2ème Ingéniorat en Informatique

Programmation Orientée Objet II

Chapitre IV
Parallélisme et Objets Distribués

2023/2024

Dr. HANNOUSSE Abdelhakim – hannousse.abdelhakim@univ-guelma.dz

Contenu du chapitre

- Multithreading
  - Multithreading vs Parallélisme
  - Multithreading en UML
  - Multithreading en Java
  - Cycle de vie d'une thread
  - Priorités entre threads
  - Synchronisation entre threads
    - Synchronisation par moniteurs
    - Synchronisation par sémaphores
- Objets distribués
  - Les sockets
    - Principe des sockets
    - Types des sockets
    - Les sockets en Java (par TCP, par UDP)
  - RMI
    - Principe du RMI
    - Architecture du RMI
    - RMI en Java
  - CORBA
    - Caractéristiques du CORBA
    - Les étapes de développement d'un objet distribué avec CORBA

Le multithreading

Introduction

- Quand plusieurs opérations doivent être exécutées de manière concurante, la division du traitement pour simuler une exécution en parallèle se nomme ***multithreading***.
- Le ***multithreading*** repose sur l'idée d'avoir plusieurs fils d'exécution, appelés ***threads***, qui peuvent s'exécuter de manière indépendante au sein d'un même programme.
- ***Un thread*** est une unité d'exécution qui représente un chemin d'exécution indépendant à l'intérieur d'un programme. Un thread s'exécute de façon autonome et parallèlement à d'autres threads.
- L'exécution de plusieurs threads se fait généralement sur une machine monoprocesseur avec des intervalles de temps partagés.
- Chaque thread a une priorité. Les threads avec une priorité plus élevée sont exécutés de préférence aux threads avec une priorité inférieure.
- Un thread **daemon** est un type particulier de thread qui s'exécute en arrière-plan et qui ne maintient pas le programme principal en vie tant qu'il reste le seul thread actif.

Multithreading vs Parallélisme (1/2)

Parallélisme	Multithreading (pseudo-parallélisme)
Exécution simultanée des tâches indépendantes.	Les threads partagent les mêmes ressources mais ont leur propre ensemble de registres.
Améliorer les performances en effectuant simultanément des tâches indépendantes à l'aide de ressources matérielles distinctes.	Faciliter l'exécution simultanée de plusieurs parties d'un programme afin d'améliorer l'efficacité en exploitant les temps d'attente, comme les opérations d'entrée/sortie.
Nécessité minimal de synchronisation entre tâches.	Nécessite une synchronisation appropriée pour éviter les conflits.
Implémentation au niveau matériel avec des processeurs multi-cœurs, multiprocesseurs.	Implémentation au niveau logiciel, souvent sur un seul processeur.

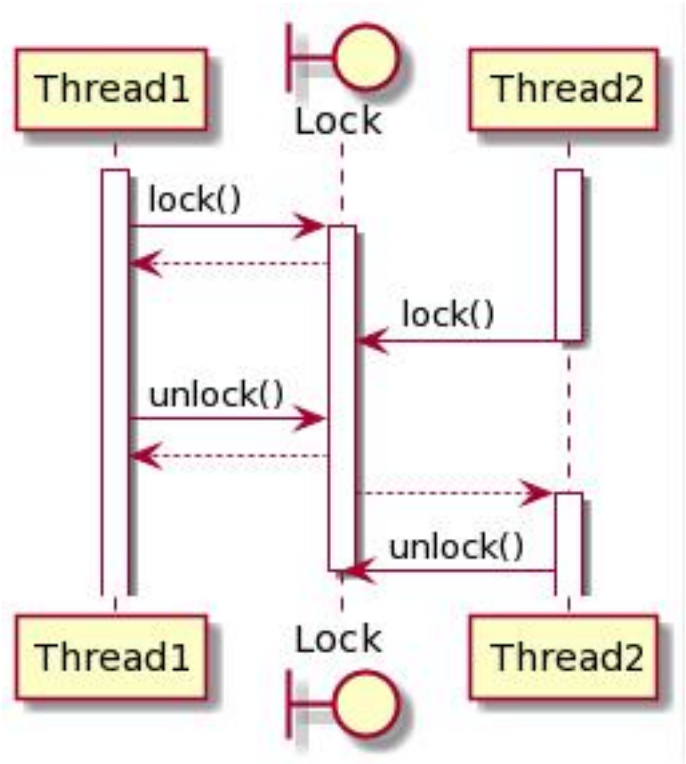
Multithreading vs Parallélisme (2/2)

- **Avantages:**
  - La création de threads est plus rapide que celle de processus (facteur 100 sur certains systèmes).
  - Il y a une meilleure utilisation des ressources de traitements des machines monoprocesseurs ➤ Un programme multithread répond mieux aux entrée/sortie en leur dédiant des threads.
  - La communication est plus facile entre threads qu'entre processus.
- **Inconvénients**
  - La commutation de contexte entre threads prend des ressources.
  - La programmation multithreads est plus complexe que celle d'une simple thread ➤ nécessite un mécanisme de synchronization.

Utilité de multithreading

- Les applications effectuant des calculs intensifs peuvent bénéficier du multithreading en divisant les calculs entre plusieurs threads ➤ Un programme de traitement d'images peut utiliser des threads distincts pour traiter différentes parties d'une image en même temps, améliorant ainsi la vitesse de traitement.
- Permet de maintenir la réactivité de l'interface utilisateur tout en effectuant des opérations intensives en arrière-plan ➤ Un navigateur web peut utiliser un thread distinct pour charger du contenu tout en permettant à l'utilisateur d'interagir avec l'interface.
- Les applications serveur peuvent traiter simultanément plusieurs requêtes en utilisant des threads distincts pour chaque client ➤ Un serveur de messagerie peut gérer plusieurs connexions de clients en parallèle.

Multithreading en UML



Le multithreading en Java

- En Java, la gestion des threads est facilitée par la classe ***Thread*** qui fait partie du package ***java.lang***. Il existe deux principales façons de créer un thread :
  - Étendre la classe ***Thread*** et redéfinir la méthode ***run()***. Lorsque la méthode ***start()*** est appelée, un thread est créé et commence l'exécution de la méthode ***run()*** de l'objet.
  - On implémente l'interface ***Runnable*** qui force la définition de la méthode ***run()***. Il suffit ensuite de construire un ***Thread*** avec l'un des constructeurs qui reçoit un ***Runnable***. On appelle ensuite ***start()*** sur le ***Thread***.

