5. Proiectare de detaliu și implementare

1. Arhitectura generală a aplicației
2. Scopul general al aplicației

Aplicația dezvoltată își propune să ofere un instrument software interactiv și extensibil pentru simularea, modelarea și analizarea strategiilor de alocare a resurselor, implicit de planificare a sarcinilor în infrastructuri de tip cloud computing, având ca scop principal reducerea consumului de energie din centrele de date, respectiv evaluarea performanțelor fiecărei politici de planificare într-un mediu controlat si configurabil. Aceasta are o valore practică ridicată deoarece oferă un cadru experimental în care pot fi testate, comparate și analizate diverse metode de planificare a sarcinilor, fie că sunt din categoria celor clasice, fie metaeuristice, toate acestea fiind simulate într-un mediu cu condiții realistice de funcționare.

Prin intermediul aplicației, utilizatorul are posibilitatea de a configura în detaliu infrastructura virtuală, prin setarea unor paramentri din interfața grafică, precum algoritmul de planificare a sarcinilor, îmbunătățirea sau nu a eficienței energetice, numărul de host-uri și mașini virtuale, capacitatea de procesare exprimată în MIPS, adică Million Instuctions Per Second, memoria RAM disponibilă, lățimea de bandă a rețelei, capacitatea de stocare, precum și numărul de sarcini, altfel spus cloudlet-uri, împreună cu caracteristicle acestora reprezentate de lungime, deadline, respectiv prioritate. După definirea parametrilor, utilizatorul poate lansa simularea, astfel aplicația creând infrastructura și aplicând algoritmul de planificare, rulând procesul de simulare în cadrul simulatorului CloudSim7G, rezultatele fiind colectate automat, prelucrate și afișate în interfață.

Așadar, aplicația nu doar că rulează simulari, ci oferă și o interfață grafică intuitivă, în care utilizatorul poate vizualiza în timp real datele rezultate, le poate salva automat într-o bază de date și le poate reîncărca ulterior pentru a realiza o analiză comparativă între simulări. Prin această abordare modulară, aplicația se dovedește a fi un instrument complet pentru cercetare, testare și demonstrație practică în domeniul de cloud computing.

Cu alte cuvinte, se oferă un cadru practic pentru studierea și compararea performanței diverselor politici de planificare si alocare a resurselor, punându-se accentul pe indicatori de performanță relevanți în contextul actual al centrelor de date, precum consumul energetic, timpul de execuție, utilizare resurselor, gradul de încărcare al mașinilor virtuale și echilibrarea sarcinilor.

1. Fluxul de date între componente

În cadrul aplicației, fluxul de date este organizat într-o manieră logică, clară și coerentă, respectând principiile separării responsabilităților. Astfel, procesul începe atunci când utilizatorul introduce in interfața grafică toți parametrii necesari pentru simulare, aceștia fiind preluați de către controlerul aplicației, care transmite la rândul său motorului de simulare pentru procesare, ce va instanția infrastructura virtuală, cum ar fi datacentere, host-uri, mașini virtuale și task-uri. După configurare, se va aplica algoritmul de planificare selectat, iar simularea va fi lansată folosind CloudSim7G, la final rezultatele fiind colectate si procesate.

Datele rezultate sunt trimise apoi în două direcții. Pe de o parte, sunt afișate în interfața grafică sub formă de tabele și grafice, pentru a permite utilizatorului o vizualizare clară si imediată, iar pe de alta parte, sunt salvate automat într-o bază de date relaționară, structurată astfel încât fiecare simulare este înregistrată împreună cu parametrii săi, valorile rezultate, respectiv graficele obținute, utilizatorul având posibilitatea ulterior să încarce simulările anterioare, fie pentru revizuire, fie pentru o comparație detaliată, fără a relansa procesul de simulare.

Așadar, aplicația funcționeză ca un sistem bidirecțional între utilizator si simulator, asigurând atât controlul complet asupra parametrilor de intrare, cât și accesul organizat la rezultatele obținute.

1. Prezentarea generală a funcționalităților

Aplicația dezvoltată oferă un set complet de funcționalități care acoperă toate etapele unui proces de simulare și analiză a planificării sarcinilor într-un mediu de tip cloud, acestea fiind integrate într-o interfață grafică accesibilă și organizate în module separate, permițând extinderea lor facilă.

Funcționalitatea de configurare a parametrilor permite utilizatorului să definească în detaliu atât în mod fizic, cât si logic infrastructura care urmează să fie simulată. Aceasta include numărul de mașini virtuale, caracteristicile fiecărei mașini virtuale, lungimea sarcinilor, politici de planificare a sarcinilor, respectiv dacă are selectată eficiența energetică sau nu. Toți acești parametri sunt preluați prin controale grafice intuitive, cum ar fi textbox-uri, dropdown-uri, radio button și multe altele, fiind validate si transmise către backend.

Astfel, după configurare, utilizatorul va putea lansa simularea apăsând butonul corespunzător, etapă în care aplicația construiește infrastructura în CloudSim, rulează simularea, respectiv calculează automat indicatorii de performanță, proces complet automatizat și transparent pentru utilizator.

Rezultatele obținute sunt afișate în interfață sub formă de tabel, în care fiecare rând corespunde unui cloudlet, iar coloanele includ indicatori de performanță precum ID-ul cloudlet-ului, durata de execuție, ID-ul mașinii virtuale, ID-ul host-ului, timpul de început și de sfârșit al execuției, consumul energetic, respectiv starea finală a sarcinii. De asemenea, nu putem omite nici faptul că aplicația oferă posibilitatea generării de grafice, care permit analiza vizuală a acestor date, evidențiind diferențele dintre algoritmi sau scenarii.

Un alt aspect important este funcționalitatea de a salva, în mod automat, într-o bază de date rezultatele după fiecare simulare, împreună cu toți parametrii definiți de utilizator. Această funcționalitate asigură nu doar păstrarea rezultatelor pentru documentare sau raportare, ci și posibilitatea reîncărcării lor ulterioare fără a relua simularea pentru eventualele comparații. Astfel, utilizatorul poate să acceseze simulările anterioare dintr-o listă afișată într-o fereastră dedicată, iar după selectarea unei simulări, datele corespunzătoare sunt preluate din baza de date și afișate atât în format tabelar, cât și sub formă de grafic, analizându-se cu ușurință eficiența energetică, respectiv timpul de execuție.

În ansamblu, aplicația oferă un mediu complet de configurare, simulare, analiză, salvare, respectiv reîncărcarea datelor, devenind un instrument valoros în procesul de cercetare și dezvoltare în domeniul de cloud computing.

1. Descrierea componentelor aplicației

Pentru a evidenția relațiile dintre clasele componente și arhitectura modulară a aplicației, în figura următoare este prezentată o diagramă UML de clase care reflectă structura principalelor entități Java uilizate în aplicație.

A screenshot of a computer

AI-generated content may be incorrect.

Figura 5.2 Diagrama UML

1. Interfața grafică (GUI)

Interfața grafică reprezintă principalul punct de interacțiune dintre utilizator și aplicație, având rolul de a facilita configurarea parametrilor de simulare, lansarea procesului de rulare, precum și vizualizarea și interpretarea rezultatelor. Aceasta a fost proiectată pentru a oferi o experiență de utilizare intuitivă, organizată logic si estetic, astfel încât să poată fi utilizată atât de persoane cu cunoștințe tehnice, cât și de utilizatori mai puțin experimentați.

