# COMMUNICATION TCP
# FONDAMENTAUX ET PERFORMANCES

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

## 1. Introduction

### 1.1.Modalités d'évaluation

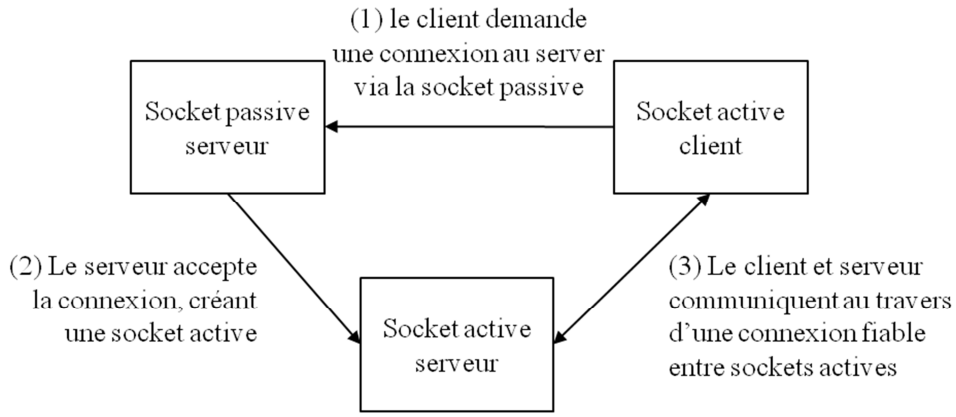
Ce TP doit être réalisé dans les créneaux impartis modulo le temps de préparation et de rédaction sur l'ensemble des séances (un volume recommandé de 1h par séance de 2h). Ce TP se fera de préférence par binôme, les monômes et trinômes sont autorisés et ce choix sera pris en compte dans les critères de notation. Le compte-rendu sera à déposer sous la plate-forme Celene univ-tours.fr, garantissant un dépôt confidentiel (pas de visibilité globale des dépôts depuis un login étudiant).

Le format des dépôts est le suivant : format de fichier pdf, nom de fichier « nom1-nom2.pdf » pour un binôme, 5 pages maximum recommandées pour le corps du texte, fournir en annexes le code, format des figures en résolution 220 dpi max. Il est avant tout attendu dans le compte-rendu une analyse des mécanismes de communication réseaux avec étude bibliographique à l'appui, illustrant les codes « clés », les résultats d'expérimentation.

### 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 le choix de 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 la gestion effective des communications TCP client-serveur. Le schéma suivant rappelle ces principes.




Dans le cadre du langage Java, différents composants sont mis à disposition dans 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 en 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 » et « Socket » sont gérées dans les constructeurs respectifs. Les opérations relatives à la gestion des sockets actives et la communication TCP 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.
- Pour une meilleure lisibilité du diagramme de communication TCP, l'opération « bind » a été désolidarisée de la construction de la socket passive du serveur (une

autre alternative est l'utilisation du constructeur « ServerSocket(int port) » concaténant l'opération « bind »).

- Les paramètres de socket sont définis par défaut sur l'adresse locale « localhost » et le port 8080, 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.

