I. matchmaking centralisé multi-critères 1 joueur contre 1 joueur :

1) les interfaces *:*

Serveur :

```
public interface ServeurItf {
	public void matchmaking(JoueurItf j);
}
```

Client/Joueur :

```
public interface JoueurItf {
	public void jouer(int duration, int summonerIdAdversaire, int
summonerEloAdversaire, int latencyAdversaire, int durationAdversaire);
	public void match();
	public int getSummonerId();
	public void setSummonerId(int summonerId);
	public int getSummonerElo();
	public void setSummonerElo(int elo);
	public int getDuration();
	public void setDuration(int duration);
	public int getLatency();
	public void setLatency(int latency);
	public Socket getSocket();
	public void setSocket(Socket s);
}
```

2) implémentation

Trois implémentations pour le matchmaking 1V1 en centralisé ont été proposés : random, naïf et une implémentation à l'aide d'une base de données spatiale.

Le projet est divisé en 4 packages :

→ centralise_1V1_random consitué des classes suivantes :

- Serveur.java : Cette classe correspond au main, elle permet de créer et lancer la ThreadServer. Ce main attend les connexions des Joueurs, puis crée des copies des joueurs qui se connectent.
- TacheServeur.java : Cette classe est chargée de matcher les joueurs entre eux. Les joueurs sont matchés deux par deux selon leur ordre d'arrivé. Cette classe possède une classe interne. Cette classe est une tâche qui se réveille toutes les secondes et qui répertorie le nombre de connexions qui ont été reçues pendant la seconde courante. Et une qui permet de logger le nombre de joueurs qui se connectent chaque seconde.

→ centralise_1V1_naif constitué des classes suivantes :

- Serveur.java : Cette classe correspond au main, elle permet de créer et lancer la ThreadServer. Ce main attend les connexions des Joueurs, puis crée des copies des joueurs qui se connectent, les placent dans une file d'attente, et réveille le thread qui s'occupe de matcher les joueurs.
- ThreadServer.java : Cette classe est chargée de matcher les joueurs entre eux. Dés l'arrivée d'un joueur on essaie de le matcher avec des critères « stricts ». Si ce n'est pas possible on l'insère dans la file d'attente. Cette classe possède deux classes internes. Une qui est une tache qui se réveille toutes les trois secondes et qui incrémente le temps des joueurs passés dans la file d'attente du serveur. Si le temps d'attente est supérieur à 5 unités la tâche matche les joueurs entre eux avec des critères plus lâches. Et une qui permet de logger le nombre de joueurs qui se connectent chaque seconde.

→ centralise_1V1_BD_Spatiale consitué des classes suivantes :

- Serveur.java : correspond au main du serveur, permet notamment l'initialisation de la connexion vers la base de données spatiale.
- ThreadServer.java : Ce thread permet de préparer les requêtes qui seront envoyées à la base de données.
- TacheTraitement.java : Ce thread traite les requêtes qui ont été préparé par le ThreadServer. Il attend les réponses de la base de données.
- TacheConnexions.java : Cette tache (ici n'est pas interne à la thread ThreadServer pour des raisons de performance) répertorie le nombre de joueurs qui se sont connectés à chaque seconde.

→ centralise_1V1_utilitaire consitué des classes suivantes :

- ServeurItf.java : correspond à l'interface du serveur.
- JoueurItf.java : correspond à l'interface de JoueurImpl
- JoueurImpl.java : classe qui correspond à l'implémentation d'un joueur. La méthode match de cette classe permet notamment la connexion au serveur, la demande de matchmaking et l'attente de résultat.
- ThreadJoueur.java : prend un JoueurImpl en paramètre et appel la fonction match de la classe JoueurImpl
- Joueur.java : correspond au main qui récupère les critères des joueurs en lisant le fichier csv, et crée les joueurs
- Stats.java : cette classe écrit dans des fichiers csv les joueurs ainsi que leur caractéristiques (summonerId, summonerElo, latence, duration, temps d'exécution).
- Exploitation_resultats.java : cette classe exploite les fichiers produit par la classe Stats. Elle lit les fichiers csv (résultats des différents matchmaking) et crée un fichier qui contient les moyennes, statistiques sur les joueurs matchés ensembles.
- NbConnexions : cette classe est un compteur partagés pour connaître le nombre de joueurs qui se connectent chaque seconde.

