ROBOT D'INDEXATION HTTP HAUTE PERFORMANCE

On rappelle que les supports de cours sont disponibles à
http://mathieu.delalandre.free.fr/teachings/dsystems.html

Mots-clés: robot d'indexation, requête HTTP, temps de réponse, parallélisation, bande passante, politique d'accès, stratégie d'indexation

1. Introduction

1.1. Modalités de réalisation

Ce TP doit être réalisé dans les créneaux impartis modulo le temps de lecture et de préparation. Il se fera de préférence par binôme mais peut se réaliser en monôme ou trinôme. Des prérequis en programmation en Java, en particulier sur les aspects réseaux et systèmes, sont nécessaires à sa réalisation. Ce TP se présente, dans sa rédaction, comme un développement « from scratch ». A partir de différents exemples de code clé, le TP permet la réécriture complète de la plateforme logicielle à mettre en œuvre.

1.2. Robot d'indexation

Ce TP s'intéresse à la problématique des robots d'indexation (ou « Web crawler »). Un robot d'indexation est un logiciel qui explore automatiquement le Web. Il est conçu pour collecter les ressources (pages Web, images, vidéos, documents etc.). Les robots d'indexation sont des composants essentiels au domaine du Web. Ils constituent la base de fonctionnement des moteurs de recherche (Google, Yahoo, etc.).

Il existe deux catégories de robot d'indexation. Les robots dits génériques réalisent une indexation tout-venant du Web. De l'autre côté, les robots spécialisés se concentrent sur un type de ressource particulier (données financières, multimédias, linguistiques, etc.).

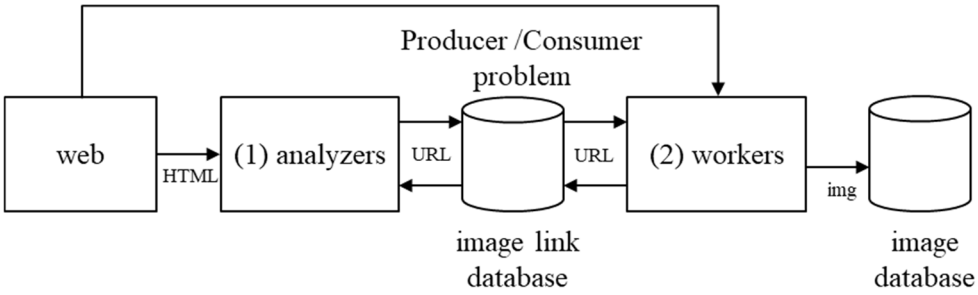
Les robots spécialisés opèrent dans un contexte de recherche concurrentielle et structurée. La recherche est focalisée sur un nombre restreint de site. Les « patterns » d'accès aux sites peuvent être programmés a priori dans les robots pour de meilleures performances en accès. Le développement de tels robots relève d'une double problématique. D'un côté les aspects performances seront à considérer compte-tenu des volumétries de données à traiter. De l'autre côté, il sera nécessaire de mettre en place des stratégies d'indexation respectueuses des politiques d'accès des serveurs Web.

Dans ce TP on s'intéressera plus particulièrement aux robots pour la collecte de données image. Ces robots détectent et collectent les images sur le Web. Les images collectées peuvent servir aux méta-moteurs de recherche image. Contrairement aux données HTML, les données image requièrent des temps d'indexation plus important de par la taille des fichiers considérés (e.g. de 1 à 70 Mpx sous google image, soit ≈ 1-40 Mo).

Comme pour tout robot, l'indexation procède généralement en deux étapes (1) l'identification des liens hypertextes image à partir des pages Web puis (2) la collecte / téléchargement des images. La phase de téléchargement peut œuvrer en assignement dit statique ou dynamique.

En assignement statique, les jeux de liens sont définis a priori puis affectés préalablement à chaque robot. Dans la pratique, il est très difficile d'estimer le temps de téléchargement des données image (estimation des tailles de fichiers, trafic Internet, proximité géographique entre le client http et le serveur, etc.) aussi l'assignement statique résulte dans des interruptions de service des robots lorsque les jeux de liens sont consommés.