Pentru a evidenția modul în care utilizatorul interacționează cu aplicația, în figura următoare este prezentată o diagramă de tip Use Case, care descrie relația dintre actorul principal, adică utilizatorul, împreună cu funcționalitățile oferite de interfața grafică, de la inițializarea simulării și selectarea parametrilor, până la salvarea, încărcarea și analiza rezultatelor.

A diagram of a diagram

AI-generated content may be incorrect.

Figura 5.2.1 Diagrama Use Case

În ceea ce privește tehnologiile utilizate, pentru a permite integrarea ușoară cu codul de simulare scris în Java, această interfață grafică a fost realizată folosind JavaFX, o bibliotecă modernă și flexibilă pentru dezvoltarea de aplicații desktop în Java, ce oferă suport nativ pentru componente grafice interactive, tabele, grafice, respectiv layout-uri personalizabile.

Pentru stilizarea interfeței, s-a utilizat CSS, ceea ce a permis separarea logicii aplicației de partea de prezentare. De asemenea, structura vizuală a interfeței a fost declarată într-un mod modular și ușor de întreținut folosind JavaFX FXML. Aceste tehnologii au oferit atât flexibilitate în dezvoltare, cât și un grad ridicat de control asupra elementelor UI.

A screenshot of a computer

AI-generated content may be incorrect.

Figura 5.2.1 Interfața principală a aplicației după rularea unei simulări

Interfața principală a aplicației este structurată clar în două secțiuni importante, așa cum se poate observa în Figura 5.2.1 Interfața principală a aplicației după rularea unei simulări. Așadar, în partea stângă a ferestrei se regăsește zona dedicată configurării parametrilor de simulare, unde utilizatorul are posibilitatea de a introduce valorile necesare pentru definirea infrastructurii cloud și a scenariului testat. Astfel, această zonă cuprinde câmpuri text pentru specificarea numărului de host-uri, capacitatea în MIPS a fiecărui host, memoria RAM a host-ului în gigabytes, precum și numărul total de mașini virtuale care urmează a fi create. Pentru fiecare mașină virtuală se pot seta valorile MIPS, RAM, lățimea de bandă exprimată în megabytes pe secundă, spațiul de stocare în megabytes și numărul de core-uri disponibile pentru fiecare host. De asemenea, utilizatorul are la dispoziție și un câmp pentru a defini numărul total de cloudlet-uri, adică sarcinile care vor fi distribuite și executate pe infrastructura virtuală definită.

Tot în partea stângă a interfeței se regăsesc și două optiuni de tip radio button care permit alegerea modului de rulare a simulării, adică fie sa ruleze folosind eficiența energetică, fie fără, în mod normal. Alegerea acestei opțiuni influențează comportamentul aplicației în raport cu adaptarea resurselor în timpul execuției, pentru a oferi o scădere considerabilă a consumui de energie. Sub această secțiune se află un dropdown din care utilizatorul poate selecta politica de planificare a sarcinilor disponibile în aplicație. După ce trecem de zona de selecție a algoritmului, se află patru butoane de acțiune, și anume butonul de „Run Simlation” ce declanșează procesul de simulare pe baza parametrilor setați, butonul de „Suggest Resources” ce oferă o funcționalitate de suport prin care aplicația poate sugera o configurație oprimă pentru infrastructură, în funcție de complexitatea sarcinilor, butonul de „Save Resuls” ce permite salvarea rezultatelor obținute într-o bază de date, respectiv „Load Results”, ce oferă posibilitatea de a accesa simulări anterioare fără a le rula din nou.

În partea dreaptă a interfeței se află zona de afișare a rezultatelor, sub forma unui tabel care reflectă detaliile fiecărui cloudlet procesat în timpul simulării. Așadar, tabelul este organizat în mai multe coloane care oferă informații relevante cum ar fi identificatorul fiecărui cloudlet, starea finală a acestuia reprezentată de SUCCESS sau FAILED, mașina virtuală pe care a fost programat, host-ul fizic pe care rulează respectiva mașină virtuală, momentul de start al execuției, timpul de finalizare, durata efectivă de procesare și nu în ultimul rând, energia consumată pentru respectiva sarcină. Sub tabel, aplicația afișează în mod automat câteva informații esențiale, ce oferă utilizatorului o imagine de ansamblu clară asupra performanței algoritmului selectat și asupra eficienței configurației introduse în procesul de simulare, precum energia totală consumată în cadrul simulării, denumirea algoritmului utilizat, timpul real necesar pentru rularea completă a simulării, respectiv durata totală de execuție a tutuor cloudlet-urilor.

1. Componenta de simulare

Componenta de simulare reprezintă nucleul aplicației, fiind responsabilă de modelarea și rularea efectivă a infrastructurii cloud definite de utilizator, alocarea sarcinilor conform politicii de planificare selectate, respectiv de colectare a tuturor rezultatelor aferente execuției.

Această componentă a fost construită peste simulatorul CloudSim 7G, care permite simularea detaliată a centrelor de date virtualizate și a comportamentului resurselor în medii distribuite.

Așadar, pentru a răspunde nevoii de a analiza comportamentul sistemului atât în scenarii statice, normale, în conditii obișnuite, cât și în scenarii cu resurse adaptabile pentru scăderea consumului de energie, au fost implementate două clase principale de simulare, și anume EnergySimulatorNormal, respectiv EnergySimulatorDynamic.

Astfel, clasa EnergySimulatorNormal este utilizată în simulările normale și folosește o abordare utilă pentru analiza clasică a algoritmilor de planificare, fără intervenții de ajustare a resurselor în timpul execuției. Aceasta construiește o infrastructură stabilă, în care toate mașinile virtuale și host-urile păstrează caracteristicile definite pe tot parcursul rulării. Metoda principală a clasei, runSimulator, afișată mai jos, este una statică și primește ca parametri toți indicatorii relevanți necesari pentru definirea configurației, cum ar fi numărul de host-uri împreună cu caracteristicile acestora, parametrii pentru mașinile virtuale, numărul de sarcini, precum și denumirea algoritmului de planificare care urmează să fie aplicat.

public static String runSimulation(int *numHosts*, int *hostMIPS*, int *hostRAM*,int *numVMs*, int *vmMIPS*, int *vmRAM*, long *vmBW*, long *vmSize*, int *pesNumber*, int *numCloudlets*, String *algorithmName*) {  
 try {  
 // Inițializare CloudSim  
 int numUsers = 1;  
 Calendar calendar = Calendar.getInstance();  
 CloudSim.init(numUsers, calendar, false);  
 // Creare Datacenter cu parametrii corecți  
 Datacenter datacenter = createDatacenterNormal("Datacenter\_0", *numHosts*, *hostMIPS*, *hostRAM*);  
 // Creare Broker  
 DatacenterBroker broker = new DatacenterBroker("Broker");  
 // Creare Cloudlets  
 List<Cloudlet> cloudletList = createCloudlets(broker.getId(), *numCloudlets*);  
 broker.submitCloudletList(cloudletList);  
 // Alegere algoritm de scheduling  
 SchedulingAlgorithm algorithm;  
 switch (*algorithmName*) {  
 case "ACO":  
 algorithm = new ACO();  
 break;

//...  
 default:  
 algorithm = new MinLengthRoundRobin();  
 }  
 // Creare VM-uri   
 List<Vm> vmList = createVMs(broker.getId(), *numVMs*, *vmMIPS*, *vmRAM*, *vmBW*, *vmSize*, *pesNumber*);  
 broker.submitGuestList(vmList);  
 algorithm.runAlgorithm(broker, vmList, cloudletList);  
 CloudSim.startSimulation();  
 return printResultsStringBuilder(broker, vmList, algorithm);