Précision sur la classe Stats.java :

Les résultats produit par la classe Stats.java sont écrit dans des fichiers csv. Trois sortes de fichiers csv sont créés :

→ statistiques_joueurs_implemX.csv
→ Statistiques_implem_tempsX.csv
→ nb_connexions_par_seconde_implemX.csv

où implem correspond respectivement à random, naïf ou BD_Spatiale. Et où X correspond au numéro du test.
Chaque ligne du premier fichier possède les informations des joueurs matchés ensemble de la façon suivante :
summonerId1,summonerElo1,latence1,duration1,temps1,summonerId2,summonerElo2,latence2,duration2,temps2
Le deuxième fichier contient le temps total d'exécution afin de pouvoir matcher l'ensemble des 100000 joueurs.
Enfin chaque ligne du dernier fichier donne le nombre de joueurs qui se sont connectés chaque seconde durant l'exécution du programme.
Ces fichiers sont ensuite exploités par la classe Exploitation_resultats.java.

2.a) Implémentation random :

2.a.1) Le pseudo-code :

L'implémentation random consiste a matcher les joueurs deux par deux en ne considérant que leur ordre d'arrivés sur le serveur. Les deux premiers joueurs qui se connectent au serveur seront matchés ensemble, etc... On ne tient pas compte de la latence ou du summonerElo pour cette implémentation.

2.b) Implémentation naïve :

2.b.1) Le pseudo-code :

Une autre façon d'implémenter l'algorithme, peut se faire en se basant sur l'algorithme utilisé dans le jeu SMITE. En effet, on peut créer une liste pour chaque critère, on insère alors de manière triée chaque client dans chacune des listes. On prend ensuite dans chaque liste des sous-listes de joueurs qui correspondent à peu près au même critère que le client C1 que l'on souhaite insérer. On fait ensuite l'intersection entre ces sous-listes de joueurs. Si l'intersection est vide on insère C1 dans chacune des listes. Si l'intersection n'est pas vide on prend le joueur qui sera le plus proche de C1 (au niveau des critères de matchmaking).

Ce qui en pseudo-code donne :

Serveur naïf SMITE :

hypothèses : On souhaite matcher les différents clients en fonctions de 3 critères. On a donc besoin de 3 listes l1, l2, l3 qui vont être des listes de Client. Par exemple l1 contiendra l'ensemble des clients non matchés triés selon l'ELO des clients. De même l2 contiendra l'ensemble des clients triés selon la latence des joueurs, et l3 contiendra l'ensemble des joueurs triés selon la durée d'attente au sein du serveur.

ListeClient l1, l2, l3
attente de connexion d'un client j1
estampille de C1 avec le temps courant
si vide (l1) alors //si l1 vide, l2 et l3 le sont forcément aussi
    insérer de manière triée j1 dans l1 (suivant les critères propres à la liste l1 appelé critere1)
    insérer de manière triée j1 dans l2
    insérer de manière triée j1 dans l3
sinon
    Soient i, j et k les indices où j1 devrait être inséré dans respectivement l1, l2 et l3
    Soient sl1, sl2 des sous-listes de l1, l2 respectivement
    sl1 = selection des joueurs de l1 (qui par exemple ont un ELO de plus ou moins 20 par rapport à j1)
    sl2 = selection des joueurs de l2
    Si !vide(res = intersections(sl1, sl2)) alors
        choisir le joueur j2 de res dont la durée d'attente au sein du serveur est la plus grande
        supprimer j2 de l1, l2 et l3
        envoyer référence de j2 à j1
        envoyer référence de j1 à j2
    Sinon
        insérer j1 en i dans l1
        insérer j1 en j dans l2
        insérer j1 en k dans l3
    FinSi
FinSi

