République Algérienne Démocratique Et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

UNIVERSITE ABDELHAMID IBN BADIS – MOSTAGANEM

FACULTE DES SCIENCES EXACTES ET DE L'INFORMATIQUE

DEPARTEMENT DE MATHEMATIQUES-INFORMATIQUE

Polycopié pédagogique

....

INTRODUCTION A LA PROGRAMMATION RESEAU

....

Polycopié de cours

Master 1 ReSys (Réseaux et Systèmes)

Filière : Informatique

Par

HOCINE Nadia

Maitre de conférences au département de mathématiques et informatique

Année : 2020 - 2021

AVANT-PROPOS

Ce polycopié de cours a été réalisé pour la matière intitulée « programmation réseaux » de l'unité méthodologie du Master Réseau et Systèmes (ReSys) de l'université d'Abdelhamid Ibn Badis, faculté des sciences exactes et informatique, département de mathématique et informatique. Il ne s'agit pas d'une matière qui concerne la mise en œuvre des réseaux et des protocoles. Cette matière se focalise uniquement sur la programmation des applications réparties en utilisant quelques interfaces de programmation existantes. Elle nécessite des pré-requis en réseaux et en programmation Java et C. Cette matière est organisée sous forme d'un cours et une séance de travaux pratiques par semaine durant un semestre.

TABLE DES MATIERES

AVANT-PROPOS **2**

TABLE DES MATIERES **3**

LISTE DE FIGURES **7**

ABBREVIATIONS **8**

INTRODUCTION **2**

PARTIE I: GÉNÉRALITÉS **3**

Chapitre 1: Réseaux TCP/IP et programmation des applications réparties **3**

1.1 Introduction 3

1.2 Réseaux TCP/IP 3

1.2.1 Couche accès réseau 3

1.2.2 Couche Internet 3

1.2.3 Couche transport 4

1.2.4 Couche application 4

1.3 Modes de communication en réseau 4

1.3.1 Communication par paquet (datagram) 4

1.3.2 Communication en flux (stream) 4

1.4 Explorer un réseau avec les commandes UNIX 5

1.4.1 Vérification des interfaces réseau avec dmesg, ifconfig et netstat 5

1.4.2 Vérification du masque de sous-réseau 6

1.4.3 Récupérer les noms des machines d'un réseau 6

1.4.4 Obtenir le manuel des services et fonctions réseaux 6

1.5 Programmation des applications réparties 6

1.5.1 Caractéristiques des applications réparties 6

1.5.2 L'architecture client/serveur 7

1.5.3 Communiquer deux applications en réseau 7

1.6 Conclusion 7

PARTIE II: PROGRAMMATION RÉSEAU EN JAVA **11**

Chapitre 2: Flux d'entrée-sortie, sérialisation et multithread en Java **11**

2.1 Introduction 11

2.2 Applications client/serveur en Java 11

2.3 Les flux d'entrée/ sortie de Java (java.io) 11

2.3.1 Les classes File, FileInputStream et FileOutputStream 11

2.3.2 DataInputStream et DataOutputStream 12

2.3.3 ObjectInputStream et ObjectOutputStream 13

2.4 Qu'est-ce que c'est la sérialisation? 14

2.4.1 Sérialisation en Java *15*
2.5 Les flux d'entrée/ sortie (java.nio) 15
2.6 Programmation multithread en Java 16
2.6.1 La classe java.lang.Thread *16*
2.6.2 Synchroniser les threads *17*
2.7 Conclusion 19
Chapitre 3: Sockets TCP/UDP en Java 24
3.1 Introduction 24
3.2 L'interface de programmation Socket 24
3.3 Sockets UDP en Java 24
3.3.1 Schéma fonctionnel de la communication client/serveur avec socket UDP en Java *25*
3.3.2 La classe InetAddress *25*
3.3.3 La classe DatagramPacket *26*
3.3.4 La classe DatagramSocket *26*
3.3.5 Méthodes d'émission/réception de paquets *26*
3.3.6 Autres méthodes *26*
3.3.7 Exemples de client/serveur avec Socket UDP en Java *27*
3.4 Socket TCP en Java 29
3.4.1 Schéma fonctionnel de la communication client/serveur en socket TCP *29*
3.4.2 La classe Socket *30*
3.4.3 Méthodes d'émission et de réception de données *30*
3.4.4 Autres méthodes *30*
3.4.5 La classe ServerSocket *31*
3.4.6 Exemples de client/serveur avec Socket TCP en Java *31*
3.5 La sérialisation des objets Sockets 32
3.5.1 Exemple d'échange d'objets sérialisé en Socket TCP *33*
3.6 Programmation réseau multithread 33
3.7 La diffusion 34
3.7.1 Diffusion intégrale (Broadcasting) *34*
3.7.2 Multidiffusion (Multicasting) *35*
3.8 Communication non bloquante: Java NIO 36
3.8.1 La classe Buffer *36*
3.8.2 Les canaux (Channel) *36*
3.8.3 Les sélecteurs *37*
3.8.4 Exemple de socket avec channel et selector *37*
3.9 Conclusion 39
Chapitre 4: Java RMI (Remote Method Invocation) 40

4.1 Introduction 40
4.2 Java RMI 40
4.3 Architecture et principe de fonctionnement de RMI 40
4.4 Développement d'une application RMI 41
4.5 Exemple d'une application client/serveur en RMI 41
4.5.1 Etape (1). Définir une interface Java et son implémentation côté serveur pour un objet distant 41
4.5.2 Etape (2). L'implémentation de l'objet distant: CalculatorImpl.java 42
4.5.3 Etape (3). Le serveur RMI: CalculatorServer.java 42
4.5.4 Etape (4) Implémentation du CalculatorClient.java 43
4.5.5 Etape (5) Compiler les fichiers et générer les amorces 43
4.5.6 Etape (6) Enregistrer l'objet dans l'annuaire 44
4.5.7 Etape (7) Lancer le serveur puis le client 44
4.6 Sécurité 44
4.7 Conclusion 44
PARTIE III: PROGRAMMATION RÉSEAU EN C 45
Chapitre 5: Sockets TCP/UDP en C 45
5.1 Introduction 45
5.2 Socket TCP 45
5.2.1 Schéma de la communication en mode connecté 45
5.2.2 Structure de l'adresse 46
5.2.3 La fonction socket() 46
5.2.4 La fonction bind() 46
5.2.5 Les fonctions listen() et accept() 47
5.2.6 La fonction connect() 47
5.2.7 Les fonctions read() et write () 47
5.2.8 Les fonctions send() et recv() 48
5.2.9 Autres fonctions 48
5.2.10 Exemples de client/serveur en Socket C 48
5.3 Socket UDP 52
5.3.1 Schéma de la communication en mode non connecté 52
5.3.2 La fonction socket() 53
5.3.3 Exemple d'un client/serveur UDP 53
5.4 Les entrée/sorties non-bloquantes en C 55
5.4.1 La fonction fcntl 55
5.4.2 La fonction select 56
5.4.3 La fonction poll 57
5.5 Conclusion 58

Chapitre 6: RPC (Remote Procedure Call)/ XDR **59**

6.1 Introduction 59

6.2 RPC 59

6.3 Architecture de RPC et principe de fonctionnement 59

6.3.1 Fonctionnement général d'un appel de fonction 60

6.3.2 Reconnaissance d'une procédure RPC 60

6.4 Techniques de développement d'un client/serveur RPC 60

6.4.1 La couche haute 60

6.4.2 La couche intermédiaire 61

6.4.3 La couche basse 61

6.5 Mise en œuvre des RPC par rpcgen 61

6.5.1 Règles d'écriture du fichier ".x" en RPCL 61

6.6 Exemple d’une application client/serveur en RPC 62

6.7 Conclusion 64

SERIES D’EXERCICES **65**

Série d’exercices I **65**

Série d’exercices II **67**

Série d’exercices III **70**

CONCLUSION **73**

REFERENCES **74**

ANNEXES **75**

Annexe A : Implémentation d’un mini serveur HTTP simplifié **75**

7.1 Le protocole HTTP 75

7.2 Requête HTTP 75

7.3 Réponse HTTP 75

7.4 Exemple d’impélemntation d’un serveur HTTP en utilisant les sockets 76

Annexe B : Socket et la programmation mobile en Java Android **79**

8.1 Création d’une activité 79

8.2 AsyncTasks 79

8.3 Exemple de serveur avec un client mobile 80

Annexe C : Correction de la série d’exercices I **82**

Annexe D : Correction de la série d’exercices II **88**

Annexe E : Correction de la série d’exercices III **97**

LISTE DE FIGURES

Figure 1: Principe de la communication en TCP.. 4

Figure 2: Les classes de gestion de flux en Java 12

Figure 3: Schéma fonctionnel de la communication entre un client et un serveur UDP en Java.......... 25

Figure 4: Schéma fonctionnel de la communication entre un client et un serveur TCP en Java........... 30

Figure 5: L'architecture d'une application client/serveur utilisant Java RMI..................................... 41

Figure 6: Processus de communication entre un client et un serveur RMI. 42

Figure 7: Récapitulatif des étapes de compilations et d'exécution de l'exemple de l'application client/serveur RMI implémentée. 43

Figure 8: Schéma fonctionnel de la communication client/serveur C en TCP.................................... 45

Figure 9: Schéma fonctionnel de la communication client/serveur C en UDP. 52

Figure 10: Architecture de RPC. 59

Figure 11: Fonctionnement général d'un appel de procédure en RPC.. 60

ABBREVIATIONS

API: Application Programming Interface

ARPANET: Advanced Research Projects Agency Network

ARP: Address Resolution Protocol

BSD: Berkeley Software Distribution

BOOTP: Bootstrap Protocol

CORBA: Common Object Request Broker Architecture

DARPA: Defense Advanced Research Projects Agency

DNS: Domain Name System

DHCP: Dynamic Host Configuration Protocol

FDDI: Fiber Distributed Data Interface

FTP: File Transfer Protocol

HTTP: Hypertext Transfer Protocol

HTML: HyperText Markup Language

IP: Internet Protocol

ICMP: Internet Control Message Protocol

JADE: Java Agent Development Framework

JRMP: Java Remote Method Invocation

JSP: JavaServer Pages

J2E : Java Platform, Enterprise Edition

JVM: Java Virtual Machine

JSON: JavaScript Object Notation

LAN: Local Area Network

MDA: Model Driven Architecture

NFS: Network File System

NTP: Network Time Protocol

OMG: Object Management Group

OSI: Open Systems Interconnection

PHP: Hypertext Preprocessor langage

RARP: Reverse Address Resolution Protocol

RMP: Remote Method Protocol

RPC: Remote Procedure Call

RMI: Remote Method Invocation

SDK: Software Development Kit

SMTP: Simple Mail Transfer Protocol

SOAP: Simple Object Access Protocol

TCP : Transmission Control Protocol

Telnet: Telecommunication network

UDP: User Datagram Protocol

WAN: Wide Area Network

W3C: World Wide Web Consortium

XDR: eXternal Data Representation

XML: Extensible Markup Language

INTRODUCTION

Un système distribué est un ensemble d'ordinateurs indépendants connectés qui se communiquent via un réseau. Cet ensemble apparaît du point de vue de l'utilisateur comme une unique entité. La mise en place des systèmes distribués permet de faire interagir des applications existantes réparties géographiquement et fonctionnellement. L'avantage est de diminuer la redondance du matériel, des informations et du traitement. Cela permet d'augmenter la performance en permettant un traitement en parallèle de tâches, de partager des ressources et de faciliter le développement et l'évolution du système [2].

On trouve généralement deux techniques utilisées pour communiquer deux applications en réseau [2]. La première se focalise sur l'utilisation du code mobile en envoyant le code source à la machine distante afin qu'elle l'exécute. C'est le cas par exemple des applets Java et des agents mobiles. Cette technique n'est pas toujours pratique et dépond du volume des données échangées et le niveau de sécurité requis.

La seconde technique se base sur l'utilisation des interfaces de programmation. En général, une interface de programmation ou API (Application Programming Interface) est un nom donné à l'ensemble de classes et méthodes qui sert de façade par laquelle une application offre des services à d'autres applications. Une API réseau est utilisée pour faciliter la communication entre les applications réparties. Ces APIs suivent généralement différents paradigmes de programmation ainsi que des modèles de systèmes distribués existants. Les interfaces de programmation réseau se basent sur l'appel des services directement de la couche transport (TCP ou UDP) ou bien sur des middlewares. Ces derniers permettent de masquer la communication avec la couche transport à travers une couche de haut niveau proposant différent services pré-implémentés.

L'objectif de ce cours est de présenter les interfaces de programmation réseau en langage Java et C. Il est structuré en trois parties:

- La première partie introduit des généralités sur les réseaux et la programmation des applications réparties. Elle inclut un seul chapitre:
  - **Chapitre 1**: décrit des principales constructions nécessaires à la communication entre les applications à travers le réseau. Ce chapitre donne également un aperçu sur les approches de programmation des applications réparties.
- La seconde partie de ce cours définit les principales interfaces de programmation en langage Java. Elle comprend trois chapitres:
  - **Chapitre 2**: permet de rappeler quelques bibliothèques et classes Java nécessaires pour implémenter une application répartie. Cela comprend par exemple l'utilisation des flux d'entrée/sortie, la notion de sérialisation ainsi que la programmation multithread.
  - **Chapitre 3**: introduit la programmation réseau en langage Java à l'aide de l'interface Socket.
  - **Chapitre 4**: présente une autre interface de programmation de plus haut niveau en Java appelée RMI (Remote Method Invocation) qui se base sur l'appel des méthodes d'accès aux objets distants.
- La troisième partie de ce cours décrit les interfaces de programmation en langage C. Elle est structurée en deux chapitres:
  - **Chapitre 5**: décrit la programmation des applications client/serveur en utilisant l'interface Socket BSD.
  - **Chapitre 6**: présente l'interface de programmation RPC (Remote procedure call) qui se base sur la programmation impérative via l'appel de procédures distantes.

En fin, des séries d'exercices sont proposées sur les principales classes et bibliothèques vues dans les chapitres précédents.

PARTIE I: GÉNÉRALITÉS

Chapitre 1: Réseaux TCP/IP et programmation des applications réparties

1.1 Introduction

Un réseau est un ensemble d'hôtes capables de communiquer en utilisant des services et des protocoles [5]. C'est le cas par exemple de la communication via le réseau Internet qui se focalise sur la famille de protocoles TCP/IP [3]. Ce chapitre introduit les modes de communication réseau via cette famille de protocoles ainsi que quelques bases de l'administration et l'exploration réseau via les commandes du système d'exploitation Unix. Il donne également un aperçu sur les modalités et les architectures de répartition des applications ainsi que les techniques de programmation utilisées.

1.2 Réseaux TCP/IP

DARPA (United States Defense Advanced Research Projects Agency) a proposée un réseau expérimental nommé ARPANET devenu opérationnel en 1975. En 1983, la série de protocoles TCP/IP fut adoptée comme standard. ARPANET se transformera alors pour s'adapter à cette série de protocoles et deviendra Internet. TCP/IP était également utilisé dans les réseaux locaux, tels que les réseaux UNIX [5].

Différent services et applications sont rendues possibles par TCP/IP. Un exemple très simple est NFS (Network File System). Il permet de rendre le réseau transparent en donnant à l'utilisateur la possibilité de voir l'hiérarchie des répertoires depuis d'autres hôtes. L'utilisateur peut alors employer n'importe quelle machine en toute transparence de la même façon qu'un système de fichier local [4].

Afin d'assurer la communication entre les machines ayant différents dispositifs et systèmes d'exploitation, TCP/IP est fondé sur une architecture multicouches. Le terme « couche » est utilisé pour faire référence aux données (appelées paquets) qui transitent sur le réseau et nécessitent de traverser successivement plusieurs niveaux de protocoles. A chaque couche un ensemble de protocoles est utilisé pour assurer un rôle dans la communication en rajoutant des informations (appelé en-tête) aux paquets qui seront par la suite transmis à la couche suivante [3].

Le modèle TCP/IP est inspiré du modèle standard OSI qui a été proposé par l'organisation internationale des standards (ISO, organisation internationale de normalisation). A l'inverse d'OSI qui utilise un modèle en sept couches, TCP/IP en utilise uniquement quatre avec des protocoles bien identifiés. Ci-après les couches de TCP/IP :

1.2.1 Couche accès réseau

Cette couche permet de masquer l'hétérogénéité des réseaux physiques en spécifiant la forme sous laquelle les données doivent être acheminées. Elle ajoute les spécifications de transmission de données sur un réseau physique tel que le réseau local accessible via par exemple Ethernet. Elle assure l'acheminement des données sur la liaison, formatage de données, conversion des signaux (analogique/numérique), etc.

1.2.2 Couche Internet

Cette couche est chargée de fournir le paquet de données et gérer l'adressage. Elle permet l'acheminement d'un paquet de données vers des machines distantes ainsi que la gestion de sa fragmentation et de son assemblage à la réception. Cette couche utilise la notion d'adressage IP. Parmi les protocoles de cette couche: IP, ARP, ICMP, RARP, etc.

Il existe en IPv4 et à l'origine cinq classes d'adresses utilisées :

- la classe A 0.0.0.0 - 127.255.255.255, soit 128 réseaux de 16M machines.
- la classe B 128.0.0.0 - 191.255.255.255, soit 16k réseaux de 64K machines.
- la classe C 192.0.0.0 - 223.255.255.255, soit 2M réseaux de 256 machines.
- la classe D 224.0.0.0 - 239.255.255.255, réservée à la multidiffusion.
- la classe E 240.0.0.0 - 255.255.255.255, pour la diffusion intégrale ou « broadcast ».

1.2.3 Couche transport

Cette couche assure l'acheminement des données ainsi que les mécanismes permettant de connaître l'état de la transmission. Elle permet à des applications de communiquer et êtres accessibles en réseau en les identifiants à travers un numéro appelé un « port ». Elle offre deux modes de communication selon le protocole utilisé TCP ou UDP.

1.2.4 Couche application

C'est la couche qui fournit un ensemble de services à l'utilisateur à travers son interface avec le système d'exploitation. Elle contient les applications permettant de communiquer en réseau grâce aux couches inférieures. Elle englobe par exemple les applications standards du réseau (Telnet, SMTP, FTP, …) mais aussi les différentes applications développées par l'utilisateur.

1.3 Modes de communication en réseau

Il existe deux moyens de communication avec lesquels se fait l'échange de paquets en réseau selon le modèle TCP/IP: la communication par paquet et la communication en flux [3].

1.3.1 Communication par paquet (datagram)

C'est une communication souple qui ne nécessite pas le pré-établissement d'une connexion entre l'émetteur et le récepteur pour les échanges simples. Elle est assurée par le protocole UDP ou User Datagram Protocol (RFC 768) de la couche transport du modèle TCP/IP. UDP découpe les messages en paquets appelés aussi « datagrammes ». Le respect de l'ordre des paquets et la fiabilité de la communication ne sont pas garanties.

Un paquet UDP se compose d'une entête avec un nombre fixe d'octets (8 octets) et une partie données qui contient un nombre variable d'octets (<= 65 535). Les applications bien connues qui se basent sur ce protocole ont des ports UDP réservés [0, 1023]. C'est le cas par exemple du serveur DNS (53), serveur BOOTP/DHCP (67) et le serveur NTP(123).

Pour programmer une application qui utilise UDP, les ports [1024, 49151] qui sont enregistrés peuvent être utilisés. Les autres ports [49152, 65535] sont dits dynamiques et/ou à usage privé.

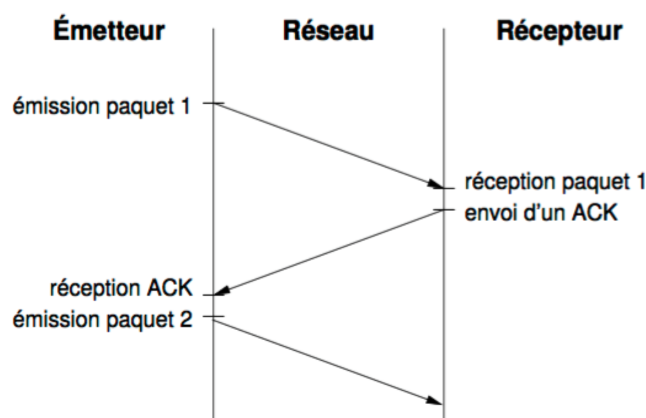


Figure 1: Principe de la communication en TCP.

1.3.2 Communication en flux (stream)

La communication en flux utilise le protocole TCP (Transmission Control Protocol, RFC 793 corrigée par RFC 1122 et 1323) qui requiert l'établissement d'une connexion entre l'émetteur et le récepteur. TCP découpe les messages en paquets et veille à ce que ces paquets soient envoyés en même ordre que leur émission. Il assure également la fiabilité de la communication en mettant en place un mécanisme d'accusé de réception et de relance de

paquets perdus. Cela est utile dans le cas d'échange de gros volumes de données de manière fiable comme par exemple en: FTP, SMTP, HTTP, etc. L'émetteur d'un paquet attend une confirmation de la réception de la part du récepteur avant d'envoyer un autre paquet. Le récepteur accuse la réception d'un paquet en envoyant un accusé de réception ou ACK (voir Figure 1).

1.4 Explorer un réseau avec les commandes UNIX

Unix est une famille de systèmes d'exploitation crée par AT&T programmés en C et en assembleur [5]. Il est basé sur un ensemble d'utilitaires depuis la ligne de commande et un interpréteur ou superviseur (le shell). Unix a été rapidement adopté dans les milieux universitaires depuis son apparition. Il a été développé par différentes jeunes start-up pour créer de nouvelles distributions dont les plus populaires à ce jour sont les variantes de BSD (telles que FreeBSD et OpenBSD), GNU/Linux, iOS et macOS.

Différentes commandes UNIX sont proposées pour configurer et explorer le réseau. Nous rappelons dans ce qui suit les principales commandes qui peuvent être utiles pour obtenir des informations et vérifier le réseau avant de programmer ou mettre en place une application répartie sous Unix [3]:

1.4.1 Vérification des interfaces réseau avec dmesg, ifconfig et netstat

Un ordinateur peut disposer de plusieurs cartes réseau auxquelles des noms sont associés. Afin d'avoir des informations sur la machine on peut utiliser la commande **dmesg**, par exemple:

```
> dmesg | grep « eth[0-9]:* "
```

Cette commande donne seulement les informations relatives aux interfaces physiques. Pour obtenir des informations sur les interfaces virtuelles, il faut employer les commandes **netstat** ou **ifconfig**. La liste des interfaces est obtenue par le biais de l'option **-a** de la commande **netstat**. De plus, l'option **–i** permet de visualiser si chaque interface est active et fournir des statistiques relatives aux cartes réseau installées. Voici un exemple d'utilisation de cette commande:

```
> netstat -ia
Kernel Interface table
Iface MTU Met RX-OK RX-ERR RX-DRP RX-OVR TX-OK TX-ERR TX-DRP TX-OVRFlg
eth0   1500   0       1107745      0      0      0      1039503      0      0      0      BRU
eth1   1500   0      126495        1      0      0      205668       0      0      0      BRU
…
```

Voici la signification des champs affichés par **netstat** :

- MTU : correspond à la taille en octets de la plus grande trame (paquet) pouvant être transmise par l'interface.
- Met : fixe le coût pour emprunter une route.
- RX-OK : indique le nombre de trames reçues sans erreur.
- RX-ERR : donne le nombre de trames endommagées qui ont été reçues.
- RX-DRP : donne le nombre de trames reçues qui ont été éliminées (mémoire insuffisante).
- RX-OVR : indique le nombre de trames qui ont été perdues.
- TX-OK : donne le nombre de trames envoyées sans erreur.
- Flg : donne des informations sur l'interface (B: broadcast, P: point à point, R: opérationnelle et U: en cours d'utilisation).

La commande **ifconfig** fixe et vérifie la configuration des interfaces réseau. Voici un exemple de son utilisation pour configurer une interface Ethernet d'un ordinateur:

```
> ifconfig eth0 212.68.194.203 netmask 255.255.255.240 broadcast 212.68.194.207
```

Les principaux arguments pour spécifier les informations de base de TCP/IP pour une interface réseau sont:

- **`interface:`** c'est le nom de l'interface réseau à configurer. Dans l'exemple donné c'est l'interface eth0.
- **`Adresse`**: elle spécifie soit l'adresse IP soit le nom de la machine. Notre exemple affecte l'adresse 212.68.194.203 à l'interface eth0.

- **`Netmask`** : c'est le masque de sous-réseau pour cette interface.
- **`Broadcast`** : c'est l'adresse de diffusion du réseau.

1.4.2 Vérification du masque de sous-réseau

Afin de s'assurer de fonctionnement des interfaces de votre réseau, chaque interface doit posséder le même masque de sous-réseau. Il faut donc vérifier cela en exécutant la commande **ifconfig** pour chaque machine du réseau, par exemple:

```
> ifconfig eth0
```

1.4.3 Récupérer les noms des machines d'un réseau

Les utilisateurs et les développeurs préfèrent généralement d'utiliser les noms des machines au lieu des adresse IP car ils sont plus facile à retenir. Les systèmes UNIX disposent d'un fichier **/etc/hosts** qui associe un ou plusieurs noms à une adresse IP. Pour consulter ce fichier, il suffit d'utiliser la commande **cat**. Voici un exemple de fichier **/etc/hosts** :

```
> cat hosts
127.0.0.1 P100.challe.be P100 localhost.localdomain localhost
212.68.198.209 gateway
212.68.194.200 isec1
```

La commande **nslookup** permet de vérifier la configuration des serveurs de noms. Par défaut, cette commande recherche l'adresse correspondant à un nom ou le nom correspondant à une adresse. D'autres types de recherche peuvent être effectués en employant l'option « q ».

```
>nslookup Default Server: P100.univ.dz
```

1.4.4 Obtenir le manuel des services et fonctions réseaux

Un manuel des services et fonctions est disponible dans toutes les versions Unix/ Linux. Il suffit d'utiliser pour cela la commande **man**. Dans l'exemple suivant le manuel de la commande nslookup sera affiché.

```
> man nslookup
```

Plusieurs fonctions, commandes et appels systèmes existent et parfois ils ne se trouvent pas dans le manuel principal. Il suffit donc de préciser quel manuel à explorer, qui sont numéroter (man2, man3, etc)

1.5 Programmation des applications réparties

Il existe plusieurs approches pour mettre en place des systèmes distribués et faire interagir leurs applications réparties géographiquement et fonctionnellement [2]. Il existe plusieurs formes de répartition sur laquelle se base la programmation des applications nécessitant la communication en réseau:

- **La distribution physique**: en prenant en compte le fonctionnement en LANs ou WANs.
- **La distribution structurelle**: en utilisant par exemple la programmation structurée et modulaire, la programmation orienté objet, la programmation par composants, la programmation par aspects, la programmation par agents, etc.
- **Distribution fonctionnelle**: qui consiste à décomposer l'application en services indépendants (affichage, calcul, stockage de données, etc.) à mettre accessible via le réseau tels que les services Web.

1.5.1 Caractéristiques des applications réparties

Les applications réparties se caractérisent généralement par [2]:

- **L'ouverture et la scalabilité**: cela veut dire que la programmation de l'application doit prendre en compte la possibilité d'extension et/ou d'ajout de communications et le partage de ressources tout en restant efficace avec l'augmentation des ressources et des utilisateurs.

- **La sécurité et la tolérance aux pannes**: l'application doit être sécurisée en incluant des mécanismes d'authentification, des signatures électroniques, etc. Elle doit également avoir la capacité de s'exécuter en environnement dégradé et prévenir les différentes pannes.
- **Transparence pour l'utilisateur**: le standard ISO identifie quelques types de transparences liés par exemple à l'accès et à la localisation. Les ressources locales et distantes doivent pouvoir être accessibles de la même manière pour l'utilisateur.

1.5.2 L'architecture client/serveur

L'architecture d'un système est sa structure en termes de composants conçue dans le but d'assurer son bon fonctionnement à court et à long termes. Elle est décrite par le placement des entités sur le réseau ainsi que les relations existantes entre ces entités. Les principales architectures utilisées sont le Méta-computing (P2P, Clusters, Grilles, Cloud Computing, Agents autonomes, …) et l'architecture client-serveur.

Dans une architecture client/serveur, toute communication d'un groupe est décomposée en un ensemble d'échanges entre deux entités: **le serveur** (interviewé) qui fournit des services et **le client** (interviewer) qui demande des services. Dans cette architecture, le client à l'initiative du dialogue et le serveur est dans l'attente d'une requête. Le serveur dispose d'une liste de services accessibles via des requêtes bien définies en utilisant son interface. A chaque requête correspond un service au niveau du serveur.

Le principe de communication dans une architecture client/serveur est le suivant [1] [2]:

- Le client émet un message contenant une requête à destination d'un serveur.
- Le serveur exécute le service associé à la requête émise par le client.
- Le serveur retourne au client un message contenant le résultat du service effectué.

1.5.3 Communiquer deux applications en réseau

Afin de masquer l'hétérogénéité d'un système distribué, il faut nécessairement utiliser des protocoles standards de communication tels que TCP/IP. Deux techniques sont généralement utilisées pour communiquer deux applications en réseau [2]:

- **Utilisation des interfaces de programmation et des middlewares:** qui utilisent les appels des services directement de la couche transport (TCP ou UDP). On trouve différentes interfaces de programmation et middlewares tels que Socket, RPC, RMI, CORBA, etc.
- **Utilisation du code mobile:** Il s'agit d'envoyer le code source à la machine distante afin qu'elle l'exécute. C'est le cas par exemple: des applets Java, agents mobiles, etc.

Pour programmer la communication entre deux applications distantes, il faut au moins deux informations : **l'adresse IP** et le **port**. On appelle ces deux données un **point de communication**. Toute communication ne peut s'effectuer qu'entre au moins deux points de communications: l'émetteur et le (ou les) récepteur(s).

- **L'dresse IP** est l'identifiant de la machine sur laquelle l'application est exécutée.
- **Le numéro de port** est un identifiant local de l'application. Il s'agit d'un port TCP ou UDP sur lequel se fait la réception ou l'envoi de données. C'est un entier d'une valeur quelconque qui doit être supérieure à 1024. En effet, les autres ports sont réservés pour les applications ou protocoles systèmes.

Le point de communication, ou l'adresse réseau, est notée **@IP:port** ou **nomMachine:port** par exemple 192.129.12.34:80 permet l'accès au serveur Web tournant sur la machine d'adresse IP 192.129.12.34. L'adresse IP peut être remplacée par le nom de la machine utilisant le serveur DNS, par exemple machine2:21.