L'assignement dynamique est la norme. Dans ce cas, les robots en charge de la collecte (ou « workers ») se synchronisent au travers d'une base données avec les machines en charge l'indentification des liens hypertexte des pages Web (ou « analysers »). Cette synchronisation relève d'un problème Producteur / Consommateur. L'architecture peut être déployée en centralisé (multithreading et moniteur) ou réparti (base de données réseaux).



La charge de téléchargement est surtout portée par les « workers ». Les données des pages Web sont souvent de tailles négligeables au regard des données images. Aussi la problématique répartie centrale est le déploiement des « workers ». Cela relève d'un problème de requête HTPP massivement parallèle. Dans ce TP on abordera cette problématique d'un point de vue architecture centralisée. Après une première mise en œuvre, les questions de performances, parallélisation et stratégies d'indexation seront abordées. Une architecture sera mise en œuvre pour répondre aux différentes contraintes rencontrées.

2. Première mise en œuvre et temps de réponse

2.1.Première mise en œuvre

Q1. Hello World : les robots d'indexation se basent sur les requêtes HTPP. Le code ci-dessous donne un exemple de requête HTTP en Java. Il nécessite de spécifier un simple URL et une taille de buffer (e.g., 1024, 2048, 4096 octets). Mettez ce code en œuvre au sein d'une classe Java. Pensez à réaliser les imports des packages java.io.* et java.net.*. Appliquez ce code à la récupération de contenu image, tel que https://i.stack.imgur.com/o1z7p.jpg

```
protected void trace(String ad, int buffer)  {
        try {
            url = new URL(ad); array = new byte[buffer];
            iS = url.openStream();

            size = 0;
            do     { size = iS.read(array); }
            while(size != -1);
```

```
                iS.close();
                }
            catch (Exception e) {}
            }
private URL url;
private byte[] array;
private InputStream iS;
private int size;
```

Ce code ne met pas en œuvre l'enregistrement des données téléchargées, mais pourrait facilement être entendu dans ce sens. Cependant, dans un objectif d'évaluation des performances de communication TCP, le couplage sauvegarde / téléchargement se présente comme une contrainte à l'évaluation. Pour la suite de ce TP, on s'intéressera surtout aux aspects communication et performances réseaux.

2.2.Temps de réponse serveur

Les données image sur le Web se présentent sous la forme de fichier image seul ou de fichier collection (archive zip ou document pdf). Elles sont souvent volumineuses de par la nature graphique des informations, cela malgré les algorithmes de compression appliqués. Un problème lié est la difficulté d'estimation du temps de téléchargement. Tout d'abord, il très difficile de connaitre a priori les tailles des images à partir des URL. Ensuite, les temps de réponse des serveurs dépendent aussi de différentes conditions comme le trafic sur Internet, la performance en accès du serveur et la distance géographique entre le client et le serveur. Nous proposons d'illustrer ces aspects ici.

Q2. Evaluation du temps de réponse - serveur : vous pouvez caractériser le temps de réponse ou débit d'un serveur à partir du code de la question Q1. Vous capturerez pour cela le temps système à l'aide de la fonction System. currentTimeMillis() à l'ouverture du flux (t_1) puis à la fermeture du flux (t_2) afin d'obtenir le temps de réponse $\Delta_t = t_2 - t_1$. Il sera ensuite possible de reformuler le temps de réponse en débit généralement exprimé en Mbit.s^{-1}. Les fonctions ci-dessous permettent la conversion octet / milli seconde en Mbit.s^{-1}. Il faudra pour cela tracer le volume de données téléchargé via l'accumulation du compteur « size » en Q1.

```
/** From bytes/ms to Mbit/s. */
protected static double toMbits(long tsize, long dt)
        { return toMbit(tsize) / mToS(dt); }

/** From bytes to Mbit. */
protected static double toMbit(long tsize)
        { if(tsize > 0) return ((double)tsize*8) / (1024.0*1024.0); return 0; }

/** From milli seconds to seconds. */
protected static double mToS(long dt)
        { if(dt > 0) return ((double)dt) / 1000.0; return 1; }
```

A partir d'un serveur, tracez l'évolution de son temps de réponse / débit sur un intervalle de temps donné (e.g. plusieurs minutes). Vous pourrez pour cela télécharger en boucle un même fichier donné. Que pouvez-vous en conclure ?