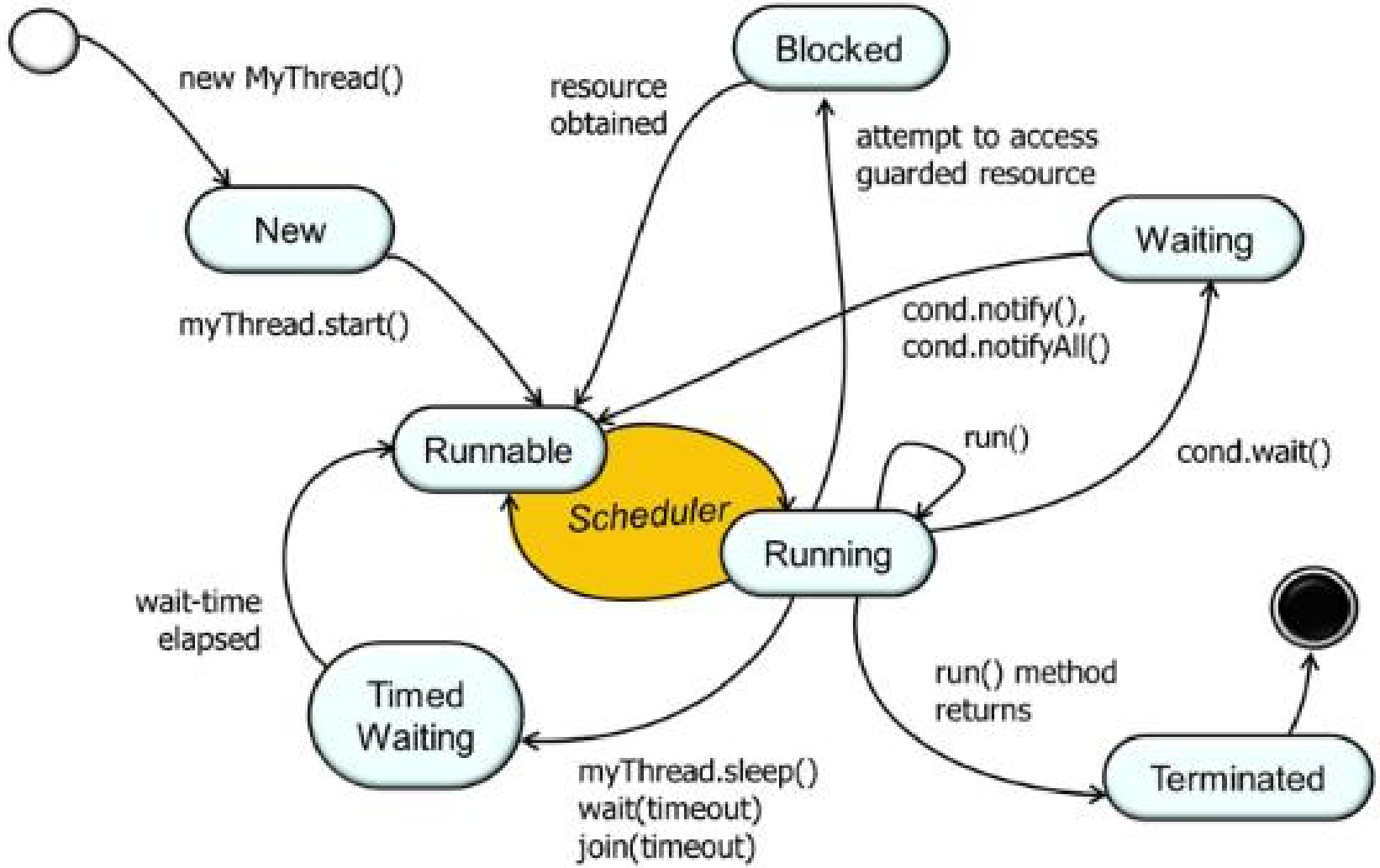
```java
class MonThread extends Thread {
    public void run() {
        // Code à exécuter dans le thread
    }
}

// Création et démarrage du thread
MonThread monThread = new MonThread();
monThread.start();
```

```java
class MonThread implements Runnable {
    public void run() {
        // Code à exécuter dans le thread
    }
}

// Création et démarrage du thread
Thread monThread = new Thread(new MonThread());
monThread.start();
```

Le multithreading en Java – Cycle de vie d'une thread



Le multithreading en Java – Méthodes de la classe *Thread*

Méthode	Description
currentThread()	Renvoie le thread actuellement en cours d'exécution.
getName()/setName()	Renvoie/Fixer le nom du thread.
isAlive()	Indique si le thread est active ou non.
start()	Lancer l'exécution du thread.
run()	Contient le code à exécuter dans le thread. C'est une méthode qui est exécutée automatiquement après que la méthode ***start()*** ait été exécutée.
sleep(n)	Arrêter l'exécution d'un thread pendant n millisecondes.
join()	Attend la fin du thread pour passer à l'instruction suivante.
getState()	Renvoie l'état du thread (NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED)
getPriority()/setPriority(int)	Renvoie/Fixer la priorité du thread.
setDaemon(boolean)	Marquer le thread comme thread démon ou thread utilisateur

Le multithreading en Java – Priorités entre threads

- La priorité entre threads est un mécanisme utilisé par le planificateur de threads pour déterminer quel thread exécuter en priorité.
- En Java, chaque thread se voit attribuer une priorité numérique, généralement comprise entre 1 et 10, où 1 est la priorité minimale et 10 est la priorité maximale. Par défaut, chaque thread hérite de la priorité de son thread parent.
- Les priorités sont définies par les constantes statiques dans la classe Thread. Les priorités minimale et maximale sont respectivement ***Thread.MIN_PRIORITY*** (1) et ***Thread.MAX_PRIORITY*** (10). La priorité par défaut est ***Thread.NORM_PRIORITY*** (5).
- La priorité **n'est pas garantie**, car elle dépend de l'implémentation spécifique du planificateur par la machine virtuelle Java (JVM).
- La priorité n'est qu'une suggestion pour le planificateur de threads et ne garantit pas toujours que le thread avec la priorité la plus élevée sera exécuté en premier. Cela dépend du fonctionnement interne du planificateur de la JVM.

Le multithreading en Java – Exemple (1/2)

```java
import java.util.Random;
class NumberGenerator implements Runnable {
    private final Random random = new Random();
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            int randomNumber = random.nextInt(100);
            System.out.println("Thread de génération : Nombre généré - " + randomNumber);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
class NumberProcessor implements Runnable {
    @Override
    public void run() {
        for (int i = 1; i < 5; i++) {
            System.out.println("Thread de traitement : Carré du nombre généré - " + (i * i));
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```java
public class Main {

    public static void main(String[] args) {

        // Création des instances des classes implémentant Runnable
        NumberGenerator numberGenerator = new NumberGenerator();
        NumberProcessor numberProcessor = new NumberProcessor();

        // Création des threads explicites
        Thread generatorThread = new Thread(numberGenerator);
        Thread processorThread = new Thread(numberProcessor);

        // Démarrage des threads
        generatorThread.start();
        processorThread.start();

        try {

            // Attendre que les threads se terminent
            generatorThread.join();
            processorThread.join();

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

        System.out.println("Fin de l'application");
    }
}
```

Le multithreading en Java – Exemple (2/2)

```java
public class Main {

    public static void main(String[] args) {

        // Création des instances des classes implémentant Runnable
        NumberGenerator numberGenerator = new NumberGenerator();
        NumberProcessor numberProcessor = new NumberProcessor();

        // Création des threads explicites
        Thread generatorThread = new Thread(numberGenerator);
        Thread processorThread = new Thread(numberProcessor);

        // Démarrage des threads
        generatorThread.start();
        processorThread.start();

        try {

            // Attendre que les threads se terminent
            generatorThread.join();
            processorThread.join();

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

        System.out.println("Fin de l'application");
    }
}
```

```
Thread de génération : Nombre généré - 50
Thread de génération : Nombre généré - 42
Thread de traitement : Carré du nombre généré - 1
Thread de génération : Nombre généré - 95
Thread de traitement : Carré du nombre généré - 4
Thread de traitement : Carré du nombre généré - 9
Thread de génération : Nombre généré - 26
Thread de traitement : Carré du nombre généré - 16
Thread de génération : Nombre généré - 66
Fin de l'application
```

