**Masina virtuala**

O mașină virtuală este un software care emulează comportamentul unei mașini fizice, permițând rularea unui sistem de operare și a aplicațiilor sale într-un mediu izolat și virtualizat. Această tehnologie este utilizată pentru a oferi portabilitate și flexibilitate, permițând rularea aceluiași software pe diferite platforme hardware și sisteme de operare. Mașinile virtuale sunt utilizate în diverse domenii, cum ar fi dezvoltarea software, testarea, consolidarea serverelor și securitatea informațiilor.

Java este un limbaj de programare de nivel înalt, conceput pentru a fi portabil și ușor de învățat. A fost dezvoltat de Sun Microsystems (și acum este deținut de Oracle Corporation). Java este popular pentru utilizarea sa în dezvoltarea de aplicații web, aplicații mobile, aplicații enterprise și multe altele. Este cunoscut pentru sintaxa sa simplă și clară, gestionarea automată a memoriei și securitatea încorporată.

JVM (Java Virtual Machine) este componenta cheie a platformei Java. Este o mașină virtuală care interpretează și execută bytecode-ul Java. JVM convertește codul sursă Java într-un format intermediar numit bytecode, care este apoi interpretat și executat de JVM. Aceasta oferă portabilitate, deoarece același bytecode Java poate fi executat pe diferite platforme hardware și sisteme de operare care au o implementare compatibilă a JVM.

Aplicațiile client-server sunt configurate pe o rețea (internet/intranet). Clienții trimit cereri către server pentru un resursă și, la rândul său, primesc răspunsuri de la server. Un calculator care poate trimite astfel de cereri pentru o resursă/serviciu este numit client, iar calculatorul care conține programul care furnizează resursa/serviciul solicitat mai multor clienți este numit server. Atât clienții, cât și serverele pot fi conectate prin intermediul unui protocol de rețea cablat/fără fir.

A computer and a cloud

Description automatically generated with medium confidence

În figura precedentă, comunicarea client-server poate fi vizualizată ca un program care rulează pe mașina clientului interacționând cu un alt program care rulează pe mașina serverului. Această comunicare prin rețea implică servicii de rețea oferite de diverse protocoale de comunicare.

Într-un sistem cu un singur procesor, aplicațiile pot comunica între ele prin intermediul memoriei partajate. Procesul producător scrie date în buffer sau în sistemul de fișiere, iar procesul consumator citește datele de acolo. În sistemele distribuite, nu există memorie partajată. În aceste sisteme, comunicarea aplicațiilor este intensă, deoarece acestea trebuie să se coordoneze între ele și să genereze rezultate în cel mai scurt timp posibil pentru comunicarea între procese. Ca rezultat, computerele folosesc diverse metode de comunicare între aplicațiile distribuite care pot fi localizate la distanță una de cealaltă.

Pentru a aborda această problemă, toate sistemele de comunicare sunt așteptate să respecte modelul Interconectării Sistemelor Deschise (OSI), care este un model conceptual care caracterizează și standardizează funcțiile de comunicare ale unui sistem de telecomunicații sau de calcul, indiferent de structura și tehnologia lor internă. Acest model împarte un sistem de comunicare în șapte straturi: fiecare strat servește stratul de deasupra și este servit de stratul de dedesubt. Printre acestea, stratul de legătură de date este unul dintre cele mai importante straturi, deoarece servește scopul detectării și corectării erorilor, asigurând astfel calitatea datelor. Acesta grupează biții în cadre și se asigură că fiecare cadru este primit corect. Plasează un model special la început și la sfârșitul fiecărui cadru pentru a-l marca; de asemenea, calculează un checksum prin adăugarea tuturor octeților la cadru într-o anumită ordine.