} catch (Exception *e*) {  
 *e*.printStackTrace();  
 return "Simulation failed due to an error!";  
 }  
}

Pe baza acestor informații, este construit un centru de date virtual, sunt generate resursele necesare și este selectat algoritmul corespunzător dintr-o colecție de implementări compatibile cu interfața SchedulingAlgorithm. Logica de alocare a sarcinilor este delegată acestei instanțe de algoritm, în timp ce infrastructura de bază rămâne neschimbată pe parcursul execuției. Este important să menționăm că rezultatele sunt preluate direct din obiectele generate de broker, fiind procesate și returnate sub formă de text structurabil, ce poate fi afișat în interfața grafică sau salvat în baza de date. Așadar, această clasă este responsabilă de rularea simulărilor în cazul în care resursele sunt constante, oferind un punct de referință stabil pentru compararea algoritmilor în absența mecanismelor create special pentru reducerea consumului energetic.

Nu putem omite nici clasa EnergySimulatorDynamic, care este responsabilă de scenariile ce joacă un rol foarte important în reducerea consumului de energie. În acest caz, componentele infrastructurii pot suferi modificări pe parcursul rulării, cum ar fi activarea sau dezactivarea mașinilor virtuale sau ajustarea MIPS, simulând astfel un comportament adaptiv al centrului de date. În plus, această clasă gestionează în mod suplimentar momentele de scalare si replanificare, iar în funcție de algoritmul utilizat, poate relansa task-uri sau redistribui sarcini pentru a oferi o scădere considerabilă a consumului de enrgie.

Această clasă păstrează structura generală a clasei prezentate anterior, dar introduce două funcționalități esențiale pentru modelarea realistă a comportamentului centrelor de date moderne și anume scalarea dinamică a mașinilor virtuale, respectiv consolidarea resurselor.

Scalarea dinamică este implementată prin metoda scaleUpVMs, care analizează în timp real nivelul de utilizare a CPU-ului de către mașinile virtuale și adaugă mașini virtuale noi dacă încărcarea medie depășește un anumit prag, mai exact 80%, iar sarcinile aflate în așteptare sunt mai numeroase decât mașinile virtuale disponibile. Astfel, sistemul simulat poate răspunde la creșteri bruște ale cererii, alocând resurse suplimentare doar când este necesar și evitând risipa.

if (avgCpuUtilization > SCALE\_UP\_THRESHOLD && waitingCloudlets > activeVMs) {  
 int newVMs = Math.min(waitingCloudlets / 2, *maxVMs* - activeVMs);  
 for (int i = 0; i < newVMs; i++) {  
 int newId = activeVMs + i;  
 Vm vm = new Vm(newId, *broker*.getId(), 500, 1, 2048, 1000, 10000, "Xen", new CloudletSchedulerTimeShared());  
 *vmList*.add(vm);  
 }  
 lastScaleTime= CloudSim.clock();  
}

Acest fragment de cod exemplifică modul în care se creează dinamic mașini virtuale noi, atunci când sistemul detectează solicitarea infrastructurii actuale, iar pentru a preveni o reacție excesivă, este inclus și un mecanism de încetinire, care blochează scalarea repetată într-un interval scurt. Astfel, vom folosi : if (CloudSim.clock() - lastScaleTime< SCALING\_DELAY) return;

Complementar mecanismului de scalare, metoda consolidateVMs simulează dezactivarea mașinilor virtuale care nu sunt necesare. Aceasta este o strategie extrem de eficientă din punct de vedere energetic, ce are ca rezultat limitarea numărului de mașini virtuale active la strictul necesar.

int activeVMs = Math.min(*vmList*.size(), (*cloudletList*.size() / 3) + 1);   
for (int i = activeVMs; i < *vmList*.size(); i++) {  
 *vmList*.get(i).setMips(0); // Oprire VM-uri neutilizate  
}

Acest cod stabilește câte mașini virtuale sunt necesare, folosind aproximativ o mașină virtuală la fiecare 3 cloudlet-uri, celelalte fiind oprite simbolic prin setarea puterii de procesare, adică MIPS, la zero. Această modificare forțează CloudSim să ignore alocarea sarcinilor ulterioare, simulând astfel oprirea acestora pentru o scădere a consumului de energie.

O metodă specială, denimită fakeConsolidateVMs, este folosită uneori pentru a forța simulatorul să stabilească corect maparea dintre mașina virtuală si host chiar și în absența unei consolidări reale. Aceasta este importantă strict pentru afișarea corectă a hostId-ului asociat fiecărei mașini virtuale.

public static void fakeConsolidateVMs(List<Vm> *vmList*, List<Cloudlet> *cloudletList*) {  
 int activeVMs = *vmList*.size() - 1 ;  
 for (int i = activeVMs; i < *vmList*.size(); i++) {  
 *vmList*.get(i).setMips(0);  
 }  
}

Atât clasa EnergySimulatorNormal, cât si EnergySimulatorDynamic moștenesc funcționalități comune de inițializare, rulare a simulării, respectivprelucrare a rezultatelor, dar se diferențiază prin comportamentul dinamic sau static asupra infrastructurii.

Înainte de lansarea unei simulări, utilizatorul definește parametrii necesari în interfața grafică, unde, ulterior, aceste date vor fi preluate de controller, apoi transmise către simulator pentru a crea instanțele necesare. Așadar, host-urile sunt inițializate cu un anumit număr de core-uri, o capacitate totală de procesare exprimată în MIPS, memorie RAM, respectiv lățimea de bandă, servind drept noduri fizice care vor găzdui mașinile virtuale. În plus, mașinile virtuale sunt create în funcție de numărul dorit și caracteristicile definite de utilizator, fiind asociate uletrior unui host, în funcție de politica de alocare și de capacitățile disponibile. Nu putem omite nici cloudlet-urile, adică sarcinile propriu-zise ce urmează a fi executate, care sunt generate pe baza lungimii lor și a numărului total specificat. Acestea sunt adăugate într-o listă gestionată de un DatacenterBroker, entitate care coordonează planificarea sarcinilor pe mașinile virtuale. Așadar, faza de setare prezentată este esențială pentru asigurarea corectitudinii simulării, deoarece reflectă în mod fidel complexitatea și dinamica unui sistem real de tip cloud.

Este important să menționăm faptul că în cadrul componentei de simulare, sunt integrați mai mulți algoritmi de planificare, fiecare cu propriul comportament și logică de alocare a cloudlet-urilor către mașinile virtuale. Acești algoritmi sunt selectați din interfața grafică, iar clasa de simulare apelează, în mod dinamic, metoda asociată algoritmului selectat. Printre algoritmii integrați se regăsesc Round Robin, FCFS, Min-Min, Max-Min, Random Scheduler, LJF, ACO, PSO, respectiv Genetic Algorithm, fiecare dintre aceștia fiind implementați într-o clasă dedicată sau în metode specifice, apelate de simulator în momentul atribuirii sarcinilor. Astfel, aceste particularități sunt esențiale în comparațiile de performanță și în studiile privind consumul energetic total sau timpul de execuție global.

