FONDAMENTAUX DE LA COMMUNICATION UDP 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 UDP, architecture client / serveur, paramètres de socket, temporisation, cas d'usage – application de Chat, cas d'usage – serveur temps

1. Introduction

1.1. Modalités d'évaluation et de mise en œuvre

Ce TP doit être réalisé dans les créneaux impartis modulo le temps de lecture et de préparation. 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.

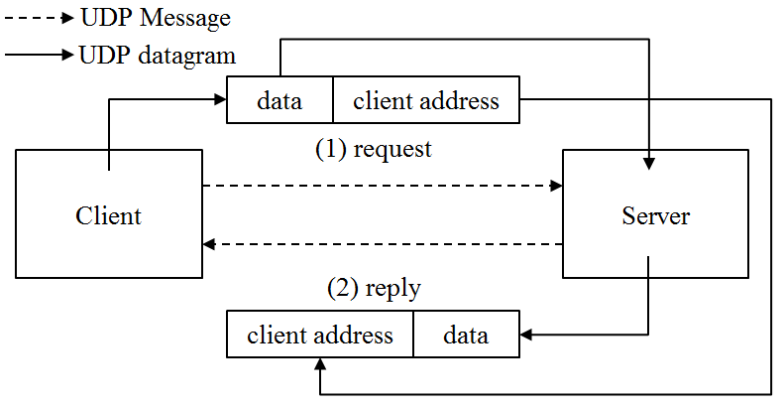
Ce TP se présente, dans sa rédaction, comme un développement « from scratch ». A partir de différents exemples de code clé, le TP permet la réécriture complète de la plateforme logicielle à mettre en œuvre. L'annexe du TP rappelle les modalités sur les plateformes et réseaux de développement nécessaires à sa réalisation du TP.

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 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). On rappelle que la liste des ports ouverts « Listening, *.* » peut être obtenue via l'utilitaire « netstat -a » en ligne de commande. 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 2kio) 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.

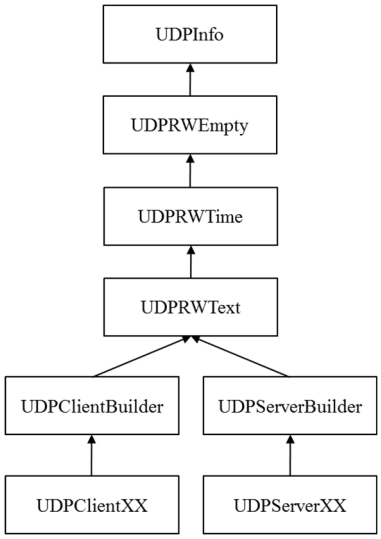
---------- Fichier 1 ----------

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

class UDPClient implements Runnable {

	InetSocketAddress isA;			// the remote address
	DatagramSocket s;			// the socket object
	DatagramPacket req, rep;		// to prepare the request and reply messages
	private final int size = 2048;		// the default size for the buffer array

	/** The builder. */
	UDPClient() {
		isA = new InetSocketAddress("localhost",8080);
		s = null; req = rep = null;
		}

	/** The main run method for threading. */
	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;		// the address
	private DatagramSocket s;		// the socket object
```

```
	private DatagramPacket req, rep;	// to prepare the request and reply messages
	private final int size = 2048;		// the default size for the buffer array

	/** The builder. */
	UDPServer() {
		isA = new InetSocketAddress("localhost",8080);
		s = null; req = rep = null;
		}

	/** The main run method for threading. */
	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"); }
		}

	}
```

---------- Fichier 2 ----------

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

public class Test {
	public static void main (String[] args) {
		new Thread(new UDPServer()).start();
		new Thread(new UDPClient()).start();
		}
	}
```



De manière à capitaliser le code dans le TP, on se propose de développer une application telle que décrite ci-dessus. 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 ici sera 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.

La 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 précédemment). 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 {

	InetSocketAddress isA;
	DatagramSocket s;
	DatagramPacket req, rep;
	final int size = 2048;

	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 {


	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. Fondamentaux de la communication UDP

2.1.Première mise en œuvre

Q1. 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 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 votre code mis en œuvre de la question Q1 au travers de quatre classes « UDPClientBuilder, UDPClientHello » et « UDPServerBuilder, UDPServerHello ». Veillez à bien tester la validité de votre code en re-évaluant la communication client-serveur mise en œuvre en Q1, sur la base de la nouvelle organisation de votre code. Les classes « UDPClientBuilder, UDPServerBuilder » pourront être utilisées dans la suite du TP pour une mise en œuvre plus rapide de vos applications de communication UDP.

