FONDAMENTAUX DE LA COMMUNICATION TCP EN JAVA

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

Mots-clés: premières mises en œuvre TCP, architecture client / serveur, paramètres de socket, échange de données texte, temporisation, tampon et transfert de données, échange large flux, cas d'usage – serveur TCP concurrent de fichier texte

1. Introduction

1.1.Modalités d'évaluation

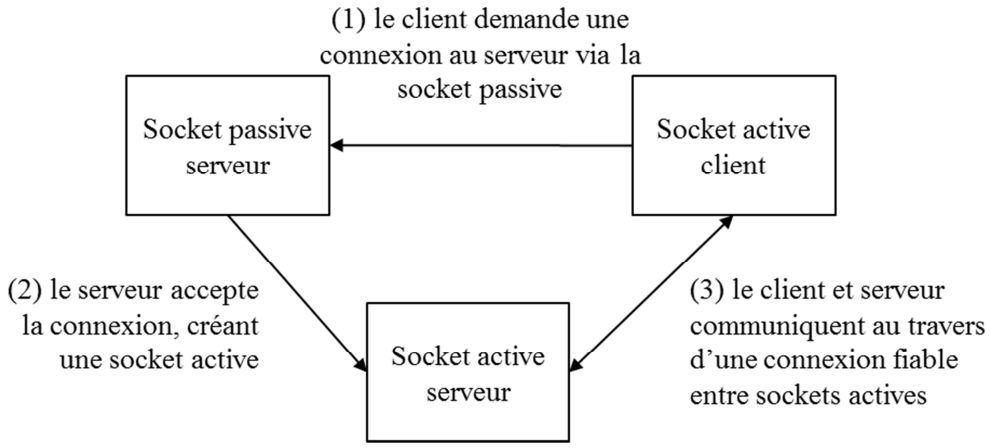
Ce TP doit être réalisé dans les créneaux impartis modulo le temps de lecture et de préparation (soit un volume recommandé de ½ h par séance de 2h). Il se fera de préférence par binôme, les monômes et trinômes sont autorisés et ce choix sera pris en compte pour l'évaluation. Le TP fera l'objet d'une revue de code lors de la rédaction du compte-rendu d'évaluation sur la matière. Dans sa forme, ce TP introduit les notions sur la programmation réseaux Java et des questions de réalisation. Pour plus de lisibilité, les parties « question » apparaissent en encadré dans le corps du TP permettant une double lecture du sujet.

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. Cependant, pour des questions d'illustration une plateforme logicielle clé en main « tcp.jar » est mise à disposition sur le lien du cours. L'aide sur cette plate-forme peut s'obtenir sous le shell en ligne de commande « java -jar tcp.jar –help ». Finalement, l'annexe du TP rappelle les modalités sur les plateformes et réseaux de développement nécessaires à la réalisation du TP.

1.2.Communication TCP Java

Ce TP s'intéresse à la communication TCP Java. Java est un langage très utilisé pour le développement de systèmes distribués, cela pour différentes raisons : portabilité via les machines virtuelles, interopérabilité via les mécanismes de sérialisation d'objets, surcouche de programmation distribuée RMI, etc. Il constitue donc un choix judicieux pour le développement de ce type de système, d'où cette orientation pour ce TP.

Concernant la communication TCP, celle-ci suit une spécification et peut donc être implémentée sous différents langages et plateformes sans distinctions apparentes e.g. C/C++, Microsoft socket « Winsock - Windows », « Berkeley socket - Unix », etc. Dans ce contexte, la question du langage est principalement qu'une affaire de syntaxe. L'implémentation d'une communication TCP passe par la manipulation d'objets/structures socket et des primitives de communication associées (« bind », « close », « listen », « accept », « connect », « read », « write »). On distingue deux catégories de socket, les sockets passives en charge de l'établissement des connexions à partir du serveur, et les sockets actives pour gérer les communications TCP client-serveur. Le schéma suivant rappelle ces principes.



Dans le cadre du langage Java, différents composants sont mis à disposition au sein de l'API pour la mise en œuvre de communication TCP, au travers des packages java.net et java.lang.

Le package java.net fournit un panel de classes pour la création et la gestion des sockets, dont les principales sont résumées dans le tableau ci-dessous.

ServerSocket	socket passive coté serveur
Socket	socket active côté client et serveur, à la suite d'un « accept » réussi
InetSocketAddress	adresse de socket i.e. adresse IP & port

En complément, le package java.lang fournit les mécanismes nécessaires pour la parallélisation des applications. En effet, les primitives de communication TCP étant à connotation système (résultant dans des blocages éventuels des applications), une modélisation sous forme de tâche est nécessaire. Pour ce faire, Java permet la modélisation des applications sous forme de « Threads », exécutés au sein de la machine virtuelle Java. Il existe deux approches pour la modélisation des « Threads » :

(1) soit par implémentation d'une interface « Runnable », Java ne supportant l'héritage multiple, cette solution présente l'avantage de tirer parti d'un éventuel héritage tout en permettant une implémentation « Thread » via la redéfinition de la méthode « run ».
(2) soit par héritage de la classe « Thread », permettant ainsi de tirer parti de toutes les fonctionnalités de cette classe pour une meilleure gestion du parallélisme.

