République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université A. Mira de Béjaia
Faculté des Sciences Exactes
Département d'Informatique

Support de cours

Programmation distribuée

Cours et exercices destinés aux étudiants de première année master.
Spécialités : Administration et sécurité des réseaux (ASR) et
Réseaux et Système Distribué (ReSyD).

Réalisé par

Dr. Nadia Battat

Année 2021/2022

Table des matières

Table des figures

Liste des tableaux

Liste des abréviations

API	Application Program Interface.
DCE	Distributed Computing Environment.
DCOM	Distributed Component Object Model.
DNS	Domain Name Service.
JMS	Java Message Service.
JNDI	Java Naming and Directory Interface.
JRMP	Java Remote Method Protocol.
JVM	Java Virtual Machine.
IP	Internet Protocol.
MOM	Message Oriented Middleware.
OMG	Object Management Group.
OSF	pen Software Foundation.
RMI	Remote Method Invocation.
RPC	Remote Procedure Call.
RRL	Remote Reference Layer.
TCP	Transmission Control Protocol.
UDP	User Datagram Protocol.
URI	Uniform Resource Identifier.

Introduction générale

Le présent polycopié rassemble une collection de cours et execices du module "programmation distribuée" qui sont dispensés majoritairement en présentiel ou en distanciel (plateforme E-learning, Universite de Béjaîa). Ce support de cours est adressé aux étudiants de première année master, spécialités : Administration et sécurité des réseaux (ASR) et Réseaux et Système Distribué (ReSyD), ainsi qu'autres lecteurs qui veulent maîtriser les concepts de la programmation distribuée en java.

Le principal but de ce support de cours est donner aux étudiants une bonne formation en la programmation parallèle, concurrente et en la programmation réseau tout en considérant les deux modes de communication à savoir : synchrone et asynchrone. Il présente les principes de développement des applications distribuées en se basant sur les schémas de conception et l'architecture logicielle de ces applications, le modèle de programmation et les différentes technologies existantes. Ce support peut également aider les étudiants à réaliser leur propre simulateur pour évaluer des solutions proposées aux problèmes de recherches scientifiques, ou à mettre en œuvre des différentes plates-formes afin de réaliser leur projet de fin de cycle.

Pour pouvoir respecter le référentiel donné par le ministère de l'enseignement supérieur et de la recherche scientifique, ce polycopié est organisé en cinq chapitres suivis des séries de TD et de TP, à travers lesquels nous pouvons approfondir les différentes notions du cours et tester les connaissances acquises (le lecteur peut ainsi se référer aux exemples donnés).

Le premier chapitre concerne une introduction générale à la programmation distribuée. Nous définissons, en premier lieu, les concepts de base des systèmes distribués. Ensuite, nous présentons les notions fondamentales de la programmation distribuée tout en spécifiant les avantages du langage java pour les applications distribuées.

Le deuxième chapitre est dédié à l'étude de tous les aspects de la programmation des threads java.

Le troisième chapitre est consacré à la programmation réseaux. Tout d'abord, nous expliquons la manipulation des adresses. Ensuite, nous détaillons la mise en œuvre du modèle client-serveur à travers les sockets UDP et TCP.

Le quatrième chapitre aborde l'intergiciel (middleware) et l'appel de procédure à distance et explique la programmation par objets en utilisant java RMI.

Enfin, le cinquième chapitre définit les notions de base relatives au service de messagerie en java (JMS, Java Messaging Service) et présente en détaille son principe de fonctionnement.

Chapitre 1

Introduction à la programmation distribuée

Les progrès remarquables des équipements informatiques et de télécommunications durant ces dernières années ont permis une forte évolution des systèmes distribués et les environnements qui les utilisent.

Le présent chapitre est principalement constitué de deux parties. La première partie est consacrée aux systèmes distribués. Dans la seconde partie, nous présentons les définitions et les principaux concepts liés à la programmation distribuée et nous citons les avantages du langage java pour les applications distribuées.

1.1 Système distribué

Dans ce qui suit, nous présentons des concepts principaux des systèmes distribués.

1.1.1 Definitions

Un *système réparti* est :
- un système qui vous empêche de travailler quand une machine dont vous n'avez jamais entendu parler tombe en panne.
- Un ensemble de machines autonomes connectées par un réseau, et équipées d'un logiciel dédié à la coordination des activités du système ainsi qu'au partage de ses ressources.

Un *système distribué* peut être défini comme :
- un réseau d'entités ayant le même but commun : celui de la réalisation d'une tâche globale à laquelle chaque entité contribue par ses calculs locaux et les communications qu'elle entreprend, sans même qu'elle ait connaissance du dessein global du système.
- Un système qui s'exécute sur un ensemble de machines sans mémoire partagée, mais que pourtant l'utilisateur voit comme une seule et unique machine.

Remarque : Certains auteurs font la distinction entre les termes systèmes distribués et systèmes répartis. Le premier, de l'anglais "*distributed systems*", induirait l'idée d'une distribution des tâches effectuée par un coordinateur (un site central par exemple), tandis que le second supposerait une coopération des tâches, en vue de la répartition du travail. Il ne nous paraît pas nécessaire d'établir une telle distinction dans ce cours. La littérature anglaise ne fait d'ailleurs pas cette différence, et parle de distributed systems, où le terme distributed signifie "*réparti dans l'espace*".

1.1.2 Architecture

La table 1.1 représente l'architecture matérielle et logicielle d'un système distribué.

L'architecture matérielle	**L'architecture logicielle**
– Machine multi-processeurs avec mémoire partagée. – Cluster d'ordinateurs dédiés au calcul/traitement massif parallèle. – Ordinateurs standards connectés en réseau.	– Système logiciel composé de plusieurs entités s'exécutant indépendamment et en parallèle sur un ensemble d'ordinateurs connectés en réseau.

TABLE 1.1 – L'architecture matérielle et logicielle d'un système distribué

1.1.3 Caractéristiques d'un système distribué

Quand on s'intéresse aux caractéristiques d'un seul ordinateur, on examine la rapidité avec laquelle il est capable d'exécuter des instructions, de la quantité de RAM dont il dispose, de la taille et de la vitesse de son disque dur. Avec les systèmes distribués, il y a d'autres paramètres à prendre en compte comme :

- Les éléments formant le système (besoin d'identification et de localisation).
- Les activités parallèles et concurrentes.
- Les composants autonomes exécutant des tâches concurrentes ou parallèles.
- Les techniques et les protocoles de communication.
- Les contraintes du réseau : fiabilité (perte de données) et temps de propagation (dépendant du type de réseau et de sa charge).
- L'hétérogénéité des machines utilisées (puissance, architecture matérielle, etc.), des systèmes d'exploitation tournant sur ces machines, des langages de programmation des éléments logiciels formant le système, et des réseaux utilisés (réseau local rapide, Internet, réseaux sans fil, etc.).
- L'asynchronisme.

1.1.4 Motivation des traitements distribués

Les raisons pour lesquelles on peut vouloir construire des systèmes distribués sont de plusieurs natures :

- Utiliser et partager des ressources distantes : Bien souvent, la principale raison pour laquelle on crée un système distribué est qu'il permet la mise en liaison d'utilisateurs avec des ressources lointaines (une imprimante, une machine, un espace de stockage, un fichier, une page web, une base de données, etc.), et le partage de ces ressources de manière contrôlée. Une partie de ces ressources sont partagées pour des raisons économiques (dans une entreprise, des imprimantes peuvent être mutualisées entre plusieurs stations de travail, par exemple).
- Améliorer la qualité de service : Connecter des utilisateurs et des ressources permet parfois de meilleurs échanges d'information, et/ou une meilleure collaboration.
- Optimiser l'utilisation des ressources disponibles : Si une seule machine ne suffit

pas à faire fonctionner une application, on peut essayer de répartir la charge sur plusieurs machines. C'est typiquement le cas des datacenter entiers qu'utilisent Google, Facebook ou Twitter pour faire face au torrent de requêtes qui leurs sont adressées. C'est aussi le cas du calcul scientifique (par exemple les simulations numériques assez lourdes qui servent au calcul des prévisions météorologiques).
- Résister aux pannes : Si une application (par exemple, un serveur web) fonctionne sur une seule machine, et que cette machine tombe en panne, l'application s'arrête. Au contraire, on peut espérer qu'une application distribuée continue à fonctionner même si une partie des machines sur lesquelles elle s'exécute cesse de fonctionner. C'est typiquement le cas des applications de partage de fichier en pair-à-pair : la déconnection d'un des participants n'empêche pas les autres de continuer le partage. La tolérance aux pannes est difficile à obtenir. Une des difficultés tient à ce qu'il est difficile de faire la différence entre une machine qui a planté et une machine qui fonctionne, mais qui répond trop lentement.

De nombreuses possibilités sont donc offertes par les systèmes distribués. On peut citer :
- Transparence à la localisation : L'utilisateur ignore la situation géographique des ressources.
- Transparence d'accès : L'utilisateur accède à une ressource locale ou distante d'une façon identique.
- Transparence à l'hétérogénéité : L'utilisateur n'a pas à se soucier des différences matérielles ou logicielles des ressources qu'il utilise.
- Transparence aux pannes (réseaux, machines, logiciels) : Les pannes et réincarnations peuvent être cachées à l'utilisateur grâce aux systèmes de réplication.
- Transparence à l'extension des ressources : Extension ou réduction du système sans occasionner de gène pour l'utilisateur et en agissant sur les performances.

1.2 Programmation des systèmes distribués

La notion d'un *programme distribué* contient l'idée d'un assemblage de machines autonomes, dispersées géographiquement et dont le seul moyen de communiquer est un réseau (qui n'a généralement pas été conçu spécialement pour elles).

IL peut être défini comme un ensemble de processus qui tournent sur un système distribué afin de fournir ou utiliser des services déterminés (tâches). Une tâche peut être soit en cours d'exécution, soit en attente d'exécution. Un processeur, ou un nœud du réseau, n'est capable d'exécuter qu'une seule tâche à la fois. Certaines peuvent être exécutées simultanément (on parle de la *programmation parallèle*), tandis que d'autres ne peuvent commencer que lorsque certaines autre sont terminées. Dans ce dernier cas, on considère qu'il y a une dépendance entre les tâches (on parle de la *programmation concurrente*). Des tâches peuvent également donner naissance à d'autres tâches (algorithmes écrits dans un style récursifs). L'exécution d'un programme distribué peut être

terminée lorsque toutes ses tâches sont terminées.

1.2.1 Programmation parallèle

Une machine parallèle est un ordinateur possédant plusieurs unités de calcul coopérant dans le but d'exécuter une tâche commune, le programme parallèle. Ces unités peuvent :

- être exécutées littéralement en parallèle sur différents processeurs (on parle de parallélisme physique).
- semblent (au programmeur et à l'application) être exécutées en parallèle sur différents processeurs (on parle de parallélisme logique). En fait, l'exécution réelle des programmes a lieu de manière intercalée sur un seul processeur.

Le parallélisme dans l'exécution de logiciels peut se produire à quatre niveaux différents :

- Niveau instructions machine : Exécutant plusieurs instructions machine simultanément.
- Niveau instructions de code : Exécutant plusieurs instructions de code source simultanément.
- Niveau unités : Exécutant plusieurs unités (routines ou sous programmes) simultanément.
- Niveau programmes : Exécutant un même programme sur des données différentes et indépendantes. Cela permet de traiter un grand volume de données à répartir sur les processeurs et à traiter en même temps.

L'écriture de programmes parallèles et distribués se heurte à plusieurs difficultés. Premièrement, autant on dispose de langages de programmation efficaces, et validés par l'usage, pour écrire des programmes séquentiels, autant ce n'est pas vraiment le cas pour la programmation concurrente. Il y a de nombreux styles différents et incompatibles d'écriture de programmes parallèles, qui sont chacun plus ou moins adaptés à un type d'architecture donnée. Deuxièmement, l'écriture de programmes parallèles se heurte à des exigences contradictoires. D'un côté, il vaut mieux que le programme soit portable, c'est-à-dire qu'il ne soit pas écrit pour une machine particulière, mais qu'il puisse être exécuté sur une certaine variété de plateformes.

1.2.2 Programmation concurrente

La concurrence est un problème ancien en informatique et en particulier dans les systèmes d'exploitation : la mise à disposition d'un nombre limité de ressources à un nombre arbitraire de tâches utilisatrices conduit à des situations de concurrence. Elle peut être :

- Disjointe : Pas de communication entre les entités concurrentes.
- Compétitive : Compétition pour l'accès à des ressources (CPU, entrées/sorties, mémoire, etc.).

– Coopérative : Coopération pour atteindre un objectif commun.

On appelle programmation concurrente les techniques permettant :
– L'expression et la manipulation (construction, destruction, lancement, arrêt, etc.) d'entités concurrentes.
– La mise en place de moyens de communication et de synchronisation entre ces entités.

1.2.3 Middlewares (Intergiciels)

Un middleware (la couche logiciel qui s'intercale entre le système d'exploitation/réseau et les éléments de l'application distribuée) est un bus à objets répartis. Il peut être constitué d'un grand nombres de programmes fonctionnant éventuellement sur différentes machines et dans des processus distincts et assurant les communications entre les objets répartis.

Un middleware permet d'offrir un ensemble de services parmi lesquels on peut citer :
– Services de communication ou d'interaction : Chaque middleware possède un modèle de communication qui permet d'assurer l'interaction transparente entre les différents composants distribués. Il existe deux modèles de communications *synchrones* et *asynchrones*.
– Service de nommage : Les middlewares offrent un service de nommage qui permet d'enregistrer, identifier et rechercher les éléments et services connectés via le middleware. Dans ce service, chaque élément du système distribué est identifié par un nom unique public qui permet de le retrouver par les autres. Le nom de l'élément est associé à une référence qui contient toutes les informations techniques nécessaires pour établir une connexion avec lui comme l'adresse IP de la machine qui l'héberge, le port de l'application qui le déploie et son adresse mémoire. Dans ce service on publie également d'autres informations relatives aux éléments distribués comme son type, son interface, la liste des services qu'ils exposent, etc.

1.2.3.1 Catégories

Il existe plusieurs grandes familles de middleware :

1. *Middlewares basés sur l'appel distant des procédures* : Dans ce type de middleware, une partie serveur offre une opération appelée par une partie cliente. Il permet d'appeler une procédure ou méthode sur un élément appelé objet distant presque aussi facilement que localement. Parmi les middlewares de cette catégorie, on peut citer : RPC (Remote Procedure Call) et Java RMI (Remote Method Invocation).

2. *Middlewares basés sur l'envoi ou diffusion de messages ou événements* : Cette famille de middleware appelée MOM (Message Oriented Middleware) s'appuie sur

des modèles de communications asynchrones qui permet aux composants distribués d'échanger des messages. Dans cette catégorie, on peut citer comme exemple : JMS (Java Message Service).

3. *Middlewares basés sur une mémoire partagée* Dans cette catégorie de middleware, les composants distribués communiquent à travers une mémoire commune distribuée. Ce modèle permet une communication asynchrone avec des interactions faibles entre les composants distribués de l'application. Parmi les middlewares de cette catégorie, on peut citer : JavaSpace de Java.

4. *Middlewares Multi agents* Les middlewares de type multi agents permettent de construire des applications distribuées dont les composants sont représentés par des agents. Un agent est une entité autonome qui possède des objectifs à atteindre, capable de communiquer et interagir avec d'autres agents. Ils sont aussi capables d'apprendre à travers ses interactions avec le monde extérieur. Ils possèdent aussi d'autres caractéristiques comme la mobilité qui leur permet de migrer d'un conteneur d'agents vers un autre selon des contraintes liées aux applications. La communication entre les agents est de type asynchrone. Parmi les middlewares multi agents, on peut citer : JADE (Java Agent DEveloppement) et SACI (Simple Agent Communication Infrastructure).

1.3 Java et la programmation des systèmes distribués

Java est un langage de programmation orientée objet. Un objet, dans un modèle de programmation, est une représentation logicielle d'une entité du monde réel (telle qu'une personne, un compte bancaire, un document, etc.). Il est l'association d'un état et d'un ensemble de méthodes qui opèrent sur cet état.