Q3. Evaluation du temps de réponse - base serveur : on se propose dans un second temps d'évaluer les variations des temps de réponse / débit entre serveurs. Le code ci-dessous donne des fonctions pour le téléchargement d'une base de liens HTTP depuis un fichier texte et son affichage. Vous pourrez utiliser ce code pour charger la base « servers.txt » disponible sur le

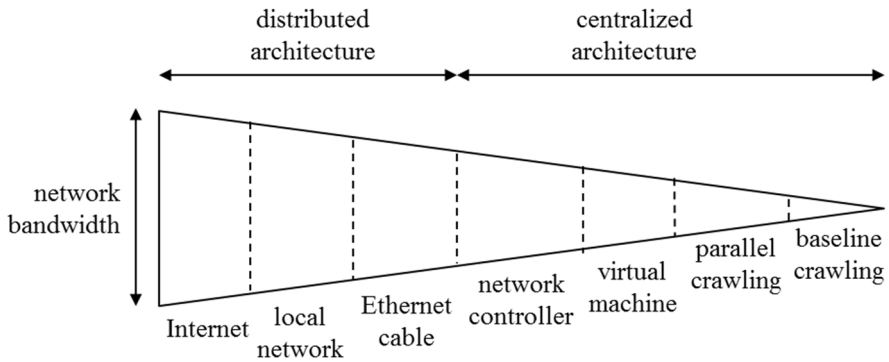
lien du TP. Cette base est constituée de dizaines de liens HTTP d'images jpg, sans redondance domaines, avec garantie de validité. Elle sera à utiliser pour la suite du TP.

```
public static ArrayList<String> sNames = new ArrayList<String>();

/** To read the server list from a text file. */
public static void listRead(String fName) {
        if(fName!=null) {
                sNames.clear();
                try {
                        flowR = new BufferedReader(new FileReader(fName)) ;
                        while((line = flowR.readLine()) != null)
                                { sNames.add(line);  }
                        }
                catch(IOException e) { }
                }
        }
private static BufferedReader flowR;
private static String line;

/** To print the server list. */
public static void printList() {
        for(int i=0; i<sNames.size();i++)
                System.out.println(sNames.get(i));
        }
```

En reprenant le code de la question Q2, caractérisez les temps de réponse des serveurs de la base « servers.txt ». Parallèlement, vous pourrez utiliser votre code pour extraire des informations sur les tailles des fichiers téléchargés ainsi que sur les temps de téléchargement. Que pouvez-vous en conclure ?

2.3.Téléchargement et performances

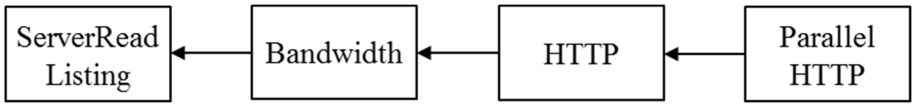
Les robots d'indexation doivent télécharger de gros volumes de données dans des temps contraints. Il est donc important d'assurer de bonnes performances. La solution est de maximiser l'utilisation de la bande passante réseau. Cette bande passante en téléchargement est de l'ordre de quelques dizaines de $Mbit.s^{-1}$ pour une connexion Wifi et de plusieurs $Gbit.s^{-1}$ pour une connexion filaire. Elle est aussi dépendante des performances des câbles Ethernet utilisés (de 100 $Mbit.s^{-1}$ à 10 $Gbit.s^{-1}$), de la carte réseau et de son paramétrage (e.g. de 100 $Mbit.s^{-1}$ à 1 $Gbit.s^{-1}$) ainsi que des éventuelles limitations de machine virtuelle. L'architecture de l'application constitue également un frein qui peut être levé par la parallélisation. Cette approche augmente significativement les performances en téléchargement d'un robot tout en augmentant les temps de réponse d'indexation.



