Projet : Jeu Sudoku en Réseau avec Java RMI

Groupe nnuméro 7 :
Nourhene Ayari
Adem Medyouni
Filière : IDS3
2024-2025

Table des matières

Introduction

Ce projet s'inscrit dans le cadre du cours de Systèmes et Applications Répartis (SAR) dispensé à la Faculté des Sciences de Tunis, sous l'encadrement de M. Heithem Abbes et Mme Thouraya Louati. Il a pour objectif de concevoir et développer une application Java distribuée basée sur la technologie RMI (Remote Method Invocation), permettant à plusieurs utilisateurs de jouer simultanément à un jeu de Sudoku en réseau.

À travers cette application, nous cherchons à mettre en œuvre les principes fondamentaux de la programmation répartie, notamment l'invocation de méthodes à distance, la gestion des interactions entre clients et serveur, ainsi que la synchronisation des données partagées. Une attention particulière est portée à l'ergonomie de l'interface graphique du client, à la validation des mouvements côté serveur, à la gestion des erreurs, ainsi qu'à la notification en temps réel des utilisateurs via des mécanismes de rappel (callbacks).

Ce projet représente une opportunité concrète d'explorer les défis liés à la conception d'applications interactives distribuées, tout en consolidant les acquis théoriques du cours à travers une implémentation pratique et fonctionnelle.

Chapitre 1

Objectifs du Projet

— Développer une application distribuée basée sur Java RMI.
— Partager une grille de Sudoku entre plusieurs clients connectés.
— Valider les mouvements côté serveur.
— Implémenter une interface graphique conviviale pour les utilisateurs.
— Gérer les erreurs et informer les clients via des callbacks.
— Ajouter une gestion du timeout pour les clients inactifs.

Chapitre 2

Architecture

L'architecture du projet est organisée autour de quatre packages :
— `shared` : contient les interfaces distantes utilisées par les clients et le serveur.
— `server` : gère la logique du jeu et la communication avec les clients.
— `client` : contient l'interface graphique du joueur.
— `model` : représente la grille et la logique de validation.

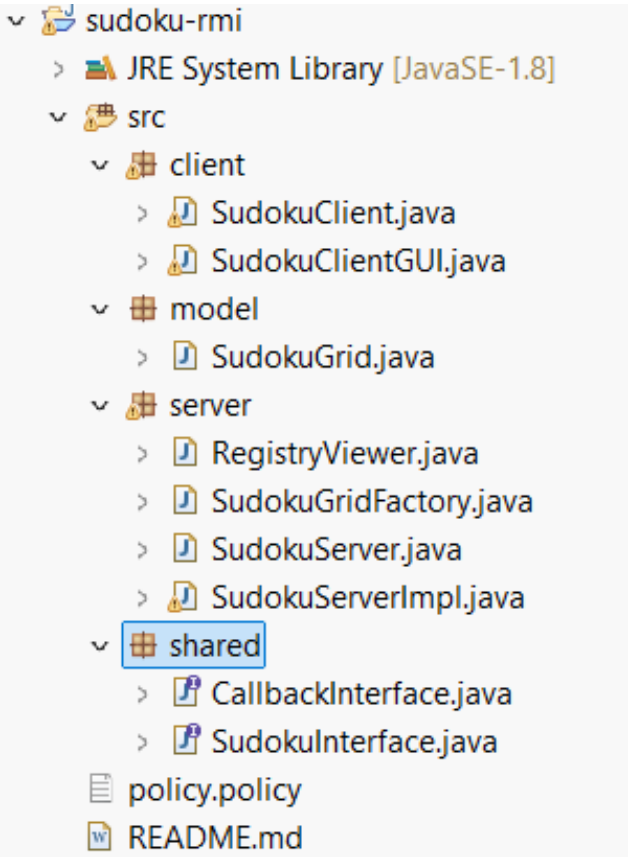


FIGURE 2.1 – Architecture du projet sous eclipse IDEA

Chapitre 3

Implémentation

3.1 Interface distante principale

L'interface distante principale sert à définir les méthodes que le serveur met à la disposition des clients dans une application Java RMI. Grâce à cette interface, les clients peuvent interagir avec le serveur à distance, par exemple pour envoyer des mouvements, recevoir des notifications ou consulter l'état du jeu. Elle joue un rôle essentiel en assurant la communication entre les différentes parties du système de façon claire et structurée.