Le code ci-dessous donne un exemple de communication TCP entre un client et un serveur, à partir d'une implémentation « Thread » par interface « Runnable ».

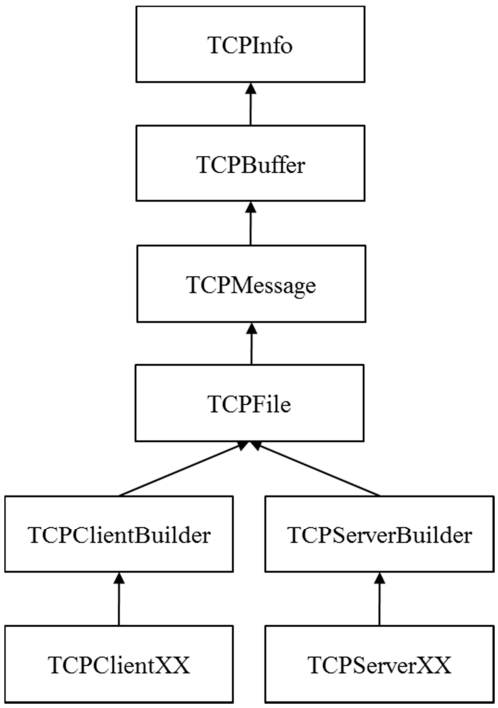
- Les deux packages java.net et java.lang sont importés (le package java.lang l'est par défaut).
- Dans ce programme, la création des objets « adresse » est gérée dans les constructeurs respectifs. Les opérations relatives à la gestion des sockets passives et actives sont reportées au sein des méthodes « run ».
- Il est à noter que les opérations standards de gestion de socket (i.e. « bind », « close », « listen », « accept », « connect », « read », « write ») sont sujettes à des exceptions Java « IOException ». Dans le code proposé, une gestion systématique des interruptions est mise en œuvre via les blocs « try / catch ». Une autre alternative est la déclaration des méthodes de classe en « throws IOException » et la gestion des interruptions depuis le programme principal.

- Java propose une écriture solidarisée pour les opérations de construction et de connexion de la socket côté client via le constructeur « new Socket(hostname, port) ». De même, côté serveur, la construction de la socket passive a été solidarisée de l'opération « bind » via le constructeur « new ServerSocket(port) ». Les paramètres de socket sont définis par défaut sur l'adresse locale « localhost » et le port 8085, mais peuvent être changés selon les configurations locales (e.g. port de 0 à 65535, adresse IPV4 e.g. 127.0.0.1).
- On rappelle également la syntaxe pour le lancement des « Threads » depuis le programme principal.

```
import java.net.*;
import java.io.*;
import java.util.*;

class TCPClient implements Runnable {

	private Socket s;				// the client socket
	private InetSocketAddress isA;			//	the remote address

	/** The builder. */
	TCPClient() {
		s = null;
		isA = new InetSocketAddress("localhost",8085);
		}

	/** The main method for threading. */
	public void run() {
		try {
			System.out.println("TCPClient launched ...");
			s = new Socket(isA.getHostName(), isA.getPort());
			System.out.println("Hello, the client is connected");
			s.close();
			}
		catch(IOException e)
			{ System.out.println("IOException TCPClient"); }
		}

	}

class TCPServer implements Runnable {

	private ServerSocket ss; private Socket s;	// the passive and active sockets
	private InetSocketAddress isA;			// the address

	/** The main method for threading. */
	TCPServer() {
		ss = null; s = null;
		isA = new InetSocketAddress("localhost",8085);
		}

	/** The main method for threading. */
	public void run( ) {
		try {
			System.out.println("TCPServer launched ...");
			ss = new ServerSocket(isA.getPort());
			s = ss.accept();
			System.out.println("Hello, the server accepts");
			s.close(); ss.close();
			}
		catch(IOException e)
			{ System.out.println("IOException TCPServer"); }
		}

	}
```

```
new Thread(new TCPServer()).start();
new Thread(new TCPClient()).start();
```

De manière à capitaliser le code pour la suite du TP, on se propose de développer une application telle que décrite ci-dessous. Cette application repartira du code mise en œuvre précédemment basé sur une implémentation « Thread » via l'interface « Runnable ». On se propose d'exploiter les possibilités d'héritage ouvertes via cette implémentation, pour venir enrichir l'application par différentes fonctionnalités qui seront développées au long du TP. L'objectif est de restreindre le développement de toute nouvelle application, ou mise en œuvre, à des classes compactes « TCPClientXX, TCPServerXX » exploitant les composants logiciels et fonctions des classes héritées.



