TEINTURIER Théophile

IE4-I92

QAEZE Noël

IE4-I5

Projet d’Info4B

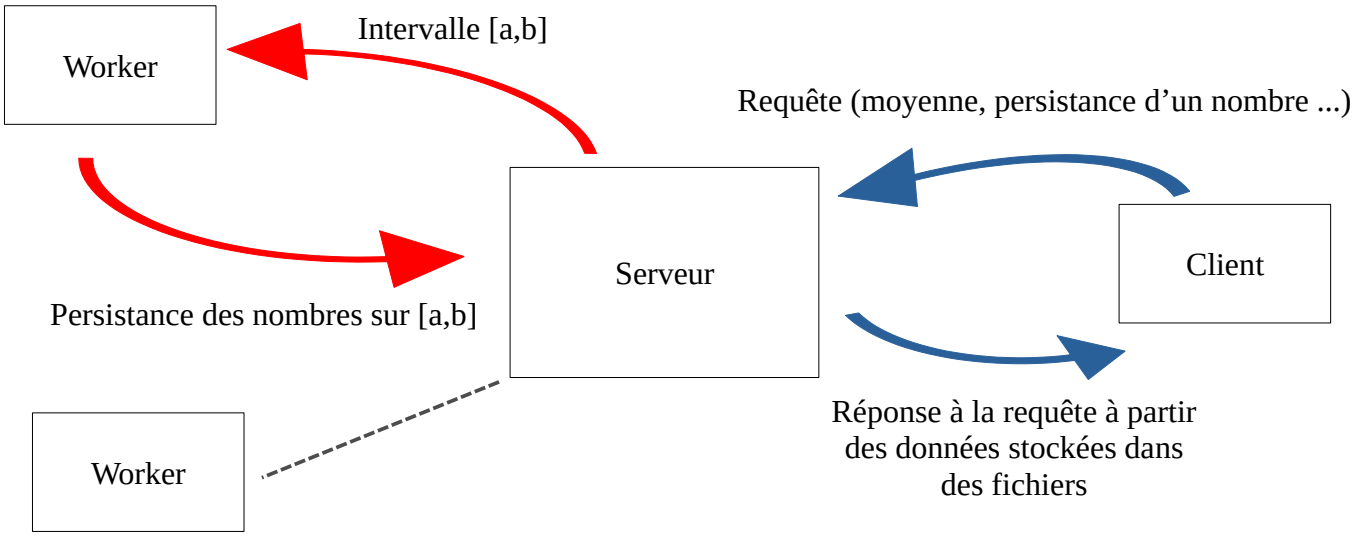
Calcul distribué de la persistance des nombres

Table des matières

Description générale de l’application

Le but de notre application est de générer un serveur capable de stocker la persistance multiplicative de chaque nombre. Mais avant de pouvoir stocker en mémoire cette information, le serveur aura besoin de la calculer. Un premier problème parvient donc dans le fait que cela prendra beaucoup de temps pour obtenir suffisamment de résultats. Ainsi, nous utiliserons des « workers » qui calculeront à la place du serveur tous ces éléments. Analysons d’un peu plus près le fonctionnement de ces workers. Tout d’abord, le serveur va envoyer à un worker un intervalle. Ce dernier va alors calculer la persistance de tous les nombres compris dans cet intervalle. Ensuite, il va retourner au serveur toutes les données qu’il a calculées pour que ce dernier puisse les stocker dans des fichiers. Comme vous l’aurez compris, le serveur servira juste de « gestionnaire » (dans cette partie de l’application) où il aura pour rôle de distribuer des intervalles à tous les workers disponibles (de façon à ce que deux workers ne travaillent pas sur les mêmes nombres) et de stocker le tout dans des fichiers.

Il ne nous reste plus qu’à parler de l’accès à ces données. Pour ce faire, nous avons mis en place un système pour que les clients puissent se connecter au serveur et obtenir des informations. Notons que comme pour les workers, il est possible d’avoir plusieurs clients connectés en même temps. Ils peuvent demander plusieurs informations comme la persistance d’un nombre, la moyenne (sur un intervalle précis ou sur l’intégralité des nombres déjà calculés)... Le serveur répondra ensuite aux requêtes posées à partir des données qu’il a stockées. Pour vous résumer la situation, voici un schéma qui relie les éléments vus précédemment :