Thread du serveur : Toutes les 3 secondes :
incrémenter la durée d'attente des joueurs au sein du serveur d'une unité
si un Joueur a attendu plus de 5 unités alors
    ListeJoueurs liste = récupérer l'ensemble des joueurs ayant attendu plus de 5 unités
    Tant que liste possède au moins 2 joueurs faire
        matcher le premier joueur j1 de liste avec le joueur j2 de liste ayant le Elo le plus de proche
        supprimer j1 et j2 de liste
        envoyer référence de j2 à j1
        envoyer référence de j1 à j2
    Fin TantQue
    Si nonVide(liste) alors // il reste un joueur avec une durée de 5
        si l1 possède au moins deux joueurs
            matcher le joueur j1 de liste avec le joueur j2 de l1 ayant le Elo le plus proche
            supprimer j1 et j2 de liste
            envoyer référence de j2 à j1
            envoyer référence de j1 à j2
        FinSi
    FinSi

Client / Joueur :

Récupérer critères (id jouur, elo, latence) de jeux depuis la base de données
attendre un temps random // permet de n'avoir pas tout le temps la même exécution
demande de connexion au serveur
demande de matchmaking au serveur
attente d'un résultat

2.b.2) indications d'implémentation :

Indications sur la base de données :

Dans la classe Joueur.java afin de récupérer les propriétés des joueurs on doit effectuer des requêtes pour interroger la base de données. Ces requêtes sont les suivantes :

```
Statement s1 = connection.createStatement();
ResultSet res1 = s1.executeQuery("select * from Summoner");
while (res1.next()) {
      summonerElo  = res1.getInt("summonerElo");
      // on ne garde que les joueurs dont le elo a une valeur significative
      if (summonerElo != -1) {
      // récupération de la latence moyenne de chaque joueur conservés
            ResultSet res2 = s2.executeQuery("select avg(userServerPing) from
games where userId = " + res1.getInt("acctId"));
            res2.next();

            // création d'un nouveau joueur ayant pour critères les critères
récupérés depuis la base de données
            j1 = new JoueurImpl(res2.getInt("userServerPing"), summonerElo);
            s.matchmaking(j1);
      }
}
```

Interroger la base de données de cette manière n'est pas du tout efficace. On peut alors au lieu de cela, à l'aide de sqlite3 mettre dans un fichier csv le résultat de la requête que l'on souhaite et ensuite récupérer les valeurs à partir de ce fichier en utilisant la fonction splite de java.
Pour obtenir le résultat de la requête de récupération des propriétés des joueurs dans un fichier on doit dans sqlite3 taper les commandes suivante :
.output « joueurs_proprietes_1V1_acctId=userId.csv » # le fichier csv est le fichier de sortie qui contiendra les résultats voulus.
.separator « , » #en csv les attributs sont séparés par des virgules
select s.acctId, s.summonerElo, avg(g.userServerPing) from summoner s, games g where g.userId = s.acctId and s.summonerElo <> -1 ;

On pourra alors récupérer les critères des joueurs en lisant directement le fichier csv.

Indications sur RMI :

Initialement le programme avait été écrit à l'aide de RMI. Cependant RMI apparaît comme être une mauvaise solution niveau performance, on va donc adapter les codes déjà effectués de telle sorte à ne plus utiliser RMI, mais plutôt les sockets java.

Il faut alors utiliser un protocole de communication entre le client et le serveur.

Message envoyé du client vers le serveur :
type de message « matchmaking summonerId summonerElo latence ».

Message envoyé du serveur vers le client :
type de message « infoJoueur duration summonerEloAdversaire latenceAdversaire durationAdversaire ».

duration correspond au temps d'attente au sein du serveur. Duration correspond donc au temps d'attente avant d'avoir trouvé un joueur compatible pour le matchmaking.

