ARCHITECTURES DISTRIBUEES

(Java orienté réseaux /architecture micro-services)

Enseignante : Dre KIE EBA VICTOIRE

Table des matières

Informations Générales

- **Intitulé du cours** : Architectures Distribuées (Java orienté réseaux / Architecture Micro-Service)
- **Niveau** : Licence 3 – Informatique de Gestion et Logiciels (IGL)
- **Volume horaire** : 60 heures
- **Prérequis** :
  - Maîtrise du langage Java (programmation orientée objet)
  - Connaissances de base en réseaux (protocoles TCP/IP, notions de socket)
  - Notions d'architecture logicielle (modularité, design patterns)

Objectifs du Cours

À l'issue de ce cours, l'étudiant sera capable de :

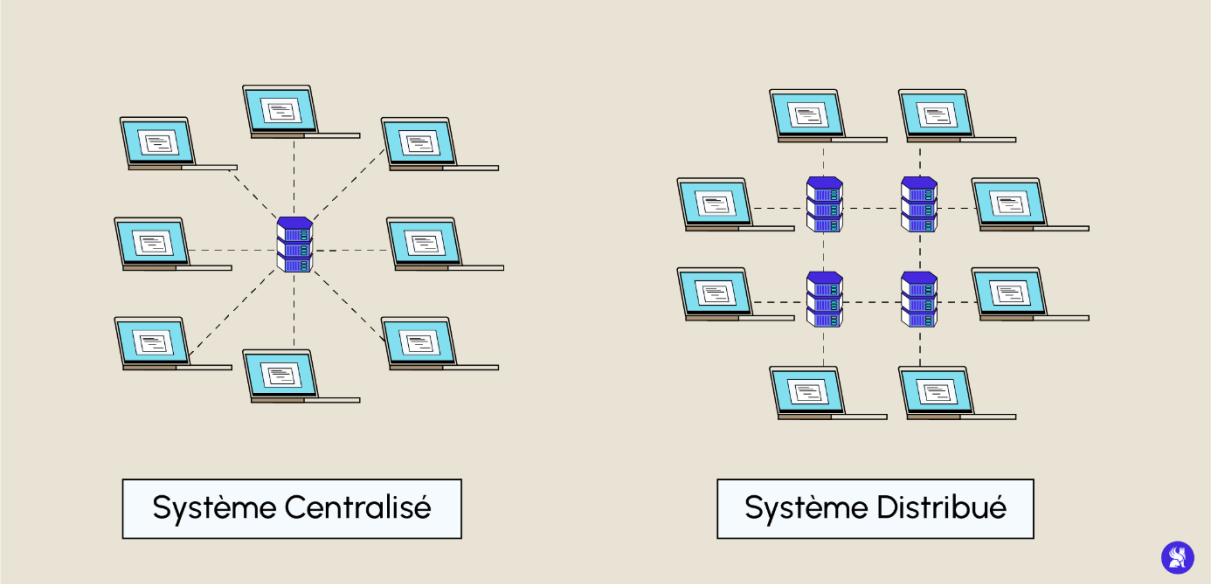
- Comprendre les fondements et les enjeux des architectures distribuées.
- Concevoir et implémenter des applications Java distribuées en utilisant des technologies réseaux (RMI, HTTP/REST).
- Maîtriser les concepts et bonnes pratiques de l'architecture microservices.

Chapitre I : Introduction aux Architectures Distribuées

1. Définitions et caractéristiques des systèmes distribués

Un système distribué est un ensemble d'ordinateurs autonomes interconnectés par un réseau, coopérant pour exécuter une tâche commune. Contrairement aux systèmes centralisés, les systèmes distribués permettent le partage des ressources, la tolérance aux pannes et une meilleure scalabilité.

Avec l'évolution exponentielle de la technologie et la facilitation de l'accès à l'information, de plus en plus de systèmes informatiques comme des applications ou leur déploiement sont reliés entre eux par un réseau et communiquent des données. Ainsi, afin de réaliser un système complexe nécessitant l'exécution de plusieurs services interconnectés, les architectures monolithiques sont devenues inutilisables et obsolètes.



Les systèmes informatiques distribués présentent plusieurs caractéristiques essentielles qui les différencient des systèmes centralisés et qui leur confèrent une grande flexibilité et efficacité.

a. Partage des ressources

Un système distribué permet le partage de diverses ressources, qu'il s'agisse de matériel, de logiciels ou de données. Cela signifie que plusieurs utilisateurs ou applications peuvent accéder simultanément aux mêmes fichiers, bases de données, processeurs ou périphériques (comme des imprimantes ou des serveurs de stockage) sans qu'ils soient physiquement localisés sur un même appareil. Cette mutualisation des ressources optimise leur utilisation et réduit les coûts liés à l'infrastructure informatique.

b. Traitement simultané

Dans un environnement distribué, plusieurs machines peuvent exécuter des tâches en parallèle. Cette caractéristique permet d'accélérer le traitement des données et d'améliorer les performances globales du système.

Par exemple, dans un cluster de calcul, chaque nœud peut prendre en charge une partie du traitement, ce qui permet de réduire considérablement les temps d'exécution des programmes et d'améliorer la réactivité des applications.

c. Évolutivité

L'un des avantages majeurs des systèmes distribués est leur capacité à s'adapter à une augmentation de la charge de travail. En ajoutant de nouveaux nœuds (ordinateurs, serveurs, ou machines virtuelles) au réseau, il est possible d'étendre les capacités de calcul et de stockage sans avoir à modifier profondément l'architecture du système. Cette évolutivité horizontale permet aux entreprises et aux organisations de s'adapter aux besoins croissants sans nécessiter un investissement disproportionné en matériel.

d. Détection des erreurs

Dans un système distribué, les mécanismes de surveillance et de tolérance aux pannes permettent d'identifier rapidement les dysfonctionnements. Lorsqu'un nœud ou un composant du système tombe en panne, les autres éléments du réseau peuvent détecter cette anomalie et éventuellement redistribuer la charge de travail pour éviter toute interruption de service. Cette capacité à détecter et à gérer les erreurs renforce la fiabilité et la disponibilité des services.

e. Transparence

Un système distribué doit offrir un haut degré de transparence afin que les utilisateurs et les applications ne perçoivent pas la complexité sous-jacente du réseau. Cette transparence se manifeste sous plusieurs formes :

- **Transparence d'accès** : un utilisateur peut interagir avec une ressource distante comme s'il s'agissait d'une ressource locale.
- **Transparence de localisation** : les ressources peuvent être déplacées sans que leur emplacement physique n'affecte les performances ou l'accès.
- **Transparence de réplication** : les copies de données peuvent être maintenues sur plusieurs nœuds sans que l'utilisateur ne doive se soucier de leur synchronisation.
- **Transparence de panne** : en cas de défaillance d'un composant, le système peut continuer à fonctionner sans interruption notable.

2. Paradigmes d'architecture distribuée

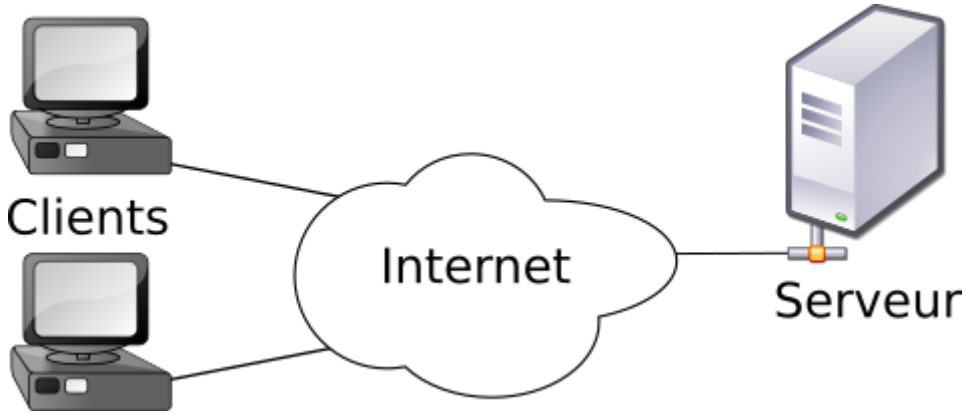
Il existe plusieurs paradigmes d'architecture distribuée, parmi lesquels on retrouve les modèles ***client-serveur***, ***pair-à-pair (P2P)*** et ***n-tiers***. Chacun de ces modèles présente des caractéristiques spécifiques adaptées à différents cas d'usage.

a. Modèle Client-Serveur

L'architecture client-serveur repose sur une communication entre deux entités distinctes :

- **Le client** : Il envoie des requêtes pour obtenir ou manipuler des données. Il peut s'agir d'une application web, d'un logiciel ou d'un terminal utilisateur.
- **Le serveur** : Il traite les requêtes des clients et leur retourne les résultats. Il centralise les ressources et assure la gestion des accès.

Ce modèle est souvent basé sur des protocoles de communication standards comme HTTP , FTP, SMTP ou SQL pour les bases de données.



Exemples d'utilisation

- **Applications Web** : Un navigateur (client) communique avec un serveur web pour récupérer des pages HTML, CSS et JavaScript.
- **Systèmes de gestion de bases de données (SGBD)** : Un serveur de base de données comme MySQL, PostgreSQL ou Oracle stocke et gère les données pour les clients.
- **Messagerie électronique** : Les protocoles comme SMTP, IMAP, POP3 permettent aux clients de récupérer et envoyer des emails via un serveur central.

Avantages de l'architecture client-serveur

- Centralisation des ressources : Facilite la gestion et la sécurisation des données.
- Simplicité d'administration : Les mises à jour et la maintenance se font sur le serveur sans nécessiter d'intervention sur chaque client.
- Contrôle d'accès : Les droits des utilisateurs peuvent être gérés efficacement.

Inconvénients de l'architecture client-serveur

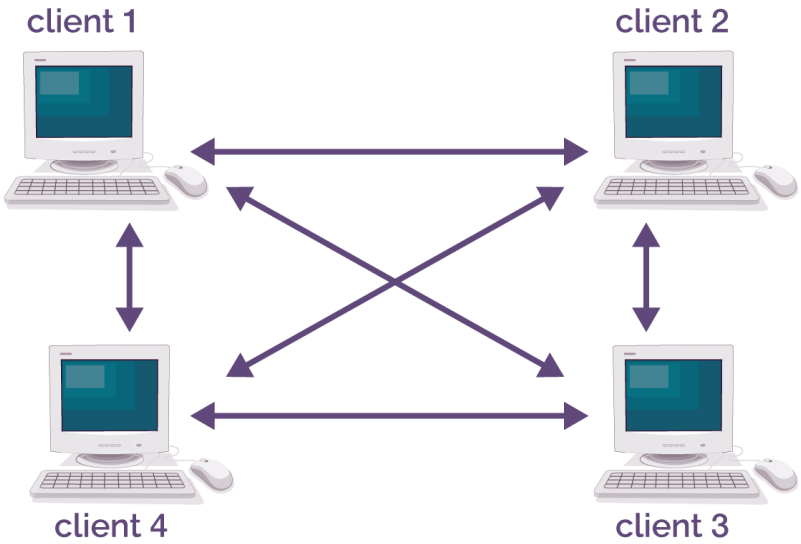
- Point unique de défaillance : Si le serveur tombe en panne, l'ensemble du service devient indisponible.
- Charge sur le serveur : En cas de forte affluence, le serveur peut devenir un goulot d'étranglement, entraînant des ralentissements.
- Coût d'infrastructure : Les serveurs doivent être performants et bien sécurisés, ce qui engendre des coûts élevés.

b. Architecture Pair-à-Pair (P2P)

Dans une architecture P2P (Peer-to-Peer), chaque nœud du réseau peut jouer simultanément le rôle de client et de serveur. Contrairement au modèle client-serveur, il n'existe pas de serveur central, ce qui permet une distribution des ressources et une meilleure tolérance aux pannes.