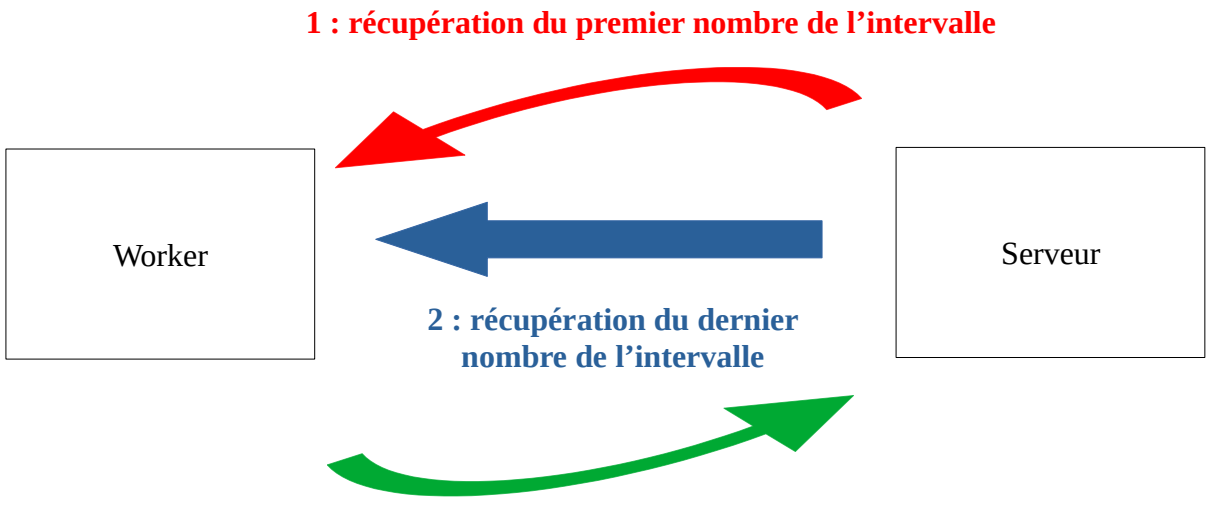


Description avancée de l’application

Maintenant que, le concept global de l’application a été abordé, nous vous proposons de regarder d’un peu plus près les différentes interactions entre le serveur, les workers et les clients, mais aussi la façon de stocker les données, le fonctionnement des workers... Ainsi, vous pourrez mieux comprendre le programme de nos différentes classes quand nous aborderons ce sujet un peu plus tard.

Interaction Serveur-Worker

Commençons par l’interaction entre le serveur et un worker. Nous avons vu précédemment que le serveur envoyait un intervalle au worker pour qu’il puisse faire les calculs à sa place. Pour ce faire, nous avons décidé de procéder de la manière suivante : le serveur envoie un message au worker comportant uniquement le premier nombre de l’intervalle puis il envoie le dernier nombre de l’intervalle (dans un autre message). Le worker va par la suite calculer la persistance des nombres de cet intervalle. Pour stocker ces données, il utilisera une « hashtable ». Notons aussi que le worker calcule au fur et à mesure la moyenne des nombres de cet intervalle (le numérateur de la moyenne) et stocke dans un tableau le nombre d’occurrences des persistances de l’intervalle. À la fin de son traitement, il renverra ces données sous forme d’un objet « Message » (que nous verrons un peu plus tard) pour que le serveur puisse les stocker. Voici un schéma qui résume la situation :



Stockage des données et interaction Serveur-Client

Parlons maintenant du stockage des données. Pour cette application, nous avons décidé de répartir toutes ces données dans différents fichiers. Comme nous avons pu le voir dans le point précédent, nous stockons les valeurs des persistances de chaque nombre dans des hashtables. Pour vous résumer la situation, nous avons décidé de stocker ces hashtables dans des fichiers (placé dans un dossier nommé « Fichier »). Notons que ces fichiers sont stockés sur la machine où le serveur est lancé et que chaque hashtable comporte les informations de 100 000 nombres. Pour pouvoir nous retrouver facilement parmi le grand nombre de fichiers que pourrait contenir notre application, nous avons décidé de leur donner des noms logiques. Ils comporteront un indice basé sur les nombres qui leur sont associés. Par exemple, le premier fichier s'appellera « 0Cm_1Cm.ser » car il comportera les nombres de 0 (inclus) à 100 000 (exclus), le deuxième fichier « 1Cm_2Cm.ser » …