Odată ce infrastructura este instanțiată și algoritmul este aplicat, simularea este lansată utilizând funcționalitățile oferite de CloudSim 7G. Cu alte cuvinte, simularea este de tip eveniment discret, adică fiecare acțiune realizată în simulator va fi gestionată în timp virtual, CloudSim colectând în fundal date privind timpul de start, timpul de finalizare, durata de procesare, respectiv consumul de resurse.

În final, la încheierea simulării, aceste date sunt extrase printr-un proces de prelucrare, formând o lista de obiecte de tip CloudletResults, unde fiecare obiect conține toate informațiile legate de execuția fiecărui task. Aceste obiecte sunt afișate în interfața grafică împreună cu valori cumulative precum energia totală consumată, respectiv timpul total de execuție al simulării și în paralel, pot fi salvate într-o bază de date pentru o analiză ulterioară.

1. Algoritmi de planificare a sarcinilor

Componenta de planificare a sarcinilor joacă un rol esențial în eficiența procesului de simulare, deoarece determină modul în care sarcinile, adică cloudlet-urile, sunt distribuite către mașinile virtuale disponibile. Astfel, în cadrul aplicației dezvoltate, s-au implementat zece algoritmi de planificare, fiecare urmărind un anumit obiectiv de performanță, și anume reducerea timpului de execuție, echilibrarea încărcării sau optimizarea consumului energetic. Algoritmii sunt scalabili din interfața grafică, iar în momentul rulării simulării, aplicația invocă automat clasa și metoda corespunzătoare algoritmului ales de utilizator.

Așadar, pentru a facilita implementarea modulară si extensibilă a diverselor politici de planificare a sarcinilor, aplicația definește o interfață comună, denumită SchedulingAlgorithm, care stabilește o convenție unificată de lucru pentru toate clasele de algoritmi implementați. Astfel, fiecare algoritm, indiferent de logica sa internă, respectă aceeași semnătură de metodă, permițând apelarea sa în mod generic de către simulator. Interfața conține o singură metodă abstractă reprezentată de:

void runAlgorithm(DatacenterBroker *broker*, List<Vm> *vmList*, List<Cloudlet> *cloudletList*).

Această metodă primește ca parametri un obiect DatacenterBroker, lista de mașini virtuale vmList, respectiv lista de sarcini cloudletList. Implementările acestei metode sunt responsabile de atribuirea fiecărei sarcini către o anumită mașină virtuală, pe baza politicii proprii de scheduling. Prin această structură, aplicația permite adăugarea rapidă de noi algoritmi, fără a fi necesare modificări în clasele de control sau în simulatorul principal. Cu alte cuvinte, această arhitectură respectă principiile programării orientate pe obiect și permite extinderea ușoară a aplicației.

1. FCFS

Nu putem omite prezentarea algoritmilor, iar algoritmul First Come First Served este situat în clasa FCFS din pachetul algorithms și este una dintre cele mai simple și directe strategii de planificare a sarcinilor, fiind adesea utilizat ca referință în comparațiile dintre algoritmi. Logica algoritmului presupune ca fiecare sarcină din lista să fie alocată către o mașină virtuală în ordinea apariției, fără a ține cont de dimensiunea task-ului, resursele disponibile ale mașinilor virtuale sau durata estimată de execuție, metodă ce presupune o procesare secvențială, echitabilă, lipsită de optimizare.

Astfel, pentru fiecare cloudlet, se selectează o mașină virtuală pe baza indexului, ceea ce înseamnă ca sarcinile sunt distribuite uniform pe toate mașinile virtuale disponibile. Avantajul principal al algoritmului constă în simplitatea sa, deoarece nu necesită calcule suplimentare, nu introduce overhead de sortare sau analiză și este foarte rapid de executat. Este ideal pentru simulări de bază, testare inițială a infrastructurii sau ca punct de comparație cu alți algoritmi mai sofisticați.

1. Round Robin

Următorul algoritm menționat este Round Robin, ce se află în clasa cu același nume, care folosește o stategie clasică de planificare a sarcinilor și urmărește distribuirea echilibrată a acestora către resursele disponibile. Principiul de funcționare al algoritmului este simplu, și anume fiecare cloudlet este atribuit unei mașini virtuale în ordine ciclică, adică după ce ultima mașină virtuală din listă primește o sarcină, următoarea sarcină revine primei mașini virtuale, iar procesul continuă până la epuizarea sarcinilor. Această rotație asigură o distribuție uniformă și previne supraîncărcarea uneia dintre mașinile virtuale.

Pentru afișarea unui exemplu, implementarea propriu-zisă a metodei runAlorithm este următoarea:

public void runAlgorithm(DatacenterBroker *broker*, List<Vm> *vmList*, List<Cloudlet> *cloudletList*) {  
 int vmIndex = 0;  
 for (Cloudlet cloudlet : *cloudletList*) {  
 Vm vm = *vmList*.get(vmIndex);  
 cloudlet.setVmId(vm.getId());  
 *broker*.bindCloudletToVm(cloudlet.getCloudletId(), vm.getId());  
 vmIndex = (vmIndex + 1) % *vmList*.size();  
 }  
 System.out.println("Round Robin applied!");  
}

Astfel, fiecare sarcină este legată explicit de o mașină virtuală prin metoda borker.bindCloudletToVm, ceea ce asigură în mod formal că respectiva sarcină va fi programată pe mașina virtuală specificată. Indexul mașinii virtuale este actualizat la fiecare iterație, conform expresiei „(vmIndex + 1) % vmList.size()”. Această abordare face ca RoundRobin să fie potrivit pentru scenariul în care toate sarcinile sunt similare ca durată și complexitate, iar obiectivul este de a menține un grad egal de utilizare a mașinilor virtuale. Totuși, dacă sarcinile sunt inegale ca dimensiune, pot apărea dezechilibre în execuție, deoarece algoritmul nu ia în considerare timpul estimat de procesare sau încărcarea curentă a mașinilor virtuale.

Spre deosebire de FCFS, care atribuie sarcinile în funcție de poziția lor în listă, RoundRobin aduce un plus de echitate prin rotație, fiind util în simulări cu task-uri omogene și infrastructuri distribuite.

1. Random Scheduler

Pe lângă algoritmii care urmează o logică secvențială sau ciclică, aplicația include și strategii cu caracter aleatoriu, cum ar fi algoritmul Random Scheduler, care oferă o abordare nedeterministă în planificarea sarcinilor, distribuind cloudlet-urile către mașinile virtuale disponibile într-o manieră aleatorie. Acesta este implementat într-o clasă dedicată numită RandomScheduler și respectă interfața SchedulingAlgorithm, asigurând compatibilitatea cu restul aplicației.

Spre deosebire de algoritmii menționați anterior, Random Scheduler nu urmează o regulă fixă de distribuție, ci selectează în mod aleatoriu o mașină virtuală pentru fiecare sarcină. Această metodă este utilă mai ales pentru simulări comparative, teste de stres sau pentru studierea comportamentului sistemului în scenarii imprevizibile.

