# Technologies Java


Houssem MAHMOUDI
houssem.mahmoudi@ensicaen.fr

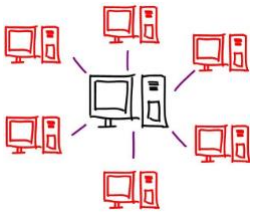


## Chapitre 2 : Application réparties et RMI

1. Application réparties
2. La technologie RMI
3. RMI vs Sockets
4. Architecture RMI
5. Stub
6. Skeleton
7. RMI Registry
8. Etapes pour créer une application RMI
9. Application Java RMI

# I. Application réparties

- Une application répartie est un système dont les composants sont répartis sur plusieurs machines connectées par un réseau.
- Ces composants communiquent et coordonnent leurs actions pour accomplir une tâche commune.
- Les systèmes répartis sont conçus pour améliorer :
  - la performance,
  - la disponibilité,
  - la tolérance aux pannes,
  - la scalabilité.



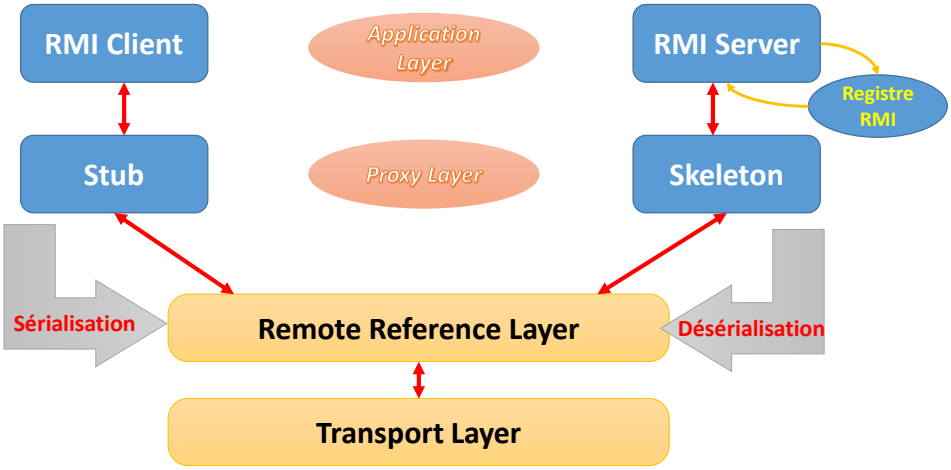


# II. La technologie RMI

- RMI (**R**emote **M**ethod **I**nvocation) est un mécanisme qui permet à un programme Java d'invoquer des méthodes sur un objet situé sur une autre machine distante.
- Avantages de RMI dans les systèmes répartis :
  - **Simplicité** : RMI permet de développer des applications réparties plus facilement en utilisant un modèle de programmation orienté objet.
  - **Sécurité** : RMI intègre des mécanismes de sécurité pour protéger les communications.
  - **Gestion des erreurs** : RMI fournit des mécanismes pour gérer les erreurs de communication et les exceptions.

# III. Architecture du système RMI

L'architecture RMI repose sur plusieurs composants clés qui permettent la communication entre les objets distants






# IV. RMI vs Sockets

| | RMI | Sockets |
|---|---|---|
| **Complexité de programmation** | Facile et simplifie le développement avec des objets distants et sérialisation automatique. | Complexe et nécessite de gérer la communication manuellement (connexion, protocole…) |
| **Communication** | Basée sur des appels de méthode distants. | Basée sur l'envoi et la réception de données binaires ou textuelles. |
| **Gestion des erreurs** | Fournit des mécanismes intégrés pour gérer les erreurs et les exceptions. | La gestion des erreurs doit être implémentée manuellement. |
| **Interopérabilité** | Limité (les deux parties doivent utiliser Java et connaître les objets distants). | Flexible (langages et formats variés peuvent être utilisés). |
| **Sécurité** | Plus lente en raison de l'abstraction et de la sérialisation. | Plus rapide grâce à l'accès direct au réseau. |

## V. Stub

- Le Stub est un proxy qui représente l'objet distant côté client. Il agit comme un intermédiaire entre le client et l'objet distant.
- Fonctionnement :
  - Le stub reçoit les appels de méthode du client.
  - Il sérialise les paramètres de la méthode (*marshalling*).
  - Il envoie la requête au serveur via le réseau.
  - Il reçoit la réponse du serveur, désérialise les résultats (*unmarshalling*), et les retourne au client.

↳ Masque la complexité de la communication réseau au client



## VI. Skeleton

- Le Skeleton est un proxy côté serveur qui reçoit les requêtes du stub et les transmet à l'objet distant.
- Fonctionnement :
  - Le Skeleton reçoit la requête du Stub.
  - Il désérialise les paramètres de la méthode (*unmarshalling*).
  - Il invoque la méthode sur l'objet distant.
  - Il sérialise le résultat de la méthode (*marshalling*) et l'envoie au stub.