2.2.Paramètres de socket

Une fois une 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(); isConnected()
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 ».

```
/** The class to get the socket info */
class UDPInfo  {

	/** The internal class/structure for socket parametters. */
	class SocketInfo {
		String lA,rA,tC;			// the local, remote address, traffic class
		int lP,rP,sbS,rbS,tO;			// the local, remote port, sending/receiving buffer size, timeout value
		boolean isIPV6,bounded,closed,connected,rU,bC;
		// is ipv6, soccket bounded, closed, connected, reuse and broadcasr parametters
		SocketInfo() { clear(); }
		void clear() {
			lA=rA=tC=null;
			lP=rP=sbS=rbS=tO=-1;
			isIPV6=bounded=closed=connected=bC=rU=false;
			}
		}

	private SocketInfo sI;		/** Reference on the internal class SocketInfo. */

	/** The builder. */
	UDPInfo() { sI = new SocketInfo(); }

	/**
	* The main class to call,
	* event is a string to characterize the call location in the code,
	* s is the socket object.
	*/
	protected void socketInfo(String event, DatagramSocket s) throws SocketException {
		if((event!=null)&(s!=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);
			}
		}
```

```
	/** Some get methods. */

	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;
		}

	/** The internal print method. */
	private 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"
					);
		}

	}
```

Cette classe propose une fonction centrale de mise en œuvre « socketInfo ». Elle permet l'affichage des paramètres d'une socket 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 « UDPInfo » peut-être incorporée via une relation d'héritage avec les builder « UDPClientBuilder, UDPServerBuilder ». Compte tenu que la communication UDP opère en mode non connectée, le traçage des paramètres peut s'illustrer à l'issue de la construction et de la fermeture des sockets, côté client et serveur. Le code ci-dessous donne un squelette de mise en œuvre pour le client.

```
class UDPClientBuilder extends UDPInfo { …. }

class UDPClientInfo extends UDPClientBuilder implements Runnable {

	public void run() {
		try	{
			setConnection(); socketInfo("client sets the connection",s);
			s.close(); socketInfo("client closes the connection",s);
			}
		catch(IOException e) { System.out.println("IOException UDPClientInfo"); }
		}
	}
```

Q3. Exploitez les fonctionnalités de la classe « UDPInfo », au travers de la méthode socketInfo(event, s), pour extraire et afficher les paramètres des sockets. Veillez à spécifier une relation d'héritage au sein des classes « UDPClientBuilder, UDPServerBuilder ». On pourra mettra ensuite mettre en œuvre le traçage des paramètres des sockets au travers de deux classes « UDPClientInfo, UDPServerInfo » dérivées de « UDPClientBuilder, UDPServerBuilder ». Ces classes pourront mettre en œuvre une simple création / fermeture socket côté client et serveur. Observez l'évolution des paramètres des sockets à différents instants de la communication, comme à la connexion et à la fermeture.

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() ». Ce paramètre est essentiel pour une bonne communication UDP, il garantit un retour d'erreur du client (ou du serveur) en cas de problème de communication (i.e. absence de message). De ce fait, il permet une libération des ports de communication et d'engager au niveau de l'application un protocole de gestion d'erreur. L'usage de est de fixer des valeurs de temporisation T_c, T_s, pour le client et le serveur respectivement, tel que $T_c << T_s$.

Q4. Exploitez la méthode « setSoTimeout() » 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. 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 message mais uniquement des attentes de réception coté client et serveur). Le code ci-dessous donne un squelette de mise en œuvre pour « UDPClientTimeout ».

