire est faite, il faudra attendre que thread soit fermé pour que celle-ci puisse s'établir. Néanmoins, les sockets sont fermés à la fin de chaque connexion (donc dès que la requête à été traitée) laissant une place disponible dans la pool. Il est donc possible de gérer plus de parties que le nombre de threads disponibles, à condition que les requêtes ne soient pas simultanées. L'accès à la hashmap *pending_game*

1. Cette classe contient les méthodes permettant le calculs du nombre de bonnes couleurs à la bonne et à la mauvaise place, l'historique des essais et des flags ainsi que la combinaison secrète, globalement toutes les infos relatives à une partie

statique et stockant les parties en cours, est protégé grâce au mot clef *synchronized* afin d'éviter que plusieurs instances de *worker* n'y effectuent des modifications.

4 Limites

Nous avons essayer de construire notre code de manière modulable et de gérer les exceptions et erreurs au mieux. Notre code nous semble globalement plutôt robuste et efficace dans son utilisation. Notons que notre programme ne fonctionne pleinement que sur firefox, la réception de la page web sur chrome restant toujours instable (seule une partie de la page web est reçue, la longueur du code reçu étant assez aléatoire). Le formatage du chunck encoding semble pourtant bon.

5 Améliorations

Même si notre code est parfaitement fonctionnel sur Firefox la première amélioration est sans aucun doute la compatibilité étendue à d'autres naviguateurs web. L'organisation de la classe *httManager* pourrait peut-être être améliorée. On aurait pu imaginer remplacer les méthodes *send()* par une seule méthode prenant en argument une *httpRequest* mais des lignes de codes supplémentaires auraient été nécessaires du côté du *worker*, sans compter que cette classe serait devenue plus dépendant du protocole http. Au final, nous avons donc opté pour une série de fonction publiques *send()*, appelant chacune une fonction générale propre à la classe, *send()* présentant plus d'arguments et s'occupant de l'envois effectif de la page via http. La description des codes d'erreurs http reste aussi obscure. Nous avons donc essayer de les placer aux bon endroits dans le code mais la distinction entre Method Not Implemented et *Method Not Allowed* n'est pas évidente, étant donné que si une méthode n'est pas implémentée, elle n'est forcément pas permise ....