Les propriétés suivantes font que les objets sont un bon mécanisme de structuration pour les systèmes distribués :

- Encapsulation : Le principe d'encapsulation dit qu'il n'est pas possible d'agir directement sur les données d'un objet ; il est nécessaire de passer par ses méthodes, qui jouent ainsi le rôle d'interface obligatoire. L'encapsulation est un outil puissant dans un environnement hétérogène : l'utilisateur d'un objet doit seulement connaître une interface pour cet objet, qui peut avoir des réalisations différentes sur différents sites.
- Héritage : L'héritage est un mécanisme de réutilisation qui permet de définir une nouvelle classe à partir d'une classe existante, à laquelle on ajoute de nouvelles données (attributs) et de nouvelles méthodes. Il faut retenir que toutes les classes dérivent d'une classe racine appelée "*java.lang.Object*". Il est donc utile pour les développeurs d'applications distribuées, qui travaillent dans un environnement

changeant et doivent définir de nouvelles classes pour traiter des nouvelles situations.
- Polymorphisme : En Java, une classe peut redéfinir certaines des méthodes hérités de sa classe de base. Cette possibilité est la clé de ce qu'on nomme le "*polymorphisme*", c'est-à-dire la possibilité de traiter de la même manière des objets de types différents, pour peu qu'ils soient issus de classes dérivés d'une même classe de base. Plus précisément, on utilise chaque objet comme s'il était de cette classe de base, mais son comportement effectif dépend de sa classe effective (dérivé de cette classe de base), en particulier de la manière dont ses propres méthodes ont été redéfinies.

1.3.1 Avantages du langage java pour les applications distribuées

Parmi les avantages, on peut citer :
- Architecture neutre : Java a été développé afin de pouvoir générer des applications qui soient indépendantes des machines et de leur système d'exploitation. Le compilateur génère un format de fichier objet dont l'architecture est neutre. Le code compilé est exécutable sur de nombreux processeurs, à partir du moment où le système d'exécution de java est présent. Pour ce faire, le compilateur java génère des instructions en byte code (ou pseudo-code) qui n'ont de lien avec aucune architecture d'ordinateur particulière. Au contraire, ces instructions ont été conçues pour être à la fois faciles à interpréter, quelle que soit la machine, et faciles à traduire à la volée en code machine natif.
- Langage dynamique : Java a été conçu pour s'adapter à un environnement en évolution constante. Les bibliothèques peuvent ajouter librement de nouvelles méthodes et variables sans pour autant affecter leurs clients.
- Distribution des objets : La manière la plus simple et la plus courante pour répartir des objets est de permettre aux objets qui constituent une application d'être situés sur un ensemble de sites répartis. Java possède des aspects liés à la distribution des objets grâce à l'utilisation de la bibliothèque RMI (Remote Methods Invocation) qui propose de la programmation d'objets distribués.
- Programmation réseau : L'une des forces de java est de pouvoir travailler sans peine en réseau. Il possède une importante bibliothèque de routines permettant de gérer différents protocoles (les protocole TCP/IP tels que HTTP et FTP, par exemple). Il offre également des mécanisme permattant la facilités de communication entre machines.
- Programmation parallèles : Java propose des exécutions parallèles de programmes grâce à une utilisation des threads (ou processus légers).
- Fiabilité et éfficacité : Java a été conçu pour que les programmes qui l'utilisent soient fiables sous différents aspects. Sa conception encourage le programmeur à traquer préventivement les éventuels problèmes, à lancer des vérifications dynamiques en cours dexécution et à éliminer les situations génératrices derreurs,

etc. Il se base sur une syntaxe simple et permet aussi une gestion automatique et protection de la mémoire.
– Sécurité : Java a été conçu pour être exploité dans des environnements serveur et distribués. Dans ce but, la sécurité n'a pas été négligée. Java permet la construction de systèmes inaltérables et sans virus.

Remarque : Java 2 Entreprise Edition (J2EE) prend en charge des applications distribuées à très large échelle.

1.4 Série d'exercices

Exercice 1 :

1. Définir les concepts suivants :
   - Système et application.
   - Système centralisé et système distribué.
   - Système distribué et application distribuée.
2. Citer les modèles d'interaction dans un système distribué.
3. Décrire les caractéristiques de chaque modèle.

Exercice 2 :

1. Expliquer quelles sont les différences entre la programmation classique et la programmation répartie.
2. Définir les deux concepts suivants : la programmation concurrente et la programmation parallèle ?
3. Comment un système d'exploitation permet-il d'avoir plusieurs tâches qui s'exécutent en même temps
4. Citer les principaux types de middleware.
5. Décrire les avantages de chaque type de middleware.

Chapitre 2

Programmation des threads java

Les threads permettent de gérer des activités parallèles au sein d'un même programme. Le contrôle de leur exécution peut se faire, au moins en partie, au sein du programme. Les threads peuvent communiquer entre eux et se partager des données. Le programme tout entier correspond lui à un " processus lourd ", géré par le système d'exploitation.

Dans ce chapitre, nous présentons les notions fondamentales de la programmation des threads java. Ainsi, nous expliquons comment créer et exécuter des threads. Ensuite, nous démontrons comment synchroniser et coordiner des threads.

2.1 Processus

Un programme est multitâche quand il peut lancer l'exécution de plusieurs parties de son code en même temps. Il s'appuie donc sur les processus ou les threads.

2.1.1 Définition

Rappelons la définition des processus donnée par *A.Tanenbaum* : un programme qui s'exécute et qui possède son propre espace mémoire : ses registres, ses piles, ses variables et son propre processeur virtuel (simulé en multi-programmation par la commutation entre processus effectuée par le processeur unique).

2.1.2 Gestion des ressources et exécution

Un processus a besoin de ressources : le processeur qui l'exécute, de la mémoire, des entrées sorties, etc. Certaines ressources ne possèdent qu'un point d'accès et ne peuvent donc être utilisées que par un processus à la fois (par exemple, une imprimante). On dit alors que les processus sont en exclusion mutuelle s'ils partagent une même ressource qui est dite critique. Il est nécessaire d'avoir une politique de synchronisation pour de telle ressource partagée.

Chaque processus peut lui-même fonctionner comme le système d'exploitation en lançant des sous-tâches internes au processus et par là même reproduire le fonctionnement de la multi-programmation. Ces sous-tâches sont nommées Threads.

2.1.3 Etats d'un processus

Un processus peut être :
- en cours d'exécution (running).
- Prêt à s'exécuter mais sans processeur libre pour l'exécuter.
- Bloqué (par manque de ressources).

2.2 Threads

Java supporte les threads. A l'inverse de la plupart des autres langages, le programmeur n'a pas à utiliser des librairies natives du système pour écrire des programmes multitâches.

2.2.1 Définitions

- Un thread est une unité d'exécution rattachée à un processus, chargée d'exécuter une partie du processus. Lorsqu'un processus est créé, un seul thread est associé au processus. Ce thread peut en créer d'autres. Chaque thread a : un identificateur unique, une pile d'exécution, des registres (un compteur ordinal) et un état.
  Exemple d'utilisation : Un serveur peut répondre à des demandes de connexions de clients en créant un thread par client.
- Multithreading : C'est la division d'un programme en plusieurs unités d'exécution (threads) évoluant en parallèle (exécution concurrente de threads). L'implémentation du multithreading varie considérablement d'une plateforme à une autre (threads Linux, threads de Win32, threads de Solaris et threads de POSIX). Le multithreading peut être implémenté :
  - au niveau utilisateur (threads utilisateur).
  - Au niveau noyau (threads noyau).
  - Aux deux niveaux (threads hybrides).

  Exemple d'utilisation : Un navigateur est un logiciel multi-thread : le chargement d'une page laisse la main à l'utilisateur.

2.2.2 Différences entre processus et thread

- Un processus léger est similaire à un processus car tous deux représentent l'exécution d'un ensemble d'instructions du langage machine d'un processeur. Du point de vue de l'utilisateur, ces exécutions semblent se dérouler en parallèle. Toutefois, là où chaque processus possède sa propre mémoire virtuelle, les processus légers appartenant au même processus père se partagent sa mémoire virtuelle. Par contre, tous les processus légers possèdent leur propre pile système.
- Les processus sont indépendants l'un de l'autre, tandis que les Threads appartiennent au même processus.
- Les processus peuvent communiquer entre eux à travers leur capacité d'échanges de données (processus distribués), tandis que les Threads peuvent avoir accès direct aux ressources occupées par leur processus.
- Changement de contexte entre processus est plus lent que celui entre Threads.

2.2.3 Avantage des threads par rapport aux processus système

- Rapidité de lancement et d'exécution.

- Partage des ressources système du processus englobant.
- Simplicité d'utilisation.

2.2.4 Utilité des threads

Les threads permettent de :
- Partager le temps alloué au processus père entre plusieurs threads, chacun d'eux exécutant une fonction précise.
- Gérer l'exécution de traitements longs sans bloquer les autres.
- Augmenter la "productivité" d'une application par l'exécution concurrente de ces threads.
- Donner la possibilité d'annulation de tâches séparables.
- Accéder à la programmation concurrente pour que plusieurs threads puissent travailler sur les mêmes données.
- Lancer plusieurs exécutions du même code simultanément sur différentes données
- Gérer des interfaces utilisateurs sophistiquées et la gestion de graphiques animés.
- Etc.

2.3 Programmation des threads java

Les threads Java sont des processus légers partageant un ensemble de codes, données et ressources au sein d'un processus lourd qui les héberge (ce processus est la machine virtuelle java (JVM, Java Virtual Machine)). A tout thread Java sont associés :
- Un objet qui détermine le code qui est exécuté par le thread.
- Un objet qui contrôle le thread et le représente auprès des objets de l'application (le contrôleur de thread).

2.3.1 La classe Thread

En java, il existe une classe *Thread* qui contient plusieurs méthodes permettant le threading d'une application. C'est-à-dire pouvoir faire exécuter plusieurs traitements en parallèle ou encore utiliser des boucles infinies sans bloquer le bon déroulement du programme. La classe Thread est définie dans le package *java.lang*. Elle implémente l'interface *Runnable*.

2.3.1.1 Constructeurs

La classe Thread peut posséder différents constructeurs (voir table 2.1).

Constructeur	**Description**
Thread()	Crée un nouveau thread dont le nom est généré automatiquement (aléatoirement).
Thread (String name)	Créer un objet thread du nom donné.
Thread(Runnable target)	*target* est le nom de l'objet dont la méthode run est utilisée pour lancer le Thread.

TABLE 2.1 – Constructeurs de la classe Thread

2.3.1.2 Méthodes

La classe Thread fournit des méthodes pour gérer les objets thread : les créer, changer leur priorité définie entre un minimum et un maximum avec une valeur moyenne par défaut. On peut attribuer un nom à un thread, le démarrer, l'interrompre, le détruire, le mettre en attente pendant un nombre de millisecondes, etc.

- int getPriority () : fournit la priorité du thread.
- void setPriority (int) : change la priorité.
- void setName (String) : définit le nom du thread.
- String getName () : fournit le nom du thread.
- static Thread currentThread () : fournit une référence sur le thread courant.
- static void sleep (long n) : le processus est suspendu pour n millisecondes.
- static int activeCount : nombre de threads du groupe.
- static int enumerate (Thread[] tabThread) : copie dans le tableau les références des threads actifs.
- void start () : la JVM appelle la méthode run() du thread.
- void run () : appel de la mèthode run() de l'objet.
- void suspend () : suspend le thread.
- void resume () : le thread redevient actif.
- void destroy () : détruit le thread.
- String toString () : fournit certaines des caractéristiques du thread.
- Etc.

Remarque : *destroy()*, *resume()*, *suspend()* et *stop()* sont des méthodes agissant sur l'état d'un thread, mais qui peuvent poser des problèmes de blocage ou de mauvaise terminaison et sont donc désapprouvées.

2.3.2 Création d'un thread

Lorsqu'une machine virtuelle java est démarrée, elle crée un premier thread applicatif, le "thread principal", qui appelle la méthode *main()* de la classe spécifiée. Ensuite des threads applicatifs peuvent être créés et détruits dynamiquement par le programme. La méthode *System.exit* permet d'arrêter la machine virtuelle. Sinon, l'exécution se poursuit jusqu'à la terminaison de tous les threads applicatifs, y compris le thread principal. En java, un thread est un objet d'une classe qui dispose d'une méthode nommée *run()* qui sera exécutée lorsque le thread sera démarré. Il peut être créé de deux manières.

– Soit en définissant une classe qui hérite de la classe *Thread* et en redéfinissant la méthode *run()*.
– Soit en définissant une classe qui implémente l'interface *Runnable*, ce qui consiste simplement à implémenter la méthode *public void run()*, et en créant une instance de Thread avec un objet Runnable en paramètre.

Dans les deux cas, il vous faudra instancier un objet de classe Thread et vous devrez mettre ou invoquer le code à "exécuter en parallèle" dans le corps de la méthode *run()* de votre classe.

Méthode de création	**Avantages**	**Inconvénients**
extends java.lang.Thread	Chaque thread a ses données qui lui sont propres.	On ne peut plus hériter d'une autre classe.
implements java.lang.Runnable	L'héritage reste possible. En effet, on peut implémenter autant d'interfaces que l'on souhaite.	Les attributs d'une classe peuvent être partagés pour tous les threads qui y sont basés. Dans certains cas, il peut s'avérer que cela soit un atout.

TABLE 2.2 – Méthodes de création d'un thread

2.3.3 Exécution d'un thread

Il est possible de créer et d'exécuter plusieurs threads dans un programme. Il suffit d'affecter à chacun un nouvel objet. Une fois créé, un thread ne fait rien tant qu'il n'a pas commencé avec la méthode *start()*. Cette méthode alloue les ressources nécessaires à l'exécution d'un thread et invoque la méthode *run()*.

Première méthode

```
public class MyThread extends Thread {
 public void run(){
  for (int i=1; i<=10; i++)  System.out.print(i + " ");
 }
public static void main(String [] args){
 MyThread t = new  MyThread();
 t.start(); /*  void start () :  la JVM appelle la méthode run()
du thread*/
}}
```

Deuxième méthode

```
public class MyRunnable implements  Runnable{
 public static void main(String [] args){
       MyRunnable t1 = new  MyRunnable ();
      new Thread(t1).start();
 }
 public void run(){
   for (int i=1; i<=10; i++)    System.out.print(i + " ");
 }

}
```

FIGURE 2.1 – Programme-2.1 : Création et exécution d'un thread

Remarques :
– Si vous invoquez la méthode *run()* (au lieu de *start()*) le code s'exécute bien mais aucun nouveau thread n'est lancé dans le système. Inversement, la méthode

start(), lance un nouveau thread dans le système dont le code à exécuter démarre par le *run()*.
- La méthode *run()* :
  - est toujours de type *public*.
  - Est héritée de la classe *Thread*.
  - Doit être redéfinie si vous désirez que votre thread fasse quelque chose.
  - Contient tout le code qui va être exécuté par le thread.

2.3.4 Ordonnancement des threads

L'ordre d'exécution est souvent aléatoire. En fait, l'ordonnanceur gère les threads de façon aléatoire : il va en faire tourner un pendant un certain temps, puis un autre, puis revenir au premier, etc., jusqu'à ce qu'ils soient terminés.

2.3.4.1 Sommeil d'un thread

L'appel de la méthode *sleep (long n)* demande que le thread correspondant soit arrêté (on dira "mis en sommeil") pour au moins la durée mentionnée (n millisecondes). Cette démarche laisse ainsi la possibilité à d'autres threads de s'exécuter à leur tour. La méthode *sleep()* est susceptible de générer une exception de type *InterruptedException* si le thread est stoppé pendant son sommeil (par un appel de la méthode *interrupt()* par exemple).

```
public class TstThr1 {
 public static void main (String args[]) {
  Ecrit e1 = new Ecrit ("Bonjour ", 10, 5);
  Ecrit e2 = new Ecrit ("Bonsoir ", 12, 10);
  Ecrit e3 = new Ecrit ("\n", 5, 15);
  e1.start();
  e2.start();
  e3.start();
 } }
class Ecrit extends Thread {
 public Ecrit (String texte, int nb, long attente) {
  this.texte = texte;
  this.nb = nb;
  this.attente = attente;
 }
 public void run () {
  try {
       for (int i=0 ; i<nb ; i++) {  System.out.print (texte) ; sleep (attente) ;   }
   } catch (InterruptedException e) {} // impose par sleep
 }
private String texte ; private int nb ; private long attente ;
}
```

FIGURE 2.2 – Programme-2.2 : Sommeil d'un thread

2.3.4.2 Configuration des priorités

La méthode *sleep()* permet d'endormir un thread au profit d'un autre. Il existe un autre moyen de gérer le passage du contrôle d'un thread à un autre : à chaque thread de java est associée une priorité c'est à dire une valeur numérique qui permet de comparer l'importance de deux threads et de donner une préférence d'exécution à l'un ou à l'autre. Chaque thread de java possède une priorité, qui se situe entre *Thread.MAX_PRIORITY* (priorité maximale d'un thread) et *Thread.MIN_PRIORITY* (priorité minimale d'un thread). *NORM_PRIORITY* : priorité attribuée par défaut à un thread.

