Compte Rendu
Info4B
Sujet 1 : Persistance des nombres

POUZIN Pierre-Emmanuel
THOMAS Allan

Table des matières

I) Analyse fonctionnelle du sujet

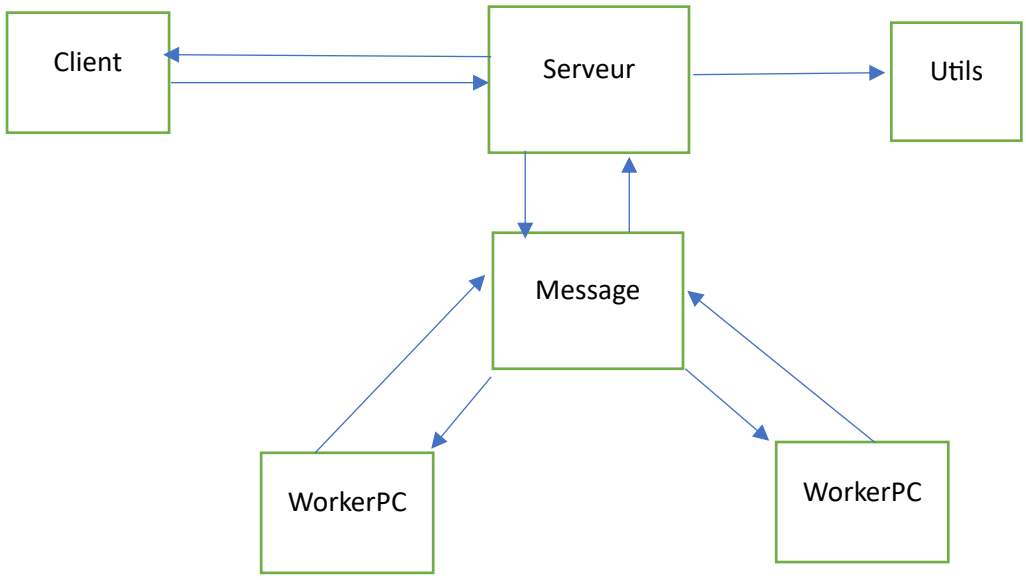
Objectif : L'objectif de ce projet est d'exploiter la puissance de plusieurs machines pour répartir le traitement d'un nombre de tache élevé et ainsi bénéficier d'une grande capacité de calcul.
Les taches consistent à calculer la persistance multiplicative des nombres.
Pour comprendre les enjeux demandés et les solutions mises en place, nous allons répartir les tâches en sous-problèmes tout le long de ce compte rendu.

a)Besoins

Nous devons mettre en place un réseau pour commencer. Nous avons donc besoin d'un serveur qui va faire faire transiter toutes les données entre lui et les machines qui feront les calculs. Ces calculs seront renvoyés au serveur qui devra les stocker dans un ou plusieurs fichiers. Chaque ordinateur qui traite les calculs devra optimiser pour utiliser au maximum les cœurs du PC. On devra aussi limiter le nombre d'envois entre les PC qui calculent et le serveur pour éviter d'utiliser trop de mémoire. Et enfin un client pourra observer les résultats des calculs en faisant une demande au serveur : il a le rôle d'un spectateur.
Nous avions aussi besoin de variables capable de stocker des nombres gigantesques : c'est pour cela que nous avons opté pour des variables de type *BigInteger*.

b)Arbre de mise en place

Petit schéma pour montrer comment vont transiter les informations, et la mise en place des packages :



c) Explication de l'utilisation des paquages

Le serveur va pouvoir recevoir plusieurs connexions de type client et WorkerPC. Il enverra les intervalles à calculer au WorkerPC via les Messages. Il mettra à disposition dans un fichier les résultats des calculs pour que le client y ai accès et puisse le consulter.
WorkerPC contient la classe Worker et WorkerPC, Worker va permettre de faire tous les calculs tandis que WorkerPC va récupérer les calculs des Workers et les envoyer au serveur via Message ;
Le client pourra alors observer en demandant au serveur les résultats des calculs.
Le package Utils permettra de gérer le stockage des données reçues par le serveur, mais aussi d'actualiser les statistiques affichées dans le client.

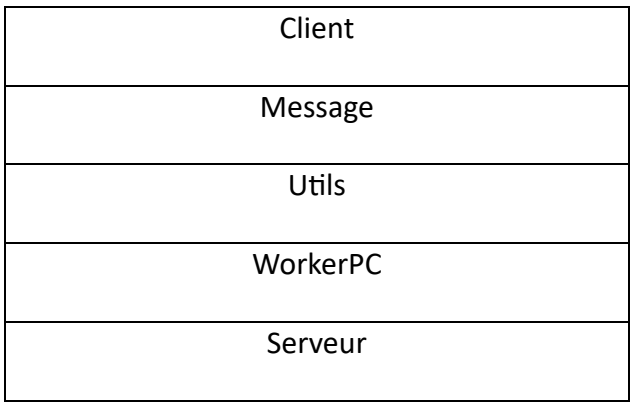
d)Structure de donnée envisagée et retenue

Pour pouvoir faire transiter les données, il va falloir utiliser une structure pour les contenir : pour cela nous avons choisis des ArrayList, qui sont facile à manipuler et qui nous permettrons facilement de gérer les résultats et de les renvoyer au serveur.
On va ensuite directement sérialiser les ArrayList de résultats grâce au package Utils dans des fichiers textes. Il ne nous est pas utile d'enregistrer les résultats dans une Hashtable, car on dispose de méthodes performantes pour désérialiser les résultats : il n'est donc pas utile de les stocker en mémoire. De plus, les statistiques sont directement calculées dès l'obtention des résultats.