Q4. Indexation parallèle : reprenez le code développé en question Q3 de façon à rendre votre implémentation parallèle. Vous pourrez pour cela étendre votre classe de mise en œuvre en thread avec implémentation de l'interface Runnable.

```
public class MyClass implements Runnable { public void run() {  } }
…
new Thread(new MyClass ()).start();
```

La figure ci-dessous propose une architecture pour cette mise en œuvre, de la lecture de base de liens en mode statique (ServerReadListing) aux classes pour le calcul de bande passante (Bandwidth), connexion HTTP (HTTP) et parallélisation (ParallelHTTP).




```
int n = 128;
ParallelHTTP.listRead("servers.txt");
for(int i=0; i<n; i++)
        new Thread(new ParallelHTTP(i,1024)).start();
        /** 1024 is the buffer size. */
```

Q5. Performances et parallèlisation : dans un deuxième temps, l'objectif sera d'évaluer les performances de votre implémentation parallèle. Pour cela, il est nécessaire de calculer la vitesse moyenne de téléchargement des threads. Cette opération peut se réaliser de deux manières. Tout d'abord au niveau de la carte réseau par logiciel d'analyse (e.g., Wireshark[1]). Cette approche a l'inconvénient d'être globale à la machine et ne peut différencier les accès réseaux du robot des autres accès du système d'exploitation et des logiciels. L'autre approche est au niveau applicatif à l'aide d'un code de synchronisation. Cette solution permet d'isoler les accès réseaux du robot mais ne peut fournir qu'un estimateur de la bande passante réelle. L'estimation reste néanmoins suffisante pour les besoins d'analyse.

Le code ci-dessous donne un exemple de fonctions pour la synchronisation en Java. Deux fonctions sont exploitées pour l'initialisation du calcul clear() et la mise à jour des données de téléchargement bandwidth(). La fonction clearBandwidth() peut être appelée de manière statique à l'initialisation de l'application par passage en argument d'une valeur d'horodatage (en ms) au lancement du premier thread. La fonction bandwidth() est, elle, appelée par chacun des threads soit en cours ou à l'issue du téléchargement en précisant en paramètres le volume de données téléchargé (en octets) et l'horodatage (en ms).

```
private static long ts1, ts2;                   /** The start and end time stamp in ms. */
private static double tsize;                    /** The total downloaded byte size in Mb. */
private static int loop;                        /** To count the loop. */

/** To clear and activate the bandwidth data, ts is the start timestamp. */
public static void clearBandwidth(long ts)
        { ts1 = ts; ts2 = -1; tsize = 0; loop = 0;  }
/**
* The synchronized trace bandwidth,
* to add a downloaded size in byte with the end timestamp.
*/
protected synchronized void bandwidth(long size, long ts) {
        loop++;
        if(size>0)
                tsize += toMbit(size);
```

[1] https://www.wireshark.org/

```
        ts2 = ts; dt = mToS(ts2)-mToS(ts1);
        if(dt>0)
                System.out.println  (
                                                                loop+"\t"
                                                                +tsize+"\t Mbit\t"
                                                                + dt+"\t s\t"
                                                                + (tsize / dt) +"\t Mbit.s"
                                                                );
        }
private double dt;
```

A partir de ce code, vous pourrez évaluer les performances de votre implémentation parallèle. Vous pourrez pour cela cibler les serveurs de la base « servers.txt » et augmenter graduellement le dégrée de parallélisme (e.g. 8, 16, 32, 64 threads) pour évaluer l'impact sur les performances. Pour une caractérisation fine, il sera recommandé de faire appel à la fonction bandwidth() en cours de téléchargement (au sein de la boucle « do while »). Afin de limiter les appels en synchronisation ainsi que le nombre d'observations, il est recommandé d'augmenter la taille du buffet (4096 ou 8192 octets). Illustrer le gain en performance l'impact en performance en fonction du niveau de parallélisme. Que pouvez vous conclure sur le nombre de threads à mettre en place au sein de votre architecture parallèle?

2.4. Politique d'accès

Un aspect complémentaire aux performances en téléchargement est la politique d'accès des serveurs Web. Dans la pratique, les serveurs autorisent un nombre de maximale de connexion par client (e.g. une adresse IP) pour équilibrer les demandes en accès. Une alternative est le calcul par le serveur d'un taux de connexion / s par client. Une surcharge peut résulter dans un rejet des connexions par le serveur et donc des téléchargements manqués pour le robot. Le serveur peut également bannir en accès pour une période donnée le client. Ces aspects sont particulièrement critiques au fonctionnement du robot, on se propose de les illustrer ici.