1.6 Conclusion

Ce chapitre a présenté quelques généralités sur le réseau TCP/IP ainsi que les principales approches de programmation des applications réparties. Le chapitre suivant introduira quelques notions et les packages de base de Java qui seront utilisés plus tard pour programmer des clients et des serveurs qui se communiquent en réseau.

PARTIE II: PROGRAMMATION RÉSEAU EN JAVA

Chapitre 2: Flux d'entrée-sortie, sérialisation et multithread en Java

2.1 Introduction

Ce chapitre rappelle quelques bibliothèques et classes Java pour implémenter une application en Java et connaitre ses propriétés pour les utiliser plus tard en communication réseau. Cela comprend l'utilisation des flux d'entrée/sortie, la sérialisation et la programmation multithread.

2.2 Applications client/serveur en Java

Java est un langage de programmation orienté objet développé par Sun Microsystems qui depuis sa création et jusqu'aujourd'hui représente l'un des langages les plus adoptés par les développeurs et les entreprises de production logicielle. La particularité des logiciels écrits en Java est leurs compilations vers une représentation binaire intermédiaire qui peut être exécutée dans une machine virtuelle Java (JVM) indépendamment d'un système d'exploitation. Java inclut différentes bibliothèques et frameworks permettant de programmer des applications sur une seule machine, en réseau, et en Web.

2.3 Les flux d'entrée/ sortie de Java (java.io)

Une entrée/sortie en Java consiste en un échange de données entre le programme et une autre source, par exemple la mémoire, un fichier, le programme lui-même. Cet échange de données utilise la notion de flux ou « stream ». Le flux peut être considéré comme un tampon mémoire utilisé pour mémoriser les données échangées. Toute opération d'entrée/sortie doit suivre le schéma suivant : ouverture, lecture ou écriture puis fermeture du flux. En effet, Java décompose les objets traitant des flux en deux catégories : les objets travaillant avec des flux d'entrée (in) pour la lecture de flux et les objets travaillant avec des flux de sortie (out) pour l'écriture de flux. Différentes classes de **java.io** peuvent être utilisées. On présentera dans ce qui suit les principales classes utilisées par la suite dans ce cours, notamment la classe File, InputStream, OutputStream, DataInputStream et BufferedInputStream [2].

2.3.1 Les classes File, FileInputStream et FileOutputStream

La classe File est la classe la plus basique utilisée pour lire et écrire des données d'un fichier. Un ensemble de méthodes peut aider à obtenir des informations sur les fichiers de notre système. Ci-après un exemple simple permettant d'ouvrir un fichier puis copier son contenu et le coller dans un autre fichier. On utilise pour cela aussi: les classes **FileInputStream** qui hérite de la classe abstraite **InputStream** (java.io) pour la lecture et la classe **FileOutputStream** qui hérite de la classe abstraite **OutputStream** (java.io) pour l'écriture. Chaque étape de traitement est indiquée dans le code source. Dans ce dernier, le bloc **try{...} catch** est utilisé pour capturer l'exception. La clause **finally** permet d'exécuter le code peu importe que l'exception soit levé ou non.

```
import java.io.*;
public class Main {
    public static void main(String[] args) {
       FileInputStream fis = null;
       FileOutputStream fos = null;
       try {
             /* On crée un flux d'entré pour lire les données du fichier test.txt */
             /* On crée un flux de sortie pour écrire des données dans le fichier test2.txt */
             fis = new FileInputStream(new File("test.txt"));
             fos = new FileOutputStream(new File("test2.txt"));

             /* On crée un tableau de byte pour indiquer le nombre de bytes lus */
             byte[] buf = new byte[8];
             /* On crée une variable n pour y affecter le résultat de la lecture */
             /* qui aut -1 quand c'est fini */
```

```
            int n = 0;
            while ((n = fis.read(buf)) >= 0) {// Tant que la lecture est possible
                fos.write(buf);
                /* On écrit dans notre deuxième fichier avec l'objet adéquat */
                for (byte bit : buf) {
                   System.out.print("\t" + bit + "(" + (char) bit + ")");
                   }
                buf = new byte[8];
            }
          }
         catch (FileNotFoundException e) {
            e.printStackTrace();
         }
         catch (IOException e) {
            e.printStackTrace();
         } finally { /* On ferme nos flux de données */
              try {
                   if (fis != null) fis.close();
                   if (fos != null) fos.close();
              } catch (IOException e) {
                   e.printStackTrace();
                }
             }
      }
 }
```

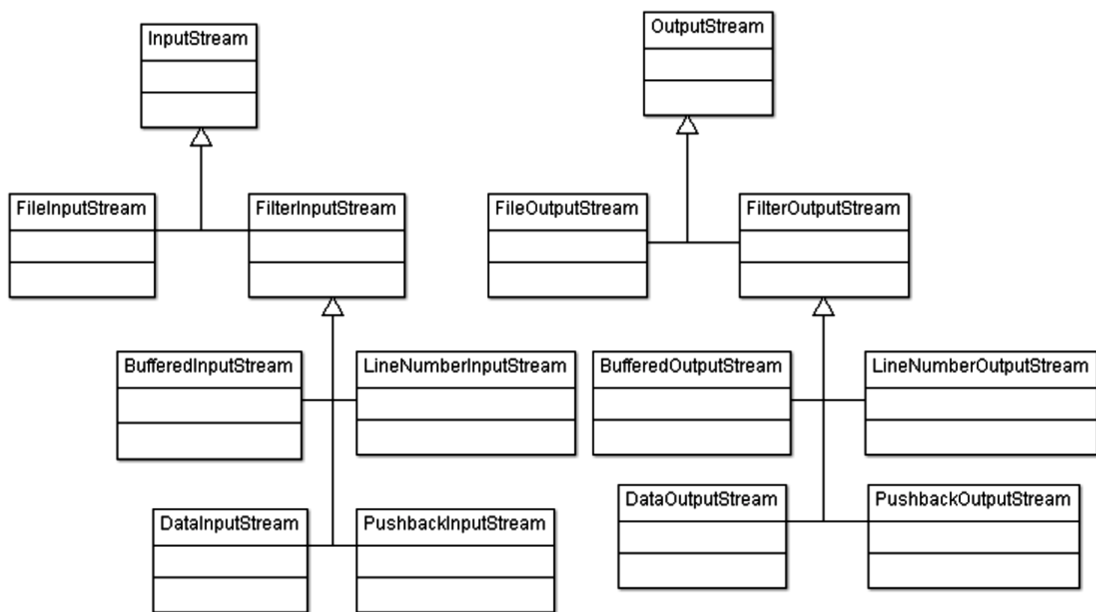


Figure 2: Les classes de gestion de flux en Java

Dans cet exemple, la lecture et l'écriture de données se fait par le moyen d'un tableau de byte qui représente chaque octet lu/écrit. Dans le cas de traitement de données de types variés comme les entiers, réels, objets Java, etc, cette méthode peut s'avérer coûteuse en termes de traitement de données. Java propose donc d'autres classes pour faciliter la lecture et l'écriture de différents types de données sur les flux, notamment DataInputStream, DataOutputStream, BufferedInputStream, BufferedOutputStream, ObjectInputStream et ObjectOutputStream.

2.3.2 DataInputStream et DataOutputStream

La figure 2 illustre les classes Java qui peuvent être utilisées pour manipuler les flux de données. Les classes qui dérivent des classes abstraites FilterInputStream et FilterOutputStream permettent notamment d'ajouter des fonctionnalités aux flux d'entrée/sortie comme par exemple les classes: **DataInputStream** et **DataOutputStream** pour lire/écrire directement des types primitifs (double, char, int) grâce à des méthodes telles que **readDouble(), readInt()** et les classes BufferedInputStream et BufferedOutputStream pour avoir un tampon facilitant la lecture/écriture de données [2]. Ci-après deux exemples d'utilisation.

Exemple 1 : Lecture d'un fichier et l'affichage de son contenu

Pour lire un fichier, il suffit de définir un flux pour récurer les données. Dans cet exemple on utilise **BufferedInputStream.** Une fois le flux est récupéré à partir du fichier, la lecture peut s'effectuer en récupérant successivement les octets à travers un tableau de byte. La méthode **read(byte[])** retourne un entier qui indique si la lecture a été bien effectué. Dans le cas où le tableau de byte est vide et la lecture échoue, la méthode retourne -1.

```
public class Main {
      public static void main(String[] args) {
            BufferedInputStream bis;
            try {
                  bis = new BufferedInputStream(new FileInputStream(new File("test.txt")));
                        byte[] buf = new byte[8];
                  while(bis.read(buf) != -1){
                        System.out.printlen(" ... ", new String(buf, StandardCharsets.UTF_8)
                  }
                        bis.close();
                  } catch (FileNotFoundException e) { e.printStackTrace(); }
                    catch (IOException e) { e.printStackTrace(); }
      }
}
```

Exemple 2: Lecture de données de différents types

Quand le fichier est composé de données de différents types et qu'on connait l'ordre, on peut donc utiliser des méthodes de lecture ou d'écriture spécifiques au type de ces données. Cela évite un post-traitement de tableau de byte (comme dans l'exemple précédent) pour la conversion de type. Voici un exemple à travers lequel on écrit sur un fichier différents types de données puis on les récupère lors de la lecture de fichier à travers les méthodes **writeType()** et **readType()** respectivement.

```
import java.io.*
public class Main {
      public static void main(String[] args) {
      DataInputStream dis; DataOutputStream dos;
      try {
            dos = new DataOutputStream(new BufferedOutputStream(
            new FileOutputStream(new  File("fichier.txt"))));
            //Nous allons écrire chaque type primitif
            dos.writeByte(100); dos.writeChar('C'); dos.writeDouble(12.05);
            dos.writeFloat(100.52f); dos.writeInt(1024); dos.writeLong(123456789654321L);
            dos.close();
            /* On récupère les données */
            dis = new DataInputStream( new BufferedInputStream( new FileInputStream(
                  new File("fichier.txt"))));
            System.out.println(dis.readByte());
            System.out.println(dis.readChar());
            System.out.println(dis.readDouble());
            System.out.println(dis.readFloat());
            System.out.println(dis.readInt());
            System.out.println(dis.readLong());
            dis.close();
      }
      catch (FileNotFoundException e) {e.printStackTrace();}
      catch (IOException e) { e.printStackTrace(); }
   }
}
```

2.3.3 ObjectInputStream et ObjectOutputStream

Les classes BufferedInputStream et BufferedOutputStream vues précédemment permettent de lire et écrire uniquement des types primitifs de données. Afin de lire des objets sur un fichier, les classes **ObjectInputStream** et **ObjectOutputStream** sont utilisées. Par exemple, on souhaite sauvegarder les objets Personne crées par notre programme dans un fichier « personne.txt » puis les récupérer et les afficher à nouveau pour vérifier si la sauvegarde a été bien effectuée. Ci-après un exemple d'implémentation:

```
public class Personne{
	private String nom, prenom;
	private int age;
	public Personne(String nom, String prenom, int age) {
		this.nom = nom;
		this.prenom = prenom;
		this.age = age;
	}
	public String toString(){
		return this.prenom+ "" + this.nom +", il est agé de :" + this.age + " ans \n";
	}
}
import java.io.*
public class Main {
	public static void main(String[] args) {
		ObjectInputStream ois; ObjectOutputStream oos;
		try {
			oos = new ObjectOutputStream(new BufferedOutputStream( new
				FileOutputStream( new File("personne.txt"))));
			/* Nous allons écrire chaque objet dans le fichier */
			oos.writeObject(new Personne("Ahmed ", "Ahmed", 50));
			oos.writeObject(new Personne("Amina", "Amina", 23));
			oos.close();

			ois = new ObjectInputStream( new BufferedInputStream( new
				FileInputStream( new File("personne.txt"))));
			try {
				System.out.println(((Personne)ois.readObject()).toString());
				System.out.println(((Personne)ois.readObject()).toString());
			} catch (ClassNotFoundException e) {// en cas d’erreur
					e.printStackTrace();
					}
			 ois.close();
			} catch (FileNotFoundException e) {
					 e.printStackTrace();
				}
			catch (IOException e) {
					e.printStackTrace();
			}
		}
}
```

Cependant, l’exécution de ce code donne un résultat qui ne peut pas être interprété. Autrement dit, le codage des données affichées n’est pas compréhensible. Pour pouvoir récupérer les données correctement, il faut utiliser la **sérialisation** de données.

2.4 Qu’est-ce que c’est la sérialisation?

La sérialisation (marshalling ou linéarisation) est un procédé qui est complètement indépendant des langages et qui permet d’encoder l’état mémoire d’un objet pour pouvoir le stocker en binaire et le « re-créer » ultérieurement. On peut stocker un objet sérialisé dans un fichier, une base de données ou le communiquer en réseau. Le principe de sérialisation est de décomposer l’objet en éléments les plus petits (jusqu’à descendre à des éléments de types de base du langage), et encoder chacun de ces éléments. Elle conserve également la structure de l’objet pour pouvoir le recomposer au moment de la « désérialisation » qui est l’opération inverse. La sérialisation concerne différents types d’objet tels que les objets composés, tableaux, listes, collections, etc. L’intérêt est de pouvoir stocker les objets et les communiquer entre les applications sans se préoccuper de l’hétérogénéité de ces applications en termes de systèmes, langages de programmation, encodage de données et leurs formats [2].

Afin de garder la structure de l’objet et sauvegarder ses données indépendamment d’un langage de programmation ou format de données, plusieurs techniques ont été proposées notamment:

- L’utilisation des langages semi-formels comme XML et JSON (essentiellement lié à JavaScript)
- Le format XDR (historique C) qui est une représentation dédiée à la sérialisation pour le langage C (cf chapitre RPC)
- Formats binaires spécifiques comme par exemple en Java.

2.4.1 Sérialisation en Java

Afin de faciliter la sérialisation des objets, Java propose une interface dédiée qui permet au développeur d'obtenir un objet sérialisé sans implémenter lui même la décomposition et l'encodage de données. Il s'agit de l'interface **Serializable** de la bibliothèque java.io [1]. Voici un exemple d'utilisation:

```
import java.io.*;
public class Obj implements Serializable{
	private String str;
	public Obj(String s) {str = s;}
	public String toString() { return str;}

}
```

Pour écrire/lire correctement un objet Personne à l'aide des classes **ObjectOutputStream et ObjectInputStream** de la section 2.3.3, il suffit de rajouter à notre code précédent l'implémentation de l'interface **Serializable** dans la classe Personne comme suit:

```
import java.io.Serializable;
public class Personne implements Serializable{
	private String nom, prenom;
	private int age;
	public Personne(String nom, String prenom, int age) {
		this.nom = nom;
		this.prenom = prenom;
		this.age = age;
		}
	public String toString(){
		return this.prenom+ "" + this.nom +", il est agé de :" + this.age + " ans \n";
	}
}
```

De plus, afin de sauvegarder les objets sérialisés, Java utilise le format binaire (.bin). Voici un exemple d'écriture/lecture des objets Personnes.

```
public class Main {
	public static void main(String[] args) throws Exception {
		Personne personne1=new Personne("Amine", "Amine", 23);
		Personne personne2=new Personne("Mohammed", "Mohammed", 22);

		/* sauvegarder les objets dans un fichier .bin */
		FileOutputStream fos = new FileOutputStream("MyObjects.bin");
		ObjectOutputStream objectOut = new ObjectOutputStream(fos);
		objectOut.writeObject(personne1);
		objectOut.writeObject(personne2);
		objectOut.close();

		/* ouvrir à nouveau le fichier puis afficher les objets */
		FileInputStream fis = new FileInputStream("MyObjects.bin");
		ObjectInputStream objectIn = new ObjectInputStream(fis);
		int objectCount=0; Personne p=null;
		while (objectCount < 2) {
			p = (Personne)objectIn.readObject();
			objectCount++;
			System.out.println("Personne n"+objectCount+" : "+p);
			}
		objectIn.close();
	}
}
```

2.5 Les flux d'entrée/ sortie (java.nio)

Le package **nio** « New I/O » de Java a été créé afin d'améliorer les performances de traitement des fichiers, du réseau et des buffers. A l'inverse des objets du package **java.io** qui traitaient les données par octets, **java.nio** permet

de traiter des blocs de données ce qui rend la lecture plus performante. Ce package repose sur l'utilisation de deux objets: **Channels** et **Buffer**[1].

Les **channels** (canaux) sont des flux qui sont amenés à travailler avec un **buffer** (tampon mémoire) avec une taille définie par le développeur. Le principe de l'utilisation des canaux est simple : lorsqu'un flux vers un fichier FileInputStream est ouvert, il suffit de récupérer un canal vers ce fichier via la méthode **getChannel()**. Le canal, combiné à un buffer, permet de lire le fichier encore plus vite qu'avec un BufferedInputStream. Ci-après un exemple d'utilisation de ces classes:

```
import java.io.*
import java.nio.*
import java.nio.channels.*;
public class Main {
       public static void main(String[] args) {
        FileInputStream fis; BufferedInputStream bis; FileChannel fc;
        try {
              fis = new FileInputStream(new File("test.txt"));
              bis = new BufferedInputStream(fis);
              //Lecture
              while(bis.read() != -1){
                            ...
                     }
              /* Création d'un nouveau flux de fichier */
              fis = new FileInputStream( new File("test.txt"));

              /* On récupère le canal */
              fc = fis.getChannel();

              /* On en déduit la taille */
              int size = (int)fc.size();

              /* On crée un buffer correspondant à la taille du fichier */
              ByteBuffer bBuff = ByteBuffer.allocate(size);

              /* Démarrage de la lecture */
              fc.read(bBuff);

              /* On prépare à la lecture avec l'appel à flip */
              bBuff.flip();

              byte[] tabByte = bBuff.array();

              } catch (FileNotFoundException e) { e.printStackTrace();}
                catch (IOException e) { e.printStackTrace();}
         }
}
```

Au lieu d'utiliser des buffers de byte et faire la conversion de type, ce package offre également un buffer par type primitif pour la lecture sur Channel. Les classes de buffer par type sont: IntBuffer, CharBuffer, ShortBuffer, DoubleBuffer, FloatBuffer, LongBuffer.

2.6 Programmation multithread en Java

Un thread est un fil d'exécution dans un processus donné qui se caractérise par: un point courant d'exécution et une pile d'exécution (stack). La JVM de Java est multithread et offre au programmeur la possibilité de créer et gérer des threads en définissant par exemple une classe héritant de la classe **java.lang.Thread**. Une autre solution consiste à associer directement votre classe à un thread en implémentant l'interface **Runnable** de Java.

2.6.1 La classe java.lang.Thread

La création de threads peut se faire en définissant une classe héritant de la classe **java.lang.Thread**. Une fois le thread est lancé, il invoque sa méthode **run()** qui précise ses tâches. Les attributs de la classe **java.lang.Thread** sont:

- String name: le nom donné au thread.
- long id: son identité.
- int priority: sa priorité en termes d'ordonnancement.

- boolean daemon: son mode d'exécution (démon ou non).
- Thread.State state: son état parmi: NEW, RUNNABLE, BLOCKED, WAITING, TIMED WAITING, TERMINATED.

Les principales méthodes de la classe **java.lang.Thread** sont:

- Thread (Runnable) : constructeur par défaut.
- Thread (Runnable, String) : constructeur avec le nom du thread en paramètre.
- void start () : pour démarrer le thread.
- void run () : pour exprimer les tâches que le thread devra faire une fois démarré.
- void join () : permet d'attendre que le thread s'arrête.
- void interrupt () : rendre le statut du thread "interrompu".
- Thread currentThread () : pour récupérer l'objet thread courant.
- void sleep (long ms) et void sleep (long ms,long ns) : pour renoncer à l'exécution pour la durée exprimée.
- void yield () : permet au thread courant de renoncer et reprendre une place dans l'ordonnanceur.

2.6.2 Synchroniser les threads

La synchronisation de threads permet de rendre le programme concurrent tout en évitant les problèmes d'ordonnancement et d'ambiguïté lors de l'accès à la mémoire. Cela nécessite la définition de « sections critiques », ou une suite d'instructions qui doivent êtres exécutées uniquement par un thread à la fois. En Java, on utilise le mot-clé **synchronized** qui permet de définir une section critique dans la déclaration des méthodes.

Cependant, une section critique n'est pas toujours isolée du reste de l'application. Il faut donc gérer comment les threads se synchronisent pour éviter le problème d'**attente d'active**. C'est le cas par exemple du problème classique de producteurs/consommateurs. Les producteurs produisent des objets et les stockent, sachant que la capacité de stockage est limitée. Les consommateurs retirent des objets du stock. Dans cet exemple, le problème de synchronisation est du aux vitesses relatives des opérations de consommation et production. Pour pouvoir synchroniser les threads, il faut utiliser des verrous. Pour se faire, Java propose d'utiliser les méthodes **wait ()** et **notify () / notifyAll ()**.

La méthode **wait ()** permet de relâcher un verrou que l'on possède et de se mettre en attente jusqu'a ce que quelqu'un nous autorise à tenter de le reprendre. La méthode **notify () / notifyAll ()** permet d'autoriser un ou plusieurs threads qui sont en attente de tenter de reprendre le verrou qu'il possédait. Dans ce qui suit, on présentera comment utiliser ces méthodes à travers l'exemple de producteurs/consommateurs. Dans cet exemple, on aura besoin de quatre classes: le produit, le stock, le producteur et le consommateur. On suppose que le stock à une capacité maximale de 10 produits.

```
/* afin de simplifier l'exemple, un produit est caractérisé uniquement par son nom */
class Produit {
       private String nom;
       public Produit(String nom) { this.nom = nom; }
       public String toString() { return this.nom; }
}


/* La classe stock inclut un tableau d'une capacité maximale limitée
 * utilisé pour sauvegarder les produits donnés par les producteurs
 */
class Stock {
       private Produit []leStock;
       private int niveauCourant;
       /* on suppose que la capacité maximale de stock est 10 */
       private static final int capacitéMaximale = 10;

       /* création du stock */
       public Stock() {
              leStock = new Produit[capacitéMaximale];
              niveauCourant = 0;
       }

       /* ajouter un produit p au stock par un producteur
```

```
        * L'utilisation de synchronized, wait()…notifyAll() permettent d'assurer qu'un seul
        *  producteur doit ajouter son produit au stock à la fois, les autres doivent attendre
        * leurs tours et recevront une notification une fois l'accès au stock est libre
        */

       public synchronized void addProduit(Produit p) {
              while (niveauCourant==capacitéMaximale) {
                     System.out.println("plein!");
                     try {
                            wait();
                     } catch(InterruptedException e) { }
              }
              leStock[niveauCourant++] = p;
              notifyAll();
       }

       /* supprimer un produit p récupéré par un consommateur */

       public synchronized Produit removeProduit() {
              while (niveauCourant==0) {
                     System.out.println("vide!");
                     try {
                            wait();
                     } catch(InterruptedException e) { }
              }
              Produit p = leStock[niveauCourant-1];
              leStock[--niveauCourant] = null;
              notifyAll();
              return p;
       }
}

/* La classe producteur utilise les threads, plusieurs producteurs peuvent êtres crées */

class Producteur extends Thread {
       private Stock stock;
       private String nomProduit;
       private Random random;
       Producteur(Stock stock,String nomProduit) {
              this.stock = stock;
              this.random = new Random();
              this.nomProduit = nomProduit;
       }

       /* la méthode run indique ce que le thread devra faire une fois il est lancé */

       public void run() {
              while (true) {
                     Produit p = new Produit(nomProduit);
                      try {
                            Thread.sleep(random.nextInt(1000)+1000);
                     } catch(InterruptedException e) { }

                     if (!stock.addProduit(p)) {
                            do {
                                   try {
                                          System.out.println("Producteur "+getId()+" plein");
                                                 Thread.sleep(random.nextInt(100)+100);
                                   } catch(InterruptedException e) { }
                            } while (!stock.addProduit(p));
                     }
                     System.out.println("Producteur "+getId()+" a rajoute "+p);
              }
       }
}

class Consommateur extends Thread {
       private Stock stock;
       private Random random;
       public Consommateur(Stock stock) {
              random = new Random();
              this.stock = stock;
       }
```

```
	public void run() {
		while (true) {
			try {
				Thread.sleep(random.nextInt(10000)+1000);
			} catch(InterruptedException e) { }
			Produit p = stock.removeProduit();
			if (p==null) {
				do {
					try {
						System.out.println("Consommateur "+getId()+ " vide");
						Thread.sleep(random.nextInt(100)+100);
					} catch(InterruptedException e) { }
					p = stock.removeProduit();
				} while (p==null);
			}
			System.out.println("Consommateur "+getId()+" a enlevé "+p);
		}
	}
}

public class Main{
	public static void main(String []args) {
		Stock stock = new Stock();
		/* simple test en créant 2 producteurs et 3 consommateurs */
		Producteur []p = new Producteur[2];
		p[0] = new Producteur(stock,"Cahier");
		p[1] = new Producteur(stock,"Stylo");
		p[0].start();p[1].start();
		Consommateur []c = new Consommateur[3];
		c[0] = new Consommateur(stock);
		c[1] = new Consommateur(stock);
		c[2] = new Consommateur(stock);
		c[0].start(); c[1].start(); c[2].start();

	}
}
```

2.7 Conclusion

Ce chapitre a rappeler les principales classes et packages Java qui aident à la gestion des flux d'entrés/sorties, threads et la sérialisation. Même si le concept de « flux » permet d'échanger les données entre processus, cela est valable uniquement pour les processus qui s'exécutent sur la même machine en local. Afin de permettre la communication distante entre deux applications, les interfaces de programmation réseau qui se basent sur les appels de services des couches UDP et TCP sont utilisées. On présentera dans le chapitre suivant le package java.net permettant de faciliter la programmation de la communication via un réseau TCP/IP.

Chapitre 3: Sockets TCP/UDP en Java

3.1 Introduction

Ce chapitre introduira la programmation réseau en langage Java à l'aide de l'interface **Socket**. Cette interface de programmation réseau permet d'appeler des services de la couche transport afin de spécifier les paquets échangés entre le serveur et le (s) client (s). L'interface de programmation Socket est proposée dans différents langages de programmation. Des exemples de programmes clients et de serveurs en Java seront fournit dans ce chapitre suivant le protocole adopté par le développeur (TCP ou UDP).

3.2 L'interface de programmation Socket

Les sockets sont inspirées de la communication interprocessus dans le système UNIX qui considère que n'importe quel échange peut s'effectuer à travers un tampon mémoire qui se manipule par la gestion des entrées/sorties. Socket (ou prise) est un point d'accès aux couches TCP/UDP permettant la communication avec un port distant sur une machine (ou application) distante. Afin de communiquer deux applications distantes à travers les sockets, il faut connaître précisément la localisation d'un des deux éléments communicants. L'application serveur, ou celle qui fournit des services, doit être connue par les clients. Le client doit connaitre le point de communication du serveur, c'est à dire **l'adresse IP et le port**, et c'est à lui d'initier la connexion ou la communication avec le serveur. Ce dernier utilise un **socket** lié à un port précis appelé « **un port d'écoute** » afin de recevoir l'ensemble de requêtes venant d'un ou plusieurs clients.

3.3 Sockets UDP en Java

Les sockets UDP sont basés sur un mode de communication par paquets de données. La communication ne nécessite pas l'établissement de la connexion entre les parties client et serveur. Elle s'initie uniquement par la réception d'un message du client par le serveur. Les primitives de communication proposées par les différents langages de programmation ont des caractéristiques communes qui sont [1]:

- **L'émission de paquets est non bloquante:** cela signifie que le serveur par exemple quand il envoie une réponse au client, il ne reste pas bloqué pour attendre la confirmation de l'envoie.
- **Réception de paquets est bloquante:** quand la partie client ou serveur possède la primitive d'attente de réception d'un message, elle reste en attente jusqu'à la réception de ce dernier.
- **La fiabilité de la communication et sa gestion n'est pas disponible:** les sockets UDP n'offrent pas la possibilité de gérer la vérification de réception de paquets ou de gérer la retransmission des paquets perdus.

La communication entre un client et un serveur via les sockets UDP passe par les étapes suivantes:

1. Le serveur crée un socket et le lie à un port UDP.
2. Le client crée un socket pour accéder à la couche UDP et le lie sur un port quelconque.
3. Le serveur se met alors en attente de réception de paquets sur son socket.
4. Le client envoie un paquet via son socket en précisant l'adresse du destinataire : couple adresse IP et port de la partie serveur.
5. Le paquet sera reçu par le serveur (sauf en cas de problème réseau). L'adresse du client (adresse IP et port) est précisée dans le paquet, le serveur peut alors lui répondre.
6. Le client reçoit le paquet de réponse.

Java intègre nativement les fonctionnalités de communication réseau au dessus de TCP-UDP/IP via le package **java.net**. Les classes utilisées pour assurer la communication via UDP sont :

- **InetAddress**: permet d'assurer le codage des adresses IP.
- **DatagramSocket** : est utilisée pour la création d'un socket UDP par le serveur et par le client.
- **DatagramPacket** : aide à créer un paquet de données à envoyer via le socket précédemment crée.

3.3.1 Schéma fonctionnel de la communication client/serveur avec socket UDP en Java

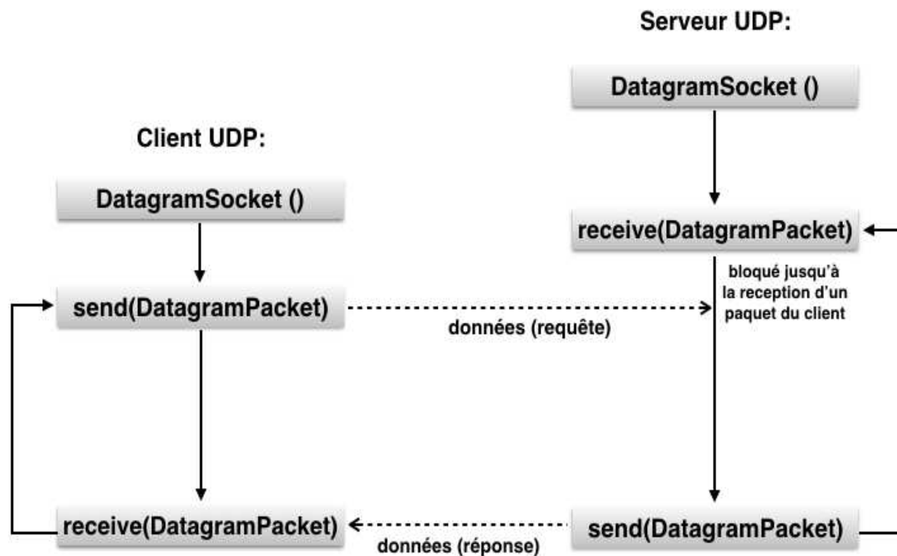


Figure 3: Schéma fonctionnel de la communication entre un client et un serveur UDP en Java.