1. Longest job First

Spre deosebire de algoritmii care nu țin cont de dimensiunea sarcinilor, algoritmul Longest Job First, situat în clasa LJF, reprezintă o strategie de planificare orientată pe prioritizarea sarcinilor cu durată mare de execuție, care încearcă să elimine timpii mari de așteptare generați de sarcinile lungi prin plasarea acestora la începutul cozii de execuție. Implementarea metodei este următoarea:

public void runAlgorithm(DatacenterBroker *broker*, List<Vm> *vmList*, List<Cloudlet> *cloudletList*) {  
 *cloudletList*.sort(Comparator.comparingLong(Cloudlet::getCloudletLength).reversed());  
 int vmIndex = 0;  
 for (Cloudlet cloudlet : *cloudletList*) {  
 Vm vm = *vmList*.get(vmIndex);  
 cloudlet.setVmId(vm.getId());  
 *broker*.bindCloudletToVm(cloudlet.getCloudletId(), vm.getId());  
 vmIndex = (vmIndex + 1) % *vmList*.size();  
 }  
 System.out.println("Longest Job First applied!");  
}

Așadar, la începutul execuției, lista de cloudlet-uri este sortată descrescător în funcție de lungimea sarcinii utilizând metoda „getCloudletLength()”, astfel cele mai costisitoare sarcini urmând a fi procesate primele, și ulterior distribuite ciclic către mașinile virtuale disponibile.

Avantajul acestui algoritm constă în reducerea timpului total de execuție pentru sarcinile lungi și în îmbunătățirea predictibilității execuției în scenarii mai complexe. Totuși, acest algoritm poate fi dezavantajos pentru sarcinile scurte, care ajung să fie executate ultimele, ceea ce poate duce la latențe ridicate pentru task-uri mici sau critice. Algoritmul este potrivit pentru simulări în care se dorește evaluarea performanței în prezența unor cloudlet-uri de dimensiuni foarte mari și în care maximizarea utilizării mașinilor virtuale în faza inițială este prioritară.

1. Max-Min

Un pas către planificarea informată este realizat prin integrarea algoritmului Max-Min, care analizează timpii de completare înainte de alocare. Cu alte cuvinte, acest algoritm, aflat în clasa Max-Min, oferă o strategie de planificare orientată către procesarea eficientă a sarcinilor cu durată mare de execuție, încercând în același timp să evite blocajele și întârzierile.

Logica algoritmului presupune următorii pași, și anume pentru fiecare cloudlet neatribuit, se determină combinația cu cel mai mic timp estimat de finalizare, adică acea mașină virtuală care îl poate executa cel mai rapid în condițiile date, apoi, dintre toate aceste valori minime, se selectează cloudlet-ul care are cel mai mare timp minim de completare, de unde și numele algoritmului Max-Min. Astfel, prin această abordare, acest algoritm reușește sa prioritizeze sarcinile cu durata mare, atribuindu-le mașinilor virtuale care le pot finaliza cel mai rapid, reducându-se semnificativ riscul ca sarcinile complexe sa fie programate ultimele și să întârzie finalizarea totală a simulării.

Un avantaj important este că acest algoritm reușeste să echilibreze execuția între mașinile virtuale, ținând cont de estimările dinamice ale timpului de procesare, și de asemenea, ține cont și de starea curentă a mașinilor virtuale, implicit de sarcinile deja atribuite, ajustând în mod continuu planificarea în funcție de evoluția execuției. Totuși, această abordare presupune un timp de calcul mult mai ridicat față de algoritmii simpli, prezentați anterior, datorită parcurgerii tuturor combinațiilor dintre sarcină și mașină virtuală, motiv pentru care este recomandat simulărilor unde acuratețea rezultatelor este mult mai importantă decat viteza de execuție a algoritmului de planificare.

1. Min-Min

Dacă algoritmul Max-Min a fost conceput pentru a acorda prioritate sarcinilor complexe, alocând cele mai lungi sarcini în funcție de combinația lor optimă cu mașinile virtuale, o strategie complementară este oferită de algoritmul Min-Min, ce se află în clasa MinMin. Așadar, algoritmul Min-Min oferă o strategie de planificare orientată spre execuția rapidă a sarcinilor scurte, ideea principală fiind selectarea, la fiecare pas, a combinației dintre cloudlet și mașina virtuală care are cel mai mic timp estimat de finalizare, apoi blocarea acelei combinații, respectiv acutalizarea sistemului.

Min-Min are avantajul de a reduce timpul timpul total de execuție în scenarii în care există un număr mare de task-uri scurte. Nu putem omite nici faptul că permite o încărcare treptată și echilibrată a mașinilor virtuale, datorită execuției rapide a sarcinilor inițiale. Ca dezavantaj, putem menționa faptul ca poate duce la întârzierea execuției pentru sarcinile lungi, care ajung sa fie planificate abia la final.

Această abordare este recomandată pentru mediile cloud în care se dorește o procesare rapidă și eficientă a task-urilor ușoare, dar poate fi combinat și cu alte strategii în cadrul unor politici hibride pentru a gestionarea sarcinilor complexe în mod eficient.

1. Min Length Round Robin

Dacă algoritmul Min-Min prioritizează sarcinile scurte în funcție de timpul minim estimat de execuție, următoarea strategie propune o soluție mai simplă dar eficientă, menținând accentul pe task-urile cu durata redusă. Min Length Round Robin, aflat în clasa cu aceeasi denumire, îmbină două concepte esențiale, și anume sortarea sarcinilor în funcție de lungimea lor, respectiv distribuirea echilibrată în stil Round Robin, ceea ce oferă o variantă hibridă, ușor de implementat, dar capabilă să asigure o bună utilizare a resurselor. Astfel, prin această abordare, sarcinile scurte sunt programate primele, iar alocarea pe mașinile vortuale se face în mod egal, indiferent de caracteristicile hardware ale acestora, oferind un echilibru între eficiență și simplitate.

Cu alte cuvinte, acest algoritm este potrivit pentru simulări în care task-urile au lungimi foarte variate, dar unde se dorește o execuție rapidă și relativ echilibrată a sacinilor simple, fiind util în special în testarea sarcinilor cu trafic intens de task-uri mici.

1. ACO

Este important sa menționăm algoritmul metaeuristic Ant Colony Optimization, sau mai bine zis ACO, ce se afla în clasa cu același nume, abordare inspirată din comportamentul natural al coloniilor de furnici și introduce un model probalistic de alocare a sarcinilor, bazat pe acumularea și evaporarea informației de tip „feromon digital”. Acest algoritm este adaptat pentru probleme complexe de planificare în care spațiul de căutare este foarte mare și nu poate fi explorat eficient prin metode deterministe clasice.

Altfel spus, ACO este un algoritm metaeuristic care simulează modul în care furnicile reale descoperă trasee optime către surse de hrană, lăsând urme de feromoni care ghidează la rândul lor, alte furnici. În contextul aplicației, această logică este transpusă prin intermediul unei populații de obiecte Ant, fiecare reprezentând o sarcină, adică Cloudlet, care trebuie atribuită unei mașini virtuale Vm. Atribuirea nu se face direct, ci în baza unui proces iterativ în care probabilitatea de alegere a unei mașini virtuale crește dacă alocarea anterioară s-a dovedit „benefică”, pe baza unui criteriu implicit, de exemplu timpul scurt de execuție.

