Quel est ce pokemon ? Edition thread et réseau

Rapport de projet

Antoine BERTHION (000566199) Romain CROUGHS (000572261)
Lucas VAN PRAAG (000569535)

Année 2023-2024

Table des matières

1 Introduction

Dans le cadre du cours de systèmes d'exploitation (*INFO-F201*), nous avons dû développer deux programmes qui communiquent grâce a des sockets, via le protocole TCP/IP. Le premier programme, le *serveur*, recoit des images sur son socket, et se charge de trouver l'image la plus similaire parmi une multitude d'images présente dans le fichier `img/`. Le deuxième programme, le `client`, peut se connecter au socket du serveur, et va lire son entrée standard, récupérer des chemins vers des images, et envoyer ces dernières sur le socket du serveur. Il nous a également été imposé d'utiliser de manière judicieuse les tâches concurrentes du côté serveur via les processus légers (*threads*).

Le but de ce projet est donc de pouvoir mettre en exécution les mécanismes de communication et de concurrence mis en place par le système d'exploitation Unix, en l'occurrence les sockets et les threads dans le standard `Posix`.

Une fois le code source compilé, les deux exécutables de ce projet auront les noms suivants :

— `img-search` pour le programme serveur

— `pokedex-client` pour le programme client

Dans ce rapport, nous allons d'abord définir le protocole de communication qui sera utilisé pour que le serveur puisse communiquer avec les différents utilisateurs. Dans le chapitre 3, nous

décrirons en détail le fonctionnement du serveur. Finalement, nous expliquerons comment le client peut envoyer de manière fiable des images vers le serveur et récupérer les réponses.

2 Protocole de communication

Comme dit dans l'introduction, nous allons utiliser le protocole TCP pour toutes les communications, afin d'avoir une communication fiable. Toutes ces communications s'effectueront sur le port 5555. Le client peut choisir l'adresse à laquelle il souhaite se connecter afin d'assurer la versatilité du programme et le serveur écoute toutes les adresses qui pourraient essayer de se connecter à lui.

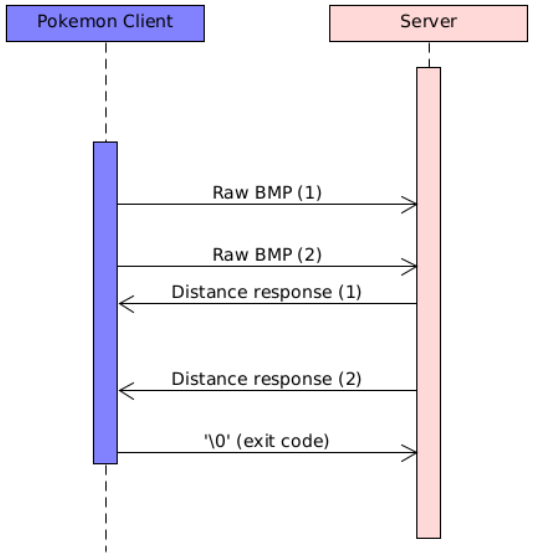

FIGURE 1 – Exemple d'une communication entre un client et un serveur

Comme décrit dans la figure 1, une fois que la connexion est établie entre le client et le serveur, le client peut envoyer ses images bitmap. Une fois que le serveur a reçu l'image, il va la traiter, comme décrit dans la section 3, et puis va renvoyer réponse sous forme d'une chaîne de caractère. Pour marquer la fin de la communication entre un client et le serveur, le client va envoyer un caractère nul ('\0') qui sera alors interprété par le serveur comme une fin de connexion.

3 Côté serveur

Le programme **server.c** agit comme le côté serveur de notre projet **pokemon-network**. Nous avons utilisé la configuration suggérée par les consignes pour les différents paramètres du socket que nous avons mis en place. Le port **5555** est ouvert sur toutes les adresses de la machine exécutant le serveur (**INADDR_ANY**) et écoute jusqu'à 10 requêtes avant d'accepter lesdites connexions.

Une fois acceptée, le serveur affectera le traitement de la demande à un thread puis remettra son socket sur écoute pour la prochaine requête. Tant que le client reste en connexion avec le serveur (représenté dans le code par la variable **communicationStatus**), le traitement et, par extension, ce thread restera actif.

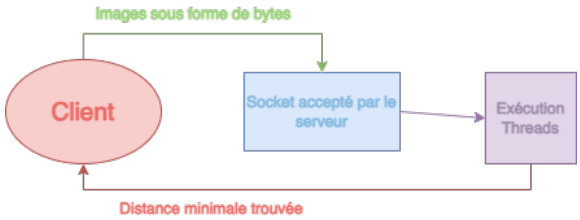

FIGURE 2 – Traitement type d'une requête du client sur le serveur.