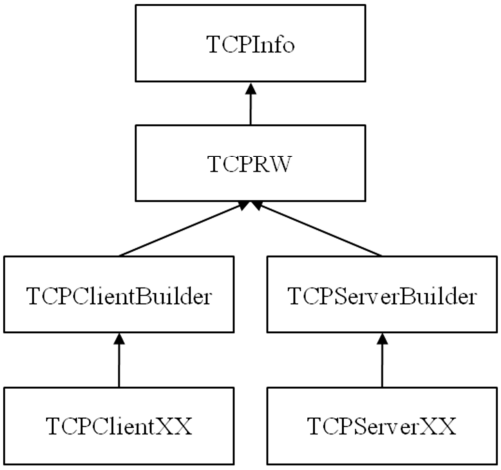
```
import java.net.*;
import java.io.*;
import java.util.*;

/** The TCP server. */
class TCPServer implements Runnable {
	private ServerSocket ss;
	private InetSocketAddress isA;
	TCPServer() {
		ss = null ;
		isA = new InetSocketAddress("localhost",8080);
		}
	public void run( ) {
		try	{
			ss = new ServerSocket();
			ss.bind(isA);
			s = ss.accept();
			s.close();
			}
		catch(IOException e)
			{ System.out.println("IOException TCPServer "); }
		}
	private Socket s;
	}

/** The TCP client */
class TCPClient implements Runnable {
	private Socket s;
	private InetSocketAddress isA;
	TCPClient() {
		s = new Socket();
		isA = new InetSocketAddress("localhost",8080);
		}
	public void run( ) {
		try	{
			s.connect(isA);
			System.out.println("Hello World, client connected");
			s.close();
			}
		catch(IOException e)
			{ System.out.println("IOException TCPClient"); }
		}
	}

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

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

## 2. Fondamentaux de la communication TCP

### 2.1.Première mise en œuvre

**Q1. Hello World:** 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 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 (vous devez pour cela échanger vos adresses et ports de communication).

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 de par différentes fonctionnalités qui seront développées au long du TP.




Une première démarche consiste à séparer le code d'initialisation des sockets (constructeurs et variables du code présenté 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.

```
abstract class TCPClientBuilder {

	Socket s;
	InetSocketAddress isA;

	TCPClientBuilder() {
		s = null;
		isA = new InetSocketAddress("localhost",8080);
		}

	void setSocket() throws IOException
		{ s = new Socket(); /* we can include more setting, later … */}

	}

class TCPClientHello extends TCPClientBuilder implements Runnable {

	public void run() {
		try {
				setSocket();
				s.connect(isA);
				System.out.println("Hello, client connected");
```

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

## 2.2.Paramètres et échange de données

**Q2. Lecture des paramètres:** 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 » et « 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 du client 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 ». Vous pourrez exploiter les fonctions de cette classe pour extraire et afficher les paramètres via une relation d'héritage mise en œuvre au sein des classes « TCPClientBuilder, TCPServerBuilder ». On pourra mettra ensuite mettre en œuvre le traçage des paramètres au travers de deux classes « TCPClientInfo, TCPServerInfo » dérivées de « TCPClientBuilder, TCPServerBuilder ».

```
class TCPInfo  {

	class SocketInfo {
		String lA,rA,tC;
		int lP,rP,sbS,rbS,tO,soLinger;
		boolean bounded, closed,isIPV6,noDelay;
		SocketInfo() {
			lA=rA=tC=null;
			lP=rP=sbS=rbS=tO=soLinger=-1;
			bounded=closed=isIPV6=noDelay=false;
			}
		}

	/** To print the 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.closed = ss.isClosed();
		ssI.bounded = ss.isBound();
		ssI.tO = ss.getSoTimeout();

		ssI.rbS = ss.getReceiveBufferSize();

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

	/** To print the 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.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"
```

```
                                                    +"closed: "+sI.closed+"\n"
                                                    +"timeout: "+sI.tO+"\tso linger: "+sI.soLinger+"\n"
                                                    +"buffer \tsend:"+sI.sbS+"\treceive:"+sI.rbS+"\n"
                                                    );
            }

    }
```

**Q3. Echange de données:** On se propose dans une étape suivante de mettre en œuvre un échange de données entre un client et un serveur, au travers de deux classes « TCPClientMsg, TCPServerMsg » dérivées de « TCPClientBuilder, TCPServerBuilder ». On se limitera ici, en première instance, à un échange de données ayant recours à un unique appel de la primitive de lecture « read ». Pour mettre en œuvre un tel échange il est nécessaire de travailler à partir des flux d'échange de données des sockets, récupérables à partir des méthodes « getInputStream(), getOutputStream() ». Les objets retournés de type « InputStream, OutputStream » permettent d'accéder aux méthodes « read, write » pour le transfert de données via les sockets. Ces méthodes travaillent à partir de flot d'octets aisément formatables en chaine 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'entré et de sortie doivent être fermés à l'aide de l'instruction « close ».