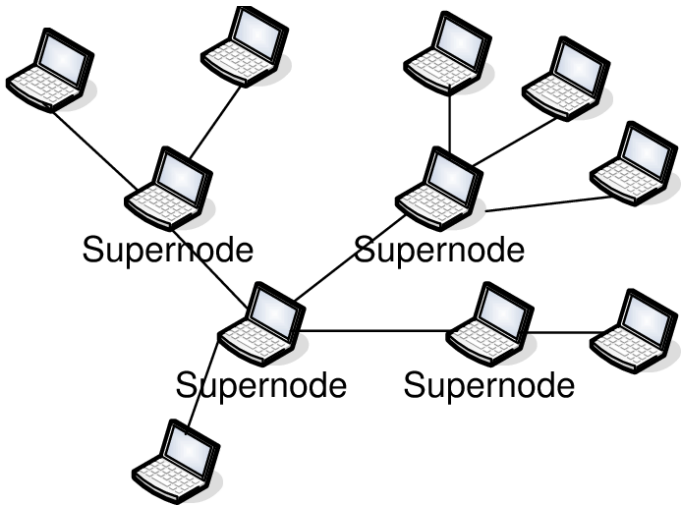
Il existe deux types de réseaux P2P :

- **P2P pur** : Tous les nœuds sont égaux et peuvent partager directement des ressources (ex. : BitTorrent).



- **P2P hybride** : L'architecture hybride est une architecture décentralisée dont chaque nœud est l'élément central d'une architecture centralisée. Ces éléments centraux sont

des super-pairs par rapport à l'architecture globale. (ex. : Skype dans ses premières versions).



Exemples d'utilisation

- Partage de fichiers : BitTorrent permet le téléchargement de fichiers en répartissant les fragments entre plusieurs utilisateurs.
- Cryptomonnaies et Blockchain : Bitcoin et Ethereum utilisent des réseaux P2P où chaque nœud stocke une copie du registre des transactions.
- Communication distribuée : Les premières versions de Skype utilisaient le P2P pour acheminer les appels sans serveurs centralisés.

Avantages de l'architecture Pair-à-Pair

- Résilience élevée : Pas de point unique de défaillance, car les ressources sont distribuées.
- Scalabilité naturelle : Plus il y a de nœuds, plus le réseau devient performant.
- Optimisation des ressources : Chaque nœud peut partager ses capacités de stockage et de calcul.

Inconvénients de l'architecture Pair-à-Pair

- Difficulté de gestion : Le réseau est plus difficile à contrôler et à sécuriser.
- Problèmes légaux : Utilisé pour le partage illégal de fichiers protégés par le droit d'auteur.
- Risques de performances inégales : Certains nœuds peuvent ralentir le réseau si leurs performances sont faibles.

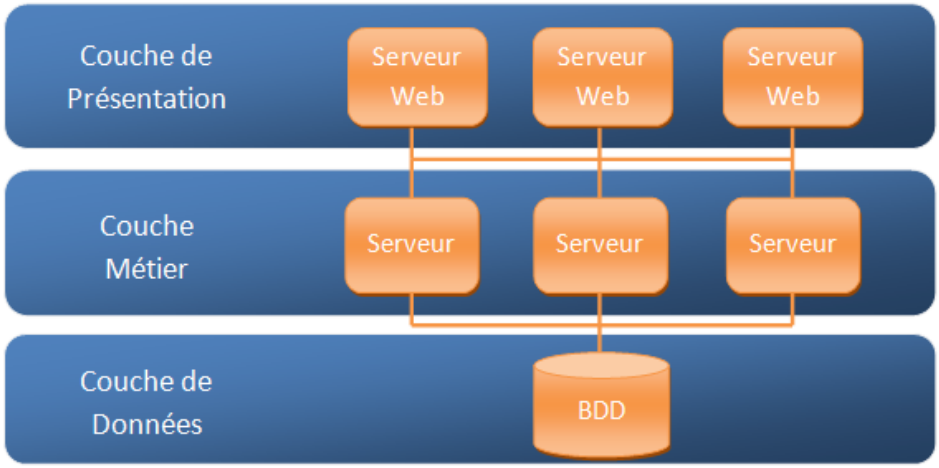
c. Architecture N-Tiers

L'architecture n-tiers divise une application en plusieurs couches logiques distinctes. Le modèle le plus courant est l'architecture 3-tiers, qui comporte :

- Tier Présentation (Interface utilisateur) : Gère l'interaction avec l'utilisateur (ex. : site web, application mobile).
- Tier Logique métier : Traite les règles de gestion et la coordination des services.
- Tier Données : Stocke et récupère les informations dans une base de données.

D'autres architectures existent avec plus de couches, par exemple :

- 4-tiers : Ajout d'une couche d'intégration pour la communication entre services.
- 5-tiers et plus : Séparation encore plus fine des responsabilités.



Explication du schéma

Ce schéma représente une **architecture distribuée** en **trois couches**, couramment utilisée pour concevoir des applications évolutives et performantes. Voici l'interprétation des différentes couches :

Couche de Présentation (Frontend)

Rôle : C'est l'interface utilisateur de l'application. Elle est responsable de l'affichage et de l'interaction avec l'utilisateur final.

Composants : Plusieurs **serveurs web**, qui traitent les requêtes des utilisateurs et les transmettent à la couche métier.

Exemples de technologies : HTML, CSS, JavaScript, Angular, React, Vue.js.

Couche Métier (Backend)

Rôle : C'est le cœur de l'application qui contient la logique métier. Elle traite les requêtes envoyées par la couche de présentation, exécute les traitements nécessaires et communique avec la base de données.

Composants : Plusieurs **serveurs d'application** (ou serveurs métier) qui permettent une répartition de la charge et garantissent la disponibilité du système.

Exemples de technologies : Java (Spring Boot), Python (Django, Flask), PHP (Laravel), Node.js.

Couche de Données (Base de données)

Rôle : Cette couche gère le stockage des données et assure leur intégrité. Elle répond aux requêtes des serveurs métier en lisant, écrivant ou mettant à jour les informations stockées.

Composant principal : **BDD (Base de Données)** qui est un serveur dédié au stockage et à la gestion des données.

Exemples de technologies : MySQL, PostgreSQL, MongoDB, Oracle Database.

Exemples d'utilisation

- ERP (Enterprise Resource Planning) : Systèmes de gestion intégrée comme SAP, Oracle ERP.
- CRM (Customer Relationship Management) : Solutions comme Salesforce, Microsoft Dynamics.
- Applications Web modernes : Une application Angular (présentation) communique avec une API Node.js (logique métier) et une base de données PostgreSQL.

Avantages de l'architecture N-Tiers

- Modularité et réutilisation du code : Chaque couche peut être développée et mise à jour indépendamment.
- Sécurité renforcée : Les accès sont mieux contrôlés entre les couches.
- Scalabilité : Possibilité de répartir la charge en dupliquant certaines couches sur plusieurs serveurs.

Inconvénients de l'architecture N-Tiers

- Complexité accrue : Nécessite plus d'infrastructure et de coordination.
- Temps de réponse potentiellement plus long : En raison des multiples communications entre les couches.
- Coût de développement plus élevé : Nécessite plus de compétences et de ressources.

d. Microservices

L'architecture microservices consiste à diviser une application en plusieurs petits services indépendants qui communiquent via des APIs. Chaque microservice gère une tâche spécifique.

- **Exemple** :

Une application e-commerce où :

- Un microservice gère les commandes.
- Un autre gère le paiement.
- Un autre gère l'inventaire.

- **Avantages** :

- Meilleure évolutivité : chaque service peut être déployé et mis à jour indépendamment.
- Résilience : une panne d'un microservice n'affecte pas toute l'application.

- **Inconvénients :**

- Complexité accrue dans la gestion des communications entre services.
- Besoin d'une bonne orchestration (ex. Docker, Kubernetes).

e. Cloud Computing

Le cloud computing est une architecture où les ressources informatiques (stockage, calcul, bases de données) sont accessibles via Internet, souvent hébergées sur des serveurs distants.

- **Exemple :**

Amazon Web Services (AWS), Microsoft Azure, Google Cloud fournissent des services comme le stockage (S3), les bases de données (BigQuery) et l'hébergement d'applications.

- **Avantages :**

- Accès aux ressources à la demande.
- Réduction des coûts d'infrastructure.
- Haute disponibilité et scalabilité.

- **Inconvénients :**

- Dépendance aux fournisseurs cloud.
- Problèmes potentiels de confidentialité des données.

3. exemples d'applications distribuées

- **Le cloud computing :** Les plateformes comme AWS, Google Cloud et Microsoft Azure offrent des services distribués de stockage, de calcul et de bases de données.

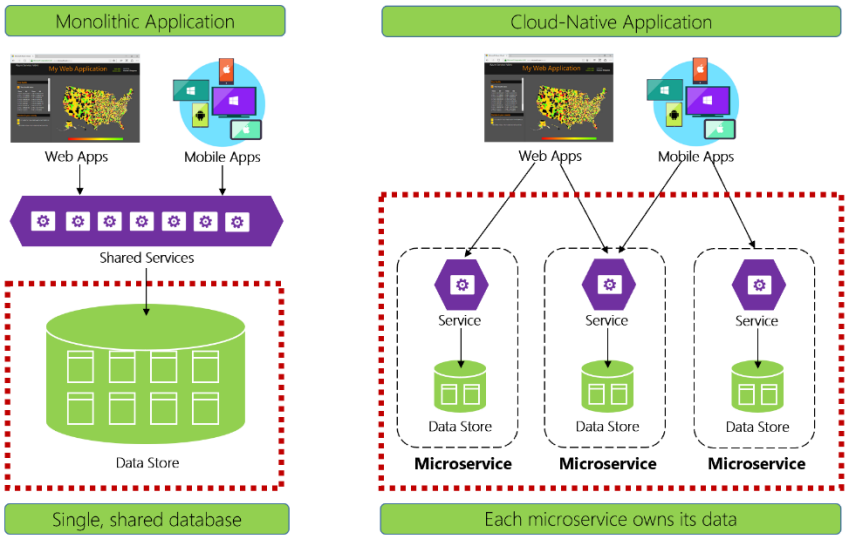


Figure 1: Gestion des données dans les applications natives cloud [1]

- **Les systèmes de blockchain :** La blockchain est une technologie distribuée assurant la transparence et la sécurité des transactions (Bitcoin, Ethereum).

Figure 2: Blockchain[2]

- **Les systèmes de streaming :** Des services comme Netflix et YouTube utilisent des architectures distribuées pour diffuser du contenu en haute qualité à des millions d'utilisateurs.

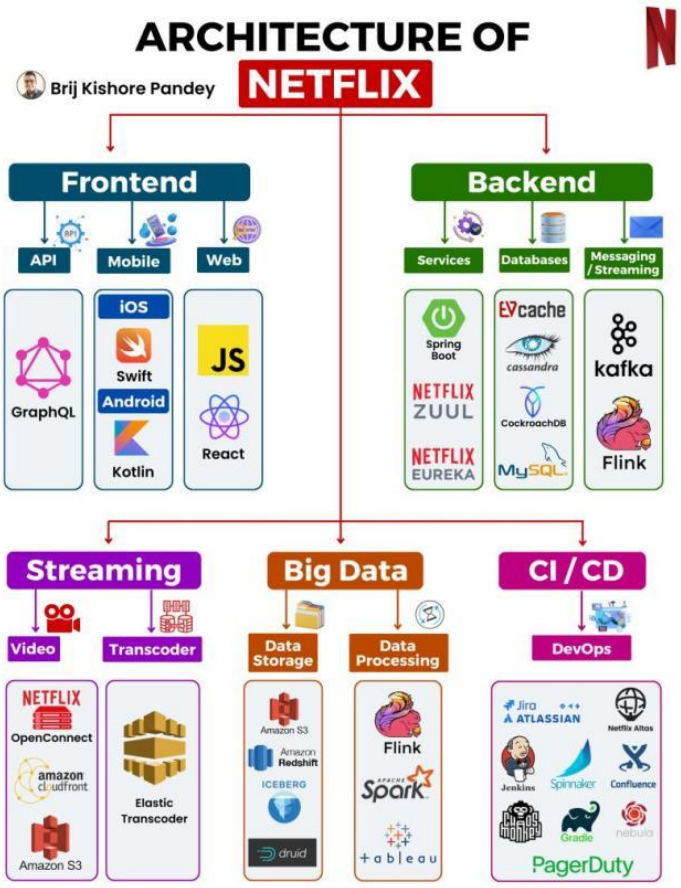


Figure 3: Architecture de Netflix [3]

