TP architecture des systèmes répartis

COMMUNICATION UDP
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).

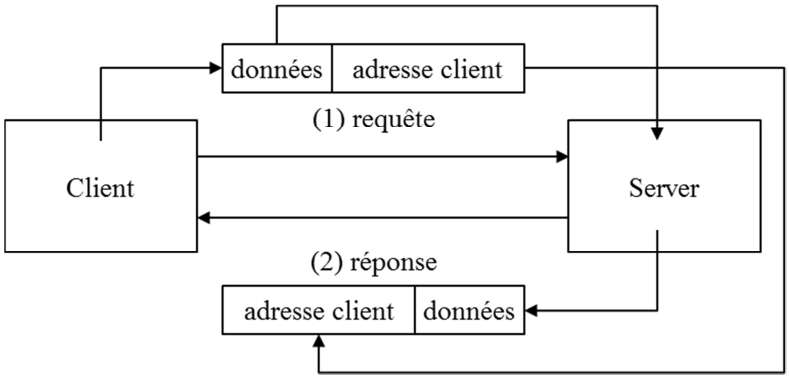
Le format des dépôts est le suivant : format de fichier pdf, nom de fichier « nom1-nom2.pdf » pour un binôme, 5 pages maximum recommandées pour le corps du texte, fournir en annexes le code, format des figures en résolution 220 dpi max. Il est avant tout attendu dans le compte-rendu une analyse des mécanismes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

1.2.Communication UDP Java

Ce TP s'intéresse à la communication UDP 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 UDP, 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 UDP passe par la manipulation d'objets/structures socket et des primitives de communication associées (« bind », « close », « connect », « disconnect », « send », « receive »). Contrairement à la communication TCP, la communication UDP se base sur l'utilisation d'un seul type de socket utilisé indifféremment du côté serveur et du côté client.

UDP diffère de TCP dans le sens ou la communication s'effectue en mode sans connexion. Cela résulte dans une architecture et communication simplifiée pour la mise en œuvre d'un échange client-serveur. Le client et le serveur échangent des datagrammes dans une phase requête-réponse, les informations liées au client sont extraites du datagramme de la requête par le serveur afin d'acheminer la réponse. 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 UDP, 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 et des paquets, dont les principales sont résumées dans le tableau ci-dessous.

DatagramSocket	Socket utilisée indifféremment côté client et serveur
InetSocketAddress	adresse de socket i.e. adresse IP & port
DatagramPacket	objet « packet » utilisé pour l'échange de données

En complément, le package java.lang fournit les mécanismes nécessaires pour la parallélisation des applications. En effet, les primitives de communication UDP 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 UDP 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 les créations des objets adresse, attraits au serveur, sont gérées dans les constructeurs respectifs du client et du serveur. Les opérations relatives à la gestion des sockets et datagrammes sont reportées au sein des méthodes « run ».
- Il est à noter que les opérations standards de gestion de socket (i.e. « bind », « send », « receive » « close ») 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 UDP, les opérations de construction des sockets (concaténant les opérations « bind ») ont été désolidarisées des constructeurs du serveur et du client. Les paramètres de socket sont définis par défaut sur l'adresse locale « localhost » et le port 8080 pour le serveur, mais peuvent

être changés selon les configurations locales (e.g. port de 0 à 65535, adresse IPV4 e.g. 127.0.0.1). Il est à noter que la construction de la socket client ne précise aucun attachement, résultant dans une affectation automatique d'adresse « localhost » et de port.