Indications sur les critères de matchmaking :

Dans cette implémentation naïve, lorsque l'on essaie de matcher un joueur avec un autre, on insère les joueurs qui sont similaires dans des sous-listes spécifiques comme ci-dessous :

```
if((tmp.getSummonerElo() > (joueur.getSummonerElo() - 20)) &&
      (tmp.getSummonerElo() < (joueur.getSummonerElo() + 20))) {
            sl1.add(tmp);
}


if ((tmp.getLatency() > joueur.getLatency() - 20) &&
      (tmp.getLatency() < joueur.getLatency() + 20)) {
            sl2.add(tmp);
}
```

Ici les joueurs sont sélectionnés telle que leur summonerElo correspondent à plus ou moins 20 unités du joueur que l'on cherche a matcher. De même pour la latence. Il suffit de changer les 20 en une autre valeur pour avoir des joueurs plus ou moins proches du joueur que l'on cherche a matcher. Puis dès que la duration est supérieure ou égale à 5 unités (ce qui correspond a 5*3 = 15 secondes), on matche les joueurs entre eux en prenant les joueurs qui minimiseront les summonerElo (on ne considère plus la latence).

2.c) Serveur implémenté avec une base de données spatiale :

2.c.1) Le pseudo-code :

Une troisième implémentation peut se faire à l'aide d'une base de données spatiale. Il suffit de placer les joueurs dans un plan (2 dimensions) selon la valeur de leur elo et la valeur de leur latence.

Le pseudo-code du serveur est alors le suivant :

Serveurs avec base de données spatiale :

création de la base de données spatiale
attente de connexion d'un client C1
insertion de C1 dans une file d'arrivée

Tache quotidienne : toutes les 3 secondes

incrémenter la duration de l'ensemble des joueurs de la base de données d'une unité

insérer l'ensemble des joueurs de la file d'arrivée dans la base de données spatiale avec estampille à 1 unité de temps

supprimer tous les joueurs de la file d'arrivée.

matchmaking

s'il y a des joueurs dont la duration est supérieure ou égale à 5 alors

broad_matchmaking

matchmaking

requête pour récupérer pour chaque joueur les joueurs se trouvant à une distance de x (donc à une distance de x au niveau elo et à une distance de x au niveau latence)

pour chaque joueur j1 résultat de la requête faire

sélectionner un joueur j2 qui n'a pas déjà été choisi par un autre joueur

envoyer référence de j2 à j1

envoyer référence de j1 à j2

FinPour

supprimer l'ensemble des joueurs qui ont pu être matchés de la base

broad_matchmaking

requête pour récupérer l'ensemble des joueurs matchés les uns avec les autres en ordre croissant de distances.

Pour chaque joueur j1 résultat de la requête

prendre le premier joueur j2 qui n'a pas déjà été choisi par un autre joueur

envoyer référence de j2 à j1

envoyer référence de j1 à j2

FinPour

supprimer l'ensemble des joueurs qui ont pu être matchés de la base

Le client est identique par rapport à la précédente implémentation.

3.c.2) indications d'implémentation :

Indications sur la base de données spatiale :

La base de données spatiale utilisée est spatialite.
La documentation de spatialite se trouve ici : http://www.gaia-gis.it/gaia-sins/spatialite-cookbook/html/tech-intro.html

Pour utiliser Spatialite dans un programme java il faut mettre dans le build path le driver JDBC sqlite 3.7.8 que l'on peut trouver à l'adresse ci-dessous :
https://www.assembla.com/code/contasystem/subversion/nodes/trunk/lib/sqlite-jdbc-3.7.8-20111025.014814-1.jar?rev=4

L'initialisation de la connexion à une base de données spatialite au sein d'un programme java se fait de la manière suivante :

```
            Class.forName("org.sqlite.JDBC");
            config = new SQLiteConfig();
            config.enableLoadExtension(true);
            conn = DriverManager.getConnection("jdbc:sqlite:nom_table"
                        ,config.toProperties());
            st = conn.createStatement();
            st.execute("SELECT load_extension('/usr/lib/libspatialite.so')");
            st.execute("SELECT InitSpatialMetadata()");
            st.execute("INSERT OR IGNORE INTO spatial_ref_sys (srid, auth_name,
auth_srid, ref_sys_name, proj4text) VALUES (4326, 'moi', 4326, 'WGS84',
'+proh=longlat +ellps=WGS84 +datum=WGS84 +no_defs')");
```