4. QCM sur l'introduction aux architectures distribuées

1. Qu'est-ce qu'une architecture distribuée ?

A. Un système informatique où tous les composants sont situés sur un seul serveur
B. Un système où les composants sont répartis sur plusieurs machines qui communiquent entre elles
C. Un système où l'ensemble des données est stocké sur un seul ordinateur
D. Un système où les utilisateurs sont connectés à une seule interface

2. Quel est l'objectif principal des architectures distribuées ?

A. Centraliser les ressources pour simplifier la gestion
B. Améliorer la performance et la disponibilité en répartissant les tâches
C. Réduire le coût des serveurs
D. Assurer la sécurité des données

3. Quelle est l'une des principales caractéristiques d'une architecture distribuée ?

A. Les composants partagent la même mémoire
B. Les composants interagissent en temps réel sans nécessiter de communication réseau
C. Les composants sont indépendants, mais interagissent via un réseau
D. Tous les composants sont installés sur un même serveur physique

4. Quel rôle joue le middleware dans une architecture distribuée ?

A. Il optimise le matériel pour améliorer les performances
B. Il permet la communication entre différents systèmes distribués
C. Il crée un réseau local pour connecter les ordinateurs
D. Il assure le stockage de toutes les données

5. Quel est un exemple typique d'architecture distribuée ?

A. Un ordinateur de bureau avec un seul processeur
B. Une application cloud avec plusieurs serveurs répartis
C. Un système d'exploitation monolithique
D. Une base de données locale sur un seul serveur

6. Qu'est-ce qu'une architecture client-serveur dans un contexte distribué ?

A. Un modèle où tous les clients communiquent directement entre eux
B. Un modèle où un client fait des demandes de services et un serveur les fournit
C. Un modèle où les serveurs se connectent entre eux pour fournir des services
D. Un modèle sans serveur centralisé

7. Quelle est une des problématiques majeures d'une architecture distribuée ?

A. La gestion d'une seule base de données
B. La gestion des pannes et de la disponibilité des systèmes
C. L'optimisation de la performance d'un seul processeur
D. La création d'interfaces utilisateurs

8. Quelle est la différence principale entre une architecture distribuée et une architecture centralisée ?

A. Une architecture distribuée possède plus de serveurs que l'architecture centralisée
B. L'architecture centralisée repose sur un serveur central, tandis qu'une architecture distribuée implique plusieurs serveurs
C. L'architecture distribuée est moins sécurisée que l'architecture centralisée
D. L'architecture centralisée ne nécessite pas de réseau

9. Quel type de communication est généralement utilisé dans une architecture distribuée ?

A. Communication par disque dur
B. Communication en mémoire
C. Communication via un réseau (par exemple, TCP/IP)
D. Communication uniquement par fichier

10. Un des avantages majeurs des architectures distribuées est :

A. Une réduction de la complexité de gestion des serveurs
B. La possibilité d'ajouter facilement des ressources sans interrompre le service
C. La centralisation complète des données
D. La diminution des coûts d'infrastructure

Réponses :

1. B
2. B
3. C
4. B
5. B
6. B
7. B
8. B
9. C
10. B

Chapitre II : Programmation Java pour architectures distribuées

Dans le cadre du développement d'applications distribuées, Java offre plusieurs bibliothèques et technologies permettant la communication entre différents processus s'exécutant sur des machines distinctes. Ce chapitre présente les concepts fondamentaux de la programmation distribuée en Java, ainsi que les outils et technologies associées.

1. **Principes de la programmation distribuée en Java**
    a. **Types de programmation distribuée**
    ❖ **Programmation Parallèle dans un Contexte Distribué**

La **programmation parallèle** dans un environnement distribué consiste à diviser une tâche en sous-tâches indépendantes qui peuvent être traitées simultanément sur plusieurs machines (nœuds) d'un réseau. Cette approche est utilisée pour améliorer la performance de systèmes en répartissant la charge de travail sur plusieurs unités de calcul (processeurs ou machines).

Modèles de Programmation Parallèle :

- **Modèle de partage de mémoire** :
    - Ce modèle est utilisé dans des systèmes où plusieurs processus accèdent à une mémoire partagée.
    - Par exemple, dans des systèmes multicœurs, plusieurs threads peuvent accéder à une même zone mémoire, ce qui nécessite de gérer la concurrence pour éviter les conflits (via des mécanismes comme les verrous ou la synchronisation des threads).
    - Utilisé dans **OpenMP** (Open Multi-Processing), qui est une API pour la programmation parallèle sur des architectures à mémoire partagée.
- **Modèle de passage de messages (Message Passing)** :
    - Dans ce modèle, les nœuds communiquent en envoyant des messages, chacun ayant sa propre mémoire locale.
    - Ce modèle est couramment utilisé dans les architectures distribuées, où chaque machine travaille indépendamment et échange des informations via un réseau. Un exemple classique est **MPI (Message Passing Interface)**.
    - Ce modèle est plus complexe car il nécessite une gestion explicite des communications entre les nœuds. Les données doivent être envoyées et reçues entre les machines, ce qui peut introduire des latences et des surcharges de réseau.

Techniques et Bibliothèques :

- **OpenMP** : Pour la programmation parallèle sur des systèmes à mémoire partagée, permettant de paralléliser le code avec des directives simples ajoutées dans le programme source.
- **MPI** : Pour les systèmes distribués, où chaque machine a sa propre mémoire, et où l'intercommunication entre processus se fait par l'échange de messages.

- **CUDA** : Utilisé pour la programmation parallèle sur des GPU (Graphics Processing Units) pour des applications de calcul intensif.

Avantages de la Programmation Parallèle :

- **Performance accrue** : La tâche est répartie sur plusieurs processeurs ou machines, ce qui permet de réduire considérablement le temps de calcul.
- **Scalabilité** : Le système peut évoluer en ajoutant davantage de processeurs ou de nœuds pour traiter plus de données.

Inconvénients de la Programmation Parallèle :

- **Gestion de la synchronisation** : Il est difficile de gérer la synchronisation entre les tâches parallèles, surtout lorsqu'elles partagent des ressources.
- **Communication coûteuse** : Dans les systèmes distribués, la communication entre les nœuds (par exemple, avec MPI) peut être un goulot d'étranglement, surtout quand il y a beaucoup de données à échanger.

❖ RMI (Remote Method Invocation)

Le **Remote Method Invocation (RMI)** est une technologie de la plateforme **Java** qui permet à un programme d'invoquer une méthode sur un objet situé sur une machine distante, comme s'il s'agissait d'un objet local. Cela permet aux applications de distribuer des tâches sur plusieurs machines tout en conservant une interface unifiée.

Principes de Fonctionnement de RMI :

a. **Interface distante (Remote Interface)** : C'est une interface Java définissant les méthodes que le serveur rendra accessibles aux clients distants. Ces méthodes sont déclarées comme throws RemoteException.

b. **Implémentation distante (Remote Object Implementation)** : C'est une classe qui implémente l'interface distante et contient la logique métier qui sera exécutée par le serveur. Cette classe est généralement enregistrée dans le registre RMI pour être accessible aux clients.

c. **Registre RMI** : Il s'agit d'un service de nommage qui permet aux clients de trouver les objets distants. Le serveur enregistre son objet distant dans le registre RMI avec un nom unique, et le client utilise ce nom pour accéder à l'objet.

d. **Sérialisation** : Lorsque le client invoque une méthode sur un objet distant, l'objet et ses paramètres sont sérialisés, envoyés à travers le réseau au serveur, désérialisés sur le serveur, et la méthode est exécutée. Les résultats sont ensuite renvoyés de la même manière.

Avantages de RMI :

- **Abstraction** : RMI permet aux développeurs de travailler comme si l'objet distant était local, simplifiant ainsi le développement des applications distribuées.
- **Transparence** : La communication réseau est transparente pour le programmeur, ce qui réduit la complexité du développement.

Inconvénients de RMI :

- **Dépendance à Java** : RMI est principalement une technologie Java, ce qui peut limiter l'interopérabilité avec des systèmes non-Java.
- **Sérialisation et latence** : La sérialisation des objets pour les transférer sur le réseau peut introduire de la latence, surtout si les objets sont complexes.

Résumé :

La **programmation parallèle** et **RMI** sont deux approches distinctes dans la programmation distribuée. Tandis que la programmation parallèle vise à accélérer les traitements en utilisant plusieurs unités de calcul en parallèle (souvent sur plusieurs nœuds ou processeurs), RMI permet à des applications de s'exécuter sur des machines distantes, en appelant des méthodes sur des objets répartis sur un réseau. Les deux peuvent être combinées dans des architectures distribuées complexes pour maximiser les performances.

Aspect	**Programmation Parallèle**	**RMI (Remote Method Invocation)**
Objectif	Exécution simultanée de tâches sur plusieurs processeurs ou machines.	Exécution de méthodes distantes sur un serveur via le réseau.
Modèle	Souvent basé sur le partage de mémoire ou le passage de messages.	Basé sur l'appel de méthodes à distance, avec sérialisation des objets.
Communication	Les processus communiquent souvent via des mécanismes comme MPI ou OpenMP.	Les objets distants communiquent via RMI et utilisent des protocoles comme RMI-IIOP.
Déploiement	Nécessite généralement des machines proches physiquement ou un réseau haute performance.	Peut être utilisé dans des environnements avec des machines distantes géographiquement.
Gestion des erreurs	Plus complexe, surtout dans des systèmes distribués avec de nombreux nœuds.	Géré via des exceptions comme RemoteException en Java.

b. Modèles de communication

Les applications distribuées reposent sur différents modèles de communication :

- **Appels de Méthodes à Distance (RMI - Remote Method Invocation)**

RMI est une technologie Java permettant à un programme d'appeler des méthodes sur un objet situé sur une autre machine, comme si l'objet était local. C'est un des moyens de communication à distance les plus populaires dans l'écosystème Java.

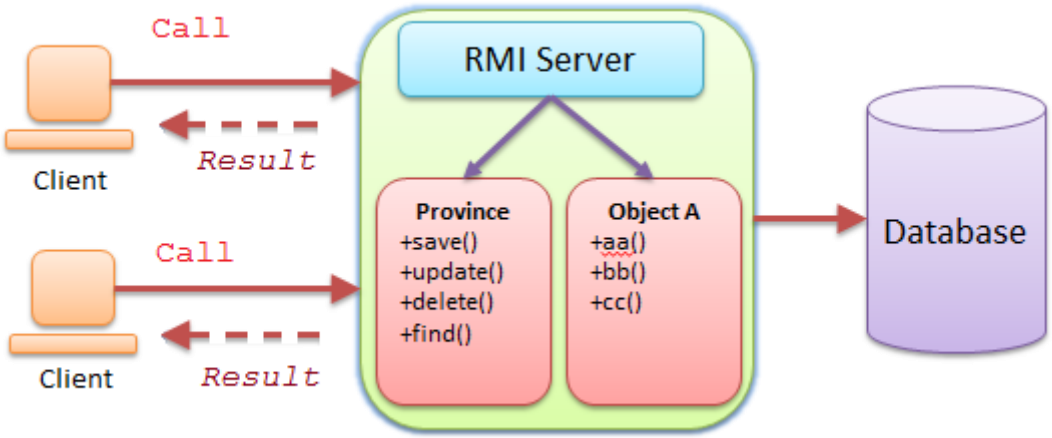
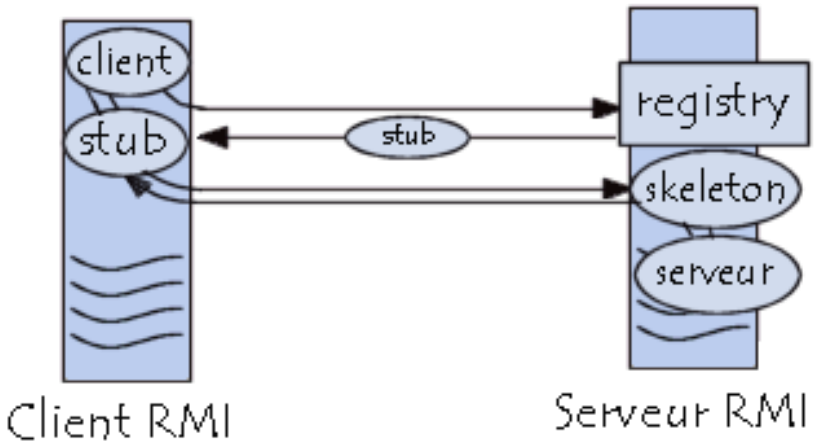


Figure 4: Architecture RMI

- **Architecture** : RMI repose sur un serveur qui expose des objets distants et un client qui peut appeler des méthodes sur ces objets distants.