- Les opérations de création de datagramme « DatagramPacket » suivent une implémentation différente dans le client et dans le serveur, afin de mettre en œuvre l'échange requête - réponse illustré précédemment. Les informations liées au client sont extraites du datagramme de la requête par le serveur, et utilisées pour instancier le datagramme de réponse.
- Ce premier exemple de mise en œuvre illustre un échange « blanc », les datagrammes encapsulent des données vides (tableau de 1kio) pour l'échange requête - réponse. On rappelle que la norme IPV4 limite la taille des datagrammes à 65 kio, mais que dans la pratique la majorité des implémentations la restreigne à 8 kio.
- On rappelle également la syntaxe pour le lancement des « Threads » depuis le programme principal.

```
class UDPClient implements Runnable {

	private InetSocketAddress isA;
	private DatagramSocket s;
	private DatagramPacket req, rep;

	private final int size = 2048;

	UDPClient() {
		isA = new InetSocketAddress("localhost",8080);
		s = null; req = rep = null;
		}

	public void run( ) {
		try {
			s = new DatagramSocket();

			req = new DatagramPacket(new byte[size],0,size,isA.getAddress(),isA.getPort());
			s.send(req);
			System.out.println("request sent");

			rep = new DatagramPacket(new byte[size],size);
			s.receive(rep);
			System.out.println("reply received");

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

	}

class UDPServer implements Runnable {

	private InetSocketAddress isA;
	private DatagramSocket s;
	private DatagramPacket req, rep;

	private final int size = 2048;

	UDPServer() {
		isA = new InetSocketAddress("localhost",8080);
		s = null; req = rep = null;
		}
```

```
	public void run( ) {
		try {
			s = new DatagramSocket(isA.getPort());

			req = new DatagramPacket(new byte[size],size);
			s.receive(req);
			System.out.println("request received");

			rep = new DatagramPacket(new byte[size],0,size,req.getSocketAddress());
			s.send(rep);
			System.out.println("reply sent");

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

	}
```

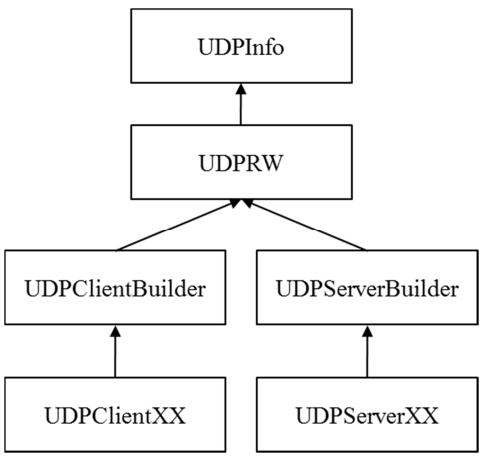
```
new Thread(new UDPClient()).start();
new Thread(new UDPServer()).start();
```

2. Fondamentaux de la communication UDP

2.1.Première mise en œuvre

Q1. Hello World: Reprenez le code présenté en introduction pour mettre en œuvre votre première communication UDP. 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 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 « UDPClientXX, UDPServerXX » 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 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 « UDPClientBuilder, UDPServerBuilder » et de leurs classes dérivées « UDPClientHello, UDPServerHello ». Le code ci-dessous donne, à titre d'exemple, une implémentation possible pour le client. Les méthodes « setConnection() », définies au sein des classes « UDPClientBuilder, UDPServerBuilder », spécifient les paramétrages des sockets. Elles pourront être appelées en première instance dans les codes de mise en œuvre (i.e. première instruction des méthodes « run ») au sein des classes d'application « UDPClientXX, UDPServerXX ».

```
class UDPClientBuilder extends UDPRW {

	InetSocketAddress isA;
	DatagramSocket s;
	DatagramPacket req, rep;

	UDPClientBuilder()
		{ isA = null;  s = null; req = rep = null; }

	protected void setConnection() throws IOException {
		s = new DatagramSocket();
		isA = new InetSocketAddress("localhost",8080);
		/* we can include more setting, later … */
		}

	}

class UDPClientHello extends UDPClientBuilder implements Runnable {

	private final int size = 1024;

	public void run() {
		try {
			setConnection();

			req = new DatagramPacket(new byte[size],0,size,isA.getAddress(),isA.getPort());
			s.send(req);
			System.out.println("request sent");

			rep = new DatagramPacket(new byte[size],size);
			s.receive(rep);
			System.out.println("reply received");

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

	}
```

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 UDP (i.e. « bind », « close », « 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 », de fermeture « i.e. closed » et connexion « i.e. connect », les tailles des tampons en envoi et en réception, les valeurs de temporisation, etc. Le tableau ci-dessous donne les différentes méthodes de la classe « DatagramSocket » permettant la récupération de ces paramètres.