```
String msOut = "Aujourd'hui, TP ASR Java." ;
OutputStream out = s.getOutputStream();
out.write(msOut.getBytes());
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();
```

De manière à mettre en place ces opérations de lecture écriture, on se propose de développer une classe « TCPRW » dont hériteront les classes « TCPClientBuilder, TCPServerBuilder ». Cette classe regroupera les fonctionnalités d'écriture et de lecture TCP pour le client et le serveur. En particulier, elle sera en charge de l'allocation du tampon d'application (i.e. « byte[] buffer ») à utiliser avant toute opération « read ». Vous veillerez à ce que ce tampon soit initialisé au moins à la même taille que le tampon d'entrée de la socket TCP, tel que vous l'avez relevé dans les paramètres de socket. En effet, les opérations de lecture / écriture sont en mode bloquant. Une mauvaise adéquation entre les tailles des tampons peut donc résulter dans une augmentation significative des opérations de changements de contexte opérées par l'ordonnanceur de la machine virtuelle Java. Au sein de cette classe « TCPRW », vous pourrez définir des méthodes de lecture et écriture pour l'échange de chaine de caractère.

Vous veillerez à contrôler les ouvertures et fermetures de flux à partir de la méthode « run » au sein des classes « TCPClientMsg, TCPServerMsg », de façon à ne pas restreindre les échanges en « half duplex ». Les variables de flux « OutputStream, InputStream » pourront enrichir les classes « TCPClientBuilder, TCPServerBuilder » et être instanciées à partir des méthodes « run ». Utilisez ensuite l'ensemble de vos fonctions pour mettre en œuvre un échange de messages entre un client et un serveur et une et plusieurs passes, au sein des classes « TCPClientMsg, TCPServerMsg ».

**Q4. Gestion des paramètres « Timeout »:** On propose à ce stade d'aller plus en avant sur la maîtrise de paramètre de configuration des sockets, entre autre ceux de temporisation. En effet, le paramètre de temporisation d'une socket peut être fixé à l'aide de l'instruction « setSoTimeout() ». Exploitez cette méthode de manière à fixer les paramètres de temporisation au sein des classes « TCPClientBuilder, TCPServerBuilder ». Ensuite, en vous basant sur vos méthodes de lecture des paramètres socket développées en début de TP, tracer l'évolution de ces paramètres au niveau client et serveur suite aux opérations « connect » et « accept ». Vous pourrez pour cela mettre en œuvre deux classes « TCPClientTimeout, TCPServerTimeout » dérivées de « TCPClientBuilder, TCPServerBuilder ». Sachant que l'on souhaite configurer notre application de telle manière que la temporisation en attente de connexion, et celle en attente des opérations de lecture / écriture, soient différentes (plus courte pour la seconde, ce qui semble logique), comment procéderiez-vous ? Une fois l'ensemble de vos paramètres définis, tester la robustesse de votre application en cas d'échec de connexion. Vous pourrez pour cela simuler ces échecs en implémentant une absence de connexion (pas d'opérations « accept, connect » coté client ou serveur) ou un échange en double lecture (pas d'envoi de données mais uniquement des attentes de réception coté client et serveur).

**Q5. Echange de données en boucle:** Sur la base d'une application sécurisée en ce qui concerne les paramètres de temporisation, il devient envisageable de gérer des échanges de données « en boucle ». On entend ici par « boucle » un échange de données en plusieurs séquences, ayant recours à de multiples appels de la primitive de lecture « read » coté serveur. Pour ce faire, il est nécessaire de redéfinir la méthode de lecture des données définie précédemment au sein de la classe « TCPRW » pour supporter la gestion en boucle de la primitive de lecture « read ». Ceci peut aisément se mettre en œuvre à partir d'une instruction « while » et du paramètre de retour de la primitive « read » à -1 en cas de fin de transfert de données, ou à « count » en cas de transfert en cours.

Mettez en œuvre un tel échange au travers de deux classes « TCPClientLMsg, TCPServerLMsg » (« LMsg » pour « Large message ») dérivées de « TCPClientBuilder, TCPServerBuilder ». Au travers de ces classes, vous veillerez à la paramétrisation des sockets sur les valeurs de temporisation, à la fois sur la mise en œuvre des primitives « connect », « accept », « read / write ». Afin d'effectuer ce type d'échange, il vous sera nécessaire coté client de charger au sein de votre application des messages de taille significative (supérieur à minima à la taille du tampon en réception). Pour ce faire, vous pourrez créer une fonction de duplication de chaine de caractères (e.g. « Java TCP »)) afin de générer des messages de taille 32, 64, 128 ou 256 kio. Un exemple de code est donné ci-dessous. De la même manière, il faudra regrouper les différentes séquences coté serveur à l'issue de chaque retour de la primitive « read ». Vous pourrez vous aider d'un objet de type « StringBuffer » et de sa méthode « «append ». Veillez également à tracer la gestion des boucles de la méthode « read », par exemple en affichant sur chaque boucle une variable incrémentée et la valeur retournée par la primitive « read » (i.e. count).