Clasa ACO implementează interfața SchedulingAlgorithm și conține mai mulți parametri specifici algoritmului, cum ar fi numărul de furnici, rata de evaporare a feromonilor, cantitatea de feromon adăugată la fiecare pas, respectiv numărul total de iterații. Obiectele Ant sunt create pentru fiecare cloudlet, iar pentru fiecare iterație se simulează atribuirea în mod aleatoriu a acestora către mașinile virtuale, urmată de actualizarea hărții de feromoni. Putem expune aceste aspecte oferind exemplul din cod:

public void runAlgorithm(DatacenterBroker *broker*, List<Vm> *vmlist*, List<Cloudlet> *cloudletList*) {  
 this.vmList = *vmlist*;  
 this.cloudletList = *cloudletList*;  
 initializePheromoneMap();  
 List<Ant> ants = new ArrayList<>();  
 for (int it = 0; it< iterations; it++) {  
 ants = createAnts();  
 for (Ant ant : ants) {  
 Vm selectedVm = selectVm(ant);  
 ant.setVm(selectedVm);  
 updatePheromone(ant, selectedVm);  
 }  
 evaporatePheromones();  
 }  
 for (Ant ant : ants) {  
 *broker*.bindCloudletToVm(ant.getCloudlet().getCloudletId(), ant.getVm().getId());  
 }  
 System.out.println("ACO Algorithm applied!");  
}

Clasa conține mai multe metode auxiliare care susțin logica principală a algoritmului, și anume metoda initializePheromoneMap() are rolul de a inițializa nivelul de feromon pentru fiecare pereche formată din cloudlet și mașină virtuală cu o valoare de pornire egală cu 1.0.

private void initializePheromoneMap() {  
 pheromoneMap = new HashMap<>();  
 for (Cloudlet cloudlet : cloudletList) {  
 for (Vm vm : vmList) {  
 String key = generateKey(cloudlet.getCloudletId(), vm.getId());  
 pheromoneMap.put(key, 1.0);  
 }  
 }  
}

Populația de furnici va fi simulată prin crearea unui obiect de tip Ant pentru fiecare cloudlet din listă folosind metoda createAnts().

private List<Ant> createAnts() {  
 List<Ant> ants = new ArrayList<>();  
 for (Cloudlet cloudlet : cloudletList) {  
 ants.add(new Ant(cloudlet));  
 }  
 return ants;  
}

Pentru actualizarea nivelului de feromon pentru o anumită alocare se va folosi metoda updatePheromones().

private void updatePheromone(Ant *ant*, Vm *selectedVm*) {  
 String key = generateKey(*ant*.getCloudlet().getCloudletId(), *selectedVm*.getId());  
 pheromoneMap.put(key, pheromoneMap.getOrDefault(key, 1.0) + pheromoneIncrease);  
}

În paralel, metoda evaporatePheromones() aplică un mecanism de evaporare globală a feromonului, reducând valorile acumulate pentru toate combinațiile, prevenind suprasaturarea deciziilor anterioare.

private void evaporatePheromones() {  
 for (String key : pheromoneMap.keySet()) {  
 pheromoneMap.put(key, pheromoneMap.get(key) \* (1 - evaporationRate));  
 }  
}

În final, generateKey() generează un identificator unic pentru fiecare pereche de cloudlet și mașină virtuală, fiind folosit în cadrul hărții de feromoni pentru a menține individualitatea fiecărei relații posibile.

private String generateKey(int *cloudletId*, int *vmId*) {  
 return *cloudletId* + ":" + *vmId*;  
}

Avantajul acestui algoritm constă în capacitatea sa de a explora un spațiu mare de soluții posibile, de a învăța din experiențele anterioare, respectiv de a converge treptat către o soluție aproape optimă. ACO este potrivit pentru aplicații de planificare a sarcinilor în scenarii reale cu cerințe multiple, resurse limitate și obiective contradictorii, precum reducerea timpului de execuție și optimizarea consumului energetic.

1. PSO

După cum s-a observat în cazul algoritmului ACO, strategiile inspirate din comportamente naturale oferă o alternativă puternică față de metodele clasice de planificare. Astfel, un alt exemplu relevant din aceaași categorie metaeuristică este algoritmul Particle Swarm Optimization, care simulează mișcarea particulelor într-un roi, respectiv modul în care acestea cooperează pentru a identifica o soluție optimă. Așadar, PSO aflat în clasa cu denumire similară este un algoritm evolutiv, care modelează procesul de optimizare prin colaborarea unui grup de particule, fiecare reprezentând o posibilă soluție. Astfel, în cadrul aplicației, fiecare particulă este o distribuție potențială a cloudlet-urilor către mașinile virtuale, reprezentată sub forma unui vector de poziții, fiecare poziție din acest vector indicând indexul mașinii virtuale pe care este programată o anumită sarcină.

Acest algoritm este implementat într-o clasă care respectă interfața SchedulingAlgorithm, iar metoda principală runAlgorithm definește logica completă a procesului de optimizare. Inițial, se generează o populație de 30 de particule, fiecare cu o distribuție aleatorie a sarcinilor, apoi pentru fiecare particulă se evaluează calitatea soluției, adică fitness-ul, care este calculat pe baza duratei maxime de execuție dintre toate mașinile virtuale, scopul fiind minimizarea respectivei valori. Dacă o particulă reușește să obțină un fitness mai bun decât iterațiile anterioare, poziția sa este reținută ca fiind ,,cea mai bună’’ individuală, însă la nivel global, se păstrează soluția cea mai eficientă dintre toate particulele. Prin urmare, pe parcursul celor 50 de iterații, pozițiile particulelor se modifică printr-un mecanism simplificat, de mutație aleatoare, care înlocuiește cu o anumită probabilitate componentele poziției cu valori noi. Codul relevant al metodei este următorul:

public void runAlgorithm(DatacenterBroker *broker*, List<Vm> *vmList*, List<Cloudlet> *cloudletList*) {  
 int numCloudlets = *cloudletList*.size();  
 int numVMs = *vmList*.size();  
 List<Particle> swarm = new ArrayList<>();  
 Random random = new Random();  
 for (int i = 0; i < PARTICLE\_COUNT; i++) {  
 int[] position = new int[numCloudlets];  
 for (int j = 0; j < numCloudlets; j++) {  
 position[j] = random.nextInt(numVMs);  
 }  
 swarm.add(new Particle(position));  
 }  
 Particle globalBest = null;  
 for (int iter = 0; iter < MAX\_ITERATIONS; iter++) {  
 for (Particle particle : swarm) {  
 double fitness = evaluate(particle, *cloudletList*, *vmList*);  
 if (fitness < particle.bestFitness) {  
 particle.bestFitness = fitness;  
 particle.bestPosition = particle.position.clone();  
 }  
 if (globalBest == null || fitness < globalBest.bestFitness) {  
 globalBest = new Particle(particle.position.clone());  
 globalBest.bestFitness = fitness;  
 }  
 }  
 for (Particle particle : swarm) {  
 for (int j = 0; j < numCloudlets; j++) {  
 if (random.nextDouble() < 0.2) {  
 particle.position[j] = random.nextInt(numVMs);  
 }  
 }  
 }  
 }  
 for (int i = 0; i < numCloudlets; i++) {  
 Cloudlet cl = *cloudletList*.get(i);  
 Vm vm = *vmList*.get(globalBest.bestPosition[i]);  
 cl.setVmId(vm.getId());  
 *broker*.bindCloudletToVm(cl.getCloudletId(), vm.getId());  
 }  
 System.out.println("Particle Swarm Optimization applied!");  
}

Cu alte cuvinte, evaluarea fiecărei particule este realizată prin metoda evaluate, care calculează durata totală de execuție pe fiecare mașină virtuală și returnează valoarea maximă, reprezentând makespan-ul. Astfel, algoritmul urmărește indirect echilibrarea încărcării între mașinile virtuale, reducând diferențele dintre acestea.

private double evaluate(Particle *particle*, List<Cloudlet> *cloudletList*, List<Vm> *vmList*) {  
 Map<Integer, Double> vmTimes = new HashMap<>();  
 for (Vm vm : *vmList*) {  
 vmTimes.put(vm.getId(), 0.0);  
 }  
 for (int i = 0; i < *cloudletList*.size(); i++) {  
 Cloudlet cl = *cloudletList*.get(i);  
 Vm vm = *vmList*.get(*particle*.position[i]);  
 double execTime = cl.getCloudletLength() / (double) vm.getMips();  
 vmTimes.put(vm.getId(), vmTimes.get(vm.getId()) + execTime);  
 }  
 return Collections.max(vmTimes.values());  
}

Avantajul major al PSO este capacitatea de a explora în paralel multiple configurații de alocare și de a converge spre o soluție performantă fără a epuiza întregul spațiu de căutare. În același timp, fiind un algoritm probabilistic, rezultatele pot varia de la o rulare la alta, ceea ce oferă flexibilitate, însă necesită și o analiză comparativă riguroasă. În concluzie, acest algoritm este potrivit pentru scenarii dinamice, cu spații mari de soluții și cerințe multiple, unde metodele deterministe nu pot garanta rezultate satisfăcătoare într-un timp rezonabil.