II) L'architecture logicielle détaillée

a)Schéma de l'architecture logiciel

Structure couche par couche de la plus haute à la plus basse. Ce principe se base sur l'architecture d'un système d'exploitation, où chaque couche à laquelle on descend a beaucoup plus de taches à gérer que les précédentes. Le serveur ici peut être perçu comme le noyau d'un SE, car c'est lui qui gère absolument tout.



b)Package serveur

La couche la plus basse, identique au noyau d'un SE. Elle va devoir gérer toutes les données qui transitent, comme ouvrir des flux pour faire passer des données et les stocker.
Dans la classe Server :

```
int numWorkerPC = 0;

int port = 1111;

// 1 - Ouverture du ServerSocket par le serveur

ServerSocket s = new ServerSocket(port);

System.out.println("SOCKET ECOUTE CREE => " + s);
```

On doit d'abord créer un ServerSocket, il s'agit d'une interface logicielle grâce à laquelle on pourra facilement exploiter les services d'un protocole réseau (ici le TCP) car on a de grosse ressources à envoyer et récupérer , c'est-à-dire que l'on va avoir un flot continu d'octets . Il est donc plus fiable et a une correction d'erreur de bout en bout, on a aussi un multiplexage, c'est-à-dire qu'il peut servir à plusieurs processus de la même machine.

```
while (numWorkerPC < maxWorkerPC) {
    /*
     * 2 - Attente d'une connexion worker pc (la méthode s.accept() est bloquante
     * tant qu'un client ne se connecte pas)
     */
    Socket soc = s.accept();
    /*
     * 3 - Pour gérer plusieurs clients simultanément, le serveur attend que les
     * clients se connectent, et dédie un thread à chacun d'entre eux afin de le
     * gérer indépendamment des autres clients
     */

    ConnexionWorkerPC cc = new ConnexionWorkerPC(numWorkerPC, soc, distrib);
    System.out.println("NOUVELLE CONNEXION - SOCKET => " + soc);
    numWorkerPC++;
    cc.start();
}
s.close();
```

On vient créer un socket pour accepter les connexions des WorkerPC. On met une condition de sortie de la boucle pour ensuite fermer le serveur. Ainsi, notre réseau est mis en place et on peut désormais accepter de nombreux WorkerPC sur le réseau.
On créer des ConnexionWorkerPC que l'on vient ajouter sur le réseau, on lui passe en paramètre « distrib », qui sera l'intervalle des persistance à calculer.
Maintenant, voyons ce qu'il se passe dans la classe ConnexionWorkerPC qui est un Thread .
Classe : ConnexionWorkerPC.
C'est là que les objets vont être envoyés et récuperés.

```
public void run() {
    try {
        while (true) {

            di = (DemandeInter) in.readObject();
            System.out.println("Demande d'interval reçu");
            if (di.getOk() == true) { // on envoie quand il y a une demande

                try {

                    out.writeObject(distributionInter);// envoie l'intervale
                } catch (IOException e) {
                    e.printStackTrace();
                }
                System.out.println("Serveur a envoye interval:");
            }

            //ping = (PingEnvoieList) in.readObject();
            //System.out.println("Ping pour recevoir un interval:");

                try {
                    messagein = (Message) in.readObject();
                } catch (ClassNotFoundException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                } // lecture message


            System.out.println("Serveur a reçu les Arrayliste de retour ");
```

Quand on start le Thread voici ce qu'il se passe : il tourne en permanance pour pouvoir envoyer les intervalles si un WorkerPC se connecte, si il a fini, il redemande un intervalle jusqu'à ce qu'on ai plus rien a envoyer.
DemandeInter est une classe qui est similaire à un ping. Le WorkerPC ping le serveur pour un intervalle. Nous reviendrons plus tard sur cette classe.
Si un ping est reçu, le serveur envoie l'intervalle à calculer au workerPC ,
De même pour « distributioninter » qui va envoyer un objet qui contient les intervalles a calculer.
Ensuite le Server va attendre les resultats sous forme de liste .

```
if (messagein != null) {
    for (int i = 0; i < messagein.getList().size(); i++) {


            gatherAverage(messagein.getList().get(i));

            gatherOccurrences(messagein.getList().get(i));

            gatherPerstMax(messagein.getList().get(i));

            gatherMedian();

            // on cree un string avec un path de la forme

            file = this.fileHandler.checkFile(this.logsPath, messagein.getListeMin().get(i) + "-" + messagein.getListMax().get(i) );

            // on serialise la liste des resultats dans file
            this.fileHandler.serializeList(file, messagein.getList().get(i));


    }
    System.out.println("Le serveur a tout ecrit dans les fichiers avec les resultats de retour");

}

if (distributionInter.getok()==false) {// si on a aucun interval a envoyer
    System.out.println("Aucun intervalle a envoyer");
    break;
}
```