Lorsqu'un objet instancié sur une machine cliente désire accèder à des méthodes d'un objet distant, il effectue les opérations suivantes :

i. Il localise l'objet distant grâce à un service de désignation : le registre RMI

ii. Il obtient dynamiquement une image virtuelle de l'objet distant (appelée stub ou souche en français). Le stub possède exactement la même interface que l'objet distant.

iii. Le ***stub*** transforme l'appel de la méthode distante en une suite d'octets, c'est ce que l'on appelle la ***sérialisation***, puis les transmet au serveur instanciant l'objet sous forme de flot de données. On dit que le stub "marshalise" les arguments de la méthode distante.

iv. Le squelette instancié sur le serveur "***désérialise***" les données envoyées par le stub (on dit qu'il les "démarshalise"), puis appelle la méthode en local.

v. Le squelette récupère les données renvoyées par la méthode (type de base, objet ou exception) puis les marshalise.

vi. Le stub démarshalise les données provenant du squelette et les transmet à l'objet faisant l'appel de méthode à distance

- **Sérialisation** : Pour que des objets puissent être transmis sur un réseau, ils doivent être sérialisés (convertis en un format transmissible, généralement en binaire).
- **Stubs et Skeletons** :
  - Le **Stub** est un objet représentant un objet distant sur le client, il permet d'envoyer des appels de méthode au serveur distant. *Le stub se situe côté client*
  - Le **Skeleton** (dans les anciennes versions de RMI) était l'objet côté serveur qui recevait et décryptait les appels de méthode et les données provenant des clients. Depuis la version 2 de Java, le skeleton n'existe plus. Seul le stub est nécessaire du côté du client et aussi du côté serveur[4].

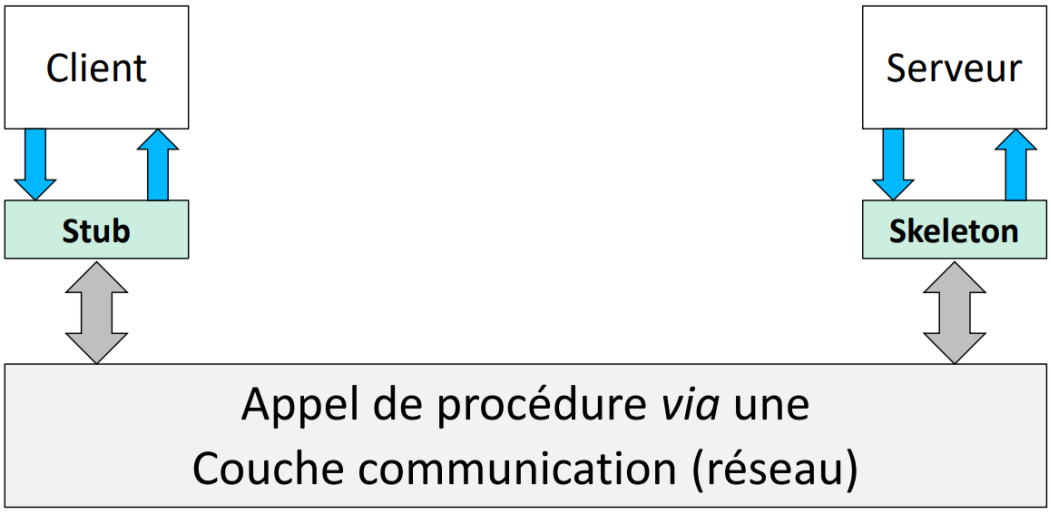
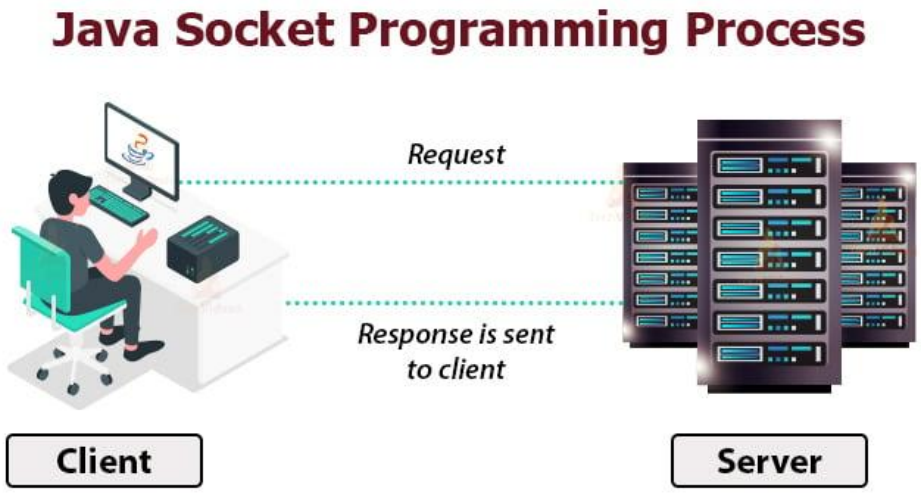


Figure 5: Fonctionnement Stub et Skeleton [5]

- **Sécurité** : RMI fonctionne généralement via des connexions TCP/IP et peut exiger des configurations de sécurité strictes (authentification et chiffrement).
- **Limites** :
    - Il est spécifique à Java, ce qui signifie qu'il n'est pas facilement interopérable avec d'autres langages de programmation.
    - La gestion des erreurs est parfois complexe dans les systèmes distribués.

➢ **Sockets**

Les **Sockets** permettent aux processus de communiquer entre eux via un réseau, à travers un protocole comme **TCP/IP** (Transmission Control Protocol / Internet Protocol) ou **UDP** (User Datagram Protocol). Ce modèle est plus bas-niveau que RMI, JMS ou les Web Services.



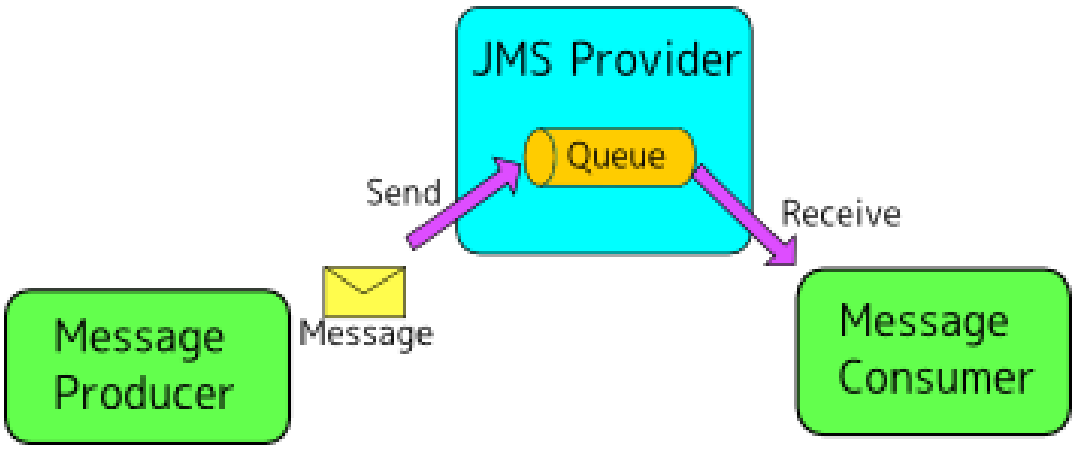
- **Types de sockets** :
    - **TCP (Stream Sockets)** : Utilise TCP pour fournir une communication fiable (basée sur une connexion). C'est le plus utilisé car il garantit que les données arrivent dans le bon ordre et sans erreurs.
    - **UDP (Datagram Sockets)** : Moins fiable, mais plus rapide, UDP ne garantit pas la réception ou l'ordre des paquets. Utilisé pour des applications où la vitesse prime sur la fiabilité (ex : jeux en ligne, vidéo en temps réel).
- **Modèle Client-Serveur** : Les sockets fonctionnent sur un modèle client-serveur. Un serveur ouvre un socket et attend une connexion, tandis qu'un client se connecte au serveur via ce socket pour échanger des données.

- **Communication bidirectionnelle** : Une fois qu'une connexion est établie, les deux parties (client et serveur) peuvent envoyer et recevoir des données de manière bidirectionnelle.
- **Protocoles** :
    - Le **TCP** garantit la livraison ordonnée des paquets de données et leur correction en cas d'erreurs.
    - L'**UDP**, bien que plus rapide, ne garantit ni l'ordre ni la livraison, mais est souvent utilisé dans les systèmes où des pertes de paquets ne sont pas critiques.
- **Limites** : La gestion des erreurs, de la sécurité, et de la gestion des connexions peut rendre l'utilisation des sockets assez complexe. De plus, cela ne prend pas en charge l'abstraction d'objet ou la gestion des transactions, contrairement à d'autres modèles comme RMI ou JMS.

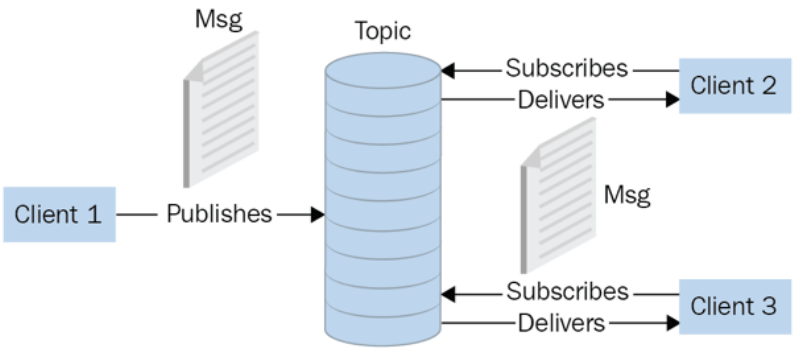
➢ **Messages (JMS - Java Message Service)**

JMS est une API Java qui permet d'envoyer, recevoir et traiter des messages dans un système distribué de manière asynchrone. Cela signifie que l'émetteur du message n'a pas besoin d'attendre une réponse immédiate de la part du récepteur.

- **Types de messages** :
    - **Queue** (File d'attente) : Le message est envoyé à une file d'attente. Seul un consommateur (ou récepteur) peut recevoir ce message. Ce modèle est dit "point à point".



    - **Topic** (Sujet) : Le message est envoyé à un sujet, et tous les abonnés à ce sujet reçoivent une copie du message. Ce modèle est dit "publication/abonnement".



- **Asynchrone** : Les messages peuvent être envoyés sans qu'il y ait de réponse immédiate. Cela permet d'alléger les charges et de traiter des tâches en arrière-plan, sans bloquer l'application principale.
- **Fiabilité** : JMS assure la **fiabilité** du message. Si un message ne peut pas être livré immédiatement, il sera réessayé ou stocké jusqu'à ce qu'il puisse être traité (en utilisant des mécanismes de persistance).
- **Couplage faible** : Grâce à la communication par messages, l'émetteur et le récepteur ne sont pas directement liés. Le récepteur peut être configuré pour écouter les messages à tout moment.
- **Scalabilité** : JMS est souvent utilisé dans des environnements distribués où la scalabilité est cruciale (par exemple, les applications d'entreprise, le traitement de commandes).
- **Limites** :
    - Utilisation principalement dans les systèmes Java, bien que des outils comme ActiveMQ ou RabbitMQ permettent de les rendre interopérables avec d'autres technologies.
    - La gestion des files d'attente peut devenir complexe avec des volumes élevés de messages.

➢ **Web Services (SOAP et REST)**

Les **Web Services** permettent à des applications hétérogènes (écrites dans des langages différents) de communiquer entre elles via des protocoles standardisés, tels que HTTP, SOAP, ou REST. Ce modèle est largement utilisé pour les services accessibles sur Internet.