Le multithreading en Java – Synchronisation entre threads

- Lorsque plusieurs threads partagent des ressources (variables, ou séquence d'instructions), des problèmes de synchronisation apparaissent presque automatiquement ➤ Il ne faut pas qu'ils le fassent en même temps.
  - **Exemple:** considérons le cas d'un thread qui cherche à effectuer une modification d'une variable tandis que d'autre thread tente simultanément de la lire.
- L'objectif de la synchronisation est d'assurer la cohérence des données partagées entre les threads, en évitant les accès concurrents qui pourraient conduire à des résultats incorrects.
- La synchronisation offre une protection des ressources partagées ou des sections critiques ➤ lorsqu'une thread accède à l'une de ces ressources, aucune autre thread n'est autorisée à y accéder tant que la première n'a pas terminé son exécution.
- Acquérir le verrou d'un objet est couteux. Pour éviter des dégradations des performances il faut que les sections critiques soient courtes et utilisées à bon escient. Un appel à une méthode synchronisée coute 4 fois plus cher.

Le multithreading en Java – Synchronisation entre threads - Moniteurs

- En Java, chaque objet possède un verrou ou moniteur. Le moniteur est une structure interne à l'objet qui agit comme un verrou pour réguler l'accès concurrentiel à ses méthodes ou à d'autres sections critiques.
- Le mot-clé ***synchronized*** est utilisé pour travailler avec les moniteurs en Java.
- Il est possible de synchroniser l'accès à une méthode en utilisant le mot-clé ***synchronized***. Lorsqu'un thread exécute cette méthode sur un objet, les autres threads ne peuvent pas l'exécuter pour le même objet. En revanche, les autres threads peuvent exécuter cette méthode pour une autre instance de la même classe.

```
public synchronized void maMéthode(...) {
    // section critique
}
```

- Lors de la fin de l'appel d'une méthode synchronisée, une principe de premier-**arrivé-premier-servis** est établie avec tout autre appel d'une méthode synchronisée sur un même objet.

Le multithreading en Java – Synchronisation entre threads - Moniteurs

- Lorsqu'un bloc de code est marqué comme ***synchronized***, cela signifie que l'accès à ce bloc est régulé par le moniteur associé au bloc.

```
synchronized (objetDeSynchronisation) {
    // Code de la section critique
}
```

- Les méthodes ***wait()*** et ***notify()*** permettent aux threads de coopérer. ***wait()*** met un thread en attente jusqu'à ce qu'un autre thread invoque ***notify()*** pour le réveiller.
- Lorsqu'un thread est à l'intérieur d'une méthode synchronisée ou d'un bloc synchronisé et appelle la méthode ***wait()***, il relâche le moniteur de l'objet sur lequel il est synchronisé ➤ d'autres threads peuvent accéder à des méthodes synchronisées sur le même objet pendant que ce thread est en attente.
- Lorsqu'un thread est réveillé après un appel à ***notify()*** ou ***notifyAll()***, il attend d'obtenir à nouveau le moniteur de l'objet avant de pouvoir reprendre son exécution.

Le multithreading en Java – Synchronisation entre threads - Moniteurs

```java
import java.util.Random;

class NumberGenerator implements Runnable {
    private final Random random = new Random();
    private int randomNumber;
    private boolean numberGenerated = false;

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            synchronized (this) {
                randomNumber = random.nextInt(100);
                System.out.println("Thread de génération : Nombre généré - " + randomNumber);
                numberGenerated = true;
                this.notify();  // Réveiller le thread en attente (NumberProcessor)
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public synchronized int getGeneratedNumber() throws InterruptedException {
        while (!numberGenerated) {
            this.wait();
        }
        numberGenerated = false;  // Réinitialiser le drapeau après la consommation du nombre
        return randomNumber;
    }
}
```

```java
class NumberProcessor implements Runnable {
    private final NumberGenerator numberGenerator;
    public NumberProcessor(NumberGenerator numberGenerator) {
        this.numberGenerator = numberGenerator;
    }
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            try {
                int generatedNumber = numberGenerator.getGeneratedNumber();
                System.out.println("Thread de traitement : Carré du nombre généré - " +
                    (generatedNumber * generatedNumber));
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
public class ExempleSync {
    public static void main(String[] args) {
        NumberGenerator numberGenerator = new NumberGenerator();
        NumberProcessor numberProcessor = new NumberProcessor(numberGenerator);
        Thread generatorThread = new Thread(numberGenerator);
        Thread processorThread = new Thread(numberProcessor);
        generatorThread.start(); processorThread.start();
        try {
            generatorThread.join(); processorThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Fin de l'application");
    }
}
```

Le multithreading en Java – Synchronisation entre threads

```
Thread de génération : Nombre généré - 53
Thread de traitement : Carré du nombre généré - 2809
Thread de génération : Nombre généré - 63
Thread de traitement : Carré du nombre généré - 3969
Thread de génération : Nombre généré - 71
Thread de traitement : Carré du nombre généré - 5041
Thread de génération : Nombre généré - 63
Thread de traitement : Carré du nombre généré - 3969
Thread de génération : Nombre généré - 21
Thread de traitement : Carré du nombre généré - 441
Fin de l'application
```

Le multithreading en Java – Synchronisation entre threads - Semaphores

- Les moniteurs sont des structures de synchronisation de haut niveau, qui permettent d'installer une exclusion mutuelle, l'attente passive, etc. sans avoir recours à des mutex ou des sémaphores.
- Les sémaphores peuvent être utilisées pour réguler l'accès à un ensemble de ressources partagées et sont utiles pour contrôler l'accès concurrentiel dans des scénarios où un nombre limité de threads peut accéder simultanément à certaines ressources.
- En Java, la classe ***Semaphore*** fait partie du package ***java.util.concurrent*** et fournit une implémentation de sémaphore.
- Avec les semaphores, les threads tentent d'acquérir le sémaphore avec la méthode ***acquire()***, effectuent des opérations critiques, puis libèrent le sémaphore avec la méthode ***release()***. Cela permet de contrôler l'accès concurrentiel aux ressources partagées.