Une première démarche consiste à séparer le code d'initialisation des sockets (constructeurs et variables du code présentés en introduction) du code de mise en œuvre de la communication (méthodes « run » du code présenté en introduction). Cette séparation peut être mise en œuvre via le mécanisme de protection « protected » des membres de classes internes à un package, généralement non mentionné. Vous pourrez réaliser cette séparation par l'implémentation des classes « TCPClientBuilder, TCPServerBuilder » et de leurs classes dérivées « TCPClientHello, TCPServerHello ». Le code ci-dessous donne, à titre d'exemple, une implémentation possible pour le client.

```
class TCPClientBuilder {

        Socket s;
        InetSocketAddress isA;

        TCPClientBuilder() {
                s = null;
                isA = null;
                }

        protected void setSocket() throws IOException {
```

```
                isA = new InetSocketAddress("localhost",8085);
                s = new Socket(isA.getHostName(), isA.getPort());
                /** we can include more setting, later … */
                }

        }

class TCPClientHello extends TCPClientBuilder implements Runnable {


        public void run() {
                try {
                        System.out.println("TCPClientHello launched ...");
                        setSocket();
                        System.out.println("Hello, the client is connected");
                        s.close();
                        }
                catch(IOException e)
                        { System.out.println("IOException TCPClientHello"); }
                }
        }
```

2. Fondamentaux de la communication TCP

2.1.Première mise en œuvre

Q1. Reprenez le code présenté en introduction pour mettre en œuvre votre première communication TCP. Pour ce faire, il est juste nécessaire d'importer ce code et de le mettre en œuvre depuis un « main » lançant un ou deux « Threads » d'exécution. Vous pourrez, si vous le souhaitez, maintenir vos échanges entre votre client et votre serveur en local, ou échanger avec des applications tierces développées par d'autres étudiants ou l'enseignant. Vous devez, pour cela, échanger vos adresses et ports de communication et modifier l'instanciation de l'objet « InetSocketAddress » côté client.

Q2. Veillez, dans un deuxième temps, à mettre en place une architecture en désolidarisation de vos classes constructeur / mise en œuvre de la question Q1. On pourra mettre en œuvre le code désolidarisé au travers de deux classes « TCPClientHello, TCPServerHello » dérivées de « TCPClientBuilder, TCPServerBuilder ». Veillez à bien tester la validité de votre code en évaluant la communication client-serveur mise en œuvre en Q1, sur la base de la nouvelle organisation de votre code. Les classes « TCPClientBuilder, TCPServerBuilder » ainsi développées pourront être utilisées dans la suite du TP pour une mise en œuvre plus rapide de vos applications de communication TCP.

2.2.Paramètres de socket

Une fois la première connexion mise en œuvre, on se propose d'analyser plus en détails les paramètres des sockets et leur évolution au travers des appels des primitives de gestion de la communication TCP (i.e. « bind », « close », « accept », « connect »). Différents paramètres pourront être étudiés comme le protocole d'adresse (IPV4 et/ou IPV6), les valeurs d'adresse et ports locaux et distants, les attributs de limitation « i.e. bound » et de fermeture « i.e. closed », les tailles des tampons en envoi et en réception, les valeurs de temporisation et le mode de fermeture de la socket « i.e. soLinger ». Le tableau ci-dessous donne les différentes méthodes des classes « ServerSocket, Socket » permettant la récupération des paramètres.

Adresse et port locaux	getLocalAddress(),getLocalPort()
Adresse et port distant	getInetAddress(),getPort()

Paramètre de fermeture et de limitation	isClosed(),isBound();
Taille des tampons en envoi et réception	getSendBufferSize(),getReceiveBufferSize()
Valeur de temporisation	getSoTimeout()
Mode de fermeture de la socket	getSoLinger()