1. Algoritmul Genetic

Dacă algoritmul PSO a explorat spațiul soluțiilor prin mișcarea coordonată a particulelor și actualizarea pozițiilor pe baza celei mai bune experiențe colective, următoarea abordare metaeuristică se inspiră din biologia evoluționistă. Algoritmul Genetic, aflat în clasa Genetic, simulează procesul de selecție naturală, în care populațiile de soluții evoluează prin încrucișare și mutație, convergând treptat către o planificare optimă. Acest model este deosebit de util în scenarii complexe, în care spațiul de căutare este destul de mare, iar combinațiile posibile între sarcini, respectiv resurse sunt numeroase.

Continuând ideea, acest algoritm este implementat în aplicație printr-o clasă care respectă interfața SchedulingAlgorithm, utilizând metoda runAlgorithm pentru a gestiona întreaga logică evolutivă. În cadrul acestei metode, fiecare soluție posibilă este reprezentată de un cromozom, adică un vector de alocări, în care fiecare poziție indica ID-ul mașinii virtuale pe care este programată o anumită sarcină. Codul relevant al acestei metode este următorul:

public void runAlgorithm(DatacenterBroker *broker*, List<Vm> *vmList*, List<Cloudlet> *cloudletList*) {  
 int numCloudlets = *cloudletList*.size();  
 int numVMs = *vmList*.size();  
 Random random = new Random();  
 List<int[]> population = new ArrayList<>();  
 for (int i = 0; i < POPULATION\_SIZE; i++) {  
 int[] chromosome = new int[numCloudlets];  
 for (int j = 0; j < numCloudlets; j++) {  
 chromosome[j] = random.nextInt(numVMs);  
 }  
 population.add(chromosome);  
 }  
 for (int generation = 0; generation < GENERATIONS; generation++) {  
 population.sort(Comparator.comparingDouble(*individual* -> fitness(*individual*, *cloudletList*, *vmList*)));  
 List<int[]> newPopulation = new ArrayList<>();  
 while (newPopulation.size() < POPULATION\_SIZE) {  
 int[] parent1 = population.get(random.nextInt(POPULATION\_SIZE / 2));  
 int[] parent2 = population.get(random.nextInt(POPULATION\_SIZE / 2));  
 int[] child;  
 if (random.nextDouble() < CROSSOVER\_RATE) {  
 child = crossover(parent1, parent2);  
 } else {  
 child = parent1.clone();  
 }  
 if (random.nextDouble() < MUTATION\_RATE) {  
 mutate(child, numVMs);  
 }  
 newPopulation.add(child);  
 }  
 population = newPopulation;  
 }  
 int[] best = population.get(0);  
 for (int i = 0; i < numCloudlets; i++) {  
 Cloudlet cl = *cloudletList*.get(i);  
 Vm vm = *vmList*.get(best[i]);  
 cl.setVmId(vm.getId());  
 *broker*.bindCloudletToVm(cl.getCloudletId(), vm.getId());  
 }  
 System.out.println("Genetic Algorithm applied!");  
}

Inițial, este generată o populație de 30 de cromozomi, fiecare reprezentând o distribuție aleatoare a cloudlet-urilor pe mașinile virtuale, iar pentru fiecare generație, cromozomii sunt sortați în funcție de fitness, pentru a evalua eficiența soluției. În acestă implementare, fitness-ul este definit ca durata totală maximă de execuție pe o mașină virtuală, având ca obiectiv minimizarea makespan-ului. Așadar, la fiecare generație sunt selectați cei mai buni indivizi care au șansa de a contribui la formarea unei noi generații, urmând ca printr-un proces de selecție aleatorie, alegerea a doi părinți pentru generarea unui copil, se face fie prin copiere directă, fie prin încrucișare, sau mai bine zis crossover, în funcție de o probabilitate prestabilită. Încrucișarea se realizează într-un punct aleatoriu, unde copilul va lua prima parte a cromozomului de la primul părinte și a doua parte de la al doilea părinte. După generare, cu o probabilitate de 10% în acest scenariu, se aplică o mutație cu scopul de a schimba în mod aleatoriu mașina virtuală atribuită unei sarcini.

Evaluarea fiecărui cromozom se face cu ajutorul metodei de fitness, care calculează timpul total de execuție al sarcinilor pe fiecare mașină virtuală și returnează valoarea maximă, astfel algoritmul cautând să distribuie uniform sarcinile pentru a reduce vârfurile de încărcare.

private double fitness(int[] *chromosome*, List<Cloudlet> *cloudletList*, List<Vm> *vmList*) {  
 Map<Integer, Double> vmLoads = new HashMap<>();  
 for (Vm vm : *vmList*) vmLoads.put(vm.getId(), 0.0);  
  
 for (int i = 0; i < *chromosome*.length; i++) {  
 Cloudlet cl = *cloudletList*.get(i);  
 Vm vm = *vmList*.get(*chromosome*[i]);  
 double exec = cl.getCloudletLength() / (double) vm.getMips();  
 vmLoads.put(vm.getId(), vmLoads.get(vm.getId()) + exec);  
 }  
 return Collections.max(vmLoads.values());  
}

În cadrul acestui algoritm, încrucișarea și mutația sunt cele două mecanisme esențiale care permit evoluția populației de soluții. Astfel, încrucișarea se realizează printr-un punct de tăiere aleatoriu, unde un copil moștenește prima parte a cromozomului de la un părinte și restul de la celălalt. Acest proces combină structuri parțial bune din soluțiile existente, favorizând apariția unor configurații mai performante, probabilitatea de încrucișare fiind de 80%, ceea ce asigură o diversitate suficientă pentru generațiile noi.

private int[] crossover(int[] *p1*, int[] *p2*) {  
 int point = new Random().nextInt(*p1*.length);  
 int[] child = new int[*p1*.length];  
 System.arraycopy(*p1*, 0, child, 0, point);  
 System.arraycopy(*p2*, point, child, point, *p1*.length - point);  
 return child;  
}

În schimb, mutația, aplicată cu o probabilitate de 10%, presupune modificarea în mod aleatoriu a unui singur element din cromozom, adică schimbarea mașinii virtuale asociată unei sarcini, având ca rol principal introducerea unei variații neprevăzute în populație, prevenind stagnarea într-o soluție locală.

private void mutate(int[] *chromosome*, int *numVMs*) {  
 int idx = new Random().nextInt(*chromosome*.length);  
 *chromosome*[idx] = new Random().nextInt(*numVMs*);  
}

Avantajul principal al acestui algoritm este capacitatea de a explora simultan mai multe regiuni ale spațiului soluțiilor și de a evolua către o configurație optimă fără a fi nevoie de cunoașterea prealabilă a structurii problemei. Aceasta este foarte eficientă în scenarii cu multe sarcini, resurse limitate și obiective multiple, precum minimizarea timpului total, echilibrarea încărcării, respectiv reducerea consumului energetic.