Le multithreading en Java – Synchronisation entre threads - Semaphores

```java
import java.util.Random;
import java.util.concurrent.Semaphore;

class NumberGenerator implements Runnable {
    private final Random random = new Random();
    private int randomNumber;
    private final Semaphore semaphore = new Semaphore(1);
    private boolean numberGenerated = false;

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            try {
                semaphore.acquire();
                randomNumber = random.nextInt(100);
                System.out.println("Thread de génération : Nombre généré - " + randomNumber);
                numberGenerated = true;
            } catch (InterruptedException e) { e.printStackTrace();}
            finally { semaphore.release(); }

            try { Thread.sleep(1000);
            } catch (InterruptedException e) { e.printStackTrace();}
        }
    }
    public int getGeneratedNumber() throws InterruptedException {
        while (!numberGenerated) Thread.sleep(10); // Petit délai pour éviter la contention
        numberGenerated = false;
        return randomNumber;
    }
}
```

Le multithreading en Java – Synchronisation entre threads - Semaphores

```
31  class NumberProcessor implements Runnable {
32      private final NumberGenerator numberGenerator;
33
34      public NumberProcessor(NumberGenerator numberGenerator) {
35          this.numberGenerator = numberGenerator;
36      }
37
38      @Override
39      public void run() {
40          for (int i = 0; i < 5; i++) {
41              try {
42                  int nbg = numberGenerator.getGeneratedNumber();
43                  System.out.println("Thread de traitement : Carré du nombre généré - " + (nbg * nbg));
44                  Thread.sleep(1000);
45              } catch (InterruptedException e) { e.printStackTrace();}
46          }
47      }
48  }
```

Le multithreading en Java – Synchronisation entre threads - Semaphores

```
49  public class ExempleSemaphore {
50      public static void main(String[] args) {
51          NumberGenerator numberGenerator = new NumberGenerator();
52          NumberProcessor numberProcessor = new NumberProcessor(numberGenerator);
53
54          Thread generatorThread = new Thread(numberGenerator);
55          Thread processorThread = new Thread(numberProcessor);
56
57          generatorThread.start();
58          processorThread.start();
59
60          try {
61              generatorThread.join();
62              processorThread.join();
63          } catch (InterruptedException e) {
64              e.printStackTrace();
65          }
66
67          System.out.println("Fin de l'application");
68      }
69  }
```

Les objets distribués: Sockets

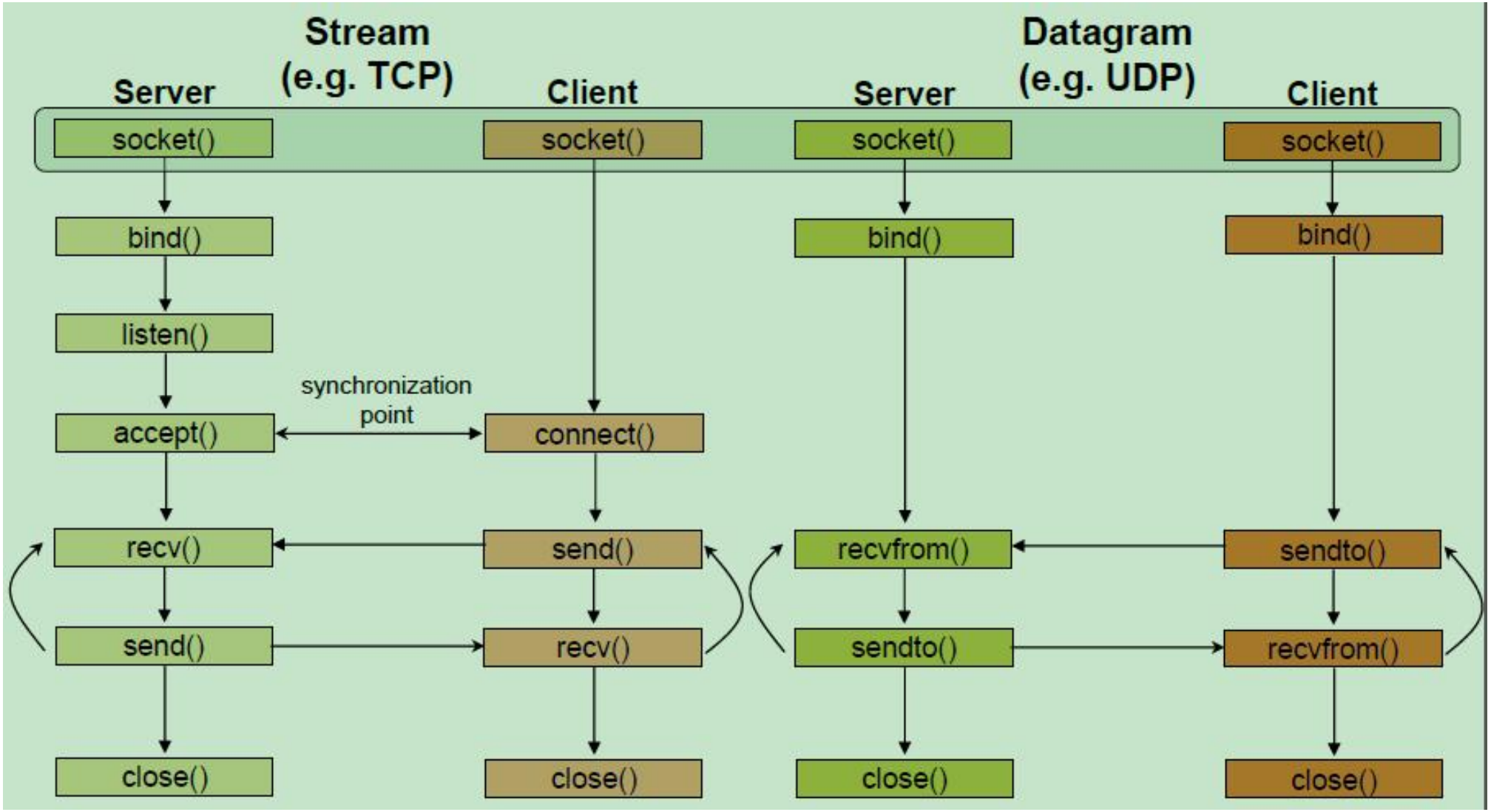
Principe des sockets

- Les sockets (ou prises) sont une abstraction logicielle qui permet la communication bidirectionnelle (deux parties peuvent envoyer et recevoir des données simultanément) entre deux programmes sur un réseau.
- Les sockets offrent une manière flexible et puissante pour que les programmes interagissent sur un réseau, permettant la création d'applications distribuées, de serveurs Web, de services de messagerie, et bien d'autres.
- Les sockets sont souvent utilisés dans une architecture client-serveur, où un programme agit en tant que serveur écoutant les connexions entrantes, et d'autres programmes agissent en tant que clients se connectant au serveur.
- La communication via des sockets repose généralement sur un protocole spécifique, tel que TCP (Transmission Control Protocol) ou UDP (User Datagram Protocol). TCP offre une communication fiable et orientée connexion, tandis qu'UDP offre une communication plus légère mais sans garantie de livraison des paquets.

Principe des sockets

- Il existe deux types de sockets principaux : les **sockets côté serveur** et les **sockets côté client**. Le socket côté serveur écoute les connexions entrantes, tandis que le socket côté client est utilisé par les clients pour se connecter au serveur.
- Les sockets sont identifiés par une adresse IP et un numéro de port. L'adresse IP identifie l'hôte, et le port spécifie le service sur cet hôte. Les sockets sont liés à une adresse et un port spécifiques lors de leur création.
- Une fois qu'une connexion est établie entre le client et le serveur, des flux de données peuvent être créés des deux côtés. Ces flux permettent la lecture et l'écriture de données sur la connexion.
- Le cycle de vie d'une communication via des sockets implique généralement la création d'un socket, la connexion (pour le client), l'acceptation de connexions entrantes (pour le serveur), la communication à travers les flux, puis la fermeture de la connexion lorsque la communication est terminée.