1. Strategia de detectare a erorilor: În această strategie, receptorul primește informații limitate. Poate doar să detecteze că a apărut o eroare și să respingă ulterior mesajul, dar nu poate corecta eroarea.
2. Strategia de corectare a erorilor: În această metodă, receptorul primește informații duplicate, fiecare bloc referindu-se la blocul anterior. Dacă apare o eroare, informațiile din blocul următor pot fi folosite pentru a corecta eroarea chiar la receptor. De exemplu, dacă un cadru constă din i biți de date și duplicate d biți de date, atunci lungimea totală ar fi i + d = n. Această lungime totală n este numită un cuvânt de cod. Aceste cuvinte de cod sunt ulterior comparate pentru a găsi numărul de biți cu care diferă. Numărul de poziții de biți în care cuvintele de cod consecutive diferă se numește distanța Hamming.

Există diferite metode de comunicare a aplicațiilor la distanță disponibile pe care le poți utiliza pe o rețea, implicate în următoarele tipuri:

* Network protocol stack
* Remote Procedure Call (RPC)
* Remote Method Invocation (RMI)
* Message queuing services (sockets)
* Stream-oriented services

RMI este o extensie specifică Java a Remote Procedure Call (RPC), orientată pe obiecte. Oferă un mecanism pentru crearea de aplicații distribuite bazate pe Java. Permite unui obiect dintr-o Mașină Virtuală Java (JVM) să interacționeze cu un alt obiect din altă JVM, prin invocarea metodelor din acel obiect. De aceea, o aplicație construită cu ajutorul RMI este considerată o aplicație care poate rula pe mai multe JVM-uri.

RMI facilitează comunicarea între aplicații care sunt desfășurate pe servere diferite și conectate în mod remote folosind obiecte numite stub și schelet. Această arhitectură de comunicare face ca o aplicație distribuită să pară ca un grup de obiecte care comunică printr-o conexiune la distanță. Aceste obiecte sunt encapsulate prin expunerea unei interfețe, care ajută la accesarea stării și comportamentului privat al unui obiect prin metodele sale

Diagrama următoare arată cum se întâmplă RMI între clientul RMI și serverul RMI cu ajutorul registrului RMI:

A diagram of a computer server

Description automatically generated

REGISTRUL RMI este un registru de obiecte remote, un serviciu de nume Bootstrap, care este utilizat de SERVERUL RMI pe același gazdă pentru a lega obiecte remote la nume. Clienții de pe gazde locale și remote apoi caută obiectele remote și fac invocări de metode la distanță. Mai jos sunt câteva dintre considerațiile de proiectare ale aplicațiilor construite cu un RMI:

* Invocarea transparentă a metodelor pe obiecte create pe mai multe JVM-uri.
* Asigurarea integrării ușoare a invocării metodelor la distanță cu logica generală de programare fără IDL extern, păstrând în același timp cea mai mare parte a semanticilor de obiect Java.
* Configurarea capacității de a distinge între un model de obiect distribuit și unul local.
* Ajutorul în construirea aplicațiilor fiabile menținând în același timp siguranța și securitatea Java.
* Extinderea suportului către mai multe protocoale de transport, diverse semanticile de referință, cum ar fi persistența și activarea leneșă, și diversele mecanisme de invocare.
* Tratarea problemelor cum ar fi rularea în spații de memorie diferite, trecerea parametrilor, legarea datelor și alte eșecuri ale RPC.
* Tratarea oricăror probleme suplimentare, cum ar fi colectarea de gunoi distribuită și încărcarea de clase distribuită.

Iată câteva dintre terminologiile importante utilizate într-o Invocare de Metode la Distanță:

* Obiect remote: Acesta este un obiect într-o anumită JVM al cărui metode sunt expuse astfel încât să poată fi invocate de un alt program desfășurat pe o altă JVM.
* Interfață remote: Aceasta este o interfață Java care definește metodele care există într-un obiect remote. Un obiect remote poate implementa mai mult de o interfață remote pentru a adopta mai multe comportamente de interfață remote.
* RMI: Aceasta este o modalitate de a invoca metodele unui obiect remote cu ajutorul unei interfețe remote. Poate fi realizată cu o sintaxă similară invocării metodelor locale.
* Stub (bucată): Acesta este un obiect Java care acționează ca punct de intrare pentru obiectul client pentru a ruta orice solicitări externe. Există în JVM-ul clientului și reprezintă manerul către obiectul remote. Dacă un obiect invocă o metodă pe obiectul stub, stub-ul stabilește RMI urmând acești pași:

1. Inițiază o conexiune către JVM-ul mașinii remote.
2. Serializează (scrie și transmite) parametrii transmiși către el prin JVM-ul remote.
3. Așteaptă un răspuns de la obiectul remote și deserializează (citește) valoarea sau excepția returnată, apoi răspunde apelantului cu acea valoare sau excepție.