Adresse et port locaux	getLocalAddress(),getLocalPort()
Adresse et port distant	getInetAddress(),getPort()
Paramètre de fermeture, de limitation et connexion	isClosed(),isBound(); isClosed()
Taille des tampons en envoi et réception	getSendBufferSize(),getReceiveBufferSize()
Valeur de temporisation	getSoTimeout(),
Mode de diffusion, réutilisation, classe de trafic	getBroadcast(),getReuseAddress(),getTrafficClass()

Il n'existe pas de mécanisme, ou méthode interne à la classe « DatagramSocket », pour la récupération et le traçage en une traite de ces différents paramètres. Il est donc nécessaire de mettre en œuvre un composant (i.e. classe extérieure) assurant cette fonction. De par le nombre important de paramètres considérés, la déclaration d'une structure (i.e. classe interne sans « réelle » méthode) peut constituer une solution pour l'allégement de l'écriture des méthodes de lecture et d'affichage. Les lectures des paramètres de taille de tampon, de temporisation et de mode de diffusion « i.e. broadcast, TrafficClass » sont sujettes à des exceptions de type « SocketException, IOException » à prendre en compte. Un autre point important est la compréhension de la primitive « bind », qui provoque l'allocation des tampons locaux. L'accès aux tailles de tampon ne pourra donc se faire préalablement à la construction de la socket, 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 « UDPInfo ». 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 « UDPClientBuilder, UDPServerBuilder ». On pourra mettra ensuite mettre en œuvre le traçage des paramètres au travers de deux classes « UDPClientInfo, UDPServerInfo » dérivées de « UDPClientBuilder, UDPServerBuilder ».

```
class UDPInfo  {

        class SocketInfo {
                String lA,rA,tC;
                int lP,rP,sbS,rbS,tO;
                boolean isIPV6,bounded,closed,connected,rU,bC;
                SocketInfo() { clear(); }
                void clear() {
                        lA=rA=tC=null;
                        lP=rP=sbS=rbS=tO=-1;
                        isIPV6=bounded=closed=connected=bC=rU=false;
                        }
                }

        protected void socketInfo(String event, DatagramSocket s) throws SocketException {
                if((s!=null)&(event!=null)) {
                        sI.clear();

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

                        sI.lA = getAdressName(s.getLocalAddress());
                        sI.lP = s.getLocalPort();
                        sI.rA = getAdressName(s.getInetAddress());
                        sI.rP = s.getPort();

                        sI.closed = s.isClosed();
```

```
                        sI.bounded = s.isBound();
                        sI.connected = s.isConnected();

                        if(!sI.closed) {
                                sI.tO = s.getSoTimeout();
                                sI.bC = s.getBroadcast();
                                sI.rU = s.getReuseAddress();
                                sI.tC = Integer.toHexString(s.getTrafficClass());
                                sI.sbS = s.getSendBufferSize();
                                sI.rbS = s.getReceiveBufferSize();
                                }

                        print(event);
                        }
                }
private SocketInfo sI = new SocketInfo();

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

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

void print(String event) {
                if(sI.closed)
                        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"
                                                +"connected: "+sI.connected+"\n"
                                                );
                else
                        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"
                                                +"connected: "+sI.connected+"\n"
                                                +"timeout: "+sI.tO+"\t broadcast: "+sI.bC+"\t reuse: "+sI.rU+"\t traffic: "+sI.tC+"\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 « non-blanc » entre un client et un serveur, au travers de deux classes « UDPClientMsg, UDPServerMsg » dérivées de « UDPClientBuilder, UDPServerBuilder ». Une application classique consiste à échanger des valeurs d'horodatage entre le client et le serveur. Compte tenu d'un échange entre un client et un serveur local, l'utilisation d'une même horloge en envoi et réception permet une estimation des temps de transmission au niveau de la connexion UDP au décalage d'horloge près que l'on considéra comme négligeable. 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 UDP. 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 par 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, int size) {
        lbuf = new byte[size];
        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;
```