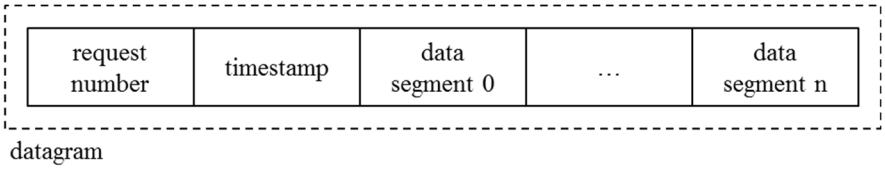
```
class UDPClientTimeout extends UDPClientBuilder implements Runnable {

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

			rep = new DatagramPacket(new byte[size],size); s.receive(rep);

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

2.3.Protocoles d'échange et cas d'usage

2.3.1. Introduction

On se propose ici d'aller plus en avant sur la communication UDP via la mise en œuvre de différents protocoles d'échange et cas d'usage. Les protocoles impliquent l'échange de différents messages entre un client et un serveur (requête, réponse, acquittement) et leur traitement. Les couches applicatives du client et du serveur doivent procéder au formatage / déformatage des données contenues dans les messages UDP, en fonction des applications /

cas d'usage visées. Ceci implique l'écriture de fonctions parfois complexes qui dépendent des structures des datagrammes UDP (identifiant, valeur d'horodatage, segment de données).

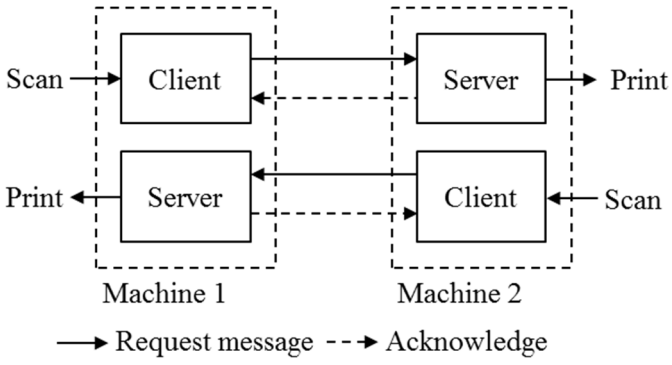


Dans ce TP, on propose d'encapsuler les fonctions de formatage / déformatage des messages au sein de classes « UDPRWxx » dont hériteront les classes « UDPClientBuilder, UDPServerBuilder ». Ces classes regrouperont les fonctionnalités d'écriture et de lecture des messages pour le client et pour le serveur, en lien avec une application donnée. Elles seront en charge de la préparation des paquets en envoi puis lecture à la réception. Il sera important au sein de ces classes de respecter les contraintes de taille pour la communication (1 kio à 8 kio par message). De façon à moduler le traitement de préparation des paquets, il faut séparer le code de création de celui d'écriture / lecture. La classe « UDPRWEmpty » ci-dessous donne des fonctions pour la création des paquets pour l'envoi et la réception.

```
class UDPRWEmpty {

	/** To prepare a sending packet at a given size e.g. 2048 kbytes. */
	protected DatagramPacket getSendingPacket(InetSocketAddress isAR, int size) throws IOException
		{ return new DatagramPacket(new byte[size],0,size,isAR.getAddress(),isAR.getPort()); }

	/** To prepare a receiving packet at a given size. */
	protected DatagramPacket getReceivingPacket(int size) throws IOException
		{ return new DatagramPacket(new byte[size],size); }

	}
```

2.3.2. Application de Chat

Une application type basée sur la communication UDP est le Chat. De par la nature asynchrone des échanges entre deux utilisateurs, une application de Chat peut être architecturée via deux modèles client - serveur. Il s'agit ici d'échanges unidirectionnels i.e. envoi d'un message de requête du client vers le serveur pour l'affichage.



Un besoin clé de cette application est le formatage / déformatage des données textuelles (i.e. chaine de caractère vers tableau d'octet et vice-et-versa). Ce formatage peut facilement être mis en œuvre en Java, les classes de l'API Java proposent toutes des fonctions de formatage objet vers tableau de d'octets « getBytes() ». Le déformatage nécessite lui un parcours du tableau d'octets pour détecter les éléments non nuls. Le code ci-dessous donne un exemple de fonctions de formatage / déformatage des données textuelles.

```
private byte[] sB;                /** The buffer array. */


/** To set the Msg to a parametter packet. */
protected void setMsg(DatagramPacket dP, String msg) throws IOException
        { toBytes(msg, dP.getData()); }

private byte[] toBytes(String msg, byte[] lbuf) {
        array =  msg.getBytes();
        if(array.length < lbuf.length)
                for(int i=0;i<array.length;i++)
                        lbuf[i] = array[i];
        return lbuf;
        }
private byte[] array;

/** To extract the txt message from a packet. */
protected String getMsg(DatagramPacket dP) {
        sB = dP.getData();
        for(int i=0;i<sB.length;i++) {
                if(sB[i] == 0)
                        { p = i; i = sB.length; }
                }
        return new String(dP.getData(),0,p);
        }
private int p;
```

Ces fonctions peuvent ensuite être mises en œuvre depuis des classes client et serveur. Le code ci-dessous donne un exemple d'appel depuis une classe client pour l'envoi de message.

```
req = getSendingPacket(isAR,2028);  setMsg(req,"hello guy");  s.send(req);
```

De façon à rendre l'application de Chat interactive, il sera nécessaire de réaliser la lecture clavier des messages à transférer. Ceci peut facilement se réaliser en Java grâce à la classe « Scanner » du package « java.util ». Le code ci-dessous donne un exemple.