* Schelet (Skeleton): Acesta este un obiect care acționează ca o poartă pe partea serverului. Acționează ca un obiect remote cu care obiectele client interacționează prin stub. Acest lucru înseamnă că orice solicitări venind de la clientul remote sunt rutate prin acesta. Dacă scheletul primește o solicitare, stabilește RMI prin acești pași:

1. Citește parametrul trimis metodei remote.
2. Invocă metoda reală a obiectului remote.
3. Serializa (scrie și transmite) rezultatul înapoi către apelant (stub).

Diagrama următoare demonstrează comunicarea RMI cu implicația stub-ului și scheletului:

A diagram of a server

Description automatically generated

Obiectele remote sunt similare obiectelor locale în ceea ce privește invocarea metodelor și returnarea rezultatelor din metode. Un obiect remote poate fi tipizat la oricare dintre interfețele remote acceptate de implementarea sa, iar operatorul instanceof poate fi folosit pentru a verifica tipul unui obiect remote sau local.

Cu toate că există similitudini între obiectele remote și cele locale menționate mai sus, acestea se comportă diferit. În timp ce clienții obiectelor remote pot interacționa doar cu interfețele remote, nu cu clasele de implementare, clienții obiectelor locale pot interacționa atât cu interfețele, cât și cu clasele de implementare. RMI este trecut printr-o valoare, în timp ce o invocare de metodă locală poate fi trecută printr-o referință. În acest context, excepțiile remote obligatorii trebuie definite într-un sistem local.

Pașii esențiali ce trebuie urmați pentru dezvoltarea unei aplicații distribuite cu RMI sunt următorii:

1. Proiectați și implementați un component care nu ar trebui să fie implicat doar în aplicația distribuită, ci și în componente locale.
2. Asigurați-vă că componentele implicate în apelurile RMI sunt accesibile peste rețea.
3. Stabiliți o conexiune de rețea între aplicațiile care trebuie să interacționeze folosind RMI.

În lista anterioară, cel mai important pas, care trebuie efectuat cu atenție, este definirea arhitecturii corecte în aplicație, cu o distincție clară între componentele care acționează ca obiecte Java disponibile pe o JVM locală și cele care sunt accesibile în mod remote. Să revizuim pașii de implementare în detaliu:

1. Definirea interfeței remote: Scopul definirii unei interfețe remote este de a declara metodele care ar trebui să fie disponibile pentru invocarea de către un client remote. Programarea interfeței în loc de programarea implementării componente este un principiu de proiectare esențial adoptat de toate cadrele moderne Java, inclusiv Spring. În același model, definirea unei interfețe remote capătă importanță și în proiectarea RMI.
2. Implementarea obiectului remote: Java permite unei clase să implementeze mai mult de o interfață în același timp. Acest lucru ajută obiectele remote să implementeze una sau mai multe interfețe remote. Clasa obiectului remote poate fi necesar să implementeze și alte interfețe locale și metode pentru care este responsabilă. Evitați adăugarea de complexitate în acest scenariu, în ceea ce privește modul în care argumentele sau valorile de returnare ale acestor metode ale componentei ar trebui scrise.
3. Implementarea clientului remote: Obiectele client care interacționează cu obiectele server remote pot fi scrise odată ce interfețele remote sunt definite cu atenție, chiar și după ce obiectele remote sunt implementate.

Acum să luăm în considerare un exemplu simplu și cum putem să-l implementăm pentru a stimula interacțiunea între mai multe JVM-uri folosind Java RMI.