Si le message reçu contient bien des resultats, alors nous allons commencer à manipuler les résultats pour calculer les statistiques :

gatherAverage() est une fonction qui calcule la moyenne des persistances :

```
// fonction qui retourne la moyenne des persistances
public void gatherAverage( ArrayList<String> list ){

    // pour tout les string de list
    for( int i = 0 ; i < ( list.size() - 1 ) ; i++ ) {

        // on augmente le nombre d'element constituant la moyenne
        this.nbElem = this.nbElem.add( BigInteger.ONE );

        // on ajoute la persistance a la somme en separant la persistance du nombre
        this.sum = this.sum.add( BigDecimal.valueOf( Integer.parseInt( list.get(i).split(":")[1] ) ) );

        // on divise la somme par le nombre d'elements ( le diviseur , la precision de resultat , le mode d'arrondissement )
        this.average = this.sum.divide( new BigDecimal( this.nbElem ), 2 , RoundingMode.CEILING );

    }

}
```

gatherOccurence() une fonction servant à trouver le nombre d'occurrences par persistance.

```
public void gatherOccurrences(ArrayList<String> list) {

    // variable pour stocker le resultat
    byte resInter;

    // pour tout les String de la list
    for (String s : list) {

        // on recupere la persistance
        resInter = (byte) Integer.parseInt(s.split(":")[1]);

        // si on a deja rencontre cette persistance
        if (this.occurences.containsKey(resInter)) {

            // on incremente la valeur associe
            this.occurences.replace(resInter, this.occurences.get(resInter).add(BigInteger.ONE));

        } else {

            // on ajoute une "ligne" ayant pour key la persistance et value 1
            this.occurences.put(resInter, BigInteger.ONE);

        }

    }
```

On parcourt la liste de résultats passée en paramètre, et a chaque fois que la persistance du nombre de la liste est trouvée dans la Hashtable occurrences, on incrémente le nombre associé à cette persistance dans occurrences. Une fois que tous les résultats ont été vérifié, this.occurrences contient le nombre d'occurrences par persistance.

On a ensuite **gatherPerstMax(),** fonction qui met dans highestPers, un attribut de type Arraylist, les nombres ayant la plus grande persistance. Chaque fois qu'une persistance plus grande a été trouvée, on vide la liste pour avoir uniquement les nombres ayant la plus grande persistance :

```java
// fonction qui met dans highestPers les nombres ayant la plus grande persistance
public void gatherPerstMax( ArrayList<String> pList) {

    // pour tout les string de pList
    for( String s : pList ) {

        // si la persistance du nombre est = a this.perstMax
        if( Integer.parseInt( s.split(":")[1] ) == this.perstMax ) {

            // on ajoute le nombre dans la liste des plus grandes persistances
            this.highestPers.add( s );

        // si la persistance du nombre est > a this.perstMax
        }else if( Integer.parseInt( s.split(":")[1] ) > this.perstMax ) {

            // on vide la liste des plus grandes persistances
            this.highestPers.clear();

            // on change la valeur de la plus grande persistance
            this.perstMax = ( byte ) Integer.parseInt( s.split(":")[1] );

            // on ajoute le nombre dans la liste des plus grandes persistances
            this.highestPers.add( s );

        }
    }
}
```

Puis la fonction **getMedian**(), qui va calculer la médiane des persistances :

```java
// fonction qui retourne la mediane des persistances
public void gatherMedian() {

    // si le nombre de persistances diferentes est pair
    if( this.occurences.size() % 2 == 0) {

        // la mediane vaut la moyenne des valeurs de rang this.occurences.size() / 2.0 et (  this.occurences.size() / 2.0 + 1 )
        this.median = (float) ( (  ( this.occurences.size() / 2.0 )  +  ( this.occurences.size() / 2.0 + 1 )  ) / 2.0 ) ;

    }else {

        // sinon la mediane vaut la valeur se trouvant au centre
        this.median = this.occurences.size() / 2;

    }

}
```

La fonction **getPerstNb**() sert à retourner le résultat pour un nombre précis :

```java
// fonction qui retourne la persistance d'un nombre
public int getPerstNb(final BigInteger pNb) {

    // on return seulement la persistance en separant le string donne par la fonction
    return Integer.parseInt( this.fileHandler.getSingleResFromFile(pNb, this.INTERVAL , this.logsPath).split(":")[1] );

}
```

La fonction **getResultsInterv**() sert à retourner une ArrayList de résultat d'un intervalle passé en paramètre :

```java
// fonction qui retourne une arraylist d'un intervalle de resultats
public ArrayList<String> getResultsInterv( final BigInteger pMIN, final BigInteger pMAX ){

    // on retourne une ArrayList<String> de resultats renvoyee par la fonction getResFromFiles
    return this.fileHandler.getResFromFiles(pMIN, pMAX, this.INTERVAL , this.logsPath);

}
```

Ces deux fonctions utilisent la classe *FileManager* du package Utils.

Si le Server n'a plus d'intervalle à envoyer, on ferme les flux entre le Server et le WorkerPC.

c)La gestion des fichiers via le package Utils

Cette classe contient 7 fonctions.