Types des sockets



Les sockets en Java

- En Java, la gestion des sockets est facilitée par le package ***java.net***, il fournit des classes pour la création d'applications réseau, y compris la communication via des sockets.
- Les classes principales pour la gestion des sockets en Java sont:
  - ***Socket*** est utilisé par les clients pour se connecter à un serveur en mode TCP.
  - ***ServerSocket*** est utilisé par les serveurs pour écouter les connexions entrantes en TCP.
  - **DatagramSocket** est utilisé pour l'envoi et la réception de datagrammes en UDP.
- L'utilisation des sockets en mode UDP (User Datagram Protocol) en Java implique quelques différences par rapport à l'utilisation de sockets TCP.

Les sockets en Java – Mode TCP

- Pour créer un socket client en Java, on instancie un objet de la classe ***Socket*** en fournissant l'adresse IP et le port du serveur auquel on souhaite se connecter.

```
Socket clientSocket = new Socket("127.0.0.1", 8080);
```

  - La plage des ports en Java pour les sockets TCP et UDP est de 0 à 65535.
    - Les ports de 0 à 1023 (**ports réservés**) sont généralement réservés pour les services système.
    - Les ports de 1024 à 49151 (**ports enregistrés**), qui peuvent être utilisés par des applications utilisateur.
    - Les ports de 49152 à 65535 (**ports privés**), qui sont généralement utilisés pour les connexions temporaires.
  - Si aucun port est spécifier lors de la création d'un socket, le système d'exploitation assigne un port éphémère (**privé**) comme port source pour la connexion sortante.

Les sockets en Java – Mode TCP

- Pour créer un socket serveur en Java, on instancie un objet de la classe ***ServerSocket*** en spécifiant le port sur lequel le serveur écoutera les connexions entrantes.

```
ServerSocket serverSocket = new ServerSocket(8080);
```

- Pour accepter les connexions entrantes sur le côté serveur, on utilise la méthode ***accept()*** de l'objet ***ServerSocket***. Cette méthode retourne un objet ***Socket*** représentant la connexion établie avec un client.

```
Socket clientSocket = serverSocket.accept();
```

- Une fois qu'une connexion est établie, on peut obtenir des flux de données associés au socket pour la communication. On utilise souvent ***InputStream*** et ***OutputStream*** pour lire et écrire des données.

```
InputStream inputStream = clientSocket.getInputStream();
OutputStream outputStream = clientSocket.getOutputStream();
```

Les sockets en Java – Mode TCP

- On utilise les méthodes de lecture et d'écriture des flux pour échanger des données entre le client et le serveur.

```
// Lecture côté serveur
BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));
String clientMessage = reader.readLine();

// Écriture côté serveur
PrintWriter writer = new PrintWriter(outputStream, true);
writer.println("Message du serveur : Bonjour, client !");
```

- Une fois la communication terminée, il est important de fermer les sockets pour libérer les ressources.

```
clientSocket.close();
serverSocket.close();
```

Les sockets en Java – Mode TCP - Exemple

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

public class TCPClient {
    public static void main(String[] args) {
        final String SERVER_IP = "127.0.0.1";
        final int SERVER_PORT = 12345;

        try {
            // Connexion au serveur
            Socket socket = new Socket(SERVER_IP, SERVER_PORT);
            System.out.println("Connecté au serveur.");

            // Flux de lecture et écriture pour communiquer avec le serveur
            BufferedReader reader = new BufferedReader(new InputStreamReader(socket.getInputStream
                ()));
            PrintWriter writer = new PrintWriter(socket.getOutputStream(), true);

            // Envoi d'un message au serveur
            writer.println("Bonjour, serveur !");

            // Lecture et affichage de la réponse du serveur
            String serverResponse = reader.readLine();
            System.out.println("Réponse du serveur : " + serverResponse);

            // Fermeture des flux et du socket
            reader.close();
            writer.close();
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Les sockets en Java – Mode TCP - Exemple

```java
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
public class TCPServer {
    public static void main(String[] args) {
        final int PORT = 12345;
        try {
            ServerSocket serverSocket = new ServerSocket(PORT);
            System.out.println("Serveur en attente de connexions...");

            // Attente de connexion d'un client
            Socket clientSocket = serverSocket.accept();
            System.out.println("Client connecté depuis : " + clientSocket.getInetAddress());

            // Flux de lecture et écriture pour communiquer avec le client
            BufferedReader reader = new BufferedReader(new InputStreamReader(clientSocket
                .getInputStream()));
            PrintWriter writer = new PrintWriter(clientSocket.getOutputStream(), true);

            // Lecture et affichage du message du client
            String clientMessage = reader.readLine();
            System.out.println("Message du client : " + clientMessage);

            // Réponse au client
            writer.println("Message reçu, merci !");

            // Fermeture des flux et du socket
            reader.close(); writer.close();
            clientSocket.close(); serverSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Les sockets en Java – Mode UDP (Côté client)

- ***DatagramSocket*** est utilisé pour créer un socket qui sera utilisé pour envoyer des datagrammes (paquets) UDP au serveur.

```
DatagramSocket socket = new DatagramSocket();
```

- Les données à envoyer doivent d'abord être transférées en bytes avant d'être encapsulées par le ***DatagramSocket***.

```
byte[] data = message.getBytes();
InetAddress serverAddress = InetAddress.getByName(ip);
DatagramPacket packet = new DatagramPacket(data, data.length, serverAddress, port);
```

- Pour envoyer un datagramme, on utilise la méthode ***send()*** de l'objet ***DatagramSocket*** en fournissant le ***DatagramPacket*** à envoyer.

```
socket.send(packet);
```

- Comme avec les sockets TCP, il est important de fermer le DatagramSocket après utilisation.

```
socket.close();
```

Les sockets en Java – Mode UDP (Côté serveur)

- ***DatagramSocket*** est utilisé pour créer un socket qui sera utilisé pour recevoir des datagrammes (paquets) UDP du client. On doit alors spécifier le port.

```
DatagramSocket socket = new DatagramSocket(port);
```

- Pour recevoir un datagramme, on crée un ***DatagramPacket*** vide pour recevoir les données, puis on utilise la méthode ***receive()*** de l'objet ***DatagramSocket***.

```
byte[] data = new byte[1024];
DatagramPacket packet = new DatagramPacket(data, data.length);
socket.receive(packet);
```

- Une fois le datagramme reçu, on peut extraire les données du ***DatagramPacket*** de manière similaire à la lecture/écriture avec des flux en TCP.

```
String message = new String(packet.getData(), 0, packet.getLength());
```

- Comme avec les sockets TCP, il est important de fermer le DatagramSocket après utilisation.