De manière à mettre en place ces opérations de lecture écriture, on se propose de développer une classe « UDPRW » dont hériteront les classes « UDPClientBuilder, UDPServerBuilder ». Cette classe regroupera les fonctionnalités d'écriture et de lecture UDP pour le client et pour le serveur. En particulier, elle sera en charge de la préparation des paquets en envoi avec insertion des valeurs d'horodatage et renseignement des adresses de destinataires. Elle s'assura également de la réception des paquets avec extraction des valeurs d'horodatage. L'estimation du temps de transmission pourra se faire au sein de la méthode de lecture, ou depuis le code applicatif du client. Vous veillerez à encapsuler vos échanges (valeurs d'horodatage) au sein de messages types d'une communication UDP, respectant les pratiques de tailles (1 kio à 8 kio).

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 « UDPClientBuilder, UDPServerBuilder ». On se propose ensuite de tester la mise en œuvre de ces paramètres au travers de deux classes « UDPClientTimeout, UDPServerTimeout » dérivées de « UDPClientBuilder, UDPServerBuilder ». Vous pourrez, tout d'abord, vérifier la prise en compte de ces paramètres au sein de vos sockets en vous basant sur vos méthodes de lecture des paramètres socket développées en début de TP. Une fois l'ensemble vos paramètres définis et vérifiés, tester la robustesse de votre application en cas d'échec de connexion. Vous pourrez pour cela simuler ces échecs en implémentant un échange en double lecture (pas d'envoi de messages mais uniquement des attentes de réception coté client et serveur).

Q5. Echange en mode connecté: UDP diffère de TCP dans le sens ou la communication s'effectue en mode sans connexion. Néanmoins, parmi les primitives UDP on compte des primitives « connect, disconnect ». La primitive « connect » ne sert pas à établir une connexion réseau à proprement parler, mais opère d'avantage comme un opérateur de filtrage au sein d'une communication UDP. Une socket locale, connectée à une socket distante via une communication UDP, appliquera par défaut un filtrage de tout message en réception autre que provenant de la socket distante. Dans la pratique, la primitive de connexion « connect » est surtout appliquée côté client, pour s'assurer d'un filtrage en amont (au niveau du tampon UDP) de toutes réponses autres que celle(s) attendue(s) par le serveur sollicité. Cette fonctionnalité évite au programmeur de prendre en charge la gestion du filtrage et une surcharge du code client.

On se propose d'illustrer la mise en œuvre de cette primitive « connect ». Vous pourrez, dans un premier temps, paramétrer la socket client en mode connectée au sein de la classe « UDPClientBuilder ». Afin d'illustrer la mise en œuvre de la primitive de connexion, une pratique consiste à simuler l'envoi de « réponses parasites » de serveurs tierces à un client. Pour ce faire, il est nécessaire de pouvoir contrôler l'affectation des paramètres d'adresse et de port côté client et côté serveur. Vous pourrez tout d'abord redéfinir les méthodes « setConnexion() » et données membres des classes « UDPClientBuilder, UDPServerBuilder » pour contrôler ces paramètres. Il vous sera nécessaire de basculer la construction des « DatagramSocket » en « new DatagramSocket(int port) ». Mettez en œuvre ensuite un échange contrôlé entre un client et un serveur, spécifiant les paramètres de connexion de part et d'autre. Vous pourrez alors dans une dernière étape simuler l'envoi de « réponses parasites », en instanciant plusieurs serveurs communiquant avec un client et en spécifiant la connexion du client sur l'un des serveurs. Vous pourrez vérifier si le filtrage est opérationnel en mettant en œuvre, par exemple, un échange de valeur d'horodatage pour marquer de manière unique les messages échangés.