Ces méthodes présentent (pour la plupart d'entre elles) une convention de nommage identique dans les deux classes, mais doivent être appelées de manière distincte au travers de deux méthodes de lecture. De par le nombre important de paramètres considérés, la déclaration d'une structure (i.e. classe interne sans méthode) peut constituer une solution pour l'allégement de l'écriture entre les méthodes de lecture et une méthode d'affichage commune. Les lectures des paramètres de taille de tampon, de temporisation et de paramètre de fermeture « i.e. soLinger » sont sujettes à des exceptions de type « SocketException, IOException » à prendre en compte. Un autre point important est la compréhension de la primitive « accept », qui provoque du coté client l'allocation des tampons locaux. L'accès aux tailles de tampon ne pourra donc se faire préalablement à l'appel de la primitive « accept » ou à l'issue de l'appel de la primitive « close ». Il en est de même pour les paramètres de temporisation et de fermeture, qu'il est préférable d'extraire lorsque la socket est active. Egalement, il n'existe pas de méthode permettant le test du protocole IP mais celui-ci peut être identifié par test du typage des objets « InetAddress » (classe abstraite) en objets « Inet4Address, Inet6Address ». Finalement, il semble opportun de concaténer les affichages des paramètres au travers d'un seul appel de fonction « System.out.println » de manière à garantir une atomicité à l'affichage. De façon à respecter l'ensemble de ces contraintes et permettre une mise en œuvre rapide de la lecture des paramètres socket, nous donnons ci-dessous le code d'une classe « TCPInfo ».

```
/** The TCPInfo to display the socket's info. */
class TCPInfo  {

	/** The internal class/structure to trace the socket parametters.  */
	class SocketInfo {
		String lA,rA,tC;
		int lP,rP,sbS,rbS,tO,soLinger;
		boolean bounded,connected,closed,isIPV6,noDelay;
		SocketInfo() {
			lA=rA=tC=null;
			lP=rP=sbS=rbS=tO=soLinger=-1;
			bounded=connected=closed=isIPV6=noDelay=false;
			}
		}

	/** To print the passive sever socket's parameters. */
	void ssInfo(String event, ServerSocket ss) throws SocketException,IOException {
		ssI = new SocketInfo();

		ssI.isIPV6 = isIPV6(ss.getInetAddress());

		ssI.lA = getAdressName(ss.getInetAddress());
		ssI.lP = ss.getLocalPort();
		ssI.bounded = ss.isBound();
		ssI.closed = ss.isClosed();
		if(!ssI.closed) {
			ssI.tO = ss.getSoTimeout();
			ssI.rbS = ss.getReceiveBufferSize();
			}
		print(event,ssI);
		}
	private SocketInfo ssI;

	/** To print the active server socket's parameters. */
	void sInfo(String event, Socket s) throws SocketException,IOException {
		sI = new SocketInfo();

		sI.isIPV6 = isIPV6(s.getInetAddress());

		sI.lA = getAdressName(s.getLocalAddress());
```

```
		sI.lP = s.getLocalPort();
		sI.rA = getAdressName(s.getInetAddress());
		sI.rP = s.getPort();
		sI.bounded = s.isBound();
		sI.connected = s.isConnected();
		sI.closed = s.isClosed();

		if(!sI.closed) {
			sI.tO = s.getSoTimeout();
			sI.soLinger = s.getSoLinger();
			sI.sbS = s.getSendBufferSize();
			sI.rbS = s.getReceiveBufferSize();
			//sI.noDelay = s.getTcpNoDelay();
			//sI.tC = Integer.toHexString(s.getTrafficClass());
			}

		print(event,sI);
		}
	private SocketInfo sI;

	private static String getAdressName(InetAddress iA) {
		if(iA != null )
			return iA.toString();
		return null;
		}

	private static boolean isIPV6(InetAddress iA) {
		if(iA instanceof Inet6Address)
			return true;
		return false;
		}

	private void print(String event, SocketInfo sI) {
		System.out.println	(
					event+":\n"
					+"IPV6: "+sI.isIPV6+"\n"
					+"local \tadress:"+sI.lA+"\t port:"+sI.lP+"\n"
					+"remote \tadress:"+sI.rA+"\t port:"+sI.rP+"\n"
					+"bounded: "+sI.bounded+"\n"
					+"connected: "+sI.connected+"\n"
					+"closed: "+sI.closed+"\n"
					+"timeout: "+sI.tO+"\tso linger: "+sI.soLinger+"\n"
					+"buffer \tsend:"+sI.sbS+"\treceive:"+sI.rbS+"\n"
					);
		}

	}
```

Cette classe propose deux fonctions centrales « sInfo, ssInfo ». Elles permettent l'affichage des paramètres des sockets actives et passives en argument de fonction. Le paramètre « event » permet lui de labéliser l'appel de la fonction dans le code de mise en œuvre. Dans la pratique, la classe « TCPInfo » peut-être incorporée via une relation d'héritage avec les classes de construction « TCPClientBuilder, TCPServerBuilder ». Compte tenu que la communication TCP opère en mode connecté, le traçage des paramètres peut s'illustrer à l'issue de la construction et fermeture des sockets et acceptation de connexion, côté client et serveur. Le code ci-dessous donne un squelette de mise en œuvre pour le serveur.

```
class TCPServerBuilder extends TCPInfo { …. }

class TCPServerInfo extends TCPServerBuilder implements Runnable {

	public void run( ) {
		try {
			setSocket(); ssInfo("The server sets the passive socket", ss);
			s = ss.accept(); sInfo("The server accepts the connexion",s);
			s.close(); sInfo("The server closes a connexion",s);
			ss.close(); ssInfo("The server closes the passive socket", ss);
			}
		catch(IOException e)
```

```
                { System.out.println("IOException TCPServerInfo"); }
        }

    }
```

Q3. Exploitez les fonctionnalités de la classe « TCPInfo », au travers des méthodes « sInfo, ssInfo » pour extraire et afficher les paramètres des sockets. Veillez à spécifier une relation d'héritage au sein des classes « TCPClientBuilder, TCPServerBuilder ». On pourra mettre en œuvre le traçage des paramètres au travers de deux classes « TCPClientInfo, TCPServerInfo » dérivées de « TCPClientBuilder, TCPServerBuilder ». Ces classes effectueront une simple création / acceptation et fermeture des sockets côté client et serveur. Observez l'évolution des paramètres des sockets à différents instants de la communication (création, connexion et fermeture) pour les sockets actives et passives.