```
socket.close();
```

Les sockets en Java – Mode UDP - Exemple

```java
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class UDPClient {
    public static void main(String[] args) {
        try {
            // Create a DatagramSocket for the client
            DatagramSocket datagramSocket = new DatagramSocket();

            // Define the server's address and port
            InetAddress serverAddress = InetAddress.getByName("localhost");
            int serverPort = 8080;

            // Create a message to send
            String message = "Hello, Server!";

            // Convert the message to a byte array
            byte[] data = message.getBytes();

            // Create a DatagramPacket to send data to the server
            DatagramPacket packet = new DatagramPacket(data, data.length, serverAddress, serverPort);

            // Send the packet to the server
            datagramSocket.send(packet);

            // Close the DatagramSocket
            datagramSocket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Les sockets en Java – Mode UDP - Exemple

```java
import java.net.DatagramPacket;
import java.net.DatagramSocket;

public class UDPServer {
    public static void main(String[] args) {
        try {
            // Create a DatagramSocket to listen on a specific port (e.g., 8080)
            DatagramSocket datagramSocket = new DatagramSocket(8080);

            // Create a byte array to store received data
            byte[] buffer = new byte[1024];

            // Create a DatagramPacket to receive data
            DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

            // Receive data from the client
            datagramSocket.receive(packet);

            // Extract and display the received message
            String message = new String(packet.getData(), 0, packet.getLength());
            System.out.println("Received from client: " + message);

            // Close the DatagramSocket
            datagramSocket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Les objets distribués: RMI

Remote Method Invocation

Principe de mécanisme RMI

- Le RMI (Remote Method Invocation) est un mécanisme qui facilite la communication entre des objets répartis sur différentes machines dans un environnement Java.
- Le RMI prend en charge des mécanismes de sécurité tels que l'authentification et l'autorisation pour protéger les appels distants.
- Il offre une manière transparente d'appeler des méthodes distantes, simplifiant le développement d'applications distribuées en Java ➤ Il permet d'invoquer des méthodes d'objets distants comme si ces objets étaient locaux.
- RMI repose sur le principe de **sérialisation**. La sérialisation est le processus de conversion d'un objet Java en une séquence d'octets pouvant être transmise sur un réseau. Les objets distants doivent être **sérialisables** pour être utilisés dans le RMI.
- Les paramètres et les valeurs de retour des méthodes distantes doivent être de types **sérialisables**. Les objets sont généralement passés par valeur, c'est-à-dire que des copies sérialisées des objets sont transférées entre le client et le serveur.

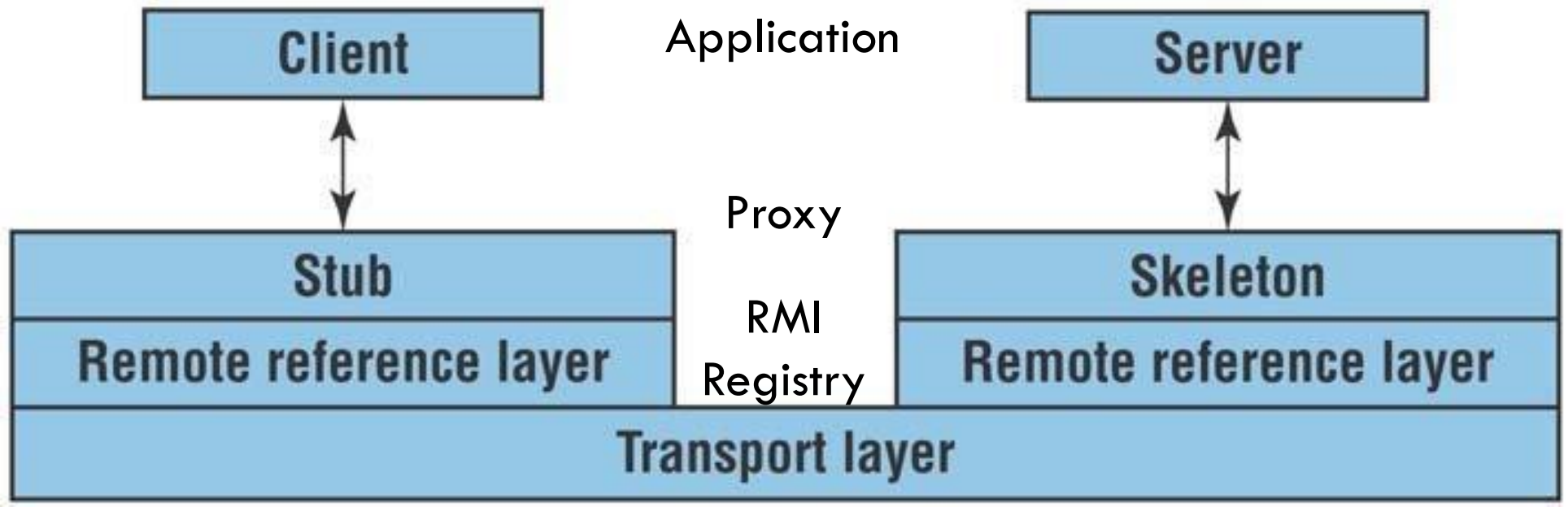
Principe de mécanisme RMI

- Pour qu'un objet distant soit accessible par d'autres clients, il doit être enregistré dans le **RMI Registry**. Cela se fait généralement côté serveur.
- Le **RMI Registry** est un service de registre dans lequel les objets distants sont enregistrés sous un nom. Les clients peuvent rechercher ces objets distants dans ce registre.

Principe de mécanisme RMI

- La communication via RMI entre 2 objets distants est facilité par deux objets spécifique:
  - **Stub (Côté Client):**
    - Représente localement l'objet distant sur le côté client.
    - Fournit une interface locale similaire à celle de l'objet distant.
    - Gère la sérialisation des paramètres, l'envoi de la requête au serveur, la réception des résultats, et la désérialisation.
    - Masque les détails de la communication distante pour le client.
  - **Skeleton (Côté Serveur) :**
    - Reçoit les requêtes du stub côté client.
    - Désérialise les données, extrait les paramètres de la méthode distante invoquée, et appelle la méthode sur l'objet distant réel.
    - Sérialise les résultats (ou les exceptions) et les envoie au stub côté client.
    - Fait la liaison entre le stub et l'objet distant réel.

Architecture RMI



RMI en Java – Définition de l'interface distante

- Il faut d'abord définir une interface Java qui expose les méthodes souhaitant ête appeler à distance. Cette interface doit étendre ***java.rmi.Remote*** et chaque méthode doit déclarer throws ***RemoteException***.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface remoteInterface extends Remote {
    String remoteMethod() throws RemoteException;
}
```

RMI en Java – Implémentation de l'objet distant

- Implémenter cette interface sur le serveur. Les objets côté serveur qui implémentent cette interface sont les objets distants.

```java
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class remoteObject extends UnicastRemoteObject implements remoteInterface {
    public remoteObject() throws RemoteException {
        // Constructeur
    }
    @Override
    public String remoteMethod() throws RemoteException {
        return "Résultat de la méthode distante";
    }
}
```

RMI en Java – Enregistrement de l'Objet Distant (Serveur)

- Il faut enregistrer l'objet distant dans le registre RMI pour qu'il puisse être trouvé par les clients.

```
import java.rmi.Naming;

public class remoteServer {
    public static void main(String[] args) {
        try {
            remoteInterface remoteObject = new remoteObject();
            Naming.rebind("remoteObject", remoteObject);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

RMI en Java – Utilisation de l'objet distant (Client)

- Le client peut rechercher et appeler des méthodes sur l'objet distant en utilisant le stub généré automatiquement.