- La valeur de Thread.MAX_PRIORITY est : 10.
- La valeur de Thread.MIN_PRIORITY est 1.
- La valeur de NORM_PRIORITY est 5.

Remarque : La configuration des priorités est une tâche ardue, car elle dépend du système d'exploitation, et le résultat dépend des autres tâches qui s'exécutent simultanément.

Lorsqu'un thread est créé, il hérite de la priorité du thread qui le crée. Cette priorité peut être modifiée en utilisant la méthode *setPriority(int)* en donnant une valeur entre *MIN_PRIORITY* et *MAX_PRIORITY*. Pour des thread candidats à l'exécution ayant la même priorité, java fait tourner le droit d'exécution d'un thread à l'autre dès que ce droit est libéré et à condition qu'il n'y ait pas un thread de priorité supérieure. La priorité est principalement utilisée pour gérer l'efficacité d'une application.

```
public class Nombres {
public static void main(String[] args ){
 Thread nb1,nb2,nb3;
 nb1= new Thread(new PrintNb(1));   nb1.start();
  nb1.setPriority(Thread.MIN_PRIORITY);

 nb2= new Thread(new PrintNb(2));   nb2.start();
  nb2.setPriority(Thread.MAX_PRIORITY);

  nb3= new Thread(new PrintNb(3)); nb3.start();
 nb3.setPriority(Thread.NORM_PRIORITY);
} }
```

```
public class PrintNb implements Runnable{
 int nb;
 public PrintNb(int nb){ this.nb = nb; }
  public void run(){
          System.out.println();
   for (int i=0;i<10;i++)    System.out.print(nb);
  }
}
```

FIGURE 2.3 – Programme-2.3 : Configuration de la priorité des threads

2.3.4.3 La méthode yield()

Un thread donné peut, à tout moment, renoncer à son droit d'exécution en appelant la méthode *yield()*.

Contrairement à *sleep(long)*, ce renoncement n'est pas une mise en sommeil pour une durée fixée au profit de tous les threads, mais seulement un passage temporaire du contrôle aux autres threads de même priorité. Notons que *yield()* n'émet pas d'exceptions, nous n'avons donc pas besoin de l'entourer d'une close *try* and *catch()*.

```
class YieldThread extends Thread {
        public void run() {
                for (int count = 0; count < 4; count++) {
                        System.out.println( count + " From: " + getName() );
                        yield();    }
        }
}

class TestYield {
        public static void main( String[] args ) {
                YieldThread first  = new YieldThread();
                YieldThread second = new YieldThread();
                first.start();
                second.start();
                System.out.println( "End" );   }
}
```

FIGURE 2.4 – Programme-2.4 : Utilisation de la méthode yield()

2.3.4.4 Interruption d'un thread

Les threads s'achevaient tout naturellement avec la fin de l'exécution de leur méthode *run()*.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class TstInter {
 public static String lireString() { String ligne_lue = null;
          try  { InputStreamReader lecteur = new
        InputStreamReader(System.in);
        BufferedReader entree = new
BufferedReader(lecteur);
        ligne_lue = entree.readLine();      }
          catch (IOException err)    { System.exit(0);   }
                return ligne_lue;  }
        public static void main (String args[]) {
                Ecrit2 e1 = new Ecrit2("bonjour ",5);
                Ecrit2 e2 = new Ecrit2("bonsoir ",10);
                Ecrit2 e3 = new Ecrit2("\n",35);
                e1.start();  e2.start(); e3.start();
        lireString();          // Lecture d'un caractère
        e1.interrupt();  // interruption premier thread
System.out.println("\n*** Arret premier thread***");
        lireString();
        e2.interrupt(); e3.interrupt();
                System.out.println("\n*** Arret de
deux derniers threads***");
        }}
```

```
class Ecrit2 extends Thread {
public Ecrit2 (String texte, long attente) {
        this.texte = texte;
        this.attente = attente;  }
 public void run() {
   try {
        while (true) { // boucle infinie
          if (interrupted()) return;
            System.out.print(texte);  sleep(attente); }
   } catch (InterruptedException e) {
        return;      // on peut omettre return ici
        }          }
        private String texte;
        private long attente;
}
```

FIGURE 2.5 – Programme-2.5 : Interruption d'un thread

Dans certains cas, on peut avoir besoin d'interrompre prématurément un thread depuis un autre thread. Ce besoin peut devenir fondamental dans le cas de ce que nous nommerons des "threads infinis", c'est-à-dire dans lesquels la méthode *run()* n'a pas de fin programmée (ce pourrait être le cas d'un thread de surveillance d'appels dans un serveur Web). Java dispose d'un mécanisme permettant à un thread d'en interrompre un autre. La méthode *interrupt()* de la classe *Thread* demande à l'environnement de positionner un indicateur signalant une demande d'arrêt du thread concerné. Par ailleurs, dans un thread, il est possible de connaître l'état de cet indicateur à l'aide de la méthode statique *interrupted()* (il existe également *isInterrupted*).

2.3.4.5 Attente de la fin d'un thread

La méthode *join()* de la classe *Thread* attend la fin de l'exécution d'un thread.

```
class CustomThread extends Thread {
 CustomThread (String name) {
    super (name) ; start();}
public void run () {
 try{   for (int i= 0; i<4; i++) {
    System.out.println( Thread.currentThread().getName()
+ " thread s'exécute....");
   Thread.sleep(1000); }
} catch (InterruptedException e ) {}
System.out.println((Thread.currentThread()).getName() +
" terminé.");}}
```

```
class JoinExample {
public static void main (String args []) {
CustomThread t1 = new CustomThread(" Premier ");
CustomThread t2 = new CustomThread(" Deuxième ");
CustomThread t3 = new CustomThread(" Troisième ");
CustomThread t4 =  new CustomThread(" Quatrième ");
try {
t1.join (); t2.join (); t3.join (); t4.join ();
} catch (InterruptedException e ) {}  }}
```

FIGURE 2.6 – Programme-2.6 : Attente de la fin d'un thread

2.3.5 Coordination de threads

Dans certains cas, il faudra éviter que deux threads puissent accéder (presque) en même temps au même objet. Ou encore, un thread devra attendre qu'un autre ait achevé un certain travail sur un objet avant de pouvoir lui-même poursuivre son exécution.

- Le premier problème peut être réglé par l'emploi de la synchronisation.
- Le second problème peut être réglé par la mise en œuvre des mécanismes d'attente et de notification.

2.3.5.1 Synchronisation des threads

La synchronisation peut consister à entremêler les exécutions des threads de manière à ce qu'ils n'accèdent à certaines données ou à certains morceaux (ou bloc) de code que chacun à leur tour alternativement.

- Une méthode plus simple est la synchronisation sur terminaison (on veut qu'un morceau de code ne s'exécute qu'après qu'un thread donné ait terminé). La méthode join() permet une telle synchronisation.

- Une autre méthode pour synchroniser un bloc de code consiste à utiliser le mot clé *synchronized*, en indiquant l'objet auquel il faut restreindre l'accès.
- Pour bloquer l'accès aux autres threads en synchronisant les méthodes, on peut également utiliser le mot clé *synchronized* dans la définition de la méthode.

2.3.5.2 Communication entre threads

Les threads ont souvent besoin de se coordonner mutuellement, en particulier quand le résultat de l'un est utilisé par un autre. Une manière de coordonner les threads est d'utiliser les méthodes *wait()* et *notify()* (ou *notifyAll()*) :

- wait() : met un thread en sommeil jusqu'à un appel à *notify()* ou *notifyAll()* sur le même objet.
- notify() : démarre le premier thread qui a appelé *wait()* sur le même objet.
- notifyAll() : démarre tous les threads qui ont appelé *wait()* sur le même objet.

```
class CustomThread1 extends Thread {
Shared shared;
public CustomThread1(Shared shared, String string){
super(string);
this.shared = shared;
start(); }
public void run (){
System.out.println (" Le résultat est "  +
shared.getResult()); }}
class CustomThread2 extends Thread {
Shared shared ;
public CustomThread2 (Shared shared , String string) {
super(string);
this.shared = shared;
start(); }
public void run (){
shared.doWork(); } }
class WaitExample {
public static void main (String args[]) {
Shared shared = new Shared();
CustomThread1 t1 = new CustomThread1(shared, " Un ");
CustomThread1 t2 = new CustomThread1(shared, " Deux "); }}
```

```
class Shared {
int data = 0 ;
synchronized void doWork(){
try{
Thread.sleep(1000);
} catch (InterruptedException e ){}

data = 1;
notify();
}
synchronized int getResult(){
try{
wait();
}catch (InterruptedException e){}
return data;
}}
```

FIGURE 2.7 – Programme-2.7 : Synchronisation et communication entre threads

2.3.6 Etats d'un thread

Au départ, on crée un objet thread. Tant que l'on ne fait rien, il n'a aucune chance d'être exécuté. L'appel de *start()* rend le thread disponible pour l'exécution. Il est alors considéré comme prêt. L'environnement peut faire passer un thread de l'état prêt à l'état en exécution.

Un thread en cours d'exécution peut subir différentes actions :

- Il peut être interrompu par l'environnement qui le ramène à l'état prêt ; c'est ce qui se produit lorsque l'on doit donner la main à un autre thread (sur le même processeur). Cette transition peut être programmée en appelant la méthode *yield()* de la classe *Thread*.
- Il peut être mis en sommeil par appel de la méthode *sleep()*. Cet état est différent de prêt car un thread en sommeil ne peut pas être lancé par l'environnement. Lorsque le temps de sommeil est écoulé, l'environnement replace le thread dans l'état prêt (il ne sera relancé que lorsque les circonstances le permettront).
- Il peut être mis dans une liste d'attente associée à un objet (appel de *wait()*). Dans ce cas, c'est l'appel de *notify()* (ou *notifyAll()*) qui le ramènera à l'état prêt.
- Il peut lancer une opération d'entrée-sortie et il se trouve alors bloqué tant que l'opération n'est pas terminée.
- Il peut poursuivre l'exécution (appel de *resume ()*), là où il a été suspendu.
- Il peut s'achever.

Remarque : Dans un environnement mono-thread, lorsqu'un thread se bloque (voit son exécution suspendue) en attente d'une ressource, le programme tout entier s'arrête.

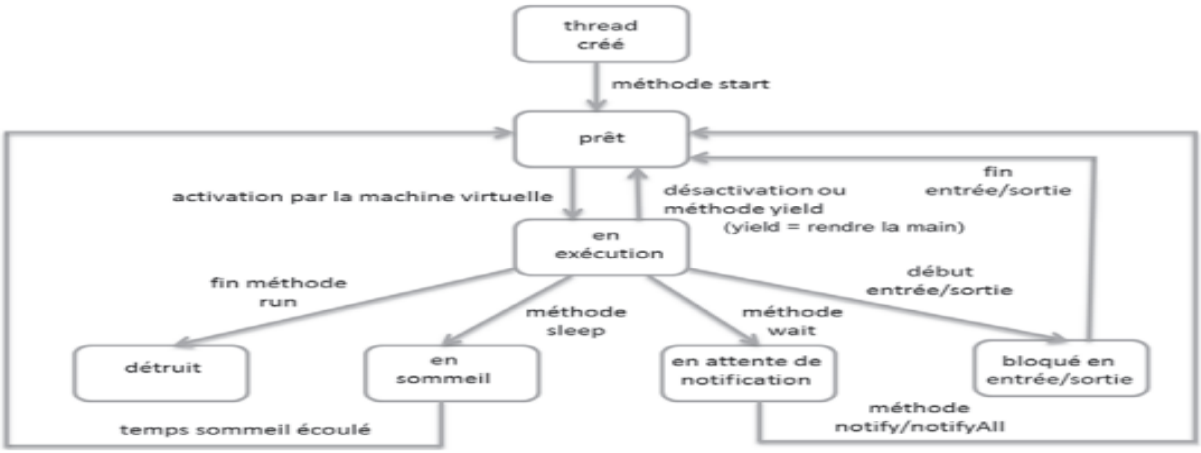

FIGURE 2.8 – Etats d'un thread

2.3.7 Groupes de threads

La classe *ThreadGroup* représente un ensemble de threads. Il est ainsi possible de regrouper des threads selon différents critères. Il suffit de créer un objet de la classe *ThreadGroup* et de lui affecter les différents threads. Un objet *ThreadGroup* peut contenir des threads mais aussi d'autres objets de type *ThreadGroup*.

Remarque : La notion de groupe permet de limiter l'accès aux autres threads. Chaque thread ne peut manipuler que les threads de son groupe d'appartenance ou des groupes subordonnés (par défaut, un thread est créé dans le groupe courant).

On peut définir un nouveau thread dans un groupe donné par ces constructeurs :

Constructeur	**Description**
Thread (ThreadGroup group, Runnable target)	Créer un objet thread faisant partie d'un groupe.
Thread (ThreadGroup group, Runnable target, String name)	Créer un objet thread avec un objet *Runnable*. Cet objet thread prend le nom indiqué et il appartient au groupe de threads désigné par *group*.
Thread (ThreadGroup group, String name)	Créer un objet thread qui fait partie d'un groupe, sous le nom indiqué.

TABLE 2.3 – Constructeurs pour créer groupes de threads

2.3.7.1 Méthodes

On peut demander un certain nombre d'informations en faisant appel à ces méthodes :

Méthodes	**Description**
Groupe.getParent()	Permet de connaître le groupe parent.
Groupe.getName()	Permet de connaître le nom du groupe.
Groupe.activeCount()	Permet de connaître le nombre de threads du groupe.
Groupe.activeGroupCount()	permet de connaitre le nombre de groupes de threads.
Groupe.enumerate(Thread[] list) et Groupe.enumerate(Thread[] list, boolean recurse)	Permettent d'énumérer les threads d'un groupe.
Groupe.enumerate(ThreadGroup[] list)et Groupe.enumerate(ThreadGroup[] list, boolean recurse)	Permettent d'énumérer les groupes. Le booléen recurse indique si on désire avoir la liste récursivement, pour chaque sous-groupe etc.

TABLE 2.4 – Méthodes de manipulation des groupes de threads

2.4 Création d'un groupe de threads

```
public class TestThreadGroup {
public static void main(String[] args) {
  Runnable runnable = new MonTraitement();
 ThreadGroup monThreadGroup = new ThreadGroup("Mon groupe de threads");
 Thread t = new Thread(monThreadGroup, runnable);
 System.out.println("groupe:" + t.getThreadGroup().getName());
 t.start();   }}
```

```
public class MonTraitement implements Runnable {
 public void run(){
                    System.out.println("Mon traitement"); }}
```

FIGURE 2.9 – Programme-2.8 : Création d'un groupe de threads

2.5 Série d'exercices

Exercice 1 :

1. Rappeler la différence entre un thread et un processus.
2. Spécifier les différents états d'un processus.
3. Préciser les différents états d'un thread.
4. Citer les avantages et les inconvénients des threads par rapport aux processus.
5. Quelles sont les ressources partagées entre threads d'un même processus.
6. Quelles sont les ressources privées à chaque thread ?

Exercice 2 :

1. Ecrire une classe Compteur qui hérite de la classe Thread. Un compteur est caractérisé par les attributs suivants : une valeur initiale de type entier, le pas de changement de la valeur du compteur, le sens de comptage (croissant ou décroissant), le nombre de comptage à effectuer, et enfin la vitesse du comptage (i.e. le temps entre deux changements de la valeur du compteur). A chaque changement la nouvelle valeur du compteur est affichée.
2. Ecrire la classe principale qui permet de lancer au moins deux compteurs différents.
3. Réécrire la classe Compteur en utilisant l'interface Runnable.
4. Modifier la classe principale pour qu'elle implémente le comportement d'un thread à travers l'utilisation de l'interface Runnable.

Exercice 3 : Voici un algorithme de tri en ordre croissant d'une tranche de tableau comprise entre les éléments d'indices debut et fin :