2.3.Echange de données texte, tampon et temporisation

On se propose dans une étape suivante de mettre en place un échange de données TCP entre un client et un serveur. Pour ce faire, on illustrera l'échange sur les données texte (ou ASCII) bien que la communication TCP puisse s'appliquer à tout type de données. Pour mettre en œuvre un tel échange, il est nécessaire tout d'abord de travailler à partir des flux sockets récupérables à partir des méthodes « getInputStream(), getOutputStream() ». Les objets retournés sont de type « InputStream, OutputStream ». Ils permettent d'accéder aux méthodes « read, write » pour le transfert de données via les sockets. Ces méthodes travaillent à partir de flots d'octets aisément formatables en chaines de caractères « String » via les méthodes « getBytes() » et le constructeur « String(byte[] bytes, int offset, int length) ». Le code ci-dessous donne un exemple en lecture et écriture. Suite à l'opération d'écriture « write », l'opération « flush » permet de forcer le transfert des données sur la connexion TCP. A l'issue des échanges, les flots d'entrée et de sortie doivent être fermés à l'aide de l'instruction « close ». Il est d'usage de contrôler les ouvertures et fermetures des flux « InputStream, OutputStream » en début / fin d'exécution de la méthode « run ».

```
String msOut = "Aujourd'hui, TP ASR Java." ;
OutputStream out = s.getOutputStream();
byte[] buffer = msOut.getBytes();
out.write(buffer);
out.flush();
out.close();

----------

InputStream in = s.getInputStream();
byte[] buffer = new byte[8192];
int count = in.read(buffer);
String msIn = new String(buffer,0,count) ;
in.close();
```

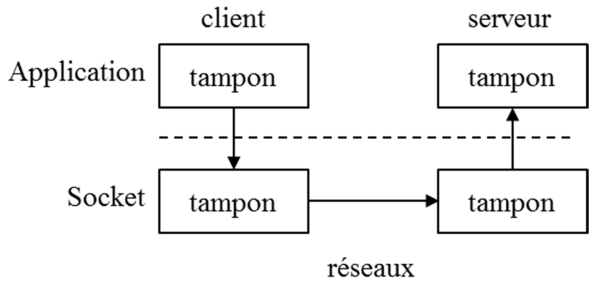
Pour illustrer l'échange de données texte, on propose de mettre en place un serveur de flux de données texte (e.g. type RSS). Les données texte seront envoyées de différents clients vers un serveur. Le serveur œuvrera comme service de lecture / affichage pour les clients. Le flux texte côté client pourra être généré par saisie clavier ou synthétiquement. L'échange TCP se fera dans un seul sens (des clients vers le serveur). Compte tenu de la nature « full duplex » de la communication TCP, l'échange peut être inverse voir à double sens.

Q4. Mettez en place un serveur et un client pour l'échange de flux texte, ou le serveur œuvrera comme service d'affichage pour les différents clients. Pour cela, implémentez deux

classes « TCPClientMessage, TCPServerMessage » dérivées de « TCPClientBuilder, TCPServerBuilder ». Les variables de flux « OutputStream, InputStream » pourront enrichir les classes « TCPClientBuilder, TCPServerBuilder » et être instanciées à partir des méthodes « run ». On se limitera tout d'abord sur cette question à un échange de données texte de quelques octets ayant recours à un unique appel des primitives « read, write ».

Un élément important lié aux opérations de lecture / écriture et la gestion des tampons applicatifs. Ces tampons sont utilisés en paramètre des méthodes « read, write » sur les flux d'entrée / sortie « InputStream, OutputStream ». Il est d'usage d'initialiser ces tampons à la création des sockets, en lien avec l'allocation des tampons systèmes comme par exemple dans la méthode « setSocket() ». En effet, les opérations de lecture / écriture sont en mode bloquant. Une mauvaise adéquation entre les tailles des tampons applicatifs / sockets peut donc résulter dans une augmentation des changements de contexte opérés par l'ordonnanceur de la machine virtuelle Java.

Comme illustré en Q3, les tailles des tampons sockets peuvent aisément être récupérées à l'aide des instructions « getSendBufferSize(), getReceiveBufferSize() ». Compte tenu de l'allocation à taille égale des tampons en réception et en envoi, et de l'allocation en réception seule de la socket passive côté serveur, il est d'usage de fixer l'allocation des tampons applicatifs sur la méthode « getReceiveBufferSize() ». Deux tampons applicatifs peuvent être instanciés sur le client ou serveur, en lien avec les tampons de réception et d'envoi. Néanmoins, dans le cadre d'un échange à sens unique (transfert du client vers le serveur ou vice-et-versa), un seul tampon peut être utilisé en lecture ou en écriture selon l'application considérée sur le client / serveur comme illustré ci-dessous. Le code ci-après donne, à titre d'exemple, un code d'initialisation du tampon serveur.




```
class TCPBuffer {

        protected byte[] buffer;
        private final int size = 8192;

        /** The set method for the buffer. */
        void setStreamBuffer(int size) {
                if(size>0)
                        buffer = new byte[size];
                else
                        buffer = new byte[this.size];
                }
        }

                ----------------

        void setSocket() throws IOException {
                isA = new InetSocketAddress("localhost",8080);
                ss = new ServerSocket(isA.getPort());
                setStreamBuffer(ss.getReceiveBufferSize());
```

```
}
```