Având în vedere cele prezentate anterior, acest algoritm oferă o strategie robustă, adaptabilă și inspirată din procese naturale, care completează setul de algoritmi avansați integrați în aplicație și extinde capacitatea de simulare și analiză în medii cloud complexe.

1. Concluzie comparativă asupra algoritmilor de planificare

Așadar, având în vedere diversitatea algoritmilor implementați, atât în ceea ce privește complexitatea, cât și obiectivele urmărite, este utilă o sinteză comparativă care să evidențieze principalele caracteristici, avantajele, respectiv limitările fiecărei strategii. Așadar, pentru a oferi o imagine de ansamblu asupra performanței și aplicabilității fiecărui algoritm în cadrul simulărilor efectuate, am creat tabelul de mai jos pentru a evidenția aceste informații esențiale. Această comparație facilitează înțelegerea diferențelor dintre metodele clasice, euristice, respectiv metaeuristice și sprijină alegerea unor soluții adecvate pentru diverse scenarii de alocare a sarcinilor în mediile de cloud computing.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algoritm | Tip strategie | Obiectiv principal | Avantaje | Limitări |
| FCFS | Clasic | Simplitate, oridine naturală | Ușor de implementat | Ignoră lungimea sarcinilor |
| RoundRobin | Clasic | Distribuție uniformă | Echilibru simplu între mașinile virtuale | Nu ține cont de durată |
| RandomScheduler | Aleator | Variabilitate, testare nedeterministă | Oferă comportament imprevizibil | Neoptimizat |
| LJF | Euristic | Prioritizarea sarcinilor lungi | Ideal pentru sarcini costisitoare | Întârzie sarcinile scurte |
| Min-Min | Estimativ | Reducerea makespan-ului | Eficient pentru sarcini scurte | Întârzie sarcinile lungi |
| Max-Min | Estimativ | Procesarea timpurie a sarcinilor mari | Previne blocajul final | Întârzie sarcinile scurte |
| MinLengthRoundRobin | Hibridă | Execuție rapidă și echilibrare simplă | Combinație eficientă și ușor de controlat | Nu se adaptează dinamic |
| ACO | Metaeuristic | Învățare iterativă | Se adaptează pe baza experienței acumulative | Necesită resurse de procesare senificative |
| PSO | Metaeuristic | Minimizarea makespan-ului prin roi | Evoluează pe baza celor mai bune soluții | Rezultate variabile |
| Genetic | Metaeuristică evolutivă | Evoluție bazată pe selecție și mutație | Combinație robustă și explorare | Sensibil la configurarea parametrilor algoritmici |

Tabel 4 Tabel comparativ al algoritmilor implementați

1. Componenta de gestionare a bazei de date

Pentru a asigura persistența și reutilizarea rezultatelor simulărilor, aplicația integrează o componentă complet funcțională de gestionare a bazei de date, construită în jurul unui sistem relațional MySQL. Aceasta permite salvarea, încărcarea, afișarea și vizualizarea rezultatelor oricărei simulări efectuate, fiind direct conectată la interfața grafică a aplicației prin butoanele Save, Load, Show More și Show Chart.

Baza de date este structurată în două tabele principale și anume simulation\_summary, care reține metadatele generale ale unei simulări, respectiv simulation\_results, care conține rezultatele detaliate pentru fiecare sarcină. Cele două tabele sunt legate între ele prin coloana simulation\_id, un UUID unic generat la fiecare simulare, formând astfel o relație de tip one to many. Această structră asigură flexibilitatea interogării și permite analizarea fiecărei execuții în detaliu, fară duplicarea inutilă a datelor. Pentru a înțelege modul în care sunt structurate și interconectate tabelele din baza de date, este utilă reprezentarea lor grafică sub formă de diagramă Entitate-Relație:

A screenshot of a graph

AI-generated content may be incorrect.

Figura 5.2 Diagrama Entitate Relație

Astfel, tabelul simulation\_summary conține câmpuri precum algoritmul utilizat, tipul de simulare dacă e de tipul eficient energetic sau nu, numărul de host-uri, mașini virtuale și cloudlet-uri, energia totală consumată, timpul real de execuție și timestamp-ul rulării. Pe baza acestuia se construiește lista principală de simulări disponibile în aplicație.

Nu putem omite nici tabelul simulation\_results care stochează rezultatele fiecărui cloudlet individual și anume identificatorul său, starea finală a simulării, mașina virtuală și host-ul pe care a fost executat, timpii de start, final și execuție, precum și energia consumată. Aceste date sunt încărcate la apăsarea butonului ShowMore și pot fi vizualizate prin grafic prin Show Chart, care generează un BarChart cu energia per cloudlet. Pentru salvarea datelor, aplicația folosește două metode separate reprezentate de saveSummary care inserează date în simulation\_summary, și saveCloudlets care inserează date în simulation\_results, păstrând legătura dintre ele prin acelați simulation\_id. La încărcare, metoda showSummaryTableWindow interoghează simulation\_summary, iar prin selecția unui rând și utilizarea simulation\_id, sunt extrase datele din simulation\_results pentru simularea respectivă. Această abordare modulară oferă un cadru solid pentru analiză și raportare, permite stocarea ordonată a datelor, și creează premisele pentru extinderea viitoare pentru raportări comparative între execuții multiple.

1. Componenta de vizualizare grafică a rezultatelor simulării

Pe lângă salvarea și afișarea tabelară a rezultatelor, aplicația oferă și o funcționalitate de vizualizare grafică, care permite interpretarea rapidă și intuitivă a consumului energetic asociat fiecărei sarcini executate. Prin butonul Show Chart, disponibil în dreptul fiecărei simulări salvate, utilizatorul poate genera un BarChart, în care axa X reprezintă identificatorii cloudlet-urilor, iar axa Y valoarea energiei consumate de fiecare.

Graficul este construit pe baza datelor extrase din tabela simulation\_results, folosind metoda createEnergyChartFromDb(simulationId), care transformă rezultatele într-o serie de tip XYChart.Series<String, Number>. Ulterior, acesta este afișat într-o fereastră separată, folosind metoda showEnergyChartWindow, care include și o secțiune de statistici sintetice, generate dinamic, precum energia totală, timpul total de execuție al cloudlet-urilor, respectiv algoritmul utilizat. De asemena, graficul este salvat automat în format png cu un nume generat din parametrii simulării, fiind astfel reutilizabil în documentație. Așadar, acest grafic este mai mult decât o reprezentare estetică, deoarece permite detectarea imediată a dezechilibrelor de alocare între mașinile virtuale, identificarea sarcinilor costisioare energetic și compararea comportamentului algoritmilor în scenarii diferite.

În concluzie, aplicația dezvoltată integrează toate componentele necesare pentru simularea completă a procesului de programare a sarcinilor în cloud computing, punând accentul pe eficiența energetică și totodată oferind o platformă extensibilă, eficientă și intuitivă, utilă pentru analiza comparativă și cercetarea aplicată.