Figure 3 résume le schéma fonctionnel de la communication entre un client et un serveur UDP écrits en Java. Voici les primitives essentielles de la communication entre le serveur et le client [2].

1. Serveur: crée un socket à l'aide de l'instanciation de la classe **DatagramSocket** et le lie à un port UDP.
2. Serveur: se met en attente de réception de requêtes en appelant la méthode bloquante **receive(DatagramPacket**). Il faut donc avant préparer un paquet **DatagramPacket** vide à mettre comme paramètre de sortie de cette méthode.
3. Client: de son côté, le client crée également un socket à l'aide de l'instanciation de la classe **DatagramSocket** pour accéder à la couche UDP et le lie sur un port quelconque. Il pourra ainsi communiquer avec le serveur.
4. Client: prépare un paquet de données **DatagramPacket** contenant sa requête puis l'envoie via son socket au serveur en précisant l'adresse IP et port de ce dernier. Il utilise pour cela la méthode **send(DatagramPacket).** Il se mettra ensuite en attente bloquante pour recevoir la réponse en appelant la méthode **receive(DatagramPacket).**
5. Le paquet sera reçu par le serveur (sauf en cas de problème réseau). L'adresse IP du client et son port se trouvent dans le paquet, le serveur peut alors lui répondre en appelant la méthode **send(DatagramPacket)** qui contient le paquet de réponse.
6. Le client reçoit le paquet de réponse et récupère ses données.

Les détails de chaque méthode seront décrits dans les sous-sections qui suivent.

3.3.2 La classe InetAddress

Afin de définir l'adresse IP du serveur, il suffit d'utiliser la classe **InetAddress**. Cette classe n'a pas de constructeurs, il faut donc passer par des méthodes statiques pour créer un objet:

- **public static InetAddress getByName (String host) throws UnknownHostException**: cette méthode crée un objet InetAddress identifiant une machine dont le nom est passé en paramètre. L'exception est levée si le service de nom (DNS...) du système ne trouve pas de machine du nom passé en paramètre sur le réseau.

D'autres méthodes sont utiles pour récupérer l'adresse et le nom de la machine:

- **public static InetAddress getLocalHost () throws UnknownHostException**: retourne l'adresse IP de la machine sur laquelle tourne le programme, c'est-à-dire l'adresse IP locale.
- **public String getHostName ()** : retourne le nom de la machine dont l'adresse est codée par l'objet InetAddress.

3.3.3 La classe DatagramPacket

Cette classe permet de préparer un paquet pour l'envoie de requêtes et/ou de réponses en mode UDP. Deux constructeurs sont possibles pour créer l'objet:

- **public DatagramPacket (byte[] buf, int length)**: la création du paquet se fait à l'aide d'un tableau d'octets qui représente le buffer dans lequel les données sont stockées. Le paramètre length précise la taille maximale des données à lire. Il ne faut pas préciser une taille plus grande que celle du tableau pour ne pas obtenir des erreurs lors de l'exécution.
- **public DatagramPacket (byte[] buf, int length, InetAddress address, int port)**: avec ce constructeur on rajoute les informations sur l'adresse IP de la machine destinataire des données ainsi son numéro de port.

D'autres méthodes sont utiles pour obtenir les données d'un paquet, notamment:

- **InetAddress getAddress () :** Si c'est un paquet à envoyer, alors la méthode retourne l'adresse de la machine destinataire. Dans le deuxième cas, il s'agit d'un paquet reçu, et donc la méthode retourne l'adresse de l'émetteur.
- **int getPort () :** si c'est un paquet à envoyer, cette méthode retourne le port du destinataire. Si c'est un paquet reçu, elle retourne le port utilisé par le programme distant pour envoyer le paquet.
- **byte[] getData () :** retourne un tableau de byte de données contenues dans le paquet.
- **int getLength () :** retourne la taille des données du paquet.

Des méthodes sont aussi utilisées pour remplir le paquet, il s'agit de:

- **void setAddress (InetAdress adr)**: positionne l'adresse IP de la machine destinataire du paquet.
- **void setPort (int port)**: précise le port de destinataire du paquet.
- **void setData (byte[] data)**: intègre les données à envoyer au paquet.
- **int setLength (int length)**: positionne la longueur des données à envoyer.

3.3.4 La classe DatagramSocket

Cette classe aide à créer un socket serveur ou client. Elle possède deux constructeurs:

- **public DatagramSocket () throws SocketException:** crée un socket en le liant à un port quelconque libre.
- **public DatagramSocket (int port) throws SocketException**: crée un socket en le liant au port local précisé par le paramètre port. L'exception est levée en cas de problème, notamment quand le port est déjà occupé.

3.3.5 Méthodes d'émission/réception de paquets

Une fois le socket est crée au niveau des deux parties client et serveur de l'application, il sera possible d'envoyer et recevoir les paquets UDP. Pour cela les deux méthodes suivantes sont utilisées:

- **public void send (DatagramPacket p) throws IOException**: permet l'envoie du paquet passé en paramètre. Une exception est levée en cas de problème d'entrée/sortie.
- **public void receive (DatagramPacket p) throws IOException**: permet de recevoir un paquet de données. Elle est bloquante tant qu'un paquet n'est pas reçu. Les données reçues sont copiées dans le tableau passé en paramètre lors de la création de p et sa longueur est positionnée avec la taille des données reçues. Les attributs adresse et de port de p sont modifiés par les données du socket distant qui a émis le paquet.

3.3.6 Autres méthodes

D'autres méthodes sont également utilisées dans le cadre de communication client/serveur en UDP:

- **public int getLocalPort ()**: retourne le port local sur lequel est lié le socket.
- **public void setSoTimeout (int timeout) throws SocketException**: permet de préciser un délai maximum d'attente pour la méthode receive qui est bloquante.
- **public void close ()**: ferme le socket et libère son port.

3.3.7 Exemples de client/serveur avec Socket UDP en Java

Dans l’exemple qui suit on crée deux classes: Serveur.java et Client.java. Dans cet exemple, le client envoie un message au serveur sur le port 7777 et la machine localhost. Le serveur répond en accusant la réception et en envoyant le même message reçu (écho). Le serveur affiche également de son côté les informations sur les paquets reçus des clients.

```
/*Serveur.java*/
import java.io.*;
import java.net.*;
public class Serveur {

public static void main(String args[]){
      DatagramSocket sock = null;
      try {
            //1. création de socket serveur sur le port local 7777
            sock = new DatagramSocket(7777);

            //un buffer pour recevoir les données entrantes dans le paquet
            byte[] buffer = new byte[65536];
            DatagramPacket paquetEntrant = new DatagramPacket(buffer, buffer.length);

            //2. Attente pour recevoir des données par l’appel de receive
            System.out.println(”Serveur crée. Attente des demandes client...”);

            //boucle de communication avec le client
            while(true){
                  sock.receive(paquetEntrant);
                  byte[] data = paquetEntrant.getData();
                  //afficher les données reçues: ip, port et message client
                  String s = new String(data, 0, paquetEntrant.getLength());
                  System.out.println(paquetEntrant.getAddress().getHostAddress() + ” : ” +
                        paquetEntrant.getPort() + ” - ” + s);
                  s = ”OK : ” + s;

                  DatagramPacket paquetSortant = new DatagramPacket(s.getBytes() ,
                  s.getBytes().length , paquetEntrant.getAddress() , paquetEntrant.getPort());

                  sock.send(paquetSortant);
            }
      }
      catch(IOException e){
            System.err.println(”IOException ” + e);
      }
  }
}
```

```
/*Client.java*/

public class Client{

  public static void main(String args[]){

      DatagramSocket sock = null;
      int port = 7777; // port du serveur String message;
      BufferedReader cin = new BufferedReader(new InputStreamReader(System.in));
       try{
            sock = new DatagramSocket();
            InetAddress host = InetAddress.getByName(”localhost”);

            // boucle de communication avec le serveur
            while(true){
                  //lire un message et envoyer le paquet
                  System.out.println(”Entrer le message à envoyer : ”);
                  message = (String)cin.readLine();
                  byte[] b = message.getBytes();
                  DatagramPacket paquetSortant =
                        new DatagramPacket(b , b.length , host , port);
                  sock.send(paquetSortant);
```

```
                    /recevoir la réponse par un buffer
                    byte[] buffer = new byte[65536];
                    DatagramPacket paquetEntrant = new DatagramPacket(buffer, buffer.length);

                    sock.receive(paquetEntrant);
                    byte[] data = paquetEntrant.getData();
                    message = new String(data, 0, paquetEntrant.getLength());
                     // afficher ip, port et message du client
                    System.out.println(paquetEntrant.getAddress().getHostAddress() + ” : ” +
                    paquetEntrant.getPort() + ” - ” + message);
              }
       }
    catch(IOException e) {
        System.err.println(”IOException ” + e);
    }
  }
}
```

Compilation et exécution

Voici les étapes de compilation et d’exécution sous Linux:

1) **javac** ServeurUdpEcho.java
2) **javac** ClientUdpEcho.java
3) **java** ServeurUdpEcho
4) **java** ServeurUdpEcho

Les mêmes commandes sont valables pour Windows. Il est également possible d’utiliser un environnement de développement (exp. Eclipse) en créant deux classes et en lançant leurs exécutions séparément.

Résultat d’exécution

Serveur: Serveur crée. Attente des demandes client...
Client: Entrer le message à envoyer : test
Serveur:127.0.0.1 : 60068 - test
Client: 127.0.0.1 : 7777 - OK : test
Client: Entrer le message à envoyer : je suis connecté
Serveur: 127.0.0.1 : 60068 - je suis connecté
Client: 127.0.0.1 : 7777 - OK : je suis connecté
Client: Entrer le message à envoyer : bye
Serveur: 127.0.0.1 : 60068 - bye
Client: 127.0.0.1 : 7777 - OK : bye

Dans cet exemple, le serveur et le client se trouvent sur la même machine (localhost). On peut également mettre le code/exécutable du programme client et du programme serveur dans des machines différentes qui sont liées par le réseau. Il suffit de mettre le nom de la machine ou l’adresse IP dans la variable « host » de cet exemple de client.

De plus, le serveur envoie le même message qu’il a reçu du client. On peut envoyer différentes requêtes selon les services proposés par un serveur. Par exemple, on souhaite que le serveur traduise cette fois-ci quelques mots envoyés par le client. Le serveur répond par le message « Hello » quand le client envois le message « Salut» et par « Good Bye » quand le client envoie le message « Au revoir ». Le serveur écrit précédemment peut être modifié comme suit:

```
public class Serveur {
    public static void main(String args[]) throws SocketException, IOException{
         /* … */
      while(true){
        /* … */
        sock.receive(paquetEntrant);
              /* récupérer le message du client*/
        byte[] data = paquetEntrant.getData();
        String s = new String(data, 0, paquetEntrant.getLength());

        String reponse=Traitement(s);
       DatagramPacket paquetSortant = new DatagramPacket(reponse.getBytes() ,
              reponse.getBytes().length , paquetEntrant.getAddress() , paquetEntrant.getPort());
```

```
        sock.send(paquetSortant);

       }catch(IOException e){
             System.err.println("IOException " + e);

         }
     }
/* de traitement de la réponse se fera à l'aide de cette méthode */
  public static String Traitement(String s){
      if(s.toUpperCase().equals("Salut")) return "HELLO";
       else if(s.toUpperCase().equals("Au revoir ")) return "GOOD BYE »;
              else return"impossible de traduire, mot inconnu";
 }
```

Cet exemple peut être enrichi côté serveur en disposant d'une base de données permettant la traduction de plusieurs mots envoyés par le client. Le traitement de requêtes ne se limite pas aux chaines de caractères, le serveur et les clients peuvent échanger différents types de données tels que les entiers, réels, etc.

3.4 Socket TCP en Java

La communication en TCP nécessite une phase de connexion explicite entre le client et le serveur avant l'échange de messages. Les données sont envoyées en socket Java comme des flux (virtuels) et non pas par paquets. TCP assure la réception de données dans le bon ordre avec une transparence pour faciliter la tâche au programmeur [1]. Voici les étapes de communication entre un client et un serveur en utilisant les Socket TCP:

1. Le serveur lie un **socket d'écoute** sur un port et appelle un service d'attente de connexion.
2. Le client crée un **socket** lié à un port quelconque puis ouvre une connexion avec le serveur sur son socket d'écoute.
3. Du côté du serveur, le service d'attente de connexion retourne un **socket de service** (associé à un port quelconque).
4. Le client et le serveur communiquent en envoyant et recevant des données via leurs sockets respectifs.

La particularité de la communication en TCP par rapport aux sockets UDP en Java est que les données échangées ne sont plus des tableaux d'octets. On utilise les flux Java. Chaque socket possède un **flux d'entrée** et un **flux de sortie.** Cette communication permet d'envoyer facilement n'importe quel objet ou donnée via des sockets.

Les classes du package **java.net** utilisées pour la communication en socket TCP sont:

- **InetAddress:** la même classe de codage des adresses IP décrite dans la partie UDP.
- **Socket** : est utilisée pour la création d'un socket par le serveur et par le client afin d'échanger des données.
- **ServerSocket** : utilisée uniquement par le serveur. Appelé un socket d'écoute, elle s'occupe uniquement de la réception des requêtes.

3.4.1 Schéma fonctionnel de la communication client/serveur en socket TCP

Figure 4 résume le schéma fonctionnel de la communication entre un client et un serveur TCP écrits en Java. Voici les étapes essentielles de la communication à travers les classes et méthodes du **java.net**:

1. Serveur: crée un socket d'écoute à l'aide de l'instanciation de la classe **ServerSocket** et le lie à un port TCP ainsi qu'un socket de service **Socket** afin de pouvoir accepter par la suite les requêtes des clients. Il fait un appel de la méthode **accept** qui permet de le mettre en attente bloquante d'une requête de client.
2. Client: de son côté, le client crée également un socket à l'aide de l'instanciation de la classe **Socket** pour accéder à la couche TCP et le lie à un port quelconque.
3. Client: prépare un message à envoyer à travers le flux de sortie **OutputStream**. Il se mettra ensuite en attente pour recevoir la réponse en utilisant son flux d'entré **InputStream.**
4. Serveur: une fois accept () retourne un résultat, cela veut dire une connexion s'est établit avec un client. Dans ce cas, le serveur récupère le message reçu à travers son flux d'entré via **getInputStream.** Une fois la requête analysée, le serveur pourra répondre au client en envoyant un message via son flux de sortie via **getOutputStream**.
5. Le client peut ensuite fermer la connexion quand la communication se termine. Le serveur aussi ferme la connexion avec ce client et il se remet en attente de nouveaux clients.

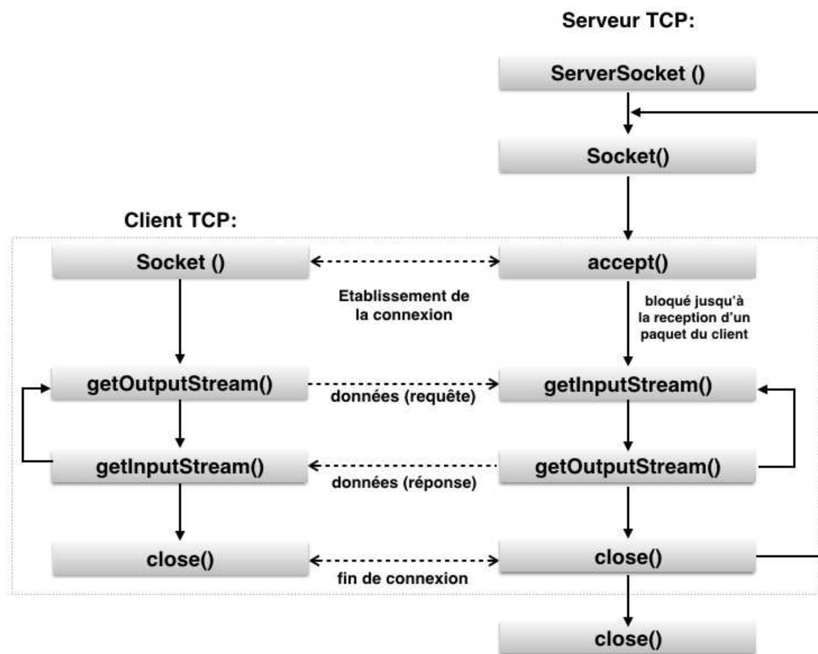


Figure 4: Schéma fonctionnel de la communication entre un client et un serveur TCP en Java.

3.4.2 La classe Socket

Cette classe possède deux constructeurs:

- **public Socket (InetAddress address, int port) throws IOException**: crée un socket local et le connecte à une machine distante.
- **public Socket (String address, int port) throws IOException, UnknownHostException** : idem que le précédent, mais avec nom de la machine au lieu de son adresse IP en argument. La création de socket lève l'exception UnknownHostException si le service de nom ne parvient pas à identifier la machine.

3.4.3 Méthodes d'émission et de réception de données

Contrairement aux sockets UDP, les sockets TCP n'offrent pas directement des services pour émettre/recevoir des données. On récupère les flux d'entrée/sorties associés au socket:

- **OutputStream getOutputStream ()**: retourne le flux de sortie permettant d'envoyer des données via le socket.
- **InputStream getInputStream ()**: retourne le flux d'entrée permettant de recevoir des données via le socket.

3.4.4 Autres méthodes

D'autres méthodes sont utilisées, notamment:

- **public void setSoTimeout (int timeout) throws SocketException**: positionne l'attente maximale en réception de données sur le flux d'entrée du socket.
- **int getPort ()**: renvoie le port distant avec lequel est connecté le socket.
- **InetAddress getAddress ():** renvoie l'adresse IP de la machine distante.
- **int getLocalPort ():** renvoie le port local sur lequel est lié le socket.
- **public close ()**: ferme le socket et rompt la connexion avec la machine distante.

3.4.5 La classe ServerSocket

Le constructeur de cette classe est utilisé pour créer le socket d'écoute dans la partie serveur. Sa syntaxe est la suivante: **public ServerSocket(int port) throws IOException**. Le socket est lié au port dont le numéro est passé en paramètre. Cette classe possède différentes méthodes, en particulier:

- **Socket accept () throws IOException**: permet l'attente de la connexion d'un client distant. Quand connexion est établie, elle retourne un socket permettant de communiquer avec le client.
- **void setSoTimeout (int timeout) throws SocketException**: positionne le temps maximal d'attente de connexion sur un « accept()». Par défaut, l'attente est infinie.

3.4.6 Exemples de client/serveur avec Socket TCP en Java

C'est le même exemple de client/serveur d'écho vu dans la section Socket UDP. Cette fois-ci, la communication entre le client et le serveur s'arrête quand le client envoie le message « stop ».

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

public class Serveur{

 public static void main(String[] args) throws IOException {
       ServerSocket serverSocket = null;
       try {
              serverSocket = new ServerSocket(7007);
       } catch (IOException e) {
              System.err.println("port occupé");
              System.exit(1);
       }
       Socket clientSocket = null;
       System.out.println ("Attente de connexion ...");

       try {
              clientSocket = serverSocket.accept();
       } catch (IOException e) {
              System.err.println("échec de connexion.");
              System.exit(1);
       }
       // la connexion est  établie , envoie de données
       System.out.println ("Connexion établie ... ");

       BufferedReader in = new BufferedReader( new
                          InputStreamReader(clientSocket.getInputStream()));
       PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);

       String inputLine;
       while ((inputLine = in.readLine()) != null) {
              System.out.println ("Serveur: " + inputLine);
              out.println(inputLine);
              if (inputLine.equals("stop"))
                     break;
       }
       out.close();
       in.close();
       clientSocket.close();
       serverSocket.close();
 }
}
```

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

public class Client{

 public static void main(String[] args) throws IOException {

       System.out.println ("Connexion au 127.0.0.1 au port 1007");
       Socket echoSocket = null;
       PrintWriter out = null;
```

```
       BufferedReader in = null;
       try {
              echoSocket = new Socket("127.0.0.1", 7007);
              out = new PrintWriter(echoSocket.getOutputStream(), true);
              in = new BufferedReader(new InputStreamReader( echoSocket.getInputStream()));
       } catch (UnknownHostException e) {
              System.err.println("machine non connue: ");
              System.exit(1);
              }catch (IOException e) {
              System.err.println("erreur d'E/S");
              System.exit(1);
       }

       // récupérer le flux et communiquer avec le serveur
       BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
       String userInput;
       System.out.print ("input: ");
       while ((userInput = stdIn.readLine()) != null) {
              out.println(userInput);
              System.out.println("echo: " + in.readLine());
              System.out.print ("input: ");
              }
       out.close(); in.close(); stdIn.close(); echoSocket.close();
  }
}
```

Dans cet exemple, le serveur et le client se trouvent sur la même machine (localhost). On peut tout même mettre le code/exécutable du client et du serveur dans des machines différentes qui sont liées par un réseau. Il suffit de mettre le nom de la machine lors de la création du socket (dans la partie client).

De plus, le serveur envoie le même message qu'il a reçu du client. On peut programmer avec les mêmes fonctions de communication précédemment vues des serveurs avec différents services. Prenant par exemple un serveur de date, qui à chaque requête d'un client, envoie la date du jour. Le client doit envoyer le message « date » pour recevoir la date du jour, « echo » pour recevoir le même message qu'il vient d'envoyer ou « stop » pour arrêter la communication. On modifiera alors le code du serveur de l'exemple précédent comme suit:

```
import java.net.*;
import java.io.*;
import java.time.LocalDate;

public class ServeurDate{

  public static void main(String[] args) throws IOException {

       /* … */
       String inputLine;
       while ((inputLine = in.readLine()) != null) {
              if (inputLine.equals("date")){
                     LocalDate d= LocalDate.now();
                     out.println(d.toString());
              }
              else if(inputLine.equals("echo"))
                     out.println(inputLine);
              else if (inputLine.equals("stop"))
                     break;
              else out.println("reformuler votre requête svp: date, echo ou stop “);
       }
       out.close();
       in.close();
       clientSocket.close();
       serverSocket.close();
  }
}
```

3.5 La sérialisation des objets Sockets

Comme expliqué précédemment, la sérialisation consiste à décomposer l'objet en plus petits éléments (aux types de base du langage), et encoder chacun de ces éléments. La sérialisation est nécessaire dans un serveur/client TCP

qui s'échangent des objets, les deux doivent connaitre la structure de l'objet à communiquer. Pour cela, objet doit implémenter l'interface Serializable. Voici un exemple:

3.5.1 Exemple d'échange d'objets sérialisé en Socket TCP

```
import java.io.*;

public class Obj implements Serializable {
	private String str;
	public Obj(String s){
		str = s;
	}
	public String toString(){
		return str;
	}
}

import java.io.*;
import java.net.*;
public class ServeurRecepteur {
   public static void main(String[] args) throws Exception {
	int objectCount = 0;
	ServerSocket ss = new ServerSocket(61234);
	while (true) {
		Socket service = ss.accept();
		ObjectInputStream objectIn = new ObjectInputStream(service.getInputStream());
		Obj object = (Obj)objectIn.readObject();
		objectCount++;
		System.out.println(”Objet numéro :”+objectCount+” est : ”+object);
	}
  }
}

import java.io.*;
import java.net.*;

public class ClientEmetteur {
   public static void main(String[] args) throws Exception {
	Obj obj1 = new Obj(”obj1”);
	Socket s = new Socket(”localhost”,61234);
	ObjectOutputStream objectOut =new ObjectOutputStream(s.getOutputStream());
	objectOut.writeObject(obj1);
	objectOut.flush();
  }
}
```

3.6 Programmation réseau multithread

Les serveurs TCP et UDP que nous avons programmés précédemment sont itératifs. Ce mode n'est en général utile que dans le cas où les communications sont courtes. Quand le traitement est conséquent, un problème se pose avec le serveur employé. En effet, pendant que le service se déroule, **ServerSocket** n'est pas dans un état acceptant (i.e. pas dans l'appel à accept()) et donc les communications entrantes sont bloquées. Il est donc nécessaire de faire un parallélisme des traitements de service en déléguant le service à un concurrent qui est un autre processus ou thread. Afin de programmer un serveur multi-clients en Java, on va se baser sur la programmation multithread vue en Chapitre 2.

Le principe est assez simple, au lieu d'associer au processus serveur directement le socket de service pour communiquer avec un client, il faut déléguer cette tâche à un thread. Cela va permettre le principal thread du serveur de s'occuper du reste des clients. A chaque client arrivé, un socket de service (respectivement un thread) est associé.

L'exemple suivant donne un aperçu sur la structure général du code source pour le serveur multi-clients. Au lieu d'avoir une seule classe pour le serveur, en créa deux classes: une principale **serveur** et la seconde est le **service** qui implémente l'interface et hérite de la classe Thread et qui est responsable de la communication avec le client. Voici la structure du code source du service:

```
class Service implements Runnable {

       private Socket maSocket;

       Service(Socket s) {
             maSocket = s;
       }

       void run() {
       // envoie et reception de données à l'aide des flux d'entré/sortie
       // ...
       maSocket.close();
       }
}
```

Le serveur s'occupe de l'écoute des demandes de connexion des clients, puis associe à chacun un thread de service afin qu'il puisse échanger avec lui des données:

```
class serveurMulti_client {
       public static void main(String[] args) throws Exception {
       // …
       ServerSocket socketAttente;
       socketAttente = new ServerSocket(PORT);

       do {
             //établissement d'une connexion (attente bloquante)
             Socket s = socketAttente.accept();

             // la communication est désormais possible, création du service
             Thread t = new Thread(new Service(s));
             // on démarre l'exécution concurrente du service
             t.start();
       } while (true);
       socketAttente.close();
    // …
   }
}
```

Concernant la partie client, elle sera la même que celle vue dans la communication avec un serveur mono-client précédent.

3.7 La diffusion

Dans les sections précédentes, les sockets TCP et UDP ont été utilisés pour la communication point à point. Cela signifie que le serveur répond à chaque requête arrivé d'un client. Cependant, il est parfois plus simple et même nécessaire de vouloir atteindre plusieurs destinataires au même temps, c'est ce qu'on appelle « la diffusion ». Il existe seulement **la diffusion non fiable** avec UDP car établir une connexion point à point fiable est déjà relativement coûteux, alors une diffusion fiable est très coûteuse. Il existe deux types de diffusion [1]:

- **Diffusion intégrale (Broadcasting):** la diffusion s'effectue en direction de toutes les machines d'un réseau donné. C'est le cas de la diffusion d'un message à tout le monde par exemple la sirène incendie, etc. Elle utilise pour cela la classe E des adresses IP (240.0.0.0 - 255.255.255.255).
- **Multidiffusion ou diffusion de groupe (Multicasting)**: la diffusion s'effectue à destination des applications abonnées à un groupe. Par exemple la radiodiffusion ou de la télévision sur laquelle on s'abonne en choisissant un canal. Elle utilise pour cela la classe D des adresses IP (224.0.0.0 - 239.255.255.255).

3.7.1 Diffusion intégrale (Broadcasting)

La diffusion intégrale (Broadcasting) s'effectue en envoyant un paquet à l'adresse de la « dernière » adresse possible du réseau. Il existe un alias standard pour désigner l'adresse de Broadcasting pour n'importe quel réseau 255.255.255.255 qui signifie donc (en théorie) « à toutes les machines du réseau Internet ». La diffusion intégrale est généralement limitée au réseau local et s'effectue en pratique sur l'adresse 255.255.255.255. Les processus en écoute sur le port concerné pourront recevoir le message transmis.

Voici un exemple de programmation d'un serveur (sender) qui envoie un message « bonjour » en broadcast et un client (receiver) qui reçoit ce message via son socket. Les même classes et méthodes de socket UDP sont utilisées pour l'envoie de messages:

```
import java.net.*;
public class Sender {
	public static void main(String args[]) {
	try {
		String s = "Bonjour ";
		DatagramSocket ds = new DatagramSocket();
		DatagramPacket dp = new DatagramPacket(s.getBytes(),s.getBytes().length, new
			InetSocketAddress("255.255.255.255",60123));
		ds.send(dp);
	} catch(Exception e) {
			e.printStackTrace();
		}
	}

}

import java.net.*;
public class Receiver {
	public static void main(String args[]) {
		try {
			byte [] data = new byte[256];
			DatagramSocket ds = new DatagramSocket(60123);
			DatagramPacket dp = new DatagramPacket(data,data.length);
			while (true) {
				ds.receive(dp);
				String s = new String(dp.getData(),0,dp.getLength());
				System.out.println("Received "+s);
			}
		} catch(Exception e) {
			e.printStackTrace();
			}
}
```

3.7.2 Multidiffusion (Multicasting)

Le multicasting s'effectue par l'utilisation d'un groupe qui est identifié par une adresse de classe D. Le choix de l'adresse est en pratique celui des adresses de la classe D sauf les adresses 224.x.x.x, 232.x.x.x, 233.x.x.x et 239.x.x.x qui sont en partie réservées. Lorsqu'un message est émis à destination d'un groupe, tous les membres peuvent recevoir un exemplaire de ce message [1].

Voici un exemple de la structure du programme serveur (sender) qui envoie un message « bonjour » à un groupe via les sockets. L'envoyeur de message utilise les mêmes classes et méthodes de socket UDP comme suit:

```
 try {
	String s = "Bonjour ...";
	DatagramSocket ms = new DatagramSocket();
	InetAddress ia = InetAddress.getByName("225.1.2.4");
	DatagramPacket dp = new DatagramPacket(s.getBytes(), s.getBytes().length,ia,28888);

	ms.send(dp);
	System.out.println("Sent");
} catch(Exception e) {
		e.printStackTrace();
	}
```

Afin de recevoir des messages de diffusion, le récepteur doit s'inscrire au groupe de diffusion. Pour cela, il doit créer un socket dédié par l'instanciation de la classe **MulticastSocket** qui écoute le port de diffusion. Ensuite, il doit joindre le groupe à travers son adresse IP de diffusion à travers la méthode **joinGroup(InetAddress).** Une fois le paquet est envoyé, le récepteur peut le lire en utilisant ce socket de diffusion. Ci-après la structure du programme d'un récepteur:

```
try {
	byte [] data = new byte[256];
	InetAddress ia = InetAddress.getByName("225.1.2.4");
	MulticastSocket ms = new MulticastSocket(28888);
	ms.joinGroup(ia);
	DatagramPacket dp = new DatagramPacket(data,data.length);
	while (true) {
```

```
            ms.receive(dp);
            String s = new String(dp.getData(),dp.getLength());
            System.out.println("Received "+s);
        }
} catch(Exception e) {
      e.printStackTrace();
}
```

3.8 Communication non bloquante: Java NIO

Les entrés/sorties à l'aide des classes du package java.io et java.net vus précédemment sont généralement bloquantes. Le demandeur de service reste bloqué pendant la réalisation de sa requête. Une entrée/sortie non-bloquante consiste à ne pas attendre si l'entrée/sortie se réalise et fait autre chose puis tente plus tard la communication. Le package **java.nio** (New Input Output) a été crée depuis SDK 1.4 pour permettre:

- Une gestion plus fine de la mémoire et de l'interaction avec le système de fichiers.
- Une gestion plus performante des entrées-sorties.
- L'utilisation d'entrées-sorties non bloquantes.