Il faut ensuite formater les données à insérer dans les tampons. Ceci soulève différentes difficultés liées à l'allocation, l'insertion / extraction des données, la détection des fins de flux et la réinitialisation du buffer. En effet, il faut veillez à recopier proprement les données des objets Java vers le tampon sans surcoût mémoire. Sur les objets « String », ceci peut facilement se faire sur parcours de la méthode « charAt() ». L'opération d'écriture « write » sur le flux « OutputStream » n'efface pas le tampon à l'issue de l'opération, ce qui peut créer des conflits en cas d'écritures successives. Enfin, la méthode « read » parcourra par défaut l'ensemble du tampon envoyé. Il faut donc s'assurer de la lecture de la partie utile du tampon à réception. Le code ci-dessous donne, à titre d'exemple, des possibles implémentations pour les méthodes de lecture et d'écriture. Ce code est basé sur une initialisation à zéro ASCII (i.e. caractère NULL) du tampon. Dans la perspective d'une communication en mode connecté, il est plus adéquat de conserver la capture et fermeture des objets « InputStream, OutputStream » depuis les codes client et serveur. Dans ce contexte, les méthodes de lecture / écriture pourront partir des arguments « InputStream, OutputStream ».

```
/** The (simple) text write method. */
void writeMessage(OutputStream out, String msOut) throws IOException {
	if((out!=null)&(msOut!=null)) {
		fillChar(msOut);
		out.write(buffer);
		out.flush();
		clearBuffer();
		}
	}

private void fillChar(String msOut) {
	if(msOut!=null)
		if(msOut.length() < buffer.length)
			for(int i=0;i<msOut.length();i++)
				buffer[i] = (byte)msOut.charAt(i);
	}

void clearBuffer() {
	for(int i=0;i<buffer.length;i++)
		buffer[i] = 0;
	}

/** The (simple) text read method. */
String readMessage(InputStream in) throws IOException {
	if(in != null) {
		in.read(buffer); count = count();
		if(count>0)
			return new String(buffer,0,count);
		}
	return null;
	}
private int count;

protected int count() {
	for(int i=0;i<buffer.length;i++)
		if(buffer[i] == 0)
			return i;
	return buffer.length;
	}
```

De manière à capitaliser le code d'échange de données, une possibilité est d'externaliser les opérations de lecture écriture au sein d'une classe « TCPMessage » dont hériteront les classes

« TCPClientBuilder, TCPServerBuilder ». Cette classe pourra regrouper les fonctionnalités d'écriture et de lecture TCP des données texte pour le client et le serveur.

Q5. Reprenez le code de la question Q4 ou vous externaliserez votre de code d'écriture et de lecture au sein de la classe «TCPMessage » dont hériteront les classes « TCPClientBuilder, TCPServerBuilder ». Veillez à initialiser votre buffer applicatif au sein des classes de construction « TCPClientBuilder, TCPServerBuilder », au regard des paramètres des sockets actives et passives. Remettez en œuvre votre communication développée en Q4 pour valider votre nouvelle implémentation. En particulier, on veillera au bon fonctionnement dans le cadre d'un échanges en multiples lectures / écritures.

On propose à ce stade d'aller plus en avant sur la maîtrise de la communication TCP au travers des paramètres de temporisation des sockets. En effet, ces paramètres peuvent être fixés à l'aide de l'instruction « setSoTimeout() » sur les sockets actives et passives. Ils garantissent un retour d'erreur côté client et serveur en cas de problème d'échange de données ou de non connexion. De ce fait, ils permettent une libération des ports et d'engager des protocoles de gestion d'erreur. Ils sont donc essentiels pour une bonne communication TCP.

L'usage est de configurer ces paramètres de telle manière que la temporisation en attente de connexion T_c soit plus longue que celle en attente des opérations de lecture / écriture T_{rw} i.e. $T_c >> T_{rw}$. La démarche de test de ces paramètres vise à simuler un échec de connexion (test de T_c) et/ou de communication (test de T_{rw}). Il est possible pour cela de lancer un serveur seul en attente de connexion, ou de mettre en place un échange en double lecture entre le client et le serveur (pas d'envoi de données mais uniquement des attentes en réception coté client et serveur). Un test groupé (multiple attente de connexion entrante, 1 connexion entrante, échange en double lecture) peut être également mis en place via l'instanciation d'un « Thread » côté serveur prenant en charge l'échange de données suite à une connexion entrante. Le code ci-dessous donne un squelette d'implémentation serveur.

```
class TCPServerTimeout implements Runnable {

	public void run() {
		try {
			ss = new ServerSocket(8080); ss.setSoTimeout(10000);
			while(true) {
				s = ss.accept();

				new Thread(new ServerTimeout(s)).start();
				}
			}
		catch(IOException e)
			{ System.out.println("IOException TCPServerTimeout"); }
		}

	private ServerSocket ss;
	private Socket s;

	}

class ServerTimeout implements Runnable {

	ServerTimeout(Socket s) {this.s = s;}

	public void run() {
		try {
			s.setSoTimeout(1000);
			/** read operation, to do … */
```

```
                    s.close();
                    }
            catch(IOException e)
                    { System.out.println("IOException ServerTimeout"); }
            }

}
```