Afin de lire sur les images sur le socket, nous devons nous assurer que le client envoie des images sous forme d'une série de bytes (et non de chemins d'accès) terminée par le caractère '\0'. La limite suggérée par les consignes pour la taille maximale des images envoyées sur le socket accepté par le serveur est de 20kb. Si elle excède 20kb, le client en est notifié et l'exécution peut se dérouler comme prévu initialement. Afin de vérifier que cette image ne dépasse pas la taille autorisée, notre buffer est intentionnellement légèrement plus grand que 20Kb, et si 20001ième octet du buffer n'est pas vide, cela signifie que le client a passé une image trop lourde. Si l'image est valide, alors un thread est associé à l'exécution de la demande particulière.

Pour chaque requête d'image, l'image de référence sera hashée à l'aide de la librairie **imgdist.h** afin de servir de référence. Ensuite, chacune des images contenues dans *'./img'* seront hashée chacune à leur tour afin de pouvoir trouver la plus grande proximité avec l'image référence.

Afin de rendre le processus plus rapide et de fluidifier le traitement, des threads sont mis à disposition de ces opérations de hash. Il faut donc faire une bonne distribution, plus ou moins équitable, de ces opérations de hash afin de favoriser une exécution fluide. La distribution choisie consiste à envoyer en alternance une image sur un thread et la suivante sur l'autre thread. Nous utilisons pour ce faire une variable de contrôle qui s'incrémente en prenant des valeurs paires et impaires.

Les résultats de chacun des threads sont ensuite comparés entre eux afin d'obtenir la plus petite valeur. Celle-ci est ensuite écrite dans un buffer puis envoyée via le socket au client à l'aide de la fonction **sendResult**, qui consiste en un write écrivant un résultat particulier dans le cas où la distance trouvée serait de -1 (erreur de hash).

Dans la figure 3, nous pouvons constater que le premier caractère de la réponse est l'*id* de la réponse. Cela permet au client de savoir quelle image est concernée lors de la réception d'une réponse.

FIGURE 3 – Format d'une réponse pour le client

4 Côté client

Le programme **client** se charge d'envoyer séquentiellement des images au programme serveur et récupère ensuite les résultats avant de les afficher sur la sortie standard (*stdout*).

Pour son bon fonctionnement, le programme va initialiser diverses variables afin de s'assurer du nombre d'images envoyées, du nombre d'images récupérées ainsi que du nombre de résultats affichés sur la sortie standard. Ces résultats sont stockés dans un tableau de pointeurs vers char (chaînes). Une limitation imposée par cette pratique mais nécessaire pour conserver l'ordre d'affichage des réponses est qu'il n'est pas possible d'envoyer plus de 256 images par client sans devoir relancer ce dernier.

Afin d'assurer la réception ordonnée des réponses, le client crée deux processus légers (*threads*) supplémentaires nommés **threadCatcher** et **threadPrinter** dont les fonctions respectives sont de récupérer les réponses du serveur et d'ensuite les afficher sur la sortie standard.

La récupération des réponses du serveur se fait selon une boucle s'exécutant comme suit : tant que l'utilisateur continue d'utiliser l'entrée standard et que toutes les réponses des images déjà envoyées n'ont pas encore été reçues, la boucle continue et le thread reste actif. Chaque réponse reçue correspond à un traitement d'une image, et chaque traitement est différencié par un ID (renseignant alors l'ordre d'envoi du client au serveur). Lorsque la réponse est lue à l'aide d'un *read*, l'ID est récupéré et correspond à l'indice auquel sera stocké la réponse dans le tableau utilisé à cet effet. Le compteur d'images récupérées est alors incrémenté, et la boucle recommence.

Pour l'affichage des réponses, une boucle permet au thread de rester actif tant que toutes les requêtes envoyées au serveur n'ont pas été affichées (une fois leur réponse récupérée) et que l'utilisateur n'a pas fermé l'entrée standard. L'impression à l'écran se fait selon le compteur d'images affichées, qui n'augmente que lorsque la réponse logiquement suivante, par ordre de requête, a pu être traitée. Ce compteur renseigne l'indice auquel accéder dans le tableau contenant les réponses.

Finalement, lorsque l'utilisateur décide de fermer l'entrée standard ou a envoyé 256 requêtes au serveur, le thread principal attend que le **threadCatcher** ai finit de récupérer les réponses de toutes ces requêtes avant de le terminer. Le **threadPrinter** finira alors son exécution en s'occupant des derniers affichages avant de se terminer lui aussi. L'espace occupé par les réponses dans le tableau de pointeurs vers char est alors libéré.

4.1 Gestion des signaux

Nos signaux sont gérés de la manière suivante du côté serveur :
— SIGINT est géré comme une fin d'entrée standard, c'est à dire que nous allons attendre que toutes les réponses soient envoyées pour éteindre le programme
— SIGPIPE ferme le programme de manière classique

5 Conclusion

A l'entame de ce projet, nous avons pu voir les nombreuses possibilités qu'offrent les communications en réseau et la concurrence à l'aide des processus légers.

Les threads permettent au serveur de pouvoir gérer plusieurs connexions en même temps, et ceci a un coût bien moindre qu'une création de processus.

L'utilisation de threads permet aussi au client de répartir les différentes tâches qui lui sont attribuées, améliorer la latence de réponse à l'utilisateur, et des mécanismes sur les signaux assurent la stabilité du programme, même en cas de fermeture précoce.