Listing 3.1 – SudokuInterface

```
public interface SudokuInterface extends Remote {
    boolean submitMove(int row, int col, int value, String playerName
        ) throws RemoteException;
    int[][] getGrid() throws RemoteException;
    void registerCallback(CallbackInterface client) throws
        RemoteException;
}
```

3.2 Interface de Callback

L'interface de callback permet au serveur d'envoyer des messages ou des notifications vers les clients, même sans que ceux-ci fassent une requête. Elle est utilisée pour informer les clients en temps réel, par exemple lorsqu'un autre joueur effectue un mouvement ou qu'une erreur se produit. Cela rend l'interaction plus dynamique et réactive dans une application distribuée.

Listing 3.2 – CallbackInterface

```
package shared;

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

public interface CallbackInterface extends Remote {
    void notifyWin() throws RemoteException;
    void notifyError(String message) throws RemoteException;
```

3.3 Extrait du Serveur

La classe SudokuServer joue le rôle du serveur principal dans l'application. Elle gère la logique du jeu, comme la vérification des mouvements des joueurs, la gestion de la grille Sudoku, et la coordination entre les différents clients connectés. Elle utilise l'interface distante pour communiquer avec les clients et peut aussi déclencher des callbacks pour envoyer des mises à jour ou des messages d'erreur.

Listing 3.3 – Validation d'un mouvement

```
public boolean submitMove(int row, int col, int value, String
   playerName) throws RemoteException {
    if (!isValidMove(row, col, value)) {
        callback.notifyMessage("Mouvement invalide. Veuillez
           r essayer.");
        callback.updateGrid(grid); // envoyer la grille actuelle
        return false;
    }
    grid[row][col] = value;
    notifyAllClients();
    return true;
}
```

```
☑ Grille initiale choisie par le serveur :

========= Grille Sudoku =========
. . . . . . . 9 .
. . 6 . . . . . 3
. 8 . 5 . 6 . . .
. 3 . . . 5 . 7 .
. . . 7 . . . . .
. . . . 2 . 5 . 1
. . . . . . . . .
. 2 . . 7 4 9 8 .
. . . . 3 . 6 . .
================================

🚀 Serveur Sudoku prêt !
```

3.4 Extrait du SudokuServerImpl

La classe SudokuServerImpl est l'implémentation concrète de l'interface distante du serveur. Elle contient le code réel qui exécute les actions définies dans l'interface, comme valider les mouvements des joueurs, enregistrer les clients et gérer les callbacks. C'est cette classe qui est exportée et rendue accessible aux clients via RMI.

Listing 3.4 – Vérification d'inactivité

```
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool
   (1);
scheduler.scheduleAtFixedRate(() -> {
    long now = System.currentTimeMillis();
    for (ClientInfo c : clients) {
        if (now - c.getLastActivityTime() > 120000) {
            clients.remove(c);
            System.out.println("Client d connect  pour inactivit .
               ");
        }
    }
}, 0, 30, TimeUnit.SECONDS);
```

On peut lancer plus d'un client au même temps ;

Listing 3.5 – Parallélisme

```
this.pool = Executors.newFixedThreadPool(10);
```

On a aussi baser notre projet sur les threads synchronisés ;

```java
@Override
public synchronized boolean submitNumber(int row, int col, int number, CallbackInterface client) throws RemoteException {
    pool.submit(() -> {
        try {
            clientActivityMap.put(client, Instant.now()); // Mise à jour de l'activité du client
            int[][] gridBefore = cloneGrid(grid.getGrid()); // Sauvegarde de la grille avant modification

            if (number < 1 || number > 9) {
                StringBuilder errorMessage = new StringBuilder();
                errorMessage.append("❌ Erreur : la valeur ajoutée viole les conditions du jeu !\n");
                errorMessage.append(printGrid(gridBefore));
                client.notifyError(errorMessage.toString());
                return;
            }
```

3.5 Client Graphique

Le client peut effectuer les actions suivantes :

— Se connecter au serveur pour rejoindre une session de jeu.
— Envoyer ses mouvements (valeurs à placer dans la grille) au serveur.
— Recevoir des messages de validation ou d'erreur via le mécanisme de *callback*.
— Visualiser l'état actuel de la grille Sudoku.
— Être notifié des mises à jour ou des actions des autres joueurs.