```
Import java.util.*;
…
private Scanner sc;
….
sc = new Scanner(System.in);
String msg = sc.nextLine();
sc.close();
```

Q5. Mettez tout d'abord en place une classe « UDPRWText » dont hériteront les classes « UDPClientBuilder, UDPServerBuilder ». Cette classe regroupera les fonctionnalités d'écriture / lecture UDP des données textuelles pour le client et le serveur. Vous pourrez ensuite mettre en place un premier échange pour l'envoi et l'affichage d'un message texte en dur (chaine pré-codée) entre un client et un serveur. Pour cela, définissez deux classes « UDPClientMsg, UDPServerMsg » dérivées de « UDPClientBuilder, UDPServerBuilder ». Etendez ensuite ce code pour intégrer la lecture clavier des données textuelles.

Un problème clé de ce type d'application est la fermeture ou détection d'erreur. En effet, dans un modèle de simple échange client vers serveur il devient impossible pour le client de détecter si le serveur reste à l'écoute. Dans ce cas, la sortie de l'application sur le « Thread » client n'est plus garantie. Une approche pour résoudre ce problème est la gestion d'un retour d'acquittement par le serveur. De ce fait, en cas d'absence de réponse d'un serveur le « Thread » client pourra être interrompu via son paramétrage de temporisation T_c. De même,

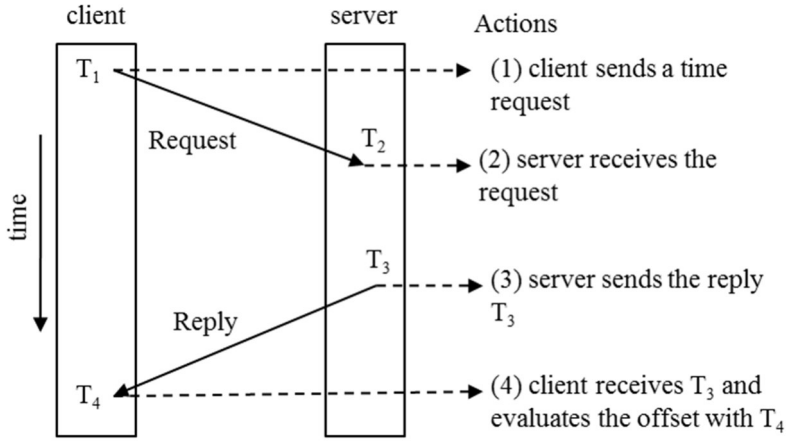
le paramètre de temporisation du serveur T_s peut être utilisé pour détecter le cas d'inactivité du client. T_c et T_s caractérisent alors les bornes max de communication UDP et d'interaction utilisateur respectivement, avec $T_c << T_s$.

Pour une gestion complète des erreurs, le déclenchement d'interruption côté serveur peut être utilisé pour provoquer la fermeture du client, et vice-et-versa. De même, la fermeture du client et du serveur peut être provoquée par commande de l'utilisateur depuis le scanner. Ces derniers aspects relèvent cependant d'avantage de problèmes de programmation système (passage de référence des « Thread ») et d'API que de programmation réseaux.

Q6. A partir de votre code développé en Q5, mettez en place ensuite une architecture complète en lançant en local un client et un serveur en échange avec une machine tierce (celle de l'enseignant, ou celle d'un autre groupe d'étudiant). Pour cela, définissez deux classes « UDPClientChat, UDPServerChat » dérivées de « UDPClientBuilder, UDPServerBuilder ». De façon à simuler un échange en continu dans votre application, vous pourrez implémenter une structure do / while dans votre code. Veillez finalement à sécuriser votre application par retour d'acquittement du serveur, pour garantir une sortie de votre « Thread » client. Testez alors le bon fonctionnement de votre application en cas de non lancement / fermeture d'application de Chat distante.

2.3.3. Serveur temps

Une autre application type de mise en œuvre du protocole UDP est le serveur temps (e.g. protocole NTP[1]). Comme illustré sur la figure ci-dessous, le principe en est de récupérer la valeur d'horloge d'un serveur pour évaluer le décalage d'horloge T_3–T_4 avec le client. Le protocole passe par l'envoi d'une requête du client puis réponse du serveur. Il permet la synchronisation temporelle de machine dans le cadre d'un traitement réparti.



Un problème lié au développement d'un serveur temps est la récupération puis comparaison des valeurs d'horloge. En java, la méthode « currentTimeMillis() » de classe « System » permet le retour de la valeur d'horloge de la machine avec une précision de l'ordre de la ms[2]. La fonction « currentTimeMillis() » retourne des primitives « long » pour l'encodage des valeurs d'horloge. Le code ci-dessous donne des fonctions d'empaquetage et désempaquetage de primitives « long » vers tableaux d'octets et datagrammes.

[1] Network Time Protocol
[2] Depuis minuit UCT 1er Janvier 1970.