```
import java.rmi.Naming;

public class remoteClient {
    public static void main(String[] args) {
        try {
            remoteInterface remoteObject =
                    (remoteInterface) Naming.lookup("rmi://localhost/remoteObject");
            String result = remoteObject.remoteMethod();
            System.out.println("Résultat de la méthode distante : " + result);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

RMI en Java - Exécution

- Lancer RMI Registry
  - Ouvrir le terminal
  - Utilisez la commande **cd** pour accéder au répertoire où se trouvent vos fichiers de classe (.class) compilés.
  - Exécutez la commande **rmiregistry**.
- Lancer la classe modélisant Serveur
- Lancer la classe modlélisant le client
- Lorsque les classes sont sauvegardées dans un package autre que celui par défaut d'un projet Eclipse:
  - Utiliser le nom complet de la classe serveur lors de l'enregistrement dans le registre RMI.
  - Utiliser l'URL complète lors de la recherche de l'objet distant.

RMI en Java - Exemple

```java
import java.rmi.Remote;
import java.rmi.RemoteException;

// Interface distante
public interface Calculator extends Remote {
    int add(int a, int b) throws RemoteException;
}

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

// Implémentation de l'interface distante
public class CalculatorImpl extends UnicastRemoteObject implements Calculator {

    public CalculatorImpl() throws RemoteException {
    }
    @Override
    public int add(int a, int b) throws RemoteException {
        return a + b;
    }
}

```

RMI en Java - Exemple

```java
import java.rmi.Naming;
public class CalculatorClient {
    public static void main(String[] args) {
        try {
            Calculator calculator = (Calculator) Naming.lookup("rmi://localhost/CalculatorService");
            int result = calculator.add(5, 10);
            System.out.println("Résultat de l'addition : " + result);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

import java.rmi.Naming;
public class CalculatorServer {
    public static void main(String[] args) {
        try {
            Calculator calculator = new CalculatorImpl();
            Naming.rebind("rmi://localhost/CalculatorService", calculator);
            System.out.println("Serveur RMI prêt.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

L'utilisation des Sockets vs RMI dans une application Java distribuée

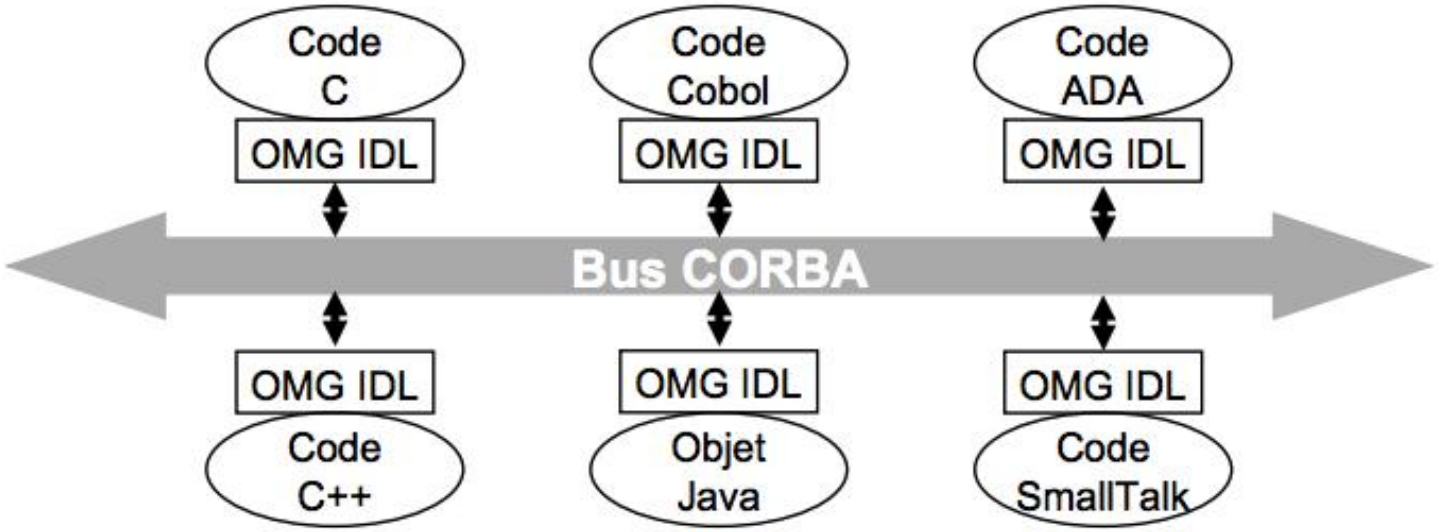
Utilisation des sockets	Utilisation du RMI
Permettent une communication directe entre les applications distribuées en transférant des flux d'octets.	Repose sur l'appel de méthodes à distance. Les clients invoquent des méthodes d'objets distants comme s'ils étaient locaux.
La sérialisation des objets doit être gérée manuellement, en convertissant les objets en séquences d'octets avant de les envoyer et en les reconstruisant à l'arrivée.	La sérialisation est gérée automatiquement par RMI. Les objets distants doivent simplement implémenter l'interface **Serializable**.
Les applications utilisant des sockets doivent définir leur propre protocole de communication, spécifiant la manière dont les données seront structurées et interprétées.	Les objets distants exposent des interfaces qui définissent les méthodes pouvant être appelées à distance.
Les applications utilisant des sockets doivent gérer manuellement l'ouverture, la fermeture et la gestion des connexions.	Les clients obtiennent des proxies RMI, qui agissent comme des représentants locaux des objets distants. RMI gère la création et la gestion des connexions.

Les objets distribués: CORBA

Common Object Request Broker Architecture

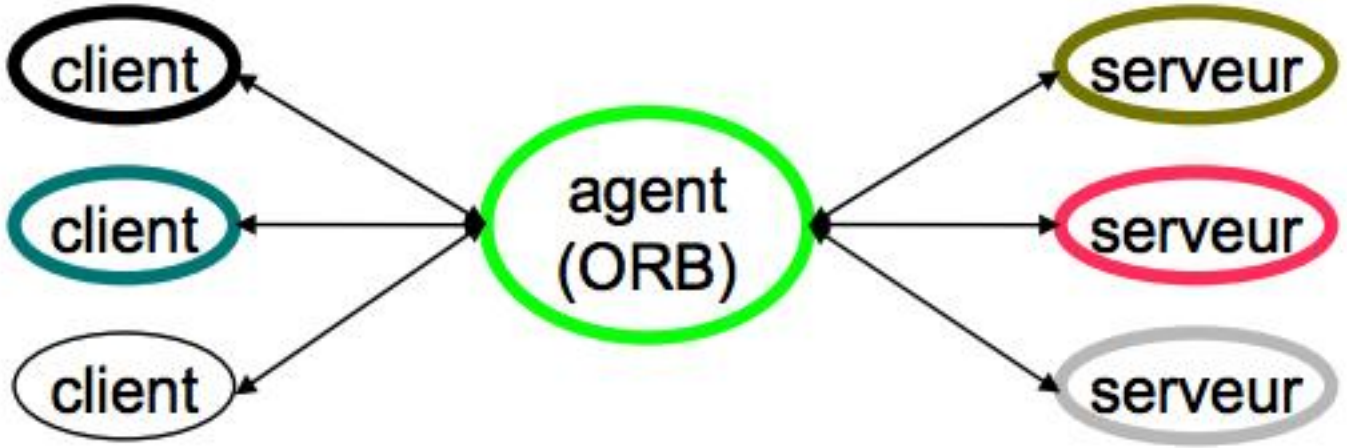
Caractéristiques du CORBA - Interopérabilité

- CORBA (Common Object Request Broker Architecture) est une architecture pour la création d'applications distribuées basées sur des objets.
- CORBA vise à fournir une solution de communication uniforme pour les systèmes distribués hétérogènes ➤ assurer la communication entre des objets dévelopés par différents langages de programmation.



Caractéristiques du CORBA – Isolation des clients et des serveurs

- Les objets clients et serveurs n'ont pas à se connaître mutuellement.
- Permet d'ajouter de nouveaux clients et de nouveaux serveurs sans modifier l'existant.
- Seul l'agent intermédiaire (ORB) connaît l'adresse et les possibilités de chacun.
- Ces informations n'ont pas à figurer dans le code des clients et des serveurs



Les étapes de développement d'un objet distribué avec CORBA

- Définir l'interface avec le langage IDL (Interface Definition Langage)
- Générer les interfaces et classes nécessaires à la distribution
  - **idlj**: génére des interfaces et des classes Java à partir d'une description IDL.
  - **idl2h/idl2cs**: génére des fichiers d'en-tête et des fichiers de source C#.
  - **idl_python**: génére des fichiers sources pour le langage de programmation Python.
  - **idl2cpp**: génére des classes C++.
- Définir le code fonctionnel de l'objet distribué
- Distribuer l'objet au travers de l'ORB
  - Initialiser l'ORB
  - Enregistrer le servant de l'objet distribué dans l'ORB
  - Rendre disponible une référence permettant de localiser l'objet distribué
  - Mettre l'ORB en attente de requêtes

Les étapes de développement d'un objet distribué avec CORBA – Définition de l'interface IDL