Listing 3.6 – Extrait du SudokuClientGUI

```java
JButton sendButton = new JButton("Valider");
sendButton.addActionListener(e -> {
    int row = ...; // lecture ligne
    int col = ...; // lecture colonne
    int value = ...; // lecture valeur
    server.submitMove(row, col, value, playerName);
});
```

3.6 Callback côté Client

Listing 3.7 – Implémentation du callback

```java
public class ClientCallbackImpl extends UnicastRemoteObject
   implements CallbackInterface {
    public void notifyMessage(String message) throws RemoteException
       {
        JOptionPane.showMessageDialog(null, message);
    }

    public void updateGrid(int[][] newGrid) throws RemoteException {
        // mise    jour de l'interface graphique
    }
}
```

3.7 Enregistrement sous l'annuaire

Le `RegistryViewer` est un outil utilisé pour interagir avec le registre RMI. Il permet d'afficher les objets enregistrés dans le registre RMI de manière dynamique, facilitant ainsi le diagnostic et la gestion des services disponibles dans un environnement distribué. Son rôle principal est de vérifier si les objets sont correctement enregistrés et accessibles par les clients, ce qui est essentiel pour garantir la communication entre le serveur et les clients dans une architecture distribuée.

Voici l'implémentation du `RegistryViewer` :

```
package server;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class RegistryViewer {
    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.getRegistry("localhost
               ", 1099);
            String[] names = registry.list();
            System.out.println("Objets enregistr s dans le registre
               :");
            for (String name : names) {
                System.out.println(" - " + name);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

3.8 Gestionnaire de sécurité

Le fichier policy.policy est un outil essentiel pour assurer la sécurité dans les applications Java distribuées, notamment pour les applications utilisant Java RMI. En définissant des permissions appropriées pour les différents composants, il permet de contrôler l'accès aux ressources sensibles et de prévenir les attaques. Cependant, son utilisation nécessite une gestion prudente et bien pensée pour éviter des erreurs de configuration et garantir une sécurité optimale dans des systèmes distribués.

```
grant {
    // Permet toutes les op rations RMI
    permission java.net.SocketPermission "*:1024-", "connect,accept,
       resolve";
    permission java.net.SocketPermission "localhost:1024-", "connect,
       accept,resolve";

    // Permet de lire les classes distantes si tu utilises un
       codebase
    permission java.io.FilePermission "<<ALL FILES>>", "read";

    // Permet de cr er des threads, n cessaire pour ExecutorService
    permission java.lang.RuntimePermission "modifyThread";
    permission java.lang.RuntimePermission "modifyThreadGroup";