Presupunem că construim o aplicație pentru a gestionarea unui stoc de produse. Să proiectăm un proiect care să poată fi găzduit pe un server.

package com.mycompany.proiect;

import java.rmi.registry.LocateRegistry;

import java.rmi.registry.Registry;

public class StockServer {

    public static void main(String[] args) {

        try {

            StockService stockService = new StockServiceImpl();

            Registry registry = LocateRegistry.createRegistry(1099);

            registry.rebind("StockService", stockService);

            System.out.println("Serverul RMI este pornit...");

        } catch (Exception e) {

            System.err.println("Eroare la pornirea serverului RMI: " + e.getMessage());

            e.printStackTrace();

        }

    }

}

Această clasă StockServer este responsabilă pentru pornirea serverului RMI și înregistrarea serviciului nostru StockService. Acest cod Java este o aplicație server care utilizează Java RMI (Remote Method Invocation) pentru a oferi servicii unor clienți. Mai întâi, creează o instanță a clasei StockServiceImpl, care implementează interfața StockService, ce definește operațiile disponibile pentru gestionarea stocului. Apoi, creează un registru RMI pe portul 1099 folosind LocateRegistry.createRegistry(1099). Aplicația își leagă apoi serviciul StockService de acest registru cu registry.rebind("StockService", stockService), astfel încât clienții să poată găsi și să utilizeze acest serviciu. Dacă există vreo eroare în timpul acestor operații, aceasta este afișată în consolă.

package com.mycompany.proiect;

import java.rmi.RemoteException;

import java.rmi.server.UnicastRemoteObject;

import java.util.HashMap;

import java.util.Map;

public class StockServiceImpl extends UnicastRemoteObject implements StockService {

    private Map<String, Integer> stock;

    public StockServiceImpl() throws RemoteException {

        super();

        this.stock = new HashMap<>();

    }

    public void addProduct(String productName, double price, int quantity) throws RemoteException {

        if (quantity < 0) {

            System.out.println("Cantitatea trebuie sa fie pozitiva.");

            return;

        }

        stock.put(productName, quantity);

        System.out.println("Produs adaugat in stoc: " + productName + ", pret: " + price + " lei, cantitate: " + quantity);

    }

    public boolean checkAvailability(String productName, int quantity) throws RemoteException {

        int availableQuantity = stock.getOrDefault(productName, 0);

        return availableQuantity >= quantity;

    }

    public void placeOrder(String customerName, String[] orderDetails) throws RemoteException {

        System.out.println("Comanda plasata de catre " + customerName + ":");

        for (int i = 0; i < orderDetails.length; i += 2) {

            String productName = orderDetails[i];

            int quantity = Integer.parseInt(orderDetails[i + 1]);

            int availableQuantity = stock.getOrDefault(productName, 0);

            if (availableQuantity >= quantity) {

                System.out.println("- " + productName + ": " + quantity);

                stock.put(productName, availableQuantity - quantity);

            } else {

                System.out.println("Stoc insuficient pentru produsul " + productName + ".");

            }

        }

    }

    public void displayStock() throws RemoteException {

        System.out.println("Stoc:");

        for (Map.Entry<String, Integer> entry : stock.entrySet()) {

            System.out.println("- " + entry.getKey() + ": " + entry.getValue());

        }

    }

}

Această clasă Java StockServiceImpl este o implementare a interfeței StockService și este destinată să fie utilizată în cadrul aplicației server RMI. Ea gestionează stocul de produse și oferă operații pentru adăugarea de produse în stoc, verificarea disponibilității, plasarea comenzilor și afișarea stocului curent. Clasa utilizează UnicastRemoteObject pentru a expune obiectul la invocare de la distanță prin RMI. Ea folosește o structură de date HashMap pentru a stoca produsele și cantitățile asociate lor. Operațiile oferite de această clasă aruncă RemoteException pentru a gestiona erorile de comunicare RMI.