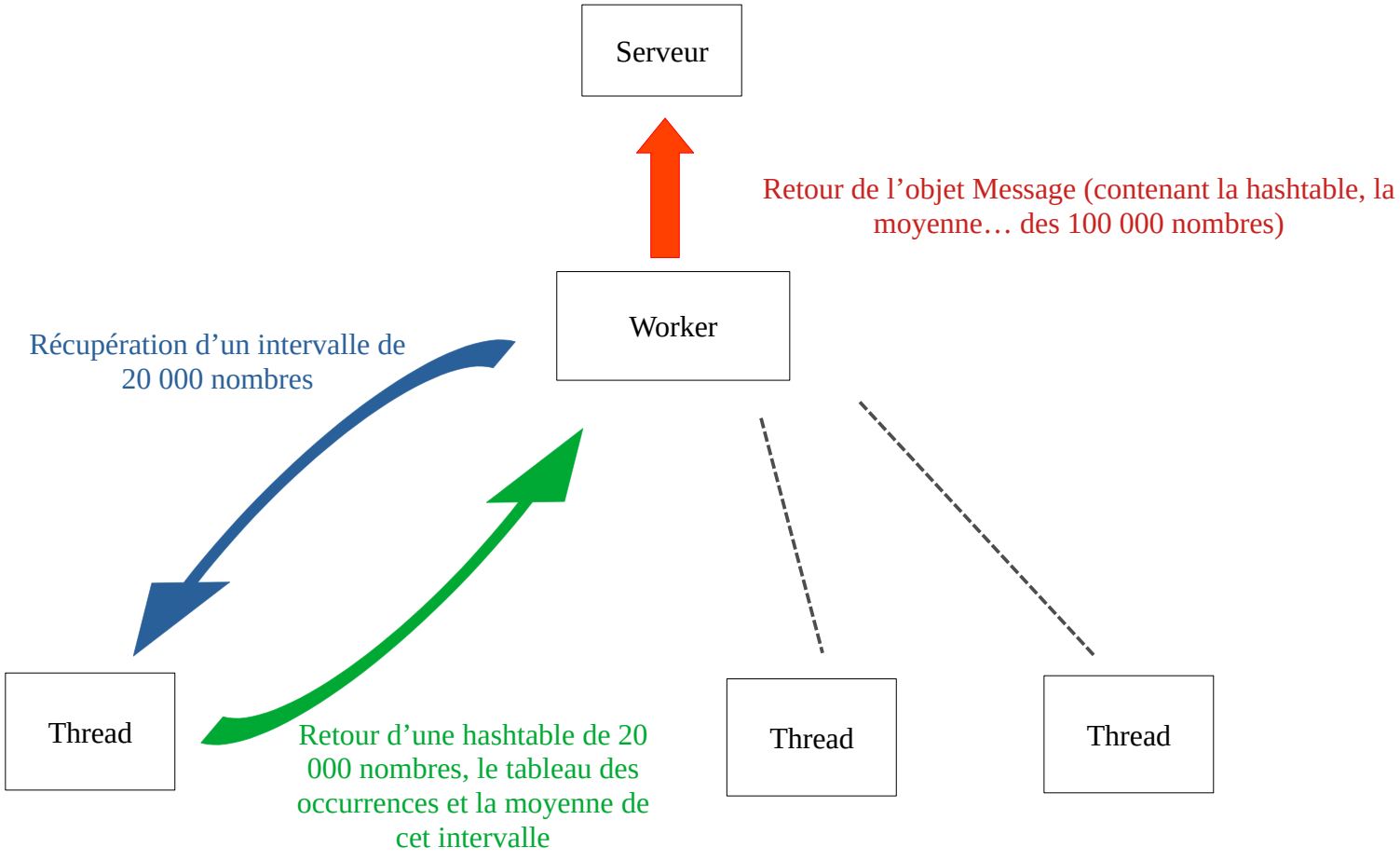
Ainsi, notre serveur ne sera pas obligé de garder en RAM l'intégralité des données pour pouvoir répondre aux besoins des clients. Attardons-nous un peu sur l'interaction de ces derniers avec le serveur. Le principe repose sur un échange de messages entre eux. Le client rentre les commandes qui répondent à ses attentes, le serveur fait son traitement et le renvoie (toujours sous forme de message) pour que l'utilisateur puisse en tirer des conclusions. Comme vous l'aurez compris, le serveur s'occupe intégralement de cette partie du traitement, même s'il doit calculer une moyenne sur un intervalle spécifique (à la place d'un worker). Nous expliquons ce choix technique par le fait que si aucun worker n'est disponible, personne ne peut gérer la demande. Nous reverrons un peu plus ce point lorsque nous parlerons de la classe Serveur. Pour terminer, nous justifions notre utilisation des hashtables par le fait qu'elles sont très pratiques pour accéder rapidement à une valeur souhaitée à partir de sa clé, ce qui amène à une meilleure performance au niveau de la rapidité d'exécution du serveur.

Revenons au stockage de nos données. Nous avons décidé de stocker le tableau du nombre d'occurrences des persistances dans un fichier « tab.ser ». Ce dernier va nous permettre d'initialiser (s'il existe) notre serveur dans le cas où on relancerait le serveur après l'avoir arrêté tout en gardant les données déjà stockées. Le stockage du tableau et des hashtables se fait à chaque fois qu'un worker renvoie des données. Notons que notre serveur garde le tableau en mémoire, mais aussi la moyenne de la persistance de l'intégralité des nombres traités (cette dernière est facilement retrouvable à partir du tableau dans le cas où on relancerait le serveur).

Fonctionnement des Workers

Passons maintenant à l'analyse du fonctionnement de nos workers. L'utilisation de ces derniers nous permet de décupler la puissance de calcul en utilisant une autre machine. Nous avions déjà vu que le serveur envoyait un intervalle au worker pour qu'il puisse faire son traitement dessus. Mais lancer un programme qui calcule toutes les persistances de ces nombres une à une n'a pas réellement d'intérêt. En effet, dans ce cas, nous n'utilisons pas tous les processeurs de la machine du worker. Nous avons donc opté pour une stratégie plus intéressante. Notons tout d'abord que le serveur fournira toujours un intervalle de 100 000 nombres pour se simplifier la tâche. En effet, dès qu'un worker retournera ses données au serveur, ce dernier pourra directement stocker la hashtable dans un fichier (car elle respectera les règles vues dans le point précédent).

Pour pouvoir mettre en place cette stratégie, nous avons décidé d'utiliser 5 « threads » par worker. Ce dernier va découper en 5 son intervalle (soit 20 000 nombres par part) pour que chaque thread fasse le traitement sur l'une de ces parts. Les threads vont retourner des hashtables au worker afin que ce dernier les fusionne et renvoie au serveur une hashtable avec les 100 000 nombres et leur persistance (elle est contenue dans l'objet Message.). Comme pour les points précédents, le schéma suivant illustre la stratégie employée :



Serveur

Maintenant que la majeure partie de nos idées vous a été présentée, nous pouvons vous expliquer concrètement le fonctionnement de notre application. Commençons par la classe Serveur.

Descriptions des attributs de la classe Serveur

Notre classe Serveur comporte une douzaine d'attributs. On peut répartir ces attributs en 3 catégories principales si on ne compte pas l'attribut « port » qui comme son nom l'indique sert pour l'identification du port du serveur. Tout d'abord, nous avons la catégorie des attributs qui servent à la gestion des connexions worker-serveur et client-serveur. Nous utilisons ces derniers afin de limiter le nombre de connexions au serveur. Ensuite, nous avons la catégorie « Affichage », cette catégorie d'attributs sert à un affichage détaillé des différentes opérations effectuées auprès du serveur. Nous pourrons étudier cet affichage d'un peu plus près, dans une partie qui lui sera entièrement dédiée. Enfin, nous avons la catégorie des attributs utilisés lors du stockage des données. Elles serviront surtout à la communication entre les clients et le serveur afin de répondre aux commandes telles que le calcul de moyenne.