Le schéma pour la table Joueurs choisi est le suivant :
Joueurs(summonerId : integer, duration : integer, geom : POINT)

pour créer cette table il faut écrire le code ci-dessous :

```
st.executeUpdate("CREATE TABLE Joueurs(summonerId INTEGER, duration INTEGER,
PRIMARY KEY(summonerId))");
st.executeUpdate("SELECT AddGeometryColumn('Joueurs', 'geom', 4326, 'POINT',
2)");
```

3. comparaison des trois implémentations :

La plateformes d'exécution :

Les tests ont été effectués des machines de SMALL qui possède chacune 24 processeurs de 6 cœurs chacun. Cependant le logiciel Spatialite dont on a besoin pour tester l'implémentation avec la base de donnée spatiale n'est pas dans installé sur cette machine. On a donc installé une machine virtuelle à l'aide de virt-manager sur une des machines de SMALL mais la machine virtuelle même paramétrée pour posséder 24 processeurs n'en possède jamais plus de 8.
La VM a été créée sur small23, le bridge br0 a déjà été configuré pour que la machine soit accessible de l'extérieur.
Ainsi pour chaque exécution le serveur est lancé sur la VM de small23 et les joueurs sont lancés depuis une machine de SMALL autre que la 23.

sur cette machine virtuelle il faut :
→ installer eclipse (eclipse-standard-kepler-SR2-linux-gtk.tar.gz)
→ installer JRE runtime 6
→ installer sqlite3
→ installer spatialite
→ télécharger sqlite-jdbc-3.7.8-20111025.014814-1.jar et le mettre dans le build path de l'implémentation avec la BD spatiale (clique droit → build path → configure build path → libraries → add external jars)
→ télécharger libspatialite.so.3.0.0
→ ajouter libspatialite.so.3.0.0 dans le LD_LIBRARY_PATH dans le bashrc
(LD_LIBRARY_PATH=path_libspatialite.so.3.0.0
export LD_LIBRARY_PATH)
Informations complémentaires :

Au final les tests ont été effectués avec les données du fichier joueurs_props_newDB.csv qui contient les informations de 499840 joueurs.

Observation : Le fait d'augmenter le nombre de threads côté client permet d'accélérer le traitement (vu que plus de clients peuvent se connecter de manière simultanée au serveur). Cependant le nombre de processeurs ne permet pas toujours de pouvoir avoir un grand nombre de threads. On obtient un connection reset ou un Cannot assign requested address si on cherche a avoir trop de threads simultanément mais que le nombre de processeurs est trop petit.

Afin d'effectuer les tests sur la VM, il a fallu d'abord faire deux tests préliminaires :
tout d'abord il a fallu tester si avec le même nombre de threads côté client et le même nombre de joueurs voulant être matchés le temps d'exécution était le même que le test soit effectué sur la VM ou sur small directement. Pour 100000 joueurs et 2200 treads le temps d'exécution pour les deux architectures est de 4 minutes 27 secondes (moyenne effectuée sur 2 tests sur chaque architecture, avec l'implémentation naïve).
Il a fallu également tester si la répartition des joueurs dans la base est uniforme. C'est à dire il a fallu tester si le fait de faire les tests sur un échantillon de 100000 joueurs plutôt que sur 499840 joueurs a un impact. Les moyennes obtenues au niveau des différents critères sont assez proches. Par contre l'écart type de la distance pour 100000 joueurs est autour de 41 et autour de 45 pour 500000 joueurs. De plus les valeurs maximales d'écart de summonerElo, latence et distance sont bien plus grandes pour les 500000 joueurs que pour les 100000 joueurs.
Ceci s'explique directement en analysant la base de données.

	Sur 499 840 joueurs de la base	Sur les 100 000 premiers joueurs de la base
max(summonerElo)	2 699	2 694
min(summonerElo)	100	100
avg(summonerElo)	995,71	1 187,17
max(latence)	2 227	1 017
min(latence)	0	7,1
avg(latence)	52,57	49,4

<u>Résultats des tests :</u>

Tous les tests ont été effectués avec 2200 threads côtés client (ce nombre peut-être modifiés dans le fichier Joueur.java) sur les 100000 premiers joueurs du fichier joueurs_props_newDB.csv et avec comme exigence d'avoir si possible une distance de 20 pour les critères de summonerElo et de latence.
Tous les tests ont été effectués 10 fois. On a alors supprimer le maximum et le minimum de chaque résultats obtenus. Les chiffres reportés dans les tableaux ci-dessous correspondent à la moyenne des 8 résultats restants.