```
trier(debut, fin) {
 si (fin - debut< 2) {
    si (t[debut] > t[fin])  echanger(t[debut], t[fin])
 }
 sinon {  milieu = (debut + fin) / 2
          trier(debut, milieu)
          trier(milieu + 1, fin)
          triFusion(debut, fin)
  }  }
```

1. On remarque que les 2 tris qui sont effectués avant la fusion sont indépendants l'un de l'autre et il est donc facile de les faire exécuter en parallèle par 2 threads. Coder une version multi-tâche de cet algorithme, qui utilise uniquement des join() pour synchroniser ces 2 threads fils et leur père.

2. Coder une autre version multi-tâche de cet algorithme en utilisant cette fois-ci des wait()-notify().

Exercice 4 : Un sémaphore est un moyen pour garantir un accès limité à une ressource. Ce sémaphore modélisé par une classe Semaphore possède un attribut entier limite désignant le nombre d'accès simultané possible à une ressource. Cette classe contiendra les méthodes acceder() et liberer(). La première méthode sera autorisée par un certain nombre de threads puis bloquera lorsque le nombre de threads aura atteint la limite. La deuxième méthode libérera un accès à la ressource. Afin de tester cette classe, créer une classe Calculateur qui représente une machine disposant d'un certain nombre de processeurs. Elle possède une méthode unique calcul() qui est bloquante lorsqu'il n'y a plus de processeur disponible. On a n threads (classe Personne) représentant des personnes employant le calculateur par la méthode calcul().

1. Simuler l'usage d'un calculateur avec 7 processeurs par 11 personnes.

Chapitre 3

Les sockets UDP/TCP et leurs mise en œuvre en Java

3.1 Introduction

Le package *java.net* offre les moyens de communication entre machines à travers les réseaux, et notamment à travers Internet. Il offre :

- la manipulation des adresses Internet.
- La communications en mode connecté (en utilisant les sockets TCP(Transmission Control Protocol)).
- La communications en mode non connecté (en utilisant les sockets UDP (User Datagram Protocol)).

Avec des sockets TCP (sockets de flux), un processus établit une connexion avec un autre processus. Pendant qu'une connexion est en place, les données affluent entre les processus dans un flux contenu. Avec les sockets UDP (sockets de datagramme), il est possible de transmettre des paquets individuels d'informations.

Dans ce chapitre nous décrivons de manière général le modèle client-serveur. Ensuite, noua détaillons la mise en œuvre en java des sockets UDP et TCP.

3.2 Le modèle client-serveur

Lorsqu'on écrit une application réseau, il est courant de parler de clients et de serveurs. La distinction est de plus en plus vague, mais celui qui initie la communication est généralement le client. De l'autre côté, celui qui accepte la requête est le serveur.

3.2.1 Définition

Dans une acception plus restrictive, est appelé client-serveur un modèle de fonctionnement logiciel dans lequel plusieurs programmes autonomes communiquent entre eux par échange de messages plutôt que par mémoire partagée. Le premier transmet une requête d'exécution du service chez le serveur en donnant le nom du service souhaité et les paramètres associés. Le second est envoyé par le serveur et contient le résultat de l'exécution du service.

3.2.2 Caractéristiques

Le modèle client-serveur garantit la protection mutuelle du client et du serveur par la séparation de leurs espaces d'adressage. Il permet également de localiser dans un serveur une fonction partagée par plusieurs clients. Le client et le serveur ne sont pas identiques, ils forment un système coopératif :

- Pour le client, un serveur se présente sous la forme d'une boite noire sur laquelle il ne possède pas d'information concernant sa mise en œuvre. Par contre les services que rend le serveur peuvent être connus du client par leur nom, les paramètres à fournir et les paramètres qui lui seront rendus après exécution du service.
- Les parties client et serveur de l'application peuvent s'exécuter sur des systèmes différents.

- Une même machine peut implémenter les côtés client et serveur de l'application via l'interface de loopback, représentée par convention par l'adresse IP 127.0.0.1.
- Un client a besoin de deux informations pour trouver et se connecter à un serveur : un nom de machine et un numéro de port.
  - Le numéro de port est un identificateur qui permet de distinguer les différents clients et les serveurs qui fonctionnent sur une même machine.
  - Le serveur doit utiliser un numéro de port fixe vers lequel les requêtes clientes sont dirigées.
  - Les ports inférieurs à 1024 sont réservés ("well-known ports"). Les ports numérotés de 0 à 511 sont les "well known ports" de l'architecture TCP/IP. Ils donnent accès aux services standard de l'interconnexion : transfert de fichiers (FTP port 21), terminal (Telnetport 23), courrier (SMTP port 25), serveur web (HTTP port 80). De 512 à 1023, on trouve les services Unix.
  - Les clients utilisent un port quelconque entre 1024 et 65535 à condition que le triplet <transport/@IP/port> soit unique.

3.2.3 Schéma classique d'organisation du serveur

Le schéma classique d'organisation du serveur est celui d'un processus cyclique qui a trois tâches essentielles :

- Recevoir, trier et conserver les requêtes avant leur exécution.
- Extraire une requête et exécuter le service demandé.
- Envoyer la réponse au client.

L'exécution proprement dite des services peut être réalisée de façon itérative (on parle de serveur itératif) ou concurrente (on parle de serveur concurrent ou multiprogrammé) :

- Avec les serveurs itératifs, les requêtes sont traitées les unes après les autres par un seul thread. Ces serveurs sont plus simples à mettre en œuvre. Cependant, les temps de réponses de ces serveurs itératifs sont plus importants.
- Lorsque le serveur peut servir plusieurs clients, il est intéressant de l'organiser comme une famille de processus coopérants pour permettre l'exécution concurrente de plusieurs requêtes et exploiter ainsi un multi-processus ou des entrées-sorties simultanées. Le schéma classique comporte un processus cyclique, le veilleur (daemon), qui attend les demandes des clients. Lorsqu'une demande arrive, le veilleur active un processus exécutant qui réalise le travail demandé. Les exécutants peuvent être créés à l'avance et constituer un pool fixe, ou être créés à la demande par le veilleur en utilisant des processus ou des threads.

3.2.4 Mise en œuvre du modèle client-serveur

Un schéma client-serveur peut être mis en œuvre soit par :

- des opérations de bas niveau en utilisant de primitives du système de communication (sockets UDP/TCP).

– des opérations de haut niveau en considérant le serveur d'application (appelé aussi middleware). L'échange des messages entre clients et serveurs peut être en mode synchrone (requête/réponse) ou en mode asynchrone (via des fils d'attente). Java RMI et CORBA sont des exemples de plates-formes qui offrent un modèle de programmation de type client-serveur.

3.3 Manipulation des adresses

Les machines connectées sur Internet peuvent être identifiées par une adresse de 32 bits. Cette adresse pour le protocole IPv4 est sous la forme de quatre nombres compris entre 0 et 255 et séparés chacun par un point. Chacun de ces octets appartient à une classe selon l'étendue du réseau. Le serveur DNS (Domaine Name Service) est capable d'associer un nom à une adresse IP.

3.3.1 Adresses Internet

Les adresses Internet sont représentées en jJava par la classe *InetAddress*. Cette classe ne possède pas de constructeur (il faut utiliser certaines méthodes statiques de la classe pour obtenir une instance de cette classe).

3.3.1.1 Principales méthodes pour manipuler des adresses Internet

Méthode	**Description**
InetAddress getLocalHost()	Rend en résultat l'adresse de la machine locale (celle où s'exécute couramment le programme).
InetAddress getByName(String Machine)	rend en résultat l'adresse associée à la chaîne de caractères *Machine*. Cette chaîne peut être soit la représentation d'une adresse de 32 bits, par exemple "131.254.52.9", soit un nom en clair "poseidon.ifsic.univ-bejaia.dz".
InetAddress getAllByName(String Host)	Renvoie un tableau des adresses Internet associées au nom d'hôte fourni en paramètre.
String getHostName()	Retourne le nom de la machine associée à une adresse.
byte[] getAddress()	Renvoie un tableau contenant les 4 nombres de l'adresse Internet.
String getHostAddress()	Renvoie l'adresse Internet sous la forme d'une chaîne de caractères.

TABLE 3.1 – Principales méthodes pour manipuler des adresses Internet

Si l'hôte est inconnu ou que le serveur de noms de domaine est inopérant, le constructeur générera une *UnknownHostException*. Les autres causes d'échec, qui déclenchent l'envoi d'une *IOException* sont multiples : machine cible refusant la connexion sur le

port précisé ou sur tous les ports, problème lié à la connexion Internet, erreur de routage des paquets, etc.

3.3.2 Adresses de sockets

La classe abstraite *java.net.SocketAddress* représente un couple constitué de l'adresse d'une machine et d'un numéro de port, indépendamment du protocole utilisé. Ces couples sont principalement utilisés pour préciser l'adresse des objets sockets utilisés par les applications pour communiquer via le réseau. Trois constructeurs permettent de créer des instances de la classe *InetSocketAddress*.

- Le plus général accepte en argument une instance de la classe *InetAddress* et un numéro de port : si la référence à l'adresse Internet fournie vaut *null*, l'adresse de socket ainsi créée correspond à l'adresse Internet non spécifiée.
- Un second constructeur accepte seulement un numéro de port et utilise également l'adresse Internet non spécifiée.
- Enfin, un troisième constructeur accepte un nom de machine (qui peut éventuellement être une adresse IP sous la forme d'une chaîne de caractère) et un numéro de port : il tente alors de résoudre le nom auprès du service de nommage (DNS) afin de déterminer l'adresse Internet correspondante.

Dans le cas où cette résolution échoue, l'adresse de socket est tout de même construite, mais est dite " non résolue " (unresolved) et marquée comme telle. La méthode *isUnresolved()* permet, ultérieurement, de tester ce statut pour savoir si une adresse de socket identifie effectivement une adresse Internet connue.

Remarque : Pour ces trois constructeurs, le numéro de port attendu est de type *int*, mais doit être compris entre 0 et 32535, sous peine de lever une exception de la classe *IllegalArgumentException*. De plus, spécifier un numéro de port à 0 indique que n'importe quel port libre de la machine pourra être utilisé à chaque fois que l'adresse de socket sera sollicitée pour une opération d'attachement. Le port effectif d'attachement pourra alors être récupéré auprès de la socket attachée.

3.3.2.1 Principales méthodes pour manipuler des adresses de socket

- La classe *InetSocketAddress* fournit des méthodes d'observation : étant donnée une instance, la méthode *getAddress()* retourne l'objet de la classe *InetAddress* correspondant, ou *null* si elle est non résolue.
- La méthode *toString()* est redéfinie dans cette classe pour représenter une adresse de socket par l'adresse IP et le numéro de port, séparés par " : ".

3.4 Les sockets

Les sockets (prises de raccordement) forment un mécanisme de communication bidirectionnel interprocessus dans un environnement distribué qui permettent de construire

des applications distribuées selon le modèle client/serveur. Elles peuvent communiquer entre elles en s'interconnectant à travers un réseau. Une communication de réseau peut se réaliser en échangeant des données de messages transmis entre des sockets. Les messages sont mis en queue au niveau de la socket transmetteur jusqu'à ce que le protocole du réseau les expédie. A leur arrivée, les messages sont mis en file au niveau de la socket réceptrice jusqu'à ce que le processus de réception les traite.

3.4.1 Définitions

La socket est :
- une interface de programmation de bas niveau pour la communication réseau.
- un point d'entrée de communication bidirectionnelle entre deux applications sur un réseau.
- outil de communication (ports de communication, interfaces) pour échanger des données entre un (des) client(s) et un serveur.
- un point d'entrée aux couches réseau TCP/UDP qui est liée localement à un port.
- un canal de communication par lequel un processus peut envoyer ou recevoir des données.
- Etc.

Remarques :
- Une socket ne peut se connecter qu'à une seule machine.
- Une socket ne peut pas être se reconnecter après la fermeture de connexion.
- Java intègre nativement les fonctionnalités de communication réseau au-dessus de TCP-UDP/IP.

3.4.2 Sockets UDP

Le langage java propose une API (Application Program Interface) standard d'accès au protocole UDP qui fait partie du paquetage *java.net*. Deux classes principales concernent le protocole UDP. La classe *DatagramPacket* permet de manipuler les datagrammes (contenant les données transportées) et la classe *DatagramSocket* encapsule l'accès aux primitives d'émission et de réception du protocole UDP.

3.4.2.1 DatagramPacket

Les objets datagrammes, instances de la classe *DatagramPacket*, encapsulent trois objets : une zone mémoire, une adresse Internet et un numéro de port UDP. Tandis que la zone mémoire est représentée par un tableau d'octets, un indice de début et une longueur, l'adresse Internet et le numéro de port peuvent être regroupés sous la forme d'une adresse de socket, dans laquelle le numéro de port représente un port UDP.

1. **Constructeurs**
   Pour créer un objet datagramme, des constructeurs sont disponibles, qui permettent d'initialiser plus ou moins finement chacun des objets représentant les paramètres du datagramme.

Constructeur	**Description**
DatagramPacket (byte[] buf, int offset, int length, InetAddress, int destport)	*offset* est l'indice du premier octet à prendre en compte dans le tableau *buf*, et *length* représente le nombre d'octets à envoyer (en général, length = taille de buf).
DaragramPacket (byte[] buf, int offset, int length, SocketAddress sockAddr)	*sockAddr* est une instance de *InetSocketAddress* correspondant aux paramètres *destAddr* et *destPort*.
DatagramPacket(byte[] buf, int lng)	Ce constructeur ne précise pas les valeurs de *offset* et/ou celles de *destAddr* et de *destPort*. La valeur par défaut de l'argument *offset* est 0 : s'il est omis, le début des données coïncide avec le début du tableau.

TABLE 3.2 – Constructeurs de la classe DatagramPacket

2. **La taille maximale des datagrammes**
   Le protocole UDP permet d'émettre ou de recevoir des datagrammes dont la taille peut atteindre 2^{16} octets. Les méthodes *setReceiveBufferSize()* et *setSendBufferedSize()* de la classe *DatagramSocket* demandent au système de mettre en place des tailles autres que celles affectées par défaut. Le système peut ne pas accepter cette demande. Les méthodes *getSendBufferSize()* et *getReceiveBufferSize()* retournent normalement des valeurs effectives pour un objet socket. Les datagrammes dont la taille dépasse celle des tampons sont perdus.

3. **Méthodes**
   La classe DatagramPacket fournit des méthodes pour modifier un objet datagramme déjà construit ou pour compléter sa définition.

Méthode	**Description**
InetAddress getAddress ()	Renvoie l'adresse du serveur.
setAddress()	Modifie l'adresse Internet.
int getPort()	Retourne le port stocké dans le paquet.
setPort()	Modifie le port de destination.
byte[] getData()	Renvoie les données contenues dans le paquet.
setData(byte[])	Met à jour les données contenues dans le paquet.
int getLength ()	Renvoie la taille des données contenues dans le paquet.

TABLE 3.3 – Principales méthodes de la classe DatagramPacket

3.4.2.2 DatagramSocket

Une fois le datagramme est prêt, il faut disposer d'un objet représentant la socket locale pour envoyer ce datagramme. Cet objet est obtenu via l'un des constructeurs de la classe *DatagramSocket*.

Constructeur	**Description**
DatagramSocket()	Crée une socket associée à un numéro de port disponible décidé par le système.
DatagramSocket (int localPort)	Crée une socket en imposant le numéro de port indiqué.
DatagramSocket (int localPort, InetAddress localAddr)	Si *localPort* a pour valeur 0, l'un des ports libres et d'accès autorisé de la machine est choisi pour l'attachement de la socket. Une valeur non nulle de *localeAddr* permet de choisir une adresse Internet particulière de la machine locale.
DatagramSocket (SocketAddress sockAddr)	Une valeur *null* passé pour l'argument *sockAddr* de ce constructeur a pour effet de produire une socket dite " non attachée ". elle sera ultérieurement attachée à une adresse de socket (qui doit être locale) avant de pouvoir être utilisée, soit explicitement, grâce à la méthode *bind()* qui accepte une adresse de socket de type *SocketAddress*, soit implicitement à l'adresse Internet non spécifiée et à un port libre.

TABLE 3.4 – Constructeurs de la classe DatagramSocket

Après la construction d'un objet socket, il est possible d'obtenir des informations sur l'attachement local à la machine avec les méthodes *getLocalPort()*, *getLocalAddress()* ou *getLocalSocketAddress()*.

Méthode	**Description**
send(DatagramPacket p)	Envoie le paquet p.
receive(DatagramPacket p)	Range le prochain paquet reçu dans l'objet p.
close()	Ferme la socket et libère le port à laquelle elle était liée.
setSoTimeout()	Précise un timeout d'attente en millisecondes pour la réception d'un message.
int getPort()	Renvoie le port associé à la socket.

TABLE 3.5 – Principales méthodes de la classe DatagramSocket

3.4.2.3 Communication unicast

Une fois l'objet datagramme construit, il peut être utilisé par une socket UDP, instance de la classe *DatagramSocket* de deux façons :

– Soit en émission, avec la méthode *send()*.

– Soit en réception, avec la méthode *receive()*.

1. **Emettre un datagramme**
   Pour émettre un datagramme UDP, il faut commencer par créer l'objet datagramme, de la classe *DatagramPacket*, dans lequel sont précisés :
   – Les données à envoyer qui doivent être contenues dans un tableau de type *byte*.
   – L'identifiant de l'application destinataire des données, spécifiant l'adresse Internet de la machine et le numéro de port de l'application. Ces informations peuvent être fournies séparément, sous la forme d'un objet de la classe *InetAddress* et d'un entier de type *int*, ou bien regroupées sous la forme d'une adresse de socket de la classe *SocketAddress*.

   Ces valeurs peuvent être précisées au moment de la construction du datagramme, en les passant en argument du constructeur.

2. **Recevoir un datagramme**
   Pour recevoir un datagramme UDP, il faut bien entendu disposer d'un objet datagramme. La seule chose à initialiser dans cet objet est la zone mémoire dans laquelle vont être stockées les données reçues. Cela peut être réalisé au moment de la construction de l'objet, ou plus tard, via les méthodes *setData()* et *setLength()*.