❖ **SOAP (Simple Object Access Protocol)**

SOAP, ou Simple Object Access Protocol, est un protocole utilisé pour échanger des informations structurées dans la mise en œuvre de services Web. Il utilise XML (Extensible Markup Language) pour formater ses messages, garantissant qu'ils sont lisibles et compréhensibles sur différents systèmes et plates-formes. Le protocole utilise généralement HTTP (HyperText Transfer Protocol) ou HTTPS (HTTP Secure) pour faciliter la communication, permettant d'envoyer et de recevoir des messages sur Internet

- **Protocole standard** basé sur XML, SOAP est un moyen formel de communiquer entre services via des messages XML. Il peut fonctionner sur plusieurs protocoles de transport, dont HTTP, SMTP, TCP.
- **Fonctionnement** : Le client envoie une requête SOAP (généralement en XML) au serveur. Le serveur traite cette requête et renvoie une réponse SOAP.
- **Sécurité** : SOAP intègre des spécifications de sécurité telles que WS-Security, qui permet d'ajouter des signatures, du chiffrement et de l'authentification aux messages.
- **Fiabilité** : SOAP propose des mécanismes de gestion des erreurs et de retransmission.
- **Limites** :
    - Plus complexe que REST et peut avoir un surcoût en termes de performance en raison de son utilisation de XML.
    - Moins flexible que REST pour les applications modernes, car il nécessite des structures plus strictes pour les messages.

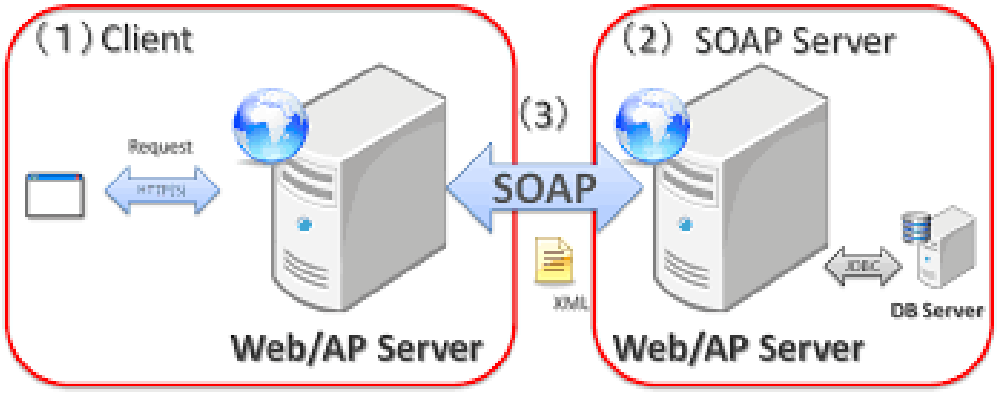
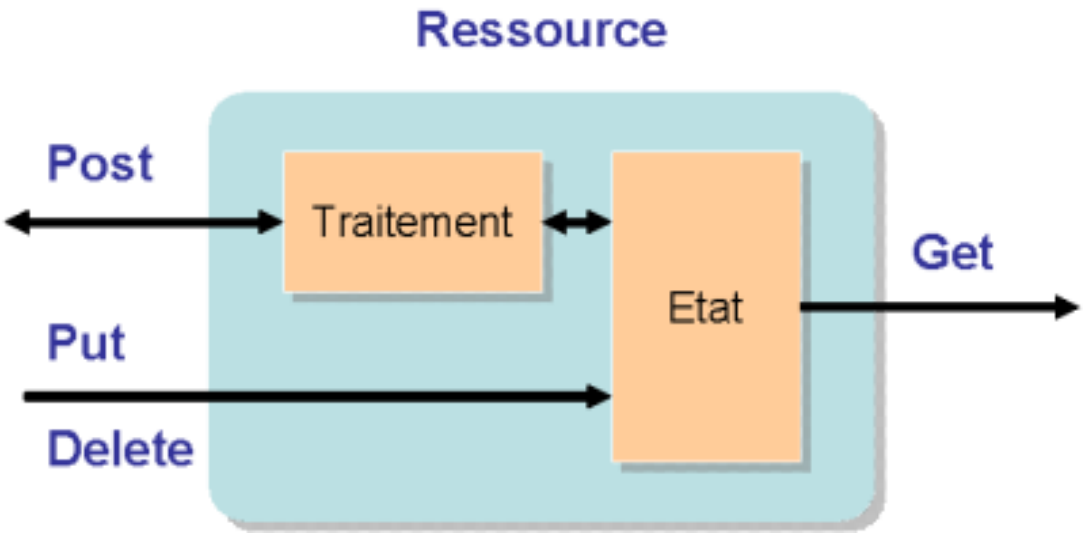


Figure 6: SOAP (Simple Object Access Protocol)

❖ **REST (Representational State Transfer)**

REST (REpresentational State Transfer) est un style d'architecture logicielle définissant un ensemble de contraintes à utiliser pour créer des services web. Les services web conformes au style d'architecture REST, aussi appelés services web RESTful, établissent une interopérabilité entre les ordinateurs sur Internet. Les services web REST permettent aux systèmes effectuant des requêtes de manipuler des ressources web via leurs représentations textuelles à travers un ensemble d'opérations uniformes et prédéfinies sans état. D'autres types de services web tels que les services web SOAP exposent leurs propres ensembles d'opérations arbitraires[6]



- **Architecture** qui utilise les verbes HTTP (GET, POST, PUT, DELETE) pour interagir avec des ressources identifiées par des URI (Uniform Resource Identifier). REST n'est pas un protocole, mais une architecture qui définit une manière d'utiliser les ressources via HTTP.

  Méthodes HTTP : Vue d'ensemble

  - **GET** : Permet d'extraire des données d'un serveur.
  - **POST** : Permet d'envoyer des données au serveur pour créer une nouvelle ressource.
  - **PUT** : Permet de mettre à jour ou de remplacer une ressource existante.
  - **DELETE** : permet de supprimer une ressource du serveur.

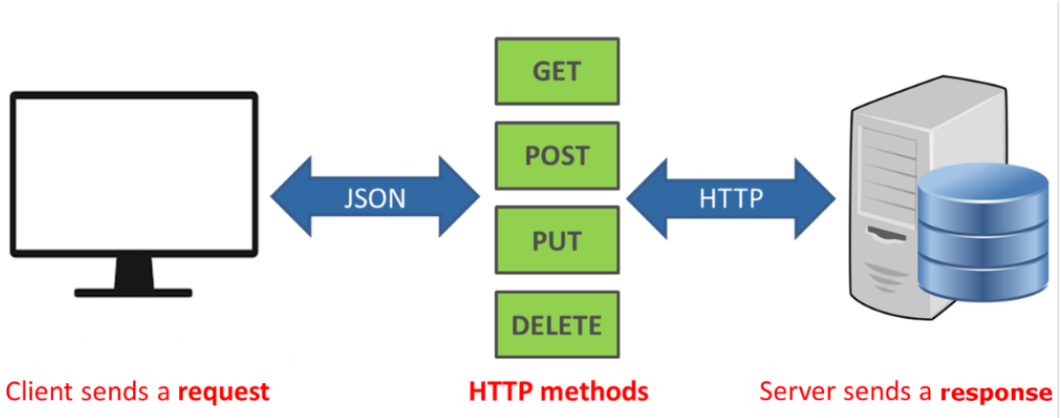


Figure 7: Architecture HTTP[7]

- **Stateless** : Chaque requête REST est indépendante et contient toutes les informations nécessaires pour être traitée (aucune information n'est stockée entre les requêtes).
- **Légèreté** : REST est plus léger que SOAP, car il n'utilise pas XML ou d'autres structures lourdes. Il peut utiliser JSON, qui est plus rapide à traiter et plus compact.

- **Flexibilité** : REST est plus simple à mettre en œuvre et est devenu le modèle préféré pour les API modernes.
- **Limites** :
    - Moins de sécurité intégrée par rapport à SOAP, bien que des mécanismes comme OAuth ou HTTPS puissent être utilisés pour sécuriser les communications.
    - Ne possède pas de normes intégrées pour la gestion des transactions complexes ou des appels synchrone/transactionnels comme SOAP.

Astuces :

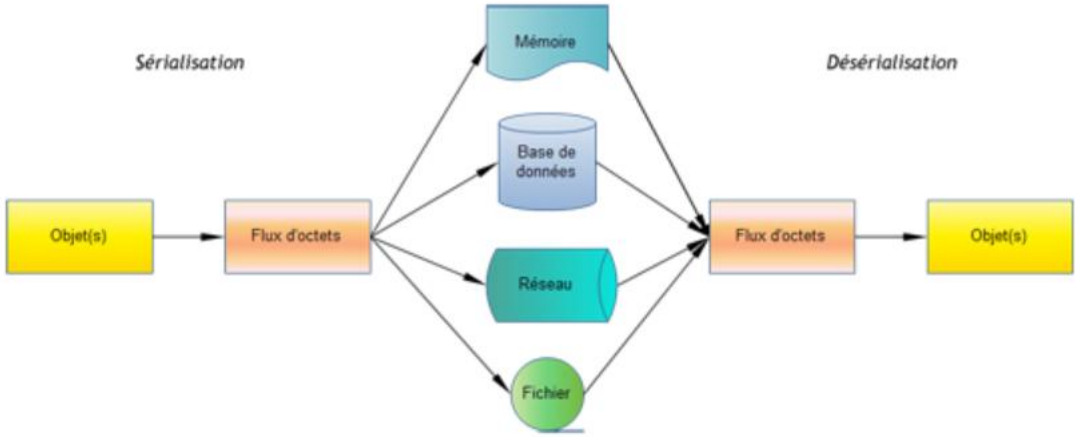
Le choix du modèle de communication dans une application distribuée dépend des exigences spécifiques :

- **RMI** est idéal pour les systèmes Java purement distribués.
- **Sockets** offrent un contrôle plus fin mais nécessitent plus de gestion.
- **JMS** est excellent pour la gestion des messages asynchrones et les architectures basées sur des queues.
- **Web Services (SOAP et REST)** sont parfaits pour les systèmes hétérogènes nécessitant une communication via le web, avec SOAP offrant des garanties de sécurité et de fiabilité, tandis que REST est plus léger et plus moderne.

c. Sérialisation et désérialisation

Les objets Java doivent être convertis en un format transmissible sur un réseau. La sérialisation permet de convertir un objet en un flux de données pouvant être transmis et reconstruit à l'autre extrémité.

La sérialisation d'objets est la capacité d'un objet à stocker une copie complète de lui-même et de tout autre objet auquel il fait référence à l'aide d'un flux de sortie (par exemple, dans un fichier externe). De cette façon, l'objet peut être recréé à partir de la copie sérialisée (enregistrée) un peu plus tard, lorsque cela est nécessaire. La sérialisation d'objets, une nouvelle fonctionnalité introduite dans JDK 1.1, fournit une fonction permettant de convertir des groupes ou des objets individuels en un flux binaire ou un tableau d'octets, pour le stockage ou la transmission sur un réseau. L'opération inverse qui consiste à créer une nouvelle instance à partir du résultat d'une sérialisation s'appelle la ***désérialisation***.



❖ **Sérialisation : Transformation des objets en flux d'octets**

L'objet est d'abord représenté sous sa forme initiale, située à **gauche de la figure (bloc jaune "Objet(s)")**. Il s'agit d'une structure de données manipulée par un programme, comme une instance de classe en programmation. Pour être stocké ou transmis, cet objet doit être converti en un format transportable.

Cette conversion produit un **flux d'octets (bloc orange "Flux d'octets" au centre gauche de la figure)**. Ce flux est une représentation binaire de l'objet, qui peut être stockée ou transmise à distance.

Le flux d'octets peut alors être envoyé vers plusieurs destinations : **la mémoire**, pour un accès rapide aux données ; **une base de données**, pour un stockage durable ; **le réseau**, pour l'échange de données entre systèmes ; et **un fichier**, pour une sauvegarde sur disque. Ces différentes destinations sont situées dans la partie centrale supérieure et inférieure de la figure.