    // Pour RMI registry
    permission java.rmi.RMIPermission "useActivationSystem";
    permission java.security.AllPermission;
};
```

3.9 Conclusion

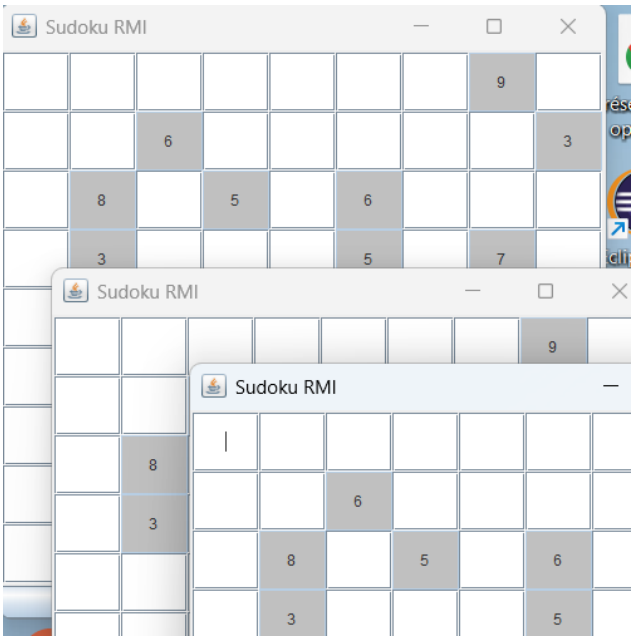
L'implémentation du jeu Sudoku en utilisant Java RMI a mis en œuvre les différents composants étudiés, tels que les interfaces distantes, les callbacks, le serveur, l'implémentation serveur et les clients. Chaque partie a joué un rôle essentiel dans l'interaction entre les utilisateurs et la gestion centralisée du jeu. Cette phase a démontré l'importance d'une bonne organisation des packages, d'une communication fluide entre les entités distribuées, ainsi que d'un traitement fiable des erreurs et des actions des joueurs. L'ensemble a abouti à un système fonctionnel, interactif et conforme aux exigences du projet.

Chapitre 4

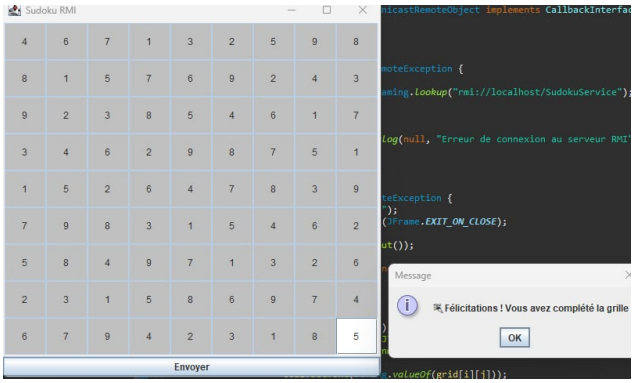
Tests et Résultats

4.1 Tests des différentes fonctionnalités de l'application

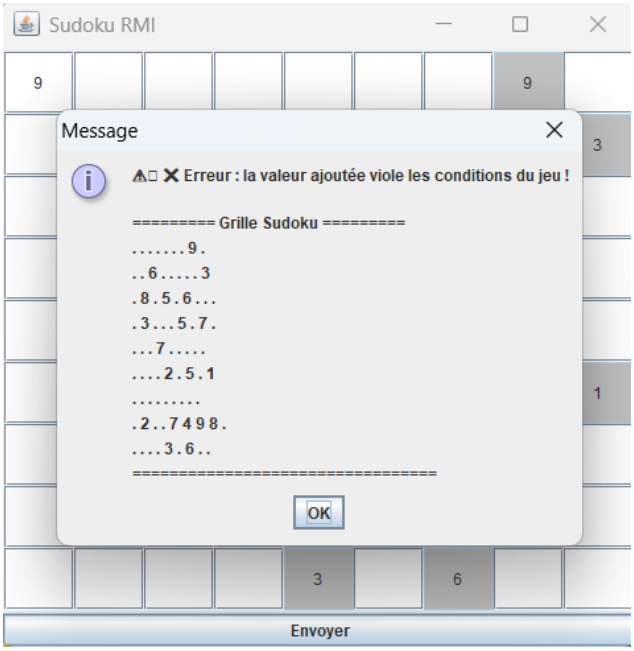
— Plusieurs clients peuvent jouer en parallèle.



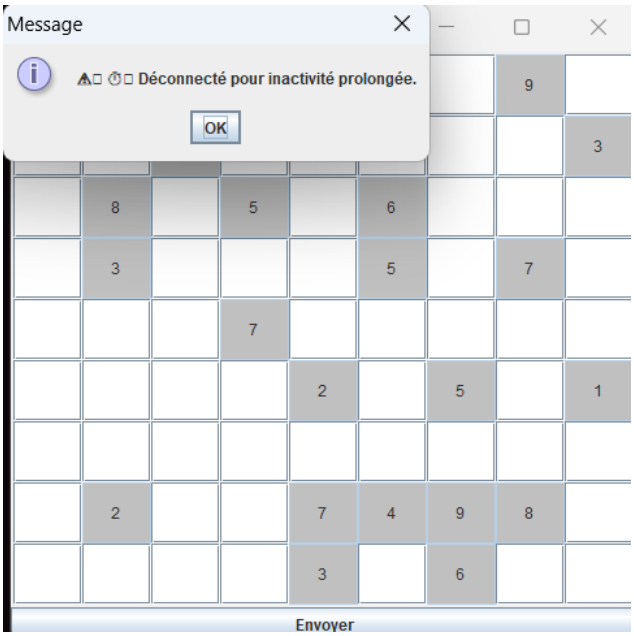
— La grille est synchronisée en temps réel entre tous les clients.
— Les règles du Sudoku sont respectées.



— Les messages d'erreur sont affichés immédiatement via le callback.



— Les clients inactifs sont automatiquement déconnectés.



4.2 Conclusion

Dans ce chapitre, nous avons validé l'efficacité et la robustesse de notre application à travers une série de tests. Les tests ont permis de vérifier le bon fonctionnement des interactions entre les clients et le serveur, notamment la gestion des mouvements, des erreurs, et des notifications. Les résultats ont démontré que le système fonctionne correctement dans des conditions variées, respectant les règles du jeu de manière fiable. Cependant, des améliorations peuvent encore être envisagées, notamment en matière de gestion des erreurs et de performance sous charge. Les tests ont aussi confirmé la stabilité du processus de déconnexion et la gestion des timeouts, garantissant une expérience utilisateur fluide.

Conclusion et Perspectives

Au terme de ce projet, nous avons conçu et implémenté un système distribué utilisant Java RMI pour simuler un jeu de Sudoku avec plusieurs clients interagissant avec un serveur central. Ce projet a permis d'explorer la puissance de RMI dans la création d'applications distribuées, offrant une communication transparente entre les objets distants. Toutefois, malgré les avantages indéniables de RMI, certains inconvénients ont été mis en évidence au cours de l'implémentation.
L'un des principaux inconvénients de l'utilisation du registre RMI réside dans sa centralisation et sa gestion statique. En effet, le RMI Registry repose sur un registre centralisé qui peut devenir un goulot d'étranglement en cas de nombreux services ou de clients concurrents. De plus, la gestion manuelle de l'enregistrement et de la recherche des objets distants peut conduire à des erreurs de synchronisation et de connectivité, surtout dans des environnements à grande échelle. Le registre ne propose pas non plus de mécanisme intégré pour la gestion des services à grande échelle, rendant la gestion de services distribués plus complexe.