Q6. Exploitez la méthode « setSoTimeout() » de manière à fixer les paramètres de temporisation de votre application. Vous pourrez pour cela mettre en œuvre deux classes « TCPClientTimeout, TCPServerTimeout » dérivées de « TCPClientBuilder, TCPServerBuilder ». Les paramètres devront être définis aussi bien au niveau des classes « TCPClientBuilder, TCPServerBuilder » que des classes « TCPClientTimeout, TCPServerTimeout » pour les sockets actives et passives. Ensuite, en vous basant sur vos méthodes de lecture développées précédemment, tracez l'évolution des paramètres au niveau client et serveur suite aux opérations « connect » et « accept ». Une fois l'ensemble de vos paramètres définis, testez la robustesse de votre application en cas d'échec de connexion et/ou de communication. Vous pourrez pour cela implémenter une absence de connexion et/ou un échange en double lecture en traitement séquentiel, parallèle pour le serveur (le serveur traite la connexion et l'échange, ou délègue l'échange), en test séparé ou groupé.

Sur la base d'une application sécurisée sur les paramètres de temporisation, il devient envisageable de gérer des échanges de données « en boucle ». Il s'agit d'échanges en plusieurs passes ayants recours à de multiples appels de la primitive « read ». Ceci peut aisément se mettre en œuvre à partir d'une structure « do / while ». Le paramètre de retour de la primitive « read » devra être analysé dans la boucle. Une valeur à -1 signifie la fin du transfert, une valeur >0 signifie un transfert en court ou « read » retourne le nombre d'octets lu à l'issue de l'appel de la primitive.

D'un autre côté, il est également nécessaire coté client de générer des flux texte de tailles significatives (e.g. plusieurs centaines de Mio ou qqs Gio). Une solution simple de mise en place est la génération synthétique / aléatoire des données texte. Cette solution présente les avantages d'un faible coût mémoire et d'écriture de code. En Java, cela peut facilement se mettre en place à l'aide d'un objet « Random ». Le code ci-dessous donne un exemple d'implémentation, à insérer dans la classe « TCPMessage ». La variable « loop » sera à paramétrer en fonction du volume désiré et de la taille des tampons applicatifs et sockets.

```
void loopWriteMessage(OutputStream out, int loop) throws IOException {
        for(int i=0;i<loop;i++) {
                fillAtRandom(buffer);
                out.write(buffer);
                out.flush();
                }
        }

private void fillAtRandom(byte[] buffer) {
        for(int i=0; i<buffer.length; i++)
                buffer[i] = (byte)r.nextInt(256);
        }
private Random r = new Random();
```

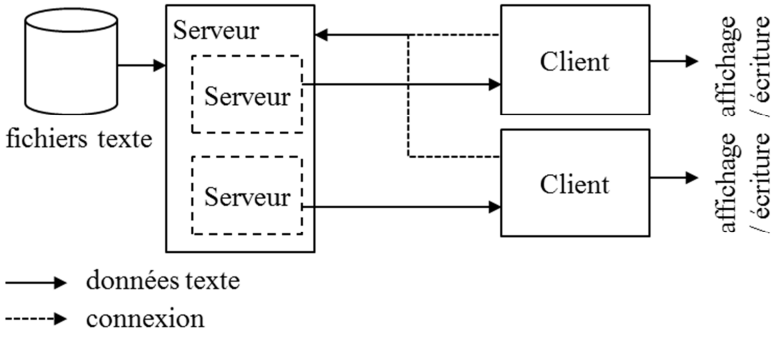
Q7. Mettez en place un échange TCP en large flux de données texte au travers de deux classes « TCPClientLMessage, TCPServerLMessage » (« LMessage » pour « Large message ») dérivées de « TCPClientBuilder, TCPServerBuilder ». Vous veillerez à la

paramétrisation des sockets sur les valeurs de tampon et de temporisation au sein des différentes classes « TCPClientBuilder, TCPServerBuilder, TCPBuffer », tel qu'illustré en Q5, Q6. Dans un premier temps, vous vous fixerez comme objectif l'échange d'un premier jeu significatif de données (e.g. 16, 32, 64 ou 128 kio). Côté serveur, vous pourrez tracer dans vos boucles les retours de la méthode « read » en affichant les numéros de boucle, la valeur retournée par la primitive « read » (i.e. count) et le volume total de données collectées sur l'échange (comme par exemple, collecté sur une variable d'accumulation de type « long »). Une fois votre code stabilisé, vous pourrez étendre votre échange à de gros volumes de données texte (e.g 32, 64, 128, 256, 512 Mio ou plus).