3. Communication UDP et performances

3.1.Tampons et communication UDP

Au même titre que la communication TCP, la communication UDP permet de spécifier les tailles de tampons. On se propose d'étudier l'impact de la paramétrisation de ces tailles de tampons sur les échanges UDP. Considérant une application client-serveur fonctionnant en envoi / réception, les tampons existent à deux niveaux (a) les tampons de réception et (b) les tampons d'émission des sockets client / serveur.

Q5. Paramétrage du tampon serveur en réception: Dans un premier temps, on propose de s'intéresser au tampon en réception (a) coté serveur. Le paramétrage de ce tampon a pour vocation de fiabiliser la réception des messages côté serveur, afin d'assurer la réception de l'ensemble des requêtes. Ce tampon est généralement paramétré comme défini en (1) avec MAX_RECV_QUEUE le nombre de message maximum (ou nombre de client total) que peut recevoir le serveur et MAX_DATAGRAM_BYTES la taille maximum observée pour les requêtes.

$$\text{BUFFER\_SIZE} = \text{MAX\_RECV\_QUEUE} \times \text{MAX\_DATAGRAM\_BYTES} \quad (1)$$

Au cas où le tampon ne serait pas spécifié comme défini en (1), le risque est une saturation du tampon de réception du serveur et la possible perte de requêtes client. On se propose ici de simuler cet impact. Il faut pour cela mettre en place une application client-serveur, au travers de deux classes « UDPClientBuffer, UDPServerBuffer » dérivées de « UDPClientBuilder, UDPServerBuilder ». Au sein de ces classes, les constructeurs pourront être redéfinis de façon à prendre en argument les paramètres MAX_RECV_QUEUE et MAX_DATAGRAM_BYTES. La taille du tampon de réception peut être paramétré à l'aide de l'instruction « setReceiveBufferSize(int) ». Cette méthode est sujette à des exceptions de type « SocketException » à prendre en compte.

De façon à saturer le tampon en réception du serveur, il sera nécessaire « d'attaquer » le serveur avec MAX_RECV_QUEUE clients et d'assurer une latence minimum de traitement côté serveur pour provoquer le remplissage du tampon. La mise en œuvre des clients peut se faire aisément à l'aide d'une boucle instanciant MAX_RECV_QUEUE « Thread » client en requête vers le serveur. En ce qui concerne la latence, une solution simple consiste à mettre en place une méthode « busywait » au sein du serveur. Un exemple est donné ci-dessous, il faut considérer des valeurs types pour BW de 1 à 5×10^6.

```
private void busywait(int BW) throws IOException {
        i = 0;
        while(i<BW)
                { i++; }
        }
private int i;
```

Sur la base des classes « UDPClientBuilder, UDPServerBuilder », vous pourrez ensuite mettre en œuvre les deux scénarios avec et sans saturation du tampon, considérant les paramètres ci-dessous.

	Thread	**MAX_RECV_QUEUE**	**MAX_DATAGRAM_BYTES**	**BW**
sans saturation	200	200	2048	1 à 5×10^6
avec saturation	200	40	2048	

Illustrer tout d'abord le cas sans saturation, afin de vous assurer du traitement de l'ensemble des requêtes par le serveur en cas de respect de (1). Vous pourrez, pour cela, tracer le nombre de requête traité côté serveur. Vous pourrez dans un deuxième temps illustrer le scénario avec saturation. En particulier, étudier l'impact du paramètre BW sur le nombre de requête traité et expliquez alors le phénomène observé. Dans un deuxième temps, analyser à l'aide de l'échange des valeurs d'horodatage les impacts en temps de réception (à même paramètre BW) dans les cas avec saturation et sans saturation. Que pouvez-vous en conclure sur le paramétrage du tampon de réception?

Q6. Paramétrage du tampon serveur en envoi:

Q7. Gestion des messages par troncature:

3.2.Communication UDP et concurrence

Q8. Mise en œuvre d'un serveur en accès concurrent

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 »).