On commence par appeler la fonction checkDirectory, pour créer | remplacer le dossier ayant comme chemin celui passé en paramètre. Elle est utilisée pour créer les dossiers de logs.

```java
// fonction pour creer | remplacer un dossier
public String checkDirectory(String pPath){

    Path toPath   = Paths.get(pPath);   // chemin du dossier a creer

    try{

        // si le dossier n'existe pas, on le cree | et si il existe, on le remplace
        toPath = Files.createDirectories(toPath);

        System.out.println("Directory created at: " + toPath.toString());

    // ou cas ou il y a une erreur
    }catch(IOException e) {
        e.printStackTrace();
    }

    // retourne le chemin du dossier sous forme de String
    return toPath.toString();

}
```

La fonction checkFile permet de faire la même chose que checkDirectory, mais cette fois pour des fichiers.

```java
// fonction qui va creer le fichier log et retourner son chemin
public String checkFile(final String pParent, final String pName) {

    // on cree un chemin de .txt : pParent/pName
    Path pathToLog = Paths.get(pParent + pName + ".txt" );

    try{

        // si il existe deja, on le supprime
        Files.deleteIfExists(pathToLog);

        // on cree le fichier a l'emplacement demande
        pathToLog = Files.createFile(pathToLog);

        System.out.println("File created at: " + pathToLog.toString());

    // ou cas ou il y a une erreur
    }catch(IOException e) {
        e.printStackTrace();
    }

    // retourne le chemin en String
    return pathToLog.toString();
}
```

Les résultats envoyés au serveur sont sérialisés par la fonction serializeList. Ils sont sérialisés dans un fichier texte ayant comme nom le minimum de l'intervalle du worker – le maximum de l'intervalle du worker. Par exemple : 0-20000.txt .

```
public void serializeList(final String pPath,final ArrayList<String> toWrite){

    try {

        FileOutputStream fos = new FileOutputStream( pPath );

        ObjectOutputStream serializer = new ObjectOutputStream( new BufferedOutputStream( fos ) );

        serializer.writeObject(toWrite);

        serializer.close();

        fos.close();

    }catch(IOException e) {
        e.printStackTrace();
    }
```

La fonction readLog permet de désérialiser un fichier grâce à son chemin, elle se base sur le même principe que la méthode précédente, mais nous allons stocker les résultats dans une liste en lisant la liste sérialisée. On créé un flux sur le fichier dans lequel on veut lire la liste. On utilise ensuite un buffer pour désérialiser la liste plus rapidement. On ferme ensuite les flux créés. La fonction retourne la liste désérialisée.

```
public ArrayList<String> readLog(final String pPath){

    ArrayList<String> list = null;

    try {

        FileInputStream fis = new FileInputStream(pPath);

        ObjectInputStream deserializer = new ObjectInputStream( new BufferedInputStream( fis ) );

        list = ( ArrayList<String> ) deserializer.readObject();

        deserializer.close();

        fis.close();

    }catch(IOException e) {
        e.printStackTrace();
    }catch(ClassNotFoundException e) {
        e.printStackTrace();
    }

    return list;
}
```

-public String **getSingleResFromFile**() cette fonction prend en paramètre un nombre, grâce auquel on va pouvoir trouver le fichier dans lequel il est. Ensuite, on parcourt la liste pour récupérer le résultat de ce nombre.

-public ArrayList<String> **getResFromSingleFile**() cette fonction va retourner une liste de résultats contenus dans une seule Log. Elle prend en paramètre un intervalle , utilise la fonction readLog() pour désérialiser les résultats et supprime les nombres en dehors de l'intervalle. Puis elle renvoie la liste « triée ».

-public ArrayList<String> **getResFromFiles**() est une fonction utilisant la fonction **getResFromSingleFile()** et parcourant plusieurs fichiers logs pour obtenir un grand nombre de résultats.

Ce package contient aussi une classe StatsClient, qui est un thread, permettant d'afficher et actualiser l'affichage des résultats sur l'interface client. Elle ne calcule rien, car les statistiques sont calculées par le serveur.

d) Package Worker

Du fait que les WorkerPC ont beaucoup de travail, c'est la deuxième couche la plus basse, donc nous allons le découper en plusieurs étape : Récupération et envoi des résultats, et calcul des persistances.

e)Récupération et envoi des résultats

Nous avons dans cette classe beaucoup d'attribut de type liste car on manipule des BigIntegers que l'on vient stocker et on récupérer aussi les intervalles que les WorkerPC ont reçu dans une liste.

```
Socket socket = new Socket("127.0.0.1", port);

System.out.println("SOCKET = " + socket);

/*
 * 5b - A partir du Socket connectant le serveur au workerPC, le client ouvre 2
 * flux : 1) un flux entrant afin de recevoir ce que le serveur envoie 2) un
 * flux sortant (ObjectOutputStream) afin d'envoyer des messages au serveur
 */
ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());

out.flush();

ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
```

Comme pour le Server, on ouvre deux flux : un entrant et un sortant ,pour récupérer et envoyer des objets. On crée aussi le Socket en mettant l'IP de la machine (ici on est en localhost) et le port sur lequel on veut se connecter.

```
while (!arreter) {
    // Appel serveur pour un nouvelle intervalle
    try {
        if (di.getOk() == true) {
            // on fait la demande d'interval
            out.writeObject(di);
            System.out.println("Demande d'intervalle au Serveur");

            try {
                distributionInter = (Distribution) in.readObject();
            } catch (IOException e) {
                e.printStackTrace();
            }

            System.out.println(" l'intervalle demandé est reçu");
```

On envoie un message au serveur pour lui dire que l'on est prêt à recevoir un intervalle, « di » est un Object qui contient un booléen , nous lisons l'intervalle reçu du serveur.

```
        pMIN = distributionInter.getMin();

        pMAX =distributionInter.getMax();

        INTERVAL = distributionInter.getInter();

        for (int i = 0; i < nbWorkers; i++) {

            // worker avec comme intervalle [ pMIN ; pMAX ]
            workers.add(new Worker(pMIN, pMAX));

            System.out.println("Worker " + workers.get(i).getId() + " has started");

            workers.get(i).start();

            // si on doit ajouter un. On le fait comme ca aucune valeur n'est calculee 2x
            if (one) {
                pMIN = pMIN.add(INTERVAL).add(BigInteger.ONE);
                one = false;
                // sinon on ajoute juste l'intervalle
            } else {
                pMIN = pMIN.add(INTERVAL);
            }

            pMAX = pMAX.add(INTERVAL);

        }
```