```java
import java.net.* ;
public class SendUDP {
public static void main (String [] args){
 if (args.length != 3) {
    System.out.println("Usage : java localhost "
+ " <addr> <port> <msg> "); System.exit(1); }

 try{ InetAddress addr =
                InetAddress.getByName(args[0]);
      int port = Integer.parseInt(args[1]);
      byte [] buf = args[2].getBytes();
      DatagramPacket packet = new
 DatagramPacket (buf, 0, buf.length, addr, port);
 DatagramSocket socket = new DatagramSocket();
         System.out.println("Socket attachée
localement à" + socket.getLocalSocketAddress());
         socket.send(packet) ;
         System.out.println("Message envoyé à " +
socket.getLocalSocketAddress()) ;
 }catch(Exception e){
  System.err.println("Problème à l'exécution :");
  e.printStackTrace(System.err);
 }
}
}
```

```java
import java.net.* ;
public class ReceiveUDP {
final static int BUF_SIZE = 1024 ;
public static void main (String [] args){
 if (args.length != 0) {
     System.out.println("Usage : java localhost " + " <addr>
<port> <msg> ") ;   System.exit(1) ; }

try{ byte [] buf = new byte[1024];
DatagramPacket packet = new DatagramPacket (buf, 0, buf.length);
  int port = Integer.parseInt(args[0]) ;
  DatagramSocket socket = null ;
  if (args.length > 1)
       socket= new DatagramSocket(port,
InetAddress.getByName(args[1]));
    else
      socket= new DatagramSocket(port) ;
         System.out.println("Attente de réception sur" +
socket.getLocalSocketAddress()) ;
  while (true) {
    socket.receive(packet) ;
    String s = new String (packet.getData(), packet.getOffset(),
packet.getLength()) ;
        System.out.println("Message reçu de " +
packet.getSocketAddress() + "\n\t" + s) ;
    packet.setData(buf, 0, buf.length) ;
 }
}catch(Exception e){
                 System.err.println("Problème à l'exécution :");
                 e.printStackTrace(System.err); }}}
```

FIGURE 3.1 – Programme-3.1 : Client-serveur en mode non connecté

3.4.2.4 UDP et la concurrence

Les serveurs sont souvent écrits de façon à permettre le traitement de plusieurs requêtes de clients en parallèle. Pourtant, il est rare de trouver des serveurs concurrents utilisant des sockets UDP, car le démultiplexage de plusieurs communications sur le même port UDP pose problème. En effet, supposons que les requêtes de deux clients aient entrainé la création de deux threads concurrents, comment savoir alors, lorsqu'un nouveau datagramme est reçu, s'il correspond à un client déjà connu et à quel thread il est destiné. Une solution simple, utilisée dans l'implantation du protocole TCP, consiste à associer à un même thread tous les messages provenant de la même adresse Internet et du même port UDP. Pour gérer cette distribution, un processus léger est dédié à la répartition des datagrammes reçus aux différents processus légers de traitement.

3.4.2.5 Le broadcast UDP

En java, le broadcast UDP est réalisé, comme pour la communication unicast grâce aux classe *DatagramSocket* et *DatagramPacket* mais en utilisant une adresse de broadcast comme adresse de destination des datagrammes à diffuser. En général, l'émission des broadcasts est autorisée par défaut pour les sockets UDP. Il est possible de contrôler cette autorisation au moyen de la méthode *setBroadcast()* qui accepte un booléen indiquant si l'on veut (true) ou non (false) autoriser l'émission. Le statut d'autorisation est consultable avec la méthode *getBroadcast()*. En ce qui concerne la réception, une socket ne peut, à priori, recevoir des broadcasts que si elle est attachée à l'adresse Internet non spécifiée (*wildcard*, est l'adresse 0.0.0.0 en IPv4).

3.4.2.6 Le multicast UDP

Un groupe de multicast UDP est défini par une adresse Internet de multicast IP (une adresse IP de classe D de 224.0.0.1 à 239.255.255.255) et un numéro de port UDP donné. Pour qu'une application participe à un groupe de multicast UDP, il faut donc qu'elle dispose de l'adresse de multicast IP sous-jacente (correspondant à un groupe de multicast IP auquel l'interface qui l'héberge participe) et qu'elle attache une socket UDP sur le port définissant le groupe de multicast UDP. Toutes les sockets du groupe de multicast seront attachées au même port UDP.

- En java, la classe *MulticastSocket* permet de programmer des applications multicast. Cette classe est une sous-classe de la classe *DatagramSocket*. Le plus simple des constructeurs permettant de créer un objet socket multicast n'a pas d'argument. Il attache la socket multicast à un numéro de port libre et à l'adresse Internet non spécifiée. Un port particulier peut être spécifié en argument d'un constructeur surchargé. Enfin, un dernier constructeur accepte un objet de la classe *SocketAddress* permettant de spécifier une adresse Internet et un port particuliers (si cet argument vaut *null*, la socket n'est pas attachée).
- En tant que sous-classe de la classe *DatagramSocket*, une socket multicast peut être utilisée comme celle-ci pour envoyer des datagrammes. En revanche, avant

de pouvoir recevoir des datagrammes du groupe de multicast, la socket multicast doit rejoindre un groupe de multicast IP. Pour cela, il faut, appeler la méthode *joinGroup()* avec, en argument, une adresse Internet de multicast. La méthode *leaveGroup()* permet à la socket de quitter un groupe de multicast particulier, dont l'adresse Internet est passée en argument. Si la socket ne fait pas partie de ce groupe, une exception *SocketException* est levée.

3.4.3 Sockets TCP

La procédure d'établissement de connexion est dissymétrique. Un processus, appelé serveur, attends des demandes de connexion qu'un processus, appelé client, lui envoie. Une fois l'étape d'établissement de connexion effectuée le fonctionnement redeviens symétrique. Il est à noter que côté serveur on utilise deux sockets : l'un, appelé socket d'écoute, reçoit les demandes de connexion et l'autre, appelé socket de service, sert pour la communication. En effet, un serveur peut être connecté simultanément avec plusieurs clients et dans ce cas on utilisera autant de sockets de service que de clients.

Deux classes interviennent :

– java.net.Socket(côté client et côté serveur).
– java.net.ServerSocket(côté serveur).

3.4.3.1 Socket client

Pour construire la socket, il faut appeler l'un des constructeurs de la classe *Socket* (voir la table 3.6).

Constructeur	**Description**
protected Socket()	Crée une socket non connecté.
Socket(InetAddress address, int port)	Crée une socket, et le connecte au numéro de port spécifé et à l'adresse indiquée.
Socket(InetAddress host, int port, boolean stream)	Utiliser à la place DatagramSocket.
Socket (InetAddress address, int port, InetAddress localAddr, int localPort)	Crée une socket et la connecte à l'adresse distante et sur le port distant indiqués.
Protected Socket (SocketImpl)	Crée une socket non connecté en utilisant la classe abstraite *SocketImpl*.
Socket (String host, int port)	Crée une socket, et la connecte sur le numéro de port indiqué de l'hôte nommé.
Socket (String host, int port, boolean stream)	Utiliser à la place DatagramSocket.
Socket (String host, int port, InetAddress localAddr, int localPort)	Crée une socket et la connecte sur le port de l'hôte distant indiqués.

TABLE 3.6 – Constructeurs de la classe Socket

Pour établir une connexion TCP, un client doit construire sa socket locale, instance

de la classe *Socket*, l'attacher, si le constructeur ne l'a pas déjà fait, à une adresse Internet et un port TCP locaux, puis demander sa connexion à la socket d'un serveur.

Pour attacher la socket à une adresse Internet et un port TCP locaux, il faut appeler la méthode *bind()* qui prend en argument une adresse de socket de la classe *SocketAddress*. Si cet argument est *null*, la socket est attachée à une adresse valide et un port TCP libre. Sinon, l'argument doit être de la classe *InetSocketAddress*. Si l'adresse Internet encapsulée dans cette adresse de socket est nulle, la socket est attachée à un port TCP libre et sinon, la socket est attachée à l'adresse Internet et au port TCP encapsulés.

Les raisons d'un échec d'attachement d'une socket à l'adresse demandée sont :

- Le port est réservé ou déjà utilisé : une exception de la classe *BindException* est levée.
- La socket est déjà attachée : une exception de la classe *SocketException* est levée. La méthode *isBound()* permet de savoir si la socket est déjà attachée.

Pour tenter de connecter la socket, il faut appeler la méthode *connect()* de la socket locale en lui passant en argument l'adresse de la socket du serveur à laquelle se connecter. La méthode connect() est surchargée afin d'accepter en second argument un délai maximum d'attente pour la connexion TCP. Si le délai expire avant l'établissement de la connexion, la méthode connect() lève une exception de la classe *SocketTimeoutException*.

Les autres méthodes qui permettent de manipuler les objets *Socket* sont :

Méthode	**Description**
InputStream getInputStream()	Renvoie un flux en entrée pour recevoir les données de la socket.
OutputStream getOutputStream()	Renvoie un flux en sortie pour émettre les données de la socket.
void shutDownInput()	Fermeture de la socket pour la lecture.
void shutDownOutput	Fermeture de la socket pour l'écriture.
getLocalAddress()	Rend l'adresse IP à laquelle la socket est connectée.
InetAddress getInetAddress()	Retourne l'adresse de la machine à laquelle on est connecté.
int getLocalPort()	Retourne le numéro de port local.
int getPort()	Renvoie le port utilisé par la socket.
void close()	Ferme la socket.

TABLE 3.7 – Principales méthodes de la classe Socket

Remarque : La plupart des methodes et les constructeurs peuvent générer une *IOException*.

La classe Socket définit un canal de transmission entre machines. Après avoir créé deux objets de type *Socket*, un sur chacun des sites communicants, et après les avoir

connectés, les communications se font au moyen de canaux d'entrées/sorties séquentielles de type *InputStream* et *OutputStream*. Ces deux classes sont définies dans le paquetage *java.io*). Elle sont des sous classes de *Object*.

InputStream, *OutputStream* et leurs classe dérivées manipulent des octets : Les flots d'entrée et de sortie de la connexion TCP vers la socket du serveur sont récupérables par les méthodes *getInputStream()* et *getOutputStream()* de l'objet socket. Ces deux méthodes retournent des flots d'octects. Les méthodes d'entrées/sorties les plus courantes sont : *read()* et *write()*.

Remarque : Les classes *InputStream*, *OutputStream*, *Reader* et *Writer* sont abstraites (i.e. qu'elles définissent des méthodes communes à tout leurs héritiers). *Reader*, *Writer* et leurs classes dérivées manipulent des caractères et les méthodes dentrées/sorties les plus courantes sont : *readLine()* et *println()*.

3.4.3.2 Sockets serveurs

La classe *ServerSocket* possède plusieurs constructeurs comme :

Constructeur	**Description**
ServerSocket()	Constructeur par défaut.
ServerSocket(port)	Crée une socket sur le port fourni en paramètre (le serveur enregistre son service sous le numéro de port fourni en paramètre).
ServerSocket (port, longueurFileAttente)	Crée une socket sur le port fourni en paramètre. *longueurFileAttente* : spécifie un nombre maximal de clients qui peuvent demander une connexion au serveur.

TABLE 3.8 – Constructeurs de la classe ServerSocket

Les principales méthodes de la classe *ServerSocket* sont :

Méthode	**Description**
Socket accept()	Attente de connexion d'un client distant. Quand la connexion est faite, elle retourne une socket permettant de communiquer avec le client. Cette methode est dite bloquante ce qui implique un type de programmation particulier (boucle infinie qui se termine seulement si une erreur grave se produit).
setSoTimeout(inttimeout)	Positionne le temps maximum d'attente de connexion sur un *accept()*. Si temps écoulé, la méthode *accept()* lève l'exception *SocketTimeoutException* (par défaut, attente infinie sur l'accept.)
close()	Ferme la socket.

TABLE 3.9 – Principales méthodes de la classe ServerSocket

3.4.3.3 Programmes clients-serveur en mode connecté

La mise en place d'un serveur élémentaire en java requiert les étapes suivantes :

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

public class TCPSer {
public static void main (String arg[]){
 try {
  ServerSocket ss = new ServerSocket (1234);

 System.out.println("J'attends une
connexion") ;

 Socket s = ss.accept();
   InputStream is = s.getInputStream();
   OutputStream os = s.getOutputStream();

 System.out.println("J'attends un nombre") ;
         int nb = is.read() ;
System.out.println("Traitement sur le nombre
: " +nb) ;
          int res = nb*2 ;
          os.write(res) ;
System.out.println("Traitement sur le nombre
: " +nb) ;

 System.out.println("Réponse envoyée…. " ) ;
        ss.close() ;
}catch (Exception e) { e.printStackTrace();}

}
}
```

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

public class TCPCl{
public static void main (String arg [] ){
 try {
Socket s =new Socket ("localhost", 1234);

      InputStream is = s.getInputStream();
     OutputStream os = s.getOutputStream();

     Scanner clavier = new Scanner (System.in);
      int nb = clavier.nextInt();
         os.write(nb);
        int res = is.read();
 System.out.println("Le résultat est: " + res);

      s.close();