2.4.Cas d'usage – transfert TCP concurrent de fichiers texte

A ce stade du TP, les aspects premières mises en œuvre, généralisation de code, lecture et paramétrage de socket, échange de données texte en large flux, paramétrisation des sockets et tampon ont été illustrés. On propose ici d'aller plus en avant sur la communication TCP via la mise en œuvre d'un cas d'usage – transfert TCP concurrent de fichiers texte. Il sera assumé ici une maîtrise des fondamentaux de la communication TCP.

Il existe sur le Web des bases de données au format texte de taille importante. A titre d'exemple, Wikipedia met à disposition différentes bases d'articles et de méta-données[1] pouvant atteindre plusieurs centaines de Gio chacune en format compressé (soit des Tio de données). On se propose sur ce cas d'usage de mettre en place une architecture client-serveur pour l'accès à de telles bases, telle que détaillée ci-dessous. Le serveur permettra l'accès aux ressources texte en mode concurrent, chaque connexion entrante donnera lieu à un traitement parallèle depuis le serveur ou un « Thread » dédié prendra en charge l'échange avec le client. Les clients accèderont aux données texte dans un but d'écriture ou d'affichage des données. Les développements du client et du serveur seront abordés dans la suite du TP.



Une des difficultés liées au traitement des gros volumes de données et la parallélisation des flux et contrôle des allocations. En effet de par les volumes de données considérés, il est important de mutualiser la lecture fichier du transfert TCP côté serveur, et de faire de même côté client pour l'écriture suite à la réception TCP. De même, dans un objectif d'affichage du flux texte côté client, il faut être attentif à l'allocation des objets texte au risque de saturer le système en mémoire. Sur les aspects fichier, les classes de flux « FileInputStream, FileOutputStream » permettent de contrôler les volumes de chargement mémoire grâce aux méthodes « read(byte[]), write(byte[]) ». Sur les aspects affichage, la classe « StringBuffer » offre différentes fonctionnalités pour l'insertion de primitives « char » via la méthode « insert() » ainsi que la remise à zéro du tampon via la fonction « delete() ».

[1] https://meta.wikimedia.org/wiki/Data_dump_torrents

Q8. Mettez en place tout d'abord un client « TCPClientFile » pour l'accès au serveur de fichiers, dérivé de « TCPClientBuilder ». Sur la base de votre code développé en Q7, ce client devra permettre tout d'abord la connexion avec le serveur, la lecture des données et le contrôle du volume téléchargé. Vous pourrez, pour tester votre client « TCPClientFile », utiliser l'application « tcp.jar » mise à disposition pour le TP ou échanger avec la machine de l'enseignant. Un serveur et une base texte de dimension 0,24 Gio y sont proposés par défaut. Dans un second temps, étendez les fonctionnalités de votre classe « TCPClientFile » pour intégrer l'affichage et/ou l'écriture des données. Afin de capitaliser votre code et de ne pas surcharger l'écriture de votre client, vous pourrez créer une classe « TCPFile » pour intégrer vos différentes fonctions, dont héritera votre classe « TCPClientBuilder ». Cette classe pourra regrouper vos différentes fonctions « print(), write() » pour la lecture, l'affichage et/ou l'écriture des fichiers texte obtenus en transfert TCP.

Q9. Reprenez la classe « TCPFile » développée en Q9 pour intégrer une fonction de transfert TCP de fichier « transfer ». Veillez, au sein de votre fonction, à paralléliser la lecture des données disque au transfert TCP pour ne pas saturer la mémoire de votre système. Vous pourrez ensuite mettre en œuvre votre fonction depuis une classe « TCPServerFile » dérivée de « TCPClientBuilder ». Pour cela, vous pourrez utiliser la base « db-small » de 0,24 Gio mise à disposition sur le lien du cours pour échanger avec votre serveur. Etendez ensuite dans un second temps votre serveur « TCPServerFile » afin d'intégrer la parallélisation des échanges. Vous pourrez pour cela créer une classe « ServerFile » prenant en charge l'échange client suite à une opération « accept » sur votre Thread d'écoute « TCPServerFile ».

3. Annexes

Pour réaliser ce TP il sera nécessaire de travailler à partir d'une machine de l'école et sous machine virtuelle pour l'accès aux 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 différentes applications 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 »).

Pour le développement réseau, il sera nécessaire de la configurer la machine virtuelle en mode « NAT[2] ». Dans ce mode, la machine est installée sans pontage sur la carte réseau (l'adresse IP de la carte réseau est unique pour la machine physique et virtuelle). Ce mode est compatible avec la politique d'attribution dynamique des adresses IP sur le réseau de l'Université, ne permettant pas d'attribution pour une machine virtuelle sans adresse MAC.

[2] Network Address Translation

Si les étudiants le souhaitent, ils peuvent également développer sur leur propre machine. Pour ce faire, il sera probablement nécessaire de paramétrer le pare-feu réseau du système d'exploitation pour ouvrir les accès. De même, de par les sécurités des routeurs de l'Université, la communication entre une machine de l'école et une machine extérieure sera bloquée. Les étudiants pourront, néanmoins, échanger pair à pair entre machines extérieures à partir du réseau Wifi de l'Université ou d'un réseau en 4G.