De nouveaux concepts dans les entrées-sorties en Java ont été proposés par ce package, notamment: les tampons mémoire (Buffers); les canaux de communication (Channels) et les sélecteurs (Selectors). Chaque classes sera décrite avec les méthodes possibles puis à la fin un exemple de client/serveur utilisant ce package sera donné.

3.8.1 La classe Buffer

Les tampons mémoire sont définis dans java.nio pour remplacer les tableaux de byte de java.io. Ils représentent des zones de mémoire contiguë, permettant de stocker une quantité de donnée fixée, d'un type primitif donné. Pour utiliser les buffers, deux classes abstraites sont proposées par java.nio:

- La classe abstraite **Buffer** qui factorise les opérations indépendantes du type primitif concerné.
- La classe abstraite **ByteBuffer** dont l'ensemble de méthodes et d'opérations est le plus complet.

Voici les méthodes utilisées pour manipuler les buffers:

- **allocate (int capacity)**: cette méthode statique est utilisée pour l'allocation de la mémoire pour le tampon.
- **get (int index)**: permet l'accès à un élément de buffer à travers son indexe.
- **int capacity** (): donne la capacité du tampon.
- **Buffer limit (int newLimit)**: fixer la taille du buffer.
- **int position** () et **Buffer position (int newPosition)**: obtenir et modifier respectivement la position courante qui est l'indice du prochain élément accessible.
- **Buffer wrap (tab[], int offset, int length)** ou **Buffer wrap(tab[])** sont des méthodes statiques utilisées pour envelopper un tableau.

3.8.2 Les canaux (Channel)

Les canaux ou **Channel** dans le package **java.nio** sont des connexions ouvertes vers des entités capables d'effectuer des opérations d'entrées-sorties comme des fichiers et des sockets. Un canal est ouvert à sa création. Il peut être utilisé jusqu'à sa fermeture. À la différence des flux de java.io, un canal peut être utilisé en **mode bloquant** et **non bloquant**. En mode non bloquant, toutes les lectures/écritures retournent immédiatement un résultat, même si rien n'est lu ou écrit. On peut alors utiliser un **sélecteur** pour avoir la possibilité d'effectuer des entrées-sorties sur différents canaux.

Les principales classes/interfaces figurent dans **java.nio.channels.***. La classe **Channel** possède principalement deux méthodes **IsOpen()** pour vérifier l'ouverture de canal et **close()** pour le fermer. La classe utilitaire **Channel** permet d'obtenir des canaux à partir de flux et des flux à partir de canaux qui sont liés entre eux (la fermeture de l'un entraîne celle de l'autre). On s'intéresse en particulier pour ce cours à l'utilisation de ces canaux pour les sockets TCP et UDP.

Canaux vers les sockets UDP

java.nio.channels.**DatagramChannel** définit un canal vers un socket UDP. On peut récupérer ce canal par la méthode **getChannel()**. Pour ouvrir ce canal, il faut utiliser la méthode **DatagramChannel.open()** qui crée et retourne un canal associé à un socket UDP (non attaché). Par défaut, ce canal est bloquant, mais on peut le modifier par l'appel de la méthode **configureBlocking (true/false)**.

Afin d'attacher le canal au socket, il faut utiliser la méthode **bind(SocketAddress)** sur le socket. Puis pour envoyer et recevoir des buffers sur le canal, les méthodes **int send(ByteBuffer src, SocketAddress target)** et **SocketAddress receive(ByteBuffer dst)** peuvent être utilisées en mode non bloquant.

Canaux vers les sockets TCP

ServerSocketChannel et SocketChannel du package java.nio.channels sont principalement proposées dans ce package.

ServerSocketChannel: c'est l'objet canal d'écoute de connexions entrantes retourné par la méthode **open()**. Il faut attacher ce ServerSocket par **bind()** avant de pouvoir accepter des connexions. Ensuite, il faut appeler **accept()** sur le canal pour accepter des connexions. Afin de spécifier si le canal soit bloquant ou non, il faut utiliser la méthode **configureBlocking(true/false)**. Le résultat de l'appel de accept est définit selon deux cas: si le canal est bloquant, elle retourne un SocketChannel quand la connexion est acceptée ou lève une IOException. Si le canal est non bloquant, elle retourne immédiatement null s'il n'y a pas de connexion.

SocketChannel: c'est l'objet canal d'écoute de connexions entrantes retourné par la méthode open() qui crée un canal de socket non connecté. Pour attacher le canal à la socket sous-jacente, il faut appeler la méthode **bind()** sur un objet Socket. Puis pour envoyer er recevoir des données sur les canaux les méthodes **read()** et **write()** sont utilisées. En mode bloquant, **write()** assure que tous les octets seront écrits mais **read()** n'assure pas que le tampon sera rempli (au moins 1 octet lu ou fin de connexion: retourne -1). En mode non bloquant, lecture comme écriture peuvent ne rien faire et retourner 0. En fin la fermeture de socket par close() entraine la fermeture du canal.

3.8.3 Les sélecteurs

Les sélecteurs sont utilisés avec les canaux en mode non bloquant. Ils héritent de la classe **SelectableChannel** afin d'offrir une meilleur gestion d'un canal**.** java.nio.channels.**Selector()** permet de créer un sélecteur qui sélectionne un certain nombre d'opérations réalisables parmi un ensemble d'opérations. Toute opération souhaitée sur le canal doit être préalablement enregistrée sur le canal par la méthode **SelectionKey register (Selector sel, int ops).** ops représente les opérations pour ce sélecteur (SelectionKey.OP_READ OP_ACCEPT OP_WRITEOP_CONNECT). SelectionKey retournée représente la clé de sélection de ce canal.

Pour utiliser un sélecteur, la méthode **select()** permet de sélectionner un ensemble de clés dont leur canaux sont prêts pour les opérations d'entrés/sorties. Cette méthode est bloquante jusqu'à la sélection d'un canal. Dès qu'une opération peut être effectuée, la méthode retourne le nombre de canaux sélectionnés. Un sélecteur maintient 3 ensembles de clés de sélection

- **key set**: ensemble des clés de tous les canaux enregistrés dans ce sélecteur.
- **selected-keyset**: ensemble des clés dont les canaux ont été identifiés comme prêts.
- **canceled-key set**: ensemble des clés qui ont été annulées mais dont les canaux n'ont pas encore été dé-enregistrés de ce sélecteur.

Les méthodes suivantes sont possibles pour la gestion des clés:

- **isAcceptable()**: vérifie si la clé du canal est prête à accepter une nouvelle connexion socket.
- **isReadable()** : vérifie si la clé du canal est prête pour la lecture.

3.8.4 Exemple de socket avec channel et selector

```
import java.nio.* ;
import java.io.IOException;
import java.util.Set;
```

```
import java.util.Iterator;
import java.net.InetSocketAddress;
public class SelectorExample {
	public static void main (String [] args) throws IOException {
		// Obtenir le selecteur
		Selector selector = Selector.open();
		System.out.println("Selector is open: " + selector.isOpen());

		// Get server socket channel and register with selector
		ServerSocketChannel serverSocket = ServerSocketChannel.open();
		InetSocketAddress hostAddress = new InetSocketAddress("localhost", 5454);
		serverSocket.bind(hostAddress);
		serverSocket.configureBlocking(false);
		int ops = serverSocket.validOps();
		// opérations possibles sur la socket: acceptation de nouvelles connexion

		SelectionKey selectKy = serverSocket.register(selector, ops, null);
		for (;;) {
			System.out.println("Waiting for select...");
			int noOfKeys = selector.select();
			// attendre qu'une opération se réalise

			System.out.println("Number of selected keys: " + noOfKeys);
			Set selectedKeys = selector.selectedKeys();
			Iterator iter = selectedKeys.iterator();
			while (iter.hasNext()) {
				SelectionKey ky = (SelectionKey) iter.next();
				if (ky.isAcceptable()) {
					// Accept the new client connection
					SocketChannel client = serverSocket.accept();
					client.configureBlocking(false);
					// Add the new connection to the selector
					client.register(selector, SelectionKey.OP READ);
					System.out.println("Accepted new connection from client: " +
					client);
				}
				else if (ky.isReadable()) {
					// Read the data from client
					SocketChannel client = (SocketChannel) ky.channel();
					ByteBuffer buffer = ByteBuffer.allocate(256);
					client.read(buffer);
					String output = new String(buffer.array()).trim();

					System.out.println("Message read from client: " + output);
					if (output.equals("Bye.")) {
						client.close();
						System.out.println("Client messages are complete
							close.");
					}
				}
				iter.remove();
			}
		}
	}
}

import java.nio.*;
import java.io.IOException;
import java.net.InetSocketAddress;
public class SocketClientExample {
	public static void main (String [] args) throws IOException, InterruptedException {
		InetSocketAddress hostAddress = new InetSocketAddress("localhost", 5454);
			SocketChannel client = SocketChannel.open(hostAddress);
		System.out.println("Client sending messages to server...");
		// Send messages to server
		String [] messages = new String [] {"Time goes fast.", "What now?", "Bye."};

		for(int i= 0; i <messages.length; i++) {
			byte [] message = new String(messages [i]).getBytes();
			ByteBuffer buffer = ByteBuffer.wrap(message);
			client.write(buffer);
			System.out.println(messages [i]);
```

```
                        buffer.clear();
                        Thread.sleep(3000);
                }
                client.close();
        }
}
```

3.9 Conclusion

Ce chapitre a présenté les principaux packages Java utilisés pour programmer une application client/serveur en utilisant des appels de la couche transport via les protocoles TCP et UDP. Les sockets TCP et UDP ont été présentés ainsi que les différentes classes et méthodes utilisées pour enrichir les parties clients et serveurs à travers l'utilisation des flux, multithread ainsi que les canaux et les sélecteurs du java.nio. Un exemple de programmation d'un serveur HTTP et un client/serveur mobile en Android sont donnés dans les Annexes A et B respectivement pour montrer le potentiel des sockets pour une variété d'applications grâce à la manipulation direct de la couche TCP/UDP. Le chapitre suivant présentera une autre interface de programmation Java de haut niveau, appelée RMI.

Chapitre 4: Java RMI (Remote Method Invocation)

4.1 Introduction

L'interface de programmation RMI (Remote Method Invocation) se focalise sur le principe orienté objet et permet l'appel distant des objets en utilisant leur méthodes. Elle se base donc sur une communication de plus haut niveau qui consiste à définir une couche offrant des services plus complexes (appelée middleware ou inter-logiciel). Cette couche est réalisée en s'appuyant sur la couche TCP/UDP qui devient cette fois-ci transparente au développeur en le comparant avec l'interface Socket [2].

4.2 Java RMI

La communication via RMI se fait via l'appel de méthodes entre les objets Java qui s'exécutent sur des machines virtuelles différentes (espaces d'adressage distincts), sur le même ordinateur ou sur des ordinateurs distants reliés par un réseau. RMI utilise directement les sockets et les échanges se font avec un protocole propriétaire: RMP (Remote Method Protocol) [1]. Les objectifs d'utilisation de cette interface de programmations sont:

- Rendre transparente la manipulation d'objets distants par le principe de délégation connue par le patron de conception « proxy ». Ce patron consiste à considérer une couche intermédiaire pour communiquer deux entités logicielles.
- Utiliser les interfaces Java pour masquer au programmeur l'appel d'objets distants et faciliter l'utilisation de ces objets.
- Préserver la sécurité via RMI Security Manager.

4.3 Architecture et principe de fonctionnement de RMI

La figure 5 illustre l'architecture d'une application client/serveur qui utilise l'interface de programmation RMI. Le serveur contient les objets distants et le client de son côté fait appel à ses objets en invoquant ses méthodes. Les principales couches de cette architecture sont :

- **Les amorces (Stub/Skeleton):** afin de communiquer un client et un serveur en RMI, il faut définir deux amorces (proxy) Stub et Skeleton. Ces amorces assurent le rôle d'adaptateurs pour le transport des appels distants sur la couche réseau. Elles réalisent également l'assemblage et le désassemblage des paramètres (sérialisation, désérialisation). Les amorces Stub et Skeleton sont créés par le générateur **rmic** appelé via une ligne de commande et chacun possède un rôle particulier [1]:
    - **Stub (côté Client) :** est le représentant local de l'objet distribué. Il initie une connexion avec la JVM distante en transmettant l'invocation à « la couche des références d'objets ». Il assemble les paramètres pour leur transfert à la JVM distante puis se met en attente des résultats de l'invocation distante. Une fois la réponse est reçu, il désassemble des valeurs et renvoie la valeur à l'appelant.
    - **Skeleton (côté Serveur) :** permet de désassembler les paramètres pour la méthode distante et fait appel à la méthode demandée. Une fois les résultats sont obtenus, il effectue l'assemblage du résultat à destination de l'appelant.
- **La couche des références d'objets:** permet d'obtenir une référence d'objet distribué à partir de la référence locale au Stub. Cette fonction est assurée grâce à un service de noms **rmiregister** qui joue le rôle de service d'annuaire pour les objets distants enregistrés. Un unique **rmiregister** est défini par JVM qui accepte des demandes de service sur le port **1099**.
- **La couche transport**: Elle connecte les deux espaces d'adressage (JVM) et gère les connexions en cours (écoute et répond aux invocations). Elle emploie un protocole **JRMP** (Java Remote Method Invocation) basé sur TCP/IP. JRMP a été modifié en supprimant la nécessité des Skeletons. Depuis la version 1.2 de Java, une même classe Skeleton générique est partagée par tous les objets distants.

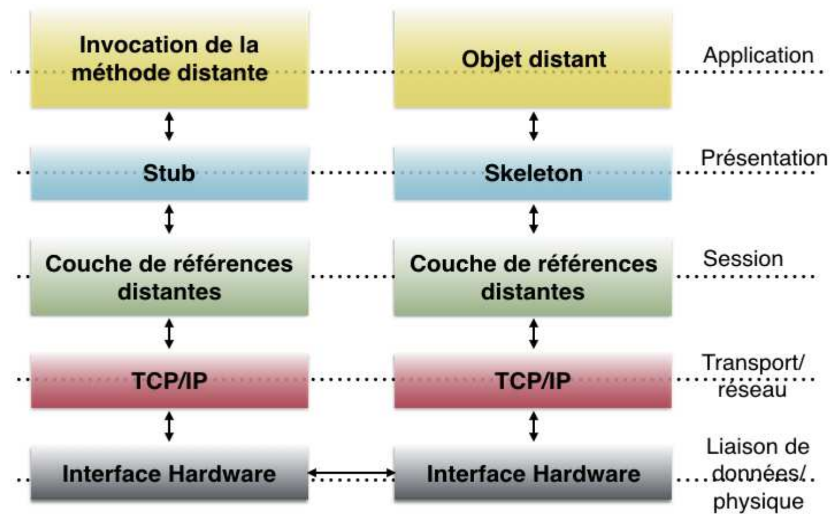


Figure 5: L'architecture d'une application client/serveur utilisant Java RMI.

4.4 Développement d'une application RMI

Afin de développer une application RMI, il faut suivre les étapes suivantes illustrées dans la Figure 6:

1. Définir une interface Java pour un objet distant.
2. Créer une classe implémentant cette interface.
3. Créer une application serveur RMI qui gère les objets distants.
4. Créer une application côté client RMI.
5. Compiler les fichiers précédents.
6. Créer les classes Stub et Skeleton (commande **rmic**).
7. Démarrer **rmiregistry** et lancer l'application serveur RMI.
8. Lancer le programme client accédant à des objets distants du serveur.

L'exemple suivant montera comment suivre ces étapes en pratique.

4.5 Exemple d'une application client/serveur en RMI

Dans cet exemple, le serveur propose des services de calcul sous forme de calculatrice distante. Les opérations à implémenter sur le serveur sont: l'addition, la soustraction, la multiplication et la division. Pour implémenter cette architecture client/serveur, il suffit de créer les classes et interfaces suivantes:

- L'interface Calculator.java
- L'objet serveur de calcul CalculatorImpl.java
- Le programme serveur CalculatorServer.java
- Le programme client CalculatorClient.java

La figure 7 illustre le récapitulatif des étapes. Voici les détails de chaque étape:

4.5.1 Etape (1). Définir une interface Java et son implémentation côté serveur pour un objet distant

Tout objet qui désire être exposé à travers les RMI doit implémenter l'interface **java.rmi.Remote**. Cette interface doit contenir des méthodes dont la signature contient une clause throws faisant apparaître l'exception **java.rmi.RemoteException**. Les paramètres ou les valeurs de retour doivent être sérialisables, i.e. implémentent l'interface **java.io.Serializable**.

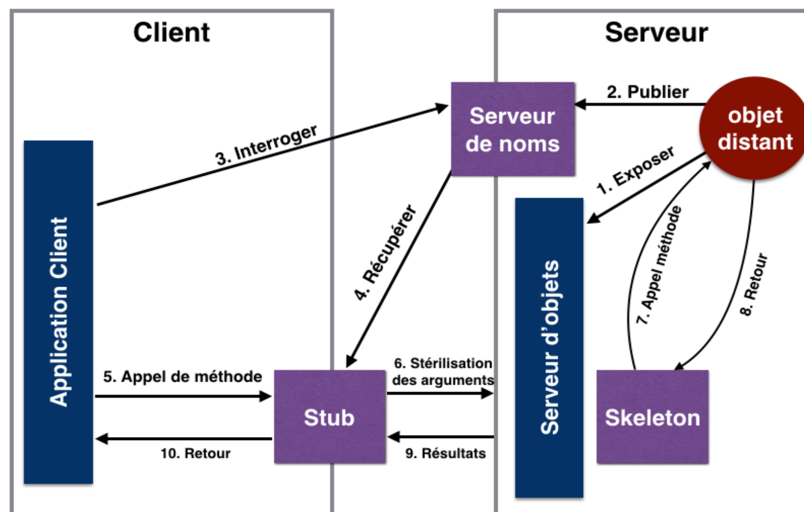


Figure 6: Processus de communication entre un client et un serveur RMI.

Voici l'interface de la classe distante: Calculator.java

```
import java.rmi.*;

public interface Calculator extends Remote {
      public long add(long a, long b) throws RemoteException;
      public long sub(long a, long b) throws RemoteException;
      public long mul(long a, long b) throws RemoteException;
      public long div(long a, long b) throws RemoteException;
   }
```

4.5.2 Etape (2). L'implémentation de l'objet distant: CalculatorImpl.java

L'implémentation de l'interface qui est une spécialisation de la classe **java.rmi.server.UnicastRemoteObject**.

```
import java.rmi.*;
import java.rmi.server.*;
public class CalculatorImpl extends UnicastRemoteObject implements Calculator {

      // il faut surcharger le constructeur vide pour déclarer l'exception RemoteException
      public CalculatorImpl() throws RemoteException {super(); }
      public long add(long a, long b) throws RemoteException{return a + b;}
      public long sub(long a, long b) throws RemoteException{return a - b;}
      public long mul(long a, long b) throws RemoteException{return a * b;}
      public long div(long a, long b) throws RemoteException{return a / b;}
 }
```

4.5.3 Etape (3). Le serveur RMI: CalculatorServer.java

L'enregistrement d'un objet peut s'effectuer à l'aide de la méthode statique: **void bind(String nom, Remote o)** de la classe java.rmi.**Naming** tel que :

- **nom** est de la forme machine:port/id (machine et port sont optionnels).
- **o** est l'objet à exposer.
- **id** c'est un nom/identifiant du service offert par l'objet à exposer.

En pratique, il faut utiliser la méthode **void rebind(String nom, Remote o)** au lieu de bind pour que l'objet soit réexposer en cas de duplication.

```
import java.rmi.Naming;
public class CalculatorServer {
      public CalculatorServer() {
              try {
                    Calculator c = new CalculatorImpl();

                    Naming.rebind("rmi://localhost:1099/CalculatorService", c);
```

```
            }
            catch (Exception e) {System.out.println("erreur: " + e); }
     }
    public static void main(String args[]) { new CalculatorServer(); }
    }
```

4.5.4 Etape (4) Implémentation du CalculatorClient.java

Pour atteindre l'objet, côté client, il suffit d'interroger le registre via la méthode statique **(Object) lookup(String nom)** de la classe Naming. Puis utiliser l'objet ordinairement pour y appeler des méthodes.

```
import  java.rmi.Naming;
public class CalculatorClient{
  public static void main(String[] args) {
       try {
             Calculator c = (Calculator) Naming.lookup( "rmi://localhost/CalculatorService");
             System.out.println("addition (4,5): "+ c.add(4, 5));
             System.out.println("multiplication (3,6):"+ c.mul(3, 6));
       } catch (MalformedURLException murle) {
                    System.out.println("java.net.MalformedURLException" + murle);
       } catch (RemoteException re) {
                    System.out.println("java.rmi.RemoteException");
      } catch (NotBoundException nbe) {
                     System.out.println("java.rmi.NotBoundException" + nbe);
       } catch (java.lang.ArithmeticException ae) {
                    System.out.println("java.lang.ArithmeticException" + ae);
       }
   }
}
```

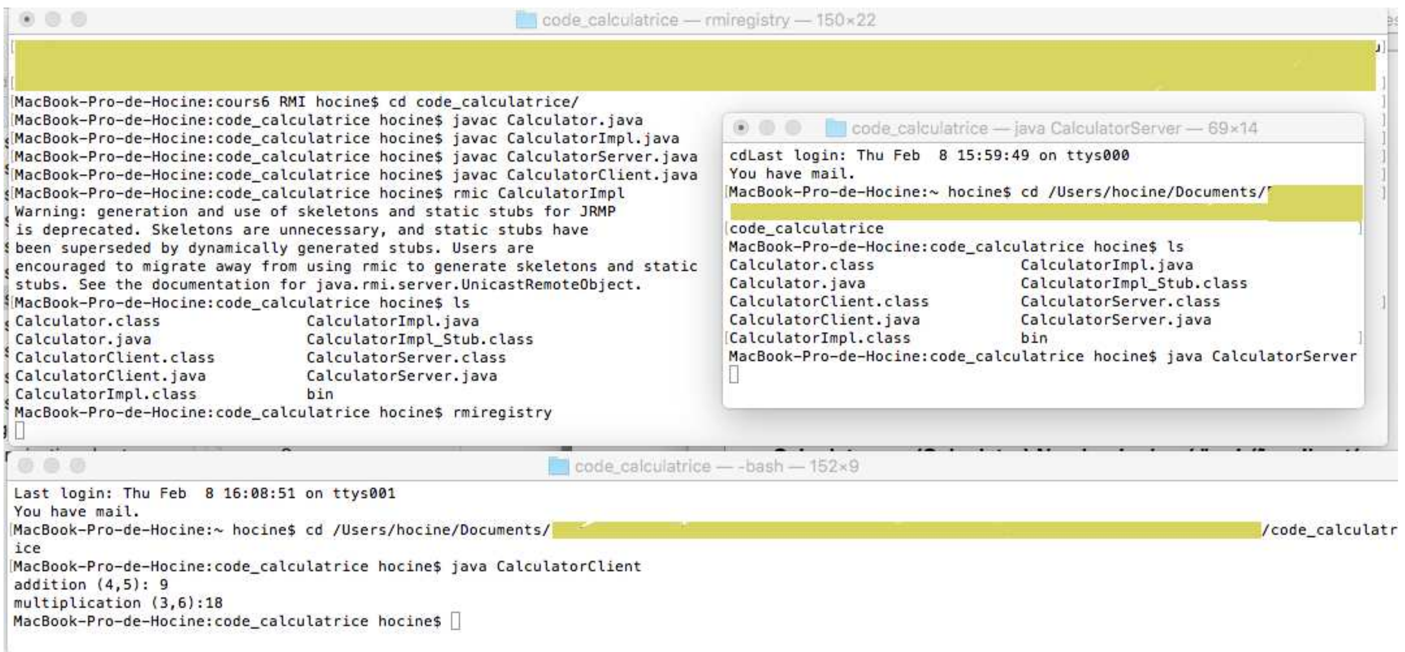


Figure 7: Récapitulatif des étapes de compilations et d'exécution de l'exemple de l'application client/serveur RMI implémentée.

4.5.5 Etape (5) Compiler les fichiers et générer les amorces

>**javac** Calculator.java
>**javac** CalculatorImpl.java
>**javac** CalculatorServer.java
>**javac** CalculatorClient.java

Puis, générer les amorces (Stub et Skeleton) à l'aide du service rmic comme suit:
>**rmic** CalculatorImpl

Ceci génère le fichier CalculatorImpl-Stub.java et CalculatorImpl-Skel.java (n'est pas généré dans v1.2 ou plus). Il compile les fichiers et supprime les fichiers sources (.java). Pour garder les sources, il suffit d'ajouter l'option keep:

>rmic keep

Cette commande conserve les fichiers sources des stubs et skeletons.

4.5.6 Etape (6) Enregistrer l'objet dans l'annuaire

Pour que l'objet soit atteignable par un client, il faut l'enregistrer dans un annuaire des objets exposés. Le service réseau correspondant à cet annuaire est fourni par défaut avec l'environnement Java, pour le démarrer :

>rmiregistry ou bien, **>rmiregistry port** qui permet de lancer l'annuaire sur un autre port que celui qui est attribué par défaut (1099).

4.5.7 Etape (7) Lancer le serveur puis le client

>java CalculatorServer
>java CalculatorClient

4.6 Sécurité

Les RMIs sont sous le couvert d'une politique de sécurité. Il existe deux façons de les paramétrer:

- Installer en interne un SecurityManager adéquat.
- Surcharger en externe la configuration des paramètres de la politique de sécurité, via la propriété java.security.policy.

Il faut lancer la JVM en spécifiant quel fichier contient les paramètres de sécurité avec la commande:

>java -Djava.security.policy = fichier ... Puis créer un fichier **security.policy** spécifiant les valeurs des paramètres :

```
grant{
      permission java.security.AllPermission;
}
```

Attention dans cet exemple toutes les permissions sont données ce qui peut exposer les objets aux attaques. Afin de contrôler l'accès, il faut donner un accès spécifiques aux utilisateurs identifiés dans le système sur quelques méthodes uniquement. Par exemple:

```
grant {
      permission java.net.SocketPermission
      "*:1024-65535","connect,accept,resolve";
      permission java.net.SocketPermission "*:80", "connect";
};
```

4.7 Conclusion

Ce chapitre a présenté une interface de programmation réseau en Java qui repose sur l'appel distant des méthodes. Elle utilise le principe d'orienté objet pour structurer les services et les appels distants. Même si cette interface semble plus facile que les Socket Java en raison de transparence de l'utilisation de la couche transport, elle se limite par la communication des applications orientées objet uniquement ce qui n'est pas supporté par tous les langages de programmation existants. Dans le chapitre suivant, on prendra l'exemple d'un des langages basés sur la programmation fonctionnelle: le langage C et on décrira ses principales interfaces de programmation réseau.

PARTIE III: PROGRAMMATION RÉSEAU EN C

Chapitre 5: Sockets TCP/UDP en C

5.1 Introduction

Le langage C est parmi les premiers langages de programmation à travers lequel la nécessité de développement des primitives de communication réseau ont vu le jour. Inspiré par le système de fichiers Unix dont le noyau est programmé en langage C, les développeurs ont vite commencé à introduire la communication entre processus et entre machine. On note en particulier l'utilisation des Socket puis l'arriver des versions de plus haut niveau comme RPC.

La notion de sockets a été introduite dans les distributions de Berkeley (UNIX), c'est la raison pour laquelle on parle parfois de sockets BSD (Berkeley Software Distribution). Comme en Java, la communication en Socket débute par la création de socket avec un identifiant unique de la connexion. Cet identifiant est passé en paramètres aux fonctions permettant d'envoyer ou de recevoir des informations à travers le socket. On distingue des primitives différentes pour la communication en TCP et en UDP.

5.2 Socket TCP

Les bibliothèques **sys/socket.h** et **arpa/inet.h** sont principalement utilisées pour la communication socket en mode connecté. A l'inverse de Java qui se focalise sur les flux, la programmation des sockets en langage C nécessite de gérer les entrés/sorties et l'encodage de données tel que l'adresse réseau. On expliquera chaque fonction proposée par ces bibliothèques et on termine cette section par un exemple d'un client et d'un serveur TCP en C.

5.2.1 Schéma de la communication en mode connecté

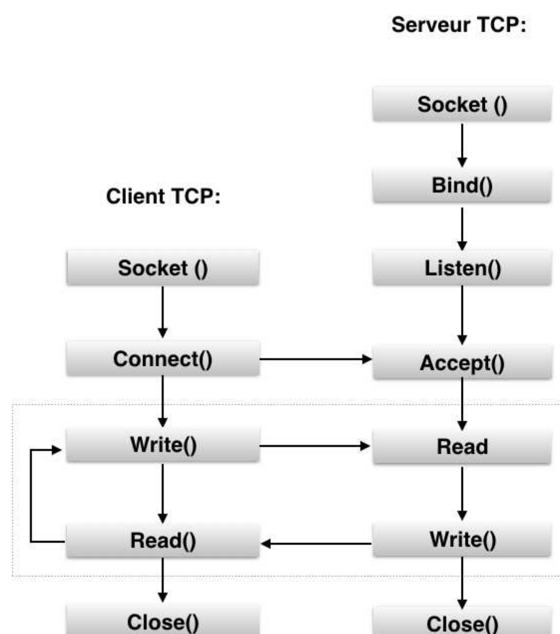


Figure 8: Schéma fonctionnel de la communication client/serveur C en TCP.