}catch (Exception e) {e.printStackTrace();}

}
}
```

FIGURE 3.2 – Programme-3.2 : Client-serveur en mode connecté

- le serveur enregistre son service sous un numéro de port et peut indiquer le nombre de clients qu'il accepte de faire bufférisér à un instant T (il crée un objet *Server-Socket*).
- Une fois le *ServerSocket* établi, le serveur écoute indéfiniment (ou se verrouille sur) les tentatives de connexion des clients. Ceci est assuré par un appel à la méthode *accept()* de *ServerSocket*.
- Après avoir créé deux objets de type *Socket*, un sur chacun des sites communicants, et après les avoir connectés, les communications se font au moyen de canaux dentrées/sorties séquentielles de type *InputStream* et *OutputStream*.
- La quatrième étape constitue la phase de traitement pendant laquelle le serveur et le client communiquent via les objets *OutputStream* et *InputStream*.
  - Envoi effectif sur le réseau des bytes : Les écritures ne provoquent pas nécessairement l'envoi immédiat de paquet TCP (La classe *PrintWriter* utilise un tampon et peut n'envoyer l'information que lorsque celui-ci est plein). Appeler la méthode *flush()* permet de demander l'envoi immédiat (utiliser TCP No delay)).

– A la cinquième étape, quand la transmission s'acheve, le serveur cloture la connexion en invoquant la méthode *close()* sur la socket.
On retourne à l'étape 2 pour récuperer le client suivant.

Remarque : Le serveur doit être demarré avant le client. Dans le cas contraire, si le client se connecte à un serveur inexistant, une exception sera levée après un time-out.

3.4.3.4 Programmation d'un serveur à plusieurs clients simultanés

Le serveur possède deux parties bien distinctes :
– Un thread principal qui utilise la *ServerSocket* et y attend des connexions. Sur réception d'une connexion, le serveur principal crée un nouveau thread qui gère uniquement cette connexion.
– Un thread auxiliaire pour chaque connexion.

Le client est un programme principal qui utilise la socket s'en sert pour se connecter au serveur.

3.4.4 Exceptions liées au réseau

Le package *java.net* définit plusieurs exceptions comme :

Constructeur	**Description**
BindException	Connection au port local impossible (le port est peut être déjà utilisé).
ConnectException	Connection à une socket impossible (aucun serveur n'écoute sur le port précisé).
NoRouteToHostException	Connection à l'hôte impossible (l'hôte n'est pas joignable : le routeur ne trouve pas le serveur ou le serveur est protégé par un firewall).
ProtocolException	Une erreur est survenue au niveau du protocle sous-jacent (problème de la configuration de la pile TCP/IP par exemple).
SocketException	Une erreur est survenue au niveau du protocle sous-jacent (problème générique de connexion).
SocketTimeoutException	Délai d'attente pour la réception ou l'émission des données écoulé.
UnknownHostException	L'adresse IP de l'hôte n'a pas pu être déterminée.

TABLE 3.10 – Exceptions liées au réseau

3.5 Série d'exercices

Exercice 1 :

1. Quelles informations sont-elles nécessaires à un client pour créer une DatagramPacket à destination d'un serveur.
2. Soit P une application qui permet un échange de messages entre seulement deux personnes via l'Internet. Quel protocole doit-on choisir (UDP ou TCP) ? Ecrire les algorithmes nécessaires (l'algorithme serveur et l'algorithme client).
3. Détailler, en 4 phases, le mécanisme Java de communication, côté client, entre un client et un serveur utilisant des sockets TCP.

Exercice 2 : On considère deux processus (S et C) connectés via une socket UDP (port 2022). Le processus C lit des chaînes au clavier et les transmet au processus S. S affiche les chaînes transmises.

1. Donner les codes correspondants lorsque S et C sont sur la même machine.
2. Réécrire ces codes en utilisant les sockets TCP.

Exercice 3 : Ecrire un programme qui permet de se connecter à un serveur web et d'afficher une page de ce serveur sous forme texte en utilisant les sockets TCP.

Exercice 4 : Ecrire une application client/serveur qui permet de savoir si une machine est active. Cette application utilise des sockets TCP.

- Le serveur attend une requête du client sur le port 1027 et lui envoie un message (par exemple contenant l'heure locale). Le serveur doit donc être capable de gérer plusieurs clients simultanément.
- Le client lit une adresse au clavier et envoie une requête au serveur de la machine correspondante pour savoir si elle est active. Dans l'affirmative, il affiche le message envoyé par le serveur.

Chapitre 4

Appel de procédure à distance (RPC) et Java RMI

4.1 Introduction

RMI (Remote Method Invocation) est fondé sur une technologie plus ancienne, que l'on appelle les appels de procédures distantes, ou RPC (Remote Procedure Calls). Le RPC permet à un programme procédural, d'appeler une procédure résidant sur un autre ordinateur, d'une manière tout aussi conviviale que si cette procédure faisait partie du même programme, fonctionnant sur le même ordinateur.

RMI repose aussi sur l'utilisation d'un middleware qui assure la communication entre des applications éventuellement distantes (ou également entre les entités distantes de l'application) en faisant abstraction de l'hétérogénéité des systèmes impliqués en termes d'architecture physique des processeurs, de système d'exploitation, de langage de programmation utilisé et de représentation des données.

Dans ce chapitre, nous présentons le principe de fonctionnement du RPC, puis nous illustrons le processus de développement d'une application RMI.

4.2 Appels de procédures à distantes (RPC)

L'abstraction procédurale est un concept fondamental de la programmation. Une procédure, dans un langage impératif, peut être vue comme une boîte noire qui remplit une tâche spécifiée en exécutant une séquence de code encapsulée, le corps de la procédure. L'encapsulation traduit le fait que la procédure ne peut être appelée qu'à travers une interface qui spécifié ses paramètres et ses résultats sous la forme d'un ensemble de conteneurs typés (les paramètres formels). Un processus qui appelle une procédure fournit les paramètres effectifs associés aux conteneurs, exécute l'appel, et reçoit les résultats lors du retour, c'est- à-dire à la fin de l'exécution du corps de la procédure.

Appel local	**Appel à distance**
– Appelant (client) et appelé (serveur) sont dans le même espace virtuel. – Même mode de pannes. – Appel et retour de procédure sont des mécanismes internes. – Considérés comme fiables (sauf aspects liés à la liaison dynamique de la procédure et à la vérification de la protection).	– Appelant (client) et appelé (serveur) sont dans deux espaces virtuels différents. – Pannes du client et du serveur sont indépendantes. – Pannes du réseau de communication (perte du message d'appel ou de réponse). – Temps de réponse du serveur peut être long (charge du réseau ou du site serveur).

Table 4.1 – Appel local Vs appel à distance

L'appel de procédure à distance peut être spécifié comme suit. Soit sur un site A un processus p qui appelle une procédure locale P (appel local). Il s'agit de concevoir un mécanisme permettant à p de faire exécuter P sur un site distant B (appel distant), en préservant la forme et la sémantique de l'appel (c'est- à-dire son effet global dans l'environnement de p). A et B sont respectivement appelés site client et site serveur,

car l'appel de procédure à distance suit le schéma client-serveur de communication synchrone par requête et réponse.

4.2.1 Principe de fonctionnement

La réalisation standard du RPC repose sur deux modules logiciels, la souche, ou talon, client (client stub) et la souche serveur (server stub). La souche client agit comme un représentant local du serveur sur le site client. La souche serveur joue un rôle symétrique. Les souches client et serveur échangent des messages via le système de communication. En outre, ils utilisent un service de désignation pour aider le client à localiser le serveur.

- Le client doit connaître l'adresse du serveur et le numéro de port de destination de l'appel. Ces informations peuvent être connues à l'avance et inscrites dans le programme. Néanmoins, pour assurer l'indépendance logique entre client et serveur, permettre une gestion souple des ressources, et augmenter la disponibilité, il est préférable de permettre une liaison tardive du client au serveur. Le client doit localiser le serveur au plus tard au moment de l'appel. Cette fonction est assurée par le service de désignation, qui fonctionne comme un annuaire associant les noms (et éventuellement les numéros de version) des procédures avec les adresses et numéros de port des serveurs correspondants.
- Un serveur enregistre un nom de procédure, associé à son adresse IP et au numéro de port auquel le veilleur (le thread daemon) attend les appels. Un client (plus précisément, la souche client) consulte l'annuaire pour obtenir ces informations de localisation. Le service de désignation est généralement réalisé par un serveur spécialisé, dont l'adresse et le numéro de port sont connus de tous les sites participant à l'application.
- La liaison inverse (du serveur vers le client) est établie en incluant l'adresse IP et le numéro de port du client dans le message d'appel.

Le thread veilleur attend les requêtes sur un port spécifié. Dans la solution séquentielle, le veilleur lui-même exécute la procédure (il n'y a pas d'exécution parallèle sur le serveur). Dans l'autre cas, un nouveau thread est créé pour effectuer l'appel. Le veilleur revenant attendre l'appel suivant et l'exécutant disparaît à la fin de l'appel. Pour éviter le coût de la création, une solution consiste à gérer une réserve de threads créés à l'avance. Ces threads communiquent avec le veilleur à travers un tampon partagé, selon le schéma producteur-consommateur. Chaque thread attend qu'une tâche d'exécution soit proposée. Quand il a terminé, il retourne dans la réserve et se remet en attente. Un appel qui arrive alors que tous les threads sont occupés est retarde jusqu' à ce que l'un d'eux se libère.

Les paramètres d'un appel nécessaires pour la génération de souches sont spécifiés dans une notation appelée langage de définition d'interface (IDL, Interface Definition Language). Une description d'interface écrite en IDL contient toute l'information qui définit l'interface pour un appel, et fonctionne comme un contrat entre l'appelant et l'appelé. Pour chaque paramètre, la description spécifié son type, son mode de passage

(valeur, copie-restauration, etc.) et les exceptions, indépendamment de tout langage de programmation. Des informations supplémentaires telles que le numéro de version et le mode d'activation du serveur peuvent être spécifiées.

4.2.2 Avantages et inconvénients

Un but du RPC était de permettre aux programmeurs de se concentrer sur les tâches obligatoires d'une application en appelant des procédures, tout en rendant transparent au programmeur le mécanisme qui permet aux parties de l'application de communiquer par l'entremise d'un réseau. Le RPC assure toutes les tâches de réseau et de distribution des données, c'est-à-dire l'empaquetage des arguments de procédures et des valeurs de retour pour la transmission via un réseau. Une application qui utilise le RPC est facilement portable d'un environnement à un autre, car elle est indépendante des protocoles de communication sous-jacents. Elle peut également être portée aisément, sans modifications, d'un système centralisé vers un réseau. Néanmoins, le maintien de la sémantique est une tâche délicate, pour deux raisons principales :

- les modes de défaillance sont différents dans les cas centralisé et distribué. Dans cette dernière situation, le site client, le site serveur et le réseau sont sujets à des défaillances indépendantes.
- Même en l'absence de pannes, la sémantique du passage des paramètres est différente (par exemple, on ne peut pas passer un pointeur en paramètre dans un environnement distribué car le processus appelant et la procédure appelée s'exécutent dans des espaces d'adressage différents).

Le RPC ne permet de transporter qu'un jeu limité de types de données simples. Par conséquent, le RPC n'est pas idéal pour le passage et le retour d'objets de java. Autre inconvénient du RPC est qu'il requiert de la part du programmeur l'étude et l'assimilation d'un langage de définition d'interface.

4.2.3 Quelques exemples d'environnements à base de RPC

On liste ci-après quelques environnements marquants :

- le RPC de Sun/ONC qui a été développé initialement pour la mise en œuvre du système de fichiers répartis NFS. Ce RPC, facile d'emploi, permet essentiellement de construire des appels de procédure entre stations Unix avec des programmes client et serveur écrits dans le langage C. Pendant très longtemps ce RPC a servi de référence pour l'écriture d'applications distribuées selon le modèle client-serveur.
- L'OSF (Open Software Foundation) a défini un RPC qui présente de nombreuses similarités avec celui de Sun. La contribution de l'OSF porte plus précisément sur l'environnement DCE (Distributed Computing Environment) qui fournit un ensemble de services distribués de base nécessaire pour la mise en œuvre d'applications réelles à grande échelle : service de désignation, service de gestion du temps, services de sécurité, service de gestion de fichiers répartis, etc. Tous ces services sont construits eux-mêmes en utilisant le RPC.

- l'OMG (l'Object Management Group) au travers de la spécification d'architecture CORBA a apporté une contributions significative au modèle client-serveur concerne l'utilisation des propriétés bien connues de l'objet (modularité, encapsulation, typage, héritage, etc.) pour décrire les services accessibles à distance.
- L'environnement DCOM (Distributed Component Object Model) disponible sur les diverses variations du système Windows de Microsoft offre une fonction similaire pour écrire des applications à base d'objets/composants COM interconnectés selon un modèle client-serveur. Le RPC disponible dans Windows est une évolution du RPC de l'environnement DCE.
- Java RMI a intégré au sein de la programmation objet la notion d'appel de procédure à distance. Il permet l'écriture d'applications client-serveur en java avec la possibilité de charger dynamiquement le code du talon client. Cette facilité permet à un client de se lier au serveur uniquement lors de l'appel.

4.3 RMI (Invocation de méthodes distantes)

L'appel de méthode à distance (ou RMI) est le nom donné à l'appel de procédure à distance lorsque le serveur est représenté par un objet distribué (distant). Dans ce cas les services disponibles sur le serveur sont définis par les méthodes (ou opérations) de l'objet. L'appel de méthode à distance est le mécanisme de base pour les applications distribuées écrites à l'aide du langage java.

4.3.1 Définitions

- RMI est middleware qui permet de réaliser des applications distribuées java à java, dans lesquelles les méthodes d'objets distribués peuvent être invoquées à partir d'autres machines virtuelles pouvant résider dans d'autres ordinateurs.
- RMI est un système d'objets distribués constitué uniquement d'objets java.
- RMI est une Application Programming Interface (intégrée au JDK 1.1 et plus) développé par JavaSoft.
- RMI est un ensemble de classes qui permettent la communication entre machines virtuelles Java (JVM) qui peuvent se trouver physiquement sur la même machine ou sur des machines distinctes (c.à.d. la manipulation des objets sur des machines distantes (objets distants) de manière similaire aux objets sur la machine locale (objet locaux)). Ces manipulations sont relativement transparentes.
- La technologie RMI se charge de rendre transparente la localisation de l'objet distribué, son appel et le renvoi du résultat. En fait, elle utilise deux classes particulières, le stub et le skeleton, qui doivent être générées avec l'outil rmic fourni avec le JDK.

4.3.2 Architecture RMI

L'architecture RMI définit la manière dont se comportent les objets, comment et quand des exceptions peuvent se produit, comment gérer la mémoire et comment les méthodes appelées passent et reçoivent les paramètres. Le système RMI contient trois couches qui sont indépendantes de l'autre :

- La couche d'application : La première couche est constituée par les modules client et serveur, on retrouve les appels de haut niveau pour exporter les objets, ou les accéder (le serveur qui expose les objets, et la manière d'accéder à ces objets et le client qui initialise les appels de haut niveau, et fait les demande d'utilisation des objets exposés.)
- La couche des amorces (stub/skelton) :
  - Le stub (côté client) est un mandataire de l'objet qui est chargé dans le client au moment de l'obtention de la référence. Cette référence implémente les mêmes interfaces que l'objet distribué. Il a pour but d'initier une connexion avec la JVM distante en transmettant l'invocation distante à la couche des références d'objets (RRL, Remote Reference Layer), assembler les paramètres arguments des méthodes distantes pour leur transfert à la JVM distante, attendre les résultats de l'invocation distante, désassembler la réponse ou l'exception renvoyée, et l'envoyer à l'appelant.
  - Skeletons (côté serveur) : Contient une méthode qui appelle les méthodes de l'objet distribué. Il a pour but de désassembler les arguments pour la méthode distante, faire un appel à la méthode demandée et assembler le résultat (réponse renvoyée ou exception) à destination de l'appelant.
- La couche des références d'objets : Cette couche est chargée du système de localisation afin de fournir un moyen aux objets d'obtenir une référence à l'objet distant. Elle est assurée par le package *java.rmi.Naming*. On l'appelle généralement registre RMI car elle référence les objets (*rmiregistry* s'exécute sur chaque machine hébergeant des objets distribués).
- La couche de transport : Cette couche permet d'écouter les appels entrants ainsi que d'établir les connexions et le transport des données sur le réseau par l'intermédiaire du protocole TCP. Les packages *java.net.Socket* et *java.net.SocketServer* assurent implicitement cette fonction. Les connexions et les transferts de données dans RMI sont effectués par Java sur TCP/IP grâce à un protocole propriétaire JRMP, (Java Remote Method Protocol) sur le port 1099.

4.3.3 Création d'une application distribuée en client-serveur avec RMI

Une application RMI est composée d'une partie client et d'une partie serveur. Le rôle d'un serveur est de créer des objets qu'on qualifie de distribués (ou distants), de les rendre accessibles à distance et enfin d'accepter des connexions de clients vers ces objets.

4.3.3.1 Développement côté serveur

RMI permet de rendre accessible à distance des objets java tout en cachant, d'un point de vue syntaxique, la répartition de l'application. La mise en œuvre de ces appels nécessite de :

1. **Définir l'interface distante**
   La première étape de la création d'une application distribuée en client-serveur avec le RMI consiste à définir l'interface distante qui décrit les méthodes distantes que le client utilisera pour interagir avec l'objet de serveur distant par l'entremise du RMI. Pour créer une interface distante, il faut définir une interface qui étend l'interface *Remote* (du package *java.rmi*). L'interface *Remote* est une interface de balisage : elle ne déclare aucune des méthodes, et ne place de ce fait aucune contrainte sur la classe d'implémentation. Un objet d'une classe qui implémente directement ou indirectement l'interface *Remote* est un objet distant et est accessible, selon les autorisations de sécurité, à partir de n'importe quelle machine java virtuelle qui a une connexion à l'ordinateur sur lequel l'objet distribué s'exécute. Les paramètres doivent être soit des types simples du langage java, soit des objets java sérialisables, soit des références vers des objets distants (i.e. des interfaces). Chaque méthode dans une interface distante *Remote* doit avoir une clause *throws* signalant le risque potentiel de *RemoteException*.
   Remarque : *RemoteException* est la classe mère d'une hiérarchie d'une vingtaine d'exceptions précisant un problème lors de l'appel de l'opération, comme, par exemple :
   – NoSuchObjectException : ne peut récupérer la référence sur l'objet distribué.
   – StubNotFoundException : un stub correct n'a pas été trouvé.
   – UnknowHostException : la machine distante n'a pas été trouvée.
   – AlreadyBoundException : le nom associé à l'objet est déjà utilisé.
   – ConnectException : connexion refusée par la partie serveur.
2. **Créer la classe de l'objet distribué**
   La deuxième étape de la création d'un objet distribué consiste à définir une classe qui implémente une interface qui étend *Remote*. La classe de serveur doit étendre la classe *UnicastRemoteObject*, qui fournit les fonctionnalités de base requises de tous les objets distribués. Le constructeur de la classe *UnicastRemoteObject* exporte l'objet afin de le rendre disponible pour des appels distants.
3. **Ecrire une classe pour instancier l'objet et l'enregistrer dans le registre**
   Ces opérations peuvent être effectuées dans la méthode *main()* d'une classe dédiée ou dans la méthode *main()* de la classe de l'objet distribué. La marche à suivre contient trois étapes :
   – Mettre en place d'un gestionnaire de sécurité (security manager) dédié qui est facultative. Ce gestionnaire (objet de la classe *RMISecurityManager*) autorisera le chargement des classes depuis une autre application. Sans security manager, il faut obligatoirement mettre à la disposition du serveur toutes les classes

dont il aura besoin (Elles doivent être dans le *CLASSPATH* du serveur). Si un gestionnaire de sécurité est mis en place, il faut définir un fichier qui va contenir la politique de sécurité qu'il doit mettre en oeuvre. Lors du lancement du serveur, l'option *java.security.policy* permet de préciser le fichier qui sera utilisé par le gestionnaire de sécurité.

- Instancier un objet de la classe distante : Cette opération est très simple puisqu'elle consiste simplement en la création d'un objet de la classe de l'objet distribué. L'objet distribué est un objet Java standard qui peut implémenter plusieurs interfaces distantes et d'autres interfaces locales. Il doit donner le code des méthodes pouvant être appelées à distance et celui des méthodes locales. L'implantation de l'objet distant hérite de j*ava.rmi.UnicastRemoteObject* et implante l'interface précédente.
- Enregistrement de l'objet distribué dans le registre de nom RMI en lui donnant un nom : Pour qu'un client puisse accéder à un objet distant, il est nécessaire que celui-ci soit enregistré dans un service de désignation. Le principe de désignation d'un serveur reprend la structure d'une URI (Uniform Resource Identifier), à savoir : *rmi ://nom_de_machine :port/nomObjetDistant*. Le nom de la machine et le port sont ceux qui sont associés au service de désignation de rmi (rmiregistry). 1099 est le numéro de port prédéfini auquel les clients et les serveurs se connectent pour accéder au rmiregistry. *nomObjetDistant* est le nom de l'objet distribué qui servira aux clients L'enregistrement des services est réalisé par l'intermédiaire de la classe *Naming* en appelant la méthode *rebind*. Cette méthode lie l'objet distribué au rmiregistry. Une autre méthode *bind* existe qui lie aussi un objet distant au registre. La méthode *rebind* sert plus fréquemment que *bind* parce qu'elle garantit que, si l'objet distribué est déjà inscrit avec son nom, une nouvelle inscription remplace la précédente, de même nom.

  Remarque : Sur le serveur, le registre de noms RMI doit s'exécuter avant de pouvoir enregistrer un objet ou obtenir une référence. Ce registre peut être lancé en tant qu'application fournie dans le JDK (rmiregistry) ou être lancé dynamiquement dans la classe qui enregistre l'objet. Ce lancement ne doit avoir lieu qu'une seule et unique fois. La méthode *createRegistry()* de la classe *java.rmi.registry.LocateRegistry* permet d'exécuter le registre. Cette méthode attend en paramètre un numéro de port.

4.3.3.2 Développement côté client

Le développement côté client repose sur :

1. **Mise en place d'un security manager**
   Comme pour le côté serveur, cette opération est facultative. Le choix de la mise en place d'un sécurity manager côté client suit des règles identiques à celui du côté serveur. Sans son utilisation, il est nécessaire de mettre dans le *CLASSPATH*

du client toutes les classes nécessaires dans la classe stub.

2. **Obtention d'une référence sur l'objet distribué à partir de son nom**
   Une fois que la méthode (ou le service) d'un objet java est inscrit comme disponible à distance, un client pour effectuer une recherche sur ce service et recevoir une référence qui l'autorise à exploiter ce service, ou, en d'autres termes, à appeler cette méthode. La méthode *lookup* de la classe *Naming*, exploitée par un client, lui permet d'obtenir une référence *Remote* à un objet distant.
3. **Appel à la méthode à partir de la référence sur l'objet distribué**
   Puisque cette méthode renvoie une référence *Remote*, le type de cette dernière doit être rétrogradé au type de l'interface *Remote* appropriée. Si le nom fourni dans l'URL n'est pas référencé dans le registre, la méthode lève l'exception *NotBoundException*.

   Trois mécanismes différents sont utilisés pour passer des arguments et en récupérer d'une méthode distante, selon le type de donnée passée :
   - Les types primitifs (*int*, *double*, *boolean*, etc.) sont passés par valeur, de la même manière qu'en appel de méthode locale java.
   - Les références à des objets distribués sont passées comme références distantes, qui permettent au récepteur d'appeler les méthodes de l'objet.
   - Les objets qui n'implantent pas l'interfaceRemote sont passés par valeur. C'est-à-dire que des copies complètes sont passées, en utilisant la sérialisation. Des objets qui ne sont pas sérialisables ne peuvent pas être passés à des méthodes distantes.

4.3.4 RMI et les threads

Un appel RMI est initié par un thread côté client mais éxecute par un autre thread côté serveur.

- Le thread du côté client est bloqué jusqu'à ce que le thread du côté serveur ait fini l'exécution.
- Un objet distant est par essence multithread.
- Pas de lien entre le thread client et le thread côté serveur.