Connexion des clients et des workers

Passons maintenant à la connexion des clients et des workers. Dans un premier temps, le serveur ouvre un unique flux via un « Socket », et surveille en continu les entrées dans le flux. Le principe est le suivant : lors de leur connexion au flux les workers et les clients envoient un premier message indiquant leur nature, c'est-à-dire si l'objet connecté est un worker ou un client. Puis, le serveur a la possibilité d'attribuer la gestion d'un flux à un Thread spécifique en fonction de l'objet qui s'y est connecté. Une fois qu'un flux se voit attribuer un Thread, le serveur en recréé un nouveau puis attend une nouvelle connexion. Le serveur dispose de deux « Thread Connexion » de nature différente, pour les workers, on aura les threads « ConnexionWorker » et pour les clients, les « ConnexionClient ».

Thread ConnexionClient

Passons maintenant à la classe ConnexionClient qui s'occupe de la communication entre le serveur et les clients. Elle nous permet de gérer un unique client. Passons rapidement en revue les différents attributs de la classe. Elle possède un attribut de type Socket (s) pour nous connecter au serveur, un de type BufferedReader (reader) et PrintWriter (writer) pour pouvoir recevoir des messages du client et lui en envoyer. Pour finir, on a un attribut de type String (qui sert uniquement à l'affichage du serveur).

Maintenant, que la connexion a été établie, il ne reste plus qu'à s'occuper des demandes du client. Dans une boucle « while » (qui s'arrête quand le client se déconnecte), on commence par récupérer la demande du client à l'aide la variable reader (il y a plusieurs propositions qui s'affichent sur son terminal) pour pouvoir effectuer les traitements nécessaires. Si le message correspond à « END », on sort de la boucle while, on ferme les différents attributs si besoin (s, reader, writer). Sinon, nous pouvons récupérer 3 messages différents qui nous permettent de savoir si l'utilisateur a besoin d'une moyenne, de la persistance d'un nombre, ou encore de la proportion des occurrences d'un ordre sur l'ensemble des nombres déjà calculés. Grâce à un switch, on appelle la fonction correspondante (Moyenne, Persistance, Proportion). Notons que ces fonctions sont des fonctions de la classe ConnexionClient. Prenons l'exemple de la fonction moyenne pour vous expliquer leur fonctionnement. Dans cette dernière, on commence par récupérer un deuxième message pour savoir si le client veut la moyenne sur la totalité des nombres déjà calculés ou sur un intervalle spécifique. Dans le cas d'une moyenne sur la totalité des nombres, on récupère cette donnée grâce à la méthode getMoyenneTotale de la classe Serveur avant de la renvoyer au client avec l'attribut writer. Dans le cas où l'utilisateur choisirait l'intervalle, on récupère les premier (a) et dernier (b) nombres de ce dernier. Puis, on calcule la moyenne à partir de la fonction MoyIntervalle(a,b). Pour vous résumer le fonctionnement de cette dernière, elle somme toutes les valeurs des ordres de ces nombres. Pour ce faire, elle récupère au fur et à mesure les hashtables stockées dans nos fichiers afin de pouvoir faire ce calcul. Elle ne récupère pas toutes les hashtables au début ; à chaque fois que le nombre qu'elle traite n'est plus sur la hashtable courante, elle remplace cette dernière par la suivante. Elle traite les nombres dans l'ordre croissant. Enfin, elle divise le résultat par le nombre de valeurs de l'intervalle. Pour finir avec notre fonction Moyenne, elle renvoie la valeur obtenue au client.

Les fonctions Persistance et Proportion sont similaires à la fonction Moyenne (on commence par récupérer des informations plus précises avant de faire notre traitement.). Nous avons donc décidé de ne pas expliquer leur fonctionnement. Pour en finir avec la classe ConnexionClient, notons qu'il existe une fonction getStr qui nous sert seulement dans l'affichage du serveur.

Thread ConnexionWorker

Lors de la création du Thread « ConnexionWorker », le serveur lui attribut la gestion d'un flux de type « Socket » auquel un worker y est connecté. Le rôle de ce thread sera de gérer les intervalles dont le worker doit calculer les persistances. Comme il a été expliqué précédemment dans la partie du stockage des données, chaque intervalle aura une taille de 100 000 nombres.

Dès sa connexion, le ConnexionWorker reçoit de la part du worker un nombre généré aléatoirement qui représente le nombre d'intervalles que va traiter le worker. Ainsi, la première

tâche du thread « ConnexionWorker » consiste à communiquer au worker l'intervalle dont il doit calculer les persistances. Toutefois, nous devons prêter attention au fait que le worker ne calcule pas le même intervalle qu'un second worker. Pour cela, on utilise la fonction Intervalle de la classe Serveur.

Dès lors que le calcul des persistances terminé, le ConnexionWorker réceptionne la liste des persistances envoyée par le worker. Il lui reste donc plus qu'une tâche à accomplir, qui sera de stocker cette liste sur le disque en faisant appel à un thread « ThreadStockage ».

Thread ThreadStockage

Situé dans le même package que le Serveur, le thread « ThreadStockage » a pour unique et simple rôle, de stocker sur le disque la liste des persistances qui lui est donnée par le ConnexionWorker. Avant de s'intéresser au stockage des données, nous allons revoir plus en détails comment ces dernières sont transmises d'une classe à une autre.