La figure 8 montre les principales fonctions utilisées dans un programme serveur et client souhaitant communiquer via les sockets BSD. Le serveur commence par créer un socket d'écoute en appelant la fonction **Socket()**. Une fois créée, il faut lier cet socket à un port et une adresse IP en utilisant la fonction **Bind()** puis le serveur se met à l'écoute à travers la fonction **Listen()**. A cette étape, le serveur est prêt à communiquer en réseau, il faut juste qu'il se met en attente de l'établissement de la connexion à travers les requêtes des clients en faisant l'appel bloquant de la fonction **Accept()**. Quand la connexion est établit avec un client, le serveur peut recevoir et envoyer des données via les fonctions **Read()** et **Write()**.

Le client de son côté, crée également un socket sur son port local afin de pouvoir communiquer en réseau. Il fait appel à la méthode **Connect()** afin d'établir la connexion avec le serveur. Une fois la connexion est acceptée, il peut envoyer sa requête à travers la fonction **Write()** et recevoir la réponse à travers la fonction **Read()**. Le client ferme ensuite la connexion en appelant la fonction **Close()**.

5.2.2 Structure de l'adresse

La programmation des sockets en C se basent sur les structures d'adresses qui sont redéfinies pour chaque famille d'adresse de réseaux. La structure utilisée pour un réseau TCP/IP est une adresse **AF_INET** qui se base sur une structure **sockaddr_in** définie dans **<netinet/in.h>** :

```
struct sockaddr_in {
      short sin_family; /* famille de protocole (AF INET) */
      u_short sin_port; /* numero de port à contacter*/
      struct in_addrsin_addr; /* adresse internet */
      char sin_zero[8]; /* initialise à zero les 8 octets*/
}
```

5.2.3 La fonction socket()

La fonction **int socket (famille, type, protocole)** renvoie un entier qui correspond à un descripteur du socket ou -1 en cas d'erreurs. Les paramètres de cette fonction sont:

- **famille**: représente la famille de protocole utilisée.
  - AF_INET ou PF_INET pour TCP/IP et UDP/IP respectivement avec une adresse sur 4 octets.
  - AF_UNIX pour les communications UNIX sur une même machine.
- **type:** indique le type de service. Par exemple:
  - SOCK_STREAM (communication par flot de données)
  - SOCK_DGRAM (communication par datagramme)
- **protocole:** permet de spécifier le protocole utilisé. Dans notre cas, on le mettra toujours à 0

Voici un exemple d'utilisation de cette fonction:

```
socket desc = socket(AF INET , SOCK STREAM , 0);
 if (socket desc == -1) {
      printf("Could not create socket");
}
```

5.2.4 La fonction bind()

La fonction **bind (socket desc, struct sockaddr*)** permet de lier un socket à une adresse réseau. La fonction bind() retourne la valeur SOCKET_ERROR en cas de problème, sinon elle retourne 0. Il existe 4 possibilités d'utiliser cette fonction :

- En spécifiant l'adresse IP et le port.
- En spécifiant l'adresse IP et en laissant le système choisir un port libre.
- En utilisant l'adresse IP : INADDR_ANY qui signifie que le socket peut-être associée à n'importe quelle adresse IP de la machine locale.
- En laissant le système choisir une adresse IP et un numéro port quelconque.
- Il est possible aussi d'utiliser la fonction **inet_addr()** pour spécifier l'adresse IP. Par exemple: inet_addr("127.0.0.1");

Voici un exemple d'utilisation de la fonction bind():

```
sockaddr_in localaddr ;
localaddr.sin family = AF INET; /* Protocole internet */

/* Toutes les adresses IP de la station */
localaddr.sin addr.s addr = htonl(INADDR ANY);
```

```
/* port d'écoute par défaut au dessus des ports réservés */
localaddr.sin port = htons(port);

if (bind(socket desc, struct sockaddr*) &localaddr, sizeof(localaddr)) = = SOCKET_ERROR) {
      /* Traitement de l'erreur; */
}
```

5.2.5 Les fonctions listen() et accept()

La fonction **int listen (int socket, int backlog)** permet de mettre un socket en attente de connexions. La fonction listen() retourne la valeur SOCKET_ERROR en cas de problème, sinon elle retourne 0. Les paramètres de cette fonction sont:

- **socket:** le socket précédemment ouvert.
- **backlog:** le nombre maximal de connexions pouvant être mises en attente.

Voici un exemple d'utilisation:

```
if (listen(socket_desc,10) = = SOCKET_ERROR) {
      // traitement de l'erreur
      }
```

La fonction **int accept (int socket, struct sockaddr * addr, int * addrlen)** permet d'établir la connexion entre le serveur et le client en acceptant un appel. La fonction accept() retourne un identificateur du socket de réponse. Si une erreur intervient elle retourne la valeur INVALID_SOCKET. Les paramètres de cette fonction sont:

- **socket:** le socket précédemment ouvert (socket local).
- **addr:** un tampon destiné à stocker l'adresse de l'appelant.
- **addrlen:** la taille de l'adresse de l'appelant.

5.2.6 La fonction connect()

Au niveau du client, la fonction **int connect (int socket, struct sockaddr * addr, int * addrlen)** est utilisée pour établir la connexion avec le serveur. La fonction connect() retourne 0 si la connexion s'est bien déroulée, sinon -1. Les paramètres de cette fonction sont:

- **socket:** le socket précédemment ouvert (socket à utiliser).
- **addr:** l'adresse de l'hôte à contacter. Pour établir une connexion, le client ne nécessite pas de faire un bind().
- **addrlen:** la taille de l'adresse de l'hôte à contacter.

5.2.7 Les fonctions read() et write ()

int read(int socket, char * buffer, int len) permet de lire dans un socket en mode connecté. La fonction read() renvoie le nombre d'octets lus. De plus cette fonction bloque le processus jusqu'à ce qu'elle reçoive des données. Les paramètres de la fonction sont:

- **socket:** le socket précédemment ouvert.
- **buffer:** un pointeur sur un tampon qui recevra les octets en provenance du client.
- **len:** le nombre d'octets lu.

int write(int socket, char * buffer, int len) permet d'écrire dans un socket (envoyer des données) en mode connecté. La fonction write() renvoie le nombre d'octets effectivement envoyés. Les paramètres de cette fonction sont:

- **socket:** le socket précédemment ouvert.
- **buffer:** l'adresse du tampon contenant les octets à envoyer au client.

- **len:** le nombre d'octets à envoyer.

5.2.8 Les fonctions send() et recv()

int recv (int socket, char * buffer, int len, int flags) permet de lire dans un socket en mode connecté. La fonction recv() renvoie le nombre d'octets lus. De plus, cette fonction bloque le processus jusqu'à ce qu'elle reçoive des données. Les paramètres de la fonction sont:

- **socket**: le socket précédemment ouvert.
- **buffer:** un tampon qui recevra les octets en provenance du client.
- **len:** le nombre d'octets à lire.
- **flags:** correspond au type de lecture à adopter (0 indique une lecture normale).

int send (int socket, char * buffer, int len, int flags) permet d'écrire dans un socket (envoyer des données) en mode connecté. La fonction send() renvoie le nombre d'octets effectivement envoyés. Les paramètres de cette fonction sont:

- **socket:** le socket précédemment ouvert.
- **buffer:** un tampon contenant les octets à envoyer au client.
- **len:** le nombre d'octets à envoyer.
- **flags:** correspond au type d'envoi à adopter (0 pour un envoi normal).

5.2.9 Autres fonctions

D'autres fonctions peuvent êtres utiles pour obtenir des informations sur la machine, telles que:
struct hostent *gethostbyaddr(&addr addr, int len, AF_INET) permettant d'obtenir la structure hostent de la machine.

La structure hostent est définie dans <netdb.h> :

```
struct hostent {
       char *h name; /* Nom de l'hôte. */
       char **h aliases; /* Liste d'alias. */
       int h addrtype; /* Type d'adresse de l'hôte. */
       int h length; /* Longueur de l'adresse. */
       char **h addr list; /* Liste d'adresses. */
}
```

struct hostent *gethostbyname(const char *name) permet également d'obtenir le nom d'hôte ou son adresse IP.

5.2.10 Exemples de client/serveur en Socket C

Le premier exemple permet de communiquer un client avec un serveur en C (comme déjà vu dans l'exemple de socket en Java). C'est un serveur d'écho, à chaque message reçu du client, il lui transmettra le même message comme retour. On présentera deux versions de serveur, un en mono-client et le second multiclients en utilisant la synchronisation des threads.

```
/*
   TCP Socket - client.c
*/
#include<stdio.h>
#include<string.h>
#include<sys/socket.h>
#include<arpa/inet.h>

int main(int argc , char *argv[]){
    int sock;
    struct sockaddr_in server;
    char message[1000] , server_reply[2000];
```

```
    //créer socket TCP
    sock = socket(AF_INET , SOCK_STREAM , 0);
    if (sock == -1) {
        printf(" impossible de créer la socket");
    }
    puts("Socket crée ");

    server.sin_addr.s_addr = inet_addr(" 127.0.0.1"); // adresse IP
    server.sin_family = AF_INET; // Pour spécifier que c'est du TCP/IP
    server.sin_port = htons( 8888 ); // port du serveur

    //se connecter au serveur
    if (connect(sock , (struct sockaddr *)&server , sizeof(server)) < 0){
        perror("échec de connect");
        return 1;
    }
    puts("Connecté… \n ");

    //communiquer infiniment avec le serveur
    while(1){
        printf("Enter un message : ");
        scanf("%s" , message);

        //Envoyer le message au serveur
        if( send(sock , message , strlen(message) , 0) < 0){
            puts("échec de Send");
            return 1;
        }

        //Recevoirla réponse du serveur
        if( recv(sock , server_reply , 2000 , 0) < 0) {
            puts("échec de recv");
            break;
        }

        puts(" Réponse du Serveur  :");
        puts(server_reply);
    }

    close(sock);
    return 0;
}
```

```
/*
TCP socket server.c
*/

#include<stdio.h>
#include<string.h>
#include<sys/socket.h>
#include<arpa/inet.h> //pour inet_addr
#include<unistd.h>    //pour write

int main(int argc , char *argv[]){
    int socket_desc , client_sock , c , read_size;
    struct sockaddr_in server , client;
    char client_message[2000];

    //créer socket
    socket_desc = socket(AF_INET , SOCK_STREAM , 0);
    if (socket_desc == -1){
        printf("erreur de création de la socket");
    }
    puts("Socket crée");
    //préparer la structure d'adresse sockaddr_in
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons( 8888 );

    //Bind
    if( bind(socket_desc,(struct sockaddr *)&server , sizeof(server)) < 0){
        perror("échec de bind");
        return 1;
```

```
    }
    puts("bind effectué");

    //Listen
    listen(socket_desc , 3); // jusqu'à trois clients en attente

    //Accepter les futures connexions
    puts(" Attente de nouvelles connexions ...");
    c = sizeof(struct sockaddr_in);

    //accepter la connexion d'un client
    client_sock = accept(socket_desc, (struct sockaddr *)&client, (socklen_t*)&c);
    if (client_sock < 0){
        perror(" échec de accept");
        return 1;
    }
    puts("Connection accepted");

    //Recevoir la requête de la part du client
    while( (read_size = recv(client_sock , client_message , 2000 , 0)) > 0 ) {
        //envoyé le message de réponse au client (echo, donc le même message)
         write(client_sock , client_message , strlen(client_message));
    }

    // traitement des erreurs
    if(read_size == 0){
        puts("Client disconnected");
        fflush(stdout);
    }
    else if(read_size == -1){
        perror("recv failed");
    }

 return 0;
}
```

Le serveur dans cet exemple est mono-client. Afin de rendre multi-clients, on utilise le même principe que la partie Java qui se base sur l'utilisation des threads. Ci-après le serveur en mode multi-clients.

```
/* server2.c */
/* Serveur d'écho qui gère des clients multiples en utilisant les threads*/
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<sys/socket.h>
#include<arpa/inet.h>
#include<unistd.h>
#include<pthread.h> //pour la création des threads

//la fonction de gestion de thread
void *connection_handler(void *);

int main(int argc , char *argv[]){
    int socket_desc , client_sock , c , *new_sock;
    struct sockaddr_in server , client;

    //Création de socket
    socket_desc = socket(AF_INET , SOCK_STREAM , 0);
    if (socket_desc == -1) {
        printf(" erreur de création de socket");
    }
    puts("Socket created");

    //Préparer la structure sockaddr_in
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons( 8888 );

    //Bind
    if( bind(socket_desc,(struct sockaddr *)&server , sizeof(server)) < 0) {
        //print the error message
        perror("bind failed. Error");
        return 1;
```

```
    }
    puts("bind effectué");

    listen(socket_desc , 3);
    puts(" serveur en attente de connexions ... »);
    c = sizeof(struct sockaddr_in);

    while( (client_sock = accept(socket_desc, (struct sockaddr *)&client, (socklen_t*)&c)) ){
        puts("Connexion acceptée");
         pthread_t sniffer_thread;
        new_sock = malloc(1);
        *new_sock = client_sock;

        /* associer un thread à un client */
       if( pthread_create( &sniffer_thread , NULL , connection_handler , (void*) new_sock) <
           0){
             perror("erreur de création de thread");
             return 1;
        }
    }

    if (client_sock < 0) {
        perror(" échec de accept");
        return 1;
    }

    return 0;
 }

/*
 * Cette fonction gère la connexion avec chaque client
 * */

void *connection_handler(void *socket_desc){

    //obtenir le descripteur de socket
    int sock = *(int*)socket_desc;
    int read_size;
    char *message , client_message[2000];

    //envoyer des messages au client
    message = "Je suis le gestionnaire de connexion … \n ";
    write(sock , message , strlen(message));

    message = "Taper un message pour avoir l’echo \n";
    write(sock , message , strlen(message));

    //Recevoir un message de la part du client
    while( (read_size = recv(sock , client_message , 2000 , 0)) > 0 ) {
        //envoyé le même message au client
         write(sock , client_message , strlen(client_message));
    }

    if(read_size == 0){
        puts("Client a déconnecté");
        fflush(stdout);
    }
    else if(read_size == -1) {
        perror("erreur de recv");
    }

    free(socket_desc);

  return 0;
}
```

Dans les exemples précédents les serveurs envoient le même message qu’ils ont reçu de la part des clients. Selon le type de serveur qu’on souhaite programmer ainsi que ses services partagées, le traitement de la requête est indispensable pour construire une réponse adéquate. Cela reste de la pure programmation et n’influence pas les fonctions de communication réseau. Il suffit de traiter chaque requête reçu de la part du client pour pouvoir répondre correctement quand le service est disponible.

Prenant un autre exemple de serveur qui cette fois-ci indique au client si le nombre envoyé par lui est positif ou négatif. Le serveur précédent peut être modifié comme suit:

```
/* server.c */
/*
TCP socket serveur traitement des nombres
*/
int main(int argc , char *argv[]){
      /* … */
    char reponse[256];
    //Recevoir la requête de la part du client
    while( (read_size = recv(client_sock , client_message , 2000 , 0)) > 0 ){
       /* … */
      int nombre=atoi(client_message);
      if(nombre<0)strcpy(reponse, " nombre négatif ! ");
      else strcpy(reponse, " nombre positif ou nul ! ");
      write(client_sock , reponse , strlen(client_message));
    }
   /* … */
  return 0;
}
```

5.3 Socket UDP

Les bibliothèques **sys/socket.h** et **arpa/inet.h** sont également utilisées pour la communication socket en mode non connecté. Avec UDP, les fonctions sont limitées à la création de sockets et à l'envoie et la réception de messages. On expliquera dans ce qui suit le principe de communication ainsi que les fonctions qui ne figurent pas dans la communication TCP. On termine cette section avec un exemple d'un client et d'un serveur UDP en C.

5.3.1 Schéma de la communication en mode non connecté

La figure 9 montre les principales fonctions utilisées dans un programme serveur et client souhaitant communiquer via les sockets UDP en C.

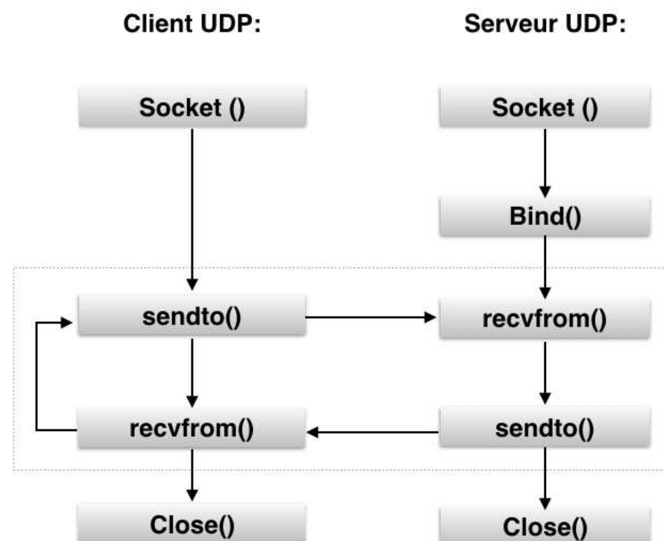


Figure 9: Schéma fonctionnel de la communication client/serveur C en UDP.

Le serveur commence par créer un socket d'écoute en appelant la fonction **Socket()**. Une fois créé, le socket doit être lié à un port et une adresse IP en utilisant la fonction **Bind()**, il sera ainsi prêt à communiquer avec les clients. Le serveur se met en attente de réception de requêtes en faisant l'appel bloquant de la fonction **rcvfrom()**. Une fois la requête reçue, le serveur peut la traiter et répondre en utilisant la fonction **sendto().**

Le client de son côté, crée également un socket sur son port local afin de pouvoir communiquer en réseau. Il peut ensuite directement envoyer sa requête à travers la fonction **sendto()** et se met en attente de réception de la réponse en appelant la fonction **rcvfrom()**. Le client ferme ensuite la connexion en appelant la fonction **Close()**.

5.3.2 La fonction socket()

C'est la même fonction socket utilisée pour la communication en mode TCP. Voici un exemple de son utilisation pour UDP:

```
int sock;
sock = socket(PF INET, SOCK DGRAM,0); // spécifie que c'est UDP
if (sock==-1) {
       perror("socket: ");
       exit(1);
}
```

Les fonctions recvfrom() et sendto()

int recvfrom (int socket, char * buf, int len, int flags, sockaddr * from, int * frlen) permet de lire des données via le socket en mode non connecté. La fonction recvfrom() renvoie le nombre d'octets lus. De plus, cette fonction bloque le processus jusqu'à ce qu'elle reçoive des données. Les paramètres de cette fonction sont:

- **socket:** socket précédemment ouvert.
- **buf:** un tampon qui recevra les octets en provenance du client.
- **len:** indique le nombre d'octets à lire.
- **flags:** le type de lecture à adopter (0 pour une lecture normale).
- **from:** l'adresse d'une structure qui contiendra l'adresse de l'émetteur.
- **frlen:** adresse d'un entier qui indique la taille de la structure de l'adresse de l'émetteur.

int sendto (int socket, char * buffer, int len, int flags, sockaddr * to, int tolen) permet d'écrire dans un socket (envoyer des données) en mode non connecté. La fonction sendto() renvoie le nombre d'octets effectivement envoyés. Les paramètres de cette fonction sont:

- **socket:** le socket précédemment ouvert.
- **buf:** un tampon contenant les octets à envoyer au client.
- **len:** le nombre d'octets à envoyer.
- **flags:** le type d'envoi à adopter (0 pour une lecture normale).
- **to:** l'adresse du destinataire.
- **tolen:** la taille de la structure de l'adresse du destinataire.

5.3.3 Exemple d'un client/serveur UDP

Voici un exemple d'une application client/serveur UDP qui utilise les fonctions décrites précédemment. Le serveur répond par le même message reçu. Si le client envoie le message « quit », le serveur ferme la connexion.

```
/* client_UDP.c */

#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <strings.h>
#include <unistd.h>

#define PORT 5678
int main(int argc,char *argv[]) {
  struct sockaddr_in addrto, addrfrom;
  int sock;
  char tampon[256];
```

```
  struct hostent *hent;

  if (argc<3) {
    fprintf(stderr,"usage: %s host message\n",argv[0]);
    exit(1);
  }
  hent = gethostbyname(argv[1]);
  if (hent==NULL) {
    fprintf(stderr,"%s: machine %s non identifiée \n",argv[0],argv[1]);
    exit(1);
  }
  sock = socket(PF_INET,SOCK_DGRAM,0); // socket UDP

  if (sock==-1) {
    perror("socket: ");
    exit(1);
  }

  addrfrom.sin_family = AF_INET;
  addrfrom.sin_port = htons(0);  // port local quelconque libre
  addrfrom.sin_addr.s_addr = htonl(INADDR_ANY); // n'importe quelle adresse

  if (bind(sock,(struct sockaddr *)(&addrfrom),sizeof(addrfrom))==-1) {
    perror("bind: ");
    close(sock);
    exit(1);
  }

  addrto.sin_family = hent->h_addrtype;
  memcpy(&(addrto.sin_addr.s_addr),hent->h_addr_list[0],hent->h_length);
  addrto.sin_port = htons(PORT);        // port destinataire
  strcpy(tampon,argv[2]);

  if (sendto(sock,tampon,256,0,(struct sockaddr *)(&addrto),sizeof(addrto))==-1) {
    perror("sendto: ");
    close(sock);
    exit(1);
  }
  if (strcmp(tampon,"quit")) {
    if (recv(sock,tampon,256,0)==-1) {
      perror("recv:");
      close(sock);
      exit(1);
    }
    printf("Recu : %s\n",tampon);
  }
  close(sock);
  return 0;
}
```

```
/* serveur_UDP.c */

#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define PORT 5678

int main(int argc,char *argv[]) {
  struct sockaddr_in addr, from;
  int sock;
  char tampon[256];
  socklen_t lg;
  int r;

  sock = socket(PF_INET,SOCK_DGRAM,0); // Protocole UDP
  if (sock==-1) {
    perror("socket: ");
```

```
    exit(1);
  }
  addr.sin_family = AF_INET;
  addr.sin_port = htons(PORT);
  addr.sin_addr.s_addr = htonl(INADDR_ANY);

  if (bind(sock,(struct sockaddr *)(&addr),sizeof(addr))==-1) {
    perror("bind: ");
    close(sock);
    exit(1);
  }
  while (1) {
    // récupérer la requête
    lg = sizeof(from);
    r = recvfrom(sock,tampon,256,0,(struct sockaddr *)(&from),&lg);
    if (r==-1) {
      perror("recv:");
      close(sock);
      exit(1);
    }

    tampon[r] = '\0';
    printf("Recu : %s depuis le port %d\n",tampon,ntohs(from.sin_port));
    if (!strcmp(tampon,"quit")) break;

   // envoyé l'echo
    if (sendto(sock,tampon,256,0,(struct sockaddr *)(&from),lg)==-1) {
      perror("sendto:");
      close(sock);
      exit(1);
    }
    printf("Envoyé : %s\n",tampon);
  }
  close(sock);
  return 0;
}
```

5.4 Les entrée/sorties non-bloquantes en C

Les appels de procédures pour la lecture sont bloquants en C. Le programme qui attend la réponse ressort de son état d'appel de lecture quand il reçoit des données ou un message d'erreur. Ceci peut causer le blocage du client qui attend la réponse d'un serveur. De même, si un client n'envoie pas les données, le serveur restera bloqué. Afin que le serveur puisse s'occuper de plusieurs clients simultanément, le socket doit être non bloquant. Cela peut s'effectuer à l'aide de l'appel système « **fcntl** » de la bibliothèque fcntl.h.

5.4.1 La fonction fcntl

La syntaxe de cette fonction définie dans le manuel d'UNIX est la suivante:

```
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

fcntl() permet de se livrer à l'une des opérations décrites plus loin sur le descripteur de fichier fd. L'opération est déterminée par la valeur de l'argument cmd. Voici un exemple pour rendre le socket non bloquant:

```
#include <fcntl.h>
/* Returns vrai en cas de success ou faux si il existe une erreur */
bool SetSocketBlockingEnabled(int fd, bool blocking){
   if (fd < 0) return false;
int flags = fcntl(fd, F_GETFL, 0);
   if (flags == -1) return false;
   flags = blocking ? (flags & ~O_NONBLOCK) : (flags | O_NONBLOCK);
   return (fcntl(fd, F_SETFL, flags) == 0) ? true : false;
}
```

5.4.2 *La fonction select*

Afin de gérer les threads sur le socket en mode non bloquant, la fonction **select** peut être utilisée. Elle permet de demander au système d'exploitation de réveiller le thread dès qu'une opération sera possible sur un descripteur. L'appel de select fournit la liste des descripteurs de fichiers sur lesquels on attend un événement. L'attente se fait sur n'importe quel type de descripteur, y compris les sockets. L'appel de select n'est pas dédié à la programmation réseau, mais en pratique très utilisé pour attendre des connexions ou des données sur les sockets. Voici la syntaxe de cette fonction:

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval
*utimeout);
```

- nfds : numéro du plus grand descripteur valide +1.
- readfds : liste de descripteurs sur lesquels on attend des données en réception.
- writefds : liste de descripteurs sur lesquels on attend de pouvoir écrire.
- exceptfds: liste de descripteurs surveillés pour conditions exceptionnelles.
- utimeout : temps maximum d'attente en mini secondes. Si cette structure est nulle, l'attente est bloquante. Le retour est immédiat si la valeur de retour est 0.

Après l'appel de cette fonction, select renvoie le nombre de descripteurs sur lesquels des opérations sont possibles. Les fd_set serontdonc modifiés. Afin de faire un autre appel par la suite, il faudra les réinitialiser.

Voici un exemple de serveur utilisant fcntl avec select pour rendre le socket non bloquant:

```
/* fonction qui permet de créer un socket serveur non bloquante */

int createServerSocket(uint16_t port) {
	/* création de la socket*/
	struct sockaddr_in a;
	int s = socket(PF_INET,SOCK_STREAM,0);
	if (s==-1) {
		fprintf(stderr,"socket problem\n");
		exit(EXIT_FAILURE);
	}
	bzero(&a,sizeof(a));
	a.sin_family = AF_INET;
	a.sin_port = htons(port);
	a.sin_addr.s_addr = htonl(INADDR_ANY);

	if (bind(s,(struct sockaddr *)&a,sizeof(a))==-1) {
		fprintf(stderr,"bind problem\n");
		close(s);
		exit(EXIT_FAILURE);
	}
	if (listen(s,0)==-1) {
		fprintf(stderr,"listen problem\n");
		close(s);
		exit(EXIT_FAILURE);
	}
/* appel de fcntl pour rendre la socket non bloquante*/
	if (fcntl(s,F_SETFL,O_NONBLOCK)<0) {
		fprintf(stderr,"problem setting non blocking state\n");
	}
	return s;
}

/* le programme principale appellant la fonction précédente et assurant la synchronisation*/

int main(int argc,char *argv[]) {
	/* … */
	/* création des sockets serveur */
	s1 = createServerSocket(61234);
	s2 = createServerSocket(61235);

	/* définir les descripteur pour gérer les appels bloquants */
```

```
        fd_set acceptSet;
        FD_ZERO(&acceptSet);
        FD_SET(s1,&acceptSet);
        FD_SET(s2,&acceptSet);

        do {
                /* sélectionner les actions prêtes à être exécutés à chaque iteration */

                memcpy(&readyToAcceptSet,&acceptSet,sizeof(fd_set));

                ready = select(MAX(s1,s2)+1,&readyToAcceptSet,NULL,NULL,NULL);

                if (ready==-1) break;

                while (ready--) {
                        int s = FD_ISSET(s1,&readyToAcceptSet)?s1:s2;
                        l = sizeof(c);

                        /* accepter une connexion non bloquante */
                        if ((d=accept(s,(struct sockaddr *)&c,&l))==-1) {
                                fprintf(stderr,"accept problem");
                                close(s);
                                exit(EXIT_FAILURE);
                        }
                        FD_CLR(s,&readyToAcceptSet);
                        if (getsockname(d,(struct sockaddr *)&local,&l)==-1) {
                                fprintf(stderr,"getsockname problem\n");
                        }
                        else
                                printf("Entered at %d",ntohs(local.sin_port));
                        do_what_you_want_or_need(d);
                }
        } while(1);
    return 0;
}
```

5.4.3 La fonction poll

Comme la fonction select, **poll** de la bibliothèque **poll.h** permet de demander au système d'exploitation de réveiller le thread dès qu'une opération sera possible sur un descripteur. Cette fonction permet de gérer une liste d'événements en utilisant la structure suivant:

```
struct pollfd{
        int fd; /* descripteur de fichier*/
        short events; /* les événements à rechercher */
        short revents; /* les événements retournés*/
}
```

L'appel de poll fournit la liste des descripteurs de fichiers sur lesquels des actions sont attendues à êtres exécutées. Voici la syntaxe de cette fonction:

```
int poll (struct pollfd fds[], nfds_t nfds, int timeout);
```

- fds: un pointeur sur un tableau des structures pollfd.
- nfds: la taille du tableau fds.
- imeout : temps maximum d'attente en mini secondes.

Les événements possibles de poll pour les sockets sont:

- POLLIN : lecture ou accept.
- POLLOUT : écriture.
- POLLPRI : lecture prioritaire (type OOB/URG).
- POLLHUP : déconnexion.
- POLLERR : erreur.

Voici un exemple d'utilisation de poll dans le programme serveur:

```
s1 = createServerSocket(61234);
s2 = createServerSocket(61235);
struct pollfd p[2];
p[0].fd = s1; p[0].events = POLLIN;
p[1].fd = s2; p[1].events = POLLIN;
int i;
socklen_t l;
struct sockaddr_in c, local;
char buffer[256];
int ready;

do {
       ready = poll(p,2,-1); // blocking on poll
       if (ready==-1) break;
       i = 0;
       while (ready--) {
              while (p[i].revents != POLLIN) i++;
              if (p[i].revents == POLLIN) do_the_job_with(p[i].fd);
       }
} while(1);
```

5.5 Conclusion

On a présenté dans ce chapitre la programmation de la communication client/serveur via les Socket en langage C. Cette interface se base sur l'utilisation de plusieurs structures de données et fonctions permettant l'accès à la couche TCP et UDP. Avec le nombre important de lignes de codes, le développement d'échange d'appels de fonctions devient une tâche complexe pour le programmeur. L'API RPC a été par conséquent proposée. Cette interface, qui sera décrite dans le chapitre suivant, permet de faciliter la programmation réseau en C tout en offrant une certaine transparence à l'utilisation de la couche transport.

Chapitre 6: RPC (Remote Procedure Call)/ XDR

6.1 Introduction

L'interface de programmation RPC (Remote procedure call) permet une communication de haut niveau par la définition d'une couche offrant des services réseau. Cette couche s'appuie sur les protocoles TCP/UDP. Alors que RMI se base sur la programmation orienté objet, RPC se base sur la programmation impérative via l'appel de procédures distantes. On présentera dans ce qui suit cette interface tout en donnant un exemple pratique de son utilisation.