```
1  // Calculator.idl
2  module CalculatorApp {
3    interface Calculator {
4      float add(in float num1, in float num2);
5      float subtract(in float num1, in float num2);
6      float multiply(in float num1, in float num2);
7      float divide(in float num1, in float num2) raises (DivideByZero);
8    };
9    exception DivideByZero {};
10 };
```

Les étapes de développement d'un objet distribué avec CORBA - Compilation et génération des classes

- Utiliser l'outil **idlj** pour compiler l'IDL et générer les stubs et skeletons Java:

```
idlj -fall Calculator.idl
```

- L'outil **idlj** va générer des fichiers Java correspondant aux stubs et aux skeletons nécessaires pour mettre en œuvre les interfaces définies dans l'IDL:
  - **Calculator.java :** contient l'interface Java générée pour les définitions d'IDL. Cette interface est utilisée par le client et le serveur pour interagir avec le service de calcul.
  - **CalculatorHelper.java :** une classes utilitaires pour faciliter l'utilisation de l'interface générée. Cette classe est utilisée pour effectuer des opérations telles que le narrowing (réduction) des objets CORBA.
  - **CalculatorHolder.java :** une classes de support pour stocker des références à des objets de l'interface ***Calculator***. Cette classe est générée pour permettre le passage par référence d'objets en tant que paramètres.

Les étapes de développement d'un objet distribué avec CORBA - Compilation et génération des classes

- **CalculatorOperations.java :** contient les signatures des méthodes définies dans l'IDL. La classe concrète du serveur et du client implémenteront ces opérations.
- **CalculatorPOA.java :** classe qui sert de base pour le skeleton côté serveur. Elle implémente toutes les méthodes d'opérations définies dans l'IDL.

Les étapes de développement d'un objet distribué avec CORBA – Développement du classe Serveur

```java
import org.omg.CORBA.*;
import CalculatorApp.*;

public class Server {
  public static void main(String[] args) {
    try {
      ORB orb = ORB.init(args, null);
      POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));

      CalculatorImpl calculatorImpl = new CalculatorImpl();
      org.omg.CORBA.Object ref = rootPOA.servant_to_reference(calculatorImpl);
      Calculator href = CalculatorHelper.narrow(ref);

      org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
      NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

      String name = "Calculator";
      NameComponent path[] = ncRef.to_name(name);
      ncRef.rebind(path, href);

      rootPOA.the_POAManager().activate();
      System.out.println("Serveur prêt et en attente de demandes ...");
      orb.run();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

```java
import org.omg.CORBA.*;
import CalculatorApp.*;

public class Client {
  public static void main(String[] args) {
    try {
      ORB orb = ORB.init(args, null);
      org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
      NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

      String name = "Calculator";
      Calculator calculator = CalculatorHelper.narrow(ncRef.resolve_str(name));

      float num1 = 10;
      float num2 = 5;

      System.out.println("Adding: " + num1 + " + " + num2 + " = " + calculator.add(num1, num2));
      System.out.println("Subtracting: " + num1 + " - " + num2 + " = " + calculator.subtract(num1,
          num2));
      System.out.println("Multiplying: " + num1 + " * " + num2 + " = " + calculator.multiply(num1,
          num2));
      System.out.println("Dividing: " + num1 + " / " + num2 + " = " + calculator.divide(num1, num2));
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Limites du CORBA (1/2)

- CORBA était autrefois une technologie largement utilisée. Cependant, son utilisation a diminué ces dernières années pour plusieurs raisons (n'est plus pris en charge par Java depuis sa version 11) :
  - **Complexité et surcharge :** Les implémentations CORBA sont souvent complexes, nécessitant aux développeurs de gérer des stubs et des skeletons, les cycles de vie des objets et de faire face à divers problèmes de configuration et de déploiement.
  - **Préoccupations de performance :** CORBA peut introduire des surcharges en termes de communication réseau et de sérialisation/désérialisation des appels de méthodes, ce qui peut impacter les performances, surtout dans les systèmes à haut débit.
  - **Défis d'intégration :** De nombreuses plateformes et outils de développement plus récents ne fournissent pas de support natif pour CORBA, rendant difficile l'intégration de composants basés sur CORBA avec des architectures de microservices ou des environnements cloud natifs modernes.

Limites du CORBA (2/2)

- **Vulnérabilités de sécurité :** CORBA est soumis à des vulnérabilités de sécurité, et les implémentations plus anciennes peuvent manquer de fonctionnalités et de mécanismes de sécurité modernes.
- **Évolution des normes :** À mesure que la technologie évolue, de nouvelles normes et protocoles émergent pour répondre aux besoins changeants du calcul distribué. Des technologies comme les services web RESTful, SOAP et gRPC ont gagné en popularité pour leur simplicité, leur flexibilité et leur support pour les pratiques de développement modernes telles que le développement basé sur les API et les architectures de microservices.