```
/** The size is in kbytes */
String duplicateMessage(int size) {
        msT = new StringBuffer();
        time = (size*1024)/8;
        for(int i=0;i<time;i++)
                msT.append(ms);
        return msT.toString();
        }
private final String ms = "Java TCP";
private StringBuffer msT; private int time;
```

Dans la pratique, la machine virtuelle Java restreint l'allocation des données membres de classe (e.g. de 8 à 16 Mb pour des tableaux de données ou chaines de caractère). De manière à tester des transferts sur de plus gros volumes (e.g. 512 Mio à plusieurs Gio), il est nécessaire de gérer l'envoi et la réception en plusieurs passes entre le client et le serveur. Reprenez les classes « TCPClientLMsg, TCPServerLMsg » afin d'intégrer, côté client, un envoi en plusieurs passes d'un message de 512 ou 1024 kio (une boucle « for » sur des messages de 256 ou 512 kio). Vous pourrez, côté serveur, redéfinir la méthode de réception en boucle pour tracer le volume de données reçues, sans encapsulation / restitution des données sous forme de chaine de caractère (ou autres) à réception.

## 3. Communication TCP et performances

### 3.1.Tampons et communication TCP

Sur la base d'une application échangeant en boucle, on se propose d'étudier l'impact de la paramétrisation des tailles des tampons sur les échanges TCP. Considérant une application client-serveur fonctionnant en envoi / réception, les tampons existent à trois niveaux (1) le tampon d'émission de la socket du client (2) le tampon en réception de la socket du serveur (3) le tampon d'application du serveur (i.e. utilisé lors de l'appel de la primitive « read »).

**Q6. Tampons en réception et application – serveur:** Dans un premier temps, on propose de s'intéresser aux tampons en réception (2) et application (3) coté serveur. L'idée est de mettre en place une application client-serveur, paramétrable concernant ces tailles des tampons, au travers de deux classes « TCPClientBuffer, TCPServerBuffer » dérivées de « TCPClientBuilder, TCPServerBuilder ». Au sein de ces classes, les constructeurs pourront être redéfinis de façon à prendre en argument les tailles de tampons niveau socket (émission ou réception) et application (pour le serveur seulement). Les tailles des tampons socket peuvent être paramétrées à l'aide des instructions « setSendBufferSize(int), setReceiveBufferSize(int) ». Ces méthodes sont sujettes à des exceptions de type « SocketException » à prendre en compte. Un autre point important est la compréhension des primitives « accept, connect » qui provoquent l'initialisation des sockets actives concernant les allocations des tampons et les paramètres de temporisation (comme vu précédemment). La paramétrisation des tailles de tampons au niveau des sockets ne pourra donc se faire préalablement à l'appel des primitives « accept, connect » ou à l'issue de l'appel de la primitive « close ». Concernant le tampon d'application du buffer, vous pourrez implémenter une nouvelle méthode au sein de la classe « TCPRW » pour paramétrer la variable « byte[] buffer » utilisée lors de la gestion en boucle de la primitive de lecture « read ».