Q6. Pic de connexion serveur: reprenez l'implémentation parallèle réalisée en Q4, Q5 de façon à évaluer la charge de connexion sur un serveur donné. Vous devrez pour cela synchroniser chacun de vos threads afin de tracer les connexions sur le serveur. Ceci peut se faire au sein de votre code de requête HTTP par l'appel d'une fonction définie en section critique à l'aide du mot-clé synchronized, à l'issue des appels openStream() et close() sur le flux InputStream puis sur la capture d'exception catch(). Le code ci-dessous donne un exemple de code de synchronisation.

```
private static int open, close, max, exc;          /** open is the active connection number. */
                                                   /** close is the close connection number. */
                                                   /** max is the observed maximum active connection number. */
                                                   /** exc is the observed exception number. */


/** To init/clear the connection counting. */
public static void clearEvents() { open = close = max = exc = 0; }

/** To record the event. */
private synchronized void recordEvent(long ts, int mode) {
        switch(mode) {
                case 1:  connect(ts);      /** To call after the openStream() with the InputStream. */
                         break;
                case 2:  close(ts);        /** To call after the close()with the InputStream. */
                         break;
                case 3:  exception(ts);    /** To call when interrupted. */
                         break;
```

```
                }
        }
private void connect(long ts) { open++; max = setMax(open-close); printEvent(ts); }
private int setMax(int v) { if(v > max) return v; return max; }
private void close(long ts) { close++; printEvent(ts); }
private void exception(long ts) { exc++; printEvent(ts); }

private void printEvent(long ts) {
        System.out.println  (
                                                mToS(ts) +"\t s \t"
                                                +"\t"+(open-close)+"\t cnx \t"
                                                +"\t"+exc+"\t exc \t"
                                                +"\t"+open+"\t cnx \t"
                                                +"\t"+close+"\t cnx \t"
                                                +"\t"+max+"\t cnx \t"
                                                );
        }
```

Ce code garantit une synchronisation via l'appel d'une fonction recordEvent() pour la modification en exclusion mutuelle des variables de comptage open, close, exc. Cette fonction prend en paramètre un temps système à obtenir via la fonction System.nanoTime().

Les fonctions connect() et close() appelées au sein de la fonction recordEvent() (mode 1 et 2 du switch case) permettent de tracer l'évolution des connexions au sein des threads. Elles modifient deux variables de comptage open et close. Elles peuvent être appelées à l'issue des fonctions openStream() et close() sur le flux InputStream. Le nombre de connexion active à un instant donné s'obtient à partir de la comparaison de ces deux valeurs open – close. Une variable max retourne le nombre de connexion maximum active observée sur la séquence.

La fonction exception() (mode 3 du switch case) trace les ruptures de connexion et lecture HTTP. Elle permet d'identifier le point de rupture au-delà duquel le serveur rejette le client. Elle doit être appelée depuis le bloc try catch de mise en œuvre de l'échange HTTP. Il sera nécessaire pour ce faire de configurer les timers en lecture et connexion des objets URL, ainsi que de tracer les exceptions SocketTimeoutException. Le code ci-dessous donne un exemple.

```
        ….
        try {
                url = new URL(ad);
                uc = url.openConnection(); uc.setConnectTimeout(7500); uc.setReadTimeout(15000);
                iS = uc.getInputStream();
                /** to do  */
                iS.close();
                }
        catch(SocketTimeoutException e) { /** to do */ }
        catch(IOException e) {}
        ….

private URL url; private URLConnection uc; private InputStream iS;
```

La fonction clearEvents() pourra être appelée de manière statique à l'initialisation de l'application, pour la mise à zéro des valeurs de comptage.