❖ **Transmission et stockage du flux d'octets**

Une fois sérialisé, le flux d'octets est stocké ou transmis à travers l'un des canaux disponibles. S'il est envoyé vers la mémoire, il pourra être traité rapidement sans besoin d'un stockage persistant. Dans le cas d'une base de données ou d'un fichier, l'objectif est de sauvegarder l'objet pour une récupération future. Lorsqu'il est transmis via le réseau, l'objectif est de le partager avec d'autres systèmes distants, souvent dans une architecture distribuée.

Chaque destination joue un rôle clé en fonction des besoins. Une base de données et un fichier permettent une persistance des données, tandis que la mémoire et le réseau facilitent un accès rapide et une distribution efficace des objets.

❖ **Désérialisation : Reconstruction des objets à partir du flux d'octets**

Lorsqu'un programme a besoin de récupérer un objet, il doit d'abord extraire son flux d'octets à partir de l'une des sources disponibles (mémoire, base de données, réseau ou fichier). Ce flux d'octets est situé au **centre droit de la figure (bloc orange "Flux d'octets")**.

Le processus de désérialisation reconstruit l'objet à partir du flux binaire. Cet objet, désormais exploitable par le programme, apparaît à **droite de la figure (bloc jaune "Objet(s)")**. Il retrouve sa structure et ses données initiales, permettant une utilisation comme s'il n'avait jamais été transformé.

❖ **Application en Architecture Distribuée**

Ce mécanisme est fondamental dans les architectures distribuées. Il permet à des systèmes distants de communiquer en échangeant des objets via le réseau. Il facilite également la sauvegarde et la récupération d'objets de manière persistante.

Résumé : La figure illustre comment un **objet (gauche)** est **sérialisé** en un **flux d'octets (centre)**, qui peut être **stocké ou transmis** via différentes méthodes. Ensuite, ce flux est **désérialisé** pour reconstruire l'**objet (droite)**, permettant son utilisation dans un programme. Ce processus est essentiel pour les architectures distribuées, facilitant le stockage, la communication et l'interopérabilité entre systèmes.

La persistance d'un objet est la capacité d'un objet à vivre ou, en d'autres termes, à « survivre » à l'exécution d'un programme. Cela signifie que tout objet créé au moment de l'exécution est détruit par le récupérateur JVM chaque fois que cet objet n'est plus utilisé. Mais si l'API de persistance est implémentée, ces objets ne seront pas détruits par le JVM, mais ils seront autorisés à « vivre », ce qui permettra également d'y accéder au prochain lancement de l'application. En d'autres termes, la persistance signifie qu'il existe une durée de vie pour un objet, indépendante de la durée de vie de l'application en cours d'exécution. Une façon d'implémenter la persistance consiste à stocker les objets quelque part dans un fichier ou une base de données externe, puis à les restaurer ultérieurement en utilisant ces fichiers ou la base de données comme sources. C'est là que la sérialisation entre en jeu. Tout objet non persistant existe tant que la JVM est en cours d'exécution. Les objets sérialisés sont simplement des objets convertis en flux, qui sont ensuite enregistrés dans un fichier externe ou transférés sur un réseau pour stockage et récupération.

d. Technologies et outils pour la programmation distribuée en Java

➢ Java RMI (Remote Method Invocation)

Java RMI permet d'exécuter des méthodes d'objets distants comme s'ils étaient locaux. Il repose sur la sérialisation et la communication par sockets.

Composants principaux :

- Interfaces distantes (définition des méthodes accessibles à distance)

- Classes d'implémentation
- Registre RMI pour localiser les objets distants

➢ Java Sockets

L'API Java fournit des classes pour la communication bas-niveau via TCP ou UDP :

- ServerSocket pour les serveurs
- Socket pour les clients
- DatagramSocket pour les connexions UDP

➢ Java Message Service (JMS)

JMS permet la communication asynchrone entre applications en utilisant des files de messages, ce qui le rend idéal pour les systèmes événementiels et les architectures orientées messages.

➢ Web Services (JAX-WS et JAX-RS)

- JAX-WS (Java API for XML Web Services) : pour développer des services SOAP.
- JAX-RS (Java API for RESTful Web Services) : pour créer des API REST.

➢ Frameworks et outils avancés

- Spring Boot (Spring Cloud) : facilite la création de microservices et leur déploiement.
- Apache Kafka : pour la gestion de messages distribués.
- gRPC : pour des communications performantes basées sur Protocol Buffers.

➢ Outils

Pour mettre en œuvre la programmation distribuée en Java, plusieurs outils et environnements peuvent être utilisés :

Environnements de développement

- Eclipse ou IntelliJ IDEA : IDE populaires pour Java
- Apache NetBeans : offre un bon support pour Java EE et les services Web

Serveurs d'application

- Apache Tomcat : pour déployer des applications web Java et des web services.

- GlassFish ou WildFly : pour Java EE et les services avancés.

Outils de test et de débogage

- Postman : pour tester les API REST.
- Wireshark : pour analyser le trafic réseau entre applications distribuées.
- JConsole : pour surveiller les applications Java distantes.

Outils de virtualisation et de conteneurisation

- Docker : pour exécuter des microservices et des applications distribuées dans des conteneurs.
- Kubernetes : pour orchestrer et déployer des applications distribuées à grande échelle.

TP D'INSTALLATION

Présentation de Intellij Idea : https://www.jetbrains.com/help/idea/discover-intellij-idea.html#IntelliJ-IDEA-editions

Installation de Intellij Idea : https://www.jetbrains.com/help/idea/installation-guide.html#snap

Première application avec Intellij Idea : https://www.jetbrains.com/help/idea/creating-and-running-your-first-java-application.html#get-started

Configurations d'exécution du serveur d'applications : https://www.jetbrains.com/help/idea/creating-run-debug-configuration-for-application-server.html

Services Web RESTful : https://www.jetbrains.com/help/idea/restful-webservices.html

- TP SUR RMI JAVA (Fichier)

2. **TD : développer et exécuter les applications distribuées en utilisant IntelliJ IDEA et Tomcat.**

Exercice 1 : Communication Client-Serveur avec Sockets

Prérequis :

- IntelliJ IDEA installé.
- JDK configuré.
- Application Java classique sans nécessiter de serveur web

Étapes pour configurer dans IntelliJ IDEA :

1. Créer un nouveau projet Java :
    - Lancez IntelliJ IDEA.
    - Créez un nouveau projet Java (File -> New -> Project -> Java).
    - Sélectionnez la version de JDK (Java 8 ou plus récent).
2. Ajouter les classes Server et Client :
    - Créez une nouvelle classe Server (clic droit sur src -> New -> Java Class).
    - Collez le code du serveur.

```
import java.io.*;
import java.net.*;
public class Server {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(5000)) {
            System.out.println("Serveur en attente de connexion...");
            Socket socket = serverSocket.accept();
            System.out.println("Client connecté");
            BufferedReader input = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter output = new PrintWriter(socket.getOutputStream(), true);
            String message = input.readLine();
            System.out.println("Message reçu du client : " + message);
            output.println("Message bien reçu !");
```

```
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explication du code :

- ServerSocket : Ce serveur écoute sur le port 5000 et attend les connexions des clients.
- accept() : Cette méthode bloque l'exécution jusqu'à ce qu'un client se connecte. Une fois qu'une connexion est acceptée, un objet Socket est créé pour la communication avec le client.
- BufferedReader et PrintWriter : Ils permettent respectivement de lire et d'écrire des messages via le flux d'entrée et de sortie du Socket.
- socket.close() : Une fois la communication terminée, le serveur ferme la connexion avec le client.

- Répétez pour la classe Client.

```
import java.io.*;
import java.net.*;
public class Client {
    public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 5000)) {
            // Création des flux de sortie et d'entrée
            PrintWriter output = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader input = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            // Envoyer un message au serveur
            output.println("Bonjour Serveur !");
                    // Lire la réponse du serveur
            String response = input.readLine();
            System.out.println("Réponse du serveur : " + response);
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

Explication du code client:

- Le client se connecte au serveur sur localhost (même machine) et le port 5000 avec new Socket("localhost", 5000).
- Le client envoie un message "Bonjour Serveur !" au serveur en utilisant un PrintWriter.
- Il lit la réponse du serveur à l'aide d'un BufferedReader.

3. Exécuter les programmes :
    - Exécutez d'abord la classe Server (clic droit sur le fichier -> Run).
    - Ensuite, exécutez la classe Client (clic droit sur le fichier -> Run).
    - Vous devriez voir la communication se produire entre le client et le serveur.

Exercice 2 : Introduction à Java RMI

Prérequis :

- IntelliJ IDEA installé.
- JDK configuré.
- Le projet doit être configuré pour utiliser Java RMI .

Étapes pour configurer dans IntelliJ IDEA :

1. Créer un nouveau projet Java avec RMI :
2. Configurer le registre RMI (RMI Registry) :

Avant d'exécuter l'application, vous devez vous assurer que le registre RMI est démarré. Pour cela, ouvrez une fenêtre de terminal et tapez rmiregistry (assurez-vous que le JDK est bien installé et configuré pour RMI).

Vous devriez voir un message indiquant que le registre est en cours d'exécution. Le registre RMI permet au client de se connecter au serveur RMI via un nom spécifique (ici, CalculatriceService).

Configuration du registre RMI (RMI Registry) sous Windows

RMI (Remote Method Invocation) est une technologie Java permettant l'exécution de méthodes sur des objets distants. Pour que cela fonctionne, il est nécessaire de configurer et démarrer un registre RMI.

1. Vérifier l'installation de Java

Avant de commencer, assurez-vous que Java est bien installé sur votre machine en exécutant la commande :

java -version

Si Java est installé, vous verrez une sortie avec la version de Java. Si ce n'est pas le cas, téléchargez et installez le JDK depuis le site officiel d'Oracle ou adoptium.net.

2. Démarrer le registre RMI

Le registre RMI fonctionne sur le port **1099** par défaut. Pour le démarrer, ouvrez un terminal (CMD) et exécutez :

start rmiregistry

Cela lance rmiregistry en arrière-plan. Si vous obtenez une erreur indiquant que la commande rmiregistry n'est pas reconnue, cela signifie que **le chemin du JDK n'est pas configuré dans les variables d'environnement**.

3. Ajouter le chemin de Java dans les variables d'environnement

Si la compilation javac *.java ne fonctionne pas, il est possible que le compilateur Java ne soit pas trouvé. Voici comment ajouter le chemin du JDK à la variable d'environnement PATH sous Windows :

a) Trouver le chemin du JDK

- Allez dans le répertoire où Java est installé (souvent dans C:\Program Files\Java).
- Trouvez le dossier correspondant à votre version du JDK (par exemple, jdk-17).
- Copiez le chemin du dossier bin, qui ressemble à ceci :
- C:\Program Files\Java\jdk-17\bin

b) Ajouter le chemin dans les variables d'environnement

- Appuyez sur **Win + R**, tapez sysdm.cpl, et appuyez sur **Entrée**.
- Allez dans l'onglet **Avancé** et cliquez sur **Variables d'environnement**.

- Dans la section **Variables système**, recherchez la variable **Path**, sélectionnez-la, puis cliquez sur **Modifier**.
- Cliquez sur **Nouveau** et ajoutez le chemin copié (C:\Program Files\Java\jdk-17\bin).
- Cliquez sur **OK** pour valider les modifications.

c) Vérifier la configuration

Fermez et rouvrez le terminal, puis exécutez :

javac -version

java -version

Si les versions s'affichent correctement, la configuration est réussie.

4. Vérifier si le port 1099 est utilisé

Après avoir lancé rmiregistry, vous pouvez vérifier si le port **1099** est bien ouvert avec la commande suivante :

netstat -an | findstr "1099"

Si un processus écoute sur ce port, la commande affichera une ligne comme :

TCP 0.0.0.0:1099 0.0.0.0:0 LISTENING

Si rien n'apparaît, vérifiez si rmiregistry a bien démarré ou si un autre processus utilise déjà ce port.