↳ Masque la complexité de la communication réseau au client

# VII. Registre RMI (RMI Registry)

- Le registre RMI est un service qui permet de publier et de localiser des objets distants.
- Fonctionnement :
  - Le serveur enregistre un objet distant dans le registre RMI avec un nom unique (*Binding*).
  - Le client interroge le registre RMI pour obtenir une référence à l'objet distant (*Lookup*).

↳ Le registre RMI fonctionne comme un annuaire centralisé.

↳ Peut être démarré avec la commande **rmiregistry** (java\bin) ou en **code Java** et écoute par défaut sur le port **1099**.



# VIII. Etapes pour créer une application RMI

1. Créer une interface qui étend l'interface "java.rmi.**Remote**"
   - ↳ Doit contenir toute les méthodes abstraites.
   - ↳ Toutes les méthodes peut lever l'exception "**RemoteException**".
2. Créer une classe qui implémente l'interface et hérite de la classe "**UnicastRemoteObject**".
   - ↳ Créer le constructeur par défaut qui lui-même peut lever l'exception.
3. Créer le serveur :
   - ↳ Lancer rmiregistry sur le port **1099**.
   - ↳ Instancier la classe et publier l'objet dans l'annuaire.
4. Créer le client :
   - ↳ Créer le stub en récupérant la référence de l'objet distant.
   - ↳ Appeler les méthodes distantes en utilisant le stub.

# VIII. Application Java RMI

1. Créer une application Java en mode Client/Serveur permettant de convertir un montant en euros en bitcoins et d'afficher la date actuelle.
2. Le client doit invoquer les méthodes à distance sur le serveur en utilisant RMI (Remote Method Invocation).
3. Le serveur sera chargé d'exécuter les tâches demandées par le client :
   - Effectuer la conversion d'un montant euros en bitcoins.
   - Fournir la date actuelle.
4. Le serveur renverra les résultats des traitements au client.



# VIII. Correction

1. Créer une interface qui étend de l'interface **Remote**

```
package rmiserver;

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

public interface IBitcoin extends Remote {
    public double conversionEuro2Bitc(double euro) throws RemoteException;
    public String showDate() throws RemoteException;
}
```

# VIII. Correction

2. Créer une classe qui implémente l'interface et hérite de la classe **UnicastRemoteObject**.

```
package rmiserver;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.text.SimpleDateFormat;
import java.util.Calendar;

public class CBitcoin extends UnicastRemoteObject implements IBitcoin {
    public CBitcoin() throws RemoteException {
        super();
    }
    @Override
    public double conversionEuro2Bitc(double euro) throws RemoteException {
        return (euro * 0.000010);
    }
```



# VIII. Correction

2. Créer une classe qui implémente l'interface et hérite de la classe **UnicastRemoteObject**.

```
    @Override
    public String showDate() throws RemoteException {
        String date = new SimpleDateFormat("yyyy-MM-dd
        HH:mm:ss").format(Calendar.getInstance().getTime());
        return date;
    }
}
```

# VIII. Correction

3. Créer le serveur

```
package serveur;

import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;
import rmiserver.CBitcoin;

public class ServeurRMI {

public static void main(String[] args) {
    try {
        System.setProperty("java.rmi.server.hostname", "127.0.0.1");
        // Démarrer l'annuaire sur le port 1099
        LocateRegistry.createRegistry(1099);
        CBitcoin od = new CBitcoin(); //Création du skeleton grâce à UnicastRemoteObject
        // Returner l'adresse IP, le port et l'adresse mémoire
        System.out.println(od.toString());
```



# VIII. Correction

3. Créer le serveur

```
        //Publier l'objet dans l'annuaire :
        Naming.rebind("rmi://127.0.0.1:1099/ebitc", od);
    } catch (Exception e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
      }
    }
}
```

Annuaire

| Référence de l'objet distant | Nom de l'objet distant |
|---|---|
| rmi://127.0.0.1:1099/ebitc | od |

# VIII. Correction

4. Créer le client

```
package client;
import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.util.Scanner;
import rmiserver.IBitcoin;
public class ClientRMI {
    public static void main(String[] args) {
       try {
          IBitcoin stub = (IBitcoin) Naming.lookup("rmi://127.0.0.1:1099/ebitc");
          //Appel de la méthode showDate par le stub
          System.out.println("Nous sommes le : " +stub.showDate());
          Scanner scanner = new Scanner(System.in);
          System.out.print("Euro : ");
```



# VIII. Correction

4. Créer le client

```
          double euro = scanner.nextDouble();
          Double bitc = stub.conversionEuro2Bitc(euro);
          System.out.printf(euro + " € en Bitcoin = %.3f ", bitc);
       } catch (Exception e) {
          e.printStackTrace();
       }
    }
}
```