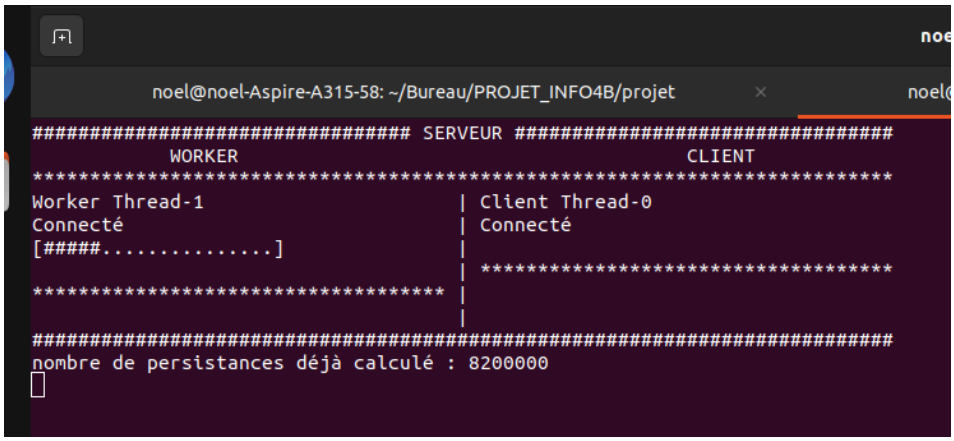
L'échange d'objet entre classe s'effectue via l'utilisation d'un « ObjectInputStream » puis d'un « ObjectOutputStream ». Au lieu d'envoyer directement la liste de type Hashtable au ThreadStockage, nous utilisons une classe intermédiaire « Message » conçue spécifiquement pour l'échange de données entre nos classes. Une fois la liste complétée, le worker stocke cette dernière dans un Message avant de l'envoyer au thread. De plus, avant l'envoi du Message le worker génère, grâce aux informations concernant l'intervalle, un nom de fichier (Exemple : 0Cm_1Cm.ser) puis le stocke lui aussi dans le Message avant de l'envoyer. Il y stocke aussi un tableau correspondant au nombre d'occurences de chaque persistance de l'intervalle.

Maintenant, que l'on sait comment la liste et son nom de fichier sont échangés entre les deux classes, on peut aborder le sujet du stockage. Pour commencer, nous avons pris la décision de stocker tous les fichiers « .ser » dans un dossier « Fichier » afin d'ordonner notre travail. Le ThreadStockage commence par vérifier si le dossier « Fichier » existe. Si c'est le cas, le thread créé un fichier avec le nom récupéré par l'objet Message puis il y stocke la liste, avant de l'enregistrer dans le dossier « Fichier ». Sinon, il commence par créer le dossier « Fichier ».

Pour finir, le thread enregistre les données du tableau dans le fichier « tab.ser » en faisant appel à la fonction « StockageTabMoy » de la classe Serveur. La finalité de cette dernière tâche est d'indiquer la nouvelle quantité de persistances calculées et stockées, puis de mettre à jour les attributs de la classe Serveur qui servent au calcul de la moyenne globale des persistances.

L’affichage

Nous avons décidé de créer un affichage plus complexe, même si cela ne l’a pas été demandé, dans l’optique de donner à notre projet un rendu plus esthétique.



On compte pour la réalisation de cet affichage, l’utilisation de 4 méthodes différentes. On peut distinguer les deux premières qui ont le même nom « getStr() » mais qui n’appartiennent pas à la même classe. L’une appartient au thread ConnexionWorker et l’autre au thread ConnexionClient. La première renvoie une ArrayList<String> contenant la première chaîne de caractères « Worker » accompagné du nom du thread, suivi de l’état du worker ; c’est-à-dire si ce dernier est connecté ou pas puis enfin d’une barre de chargement. Enfin, la seconde effectue le même procédé mais cette fois avec le client et sans la barre de chargement. Notons que nous utilisons une méthode « affichePourcentage » présente dans la classe ConnexionWorker, qui s’occupe de générer la barre de chargement en fonction du nombre d’intervalles déjà traité, par le worker par rapport au nombre total d’intervalles qu’il est censé traiter. Pour finir on a la méthode « Affichage » située dans la classe serveur, qui s’occupe du rendu final. Elle a à sa disposition deux ArrayList<String> qui lui permettent de stocker toutes les ArrayList de chaque client et worker renvoyées par les méthodes « getStr() ». On a une ArrayList qui s’occupe des workers et l’autre des clients et c’est grâce à elles que l’on peut effectuer un affichage en colonne comme il est illustré ci-dessus. Cet affichage en colonne est réalisé dans une boucle « while ». A la sortie de cette boucle la méthode fini par un dernier affichage qui est l’affichage d’un message indiquant l’intervalle de nombres dont on possède les persistances.

Message

Cette classe nous sert uniquement à échanger des objets entre classes. Elle est composée d'attributs et d'accesseurs qui permettent de modifier ou de récupérer ces mêmes attributs. Parmi les attributs, on a une « Hashtable » qui nous permet de stocker la liste des persistances calculées. Nous avons ensuite, un « String » pour le nom que porte le fichier où sera stockée la liste de persistances, suivi d'un tableau « int[] » représentant le nombre d'occurrences des persistances de l'intervalle, et enfin, on utilise un « double » qui stocke la somme de toutes les persistances de l'intervalle. Cette classe nous permet au final d'échanger une grande quantité et diversité de données en une seule fois.

Worker

Description de la classe Worker

Lors de son lancement, le worker établie une première connexion avec le serveur via ces attributs « BufferReader », « Socket », « ObjectOutputStream » et « PrintWriter ». Grâce à elle, il s'identifie auprès du serveur comme worker et comme vu précédemment le serveur délègue la gestion du worker à un thread ConnexionWorker.