Mettez en œuvre vos deux classes depuis un programme principal. Vous pourrez dans un premier temps vérifier que la paramétrisation des tailles de tampons est bien prise en compte à l'aide de vos méthodes d'affichage de la classe « TCPInfo ». Vous devrez prendre en compte que, dans certains cas, les instruction « setSendBufferSize(int), setReceiveBufferSize(int) » sont avant tout des recommandations, laisser à libre interprétation des SE. Ensuite, mettez en œuvre un échange en passe multiple en vous basant sur trois configurations différentes pour les tailles de tampon telles que listées ci-dessous. Dans ces configurations, les tailles des tampons en émission et réception sont définies à l'identique de façon à ne pas impacter les échanges système / application depuis la connexion TCP. Que pouvez-vous en conclure ?

| | Tampon d'envoi client | Tampon de réception serveur | Tampon d'application serveur |
|---|---|---|---|
| **Cas 1** | 1024 octets | 1024 octets | 1024 octets |
| **Cas 2** | 64 octets | 64 octets | 1024 octets |
| **Cas 3** | 1024 octets | 1024 octets | 64 octets |

**Q7. Tampons en émission et réception - client, serveur:** Dans un deuxième temps, on propose de s'intéresser aux tampons en émission (1) et réception (2) coté client et serveur. Ces tampons étant utilisés directement au niveau de la communication TCP, il n'existe aucun moyen au niveau de l'application pour superviser leur état (e.g. retour d'interruption de fonction « read »). On ne peut donc estimer directement l'impact de leur paramétrisation sur les échanges TCP. Une façon de procéder consiste à mettre en œuvre un échange de valeurs d'horodatage (i.e. « timestamps »). Compte tenu d'un échange entre un client et un serveur, l'utilisation d'une même horloge en envoi et réception permet une estimation des temps de transmission au niveau de la connexion TCP (au décalage d'horloge « i.e. skew » près que l'on considéra comme négligeable ici).

L'obtention des valeurs d'horodatage peut aisément se faire via l'appel à la méthode système « System.nanoTime() » avec une précision au niveau de la nanoseconde. Le problème est ensuite l'échange des valeurs obtenues via la connexion TCP. Java propose pour cela des classes d'encapsulation des flots « InputStream, OutputStream » telles que « DataOutputStream, DataInputStream ». Ces classes d'encapsulation prennent en charge pour le développeur le formatage automatique des données (i.e. octets vers primitives et/ou objets) et la gestion du flux d'échange. Ce sont des objets complexes reposant sur des tampons internes et des mécanismes de contrôle de flux pour un formatage cohérent des données au niveau de l'application. L'utilisation de telles classes soulève deux problèmes (1) un surcoût de traitement non négligeable pouvant impacter l'estimation du temps de communication (2) des difficultés dans l'échange de données de grande dimension de part les mécanismes de formatage. De façon à lever ces problèmes une solution plus bas niveau consiste à implémenter des mécanismes légers de formatage des données, indépendamment des classes « DataOutputStream, DataInputStream ». Le code ci-dessous donne deux implémentations de fonctions pour les conversions « long » vers « byte[] » et réciproquement.