On récupère le min et le max a calculer et INTERVAL est le pas .Le nombre de Workers correspond au nombre de cœur disponible sur la machine. On crée alors les Workers que l'on ajoute à notre liste qui contient tous les Workers et on leur passe l'intervalle à calculer.

```
// on regarde si les worker on finit
while (!fini) {
    fini = true;
    for (Worker w : workers) {

        // si le thread est en cours d'execution
        if (w.getState() == Thread.State.RUNNABLE)
            fini = false;

        // sinon si il est interrompu
        else if (w.isInterrupted())
            System.out.println("Thread " + w.getId() + " has been interrupted");
    }

}
```

Cela permet d'attendre que les Workers aient fini leur travail. Nous ne sommes pas partis sur une file d'attente car nous avons préféré attendre que tous les Workers terminent leur calcul. On récupère les résultats sur WorkerPC, puis on les envoie au Server et on redemande un intervalle. Cela évite de faire de nombreuses requêtes au Serveur.

```
// pour tous les workers
for (int i = 0; i < workers.size(); i++) {

    retour.add(new ArrayList<String>());

    retour.get(i).addAll(workers.get(i).getList());

    listePmin.add(workers.get(i).getMIN().toString());

    listePmax.add(workers.get(i).getMAX().toString());
}


System.out.println("une Liste est prete a envoyé");

Message message = new Message(retour, listePmin, listePmax);
// les listes sont bien remplit verif faite
out.writeObject(message);
```

On vient ensuite stocker les résultats et les envoyer au Server. Quand on sort de la boucle on ferme les flux.

b)Les calculs

Les calculs sont effectués par les Workers, qui sont des Threads. La classe Worker contient cinq fonctions .
-**getList**() // retourne la liste des résultats qui sera envoyée au serveur.
-**getmin**() et **getmax**() retournent respectivement le minimum et maximum de l'intervalle spécifié.

- **persistence**(final BigInteger MIN ,final BigInteger MAX)

Le but de cette méthode est de calculer la "persistance multiplicatrice" de chaque nombre entier entre MIN et MAX (inclus) et de stocker ces informations dans une liste.

La méthode commence par initialiser la persistance à 0 et le nombre d'itérations à effectuer à MAX-MIN+1 (plus 1 car on commence à compter à partir de MIN). Elle définit également un offset pour pouvoir ne pas commencer à 0 et initialise un BigInteger "nbInt" à la valeur de l'offset. Certains attributs sont marqués *final*, car ils ne seront pas modifiés par la suite.

```
int         persistance = 0;                          // persistance du nombre
BigInteger  iterations  = BigInteger.ONE.multiply(MAX).add(BigInteger.ONE); // nb d'iterations. On fait + 1 car dans les boucles on commence a 0
final BigInteger offSet = MIN;                        // un offset pour ne pas commencer a 0; pas de modif = const
BigInteger  nbInt       = offSet;                     // un BigInteger initialise a la valeur de l'offset
BigInteger  nbInter     = BigInteger.ONE;             // un BigInteger initialise a 1

String      nb          = nbInt.toString();           // nbInt mais en type String ( pour les calculs )
```

```
// tant que le nombre d'iterations restantes est > 0
while( iterations.subtract(offSet).compareTo(BigInteger.ZERO) == 1 ) {
    // la taille du nombre intermediaire diminue : on boucle tant que taille > 1
    while( nb.length() > 1 ){
        // on parcours tout le nombre nb
        for( int j = 0 ; j < nb.length()  ; j++ ) {
            //nbInter = nbInter * ( nb.charAt(i) ) ;
            nbInter = nbInter.multiply( BigInteger.valueOf( Character.getNumericValue( nb.charAt( j ) ) ) );
        }
        // on met le nombre a calculer a la valeur intermediaire
        nb = nbInter.toString();
        // reinitialisation du nombre intermediaire
        nbInter = BigInteger.ONE;
         // incrementation de la persistance de ce nombre
        persistance++;

    }

    this.list.add( nbInt.toString() + ":" + persistance );
    // on incremente le nombre de base
    nbInt = nbInt.add( new BigInteger("1") );
    // on met dans nb la conversion de nbInt en String
    nb = nbInt.toString();
    // on reset la persistance
    persistance = 0;
    // on decremente le nombre d'iterations restantes
    iterations = iterations.subtract(BigInteger.valueOf(1));
```

Ensuite, la méthode effectue une boucle "while" pour chaque nombre entier entre MIN et MAX (inclus) et stocke les resultats dans une liste. Dans cette boucle, la méthode effectue une autre boucle "while" pour chaque chiffre du nombre, on multiplie chaque chiffre pour obtenir un nombre intermédiaire et met le nombre intermédiaire à la place du nombre initial. La boucle continue jusqu'à ce que le nombre ne comporte plus qu'un seul chiffre, moment où la persistance est incrémentée et la valeur du nombre est stockée dans la liste qui est déclaré en dehors de la fonction.