Finalement, il est également possible de compléter le code de traçage des connexions à un serveur avec une fonction de calcul de vitesse de connexion à insérer dans la fonction connect. Le code ci-dessous en donne un exemple.

```
/** To collect the timestamps in a circular buffer and get the connection rate. */
class TimestampBuffer {
        private long tab[];             /** The timestamp circular buffer. */
```

```
private int size, n;   /** The buffer size and position n. */

/** The builder. */
protected TimestampBuffer() { size=2048; tab = new long[size]; clear(); }

protected void clear() { for(n=0; n<size; n++) tab[n] = -1;  n=0; }

/** To push a new timestamp in the circular buffer. */
protected void push(long t) { n = n%size; tab[n]=t; n++; }

/** To compute the connection rate in c.s, it looks in the buffer the timestamps such as t2-t1 > delta. */
protected double rate(long t2, double delta) {
        crate  = 0; gap = sTons(delta);
        for(i=0; i<size; i++) {
                t1 = tab[i];
                if(window(t1,t2,gap))
                        crate = crate + 1;
                }
        return crate / delta;
        }
private double crate; private long gap, t1; private int i;

/** To convert a gap in second (double) to nano second (long). */
private long sTons(double v)
        { return (long)(v * Math.pow(10,9)); }

/**
* To test if the timestamp is in the window gap.
* using the rule t2-t1 < gap
*/
private boolean window(long t1, long t2, long gap) {
        if(t1 != -1)
                if((t2-t1)<gap)
                        return true;
        return false;
        }

}
```

Vous pourrez mettre en œuvre ce code depuis un programme / thread pour lancer une « charge » (ou « attaque ») de connexion sur un serveur de votre choix (e.g. 1024, 2048 ou 3096 connexions simultanées sur un même serveur). Un exemple de code est donné ci-dessous. Dans la pratique, la majorité des serveurs bloquent / rejettent les connexions au-delà d'un certain seuil (e.g. 700 à 1500 connexions actives). Le contrôle peut aussi se faire sur le taux de connexion par seconde du client. Aussi, pour faire passer un grand nombre de connexion sur un serveur faut-il lisser la charge. Un moyen est de contrôler par mécanisme de trigger le lancement des threads, de façon à espacer les connexions. Le paramétrage de la valeur de temporisation du trigger dépend de la réponse du serveur, de la taille des données à télécharger, de la bande passante locale et du seuil en blocage du serveur. Il est donc très dépendant du cas d'échange.

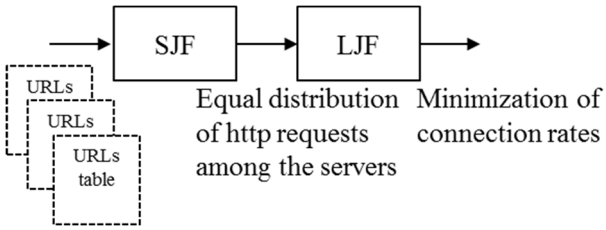
```
try     {
        for(int i=0; i<1024; i++) {
                new Thread(new MyThread()).start();
                Thread.sleep(100);
                }
        }
catch(InterruptedException  e) {}
```

Mettez en œuvre ce code pour évaluer les réponses des serveurs de la base « servers.txt » aux charges de connexion. Vous pourrez, pour cela, paramétrer au cas par cas les valeurs de

temporisation pour vous assurer du non blocage de votre robot d'indexation sur des serveurs donnés. Différentes stratégies peuvent-être mises en œuvre (i) en pic de charge (faible paramètre de temporisation et grand nombre de thread) (ii) en agrégation lente (fort paramètre de temporisation et nombre de thread) (iii) en incrémental (réduction progressive du paramètre de temporisation). Que pouvez-vous en conclure?

3. Stratégie d'indexation

Les différentes questions précédentes abordées dans ce TP ont permis d'illustrer les contraintes liées au développement d'un robot d'indexation d'images: difficultés d'estimation des temps de téléchargement, fort impact du parallélisme sur les performances, contraintes de charge / connexion sur les sites / serveurs, etc. Pour répondre à ces différents aspects, un ordonnanceur peut être implémenté afin de contrôler la stratégie de visite du robot. Considérant des tables de liens / URLs d'images à télécharger sur des serveurs donnés (avec une table par serveur), on peut appliquer un critère SJF (au sens nombre de connexion minimum) afin de garantir une stratégie de visite respectueuse. Le critère JSF peut-être décliné, au besoin, en pondération en fonction des performances observées de chacun des serveurs. En cas d'équivalence du nombre de connexion, les dates de dernière connexion peuvent être utilisées pour limiter la saturation sous la forme d'un critère LJF (au sens temps d'interconnexion maximal prioritaire).