6.2 RPC

L'invocation de fonctions en local nécessite le transfert des données en utilisant le même langage, la même représentation des données et le même espace d'adressage. Afin de permettre un appel de fonctions à distance, il faut prendre en compte la gestion des erreurs, l'hétérogénéité des systèmes et l'espace d'adressage. Pour répondre à ces défis, le protocole RPC a été proposé. C'est un protocole de communication interprocessus de haut-niveau qui se base sur les sockets. En effet, RPC/XDR d'Unix est un service réseau ancien (ARPANET) qui permet d'appeler une fonction distante nommée un service RPC.

Parmi les avantages de l'utilisation de RPC par rapport aux sockets sont : la simplicité de communication, la transparence des couches réseaux ainsi que la gestion de l'hétérogénéité via le standard XDR. Le but est de simplifier la programmation d'applications réparties pour ne pas se préoccuper de la localisation de la procédure. Comme RMI, RPC permet de développer des nombreuses applications client-serveur avec des services variés.

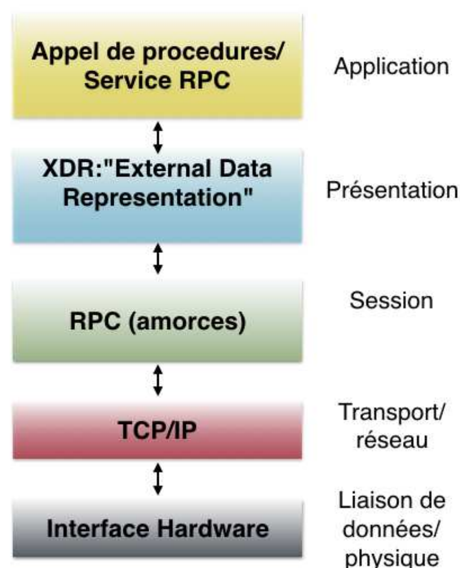


Figure 10: Architecture de RPC.

6.3 Architecture de RPC et principe de fonctionnement

Une application RPC utilise une présentation de données via XDR (eXternal Data Representation). C'est un format d'encodage des diffèrent types de données qui est indépendant d'un langage de programmation donné. XDR permet de normaliser la représentation des données échangées. Une fois les données sont codées, une couche jouant le rôle de proxy, appelée les amorces, est utilisée pour l'appel de services TCP et UDP en toute transparence par rapport au programmeur. La figure 10 illustre les différentes couches d'une application RPC.

6.3.1 Fonctionnement général d'un appel de fonction

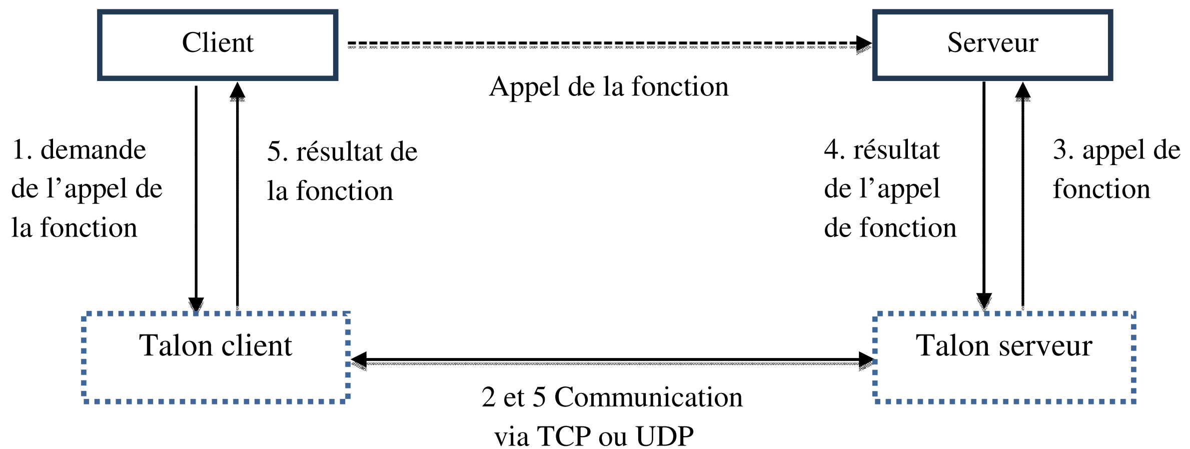


Figure 11: Fonctionnement général d'un appel de procédure en RPC.

Une communication client/serveur en RPC passe par les étapes suivantes (voir Figure 11):

- Le client s'adresse au service de nommage **portmap** de la machine distante afin de récupérer des informations sur les services proposés. Portmap est un logiciel daemon qui convertit les numéros de programmes RPC (services proposés) en numéros de port.
- Le client fait ensuite appel aux services via son proxy en utilisant des fonctions de codage/décodage de données via XDR.
- Le processus serveur est en général endormi. Il ne se réveille que pour servir la requête d'un client (via son proxy).
- Une fois la requête est traitée, les résultats sont transmis au client sous format XDR. Le time-out des RPC intervient à ce niveau. Quand il n'y a pas de réponse du serveur au bout d'un certain temps définit par le programmeur lors d'un appel RPC, le processus client redevient actif.

Le choix du protocole (TCP ou UDP) pour le serveur est laissé au programmeur. Néanmoins, UDP ne permet qu'un échange de données de taille inférieure à 8 Ko.

6.3.2 Reconnaissance d'une procédure RPC

Un serveur RPC peut avoir à gérer plusieurs procédures. Pour s'adresser à un serveur RPC, le client doit connaître: le nom de la machine (adresse IP), le numéro de port, le numéro de programme, le numéro de la procédure à exécuter, le numéro de version et le protocole à employer (TCP ou UDP). L'ensemble de ces informations est accessible à l'utilisateur via la commande : **> rpcinfo -p hostname**

6.4 Techniques de développement d'un client/serveur RPC

Il existe plusieurs façons de développer des programmes RPC, par l'utilisation des fonctions RPC de:

- La couche haute de RPC.
- La couche intermédiaire de RPC.
- La couche basse de RPC.
- RPCL : un langage de spécifications et son compilateur rpcgen.

6.4.1 La couche haute

Cette couche de RPC est totalement transparente à l'utilisateur. Elle propose néanmoins à l'utilisateur un ensemble d'autres routines tel que **rusers()** qui permet d'avoir des informations sur les utilisateurs en session sur le réseau ; **havedisk()** qui permet de savoir si une machine du réseau possède un disque dur ; etc. Les services offerts sont divers, voir **/usr/lib/librpcsvc.a** pour avoir l'ensemble de services proposés.

6.4.2 La couche intermédiaire

Dans cette couche, on a à écrire le programme serveur et le programme client et à créer des services de la couche de haut niveau. Elle regroupe peu de fonctions :

- **int registerrpc()**: permet d'initialiser le serveur en le déclarant au service de nommage et on crée l'interface réseau (socket).
- **void svc_run()**: permet, côté serveur, d'attendre un appel RPC, transmet la demande à la procédure RPC et renvoie les résultats au client s'il y en a.
- **int callrpc()**: cette routine permet, côté client, de faire des appels RPC au serveur.

En utilisant la couche intermédiaire, le programmeur est limité dans la configuration du système. Il doit également développer lui même l'encodage et le décodage de données.

6.4.3 La couche basse

Regroupe un ensemble d'une vingtaine de fonctions plus complet par rapport à la couche intermédiaire. L'utilisation de cet ensemble de fonctions est beaucoup plus complexe que la couche intermédiaire. Cette couche est utilisée généralement lorsque le protocole de communication et les délais de temporisation, qui sont fixés par défaut à 30 secondes, ne sont pas satisfaisants. Elle permet de rajouter différents paramètres tels que le développement de temporisateurs, des RPC asynchrones et des services d'authentification du client.

6.5 Mise en œuvre des RPC par rpcgen

rpcgen est un outil (compilateur) qui permet de créer très rapidement et avec peu de connaissances des programmes serveur et client RPC. Il suffit de spécifier une interface de services dans un fichier "**.x**" (par exemple "**pgr.x**") avec le langage RPC (RPCL). La compilation de ce fichier (par **rpcgen**) permet de produire :

- Les amorces côté client (**pgr_clnt.c**) et côté serveur (**pgr_svc.c**).
- Un fichier gérant les données sous format XDR (**pgr_xdr.c**).
- Un fichier header contenant les constantes (**pgr.h**).

Le programmeur ajoute deux fichiers: un fichier contenant le programme serveur qui inclut l'implémentation des procédures RPC (**serveur.c**) et un fichier qui contient le programme client appelant les fonctions (**client.c**).

Afin d'obtenir l'exécutable de l'application il faut de plus:

- Compiler **client.c** avec **pgr_clnt.c** et **pgr_xdr.c** pour obtenir le programme exécutable client.
- Compiler **serveur.c** avec **pgr_svc.c** et **pgr_xdr.c** pour obtenir le programme exécutable serveur.
- Lancer l'exécution du programme serveur puis celui du client.

6.5.1 Règles d'écriture du fichier ".x" en RPCL

Afin de spécifier l'interface de services RPC, le langage RPCL est utilisé. Il permet de définir les programmes et les fonctions qui seront partagées sur le réseau.

Un programme RPC est définit comme un ensemble de fonctions/services. Chaque programme possède un nom avec un numéro associé, en une ou plusieurs versions. Chaque version définit une liste de fonctions dont chacune possède un numéro unique. RPCL se base sur une structure de données qui similaire au langage C. Les types de données possibles en RPCL sont:

- **vide** : void.
- **entier**: int, short, long.
- **flottant**: double, float.
- **octet**: char.
- **booléen**: bool.

- **chaine de caractères**: string name<MAX>.
- **tableau statique**: int data[SIZE].
- **tableau dynamique**: int data<MAXSIZE>.
- **structure**: struct p{int a; int b};

Afin de définir les fonctions, la règle fondamentale à respecter est qu'une fonction ne prend qu'un seul paramètre. On doit donc définir une structure de données dédiée à la fonction si on veut passer plusieurs valeurs en paramètre. Voici le format général d'un fichier « .x » écrit en RPCL:

```
program NOMPROGRAM {
       version PROGVERS {
              type_retour NOMSERVICE1(type_param)=1;
              type_retour NOMSERVICE2(type_param)=2;
              ...
       }=NUMEROVERSION
       } = NUMEROPROGRAM
```

Le numéro de version et de procédure sont déterminées par le programmeur, il doit choisir des identifiants uniques pour éviter le problème d'ambiguïté lors de l'appel. Concernant le numéro du programme, il est affecté en fonction d'une plage de valeurs:

- de 00.00.00.00 à 1f.ff.ff.ff : Sun.
- de 20.00.00.00 à 3f.ff.ff.ff : Admin local.
- **de 40.00.00.00 à 5f.ff.ff.ff : Développeur.**
- de 60.00.00.00 à ff.ff.ff.ff : réservé.

Le programme serveur contient l'implémentation des fonctions définies précédemment dans le fichier.x suit la structure générale suivante:

```
#include "pgr.h"
Type_retour *fonction1(Type par,…,  svc_req *req) { /* … */ }
Type_retour *fonction2(Type par,…,  svc_req *req) { /* … */ }
```

Le client **client.c** appelle les fonctions et contient essentiellement les instructions générales suivantes:

```
#include "pgr.h"
int main() {
  CLIENT *client;
  client = clnt_create("nom_machine_serveur", NOMPROGRAM,NUMEROVERSION, "udp"); // ou tcp
  if (client == NULL) {
              perror(" erreur creation client");
              exit(1);
  }
  // appeler les fonctions
   NOMSERVICE1(…) ;
}
```

6.6 Exemple d'une application client/serveur en RPC

Dans cet exemple un programme inclut deux fonctions : la création d'un rectangle à partir des points 2D ainsi que le calcul de surface d'un rectangle. Pour cela on définit le fichier **géométrie.x** qui contient la specification de ce programme en RPCL:

```
/* geometrie.x */

struct point {
       int x;
       int y;
};
```

```
struct rectangle {
       struct point p1;
       struct point p2;
};
struct coordonnees {
       int x1;
       int x2;
       int y1;
       int y2;
};

program GEOM_PROG {
       version GEOM_VERSION_1 {
              int SURFACE_RECTANGLE(rectangle) = 1;
              rectangle CREER_RECTANGLE(coordonnees) = 2;
       } = 1;
} = 0x20000001;
```

Ensuite, pour obtenir les amorces et le fichier XDR de l'encodage de donner, il faut utiliser le compilateur rpcgen avec la ligne de commande suivante: **>rpcgen** geometrie.x.

Le résultat de cette commande produira quatre fichiers:

- **geometrie.h** : qui contient la définition de toutes les structures et des signatures des opérations.
- **geometrie_xdr.c** : qui contient les fonctions de codage XDR des structures de données
- **geometrie_clnt.c** et **geometrie_svc.c**: qui sont les amorces côtés client et serveur. Dans la dernière version, un seul fichier est généré pour les amorces qui s'utilise côté client et côté serveur.

L'étape suivante consiste à créer un nouveau fichier **serveur.c** qui contient l'implémentation des fonctions définies précédemment:

```
#include "geometrie.h"
int *surface_rectangle_1_svc(rectangle *rect, svc_req *req) {
       static int result;
       result = (rect -> p1.x - rect -> p2.x) *(rect -> p1.y - rect -> p2.y);
       return &result;
}
rectangle creer_rectangle_1_svc(coordonnees *coord, svc_req *req) {
       static rectangle rect;
       rect.p1.x = coord -$>$ x1;
       rect.p1.y = coord -$>$ y1;
       rect.p2.x = coord -$>$ x2;
       rect.p2.y = coord -$>$ y2;
       return &rect;
       }
```

Ensuite, il faut créer un nouveau fichier **client.c** qui appelle les fonctions:

```
#include "geometrie.h"
int main() {
       CLIENT *client;
       rectangle *rect;
       coordonnees coord;
       client = clnt_create("localhost", GEOM_PROG,GEOM_VERSION_1, "udp");
       if (client == NULL) {
              perror(" erreur creation client");
              exit(1);
         }
       coord.x1=12;
       coord.x2=20;
       coord.y1=10;
       coord.y2=15;
       rect = creer_rectangle_1(&coord, client);
       printf(" rectangle cree ");
}
```

Compilation et exécution

Il faut compiler les partie serveur et client comme suit:

```
>gcc -c geometrie_xdr.c
```

```
>gcc –c geometrie_clnt.c
>gcc –c client.c
>gcc –o client client.o geometrie_clnt.o geometrie_xdr.o
>gcc –c geometrie_svc.c
>gcc –c serveur.c
>gcc –o serveur geometrie_svc.o serveur.o geometrie_xdr.o
```

En suite, exécuter le serveur et le client :

```
>./serveur&
>./client
```

6.7 Conclusion

Ce chapitre a présenté l'interface de programmation RPC. Même si cette interface offre une transparence par rapport à l'appel des services de la couche transport, elle ne reste néanmoins pas suffisamment pratique pour développer des applications complexes contenant plusieurs programmes et services. La gestion de ces derniers reste une tâche coûteuse en termes de développement. Une raison pour laquelle, en C les sockets restent l'option la plus favorisée en pratique par les développeurs.

SERIES D’EXERCICES

Série d’exercices I

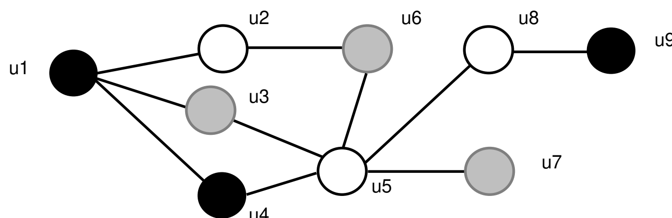
Objectifs

- *Rappel sur les flux d’entrée/sortie Java.*
- *Programmer une application qui gère la synchronisation des threads en Java.*
- *Comprendre le concept de sérialisation et son implémentation en Java.*

Exercice 1: flux d’entrée/sortie Java

On souhaite implémenter un système de gestion de posts sur un réseau social particulier. Chaque utilisateur de ce réseau est caractérisé par son nom, prénom, email et un numéro de téléphone. Un utilisateur a pour mission d’effectuer des « posts » sur son propre compte qu’il le partage automatiquement avec les utilisateurs avec qui il est connecté (liaison directe uniquement). Les utilisateurs peuvent avoir aussi trois types de badges à choisir dès la création de leurs comptes: blanc, gris et noir. Le badge blanc indique que les posts sont uniquement de type commentaire textuel; le badge gris signifie que les posts sont des hash tags (uniquement de type #text) tant dis que le badge noir signifie que les posts sont des numéros représentants des codes de produits publicitaires.

1. Proposer une implémentation en Java pour représenter les utilisateurs de ce réseau. Pour cette première partie, ne considérez pas les relations entre les utilisateurs ni le partage de posts. Pour cela vous allez:
   a. Créer une classe Utilisateur qui permet de décrire ses caractéristiques, afficher ses détails et lui permettre de rajouter des posts sur son compte. Les posts doivent êtres enregistrés dans un fichier.
   b. Créer une classe contenant la méthode main () qui permet de créer un ensemble d’utilisateurs, de rajouter des posts dans son compte puis afficher la liste des utilisateurs avec leurs posts respectifs.
2. Dans cette seconde partie, nous allons représenter les relations entre les utilisateurs via un graphe (ci-après une illustration des relations possibles entre des utilisateurs u_i):
   a. Proposer une implémentation de ce réseau.
   b. Modifier votre classe main pour afficher l’ensemble des utilisateurs avec leurs connexions.



Exercice 2 : multithread, sérialisation

Une société de relais de colis à l'aéroport se spécialise dans la redistribution internationale des colis venants de différents fournisseurs. Elle stocke temporairement les colis dans son local à l'aéroport. Ce local a une capacité limité de 10 colis uniquement. Les équipes partenaires de cette société peuvent récupérer les colis et les distribuer. Un colis est caractérisé par un identifiant unique qui inclut son code de destination, la date d'expédition ainsi que le nom et l'adresse du récepteur.

1. Proposer une implémentation en Java multithread pour représenter la gestion de cette société. Utiliser la synchronisation des threads pour éviter les incohérences.

2. On souhaite sauvegarder nos objets Colis pour garder une trace. Modifier cette classe pour qu'elle produise uniquement des objets sérialisés. Modifier aussi votre méthode main pour créer un fichier « poster.bin » contenant vos objets auparavant crées.

Série d’exercices II

Objectifs

- *Implémenter des applications client/serveur en Java communicant à l’aide de l’interface de programmation Socket en TCP et en UDP.*
- *Transmettre des objets en réseau à l’aide de la sérialisation en Java*
- *Implémenter un serveur multi-clients à l’aide des Threads Java*
- *Implémenter des applications client/serveur qui utilisent l’interface de programmation RMI.*

Rappel sur les Sockets UDP

Les sockets sont inspirés de la communication interprocessus dans le système UNIX qui considère que n’importe quel échange peut s’effectuer à travers un tampon mémoire qui se manipule par la gestion des entrées/sorties. Afin de communiquer deux applications distantes à travers les sockets, le client doit en particulier connaitre le point de communication du serveur, c’est à dire l’adresse IP et le port, et c’est à lui d’initier la connexion ou la communication. Java intègre nativement les fonctionnalités de communication réseau via le package **java.net.**

Les sockets UDP sont basés sur un mode de communication par paquets de données. La communication ne nécessite pas l’établissement de la connexion entre le client et le serveur. Elle s’initie uniquement par la réception d’un message du client par le serveur. Les étapes de la communication entre le client (clientUdp.java) et le serveur (serveurUdp.java) sont:

1. **Serveur**: crée un socket à l’aide de l’instanciation de la classe **DatagramSocket** et le lie à un port.
2. **Serveur**: se met en attente de réception de requêtes en appelant la méthode bloquante **receive (DatagramPacket**). Il faut donc avant préparer un paquet **DatagramPacket** vide à mettre comme paramètre de sortie de cette méthode.
3. **Client**: crée également un socket à l’aide de l’instanciation de la classe **DatagramSocket** pour accéder à la couche UDP et le lie à un port quelconque. Il pourra ainsi communiquer avec le serveur.
4. **Client**: prépare un paquet de données **DatagramPacket** contenant sa requête puis l’envoie via son socket au serveur en précisant l'adresse IP et port de ce dernier. Il utilise pour cela la méthode **send(DatagramPacket).** Il se mettra ensuite en attente bloquante pour recevoir la réponse en appelant la méthode **receive (DatagramPacket).**
5. **Serveur** : le paquet sera reçu par le serveur. L'adresse du client est précisée dans le paquet, le serveur peut alors lui répondre en appelant la méthode **send (DatagramPacket)** qui contient le paquet de réponse.
6. **Client** : le client reçoit le paquet de réponse et récupère ses données.

Exercice 1 : client/serveur UDP

On souhaite implémenter une application client/serveur en Java qui gère l’authentification des clients à l’aide des sockets UDP.

1. Ecrire un serveur d’authentification mono-client. A chaque lancement, le serveur demande à l’administrateur (un utilisateur local utilisant la console) de saisir une suite de login et mot de passe (des chaines de caractères) qui représentent les infirmations des clients autorisés à accéder au serveur. Une fois la liste des clients est donnée par l’administrateur, le serveur peut par la suite accepter les demandes entrantes de connexions de clients. Lors d’une tentative de connexion, le serveur vérifie de son côté les informations du client et lui répond par:
    a. un message de confirmation de connexion quand les informations sont correctes (c.a.d. le client se trouve bien dans la liste donnée par l’administrateur)
    b. ou bien un message d’erreur si l’une des informations ou les deux (login et mot de passe) ne sont pas correctes.

c. ou bien un message de refus de connexion après trois mauvaises tentatives de connexion d'un client. Le serveur dans ce cas affiche aussi de son côté sa liste noire modifiée. Cette liste contient les clients qui ont fait trois mauvaises tentatives de connexion successives.

2. Ecrire un programme client qui tente de s'authentifier au prêt du serveur précédent en donnant un login et un mot de passe. Le client essaie de s'authentifier d'une manière continue au serveur et arrête cette demande quand il obtient un message de confirmation de connexion. En fin, un client peut uniquement s'authentifier et il ne peut pas s'inscrire. L'inscription se fait seulement par l'administrateur au niveau du serveur.

Rappel sur les Sockets TCP

La communication en TCP nécessite une phase de connexion explicite entre le client et le serveur avant l'échange de messages. De plus, les données échangées ne sont pas des tableaux d'octets mais des flux Java. Chaque socket possède un **flux d'entrée** et un **flux de sortie.** Cette communication permet d'envoyer facilement n'importe quel objet ou donnée. Voici les principales étapes de la communication:

1. **Serveur**: crée un socket d'écoute à l'aide de l'instanciation de la classe **ServerSocket** et le lie à un port ainsi qu'un socket de service **Socket** afin de pouvoir accepter par la suite les requêtes des clients. Il fait un appel de la méthode **accept** qui permet de le mettre en attente bloquante d'une requête de la part d'un client.
2. **Client**: crée un socket à l'aide de l'instanciation de la classe **Socket** pour accéder à la couche TCP et le lie à un port quelconque.
3. **Client**: prépare un message à envoyer à travers le flux de sortie **OutputStream**. Il se mettra ensuite en attente pour recevoir la réponse en utilisant son flux d'entré **InputStream.**
4. **Serveur**: une fois accept () retournera un résultat, cela veut dire une connexion s'est établit avec un client. Dans ce cas, le serveur récupère le message reçu à travers son flux d'entré **getInputStream.** Quand la requête est analysée, le serveur pourra répondre au client en envoyant un message via son flux de sortie **getOutputStream**.
5. Le client peut ensuite fermer la connexion quand la communication se termine. Le serveur aussi ferme la connexion et il se remet en attente de nouveau clients.

Pour rendre le serveur multi-clients, le principe est simple : au lieu d'associer au processus serveur directement le socket de service pour communiquer avec un client, il faut déléguer cette tâche à un thread. Cela va permettre le principal thread du serveur de s'occuper du reste des clients. A chaque client arrivé, un socket de service (respectivement un thread) est crée. Au lieu d'avoir une seule classe pour le serveur, en créa deux classes: une principale serveur.java et la seconde est le service qui hérite de la classe Thread et qui est responsable de la communication avec le client.

Exercice 2. client/serveur TCP

Ecrire un programme qui permet une communication client/serveur en utilisant le protocole TCP via l'interface de programmation Socket. Le serveur reçoit les commandes suivantes du client :

- HEL: dans ce cas le serveur répond au client par le message « hello ».
- DAT: dans ce cas le serveur répond par la date du jour.
- ADD a b: dans ce cas le serveur répond au client par le résultat de « a+b », tel que a et b sont deux entiers.
- MIN a b : dans ce cas le serveur répond par le résultat de « a-b », tel que a et b sont deux entiers.
- MUL a b: dans ce cas le serveur répond par le résultat de « a*b », tel que a et b sont deux entiers.

Dans le cas où le serveur reçoit un message non compréhensible, il répond par même message envoyé suivie du message : "demande à reformuler ".

Exercice 3. transmission des objets et le concept de sérialisation

On souhaite programmer un serveur TCP qui calcule la distance entre les points 2D envoyés par les clients. Pour cela nous avons besoin d'implémenter trois classes:

1. La classe « Point » permettant la création d’un point à partir de ses coordonnées x et y, d’afficher le point et calculer sa distance avec un autre point. Voici un rappel sur le calcul de distance en deux dimensions :

```
int dx = Math.abs(x - point2.getX());
int dy = Math.abs(y - point2.getY());
double distance=Math.sqrt((dx * dx) + (dy * dy));
```

2. La classe « Client» qui permet de saisir les coordonnées d’un point 2D puis l’envoyer au serveur (en objet sérialisé).

3. La classe « Serveur » qui permet de recevoir des objets points 2D et les stocker dans une liste. A chaque réception d’un objet « Point », le serveur doit afficher ce point ainsi que sa distance avec le premier élément de la liste. On suppose que la distance est 0 quand le serveur possède uniquement un seul point.

Exercice 4. client/serveur TCP multi-threads

On veut créer un serveur de date qui gère plusieurs connexions clientes à la fois sans bloquer tous les interlocuteurs. Le serveur permet de répondre à chaque client par la date du jour quand ce dernier envoie le message ”date”. Dans le cas ou le serveur reçoit un autre message, il lui répond par le message : ” demande à reformuler ”. Quand un client répète le même message trois fois sans succès, le serveur ferme la connexion et affiche ”tentative de piratage” de son côté.

Rappel sur RMI

La communication via RMI se fait via l'appel de méthodes entre objets Java s'exécutant sur des machines virtuelles. Le serveur contient les objets distants et le client de son côté fait appel à ses objets en invoquant ses méthodes. RMI génère des amorces qui jouent le rôle de proxy permettant de communiquer le client avec le serveur. RMI utilise également un service de noms **rmiregister** qui joue le rôle de service d'annuaire pour les objets distants enregistrés. Un unique **rmiregister** est défini par JVM et qui accepte des demandes de services sur le port **1099** par défaut. Afin de développer une application RMI, il faut:

1. **Définir une interface Java** (par exemple MonInterface.java) **et son implémentation** (par exemple MonInterfaceImpl.java) au niveau du serveur pour un objet distant. Tout objet qui désire être exposé à travers les RMI doit implémenter l'interface **java.rmi.Remote**. Cette interface doit contenir des méthodes dont la signature contient une clause **throws** faisant apparaître l'exception **java.rmi.RemoteException**. L'implémentation de l'interface est une spécialisation de la classe **java.rmi.server.UnicastRemoteObject.**
2. **Créer une classe serveur RMI** (par exemple serveur.java) qui permet l'enregistrement d'un objet à l'aide de la méthode statique **bind ()** de la classe java.rmi.**Naming.**
3. **Créer une classe client RMI** (par exemple client.java): le client doit interroger le registre via la méthode statique **(Object) lookup(String nom)** de la classe **Naming** pour atteindre l'objet. Puis, on peut utiliser l'objet ordinairement pour y appeler des méthodes.

Exercice 5 : client/serveur RMI

Ecrire un programme en mode client/serveur qui utilise l’interface de programmation RMI pour communiquer un objet distant « Service ». Cet objet possède deux méthodes distantes:

1. **puissance (int a, int b)** qui retourne a puissance b.
2. **nbChiffres (int c)** qui retourne le nombre de chiffres de l’entier c. Par exemple si c=34234 le nombre de chiffres est 5.

Série d’exercices III

Objectifs

- *Implémenter des applications client/serveur en langage C communicant à l’aide de l’interface de programmation Socket BSD en TCP et en UDP.*
- *Implémenter un serveur multi-clients en C.*
- *Implémenter des applications client/serveur en langage C communicant à l’aide de l’interface de programmation RPC.*

Rappel sur les Sockets TCP en C

Les bibliothèques **sys/socket.h** et **arpa/inet.h** sont principalement utilisées pour la communication socket en mode connecté. Voici les étapes de communication client/serveur :

Le serveur commence par créer un socket d’écoute en appelant la fonction **Socket ()**. Une fois créé, il faut lier le socket à un port et une adresse IP en utilisant la fonction **bind ().** Puis, le serveur se met à l’écoute à travers la fonction **listen ()**. A cette étape, le serveur est prêt à communiquer en réseau, il faut juste qu’il se met en attente de l’établissement de la connexion à travers les requêtes des clients en faisant l’appel bloquant de la fonction **accept ()**. Quand la connexion est établit avec un client, le serveur peut recevoir et envoyer des données via les fonctions **read ()** et **write()**.

Le client de son côté, crée également un socket sur son port local afin de pouvoir communiquer en réseau. Il fait appel à la méthode **connect ()** afin d’établir la connexion avec le serveur. Une fois la connexion est acceptée, il peut envoyer sa requête à travers la fonction **write ()** et recevoir la réponse à travers la fonction **read()**. Le client ferme ensuite la connexion en appelant la fonction **close ()**.

Exercice 1. sockets TCP C

Ecrire un programme client/serveur en langage C qui utilise l’interface de programmation Socket en mode connecté. Selon le message et le nombre n donné par le client, le serveur répond par :

- la factorielle de n pour le message ”fac ”.
- la racine carrée de n pour le message ”rac ”.
- n puissance 2 pour le message ”car ”.

Le serveur répond par un message d’erreur en cas de nombre donné inapproprié. Il affiche sinon le port et le résultat du calcul.