➢ **Créer un nouveau projet Java dans IntelliJ IDEA**

a. Ouvrir IntelliJ IDEA :

- Lancez **IntelliJ IDEA**.
- Sélectionnez **New Project**.

b. Configurer le projet Java :

- Sélectionnez **Java** dans la liste des types de projets.
- Assurez-vous que le **JDK** est bien configuré (Java 8 ou supérieur).
- Cliquez sur **Next**.

c. Nommer le projet :

- Donnez un nom à votre projet, par exemple, **RMIExample**.
- Choisissez un emplacement où vous souhaitez enregistrer le projet et cliquez sur **Finish**.

➢ **Créer l'interface distante (Calculatrice.java)**

a. Dans le répertoire src, faites un clic droit et sélectionnez **New -> Java Class** pour créer une classe **Calculatrice**.

b. Ajoutez le code suivant pour définir une **interface distante** qui sera utilisée par le client et le serveur :

```
import java.rmi.Remote;

import java.rmi.RemoteException;

public interface Calculatrice extends Remote {

    int addition(int a, int b) throws RemoteException;

}
```

- L'interface Calculatrice étend Remote pour spécifier qu'elle sera utilisée à distance via RMI.
- La méthode addition est déclarée avec throws RemoteException pour indiquer qu'elle peut lever des exceptions liées à RMI.

➢ **Créer l'implémentation du serveur (CalculatriceImpl.java)**

a. Créez une nouvelle classe **CalculatriceImpl** pour implémenter l'interface **Calculatrice**.

b. Le code de la classe CalculatriceImpl ressemble à ceci :

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


public class CalculatriceImpl extends UnicastRemoteObject implements Calculatrice {
    protected CalculatriceImpl() throws RemoteException {
        super();
    }
    @Override
    public int addition(int a, int b) throws RemoteException {
        return a + b;
    }
```

```
}
```

- **UnicastRemoteObject** : Cette classe permet à l'objet de communiquer via RMI. L'appel à super() initialise l'objet pour qu'il puisse être exporté.
- La méthode addition est implémentée pour additionner deux entiers et renvoyer le résultat.

➢ **Créer le serveur RMI (ServerRMI.java)**

a. Créez une nouvelle classe **ServerRMI** dans laquelle vous lancerez le serveur RMI.

b. Le code de la classe ServerRMI ressemble à ceci :

```
import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;
public class ServerRMI {
    public static void main(String[] args) {
        try {
            // Créer le registre RMI sur le port 1099
            LocateRegistry.createRegistry(1099);
            System.out.println("Registre RMI créé.");
            // Créer une instance de l'implémentation de Calculatrice
            CalculatriceImpl calculatrice = new CalculatriceImpl();
            // Lier l'implémentation au registre sous le nom "CalculatriceService"
            Naming.rebind("rmi://localhost/CalculatriceService", calculatrice);
            System.out.println("Serveur RMI prêt...");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- **LocateRegistry.createRegistry(1099)** : Crée le registre RMI sur le port **1099**, ce qui permet aux clients de se connecter à ce serveur.

- **Naming.rebind()** : Enregistre l'objet calculatrice dans le registre sous le nom "CalculatriceService", ce qui le rend accessible à partir de ce nom.

➢ **Créer le client RMI (ClientRMI.java)**

a. Créez une nouvelle classe **ClientRMI** pour faire la demande à notre serveur.

b. Le code du client ClientRMI ressemble à ceci :

```
import java.rmi.Naming;

public class ClientRMI {
    public static void main(String[] args) {
        try {
            // Obtenir le stub du serveur via le registre RMI
            Calculatrice calculatrice = (Calculatrice) Naming.lookup("rmi://localhost/CalculatriceService");

            // Appeler la méthode addition sur le serveur
            int resultat = calculatrice.addition(5, 3);
            System.out.println("Résultat de l'addition : " + resultat);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- **Naming.lookup()** : Permet au client de localiser l'objet distant via le registre RMI.
- Le client appelle la méthode **addition** et affiche le résultat.

➢ **Exécuter le projet RMI**

a. **Démarrer le registre RMI** :

- Avant d'exécuter le serveur, vous devez démarrer le **registre RMI** dans une fenêtre de terminal.
- Ouvrez une fenêtre de terminal et tapez rmiregistry (assurez-vous que le JDK est installé et accessible dans votre PATH).

- Laissez cette fenêtre ouverte, car elle est nécessaire pour la communication RMI entre le client et le serveur.

b. **Exécuter le serveur RMI** :

- Dans IntelliJ IDEA, faites un clic droit sur **ServerRMI.java** et sélectionnez **Run 'ServerRMI.main()'**.
- Vous devriez voir le message : "Serveur RMI prêt...".

c. **Exécuter le client RMI** :

- Faites un clic droit sur **ClientRMI.java** et sélectionnez **Run 'ClientRMI.main()'**.
- Le client se connectera au serveur, appellera la méthode **addition(5, 3)**, et le résultat sera affiché dans la console du client : "Résultat de l'addition : 8".

Exercice 3 : Amélioration avec Multithreading et Sécurité

Prérequis :

- Utilisation de Tomcat pour l'aspect serveur web ou services web (pour le cas où vous voudriez faire un service REST ou SOAP avec Tomcat).

1. Multithreading dans le serveur Socket (Server avec plusieurs clients) :

Pour gérer plusieurs clients simultanément, vous pouvez modifier le code du serveur pour utiliser des threads. Voici comment faire dans IntelliJ IDEA :

1. Ajouter la gestion du multithreading dans le serveur :

Modifiez le serveur pour qu'il crée un nouveau thread pour chaque client qui se connecte.

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


public class Server {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(5000)) {
            System.out.println("Serveur en attente de connexion...");
            while (true) {
                Socket socket = serverSocket.accept();
                System.out.println("Client connecté");
```

```
                // Nouveau thread pour chaque client
                new Thread(new ClientHandler(socket)).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

class ClientHandler implements Runnable {
    private Socket socket;

    public ClientHandler(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        try {
            BufferedReader input = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter output = new PrintWriter(socket.getOutputStream(), true);

            String message = input.readLine();
            System.out.println("Message reçu : " + message);
            output.println("Message bien reçu !");
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
```

```
        }
    }
}
```

2. Exécuter dans IntelliJ IDEA :

Exécutez le serveur et plusieurs clients (ils seront gérés en parallèle).

2. Sécuriser la communication avec SSL/TLS (RMI ou Socket) :

a. Configurer SSL pour Socket :

- Si vous utilisez des sockets, vous pouvez activer SSL pour sécuriser les échanges en utilisant SSLSocket et SSLServerSocket.
- Cela nécessite un keystore et un certificat SSL valide pour le chiffrement des échanges.

b. Configurer SSL pour RMI :

- Pour sécuriser RMI, vous devez configurer un serveur RMI utilisant SSL. Cela nécessite un keystore Java et l'utilisation de la bibliothèque JSSE (Java Secure Socket Extension).

Exercice 4 : Utilisation de Tomcat pour une application web

Si vous souhaitez déployer une application web dans un environnement distribué en utilisant Tomcat, voici comment procéder :

Étapes pour configurer un projet Java Web dans IntelliJ IDEA avec Tomcat :

1. Configurer un serveur Tomcat dans IntelliJ IDEA :
   - Allez dans File -> Settings -> Build, Execution, Deployment -> Application Servers.
   - Ajoutez votre installation Tomcat en choisissant le répertoire d'installation de Tomcat sur votre machine.
2. Créer un projet Web :
   - Créez un projet de type Java EE (Dynamic Web Project).
   - Dans le répertoire WEB-INF, créez les servlets nécessaires pour la gestion des requêtes HTTP.
   - Développez des servlets pour les interactions avec les clients.
3. Déployer sur Tomcat :
   - Une fois votre projet créé, configurez un artefact pour déployer l'application sur Tomcat.

- Allez dans Run -> Edit Configurations et sélectionnez votre serveur Tomcat.
- Ajoutez le projet comme artefact à déployer.
- Cliquez sur Run pour démarrer le serveur Tomcat avec votre application.

Exemple : Servlet Simple

```
@WebServlet("/HelloServlet")
public class HelloServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        response.getWriter().write("Bonjour, ceci est un servlet!");
    }
}
```

Chapitre III : Approche Microservices

1. **Concepts et design des microservices**

a. Passage d'une architecture monolithique à une architecture microservices

L'architecture monolithique est un style de conception de logiciels où une application est développée et déployée comme un seul bloc indivisible. Ce modèle a longtemps été populaire, mais il présente plusieurs limitations, notamment en matière de scalabilité, de maintenabilité et de flexibilité dans le déploiement. Les modifications dans une partie du système peuvent avoir un impact sur l'ensemble de l'application, rendant ainsi le développement et la maintenance plus complexes.

L'architecture microservices, quant à elle, consiste à diviser une application en petits services autonomes qui communiquent entre eux via des API. Chaque microservice représente un domaine fonctionnel spécifique de l'application et peut être développé, déployé, mis à l'échelle et maintenu indépendamment des autres services. Ce changement permet d'améliorer la flexibilité, la résilience et la capacité de mise à l'échelle des systèmes logiciels.

b. Avantages et inconvénients des microservices

❖ Avantages des microservices

➢ **Scalabilité indépendante** : Chaque microservice peut être mis à l'échelle indépendamment des autres, ce qui permet une gestion plus fine des ressources.

- **Déploiement autonome** : Les équipes peuvent déployer et mettre à jour les microservices de manière indépendante, réduisant ainsi les risques et les temps d'indisponibilité.
- **Flexibilité technologique :** Chaque microservice peut être développé avec la technologie la plus appropriée pour son fonctionnement, offrant une grande liberté d'innovation.
- **Résilience accrue** : Si un microservice échoue, il est plus facile d'isoler et de réparer le problème sans affecter l'ensemble du système.

❖ **Inconvénients des microservices**

- **Complexité de gestion** : Le déploiement et la gestion de plusieurs microservices peuvent devenir complexes, en particulier lorsqu'il s'agit de coordonner les mises à jour et la communication entre les services.
- **Surcharge réseau** : La communication entre microservices repose souvent sur des appels réseau, ce qui peut introduire de la latence et une surcharge.
- **Coûts supplémentaires** : Les microservices nécessitent une infrastructure supplémentaire pour leur gestion (outils de supervision, de logging, de gestion des API, etc.), ce qui peut augmenter les coûts de mise en œuvre.
- **Problèmes de gestion des transactions** : Dans un système de microservices, la gestion des transactions distribuées est plus complexe que dans une architecture monolithique.

2. Principes de conception : Domain Driven Design (DDD) et bounded contexts

Le Domain Driven Design (DDD) est une approche de conception de logiciels qui se concentre sur la modélisation du domaine d'un problème à travers une collaboration étroite entre les développeurs et les experts métier. DDD est souvent utilisé pour concevoir des systèmes complexes, notamment dans les architectures microservices, où chaque microservice peut correspondre à un domaine métier spécifique.

Le concept de **Bounded Context** est un élément central du DDD. Un **Bounded Context** représente un sous-ensemble du domaine dans lequel un modèle spécifique s'applique. Chaque microservice correspond généralement à un Bounded Context. Ce concept permet d'isoler les services et de garantir que chaque service fonctionne de manière autonome avec ses propres règles et modèles de données, tout en minimisant les dépendances avec d'autres services.

o Patterns d'intégration et de communication (synchrones vs asynchrones)

Les microservices doivent souvent communiquer entre eux pour échanger des données ou exécuter des processus complexes. Il existe deux principaux types de communication entre microservices : la communication synchrone et la communication asynchrone.

❖ Communication synchrone

Dans une communication synchrone, un microservice envoie une requête à un autre microservice et attend une réponse avant de continuer. Ce type de communication est souvent réalisé via des appels HTTP (API REST) ou des messages RPC (Remote

Procedure Call). Les avantages de la communication synchrone incluent la simplicité et la facilité d'implémentation. Cependant, elle présente des inconvénients, tels que des problèmes de latence et une dépendance entre les services. Si un microservice devient lent ou indisponible, cela peut affecter toute l'application.