```java
import java.rmi.*;
public interface Information extends Remote {public String getInformation() throws RemoteException;}
```

```java
import java.rmi.*;
import java.rmi.server.*;
public class TestRMIServer extends UnicastRemoteObject implements Information {
public TestRMIServer() throws RemoteException { super(); }
public String getInformation()throws RemoteException { return "Test RMI….."; }}
```

```java
import java.rmi.*;
import java.rmi.registry.LocateRegistry;
public class Serveur {
public static void main(String[] args){
try {
      int RMI_port = 1099;
      LocateRegistry.createRegistry(RMI_port) ;
      System.out.println("Mise en place du Security Manager ...");
      System.setSecurityManager(new java.rmi.RMISecurityManager());
      TestRMIServer testRMIServer = new TestRMIServer();
      System.out.println("Enregistrement du serveur");
      Naming.rebind("rmi://localhost:1099/TestRMI", testRMIServer);
      System.out.println("Serveur lancé");
} catch (Exception e) {System.out.println("Exception capturée: " + e.getMessage()); } }}
```

```java
import java.rmi.*;
public class Client { public static void main(String[] args) {
       System.setSecurityManager(new RMISecurityManager());
       try {
           Remote r = Naming.lookup("rmi://127.0.0.1/TestRMI");
           if (r instanceof Information) {
                           String s = ((Information) r).getInformation();
                           System.out.println("Information reçue : " + s);   }
    } catch (Exception e) {}}}
```

FIGURE 4.1 – Programme-4.1 : Client-serveur avec RMI

4.3.5 Avantages

Les principaux avantages à utiliser RMI sont les suivants :

- RMI est très simple à mettre en œuvre et les concepts utilisés sont faciles à comprendre.
- Il permet le transfert transparent d'objets d'une application à une autre et ces objets ne subissent pas de changement lors des transferts (l'integrite des objets transferés est respectee).
- RMI permet une gestion fine de la sécurité. Il bénéficie de la sécurité de java et d'un *RMISecurityManager* qui doit être paramétré pour gérer par exemple des listes d'accès.
- Il permet l'exécution en parallèle d'applications (locale ou distante).
- Java RMI est un mécanisme de base qui permet de construire d'autres modèles de programmation de plus haut niveau (Par exemple JavaSpace).

4.3.6 Limites

- Java RMI ne permet que l'appel de méthodes sur des objets java.
- Java RMI est lent et la sérialisation java est un mécanisme redoutable car on peut transporter de très gros volume de données sans s'en rendre compte.

4.4 Série d'exercices

Exercice 1 :

1. Citer les limites de RPC.
2. Quelle est la différence entre une communication synchrone et asynchrone entre objets. RMI utilise-t-il une communication synchrone ou asynchrone.
3. Réaliser une application distribuée de discussion qui permet d'échanger des messages entre le serveur et le client en utilisant RMI.

Exercice 2 : On veut mettre en place un objet RMI serveur, qui offre une méthode dont voici la signature long fact(int n). Cette méthode permet de calculer le factoriel d'un nombre entier positif n.

1. Ecrire le code du serveur RMI et celui de l'interface RMI.
2. Fournir le code d'un client RMI dont le but d'utiliser cette méthode offerte par le serveur.

Exercice 3 : Nous disposons d'un service qui représente un contrôleur de température TemperatureSensor qui offre les opérations de gestion de la température d'un système industriel. Les méthodes offertes par ce service sont les suivantes :

- void augmenterTemp (double tempVal) ;
- void diminuerTemp (double tempVal) ;
- double lire_temp () ;

1. On souhaite rendre chacune de ces méthodes accessibles à distance de manière à ce qu'elles définissent l'interface TemperatureSensorInterface entre le client et le serveur. Ecrire cette interface.
2. Ecrire la classe TemperatureSensor qui matérialise le service offrant les opérations augmenterTemp(), diminuerTemp() et lire_temp().
3. Ecrire le programme du serveur qui doit être installé sur une machine donnée.
4. Ecrire le programme du client qui permet d'invoquer ces méthodes.

Exercice 4 : L'objectif de l'exercice est de concevoir une application distribuée de gestion bibliographique. Le serveur maintient à jour une base de données dans laquelle sont référencés ces livres, caractérisés par leur titre (chaîne de caractères) et un numéro ISBN (numéro identifiant unique au niveau mondial pour tous les livres édités, représenté par un entier long). Cette base de données, sera représentée sous la forme d'un objet de type Hashtable. Les opérations autorisées sur cette base de données sont :

- L'ajout d'un nouveau livre caractérisé par son numéro ISBN et son titre.
- La suppression d'un livre, référencé grâce à son numéro ISBN.
- L'affichage de tous les livres de la base de données.

1. Quelles classes sont nécessaires ? Lesquelles doivent être distribuées ? Lesquelles peuvent être locales ?
2. Implanter l'interface Bibliotheque.java qui définit les opérations de gestion de la bibliothèque.
3. Ecrire la classe livre.java et la classe bibliothequeIMPL.java.
4. Ecrire le programme de serveur BibServeur.java.
5. Ecrire la classe BibClient.java.

Chapitre 5

Introduction à JMS (Java Message Service)

5.1 Introduction

Les modèles de communication asynchrones sont mieux adaptés que les modèles synchrones (de type client-serveur) pour gérer les interactions entre systèmes faiblement couplés. Le couplage faible résulte de plusieurs facteurs de nature spatiale ou temporelle : l'éloignement géographique des entités communicantes, la possibilité de déconnexion temporaire d'un partenaire en raison d'une panne ou d'une interruption de la communication (pour les usages mobiles par exemple). Les modèles de communication asynchrones prennent en compte l'indépendance entre entités communicantes et sont donc mieux armés pour traiter ces problèmes.

5.2 JMS (Java Messaging Service)

5.2.1 Définitions

- JMS est la spécification d'un service de messagerie en Java. Plus précisément JMS décrit l'interface de programmation pour utiliser un bus à messages asynchrone, c'est- à-dire la manière de programmer les échangés entre deux composants applicatifs.
- JMS est une API standard de Java EE (Enterprise Edition). Elle permet de fournir à java un intergiciel de communication par messages, en quelque-sorte un service de mail pour applications.

Pour pouvoir utiliser JMS, il faut disposer d'une implémentation externe en java. Il existe de nombreuses implémentations, tant commerciales qu'Open Source. Citons par exemple : WebSphere MQ d'IBM, Sonic MQ de Progress Software mais aussi Fiorano et Swift MQ pour les versions commerciales ; ActiveMQ, Joram, OpenJMS pour les versions Open Source.

5.2.2 Structure d'une application JMS

Dans la structure d'une application JMS, on distingue les composants suivants :

- Le système de messagerie JMS (JMS provider). Dans la mise en œuvre du système de messagerie on distingue le service de base qui met en œuvre les abstractions du modèle de programmation JMS et une bibliothèque de fonctions liée aux programmes utilisateurs qui met en œuvre l'interface de programmation JMS.
- Les clients JMS sont les programmes (applications), écrits en langage Java, qui produisent (émettent) et consomment (reçoivent) des messages selon des protocoles spécifiés par l'API JMS.
- Les messages JMS sont des objets qui permettent de véhiculer des informations entre des clients JMS. Différents types de messages sont utilisables : texte structuré (par exemple un fichier XML), format binaire, objets Java (sérialisés), etc.

5.2.3 Modes de communication

Deux modes de communication sont fournis par la spécification JMS :

5.2.3.1 Modèle de communication point à point

La communication point à point est fondée sur des files de message (Queues). Un message est adressé à une queue par un client producteur d'où il est extrait par un client consommateur. Un message est consommé par un seul client. Le message est stocké dans la queue jusqu'à sa consommation ou jusqu'à l'expiration d'un délai d'existence. La consommation d'un message peut être synchrone (retrait explicite par le consommateur ou asynchrone (activation d'une procédure de veille préenregistrée chez le consommateur) par le gestionnaire de messages. La consommation du message est confirmée par un accusé de réception généré par le système ou le client.
Le mode de communication en point à point est bien adapté à un workflow d'entreprise où les documents circulent avec un temps de traitement spécifique.

5.2.3.2 Modèle de communication Publish/Subscribe

La communication multipoints est fondée sur le modèle publication/abonnement (Publish/Subscribe). Un client producteur émet un message concernant un sujet prédéterminé (Topic). Tous les clients préalablement abonnés à ce Topic reçoivent le message correspondant. Le modèle de consommation (implicite/explicite) est identique à celui du mode point à point.
Le mode de communication par événement convient au contraire pour diffuser des alertes à plusieurs destinataires, par exemple des informations de bourse à des courtiers, ou bien des anomalies réseau en télécommunications, des points de synchronisation en vidéo, etc.

Remarque : La version 1.1 de la spécification JMS (dénotée JMS 1.1) unifie la manipulation des deux modes de communication au niveau du client JMS en introduisant la notion de Destination pour représenter soit une queue de message, soit un Topic. Cette unification simplifié l'API (les primitives de production/consommation sont syntaxiquement fusionnées) et permet une optimisation des ressources de communication. Elle permet également d'utiliser le mode transactionnel dans les deux modèles de communication. Bien entendu cela ne remet pas en cause la sémantique propre à chacun de ces modes de communication, qui doit être prise en compte dans la programmation du client JMS. Enfin, la spécification JMS définit des options de qualité de service, en particulier pour ce qui concerne les abonnements (temporaires ou durables) et la garantie de délivrance des messages (propriété de persistance).

5.2.4 Objets JMS

JMS définit un ensemble d'objets communs aux deux modèles de communication point à point et multipoints :

5.2.4.1 ConnectionFactory

JMS ne définit pas une syntaxe d'adressage standard. En lieu et place, il définit l'objet *Destination* qui encapsule les différents formats d'adresses des systèmes de messagerie. L'objet *ConnectionFactory* encapsule l'ensemble des paramètres de configuration définit par l'administrateur, il est utilisé par le client pour initialiser une connexion avec le système de messagerie.

5.2.4.2 Connection

Un objet *Connection* représente une connexion active avec le système de messagerie. Cet objet supporte le parallélisme, c'est à dire qu'il fournit plusieurs voies de communications logiques entre le client et le serveur. Lors de sa création, le client peut devoir s'authentifier. L'objet *Connection* permet de créer une ou plusieurs sessions. Lors de sa création une connexion est dans l'état stoppé. Elle doit être explicitement démarrée pour recevoir des messages. Une connexion peut être temporairement stoppée puis redémarrée ; après son arrêt, il peut y avoir encore quelques messages délivrés ; arrêter une connexion n'affecte pas sa capacité à transmettre des messages. Du fait qu'une connexion nécessite l'allocation de nombreuses ressources dans le système de messagerie, il est recommandé de la fermer lorsqu'elle n'est plus utile.

5.2.4.3 Destination

Cet objet désigne la destination des messages pour un producteur et la source des messages attendus pour un consommateur. La destination désigne une *Queue* de messages dans le modèle de communication point à point et un *Topic* dans le modèle de communication multipoints.

5.2.4.4 Session

Un objet *Session* est un contexte mono-thread pour produire et consommer des messages. Il répond à plusieurs besoins :
- Il construit les objets *MessageProducer* et *MessageConsumer*.
- Il crée les objets *Destination* et *Message*.
- Il supporte les transactions et permet de grouper plusieurs opérations de réception et d'émission dans une unité atomique qui peut être validée (resp. invalidée) par la méthode *Commit* (resp. *Rollback*).
- Il réalise l'ordonnancement des messages reçus et envoyés.

– Il gère l'acquittement des messages. Bien qu'une session permette la création de multiples objets *MessageProducer* et *MessageConsumer*, ils ne doivent être utilisés que par un flot d'exécution à la fois (contexte mono-thread). Pour accroître le parallélisme un client JMS peut créer plusieurs sessions, chaque session étant indépendante. Dans le mode publish/subscribe, si deux sessions s'abonnent à un même sujet, chaque client abonné (Topic Subscriber) reçoit tous les messages émis sur le *Topic*. Du fait qu'une session nécessite l'allocation de ressources dans le système de messagerie, il est recommandé de la fermer lorsqu'elle n'est plus utilisée.

5.2.4.5 Message Producer

Un client utilise un objet *MessageProducer* pour envoyer des messages à une destination. Un objet *MessageProducer* est créé appelant la méthode *CreateProducer* de l'objet *Session* avec un objet *Destination* en paramètre. Si aucune destination n'est spécifée un objet Destination doit être passé à chaque envoi de message (en paramètre de la méthode *Send*). Un client peut spécifier le mode de délivrance, la priorité et la durée de vie par défaut pour l'ensemble des messages envoyés par un objet *MessageProducer*. Il peut aussi les spécifier pour chaque message.

5.2.4.6 Message Consumer

Un client utilise un objet *MessageConsumer* pour recevoir les messages envoyés à une destination particulière. Un objet *MessageConsumer* est créé en appelant la méthode *CreateConsumer* de l'objet *Session* avec un objet *Destination* en paramètre. Un objet *MessageConsumer* peut être créé avec un sélecteur de message pour filtrer les messages à consommer. JMS propose deux modèles de consommation (synchrone et synchrone) pour les messages.