Exercice 2. sockets TCP C- multi-clients

Modifier le code de l’exercice précédent. Cette fois-ci le serveur doit pouvoir répondre à plusieurs clients simultanément.

Rappel sur les Sockets UDP en C

Les bibliothèques **sys/socket.h** et **arpa/inet.h** sont également utilisées pour la communication socket en mode non connecté. Voici les étapes de communication client/serveur :

Le serveur commence par créer un socket d’écoute en appelant la fonction **socket ()**. Une fois créé, il faut lier le socket à un port et une adresse IP en utilisant la fonction **bind ()**, il sera ainsi prêt à communiquer avec les clients. Le serveur se met en attente de reception de requêtes en faisant l’appel bloquant de la fonction **rcvfrom ()**. Une fois la requête reçue, le serveur peut la traiter et répondre en utilisant la fonction **sendto ().**

Le client de son côté crée également un socket sur son port local afin de pouvoir communiquer en réseau. Il peut ensuite directement envoyer sa requête à travers la fonction **sendto ()** et se met en attente de

reception de la réponse en appelant la fonction **rcvfrom ()**. Le client ferme ensuite la connexion en appelant la fonction **close ()** qui ferme le socket.

Exercice 3. sockets UDP C

Ecrire un programme client/serveur en C qui utilise l'interface de programmation Socket. La communication se fait via le protocole UDP. Le serveur permet de calculer la factorielle d'un nombre envoyé par le client. Voci quelques particularités à traiter :

- Le serveur répond par un message d'erreur quand le message envoyé par le client est un nombre négatif.
- Le serveur affiche le port et le résultat du calcul quand le nombre envoyé par le client est positif.

Rappel sur RPC

RPC se base sur la programmation impérative via l'appel de procédures distantes. Une application client/serveur RPC utilise une présentation de données via XDR. Lors de la communication client/serveur en RPC, le client s'adresse au service de nommage **portmap** de la machine distante afin de récupérer des informations sur les services proposés. Portmap est un logiciel daemon qui convertit les numéros de programmes RPC (services proposés) en numéros de port. Le client fait ensuite appel aux services via son proxy (pour assurer la transparence) et en utilisant des fonctions de codage/décodage de données via XDR. Pour s'adresser à un serveur RPC, le client doit connaître : le nom de la machine (adresse IP) ; le numéro de port ; le numéro de programme ; le numéro de la procédure à exécuter ; le numéro de la version; le type de protocole à employer (TCP ou UDP).

Afin de mettre en œuvre une application via RPC, **rpcgen** peut être utilisé. C'est un outil (compilateur) qui permet de créer très rapidement et avec peu de connaissances des programmes serveur et client RPC. Il suffit de spécifier une interface de services dans un fichier "**.x**" par exemple "**pgr.x**" avec le langage RPC (RPCL). Voici le format général d'un fichier « .x » écrit en RPCL:

```
program NOMPROGRAM {
       version PROGVERS {
              type_retour NOMSERVICE1(type_param)=1;
              type_retour NOMSERVICE2(type_param)=2;
              ...
       }=NUMEROVERSION
  } = NUMEROPROGRAM
```

Le numéro de la version et de la procédure sont déterminées par le programmeur. Leurs identifiants doivent êtres uniques pour éviter le problème d'ambiguïté lors de l'appel. Concernant le numéro du programme, sa valeur doit être entre **40.00.00.00** à **5f.ff.ff.ff.**

Le serveur qui contient l'implémentation des fonctions définies précédemment dans le fichier.x suit la structure générale suivante:

```
#include "pgr.h"
Type_retour *fonction1(Type par,…,  svc_req *req) { /* … */ }
Type_retour *fonction2(Type par,…,  svc_req *req) { /* … */ }
```

Le client **client.c** appelle les fonctions et contient essentiellement les instructions générales suivantes:

```
#include "pgr.h"
int main() {
  CLIENT *client;

  client = clnt_create("nom_machine_serveur", NOMPROGRAM,NUMEROVERSION, "udp"); // ou tcp
```

```
  if (client == NULL) {
              perror(" erreur creation client");
              exit(1);
  }
  // appeler les fonctions
   NOMSERVICE1(…) ;
}
```

Exercice 4. RPC

Ecrire un programme en mode client/serveur qui utilise l’interface de programmation RPC.

1. Le serveur propose l’appel de deux procédures:
    a. **carre** qui retourne le carré d’un nombre entier passé en paramètre.
    b. **cube** qui retourne le cube d’un membre entier passé en paramètre.
2. Ecrire une interface en RPCL qui permet de définir un programme proposant deux procédures :
    a. **estValide** qui permet de vérifier si la date donnée est valide. On suppose que la date est représentée uniquement par le nombre d’haures, de minutes et de secondes.
    b. **difference** qui calcule la différence entre deux dates passées en paramètres.

CONCLUSION

Les systèmes répartis sont indispensables pour répondre à des aspects économiques des institutions et des entreprises. Ils permettent de proposer des services et des applications accessibles via un réseau aidant ainsi à réduire les coûts, à créer des systèmes évolutifs et à augmenter la performance de calcul et de stockage de donnés. Différentes méthodes et outils ont été proposés pour simplifier le développement des applications réparties, en particulier les applications client/serveur. Ce cours a présenté les principales interfaces de programmation et middleware utilisés pour créer une application répartie en utilisant les Socket, RPC et RMI.

D'autres interfaces et outils permettent aussi de communiquer les applications réparties en réseau qui n'ont pas pu êtres entamées dans ce cours et qui peuvent faire l'objet de recherche. C'est le cas par exemple des aspects de programmation relatifs à certain type d'applications comme la programmation des applications CORBA de l'OMG, la programmation des Applets en Java, la programmation Web (par exemple les Applet en J2EE), la programmation des agents mobiles (par exemple en utilisant la plateforme multi-agent JADE de Java) et Internet des objets avec SOAP.

En fin, le développement de nouveaux middlewares reste un défi pour non seulement de communiquer les applications en réseau, mais également assurer la facilité de sa mise en ouvre, la sécurité des échanges, et la prise en charge d'hétérogénéité des systèmes, des langages et des données échangées. Ces recherches sont néanmoins contraintes par différents aspects tels que le type d'applications et le respect des standards tels que W3C et OMG.

REFERENCES

1. Elliotte Rusty Harold « Programmation réseau avec Java », 2e édition O'REILLY – 21 mars 2001.

2. Annick Fron, Architectures réparties en Java, Paris : Dunod, 2012 (disponible à la bibliothèque universitaire UMAB)

3. Matthew Neil, Stones Rchard, Programmation Linux, Amazon, 2002 (disponible à la bibliothèque universitaire UMAB)

4. Frédéric Jacquenod, Administration des réseaux, Paris : CampusPress, 2002 (disponible à la bibliothèque universitaire UMAB)

5. De Tony Bautts, Terry Dawson et Gregor N. Purdy. Administration réseau sous Linux, infrastructure, services et sécurité. Traduit par Guillaume Allègre, Eric Barons et François Cerbelle, 1995

6. Maximiliano Firtman. Développer pour le Web mobile. Pearson 22 avril 2011

ANNEXES

Annexe A : Implémentation d’un mini serveur HTTP simplifié

Cette section donne un exemple d’implémentation d’un client et d’un serveur en utilisant les sockets Java. Il s’agit de développer un mini serveur HTTP qui reçoit des requêtes d’un client Web et envoie la réponse. Le client interagit avec ce serveur via le navigateur Web. On rappellera dans ce qui suit le principe du protocole HTTP, le format de requêtes et de réponses ainsi que les détails de leur implémentation à travers un exemple.

7.1 Le protocole HTTP

HTTP (Hyper Text Transfer Protocol) est un protocole conçu pour le World Wide Web, à l'origine constitué de pages statiques liées entre-elles par des liens hypertextes. Le client envoie une requête au serveur (port par défaut: 80 ou 443 en sécurisé). Le protocole HTTP est dit « sans état » car chaque requête est indépendante des autres. La communication en HTTP se focalise sur une architecture client/serveur. Le serveur possède des ressources que le client demande en formulant des requêtes sous formes d’URL.

Il existe deux types de ressources Web: Les ressources statiques comme par exemple les pages HTML, images, son et vidéos et les ressources dynamiques qui dépendent de la partie qui interagit :

- Côté client: applet, Javascript/JQuery, Plugin, ActiveX, ...
- Côté serveur : CGI, servlets/JSP, scripts serveur (php), …

7.2 Requête HTTP

Une requête HTTP doit suivre le format spécifique suivant:

- 1ère ligne : METHODE /chemin/vers/la/ressource HTTP/version .
- Les lignes suivantes d'en-têtes avec des paires de « clé: valeur ».
- Une ligne vide de séparation en-tête/corps après.
- En suite le corps (facultatif) de la requête.

Les méthodes possibles de HTTP sont:

- OPTIONS : pour obtenir les capacités du serveur HTTP.
- GET: pour récupérer une ressource.
- HEAD: pour obtenir les en-têtes d'une ressource, sans le corps.
- POST: pour envoyer des données vers une ressource.
- PUT : pour enregistrer une ressource sur le serveur.
- DELETE: pour supprimer une ressource du serveur.
- TRACE : requête de diagnostic pour avoir la trace.
- CONNECT : pour ouvrir un tunnel afin d'envoyer des données brutes.

Pour envoyer des données d’une façon dynamique, HTTP utilise deux moyens:

- L’envoi de formulaires (texte, fichiers) par le chemin de ressource et/ou le corps.
- Le suivi de sessions par un en-tête spécifique (cookie).

7.3 Réponse HTTP

Un exemple de réponse du serveur HTTP est le suivant :

```
HTTP/1.1 200 OK
Server: monServeur v3
Date: Thu, 09 Oct 2020 09:30:24 GMT
... autres en-têtes...
```

```
Content-Type: text/html
Content-Length: 2037
<html>
...
</html>
```

Le code affiché de la réponse dépends du résultat de la requête. Les codes suivant sont affichés, dont chacun possède une signification :

- 1xx: information, réponse intermédiaire du serveur (exemple : 118 connection timedout).
- 2xx : succès de la requête (exemple : 200 OK).
- 3xx : redirection de la ressource (exemples : 300 multiple choices).
- 4xx : erreur du client HTTP (exemples : 404 not found).
- 5xx : erreur au niveau du serveur empêchant la satisfaction de la requête (exemples : 500 internal server error).

7.4 Exemple d’impélemntation d’un serveur HTTP en utilisant les sockets

En utilisant les sockets TCP, on proposera un exemple d’implémentation d’un simple serveur HTTP. Ce dernier reçoit une requête de type URL du navigateur Web sur le port 8888 et répond au client via un contenu html. Une fois le client se connecte au serveur via l'URL, ce dernier construit une réponse qui respecte le protocole HTTP (type de méthode, URI, version HTTP, entête, contenu, ...) et l'envoie au client (navigateur Web). Dans l'exemple qui suit, on implémente un simple serveur Web qui répond au client par la date du jour et le nombre de fois que la page a été visitée.

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


public class Httpserver implements Runnable {
  public static final String serverID; /* identifiant du serveur */
  public static final String prefix;/* chemin vers les ressources */
  public static final String shortPrefix;/* chemin réduit vers les ressources */
  public static final int PORT = 8888; /* le port utilisé pour accéder au serveur */
  public static int nbVisites;/* compteur de nombre de visites  */

  static {
    gid = 1;
    serverID = "simple http server ";
    prefix = "//Users/hocine/Documents/dossier/test.webarchive";
    shortPrefix = "/~hocine";
  }

  private int id;
  private Socket s;

  public Httpserver(Socket s) {
    this.s = s;
    this.id = nbVisites++;
    System.out.println(id+"Nouvelle connexion d’un client");
 }

/* méthode à executer quand le serveur (thread) est lancer */

  public void run() {
    try {
      BufferedReader bf = new BufferedReader(new InputStreamReader(s.getInputStream()));
      PrintWriter pw = new PrintWriter(s.getOutputStream());
     main_loop:
      while(true) {
       /* vecteur de la requête */
        Vector<String> request = new Vector<String>();
        do {
          /* Ligne de requête
```

```
          String s = bf.readLine();
          if (s==null) break main_loop;
          if (s.equals("")) break;
          request.add(s);
        } while(true);

        /* traitement de la requête */
        StringTokenizer st = new StringTokenizer(request.elementAt(0));
        String method = st.nextToken();

        if (method.equals("GET")) {

          /* traitement de l'url */
          String rawURL = st.nextToken();
          String url = rawURL;

          /* supprimer les caractères non utiles */
          if (url.indexOf('?')!=-1) url = url.substring(0,url.indexOf('?'));
          if (url.startsWith(specialPrefix)) url = url.substring(specialPrefix.length());

          /* ajouter le préfixe à l'url */
          url = prefix+url;

          File f = new File(url);
         /* créer le fichier de réponse si l'url obtenue est bien formulée */

          if (f.exists()) {
             pw.println("HTTP/1.1 200 OK");
             pw.println("Server: "+serverID);
             pw.println("Content-Length: "+(int)f.length());
             pw.println();

             FileReader fr = new FileReader(f);
             char [] buffer = new char[(int)f.length()];
             fr.read(buffer);
             pw.print(buffer);
          }
          else { /* url non valide */
             System.err.println(id+""+rawURL+" not found");
             pw.println("HTTP/1.1 404 Not Found");
             String msg = "<html><head><title>404 Not Found</title><body>"+
             rawURL+" not found.</body></html>";
             pw.println("Content-Length: "+msg.length());
             pw.println("Server: "+serverID);
             pw.println();
             pw.print(msg);
          }
       }
     else { /* méthode non reconnue*/
          System.err.println(id+"Method "+method+" not supported");
          pw.println("HTTP/1.1 405 Method Not Allowed");
          String msg = "<html><head><title>405 Method not allowed</title><body>"+
            method+" actually not supported.</body></html>";
          pw.println("Content-Length: "+msg.length());
          pw.println("Server: "+serverID);
          pw.println();
          pw.print(msg);
     }
     pw.flush();
   }
   bf.close();
   pw.close();
   System.out.println(id+" rip");
   }
   catch(Exception e) {
      System.err.println(id+"erreur dans le service");
      e.printStackTrace();
   }
 }
/* Le main pour lancer le serveur*/
public static void main(String []args) {
    System.out.println(" Serveur à l'écoute");
    try {
```

```
      ServerSocket ss = new ServerSocket(PORT);
      while (true) {
        Socket s = ss.accept();
        new Thread(new Httpserver(s)).start();
      }
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

Dans cet exemple le serveur Web répond au client uniquement par le la date du jour et le nombre de fois que la page est visitée quelque soit la requête. On peut enrichir ce serveur en traitant différentes requêtes et en programmant une variété de résultats. Il sera intéressant de gérer le style et le format de la réponse via d’autres langages et formats tels que XML, JSP, CSS, etc.

Annexe B : Socket et la programmation mobile en Java Android

Android est un système d'exploitation pour les supports mobiles tels que les téléphones et les tablettes. Il a été crée en 2004 et racheté par Google en 2005. Pour développer une application mobile en Android, il faut utiliser un environnement de développement tels que Android Studio qui facilite la tâche de programmation en fournissant une interface ainsi qu'un ensemble de bibliothèques et outils appelé SDK Android [6].

Une application Android est composée essentiellement de :

- Sources Java (ou Kotlin) compilés pour une machine virtuelle.
- Ressources : c'est les fichiers utilisés par l'application et qui peuvent êtres en format XML, images, etc.
- Manifeste: un fichier qui décrit le contenu du logiciel tel que les fichiers, les demandes d'autorisations, les signatures des fichiers, etc. Voici un exemple de déclaration de ce fichier:

```
<?xml version="1.0" encoding="utf-8"?>
      <manifest ... >
            <application android:icon="@drawable/app_icon.png" ...>
                  <activity android:name=".MainActivity" ... />
                  <activity android:name=".EditActivity" ...
            </application>
      </manifest>
... />
```

L'application peut également contenir éventuellement:

- Services : ou les processus qui tournent en arrière-plan.
- Fournisseurs de contenu : qui représentent les données
- Récepteurs d'annonces : qui sont utilisés pour gérer des événements envoyés par le système aux applications.

8.1 Création d'une activité

Une application mobile est structurée en un ensemble d'activités dont chacune corresponds à ce que l'interface ou l'écran de l'utilisateur peut contenir. Chaque écran de l'utilisateur est géré par une instance de la classe Activity. Pour spécifier ce que l'écran doit contenir, il suffit de surcharger la méthode **onCreate** de la classe **Activity** comme suit:

```
public class MainActivity extends Activity {
      @Override
      protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.main);
            }
 }
```

setContentView(identifiant de ressource) permet de mettre en place l'interface qui est composée de différentes vues telles que les boutons, zones de texte, etc. Chaque vue est gérée par un objet Java créés par l'intermédiaire d'un fichier XML. Différentes méthodes peuvent être utilisées telles que les écouteurs des vues, des menus et des chargeurs. On s'intéresse dans ce qui suit sur la programmation réseau à travers une application mobile.

8.2 AsyncTasks

Les fonctions d'une classe Activity sont exécutées par un seul thread appelé « Main thread ». Ce thread est par défaut endormit et se réveille lors d'un événement. Afin d'éviter que ce thread soit actif en permanence et causera le blocage de l'application, il faut implémenter des tâches asynchrones à l'aide de **AsyncTask**. Il s'agit d'une classe générique permettant de créer un autre thread indépendant de l'interface utilisateur et qui peut aider à la gestion de l'application pour permettre par exemple la communication en réseau [6]. Une tâche asynchrone AsyncTask est définie par les méthodes suivantes:

- **doInBackground :** permet de spécifier le traitement à faire pour le thread qui est indépendant de l'interface utilisateur.

- **onPostExecute :** cette méthode est appelée après doInBackground pour afficher des résultats sur l'interface.
- **Constructeur :** permet de passer des paramètres lors de la création de la tâche.
- **onPreExecute :** utilisée pour initialiser le traitement avant l'appel de la méthode doInBackground.
- **onProgressUpdate :** permet de mettre à jour l'interface.

AsyncTaskest une classe générique. Les paramètres de cette classe sont : AsyncTask<Params, Progress, Result>

- **Params :** est le type des paramètres de doInBackground.
- **Progress :** est le type des paramètres de onProgressUpdate.
- **Result :** est le type du paramètre de onPostExecute qui est aussi le type du résultat de doInBackground.

8.3 Exemple de serveur avec un client mobile

On souhaite implémenter une application client/serveur dont le client cette fois-ci est mobile et programmé en Java Android. Le serveur reçoit un message à partir de téléphone mobile et le montre à la console. Le code du serveur est similaire aux exemples déjà expliqués en sockets Java dans les sections précédentes en version non mobile. Ci-après un extrait de code du serveur.

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

public class Serveur {

    /* … */

    public static void main(String[] args) {
        try {
            serverSocket = new ServerSocket(PORT); // Server socket

        } catch (IOException e) {
            System.out.println("Impossible de lire sur le port donné");
        }

        while (true) {
            try {

                clientSocket = serverSocket.accept();
                inputStreamReader = new InputStreamReader(clientSocket.getInputStream());
                bufferedReader = new BufferedReader(inputStreamReader);

                /* Lirele message du client*/
                message_entrant = bufferedReader.readLine();
                /* Afficher le message sur la console */
                System.out.println(message_entrant);

                /* envoyer une confirmation de réception au client */
                printwriter = new PrintWriter(clientSocket.getOutputStream(), true);
                printwriter.write(" Bien reçu ");

               /* … */

            }
            /* … */
            } catch (IOException ex) {
               /* … */
            }
        }

    }

}
```

Au niveau du client, il faut faire la synchronisation des threads pour pouvoir communiquer en réseau. Cela s'effectue grâce à l'appel d'AsyncTask. Le client de cet exemple récupère le message entré par l'utilisateur à partir de la zone de texte puis l'envoie au serveur quand l'utilisateur appuie sur le button Envoyer.

```
public class MainActivity extends Activity {

    private Socket client;
    private PrintWriter printwriter;
    private Button button;
    private String message_sortant;
    private BufferedReader in_bufferedReader;
    private String message_entrant;
    private InputStreamReader in_streamReader;


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

       // Mettre une référence au button « envoyer »

        button = (Button) findViewById(R.id.button1);

        // ajouter un Listner au bouton pour detecter l'événement

        button.setOnClickListener(new View.OnClickListener() {

            public void onClick(View v) {
                message_sortant = ("");
                SendMessage sendMessageTask = new SendMessage();
                sendMessageTask.execute();

            }
        });
    }

    private class SendMessage extends AsyncTask<Void, Void, Void> {

        @Override
        protected Void doInBackground(Void... params) {
            try {
                /* … */
             /* se connecter au serveur */
                client = new Socket("IP_adress", PORT);

                /* envoyer un message*/
                Log.d("Connexion", " ... ");
                printwriter = new PrintWriter(client.getOutputStream(), true);
             /* … */
                printwriter.write(message_sortant);
             /* … */

                } catch (Exception e) {
            }
            return null;
        }
    }
}
```

Annexe C : Correction de la série d’exercices I

Exercice 1:

```
/***********************************************************************************************/
/*        Dévelopement d’un mini-réseau social en utilisant les flux d’entrée-sortie          */
/***********************************************************************************************/

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Utilisateur {

    private String nom, prenom, email, telephone, badge;
    private String posts;

    public Utilisateur(String nom, String prenom, String email, String telephone, String badge){
        this.nom = nom;
        this.prenom = prenom;
        this.email = email;
        this.telephone = telephone;
        this.badge = badge;
        try {
            //Creation des flux d'entrées sorties correspondant à l'utilisateur
             FileOutputStream fos_infos = new FileOutputStream(new
                                           File(this.telephone+"_user_infos.txt"));
             FileOutputStream fos_posts = new FileOutputStream(new
                                          File(this.telephone+"_user_posts.txt"));
            //On écrit alors dans le fichier contenant les infos de l'utilisateur
             String infos = "User" +this.nom+this.prenom+this.email+this.telephone+this.badge;
            byte b_tab[] = infos.getBytes();
            fos_infos.write(b_tab);
            fos_infos.close();
          } catch (FileNotFoundException ex) {
            System.err.println("FileNotFoundException " + ex);
          } catch (IOException e) {
            System.err.println("IOException" + e);
          }
       }

    /* méthode qui affiche les détails sur l'utilisateur */
    public String getDetails(){
        String s= "\nNom : "+this.nom+"\nPrénom : "+this.prenom;
        s += "\nEmail : "+this.email+"\nTéléphone : "+this.telephone;
        s +="\nPOSTS : ";
        String posts[] = this.get_posts();
        for(int i=0;i<posts.length;i++){
            s += "\n\t";
            s = s+"\n\t"+posts[i];
        }
        return s;
    }
    public String toString(){
        return this.prenom+""+this.nom+" ("+this.telephone+")";
    }

    /* méthode qui retourne le fichier contenant les informmations utilisateur */
    public File get_infos_file(){
        File f1 = new File(this.telephone+"_user_infos.txt");
        if(f1.exists()){
            return f1;
        }
        return null;
    }
```

```
/* méthode qui retourne le fichier contenant les posts de l'utilisateur */
public File get_posts_file(){
    File f1 = new File(this.telephone+"_user_posts.txt");
    if(f1.exists()){
        return f1;
    }
    return null;
}

/* méthode qui écrit les posts de l'utilisateur si son badge est blanc ou gris */
public boolean new_post(String data){
    File f = this.get_posts_file();
    if(f != null){
        try {
            FileInputStream fin = new FileInputStream(f);
            byte b_tab[] = new byte[4];
            String contenu = "";
            int k = 0;
           /*…*/
            while((k = fin.read(b_tab)) >=0){
                for(byte b : b_tab){
                    contenu = contenu+(char)b;
                }
                b_tab = new byte[4];
            }
            FileOutputStream fos = new FileOutputStream(f);
            contenu = contenu+data;
            byte b_tab1[] = contenu.getBytes();
            fos.write(b_tab1);
            fos.close();
            return true;

        } catch (FileNotFoundException ex) {
               System.err.println("FileNotFoundException " + ex);
        } catch (IOException e) {
               System.err.println("IOException " + e);
        }
    }
    return false;
  }

/* méthode qui écrit les posts de l'utilisateur si son badge est noir */
public boolean new_post(int numero){
    File f = this.get_posts_file();
    if(f != null){
        try {
            FileInputStream fin = new FileInputStream(f);
            byte b_tab[] = new byte[8];
            String contenu = "";
            int k = 0;
            while((k = fin.read(b_tab)) >=0){
                for(byte b : b_tab){
                    contenu = contenu+(char)b;
                }
                b_tab = new byte[4096];
            }
            FileOutputStream fos = new FileOutputStream(f);
            contenu = contenu+Settings.sep3+numero;
            byte b_tab1[] = contenu.getBytes();

            fos.write(b_tab1);
            fos.close();
            return true;

           } catch (FileNotFoundException ex) {
               System.err.println("FileNotFoundException " + ex);
           } catch (IOException e) {
               System.err.println("IOException " + e);
           }
       }
       return false;
    }
```

```
       /* méthode qui permet de lire les posts */
       public String[] get_posts(){
       File f = this.get_posts_file();
       if(f.exists()){
              try {
                 FileInputStream fin = new FileInputStream(f);
                 byte b_tab[] = new byte[4096];
                 String contenu = "";
                 int k = 0;
                 while((k = fin.read(b_tab)) >=0){
                     for(byte b : b_tab){
                         contenu = contenu+(char)b;
                     }
                 }
                 String posts[] = contenu.split(",");
                 return posts;
               } catch (FileNotFoundException ex) {
                     System.err.println("FileNotFoundException " + ex);
             } catch (IOException e) {
                  System.err.println("IOException " + e);
             }
         }
         return null;
     }

     @Override
     public boolean equals(Object o){
         Utilisateur u = (Utilisateur)o;
         return (this.email.equals(u.email) || this.telephone.equals(u.telephone));
     }

     public int hashCode(){
         return (this.telephone+this.email).hashCode();
     }

}

/*******************************************************************************************/

import java.io.IOException;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.Map;

/* la classe contenant la méthode principale main */
public class ReseauSocial {

    public static void main(String[] args) {

        /* créer des exemples d'utilisateurs et posts pour tester les classes précédentes */
        Utilisateur user1 = new Utilisateur("User1", "mohammed", "mohammed@gmail.com",
                           "0543061537", "blanc");
        user1.new_post("mon premier post");
        user1.new_post("mon second post");
        user1.new_post("mon troisième post");
        System.out.println(user1.getDetails());

        Utilisateur user2 = new Utilisateur("User2", "Imen", "imen@gmail.com",
                           "0643566544", "gris");
        user2.new_post("#je_suis_la");
        user2.new_post("#covid");
        System.out.println(user2.getDetails());

        Utilisateur user3 = new Utilisateur("User3", "Ihab", "ihab@gmail.com",
                           "0713262532", "noir");
        user3.new_post(23432);
        user3.new_post(23221);
        System.out.println(user3.getDetails());

/*********************************** Partie 2*********************************************/
        Relations r = new Relations();
        r.ajouterLien(user1, user2);
```

```
        r.ajouterLien(user1, user3);
        r.ajouterLien(user2, user3);
        r.afficherRelations();

       }

/*******************************************************************************************/
    /* établir le réseau social à l'aide d'une HashMap */
    public static class Relations{

         HashMap<Utilisateur, LinkedList<Utilisateur>> hm = new HashMap<>();

       /* méthode qui ajoute un lien entre deux utilisateur */
          public void ajouterLien(Utilisateur user_src, Utilisateur user_dest){
            LinkedList<Utilisateur> adjacents;
            if(hm.containsKey(user_src))
                    adjacents = hm.get(user_src);
            else
                    adjacents = new LinkedList<>();
            adjacents.add(user_dest);
            hm.put(user_src, adjacents);
        }

        /* méthode qui affiche les liens entre les utilisateurs */
        public void afficherRelations(){
            for(Map.Entry m : hm.entrySet()){
                System.out.println(m.getKey()+"------>"+m.getValue());
            }
        }
    }
}
```

Exercice 2:

```
/*******************************************************************************************/
/*                         Multithreds et sérialisation en Java                            */
/*******************************************************************************************/

/* colis */
public class Colis {
    public String identifiant;

    public Colis(String identifiant) {
        this.identifiant = identifiant;
    }
}

/*******************************************************************************************/

import java.util.ArrayList;
public class Stock {

    public ArrayList<Colis> contenu;
    public final int capacite; // capacité maximale du stock

    public Stock(int capacite){
        this.contenu = new ArrayList<Colis>();
        this.capacite = capacite;
    }

   /* ajouter un colis au stock */
    public synchronized boolean ajouter(Colis colis){
        while(this.contenu.size() == this.capacite){
            System.out.println("*** Plein ***");
            try {
                wait();
            } catch (InterruptedException e) {
                System.err.println("InterruptedException " + e);
            }
        }
```

```
        this.contenu.add(colis);
        notifyAll();
        return true;
    }

    /* retirer un colis du stock */
    public synchronized boolean retirer(){
        while(this.contenu.size() == 0){
            System.out.println("*** Vide ***");
            try {
              wait();
            }
           catch (InterruptedException ex) {
              System.err.println("InterruptedException " + ex);
           }
        }
        this.contenu.remove(0);
        notifyAll();
        return true;
    }
}

/*********************************************************************************************/
import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Fournisseur extends Thread{
    private Stock stock; // le stock utilisé
    private String identifiant_colis; // colis à préparer
    private Random random; // definer un temps aléatoire d'attente avant la préparer le colis

    public Fournisseur(Stock stock, String identifiant_colis){
        this.identifiant_colis = identifiant_colis;
        this.stock = stock;
        this.random = new Random();
    }

    /* Définir la fonction du thread une fois lancé */
    int k = 1;
    public void run(){
        while(true){
            Colis colis = new Colis(this.identifiant_colis+k);
            try {
                Thread.sleep(this.random.nextInt(2000) + 1000);
            } catch (InterruptedException ex) {
                 System.err.println("InterruptedException " + ex);
            }
            if(!this.stock.ajouter(colis))
                    System.out.println(" Le stock "+this.getId()+" est plein");
            else
                    System.out.println("Colis <"+colis.identifiant+"> ajouté par le
                    fournisseur "+this.getId());
        }
    }
 }

/*********************************************************************************************/
import java.util.Random;
public class Partenaire extends Thread{
    private Stock stock;
    private Random random;

    public Partenaire(Stock stock){
        this.stock = stock;
        this.random = new Random();
    }
    int k = 1;
    public void run(){
        while(true){
            try {
                Thread.sleep(this.random.nextInt(1000) + 1000);
            } catch (InterruptedException ex) {
```