❖ **Communication asynchrone**

Dans une communication asynchrone, un microservice envoie une requête et continue son travail sans attendre une réponse immédiate. Les messages sont généralement envoyés via des systèmes de messagerie comme Kafka, RabbitMQ ou JMS. L'avantage principal de la communication asynchrone est qu'elle permet une plus grande résilience, car les services peuvent continuer à fonctionner même si d'autres services sont temporairement hors ligne. Toutefois, cela ajoute de la complexité au niveau de la gestion des erreurs et des transactions.

3. **Communication et coordination**

a. API REST, messaging (RabbitMQ, Kafka)

Une API REST (Representational State Transfer) est une interface qui permet à différents systèmes de communiquer via des requêtes HTTP. Elle permet d'exposer des services et de les rendre accessibles à d'autres applications ou systèmes. Les APIs REST utilisent généralement les méthodes HTTP telles que GET, POST, PUT, DELETE pour interagir avec les ressources.

Pour des architectures plus complexes, notamment dans des systèmes distribués, les messages peuvent être échangés de manière asynchrone via des queues de messages. Parmi les outils populaires pour implémenter un système de messaging, on retrouve RabbitMQ et Kafka. Ces systèmes permettent de transmettre des messages entre des producteurs et des consommateurs de manière fiable et scalable.

❖ **RabbitMQ :**

RabbitMQ est un broker de messages qui permet la communication asynchrone entre les différents composants d'un système. Il est basé sur le modèle de file d'attente (queue), où les producteurs de messages (publishers) envoient des messages dans des queues, et les consommateurs (consumers) récupèrent ces messages pour traitement.
Avantages : facilité d'installation, prise en charge de nombreux protocoles, robuste et fiable.

❖ **Kafka** :

Kafka est une plateforme de streaming distribuée qui permet de gérer des flux de données en temps réel. Contrairement à RabbitMQ, Kafka est conçu pour gérer de très gros volumes de données, souvent en temps réel, et peut traiter des événements à une grande échelle.
Avantages : haute performance, scalabilité horizontale, gestion de logs distribués.

a. Gestion des transactions distribuées et cohérence éventuelle

La gestion des transactions distribuées est un enjeu majeur dans les architectures microservices, car chaque microservice peut gérer sa propre base de données et donc avoir des transactions indépendantes. Lorsqu'il s'agit de coordonner des transactions qui impliquent plusieurs microservices, cela devient une problématique complexe.

Dans un environnement distribué, il est essentiel d'assurer la cohérence des données entre les différents services tout en garantissant que les transactions soient effectuées de manière fiable. Une des méthodes les plus courantes pour gérer cela est le modèle de cohérence éventuelle, qui assure que, bien que les données ne soient pas immédiatement cohérentes entre tous les services, elles finiront par l'être.

Quelques stratégies de gestion des transactions distribuées :

- **Le modèle SAGA** : C'est un ensemble de transactions locales qui sont exécutées de manière séquentielle. Si une transaction échoue, des actions compensatoires sont effectuées pour revenir à un état valide.
- **Deux-Phase Commit (2PC)** : Bien qu'il garantisse la cohérence stricte, ce modèle a un coût élevé en termes de performance et peut rencontrer des problèmes en cas de pannes réseau.

b. Exemples pratiques d'architecture orientée événements

Une architecture orientée événements (Event-Driven Architecture, EDA) est un modèle d'architecture où les composants du système réagissent à des événements qui se produisent au sein de l'application ou du système. Cela permet aux systèmes de communiquer de manière asynchrone et découplée, ce qui rend l'application plus flexible, évolutive et résiliente.

Exemple 1 : E-commerce - Gestion des commandes

Dans un système de e-commerce, lorsqu'un client passe une commande, un événement 'commande créée' est généré. Ce message peut être écouté par différents microservices, tels que :
- Un microservice de gestion des stocks qui vérifie la disponibilité des produits.
- Un microservice de paiement qui s'assure que le paiement est effectué.
- Un microservice d'envoi de notifications qui alerte le client que sa commande est en cours de traitement.

Cette architecture permet de séparer les différentes fonctionnalités tout en réagissant à un seul événement, le tout de manière asynchrone.

Exemple 2 : IoT - Gestion des capteurs
Dans un système IoT, chaque capteur envoie des données en temps réel (par exemple, température, humidité) sous forme d'événements. Ces événements sont traités par plusieurs services qui peuvent effectuer des actions différentes en fonction de la nature de l'événement. Par exemple :
- Un service de traitement des données peut analyser les températures pour détecter

des anomalies.
- Un service de stockage des données peut conserver les mesures dans une base de données.
- Un service de notification peut envoyer une alerte si la température dépasse un certain seuil.

Chapitre IV : Mise en Œuvre des Microservices avec Java

1. Présentation de Spring Boot et ses avantages pour les microservices

1.1. Qu'est-ce que Spring Boot ?

Spring Boot est une extension du Framework Spring qui facilite le développement d'applications Java en réduisant le besoin de configurations complexes. Il permet de créer rapidement des applications autonomes, prêtes à être exécutées en production.

Les principaux avantages de Spring Boot :

- Auto-configuration : Spring Boot détecte automatiquement les dépendances et configure les composants nécessaires.
- Serveur embarqué : Pas besoin d'installer Tomcat, tout est intégré dans l'application.
- Facilité de développement : Moins de code et de configuration manuelle.
- Compatible avec Spring Cloud : Idéal pour gérer des microservices.
- Support des API REST, bases de données et systèmes distribués.

1.2. Pourquoi utiliser Spring Boot pour les microservices ?

Dans une architecture microservices, chaque service est autonome et communique avec les autres via une API REST.
Spring Boot est idéal pour cela car :

- Il permet de démarrer un microservice en quelques minutes grâce à son écosystème prêt à l'emploi.
- Il facilite la communication entre services via Spring Cloud (Eureka, Feign, Config Server, etc.).
- Il s'intègre parfaitement avec Docker et Kubernetes pour le déploiement.

2. Création d'un microservice

Nous allons maintenant créer un microservice simple avec Spring Boot en utilisant IntelliJ IDEA.

2.1. Création du projet Spring Boot dans IntelliJ IDEA

Étape 1 : Ouvrir IntelliJ IDEA et créer un projet Spring Boot

a. Ouvrez IntelliJ IDEA.
b. Allez dans File → New → Project.
c. Sélectionnez Spring Initializr.
d. Cliquez sur Next.

Étape 2 : Configuration du projet

Dans la fenêtre de configuration :

- Group : com.example
- Artifact : product-service
- Type de projet : Maven
- Langage : Java
- Spring Boot Version : Choisissez la dernière version stable (ex: 3.1.x)
- Dépendances à ajouter :
  - Spring Web (pour exposer une API REST)
  - Lombok (pour simplifier la gestion des classes)
  - Spring Boot DevTools (pour rechargement automatique)
  - Spring Boot Actuator (pour la surveillance des services)

Cliquez sur Finish pour générer le projet.

2.2. Création d'une API REST basique

Nous allons maintenant définir une API REST qui retourne une liste de produits.
Étape 1 : Création de l'entité Product
Dans src/main/java/com/example/productservice/model, créez une classe Product.java :

```
package com.example.productservice.model;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class Product {
    private Long id;
    private String name;
    private double price;
}
```

Explication :

@Data : génère automatiquement les getters, setters et toString().

@AllArgsConstructor : génère un constructeur avec tous les arguments.

@NoArgsConstructor : génère un constructeur sans argument.

Étape 2 : Création du contrôleur REST ProductController

Dans src/main/java/com/example/productservice/controller, créez ProductController.java :

```
package com.example.productservice.controller;
```

```
import com.example.productservice.model.Product;
import org.springframework.web.bind.annotation.*;

import java.util.Arrays;
import java.util.List;

@RestController
@RequestMapping("/products")
public class ProductController {

    @GetMapping
    public List<Product> getAllProducts() {
        return Arrays.asList(
                new Product(1L, "Laptop", 1200.50),
                new Product(2L, "Smartphone", 750.99)
        );
    }

    @GetMapping("/{id}")
    public Product getProductById(@PathVariable Long id) {
        return new Product(id, "Sample Product", 100.0);
    }
}
```

Explication :

@RestController : indique que cette classe expose des API REST.

@RequestMapping("/products") : définit l'URL de base de l'API.

@GetMapping : expose des endpoints accessibles en HTTP.

Lancez le projet en exécutant ProductServiceApplication.java et testez l'API dans votre navigateur ou avec Postman :

- http://localhost:8080/products (affiche tous les produits)
- http://localhost:8080/products/1 (affiche le produit avec ID 1)

3. Gestion de la configuration et des dépendances

3.1. Configuration avec application.properties

Spring Boot permet de gérer les paramètres du microservice via le fichier application.properties.

Ajoutez ce fichier dans src/main/resources :

```
server.port=8081
spring.application.name=product-service
```

Explication :

- server.port=8081 : Définit le port du microservice.
- spring.application.name=product-service : Définit son nom dans un environnement distribué.

3.2. Gestion des dépendances avec Maven

Spring Boot utilise Maven pour la gestion des dépendances.
Dans pom.xml, vérifiez que les dépendances suivantes sont présentes :

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

Explication :

- spring-boot-starter-web : Pour créer des API REST.
- spring-boot-starter-actuator : Pour la surveillance du microservice.
- lombok : Pour réduire le code répétitif.

2. Intégration avec Spring Cloud
    a. Service Discovery (Eureka, Consul)

Le Service Discovery est un mécanisme important dans l'architecture des microservices. Il permet à chaque service de découvrir dynamiquement les autres services avec lesquels il doit communiquer, sans avoir besoin de connaître à l'avance leurs adresses. Cela est particulièrement utile dans des environnements dynamiques où les adresses IP des services peuvent changer fréquemment, comme dans les environnements cloud ou avec des containers Docker.

❖ **Eureka - Service Discovery de Spring Cloud**

Eureka est une solution populaire de Service Discovery fournie par Spring Cloud. Les services client peuvent s'enregistrer dans Eureka, et Eureka maintient un registre des

services disponibles. Chaque service, après avoir été enregistré, peut être découvert et appelé par d'autres services dans l'écosystème.

Étapes pour mettre en place Eureka :

- Ajouter la dépendance Eureka Server dans le fichier pom.xml de votre service de discovery :

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

- Ajouter l'annotation @EnableEurekaServer dans la classe principale pour activer Eureka.

```
@EnableEurekaServer
@SpringBootApplication
public class EurekaApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaApplication.class, args);
    }
}
```

- Configurer Eureka Server dans le fichier application.properties.

```
spring.application.name=eureka-server
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

- Chaque microservice qui veut se découvrir doit ajouter cette dépendance :

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

- Ajouter l'annotation @EnableEurekaClient dans les services qui se veulent être des clients.

```
@EnableEurekaClient
@SpringBootApplication
public class ProductServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(ProductServiceApplication.class, args);
    }
}
```

RECHERCHES POUR LES ETUDIANTS

b. API Gateway (Spring Cloud Gateway)

c. Configuration Spring Cloud Config

1. Sécurité dans les architectures distribuées

   a. Concepts d'authentification et d'autorisation

   b. Mise en œuvre de la sécurité dans les microservices (OAuth2, JWT)

   c. Sécurisation des communications (SSL/TLS)

Bibliographie et webographie

[1] « Gestion des données dans les applications natives cloud ». Consulté le: 12 février 2025. [En ligne]. Disponible sur: https://learn.microsoft.com/fr-fr/dotnet/architecture/cloud-native/distributed-data
[2] « Figure blockchain ». Consulté le: 12 février 2025. [En ligne]. Disponible sur: https://blog.alphorm.com/blockchain-architecture-securite-transactions
[3] « Architecture Netflix ». Consulté le: 12 février 2025. [En ligne]. Disponible sur: https://www.programmez.com/actualites/netflix-une-impressionnante-architecture-35899
[4] « chapitre3.i.pdf ».
[5] « CM_IS_objets_distribues.pdf ».
[6] « Representational_state_transfer ». Consulté le: 18 février 2025. [En ligne]. Disponible sur: https://fr.wikipedia.org/wiki/Representational_state_transfer
[7] « http-methods-explained-understanding-get-post-put-delete-shaaban-nqyyf ». Consulté le: 17 février 2025. [En ligne]. Disponible sur: https://www.linkedin.com/pulse/http-methods-explained-understanding-get-post-put-delete-shaaban-nqyyf/