Enfin, la méthode incrémente le nombre de base, réinitialise la persistance, et diminue le nombre d'itérations restantes jusqu'à ce qu'il n'y en ait plus. Finalement, la liste contiendra des chaînes de caractères représentant chaque nombre entre MIN et MAX et sa persistance multiplicatrice correspondante.

Les résultats sont sous la forme suivante : 777:4 | nombre:persistance

-run() qui lance la méthode persistance() ;

f)Package Message

Passons à la couche supérieure. Nous en avons parlé plus haut mais nous n'avions pas défini ce que cela était. Le Package Message comporte plusieurs classes similaires qui servent à faire transiter les objets, ces classes sont en « implements serializable » car quand on fait passer un objets on a besoin de le sérialiser.

i)Classe Distribution

```
public class Distribution implements Serializable {

    public BigInteger min;
    public BigInteger max;
    public BigInteger inter;
    public boolean ok;

    public Distribution(BigInteger p_MIN, BigInteger p_max,BigInteger p_interval) {

        this.min= p_MIN;
        this.max = p_max;
        this.inter = p_interval;
        ok=true;

    }
```

C'est cette classe qui va permettre de faire passer les intervalles du Server au WorkerPC.

ii) class Message

```
public class Message implements Serializable {

    private ArrayList< ArrayList < String > > liste;
    private ArrayList < String >  pMin;
    private ArrayList < String >  pMAx;
```

Cette classe va faire transiter les listes de résultats qui contiennent les maximums et minimums des intervalles et la liste de résultats qui contient la persistance .

iii)class DemandeInter

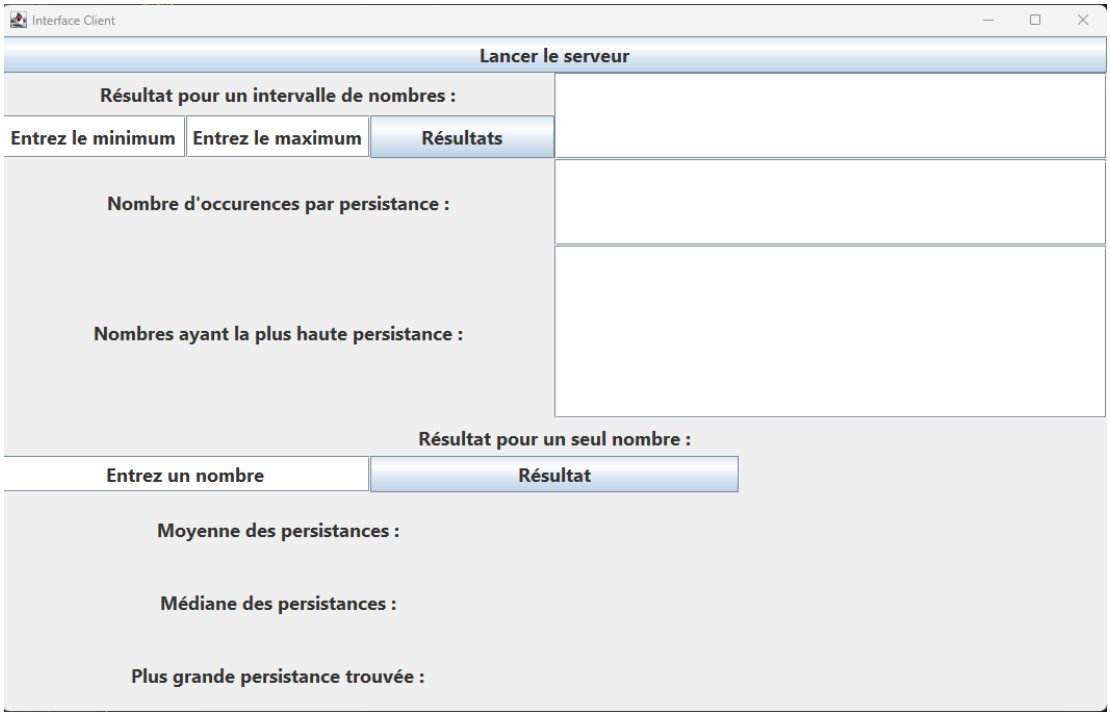
```
public class DemandeInter implements Serializable {

    public  boolean ok;
```

Cette classe contient un seul booléen qui joue le rôle d'un ping pour le serveur pour savoir s'il doit distribuer ou non l'intervalle.

g)Le Client

C'est la couche la plus haute car elle ne fait aucun calcul, elle a le rôle d'observateur. Il faut savoir que notre réseau n'était pas fonctionnel, alors pour simuler un client, nous avons pris le programme de base qui fonctionnait sur un seul pc et avons rajouté une interface graphique.

i)Interface et client



L'interface joue un peu le rôle du réseau, elle fait le lien entre le Server et le client mais ceci n'est qu'une simulation. Cette interface a été faite avec la bibliothèque swing de Java.
Le client peut entrer un nombre pour trouver sa persistance, rechercher un intervalle, voir la moyenne ,avoir la médiane ou encore voir la plus haute persistance sous forme de liste. Tous ces calculs se font en temps réel et le client peut y avoir accès même si les Worker n'ont pas fini de travailler. Les statistiques (médiane, moyenne, plus grande persistance, nombres ayant la plus grande persistance et nombre d'occurrences par persistances) sont actualisées en continu grâce à une instance de la classe StatsClient du package Utils.