```
private byte[] toBytes(long data) {
        lbuf = new byte[8];
        for(int i=0;i<8;i++)
                lbuf[i] = (byte)((data >> (7-i)*8) & 0xff);
                return lbuf;
        }
private byte[] lbuf;

private long getLong(byte[] by) {
        value = 0;
        for (int i = 0; i < 8; i++)
                { value = (value << 8) + (by[i] & 0xff); }
        return value;
        }
private long value;
```

Implémentez deux classes « TCPClientBufferTiming, TCPServerBufferTiming » pour l'échange de données horodatées dérivées de « TCPClientBuilder, TCPServerBuilder », puis mettez-les en œuvre depuis votre programme principal. A l'instar des classes « TCPClientBuffer, TCPServerBuffer », ces deux classes devront supporter le paramétrage des tailles de tampons niveau socket et application. Vous pourrez vérifier que la

paramétrisation des tailles de tampons est bien prise en compte à l'aide de vos méthodes d'affichage de la classe « TCPInfo ». Redéveloppez ensuite des méthodes de lecture / écriture au sein de la classe « TCPRW » supportant l'échange des valeurs d'horodatage. Vous pourrez alors à réception estimer les temps de transmission TCP par comparaison des valeurs en envoi et en réception. Mettez en œuvre un échange en passe multiple en vous basant sur trois configurations différentes pour les tailles de tampon telles que listées ci-dessous. Dans ces configurations, les tailles des tampons en réception et application coté serveur sont définies à l'identique de façon à ne pas impacter les échanges entre couche système et applicative. Sur la base des estimations de temps de communication, que pouvez-vous en conclure ?

| | **Tampon d'envoi client** | **Tampon de réception serveur** | **Tampon d'application serveur** |
|---|---|---|---|
| **Cas 1** | 1024 octets | 1024 octets | 1024 octets |
| **Cas 2** | 2048 octets | 64 octets | 64 octets |
| **Cas 3** | 64 octets | 2048 octets | 2048 octets |

**Q8. Communication en mode « window scaling »:** Finalement, sur la base des conclusions précédentes, on se propose dans une dernière étape d'étudier l'influence du surdimensionnement (> 65 ko, e.g. 96 ou 128 ko) des tampons en émission (1) et réception (2), sur la communication TCP. En effet, les informations de paramétrisation des tampons en émission et réception sont échangés durant l'étape de connexion client-serveur via les opérations « connect » et « accept », permettant l'augmentation du débit d'échange en mode « window scaling ». En particulier, le cas du dimensionnement du tampon en réception est crucial, et doit être mis en place par paramétrisation de la socket passive, héritée ainsi par les sockets actives (i.e. la taille doit être spécifiée avant toute opération « accept »). Il devient en effet impossible, à l'issue de la phase de connexion, de re-provoquer l'échange des informations de paramétrisation, d'où l'impossibilité d'initier le mode « window scaling » une fois l'opération « accept » passée sans surdimensionnement du tampon.

Implémentez deux classes « TCPClientBufferTimingHuge, TCPServerBufferTimingHuge » pour l'échange de données horodatées dérivées de « TCPClientBuilder, TCPServerBuilder », puis mettez-les en œuvre depuis votre programme principal. A l'instar des classes « TCPClientBuffer, TCPServerBuffer », ces deux classes devront supporter le paramétrage des tailles de tampons niveau socket et éventuellement application. Vous pourrez vérifier que la paramétrisation des tailles de tampons est bien prise en compte à l'aide de vos méthodes d'affichage de la classe « TCPInfo ». Redéveloppez ensuite une méthode d'écriture au sein de la classe « TCPRW » supportant l'échange des valeurs d'horodatage pour de grand flux de données. Pour ce faire, il faudra redéfinir la fonction d'écriture coté client afin de gérer des envois par paquet (e.g. 512 Mo par paquet de 32 ko ou 64 ko). Vous pourrez alors à réception estimer les temps de transmission TCP par comparaison des valeurs en envoi et en réception, puis analyser les retours de la fonction « read » pour vérifier les valeurs maximum de données stockées au sein du tampon de réception. Il faudra, à ce niveau, vérifier que ces valeurs dépassent bien la limite de 65 ko, car certaines implémentations natives des sockets (e.g. Winsock) peuvent désactiver le mode « window scaling » par défaut sur certaines versions de système (Vista et antérieur).

## 3.2.Communication TCP et classe de trafic

Q9. Ajouter une question sur les classes de trafic

## 4. Annexes

**Virtualisation**. Pour réaliser ce TP il sera nécessaire de travailler à partir d'une machine virtuelle (cf documentation « virtualisation des salles machine Polytech »). Rendez-vous sur le répertoire « VM_Production » puis lancez la machine virtuelle « TP Java » directement à partir de la version production (il n'est pas nécessaire de recopier cette 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 Windows XP incluant différentes applications Java (SDK 1.6, Eclipse, etc.). Cette machine virtuelle est configurée en mode « bridget » donnant toute la latitude sur les accès réseaux. 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 exécutable 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 »).