Dans un premier temps, le worker reçoit un intervalle de la part du ConnexionWorker. Pour le calcul des persistances, le worker s'aide de threads « ThreadCalcul ». Il divise l'intervalle en 5 intervalles et pour chacun des intervalles, il fait appel à un thread pour calculer les persistances de ce dernier. Une fois les threads lancés le worker attend qu'ils aient tous finis leurs calculs. Nous avons utilisé un attribut « Termine » de type booléen propre au « ThreadCalcul » pour nous indiquer si le thread auquel il appartient a fini son calcul ou non. Notons l'utilisation d'une ArrayList qui stocke toutes les adresses des ThreadCalcul qu'aura lancé le worker pour le calcul d'un seul intervalle. Grâce à cette liste et à l'attribut « Termine » nous vérifions dans une boucle while que chaque thread ait fini son calcul. Le worker continuera cette vérification tant que tous les threads continueront de calculer.

Avant de recevoir l'intervalle, le worker génère un nombre aléatoire et l'envoie au ConnexionWorker. Ce nombre correspond au nombre d'intervalles qu'il va traiter. Après la réception de l'intervalle, le worker génère un nom de fichier pour l'intervalle via sa méthode « generateurNom » A la suite de cette étape, le worker créé une instance Message pour stoker le nom de fichier.

Parmi ces attributs, le worker possède un tableau d'entier qui sert à récupérer le nombre d'occurrences de chaque persistance pour l'intervalle qui lui a été donné. Ce dernier, tout comme l'attribut « moy », est modifié par les ThreadCalcul lors de leur exécution. « moy » et le tableau

sont modifiés ou incrémentés après chaque calcul d'intervalle. A la fin, le worker stocke la liste de persistances, le tableau d'occurrences et la somme des persistances dans l'instance Message et l'envoie au ConnexionWorker.

Méthodes de la classe Worker

Dans la partie précédente, nous abordons l'utilisation de la méthode « generateurNom » dans le but de généré un nom de fichier. Dans les détails, cette méthode prend en paramètres deux entiers qui représentent le premier et le dernier nombre de l'intervalle donné au worker. La méthode incrémente d'un la valeur du dernier nombre et elle divise les deux entiers par 100,000. Étant donné que l'intervalle contient 100,000 nombres, la division des deux nombres nous renvoie deux entiers, par exemple pour l'intervalle [0 ; 99,999] on obtient 0 et 1. Ainsi, on utilise ces valeurs pour créer un nom de dossier, ce qui fait que pour le même exemple, on obtient le nom « 0Cm_1Cm.ser ».

La dernière tâche des ThreadCacul est de compléter les données de la liste des persistances, du tableau des occurrences et de la somme totale des persistances. Pour effectuer cette tâche, le thread fait appel à la méthode « remplitWorker » de la classe worker. Cette méthode prend en paramètre la liste (hashtable) des persistances des nombres de l'intervalle, le tableau des occurrences et la somme des persistances calculés, ces paramètres proviennent tous du ThreadCalcul et ils résultent des opérations effectuées sur l'intervalle donné au ThreadCalcul. Ainsi, le rôle de la méthode est d'ajouter ces données aux siennes.

Thread ThreadCalcul

Après la description du worker et l'explication de ses méthodes, on peut déjà se faire une idée du principe du ThreadCalcul. Étudions maintenant plus en détails son fonctionnement. Nous rappelons que les rôles de ce thread sont de calculer les persistances de l'intervalle donné, de compléter en même temps le tableau stockant les occurrences de chaque persistance, de sommer l'ensemble des persistances qu'il aura calculées et de stocker cette valeur.

Le calcul de la persistance d'un nombre est effectué par deux méthodes. Tout d'abord, on a la méthode « persistanceNombre » qui prend en paramètre un entier, il isole dans une liste chaque chiffre du nombre puis les multiplie entre eux avant de retourner le produit obtenu. Enfin, on a la méthode « ordrePersistance » qui dans une boucle « while » fait appel à la méthode « persistanceNombre » avec un nombre « x » en paramètre. La boucle continue de s'exécuter le code qui la compose tant que le nombre retourné par la méthode utilisée est composé de plus de 1 chiffre. Ainsi, on compte le nombre de fois que l'opération est effectuée, ce qui nous donne à la fin la persistance du nombre « x ».

Client

Description de la classe Client

Comme nous avons pu le voir dans la description générale de notre application, chaque client va pouvoir se connecter au serveur (chacun aura sa propre « session ») et ainsi accéder aux données. Pour pouvoir réaliser cette partie de notre application, nous avons utilisé la classe Client. Pour commencer, listons quelques attributs de cette classe. Il y a tout d'abord une variable (de type int) pour stocker le numéro de port utilisé par le serveur pour pouvoir se connecter. On peut aussi noter la présence d'une variable de type PrintWriter (writer) et une de type BufferedReader (reader) qui vont nous servir pour la communication avec notre serveur. La variable de type Socket (s) va nous permettre de nous connecter au serveur. Enfin, il y a une variable de type Scanner qui va permettre au client des taper au clavier les commandes associées aux requêtes. Notons que tous ces attributs sont déclarés avec le mot-clé static afin de pouvoir les utiliser dans le main de notre classe (les méthodes de notre classe subiront aussi le même traitement).