III) Jeux de test montrant le fonctionnement du programme

Programme avec le réseau

Le programme contenant le réseau n'est pas fonctionnel donc nous allons montrer les parties qui fonctionnent.

Lancement du Serveur :

```
SOCKET ECOUTE CREE => ServerSocket[addr=0.0.0.0/0.0.0.0,localport=2222]
```

Lancement Worker :

```
SOCKET = Socket[addr=/127.0.0.1,port=2222,localport=55402]
Demande d'intervalle au Serveur
 l'intervalle demandé est reçu
50000
Worker 15 has started
Worker 16 has started
Worker 17 has started
Worker 18 has started
Worker 19 has started
Worker 20 has started
Worker 21 has started
Worker 22 has started
Worker 23 has started
Worker 24 has started
Worker 25 has started
Worker 26 has started
une Liste est prete a envoyé
```

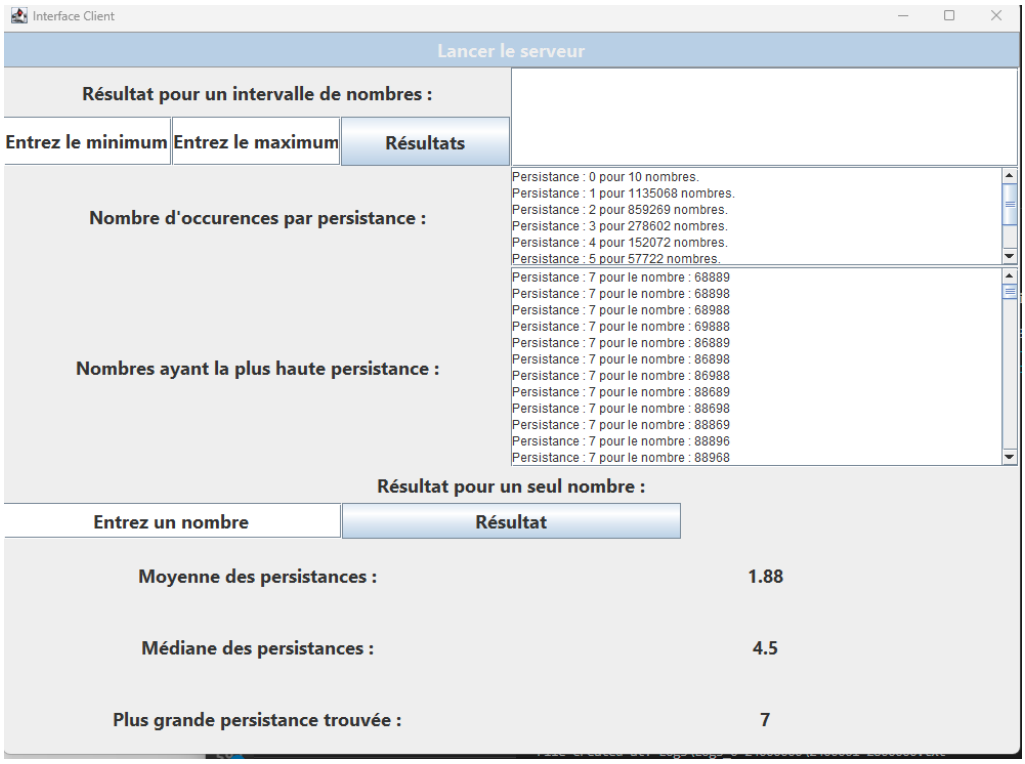
L'intervalle a bien été reçu, les Workers font leur travail .

Coté Serveur :

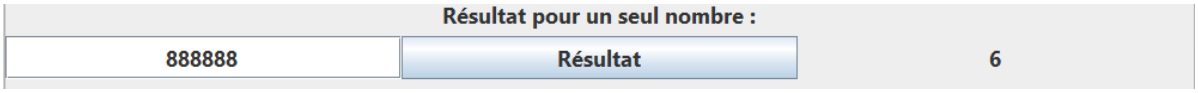
```
SOCKET ECOUTE CREE => ServerSocket[addr=0.0.0.0/0.0.0.0,localport=2222]
Serveur a cree les flux
NOUVELLE CONNEXION - SOCKET => Socket[addr=/127.0.0.1,port=55402,localport=2222]
Demande d'interval reçu
Serveur a envoye interval:
```

Les flux sont bien ouverts, on a la connexion d'un WorkerPC, la demande d'intervalle s'effectue mais le Serveur ne reçois jamais la liste de retour. Pourtant, les listes sont remplies, les autres objets fonctionnent mais le transfert des listes ne veut pas se faire. Nous nous arrêterons donc ici pour la démonstration de la partie réseau le reste se fera sur l'autre programme fonctionnel.