	Random	Naïf	BD_Spatiale
Temps total pour matcher les 100000 joueurs (en ms)	234 929	264 494	2 016 598
Ecart-type du temps total d'exécution (entre les 8 exécuttions)	431,2	931,2	29 568,8
Nombre de paires de joueurs matchés ensembles et dont l'écart de summonerElo est de 0	27	1 375	1 790
Ecart-type (entre les 8 exécutions) du nombre de joueurs matchés avec un écart de summonerElo de 0	3,3	18,5	22,5
Nombre de paires de joueurs matchés ensemble et dont l'écart de latence est de 0	751	1 454	1 902
Ecart-type (entre les 8 exécutions) du nombre de joueurs matchés avec un écart de latence de 0	19,7	26,3	24,1
Nombre de paires de joueurs matchés ensembles et dont l'écart de distance est de 0	0	40	77
Ecart-type (entre les 8 exécutions) du nombre de joueurs matchés ensembles avec un écart de distance de 0	0,5	7,0	4,7
Nombre de joueurs matchés au bout de 0 ms	60 397	45 344	0
Ecart-type (entre les 8 exécutions) du nombre de joueurs matchés au bout de 0 ms	63,4	609,8	0

	Random	Naïf	BD_Spatiale
Distance moyenne entre les paires de joueurs matchés ensembles	617.8	16,9	18,0
Ecart-type moyen intra-exécution de la distance moyenne	455,2	42,5	55,7
Ecart-type (entre les 8 exécutions) de la distance moyenne	1,3	0,1	0,1
Temps moyen de match des joueurs (en ms)	1	379	39 116
Ecart-type intra-exécution du temps de match des joueurs	3,7	1 865,9	120 731,3
Ecart-type (entre les 8 exécutions) du temps de matchs des joueurs	0	1	639,1

Conclusion :

Les solutions apportées par l'implémentation naïve ainsi que l'implémentation à l'aide de la BD spatiale permettent d'obtenir un matchmaking entre des joueurs de critères assez similaires. Cependant l'implémentation avec la BD Spatiale est beaucoup plus lente.
L'implémentation random quant à elle ne permet pas un bon matchmaking bien que le temps pour matcher les joueurs soit optimal.
Si on prend en compte le temps de match, ainsi que les critères de matchmaking, l'implémentation naïve est la plus performante.
Pour améliorer les performances de la BD spatiale, il est possible de monter la base de données dans tmpfs (sur SMALL cependant les performances de la BD n'ont pas été améliorées malgré le mount).

II. matchmaking centralisé mutli-critères 5 joueurs contre 5 joueurs :

Les interfaces pour les algorithmes centralisé multi-critères 5 joueurs contre 5 joueurs restent similaires aux interfaces utilisées pour les algorithmes centralisés multi-critères 1 joueur contre 1 joueur.

1) implémentation

1.a) implémentation random

1.a.1) pseudo-code

Pour obtenir les équipes de joueurs. Il faut prendre en compte la valeur de premadeSize. Si la valeur de premadeSize est de 1 pour chacun des joueurs. Le pseudo-code de l'algorithme de matchmaking peut-être le suivant :

```
liste = liste de joueurs
sum1 = 0      // correspond à la somme des Elo des joueurs de la première équipe
sum2 = 0      // correspond à la somme des Elo des joueurs de la deuxième équipe
id1 = null    // ensemble des joueurs de la 1 ère équipe
id2 = null    // ensemble des joueurs de la 2 ème équipe

Si liste.size() >= 10 alors
        pour i allant de 1 à 10 faire
                min = MAX_VALUE
                e1 = liste(i)
                pour j allant de 2 à liste.size() faire
                        si !appartient(liste(j), id1) et !appartient(liste(j), id2) alors
                                si (liste(j).summonerElo – e1.summonerElo) < min alors
                                        min = liste(j).summonerElo – e1.summonerElo
                                        e2 = liste(j)
                                FinSi
                        FinSi
                FinPour
                si ((sum1 < sum2) et (e1.summonerElo < e2.summonerElo)) ou ((sum2 < sum1) et (e2.summonerElo < e1.summonerElo)) alors
                        sum1 += e2.summonerElo
                        id1.add(e2)
                        sum2 += e1.summonerElo
                        id2.add(e1)
                sinon
                        sum1 += e1.summonerElo
                        id1.add(e1)
                        sum2 += e2.summonerElo
                        id2.add(e2)
                FinSi
        FinPour
```