Q7. Ordonnancement et indexation parallèle: on se propose ici de de mettre en œuvre des threads en continue (boucle while) téléchargeant à la suite plusieurs fichiers tout en basculant les serveurs cibles dans le cadre d'une stratégie d'indexation. Pour les besoins de l'exemple et facilité d'implémentation, on considérera ici des tables de serveur infinies i.e. un même fichier en boucle sera téléchargé par serveur.

Une première approche « hello world » consiste à laisser chaque thread en charge de la collecte des données sur l'ensemble des serveurs avec un lancement asynchrone. Il suffit pour cela de reprendre le code de la question Q4 en substituant la boucle « for » par une boucle « while » et d'initialiser l'index d'exploration « i » des serveurs avec le numéro du thread N (i.e. i = N % nombre total de serveur). Sans garantir un calcul optimal des critères SJF / LJF, cette approche a le mérite de la simplicité.

L'alternative est ensuite la mise en place d'un ordonnanceur SJF/LJF avec calcul optimal des critères. Le code ci-dessous illustre comment mettre en œuvre une telle architecture via un thread et un code d'ordonnancement. Le code d'ordonnancement se compose de 3 fonctions synchronisées au travers d'une fonction en exclusion mutuelle schedule(): la fonction de sélection du serveur cible cmin(), les fonctions de mise à jour des tables d'ordonnancement close() et add(). Différentes fonctions sont également proposées pour le calcul les statistiques d'ordonnancement et affichage statistics(), print().

```
/** The scheduler.  */
class Scheduler extends HTTPB {
```

```
	private static byte[] ptable;		/** The priority table. */
	private static long[] ctable;		/** The connection table, the timestamp. */
	private static long[] lctable;		/** The last connection table, the timestamp. */
	private static long cs;			/** To trace the context swith. */

	/** To init the scheduler. */
	public static void clearScheduler() {
		if(sNames != null)
			if(sNames.size() > 0) {
				ptable = new byte[sNames.size()];
				ctable = new long[sNames.size()];
				lctable = new long[sNames.size()];
				for(int i=0; i< ptable.length; i++)
					{ ptable[i] = 0; ctable[i] = 0; lctable[i] = 0; }
				cs = 0;
			}
		}

	/** To return the scheduled server. */
	protected synchronized int schedule(int loop, int aT, long st) {
		if(loop!=0)				/** If we have a first scheduling round, we close the connexion. */
			close(aT);
		aT = cmin(); add(aT); cs++;	/** The min function returns the server target, the add function registers the scheduling operation. */
		statistics(st,aT); print();	/** To print the scheduling criteria. */
		return aT;
		}


	/** To close a connection (remove a connection at given position.) */
	private void close(int aT)
		{ ptable[aT]--; }

	/** To return the scheduled server based on the cmin criteria. */
	private int cmin() {
		cn = Byte.MAX_VALUE;
		aT = 0; ts  = Long.MAX_VALUE;
		for(int i=0; i<ptable.length;i++) {
			if(ptable[i]<cn)
				{ aT = i; cn = ptable[i]; ts = ctable[i]; }
			if((ptable[i]==cn)&(ctable[i]<ts))
				{ aT = i; cn = ptable[i]; ts = ctable[i]; }
			}
		return aT;
		}
	private byte cn; private int aT; private long ts;

	/** To add a connection (add a connection at given position.) */
	private void add(int aT) {
		ptable[aT]++;
		if(ctable[aT]!=0)
			lctable[aT] = ctable[aT];
		ctable[aT] = System.nanoTime();
		}

	/** To set the scheduling criteria. */
	private void statistics(long ts, int aT) {
		/** to fix the event timestamp dt. */
		dt = ctable[aT] - ts;

		/** To compute the cmin, mean, max. */
		cmin = Byte.MAX_VALUE; cmean = 0; cmax = Byte.MIN_VALUE;
		for(int i=0; i<ptable.length;i++) {
			if(ptable[i]<cmin)
				cmin = ptable[i];
```

```
                        cmean += ((double)ptable[i]);
                        if(ptable[i]>cmax)
                                cmax = ptable[i];
                        }
                cmean = cmean / ((double)ptable.length);

                /** To get the connexion rate. */
                cr = ((double)cs) / nToS(dt);

                /** To get the mean connexion gap. */
                cgap = 0; n = 0;
                for(int i=0; i<ctable.length;i++) {
                        if((ctable[i]!=0)&(lctable[i]!=0))
                                { n = n+1; cgap += ctable[i] - lctable[i]; }
                        }
                cgap = cgap/n;
                }

        private long dt;                                                        /** The event timestamp.
*/
        private byte cmin, cmax; private double cmean;  /** To compute the connexion min, mean and max. */
        private double cr;                                                      /** To compute the mean
connexion rate per second. */
        private long cgap; private long n;                              /** To compute the connexion gap. */

        /** The backup function. */
        protected void print() {
                System.out.println(
                                htp(nToS(dt))+"\t s \t"
                                +"cmin"+"\t"+cmin+"\t"
                                +"cmean"+"\t"+htp(cmean)+"\t"
                                +"cmax"+"\t"+cmax+"\t"
                                +"crate"+"\t"+htp(cr)+"\t c/s \t"
                                +"cgap"+"\t"+htp(nToS(cgap))+"\t s \t"
                                );
                }

        }
```