```
                System.err.println("InterruptedException " + ex);
            }
            if(!this.stock.retirer())
                    System.err.println(" Le stock du "+this.getId()+" est vide");
            else
                    System.out.println("Le partenaire "+this.getId()+ " a retiré un colis !");
        }
    }
}

/*****************************************************************************************/
import java.util.ArrayList;
/* programme principal */
public class GestionColis {

    public static ArrayList<Colis> colis_en_attente = new ArrayList<Colis>();
    public static void main(String[] args){

        Stock stock = new Stock(10); // créer un stock de capacité de 10 colis maximum

        Fournisseur fournisseurs[] = new Fournisseur[3]; // créer des  objets fournisseurs
        fournisseurs[0] = new Fournisseur(stock, " Outils de travail ");
        fournisseurs[1] = new Fournisseur(stock, " Chaussures ");
        fournisseurs[2] = new Fournisseur(stock, " Vêtements ");

       // lancer les fournisseurs
        fournisseurs[0].start();
        fournisseurs[1].start();
        fournisseurs[2].start();

        Partenaire partenaires[] = new Partenaire[3]; // créer des objets partenaires
        partenaires[0] = new Partenaire(stock);
        partenaires[1] = new Partenaire(stock);

        // lancer les partenaires
        partenaires[0].start();
        partenaires[1].start();
    }
}
```

Annexe D : Correction de la série d'exercices II

Exercice 1:

```
/*****************************************************************************************/
/*    Application client/serveur permettant un processus d'authentification des utilisateurs  */
/*                  Elle utilise l'interface de programmation Socket UDP                   */
/*****************************************************************************************/

/*Client UDP Java */
import java.io.*;
import java.net.*;

public class Client{
   public static void main(String args[]){
       DatagramSocket sock = null; // socket udp
       int port = 7777; // port du serveur (quelqonque libre > 1024)
       String message;
       boolean notconnected=true;
       BufferedReader cin = new BufferedReader(new InputStreamReader(System.in));
       try{
             sock = new DatagramSocket(); // créer le socket pour échanger les données

             InetAddress host = InetAddress.getByName("localhost");
             // adresse ip du serveur testé en local (localhost)
             // ou mettre l'adresse ip ou le nom de la machine du réseau

             while(notconnected){
                   //lire le login et l'envoyer au serveur
                   System.out.println("Entrer le login : ");
                   message = (String)cin.readLine();
                   byte[] b = message.getBytes();

                   DatagramPacket paquetSortant = new DatagramPacket(b , b.length , host ,
                                                 port);
                   sock.send(paquetSortant);

                   System.out.println("Entrer le pwd : ");
                   message = (String)cin.readLine();
                   b = message.getBytes();

                   paquetSortant = new DatagramPacket(b , b.length , host , port);
                   sock.send(paquetSortant);

                   //reçevoir la réponse par un buffer

                   byte[] buffer = newbyte[65536];
                   DatagramPacket paquetEntrant = new DatagramPacket(buffer, buffer.length);
                   sock.receive(paquetEntrant);

                   byte[] data = paquetEntrant.getData();
                   message = new String(data, 0, paquetEntrant.getLength());

                   // afficher les details des données reçues
                   System.out.println(message);
                   if(message.equals("vous_etes_connecte"))
                         notconnected=false;
              }
        }
       catch(IOException e){
             System.err.println("IOException " + e);
       }
   }

}
```

```
/*****************************************************************************************/
/* L'utilisateur */

public class User {
	private String login;
	private String pwd;

	public User(String login, String pwd) {
		super();
		this.login = login;
		this.pwd = pwd;
	}
	public String getLogin() {
		return login;
	}
	public void setLogin(String login) {
		this.login = login;
	}
	public String getPwd() {
		return pwd;
	}
	public void setPwd(String pwd) {
		this.pwd = pwd;
	}
	@Override
	public String toString() {
		return "User [login=" + login + ", pwd=" + pwd + "]";
	}
}

/*****************************************************************************************/
/*Serveur UDP Java */

import java.io.*;
import java.net.*;
import java.text.DateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.Scanner;

public class ServeurAuthentification {
	static ArrayList<User>liste_clients = new ArrayList<User>(); // liste des utilisateurs
	static ArrayList<User>liste_noire = new ArrayList<User>(); // la liste noire

	static boolean pwd_valid(String pwd, String login) {
		for (User uu : liste_clients) {
			if (uu.getPwd().equals(pwd) &&uu.getLogin().equals(login))
				return true;
		}
		return false;
	}
	Static boolean Est_dans_liste_noire(String pwd, String login) {
		if (liste_noire.isEmpty())
			return false;
		for (User uu : liste_noire) {
			return (uu.getPwd().equals(pwd) &&uu.getLogin().equals(login));
		}
		Return false;
	}

	public static void main(String args[]) {
		DatagramSocket sock = null; // socket du serveur
		int nombre_clients = 0;
		int compt = 0;
		User u;
		try {
			// 0. L'admin donne l'ensemble de logins et mots de passes
			Scanner sc = new Scanner(System.in);

			System.out.println("Merci d'entrer le nombre de clients ");
			nombre_clients = sc.nextInt();
			String login = sc.nextLine();
```

```
                for (inti = 0; i<nombre_clients; i++) {
                      System.out.println("Merci de donner un nouveau login ");
                      login = sc.nextLine();
                      System.out.println("Merci de donner le pwd associé ");
                      String pwd = sc.nextLine();
                      u = new User(login, pwd);
                      liste_clients.add(u);
                      System.out.println("entree " + u.getLogin());
                      System.out.println(u.getPwd());

                }

                // 1. création de socket serveur qui écoute le port 7777
                sock = new DatagramSocket(7777);

                // un buffer pour reçevoir les données entrantes
                byte[] buffer = newbyte[65536];
                DatagramPacket paquetEntrant = new DatagramPacket(buffer,
                                              buffer.length);

                // 2. Attente pour reçevoir des données
                System.out.println("Attente des demandes client...");

                // boucle de communication
                while (true) {
                      sock.receive(paquetEntrant);
                      byte[] data = paquetEntrant.getData();
                      String s = new String(data, 0, paquetEntrant.getLength());

                      sock.receive(paquetEntrant);
                      data = paquetEntrant.getData();
                      String s2 = new String(data, 0, paquetEntrant.getLength());
                      String s3;
                      if (pwd_valid(s2, s)){
                            s3 = " vous_etes_connecte";
                      }
                      else if (Est_dans_liste_noire(s, s2))
                            s3 = " non autorisé à contacter ce serveur";
                            else {
                                  s3 = "Erreur login ou pwd: ";
                                  compt++;

                                  if (compt == 3) {
                                        User c = new User(s, s2);
                                        liste_noire.add(c);
                                        System.out.println(liste_noire.size());
                                         for(User f:liste_noire)
                                           System.out.println(f.getLogin()+""+ f.getPwd());
                                         compt=0;
                                  }
                            }
                      DatagramPacket paquetSortant =  new DatagramPacket(s3.getBytes(),
                                                        s3.getBytes().length,
                      paquetEntrant.getAddress(), paquetEntrant.getPort());
                      sock.send(paquetSortant);
                }
          }
          catch(IOException e){
                System.err.println("IOException " + e);
          }
    }
}
```

Exercice 2:

```
/*********************************************************************************************/
/*        Serveur de calcul et son client communiquant à l'aide des sockets TCP en Java      */
/*********************************************************************************************/

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
```

```
import java.text.DateFormat;
import java.util.Date;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;

public class ServeurTcp {
    public static void main(String[] args){
        ServerSocket sock; // socket d’écoute du port
        try {
            sock = new ServerSocket(7607); // port libre > 1024
            System.out.println("Creation de socket Serveur.");

            System.out.println("Attente de message...");
            Socket soc = sock.accept();
           // accepter la demande de connexion via le socket de service

           BufferedReader bin = new BufferedReader(new InputStreamReader(soc.getInputStream()));
            PrintWriter pwout = new PrintWriter(soc.getOutputStream(), true);
            String data;

            int i = 0;
            while((data = bin.readLine()) != null){
                String c = ""+s;
                //Analyse de la commande envoyée par le client
                String commande = s.substring(0, 3);

                switch(commande){
                    case "HEL" :
                        s = "Hello !";
                        break;
                    case "DAT" :
                        DateFormat shortDateFormat = DateFormat.getDateTimeInstance
                                 (DateFormat.SHORT,DateFormat.SHORT);
                        s = shortDateFormat.format(new Date());
                        break;
                    case "ADD" :
                        // On calcule la somme des deux entiers passés en paramètres
                        String[] tab = s.split("");
                        if(tab.length >= 3){
                            if(!tab[1].equals("")){
                                int a = Integer.parseInt(tab[1]);
                                if(!tab[2].equals("")){
                                    int b = Integer.parseInt(tab[2]);

                                    s = a+" + "+b+" = "+(a+b);
                                    break;
                                }
                            }
                        }

                        s = "Erreur d'utilisation de la commande ADD";
                        break;
                    case "MUL" :
                        //Multiplication
                        String[] tab2 = s.split("");
                        if(tab2.length >= 3){
                            if(!tab2[1].equals("")){
                                int a = Integer.parseInt(tab2[1]);
                                if(!tab2[2].equals("")){
                                    int b = Integer.parseInt(tab2[2]);

                                    s = a+" * "+b+" = "+(a*b);
                                    break;
                                }
                            }
                        }
                        s = "Erreur d'utilisation de la commande MUL";
                        break;
                    case "MIN" :
                        // On retourne le minimum
                        String[] tab1 = s.split("");
```

```
                        if(tab1.length >= 3){
                            if(!tab1[1].equals("")){
                                int a = Integer.parseInt(tab1[1]);
                                if(!tab1[2].equals("")){
                                    int b = Integer.parseInt(tab1[2]);
                                    s = a+" - "+b+" = "+(a-b);
                                    break;
                                }
                            }
                        }
                        s = "Erreur d'utilisation de la commande MIN";
                        break;
                    default :
                        s = "<<"+c+">> demande à réformuler...";
                }

                pwout.println(s);
                }
             pwout.close();
            bin.close();
            soc.close();
            sock.close();
            }
          catch(IOException e){
                System.err.println("IOException " + e);
          }
     }
}
```

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

public class ClientTcp{
    public static void main(String[] args) throws IOException {

        Socket socketClient = null;
        PrintWriter out = null;
        BufferedReader in = null;

        try {
            // création de socket
            socketClient = new Socket("127.0.0.1", 7607);
            out = new PrintWriter(socketClient.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(socketClient.getInputStream()));
            }
        catch (UnknownHostException e) {
             System.err.println("machine non connue: ");
             System.exit(1);
        }
        catch (IOException e) {
             System.err.println("erreur d'E/S");
             System.exit(1);
        }

       BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
       String userInput;
       System.out.print ("input: ");

       // boucle de communication
       while ((userInput = stdIn.readLine()) != null) {
             out.println(userInput);
             System.out.println("réponse: " + in.readLine());
       }

       out.close();
       in.close();
       stdIn.close();
       socketClient.close();

    }
}
```

Exercice 3:

```
/********************************************************************************************/
/*        Objets echangés entre un client et un serveur à l'aide des sockets TCP            */
/********************************************************************************************/

import java.io.*;
public class Point implements Serializable{
   /* un point est représenté par deux coordonnées x et y */
   private int x;
   private int y;

   public Point (int px, int py){
       x = px;
       y = py;
    }
    public Point (){
       this (0, 0);
    }
    public void setX(int px){
       x = px;
    }
    public int getX(){
       return x;
    }
    public void setY(int py){
       y = py;
    }
    public int getY(){
       return y;
    }
   /* claculer la distance entre ce point avec un autre passé en paramètre */
    public double distanceFrom (Point pt){
       int dx = Math.abs(x - pt.getX());
       int dy = Math.abs(y - pt.getY());
       return Math.sqrt((dx * dx) + (dy * dy));
    }
    public String toString(){
       String str = "(" + x + ", " + y +")";
       return str;
    }
}

/********************************************************************************************/
import java.io.*;
import java.net.*;
import java.util.Scanner;
public class Client {
       public static void main(String[] args) throws Exception {
              Scanner sc=new Scanner(System.in);
              System.out.print ("Donnez les coordonnées de votre point x: ");
              int x=sc.nextInt();
              System.out.println ("y: ");
              int y=sc.nextInt();
              Point p1 = new Point(x,y);

              // création de socket
              Socket s = new Socket("localhost",8888);
              ObjectOutputStream objectOut =new ObjectOutputStream(s.getOutputStream());
              // envoyer l'objet au serveur
              objectOut.writeObject(p1);
              objectOut.flush();
       }
}

/********************************************************************************************/
import java.io.*;
import java.net.*;
import java.util.ArrayList;

public class Serveur {
       public static void main(String[] args) throws Exception {
              /*liste pour sauvegarder temporairement les points reçus par le client */
```

```
            ArrayList<Point> l=new ArrayList<Point>();

            ServerSocket ss = new ServerSocket(8888);
            while (true) {
                  Socket service = ss.accept();
                  ObjectInputStream objectIn= new ObjectInputStream(service.getInputStream());
                   // récuperation de l'objet Point envoyé par un client
                  Point p = (Point)objectIn.readObject();
                  double d=0;
                  if(!l.isEmpty()){
                        Point p2 = l.get(l.size());
                        d=p.distanceFrom(p2);
                        l.add(p);
                  }
                  else{
                        l.add(p);
                  }

                  System.out.println("point recu: "+ p);
                  // calcul de distance
                  System.out.println("Distance avec le point précedent "+ d);
            }
      }
}
```

Exercice 4:

```
/*****************************************************************************************/
/*                              Serveur TCP multi-Clients                               */
/*****************************************************************************************/
import java.net.*;
public class ServeurMultiThread {

    public static void main (String[] args) {
        try {
           // création de socket d'écoute sur le port
            ServerSocket serveur = new ServerSocket(5566);
            System.out.println("serveur en écoute ...");

            while (true) {
                // creation de socket de service
                Socket client = serveur.accept();
                System.out.println(" un client arrive, nouveau thread crée ...");

                // associer la gestion de communication à un nouveau thread gestionnaire
                Gestionnaire gestionnaire = new Gestionnaire(client);
                gestionnaire.start();
            }
        }
        catch (Exception e) {
            System.err.println("Exception:" + e);
        }
    }
}

/*****************************************************************************************/
/* gestionnaire de communication avec un client */
import java.io.*;
import java.net.*;
import java.time.LocalDate;
import java.util.HashMap;

public class Gestionnaire extends Thread {

   Socket client;
   private HashMap<String,Integer> requestProtector=new HashMap<>();

   Gestionnaire (Socket client) {
        this.client = client;
   }
   public void run () {
        try {
            BufferedReader reader = new BufferedReader(new
```

```
                    InputStreamReader(client.getInputStream()));
            PrintWriter writer = new PrintWriter(client.getOutputStream(), true);

             while (true) {
                String line = reader.readLine();
                if (line.equals("date")){
                    LocalDate d= LocalDate.now();
                    writer.println(d.toString());
                }else{
                    protect(line);
                     writer.println("demande à reformuler");
                }

                if (line.equals("quitter"))
                    break;
            }
        }
        catch (Exception e) {
            System.err.println("Exception: client deconnecté.");
        }
        finally {
            try { client.close(); }
            catch (Exception e ){  }
        }
    }
    private void protect(String s) throws IOException {
        if(requestProtector.containsKey(s)){
            requestProtector.put(s,(requestProtector.get(s)+1) );
        }
       else{
            requestProtector.put(s,+1);
        }
        if(requestProtector.get(s)>=3){
            client.close();
            System.out.println("---------Tentative de piratage!!!-----------");
        }
    }
}

/************************************************************************************************/
import java.io.*;
import java.net.*;
public class ClientTcp {
    public static void main(String[] args) throws Exception {
        String userInput;
        Socket socket = new Socket("127.0.0.1", 5566);
        PrintWriter    out = new PrintWriter(socket.getOutputStream(), true);
        BufferedReader   in = new BufferedReader(new InputStreamReader(
                             socket.getInputStream()));
        BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
         System.out.println("[tapez 'quitter' pour deconnecter]");
        System.out.print ("input: ");

        while (!(userInput = stdIn.readLine()).equals("quitter")) {
            out.println(userInput);
            System.out.println( in.readLine());
            System.out.print ("input: ");
        }
        out.close();
        in.close();
        stdIn.close();
        echoSocket.close();
    }
}
```

Exercice 5:

```
/************************************************************************************************/
/*  Serveur de calcul et son client commuiquant à l'aide de l'interface de programmation RMI */
/************************************************************************************************/

import java.rmi.*;
public interface Calculator extends Remote{
```

```
        public double puissance (int a, int b) throws RemoteException;
        public long nbChiffres (int a) throws RemoteException;
}
/*********************************************************************************************/
import java.rmi.*;
import java.rmi.server.*;
public class CalculatorImpl extends UnicastRemoteObject implements Calculator {
        public CalculatorImpl() throws RemoteException {
                super();
        }
        public double puissance (int a, int b) throws RemoteException{
                return Math.pow(a, b);
        }
        public long nbChiffres(int a) throws RemoteException {
                return String.valueOf(a).length();
        }
}

/*********************************************************************************************/
import java.rmi.Naming;
public class CalculatorServer {
        public CalculatorServer(){
                try{
                        Calculator cal = new CalculatorImpl();
                        Naming.rebind("rmi://localhost:1099/CalculatorService", cal);
                }catch (Exception e) {
                        System.out.println("erreur: "+ e);
                        }
                }
        public static void main(String[] args) {
                CalculatorServer ser = new CalculatorServer();
        }
}

/*********************************************************************************************/
import java.net.MalformedURLException;
import java.rmi.*;
public class CalculatorClient {
        public static void main(String[] args) {
                try{
                        Calculator cal = (Calculator) Naming.lookup(
                                          "rmi://localhost/CalculatorService");
                        System.out.println(" Puissance (5,2): "+ cal.puissance(5,2));
                        System.out.println(" NombreChiffres(2020):"+ cal.nbChiffres(2020));
                    }catch (MalformedURLException murle){
                        System.out.println("java.net.MalformedURLException" + murle);
                    } catch (RemoteException re){
                        System.out.println("java.rmi.RemoteException");
                    }catch (NotBoundException nbe){
                        System.out.println("java.rmi.NotBoundException" + nbe);
                    }catch (java.lang.ArithmeticException ae){
                        System.out.println("java.lang.ArithmeticException" + ae);
                    }
              }
    }
```

Annexe E : Correction de la série d'exercices III

Exercice 1:

```
/****************************************************************************************/
/*               Serveur et Client communiquant à l'aide de Socket UDP
*/
/****************************************************************************************/
/* serveurUDP.c */
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#define PORT 5678
int main(int argc,char *argv[]) {
  struct sockaddr_in addr, from;
  int sock;
  char tampon[256];
  socklen_t lg;
  int r;
  int fact;
  sock = socket(PF_INET,SOCK_DGRAM,0); // famille de protocoles UDP/IP
  if (sock==-1) {
    perror("socket: ");
    exit(1);
  }
  addr.sin_family = AF_INET; // famille d'adresse
  addr.sin_port = htons(PORT); // port
  addr.sin_addr.s_addr = htonl(INADDR_ANY); // adresse IP

  if (bind(sock,(struct sockaddr *)(&addr),sizeof(addr))==-1) {
    perror("bind: ");
    close(sock);
    exit(1);
  }
  while (1) {
    // récupérer le message
    lg = sizeof(from);
    r = recvfrom(sock,tampon,256,0,(struct sockaddr *)(&from),&lg);
    if (r==-1) {
      perror("recv:");
      close(sock);
      exit(1);
    }
    // Print it
    tampon[r] = '\0';
    printf("Recu : %s depuis le port %d  \n ",tampon,ntohs(from.sin_port));
    if (!strcmp(tampon,"quit")) break;

   // calculer la factorielle du nombre reçu
   int nombre=atoi(tampon);

    if(nombre<0){
        strcpy(tampon, "nombre negatif !");
       if (sendto(sock,tampon,256,0,(struct sockaddr *)(&from),lg)==-1) {
      perror("erreur de transmission");
      close(sock);
      exit(1);
    }
   }
   else{
    int i;
    fact=1;
    for(i=1; i<=nombre; i++){
       fact=fact*i;
    }

    sprintf(tampon, "%d", fact);
    strcat(tampon, " et sa factorielle");
```

```
    if (sendto(sock,tampon,256,0,(struct sockaddr *)(&from),lg)==-1) {
      perror("erreur de transmission");
      close(sock);
      exit(1);
    }
   }
 }
 close(sock);
 return 0;
}

/*****************************************************************************************/
/* clientUDP.c */
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <strings.h>
#include <unistd.h>
#define PORT 5678

int main(int argc,char *argv[]) {
  struct sockaddr_in addrto, addrfrom;
  int sock;
  char tampon[256];
  struct hostent *hent;

  if (argc<3) {
    fprintf(stderr,"usage: %s host message\n",argv[0]);
    exit(1);
  }
  hent = gethostbyname(argv[1]);
  if (hent==NULL) {
    fprintf(stderr,"%s: host %s unknown\n",argv[0],argv[1]);
    exit(1);
  }
  sock = socket(PF_INET,SOCK_DGRAM,0); // famille de protocoles UDP/IP
  if (sock==-1) {
    perror("socket: ");
    exit(1);
  }
  addrto.sin_family = hent->h_addrtype; // adresse de destination
  memcpy(&(addrto.sin_addr.s_addr),hent->h_addr_list[0],hent->h_length);
  addrto.sin_port = htons(PORT);        // port

  strcpy(tampon,argv[2]);
  if (sendto(sock,tampon,256,0,(struct sockaddr *)(&addrto),sizeof(addrto))==-1) {
    perror("sendto: ");
    close(sock);
    exit(1);
  }
  if (strcmp(tampon,"quit")) {
    if (recv(sock,tampon,256,0)==-1) {
      perror("recv:");
      close(sock);
      exit(1);
    }
    printf(" %s\n", tampon);
  }
  close(sock);
  return 0;
}
```

Exercice 2:

```
/*****************************************************************************************/
/*                Serveur et Client communiquant à l'aide de Socket TCP
*/
/*****************************************************************************************/
/* serveurTCP.c */
#include<stdio.h>
```

```
#include<math.h>
#include<string.h>
#include<sys/socket.h>
#include<arpa/inet.h>
#include<unistd.h>
#include <stdlib.h>

int main(int argc , char *argv[]){
    int socket_desc , client_sock , c , read_size;
    struct sockaddr_in server , client;
    char client_message[2000];

    socket_desc = socket(AF_INET , SOCK_STREAM , 0); //créer socket TCP
    if (socket_desc == -1)
    {
        printf("impossible de créer le socket");
    }
    puts("Socket crée");

    //Préparer sockaddr_in
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons( 8800 );

    if( bind(socket_desc,(struct sockaddr *)&server , sizeof(server)) < 0){
        perror("échec de bind");
        return 1;
    }
    //Listen
    listen(socket_desc , 3);

    //accepter des connexions
    puts("Attente de réception de demandes de connexions ...");
    c = sizeof(struct sockaddr_in);

    //accepter les connexions de clients connection
    client_sock = accept(socket_desc, (struct sockaddr *)&client, (socklen_t*)&c);
    if (client_sock < 0){
        perror("erreur de accept ");
        return 1;
    }

    //Recevoir des messages de la part d'un client

    while( (read_size = recv(client_sock , client_message , 2000 , 0)) > 0 ){

        char client_response[2000]=" oui ";
        //Envoyer la réponse
        char cmd_message[2000];
        char cmd_nbr[2000];

        // extraire la commande et les nombres
        if(strlen(client_message)!=0){
              strncpy(cmd_message, client_message, 4 );
              strcpy(cmd_nbr, &client_message[4]);
        }
        int nombre=atoi(cmd_nbr); int res=0;
        if(strncmp(cmd_message, "fac", 3)==0){
            int i;
            res=1;
            for(i=1; i<=nombre; i++){
             res=res*i;
             }
             strcpy(client_response, "le resultat est :");
             sprintf(client_response, "%d", res);

         }
         else if(strncmp(cmd_message, "rac", 3)==0){
            res=0;
            res= sqrt(nombre);
            strcpy(client_response, "le resultat est :");
            sprintf(client_response, "%d", res);
         }
         else if(strncmp(cmd_message, "car", 3)==0){
```

```
            res=0;
            res= nombre*nombre;
            strcpy(client_response, "le resultat est :");
            sprintf(client_response, "%d", res);
         }
         else
            strcpy(client_response, "erreur: commande non correcte");
         write(client_sock , client_response , strlen(client_response));
    }
    if(read_size == 0){
        puts("Client disconnected");
        fflush(stdout);
    }
    else if(read_size == -1){
        perror("recv failed");
    }
    return 0;
}

/*********************************************************************************************/
/* clientTCP.c */
#include<stdio.h>
#include<string.h>
#include<sys/socket.h>
#include<arpa/inet.h>

int main(int argc , char *argv[]){
    int sock;
    struct sockaddr_in server;
    char message[1000] , server_reply[2000];

    //création de socket TCP
    sock = socket(AF_INET , SOCK_STREAM , 0);
    if (sock == -1){
        printf("erreur de création de socket");
    }
    server.sin_addr.s_addr = inet_addr("127.0.0.1");
    server.sin_family = AF_INET;
    server.sin_port = htons(8800);

    //se connecter avec le serveur
    if (connect(sock , (struct sockaddr *)&server , sizeof(server)) < 0){
        perror(" erreur de connexion");
        return 1;
    }
    puts("Connecté\n");

    // boucle de communication
    while(1){
        printf("Enter message : ");
        gets(message);
        if( send(sock , message , strlen(message) , 0) < 0){
            puts(" échec de transmission de données");
            return 1;
        }
        //recevoir une réponse du serveur
        if( recv(sock , server_reply , 2000 , 0) < 0){
            puts("erreur de recv");
            break;
        }
        puts("Réponse du serveur :");
        puts(server_reply);
    }
    close(sock);
    return 0;
}
```

Exercice 3:

```
/*********************************************************************************************/
/*                                Serveur multi-clients en C                                 */
/*********************************************************************************************/
```

```c
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<sys/socket.h>
#include<arpa/inet.h>
#include<unistd.h>
#include<pthread.h>

//la function qui gère les threads
void *connection_handler(void *);

int main(int argc , char *argv[]){
    int socket_desc , client_sock , c , *new_sock;
    struct sockaddr_in server , client;

    //créer socket
    socket_desc = socket(AF_INET , SOCK_STREAM , 0);
    if (socket_desc == -1){
        printf("Could not create socket");
    }
    puts("Socket crée");

    //Préparer la structure sockaddr_in
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons( 8888 );

    if( bind(socket_desc,(struct sockaddr *)&server , sizeof(server)) < 0){
        //print the error message
        perror("echec de bind");
        return 1;
    }

    listen(socket_desc , 3);

    //accepter de nouvelles connexions
    puts("Attente de connexions ...");
    c = sizeof(struct sockaddr_in);

    while( (client_sock = accept(socket_desc, (struct sockaddr *)&client, (socklen_t*)&c)) ){
        puts("connexion acceptée ");
        pthread_t sniffer_thread;
        new_sock = malloc(1);
        *new_sock = client_sock;

        if( pthread_create(&sniffer_thread, NULL, connection_handler, (void*) new_sock) < 0){
            perror("erreur de création de thread");
            return 1;
        }
    }
    if (client_sock < 0){
        perror("erreur de accept");
        return 1;
    }
    return 0;
}

/* Gérer la connexion avec un client */
void *connection_handler(void *socket_desc){

    //obtenir le describteur de socket
    int sock = *(int*)socket_desc;
    int read_size;
    char *message, client_message[2000];
    //Envoyer quelques messages au client
    message = "Bienvenue\n";
    write(sock , message , strlen(message));
    message = "Tapez un message \n";
    write(sock , message , strlen(message));
    //Reçevoir un message de la part du client
    while( (read_size = recv(sock , client_message , 2000 , 0)) > 0 ){
        //Echo: renvoyer le même message
        write(sock , client_message , strlen(client_message));
    }
```

```
    if(read_size == 0){
        puts("Client deconnecté");
        fflush(stdout);
    }
    else if(read_size == -1){
        perror("erreur de recv ");
    }
    free(socket_desc);
    return 0;
}
```

Exercice 4:

```
/*********************************************************************************************/
/*                     Partie 1 : serveur et client RPC                                      */
/*********************************************************************************************/

/* Fichier décrivant les procédures en RPCL (.x) */
program CALCULPROG {
    version CALCULVERSION {
        int CARREE(int) = 1;
       int CUBE(int) = 2;
    } = 1;
} = 0x20000001;

/*********************************************************************************************/

/* serveur.c */

#include <stdio.h>
#include "calcul.h"
int * carre_1_svc( int *x, struct svc_req * req){
    return *x**x;
}
int * cube_1_svc( int *x, struct svc_req * req){
    return *x**x**x;
}

/*********************************************************************************************/
/* client.c */
#include <stdio.h>
#include <rpc/rpc.h>
#include "calcul.h"

main(int argc, char ** argv){
    CLIENT * client;
    int * result;
    char * server;
    char * message;

    if (argc != 3) {
      fprintf(stderr, "Usage : %s <host><message>\n", argv[0]);
      exit(1);
    }
    server = argv[1];
    message = argv[2];

    /* créer le client */
    client = clnt_create(server, CALCULPROG, CALCULVERSION, "tcp");

    /* vérifier si la connexion est établit avec le serveur */

    if (client == NULL) {
      clnt_pcreateerror(server);
      exit(1);
    }

    /* appel de procedures distantes */
    result = carre_1(2, client);
    if (result == (int *) NULL) {
      /* erreur lors de l’appel du serveur */
      clnt_perror(client, server);
```

```
        exit(1);
     }
   else
        printf("%d", *result);

   if (*result == 0) {
        fprintf(stderr,"%s : impossible d'affciher le message \n",argv[0]);
        exit(1);
     };

    printf("message envoyé au serveur %s\n",server);
    clnt_destroy(client);
    exit(0);
}


/*****************************************************************************************/
/*                                    Partie 2 : RPCL                                    */
/*****************************************************************************************/

/* Fichier décrivant les procédures en RPCL (.x) */
struct  date {
       int heure;
       int minute;
       int seconde ;
};
struct comparaison {
       struct date d1;
       struct date d2;
};

program DATE_PROG {
       version DATE_VERSION_1 {
              int ESTVALIDE(date) = 1;
              int DIFFERENCE(comparaison c) = 2;
       } = 1;
} = 0x20000002;
```