1.b) implémentation naïve

1.b.1) pseudo-code

Ici on propose un algorithme qui ne tient pas compte de la valeur de premadeSize des joueurs.

Afin que les valeurs des équipes soient le plus proches possibles on va diviser les joueurs en plusieurs listes. La thread qui s'occupera du matchmaking possèdera une matrice de listes de joueurs. Lorsqu'un joueur se connectera au serveur on regardera sa valeur de summonerElo et sa valeur de latence et on mettra le joueur dans la liste qui convient.
Les liste correspondent à des cases de la matrice. La matrice est divisé en carré de 40 sur 40. Ainsi les joueurs dans une case de la matrice correspondent à des joueurs qui ont un summonerElo et une latence proche. Ces deux valeurs ne différent pas plus de 40 unités.

```
Matrice[][] = matrice de listes [75][75]
dés qu'un joueur j s'est connecté l'insérer dans Matrice[j.summonerElo / 40][j.latence / 40]
Si taille(Matrice[j.summonerElo / 40][j.latence / 40]) == 10
    formerEquipe(j.summonerElo / 40, j.latence / 40)
FinSi
```

formerEquipe se fera de la même manière que pour l'implémentation random c'est à dire formera deux équipes de telles sortes à ce que ces deux équipes soient à peu près équilibrée.

Toutes les secondes une tâche écrira dans un fichier de log le nombre de connexions qu'il y a eu durant la seconde courante.

```
Toutes les 5 secondes une tâche se réveillera. Cette tâche effectuera :
AMatcher = liste d'indices
pour i allant de 0 à taille(Matrice) faire
    pour j allant de 0 à taille(Matrice[0]) faire
        pour tous les joueurs js de Matrice[i][j] faire
            incrémenter la duration de js
            Si js.duration >= 12 alors
                insérer (i, j) dans AMatcher
            FinSi
        FinPour
    FinPour
FinPour
Si !vide(AMatcher) alors
    pour i allant de 0 à taille(AMatcher) faire
        chercherJoueurs(i, j)
    FinPour
FinSi
```

```
chercherJoueurs(i, j)
    int m = 0 ;

    TantQue (taille(Matrice[i][j] < 10 && m < tailleMatrice) {
        m++ ;
        pour k allant de -m à m faire
            pour l allant de -m à m faire
                Si ((i+k < taille(Matrice)) && (i+k >= 0) && (i+l < taille(Matrice))
                && (i+l >= 0)) alors
                    Si (taille(Matrice[i+k][j+l] != 0) alors
                        TantQue (((taille(Matrice[i+k][j+l])) != 0) ||
                        (taille(Matrice[i][j]) != 10))) faire
                            ajouter Matrice[i+k][j+l].First à Matrice[i][j]
                            supprimer Matrice[i+k][j+l].First de
                                Matrice[i+k][j+l]
                        FinTantQue
                        Si (taille(Matrice[i][j] == 10) alors
                            goto Fin
                        FinSi
                    FinSi
                FinSi
            FinPour
        FinPour
    FinTantQue

    Fin : formerEquipe(i, j)
```

1.b.2) indications d'implémentation :

Par la suite le code a été paramétré de telle sorte que les dimensions d'une case de la matrice ainsi que le critère de summonerElo et de latence soient entrés en argument.
Si arg est l'argument entré par l'utilisateur alors les dimensions d'une case de la matrice seront de 3000/arg sur 3000/arg (3000 est une borne maximum pour le summonerElo comme pour la latence), et toutes les valeurs 40 seront remplacées par arg.