L'ordonnanceur peut ensuite être mis en œuvre depuis un thread principal, de manière équivalente aux questions Q5 et Q6, tel que illustré ci-dessous. Veillez à initialiser l'ordonnanceur via la fonction statique clearScheduler() au lancement de l'application.

```
        private long st;                /** The start time. */

        public MyClass(long st) { this.st = st; }

        /** The run method. */
        public void run() {
                loop = 0; aT = 0;
                while(true) {
                        aT = schedule(loop, aT, st); loop++;
                        trace(st, aT, sNames.get(aT), 2);
                        }
                }
        private int loop, aT;
```

Mettez en œuvre un tel ordonnanceur. Vous pourrez pour cela définir un niveau de parallélisme permettant de bonnes performances dans votre système (e.g. 64, 128, 256 threads). Calculez tout d'abord, à l'aide du code de la question Q5, la vitesse de téléchargement moyenne de votre système sur un tel paramétrage. Vous pourrez, pour les besoins de comparaison, analyser les performances de votre bande passante via des

plateforme en ligne (e.g. https://www.nperf.com/). Visualisez ensuite finalement les retours / statistiques d'ordonnancement. Que pouvez-vous en conclure ?

Annexes

Pour réaliser ce TP, vous aurez deux alternatives.

Machines propres: la première sera de développer sur votre propre machine en connexion Wifi (UTspot) sur le réseau de l'école. Il sera nécéssaire pour cela de disposer d'un SDK et IDE Java. Compte-tenu des contraintes de bande passante de l'Université, les communications via le réseau Wifi de l'Université seront bornées (de 20 à 100 $Mbit.s^{-1}$ en moyenne à tester avec un service de performance en ligne[2]).

Machines de l'école: la seconde alternative sera de travailler à partir d'une machine de l'école (sous machine virtuelle et environnements de développement). Rendez-vous sur le répertoire « VM_Production » puis lancez la machine virtuelle « DEVELOPPEMENT » directement à partir de la version production (i.e. il n'est pas nécessaire de recopier la machine dans le répertoire « VM_Source »). On rappelle que le lancement de la machine s'effectue par simple clic sur le fichier « .vmx ». La machine lancée correspond à une image de l'OS incluant un SDK et IDE Java (Java SE, Eclipse, etc.). Vous pourrez travailler à partir de l'IDE Eclipse, ou directement en ligne de commande et éditeur texte selon vos préférences. Dans le dernier cas, vous devrez inclure dans la variable système « path » le chemin des exécutables Java (i.e. répertoire « bin ») du SDK pour utiliser le compilateur (par défaut, seul le chemin de la JRE est préconfiguré et donc l'appel de l'interpréteur « java »).

L'accès réseaux est borné de par la configuration / architecture des machines et bande passante réseaux de l'école. Les différentes machines sont configurées avec des cartes 1 $Gbit.s^{-1}$ sur une bande passante de 10 G $bit.s^{-1}$.

[2] https://www.nperf.com/