package com.mycompany.proiect;

import java.rmi.Remote;

import java.rmi.RemoteException;

public interface StockService extends Remote {

    void addProduct(String productName, double price, int quantity) throws RemoteException;

    boolean checkAvailability(String productName, int quantity) throws RemoteException;

    void placeOrder(String customerName, String[] orderDetails) throws RemoteException;

    void displayStock() throws RemoteException;

}

Această este o interfață Java numită StockService, care definește operațiile disponibile pentru gestionarea stocului de produse în cadrul aplicației RMI. Interfața extinde Remote, indicând faptul că metodele sale pot fi invocate de la distanță prin RMI. Metodele definite în această interfață includ adăugarea de produse în stoc, verificarea disponibilității, plasarea comenzilor și afișarea stocului curent. Toate metodele aruncă RemoteException pentru a gestiona erorile de comunicare RMI.

package com.mycompany.proiect;

import java.rmi.registry.LocateRegistry;

import java.rmi.registry.Registry;

public class ProiectClient {

    public static void main(String[] args) {

        try {

            // Obține referința la registrul RMI

            Registry registry = LocateRegistry.getRegistry("localhost", 1099);

            // Obține referința la serviciul StockService din registrul RMI

            StockService stockService = (StockService) registry.lookup("StockService");

            // Utilizarea serviciului RMI pentru a executa instrucțiunile

            stockService.addProduct("Laptop", 2500, 10);

            stockService.addProduct("Telefon", 1500, 20);

            boolean availableLaptop = stockService.checkAvailability("Laptop", 3);

            boolean availablePhone = stockService.checkAvailability("Telefon", 25);

            System.out.println("Disponibilitate Laptop: " + availableLaptop);

            System.out.println("Disponibilitate Telefon: " + availablePhone);

            // Plasarea unei comenzi

            String[] orderDetails = {"Laptop", "2", "Telefon", "5"};

            stockService.placeOrder("John\_Doe", orderDetails);

            // Afișarea stocului actualizat

            stockService.displayStock();

        } catch (Exception e) {

            System.err.println("Eroare la conectarea la serverul RMI: " + e.getMessage());

            e.printStackTrace();

        }

    }

}

Acesta este un client Java care utilizează serviciul RMI pentru a interacționa cu serverul care gestionează stocul de produse. În main method, clientul obține referința la registrul RMI local, apoi caută serviciul StockService în registrul respectiv. După obținerea referinței la serviciu, clientul folosește metodele definite în interfața StockService pentru a adăuga produse în stoc, a verifica disponibilitatea acestora, a plasa comenzi și a afișa stocul actualizat. În caz de eroare în timpul conexiunii la serverul RMI sau în timpul executării operațiilor, se afișează un mesaj de eroare și se tipărește stack trace-ul corespunzător.

Bibliografie

1. Goldberg, R.P. (1974). "Survey of Virtual Machine Research". IEEE Computer Society.
2. Popek, G.J.; Goldberg, R.P. (1974). "Formal Requirements for Virtualizable Third Generation Architectures". Communications of the ACM.
3. Smith, J.M.; Nair, R. (2005). "Virtual Machines: Versatile Platforms for Systems and Processes". Morgan Kaufmann.
4. Lester, J.; Ho, T. (2010). "Virtual Machine Design and Implementation in C/C++". Wordware Publishing.
5. "Virtualization Essentials" by Matthew Portnoy. Sybex.
6. Schildt, H. (2014). "Java: The Complete Reference, Ninth Edition". McGraw-Hill Education.
7. Venners, B. (1997). "Inside the Java Virtual Machine". McGraw-Hill Education.
8. Oaks, S. (2018). "Java Performance: The Definitive Guide". O'Reilly Media.
9. Lindholm, T.; Yellin, F. (2014). "The Java Virtual Machine Specification, Java SE 8 Edition". Addison-Wesley Professional.
10. Holmes, D. (2001). "The Java Language Specification, Third Edition". Addison-Wesley Professional.
11. Pattamsetti, R. M. R. (2017). “Distributed Computing in Java 9”. Packt Publishing Ltd.