2) Comparaison des deux implémentations :

La plateformes d'exécution :

La plateforme d'exécution est similaire à celle utilisé pour les tests centralise_1V1. C'est à dire que le serveur est lancé sur la vm de small23 et les joueurs sont lancés depuis une autre machine small.

100 000 joueurs se connectent, à l'aide d'un pool de threads de 2200 threads, au serveur pour être matchés.
Les résultats présentés ici pour l'algorithme naïf ont été fait de telle sorte d'avoir si possible une distance de 20 pour les critères de summonerElo et de latence (c'est à dire qu'une case de la matrice à une taille de 40 sur 40).

Tous les tests ont été effectués 10 fois et le minimum et maximum ont été supprimés. Ne sont reportés dans ce tableau uniquement les moyens des 8 résultats restants.

	Random	Naïf
Temps total d'exécution (en ms)	234437	327705
Ecart-type (entre les 8 exécutions)	427,1	327,1
Temps moyen pour que les joueurs soit assigné a une équipe et matché contre une autre	10	1931
Ecart-type intra exécution du temps de matchmaking des joueurs	9,7	2031,5
Ecart-type (entre les 8 exécutions) du temps de matchmaking	0	11,2
Distance moyenne entre les équipes adverses	1081,3	66,6
Ecart-type intra exécution de la distance entre les équipes adverses	9,7	2031,5
Ecart-type (entre les 8 exécutions) de la distance entre les équipes adverses	5,4	0,4

Conclusion :

La version Random est faiblement plus rapide que la version Naïve. La version Naïve offrant de bien meilleure distance de matchmaking et donc des combats plus équilibrés que la version Random, elle est plus performantes.

III. matchmaking décentralisé multi-critères 1 joueur contre 1 joueur :

1) Les interfaces :

nœud/Joueur :

```
public interface JoueurItf extends Remote {
	public JoueurImpl matchmaking(JoueurImpl j) throws RemoteException;
	public void jouer(JoueurImpl j) throws RemoteException;
}
```

2) Le pseudo-code :

L'algorithme correspond à l'algorithme de Shafran, sauf que le cache ne sera pas une FIFO comme dans l'algorithme originel mais une structure favorable pour conserver différents critères.

nœud/joueur : (le pseudo-code est identique à celui présenté dans l'algorithme de Shafran)

Reception <requête> d'un client C2 par C1
Si C1 possède des crictères similaires à C2 alors
contacter C2 (appel à la méthode jouer de C2)
Sinon
si un client C3 du cache local de C1 possède des critères similaires à ceux de C2 alors
retourner la référence de C3 à C2
Sinon
insérer C2 dans le cache local
contacter des nœuds voisins (appel à la méthode matchmaking sur ces nœuds voisins au nom de C2)
FinSi
FinSi

Dés que C1 veut jouer
récupérer critères de jeux (depuis la base de données)
TantQue C1 n'a pas joué
Si C1 possède dans son cache local un client C2 qui possède des critères similaires alors
armer un timer
contacter C2 (appel à la méthode jouer de C2)
si expiration du timer sans avoir pu contacter C2
goto Tant Que
Sinon
Fin
FinSi
Sinon
contacter des nœuds voisins (appel à la méthode matchmaking sur ces nœuds voisins au nom de C2)
FinSi
FinTantQue

Quelques liens :

Lien de l'algorithme de matchmaking pour le jeu SMITE :
http://www.smitefrance.fr/index.php?article6/le-fonctionnement-du-matchmaking-de-smite

principe :

file d'attentes ordonnées en fonction des niveaux ELO des joueurs.
Les joueurs jouent à 5 contre 5 dans ce jeu. Une fois qu'il y a dix joueurs dans la liste, le système cherche toutes les combinaisons d'équipe de 5 possible afin de trouver la combinaison où la somme des ELOs de chaque côté est équivalente.

lien qui reprend Knowledge « Matchmaking in P2P e-marketaces : soft constraints and compromise matching » en plus étoffé.
http://www.tmrfindia.org/eseries/ebookV1-C3.pdf