```
private byte[] sB;              /** The buffer array. */
private long tstamp;            /** The timestamp. */

/** To get the local time. */
protected long getLocalTime()
        { return System.nanoTime(); }

/** To get the timestamp. */
protected long getTimeStamp()
        { return System.currentTimeMillis(); }

/** To get a sending packet with a timestamp. */
protected DatagramPacket getTimeSendingPacket(InetSocketAddress isA, int size) throws IOException {
        tstamp = getTimeStamp(); sB = toBytes(tstamp, new byte[size]);
        return new DatagramPacket(sB,0,sB.length,isA.getAddress(),isA.getPort());
        }

/** To set the timestamp to a parametter packet. */
protected void setTimeStamp(DatagramPacket dP)
        { tstamp = getTimeStamp(); sB = toBytes(tstamp, dP.getData()); }

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

/** To extract timestamp from a receiving packet. */
protected long getTimeStamp(DatagramPacket dP) { return getLong(dP.getData()); }

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;
```

Dans la pratique, la comparaison de valeurs d'horloge doit tenir compte d'un ajustement lié au temps de communication (T_2–T_1, T_4–T_3 sur le schéma). Le décalage entre le serveur et le client s'obtient donc en calculant $T_3 + k - T_4$, avec k une constante estimant le temps de communication UDP moyen / médian sur le réseau considéré. Une façon simple d'obtenir une approximation de k est de considérer T_3–$T_2 << T_4$–T_1. Dans ce cas, k s'obtient par calcul de la valeur T_4–T_1 / 2 ou T_4–T_1 est l'intervalle temps côté client entre l'envoi de la requête et la réception de la réponse par le serveur. En java, la méthode « nanoTime() » de classe « System » permet l'horodatage d'évènement une précision de l'ordre de la ns.

Q7. Mettez tout d'abord en place une classe « UDPRWTime » dont hériteront les classes « UDPClientBuilder, UDPServerBuilder ». Cette classe regroupera les fonctionnalités d'écriture et de lecture UDP des données temps pour le client et le serveur. Définissez deux classes « UDPClientTime, UDPServerTime » dérivées de « UDPClientBuilder, UDPServerBuilder » pour la mise en place de votre serveur temps. Vous pourrez vous synchroniser avec une machine tierce (de l'enseignant ou d'un autre groupe d'étudiants) sur une période de temps donnée (qqs minutes). Veillez à implémenter un mécanisme de trigger au sein de votre classe « UDPClientTime » à l'aide de la fonction « Thread.sleep() » pour contrôler la fréquence de synchronisation (e.g. toutes les 250 ms). Vous pourrez, en première instance, considérer la valeur d'ajustement comme nulle i.e. k=0.

Q8. Dans un deuxième temps on se propose d'évaluer plus précisément la valeur d'ajustement k. Définissez deux classes « UDPClientNTP, UDPServerNTP » dérivées de « UDPClientBuilder, UDPServerBuilder ». Dans la pratique, ces classes seront très proches des classes « UDPClientHello, UDPServerHello » développées en Q1. Au sein de ces classes, complétez la méthode « run » du client pour l'estimation de la valeur $k = T_4 – T_1 / 2$ à l'aide la fonction « nanoTime() ». Pour un calcul objectif, il sera nécessaire d'estimer k en plusieurs passes. Vous pourrez pour cela implémenter un mécanisme de trigger au sein de votre classe « UDPClientNTP » à l'aide de la fonction « Thread.sleep() » pour contrôler la fréquence de communication (e.g. toutes les 250 ms). Reportez finalement la valeur de k obtenu dans votre serveur temps Q7 pour une synchronisation effective des horloges.

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 la communication réseau, vous pourrez tout d'abord tester en local (le client et le serveur communiquent sur la même machine via la carte réseau). Dans un second temps, vous pourrez communiquer de votre machine virtuelle vers une machine native comme celle de l'enseignant. Il sera pour cela nécessaire de configurer votre machine virtuelle en mode « NAT[3] ». Dans le mode NAT, 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.

Une dernière alternative est de développer sur votre propre machine. Pour ce faire, il sera probablement nécessaire de paramétrer le pare-feu réseau de votre 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 / 5G.

[3] Network Address Translation