En termes de performance, RMI peut rencontrer des limitations, notamment en ce qui concerne la gestion des objets distants volumineux ou la nécessité de transmettre des objets complexes entre le client et le serveur. De plus, bien que RMI soit assez bien intégré à l'écosystème Java, son utilisation dans des architectures hétérogènes ou dans des environnements cloud modernes peut être limitée.

Perspectives sur d'autres technologies

Afin de surmonter les limitations du RMI Registry, plusieurs alternatives technologiques peuvent être envisagées. Parmi elles, les microservices et les solutions basées sur des technologies telles que **gRPC** ou **RESTful APIs** apparaissent comme des choix pertinents. Ces technologies permettent une meilleure scalabilité, une plus grande flexibilité dans la gestion des services, et sont souvent plus adaptées à des environnements de cloud computing.
1. **gRPC** : Cette technologie permet la communication entre services via des protocoles plus modernes et efficaces, tels que HTTP/2. gRPC permet également d'automatiser la sérialisation des données et offre des mécanismes de gestion de la performance tels que le streaming bidirectionnel et le multiplexage. Elle est également compatible avec de nombreux langages, ce qui facilite l'interopérabilité dans des architectures hétérogènes.
2. **RESTful APIs** : Bien que plus simple à mettre en place que RMI, les API REST sont largement utilisées pour la communication entre services distribués. Elles sont particulièrement efficaces dans des environnements cloud et mobile, car elles reposent sur le protocole HTTP, largement supporté et accessible.

En conclusion, bien que RMI soit une technologie adaptée pour des applications distribuées simples, son utilisation dans des systèmes à grande échelle ou des environnements modernes peut rencontrer certaines limites. L'exploration de technologies plus récentes comme gRPC ou les architectures basées sur des microservices offre des perspectives intéressantes pour la création de systèmes distribués plus robustes et évolutifs.

Commandes d'Exécution

Compilation :

```
javac client/*.java model/*.java server/*.java shared/*.java
```

Démarrage du registre RMI :

```
start rmiregistry
```

Lancement du serveur :

```
java -Djava.security.policy=../src/security/policy.policy server.SudokuServer
```

Lancement du client :

```
java -Djava.security.policy=../src/security/policy.policy client.SudokuClientGUI
```

Libération du Port en cas d'utilisation en cours :

```
netstat -ano | findstr :1099
taskkill /PID <PID> /F
```