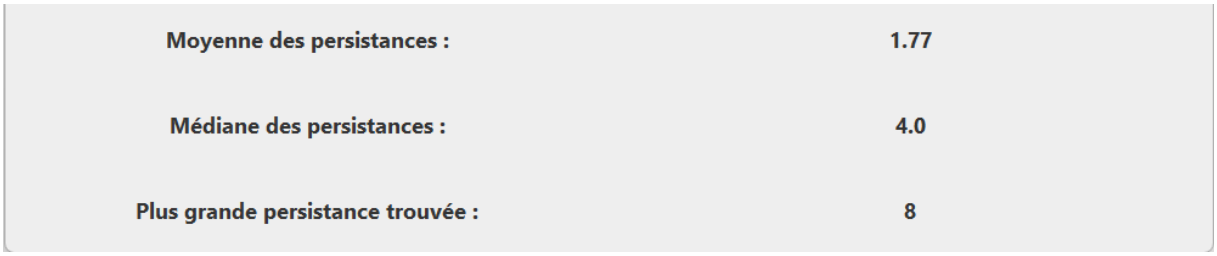
Programme sans le réseau



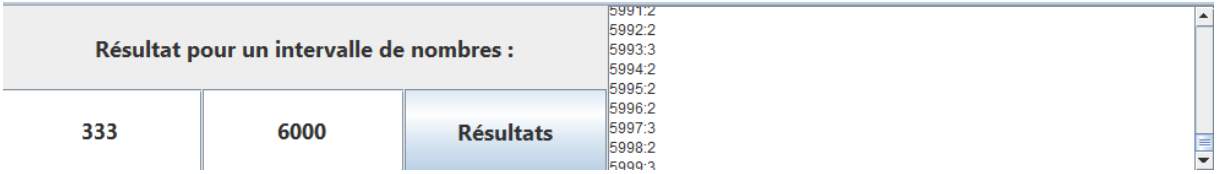
On ouvre l'interface et on lance le serveur. On voit les statistiques sont mis à jour en permanence.



La recherche pour un nombre compris dans l'intervalle total fonctionne parfaitement.



La moyenne, médiane, et la plus grande persistance sont mis à jour en permanence jusqu'à la fin (l'affichage ne s'actualise plus quand



L'intervalle de recherche fonctionne mais pas totalement, car certains intervalles ne sont pas pris. Les intervalles ne voulant pas s'effectuer dans la recherche sont « aléatoires ». Nous avons essayé de résoudre le problème, en vain. Nous savons où se situe le problème, mais… malgré de nombreuses tentatives, nous n'avons pas réussis à résoudre le problème.

```
Directory created at: Logs
Directory created at: Logs\Logs_0-24000000
Worker 26 has started
Worker 27 has started
Worker 28 has started
Worker 29 has started
Worker 30 has started
Worker 31 has started
Worker 32 has started
Worker 33 has started
Worker 34 has started
Worker 35 has started
Worker 36 has started
Worker 37 has started
File created at: Logs\Logs_0-24000000\0-400000.txt
File created at: Logs\Logs_0-24000000\400001-800000.txt
File created at: Logs\Logs_0-24000000\800001-1200000.txt
File created at: Logs\Logs_0-24000000\1200001-1600000.txt
File created at: Logs\Logs_0-24000000\1600001-2000000.txt
File created at: Logs\Logs_0-24000000\2000001-2400000.txt
File created at: Logs\Logs_0-24000000\2400001-2800000.txt
File created at: Logs\Logs_0-24000000\2800001-3200000.txt
File created at: Logs\Logs_0-24000000\3200001-3600000.txt
File created at: Logs\Logs_0-24000000\3600001-4000000.txt
File created at: Logs\Logs_0-24000000\4000001-4400000.txt
File created at: Logs\Logs_0-24000000\4400001-4800000.txt
Worker 38 has started
Worker 39 has started
Worker 40 has started
Worker 41 has started
```

Du coté du terminal, on voit bien que le dossier est créé, que les Worker fonctionnent. Une fois que les Worker ont fini, on voit bien les fichiers sont créés.

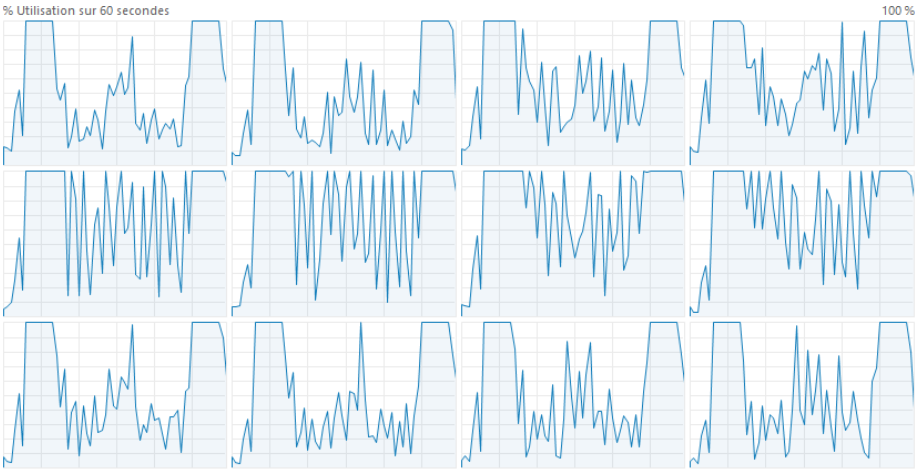


Figure 1Graphique de l'utilisation des ressources sur le processeur

Voici un graphique qui nous montre l'utilisation des ressources du processeur, on peut observer une bonne répartition des calculs et donc une bonne optimisation du programme.

IV)Résumé du projet

De nombreuse heures de travail ont été requises pour effectuer ce projet , qui regroupe de nombreuses problématiques dont nous avons pris connaissance lors de ce semestre : réseau, sérialisation, Hashtable, ordonnancement …
Malgré cela, nous n'avons pas réussi à faire fonctionner la partie réseau intégralement. Mais nous sommes content de ce que nous avons accomplis, et des nombreuses connaissances que nous avons acquises que ce soit pendant le projet ou lors du semestre, qui nous seront utiles lors de la poursuite de notre parcours*.