Analysons maintenant le code du main de cette classe pour mieux comprendre son fonctionnement. Il commence par initialiser les quatre dernières variables que nous avons vu. Dans le cas de la variable de type Socket, on utilise le premier argument du main (fournis lors de l'exécution du programme dans un terminal). Ce dernier correspond en fait à l'adresse IP du serveur (il nous sert pour établir la connexion entre eux.). Ensuite, on envoie à l'aide de la variable writer un message pour que le serveur puisse comprendre qu'il a affaire à un client (et pas à un worker). Ensuite, un message listant toutes les commandes disponibles s'affiche dans le terminal du Client. Tant que le message tapé par le client ne correspond pas à « END », l'utilisateur va pouvoir rentrer l'une différentes commandes pour accéder au contenu voulu. Voici à quoi cela peut ressembler :

```
Voici les commandes du serveur : Moyenne, Persistance, Proportion, END
```

Il pourra par exemple rentrer « Moyenne » pour pouvoir faire une requête en rapport avec la moyenne de la persistance des nombres. Chaque commande va nous permettre d'appeler une méthode de la classe pour continuer le traitement plus spécifique de la requête. On utilise un switch en fonction du message rentré pour pouvoir appeler la bonne méthode. Notons que nous ne prenons pas en compte le fait que le client puisse se tromper lors de la saisie des commandes. Dans le cas où l'utilisateur rentrerait END, on ferme les différentes variables vues précédemment et le programme prend fin.

Méthodes de la classe Client

La classe Client possède quelques méthodes pour récupérer les informations souhaitées par l'utilisateur. Notons que comme pour le cas de la classe ConnexionClient, elles se ressemblent plus ou moins, car elles permettent de rentrer plus de précisions par rapport à ses attentes (de la même manière que précédemment, à l'aide de la variable writer). C'est ici que la variable reader va nous servir ; elle nous permet de récupérer sous forme de chaîne de caractères le résultat avant de l'afficher dans le terminal.

Prenons l'exemple de la méthode Moyenne (pour continuer sur l'exemple de la classe ConnexionClient). On commence par afficher un message pour proposer une moyenne sur la totalité de ce qui a déjà été calculée ou sur un intervalle. Si l'utilisateur a rentré la commande sur la totalité, le programme a juste à récupérer le résultat renvoyé par le serveur (avec la variable reader) et à l'afficher. Dans le cas de la demande sur un intervalle, il faut rajouter deux étapes où l'utilisateur est invité à taper les premier et dernier nombres de l'intervalle.

Voici donc le fonctionnement de l'une de ces fonctions. Nous pouvons remarquer que le client ne récupère pas à proprement parler les données souhaitées. En réalité, on affiche juste le résultat sur son terminal. Notons qu'il existe les fonctions Proportion (pour le pourcentagedu nombre d'occurrences d'un ordre sur l'intégralité des nombres déjà calculés) et Persistance (pour la persistance d'un nombre spécifique). De la même manière que pour la classe ConnexionClient, nous n'expliquerons pas leur fonctionnement, car il est très similaire à celui de la fonction Moyenne.

Test de l'application

Pour terminer, nous vous proposons de suivre un test de l'application pour illustrer nos propos. Pour ce faire, nous allons commencer par lancer un serveur. On ouvre le serveur sur le port 8000.

```
theo@DELL-THEO:~/Bureau/Info4B/TeinturierQaeze/projet$ java mainServeur
Ouverture du serveur sur le port 8000
```

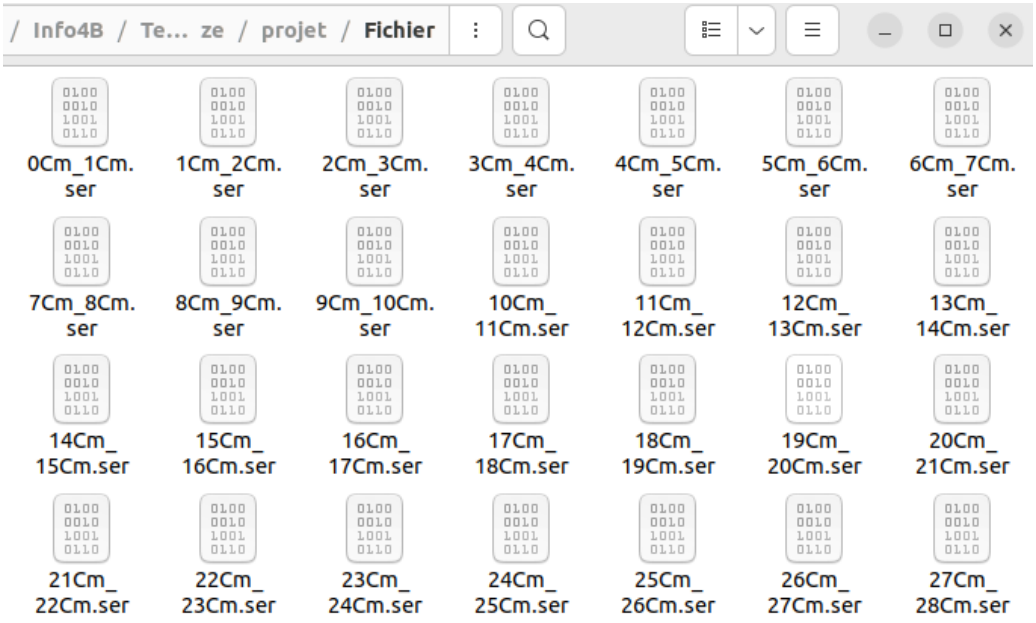
Pour l'instant il n' y a aucune donnée stockée. Connectons donc un worker. Il commence ses calculs dès que le serveur lui envoie un intervalle.

```
theo@DELL-THEO:~/Bureau/Info4B/TeinturierQaeze/projet$ java mainWorker 127.0.0.1
Calcul [0 , 99999]
Calcul [100000 , 199999]
Calcul [200000 , 299999]
Calcul [300000 , 399999]
Calcul [400000 , 499999]
Calcul [500000 , 599999]
Calcul [600000 , 699999]
Calcul [700000 , 799999]
```