– Dans le modèle synchrone un client demande le prochain message en utilisant la méthode *Receive* de l'objet *MessageConsumer* (mode de consommation Pull).
– Dans le mode asynchrone, il enregistre au préalable un objet qui implémente la classe *MessageListener* dans l'objet *MessageConsumer*. Les messages sont alors délivrés lors de leur arrivée par appel de la méthode *onMessage* sur cet objet (mode de consommation Push).

Remarque : Les objets cités se déclinent selon le modèle de communication choisi, comme le montre la table 5.1.

	Point à point	**Publish/Subscribe**
Destination	Queue	Topic
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver	TopicSubscriber

TABLE 5.1 – Objets JMS

5.2.5 Messages JMS

Un message JMS est compose de trois parties : un en-tête, des propriétés et un corps.

5.2.5.1 En-tête des messages

Un en-tête de message JMS contient l'ensemble des données utilisées à la fois par le client et le système pour l'identification et l'acheminement du message. Il est composé des champs suivants :

- JMSDestination : Ce champ contient la destination du message, il est fixé par la méthode d'envoi de message en fonction de l'objet *Destination* spécifié.
- JMSDeliveryMode : Ce champ définit le mode de délivrance du message (persistant ou non) ; il est fixé par la méthode d'envoi de message en fonction des paramètres spécifiés.
- JMSMessageId : Ce champ contient un identificateur qui identifié de manière unique chaque message envoyé par un système de messagerie. Il est fixé par la méthode d'envoi de message et peut-être consulté après envoi par l'émetteur.
- JMSTimeStamp : Ce champ contient l'heure de prise en compte du message par le système de messagerie, il est fixé par la méthode d'envoi de message.
- JMSReplyTo : Ce champ contient la Destination à laquelle le client peut éventuellement émettre une réponse. Il est x e par le client dans le message.
- JMSExpiration : Ce champ est calculé comme la somme de l'heure courante (GMT) et de la durée de vie d'un message (time-to-live). Lorsqu'un message n'est pas délivré avant sa date d'expiration il est détruit ; aucune notification n'est définie pour prévenir de l'expiration d'un message.
- JMSCorrelationId, JMSPriority, JMSRedelivered, JMSType, etc.

5.2.5.2 Propriétés

En plus des champs standards de l'en-tête, les messages permettent d'ajouter des champs optionnels sous forme de propriétés normalisées, ou de propriétés spécifiques à l'application ou au système de messagerie. Les propriétés permettent

à un client JMS de sélectionner les messages en fonction de critères applicatifs. Un nom de propriété doit être de type *String* ; une valeur peut être : *nul*l, *boolean*, *byte*, *short*, *int*, *long*, *float*, *double* et *String*. Un client peut ainsi définir des filtres en réception dans l'objet *MessageConsumer* à l'aide d'une chaîne de caractère dont la syntaxe est basée sur un sous-ensemble de la syntaxe d'expression de conditions du langage SQL.

5.2.5.3 Corps du message

JMS définit différents types de corps de message afin de répondre à l'hétérogénéité des systèmes de messagerie :

- StreamMessage : Un message dont le corps contient un flot de valeurs de types primitifs java. Il est rempli et lu séquentiellement.
- MapMessage : Un message dont le corps est composé d'un ensemble de couples noms-valeurs. Le client reçoit la totalité du message, il peut n'extraire que la partie utile du message et ignorer le reste. Les paires sont positionnées au moyen de méthodes sur l'objet *Message* (l'ordre des mises à jour est quelconque).
- TextMessage : Un message dont le corps est une chaîne de caractère (*String*).
- ObjectMessage : Un message dont le corps contient un objet java sérialisable contenant l'ensemble des informations utiles. Le client utilise alors l'interface de cet objet pour obtenir l'information utile.
- BytesMessage : Un message dont le corps est composé d'un flot d'octets. Ce type de message permet de coder un message conformément à une application existante.

5.2.5.4 Construction de messages

Les messages sont créés au moyen de primitives de l'interface *Session* en fonction du type de message désiré : *createBytesMessage*, *createTextMessage*, *createMapMessage*, *createStreamMessage*, et *createObjectMessage*.

Point à point	**Publish/Subscribe**
Utilisation d'un BytesMessage	– byte[] data ; – BytesMessage message = session.createByteMessage() ; – message.writeBytes(data) ;
Utilisation d'un TextMessage	– StringBuffer data ; – TextMessage message = session.createTextMessage() ; – message.setText(data) ;
Utilisation d'un MapMessage	– MapMessage message = session.createMapMessage() ; – message.setString("Name", " &") ; – message.setDouble("Value", doubleValue) ; – message.setLong("Time", longValue) ;
Utilisation d'un StreamMessage	– StreamMessage message = session.createStreamMessage() ; – message.writeString(" &") ; – message.writeDouble(doubleValue) ; – message.writeLong(longValue) ;
Utilisation d'un ObjectMessage	– ObjectMessage message = session.createObjectMessage () ; – message.setObject(obj) ;

TABLE 5.2 – Messages JMS

5.2.6 Principe de fonctionnement

Un client JMS exécute la séquence d'opérations suivante :
- Il recherche un objet *ConnectionFactory* dans un répertoire en utilisant l'API JNDI (Java Naming and Directory Interface).
- Il utilise l'objet *ConnectionFactory* pour créer une connexion JMS, il obtient alors un objet *Connection*.
- Il utilise l'objet *Connection* pour créer une ou plusieurs sessions JMS, il obtient alors des objets *Session*.
- Il utilise le répertoire pour trouver un ou plusieurs objets *Destination*.
- Il peut alors, à partir d'un objet *Session* et des objets *Destination*, créer les objets *MessageProducer* et *MessageConsumer* pour émettre et recevoir des messages.

5.2.6.1 Communication en mode Point à point

Un client utilise un objet *QueueConnectionFactory* pour créer une connexion active (objet *QueueConnection*) avec le système de messagerie. L'objet *QueueConnection* permet au client d'ouvrir une ou plusieurs sessions (objets *QueueSession*)

pour envoyer et recevoir des messages. L'interface des objets *QueueSession* fournit les méthodes permettant de créer les objets *QueueReceiver*, *QueueSender* et *QueueBrowser*.

5.2.6.2 Communication en mode Publish/Subscribe

Le modèle JMS Publish/Subscribe définit comment un client JMS publie vers et s'abonne à un n œud dans une hiérarchie de sujets. JMS nomme ces n œuds des Topics. Un objet *Topic* encapsule un nom de sujet du système de messagerie. JMS ne met aucune restriction sur la sémantique d'un objet *Topic*, il peut s'agir d'une feuille, ou d'une partie plus importante de la hiérarchie (Pour s'abonner à une large classe de sujets par exemple). Un client utilise un objet *TopicConnectionFactory* pour créer un objet *TopicConnection* qui représente une connexion active avec le système de messagerie. Un objet *TopicConnection* permet de créer une ou plusieurs sessions (objet *TopicSession*) pour produire et consommer des messages. L'interface des objets *TopicSession* fournit les méthodes permettant de créer les objets *TopicPublisher* et *TopicSubscriber*. Un client utilise un objet *TopicPublisher* pour publier des messages concernant un sujet donné. Il utilise un objet *TopicSubscriber* pour recevoir les messages publiés sur un sujet donné. Les objets *TopicSubscriber* classiques ne reçoivent que les messages publiés pendant qu'ils sont actifs. Les messages filtrés par un sélecteur de message ne seront jamais délivrés (dans certains cas une connexion peut à la fois émettre et recevoir sur un sujet). L'attribut *NoLocal* sur un objet *TopicSubscriber* permet de ne pas recevoir les messages émis par sa propre connexion.

5.2.6.3 Exemple d'un code pour producteur JMS

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
 import javax.jms.Session;
 import org.apache.activemq.ActiveMQConnectionFactory;
public class producer{
public static void main (String [] args){
try{ //Create a ConnectionFactory
      ConnectionFactory connectionFactory = new
ActiveMQConnectionFactory("tcp://localhost:61616") ;
//Create a Connection
      Connection connection = connectionFactory.createConnection() ; connection.start();
//Create a Session
      Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
//Create the destination (Topic ou Queue)
     Destination destination = session.createTopic("DTopic");
    // Destination destination = session.createQueue("DQueue");
// Create a MessageProducer from the session to the Topic or Queue
     MessageProducer producer = session.createProducer(destination);
     producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
 // Create a message
     String text = " Hello! "; TextMessage message = session.createTextMessage (text);
//Tell the producer to send the message
       producer.send(message);
// close up
        session.close();  connection.close();
} catch (Exception e) {e.printStackTrace();}
```

FIGURE 5.1 – Programme-5.1 : Code pour producteur JMS

5.2.6.4 Exemple d'un code pour consommateur JMS

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
 import javax.jms.Session;
 import org.apache.activemq.ActiveMQConnectionFactory;
public class consumer{
public static void main (String []args){
try{//Create a ConnectionFactory
      ConnectionFactory connectionFactory = new ActiveMQConnectionFactory("tcp://192.168.43.8:61616");
//Create a Connection
      Connection connection = connectionFactory.createConnection();
// Start Connection
      connection.start();
//Create a Session
      Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
//Create the destination (Topic ou Queue)
     Destination destination = session.createTopic("DTopic");
    // Destination destination = session.createQueue("DQueue");
// Create a MessageConsumer from the session to the Topic or Queue
     MessageConsumer consumer= session.createConsumer (destination);
// Create JMS Listener message
      consumer.setMessageListener(new MessageListener) {
       public void onMessage (Message message) {
        try {   if (message instanceof TextMessage) {
                    TextMessage textMessage = (TextMessage) message;
                   String text = textMessage.getText(); System.out.println ("Received: " + text);    }
         } catch (JMSException e) {e.printStackTrace();}  }};
// close up
         session.close();  connection.close();
 } catch (JMSException e) {e.printStackTrace();}
```

FIGURE 5.2 – Programme-5.2 : Code pour consommateur JMS

5.3 Série d'exercices

Exercice 1 :

1. Définir le middleware orienté message (MOM).
2. Définir les concepts suivants : JNDI et JMS ?
3. Expliquer quelles sont les différences entre un MOM et JMS.
4. A quoi sert le JNDI lorsqu'il est utilisé avec JMS ?
5. Citer les avantages et les inconvénients de JMS.
6. Citer au moins trois applications distribuées qui peuvent être développées avec JMS.

Exercice 2 : Donner le code de producteur JMS et consommateur JMS en considérant la communication en mode publish/subscribe.

Exercice 3 : Ecrire un programme distribué développé avec JMS qui simulent une application de discussion de type chat.

Exercice 4 : Ecrire un programme distribué développé avec JMS qui permet de gérer une bibliothèque (voir l'exercice 4 du chapitre 4).

Conclusion générale

Ce support de cours permet d'acquérir les concepts nécessaires pour mettre en œuvre des applications distribuées en java.
Un premier volet de ce document est consacré à la programmation concurrente et parallèles en se basant sur le multi-threading.
Le deuxième volet de ce support est dédié à étudier les services de communications entre un serveur et un (ou des) client(s) en utilisant des sockets TCP/UDP.
Le troisième volet est consacré à la réalisation d'une application distribuée qui permet d'offrir un ou plusieurs services de communication entre les entités distribuées en utilisant le middleware RMI.
Dans ce support de cours, nous avons également présenté un autre modèle de communications asynchrones (le middleware JMS), qui permet aux entités de l'application distribuéé d'échanger des messages.
Toutefois, malgré l'effort fourni pour sa réalisation, il reste perfectible. Toute remarque éventuelle qui pourrait l'améliorer sera la bienvenue.

Bibliographie

[1] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems Concepts and Design*, 2^{nd} Ed. Addison-Wesley Publishers Ltd., 1994.

[2] A. Tanenbaum. *Systèmes d'exploitation : systèmes centralisés et systèmes distribués*. Interéditions, Paris, 1994.

[3] V. K. Garg. *Concurrent and Distributed Computing in Java*. Willey-Interscience. 2004

[4] C. Delannoy. *Programmer en Java*. Eyrolles. 2007.

[5] A. Fron. *Architectures réparties en JAVA*. Dubod. 2007.

[6] http ://www.jmdoudoux.fr.2019.

[7] http ://www.cs.rutgers.edu/ pxk/rutgers/index.html

[8] http ://www-sop.inria.fr/members/Francoise.Baude/AppRep/

Annexe

Intitulé de la matière : Programmation distribuée.

Crédits : 06

Coefficients : 03

Objectifs de l'enseignement

1. Apprendre les bases et fondements de programmation d'applications réparties en terme de :
   - Modèles de programmation.
   - Apprendre les schémas de conception des applications réparties.
   - Architecture logicielle des applications et du middleware.
2. Maîtriser les principales solutions et techniques existantes :
   - Initiation à la pratique de la mise en oeuvre des différentes plates-formes

Connaissances préalables recommandées : Programmation java, Réseaux.

Contenu de la matière :

- Chapitre 1 : Introduction à la programmation distribuée.
- Chapitre 2 : Programmation des Threads Java.
- Chapitre 3 : Les sockets UDP/TCP et leurs mise en uvre en Java.
- Chapitre 4 : Appel de procédure à distance (RPC) et Java RMI.
- Chapitre 5 : Introduction à JMS (Java Message Service).

Modes d'évaluation : Contrôle Continu et examen.