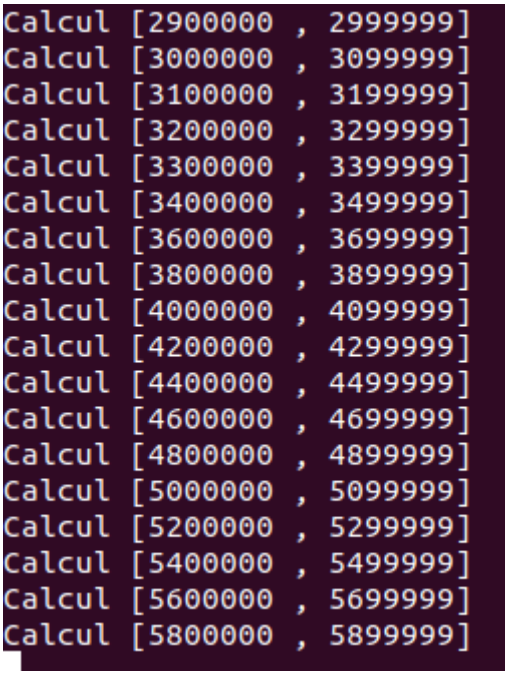
Pour s'assurer que le worker est bien connecté et travail, on peut se référencer à l'affichage du serveur.

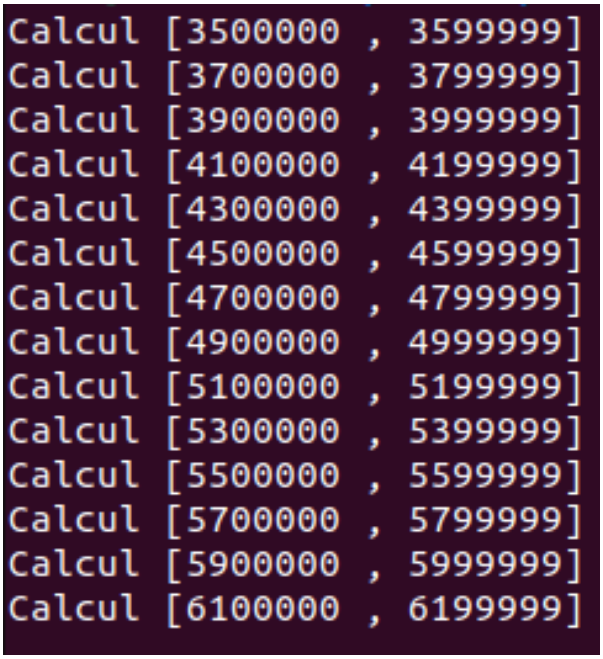
```
################################## SERVEUR ##################################
          WORKER                                          CLIENT
*****************************************************************************
Worker Thread-0                    |
Connecté                           |
[##................]              |
                                   |
********************************** |
                                   |
#############################################################################
Persistance calculée pour les nombres de 0 à 199999
```

Vérifions que les fichiers se stockent bien au fur et à mesure dans le dossier « Fichier ».



Maintenant, assurons-nous que deux workers qui travaillent en même temps ne traitent pas les mêmes intervalles. Les deux images suivantes nous confirment bien ce détail.





Pour finir avec ce test de l’application, connectons un client pour pouvoir récupérer toutes les données possibles.

```
theo@DELL-THEO:~/Bureau/Info4B/TeinturierQaeze/projet$ java Client 127.0.0.1

Voici les commandes du serveur : Moyenne, Persistance, Proportion, END
```

Un message avec les différentes commandes s’affiche. De la même manière que pour le worker, il est possible de vérifier s’il est connecté grâce à l’affichage du serveur. Demandons une moyenne sur la totalité des valeurs déjà calculées.

```
Voici les commandes du serveur : Moyenne, Persistance, Proportion, END
Moyenne
Moyenne totale (TOTALE) ou sur un intervalle (INTERVALLE) ?
TOTALE
La moyenne sur toutes les valeurs déjà calculées est 1.7610977391304348
```

Voici ce qui se passe sur un intervalle spécifique (notons que les deux nombres choisis ne sont pas stockés sur le même fichier.) :

```
Voici les commandes du serveur : Moyenne, Persistance, Proportion, END
Moyenne
Moyenne totale (TOTALE) ou sur un intervalle (INTERVALLE) ?
INTERVALLE
Les chiffres de cette intervalle doivent être inférieurs à 11500000
Ecrivez le premier chiffre de l'intervalle
25
Ecrivez le dernier chiffre de l'intervalle
350000
La moyenne sur cet intervalle est La moyenne de la persistance des nombres entre 25 et 350000 est de 2.006669028733399
```

Pour finir, voici ce qui se passe pour les autres commandes du client :

```
Voici les commandes du serveur : Moyenne, Persistance, Proportion, END
Persistance
Veuillez rentrer le nombre choisi, il doit être inférieur à 4900000
345
Sa persistance est de 2

Voici les commandes du serveur : Moyenne, Persistance, Proportion, END
Proportion
Choisissez un ordre de persistance (entre 0 et 11), nous vous rendrons le pourcentage par rapport au nombre de valeurs déjà calculées
1
Les nombres de persistance 1 représentent 45.6334081632653% des nombres déjà calculés

Voici les commandes du serveur : Moyenne, Persistance, Proportion, END
END
theo@DELL-THEO:~/Bureau/TeinturierQaeze$
```