3. Studiu Bibliografic

1. Introducere in Cloud Computing

În cadrul celei de-a patra revoluții industriale, cunoscută drept ,,4IR’’, tehnologiile informatice au evoluat semnificativ înspre descentralizare și minimizare în ceea ce privește latența [1], ceea ce a condus la apariția a trei paradigme fundamentale de calcul, și anume Cloud Computing, Fog Computing și Edge Computing. Aceste trei modalități de procesare a datelor se deosebesc prin poziționarea lor în arhitectura rețelei, proximitatea față de sursa datelor, respectiv prin capacitatea de a satisface cerințele specifice legate de latență, scalabilitate ori distribuție geografică [2].

Diagram of a cloud network

AI-generated content may be incorrect.



Figura 1. Domeniile de aplicare ale cloud computing-ului, fog computing-ului și edge computing-ului (conform Mahmud et al. [3]).

Piața a experimentat o expansiune rapidă ca urmare a integrării cloud computing-ului ca normă globală pentru procesarea și stocarea datelor. Această evoluție a fost facilitata de accesibilitatea crescândă a dispozitivelor ,,smart’’, fie că vorbim despre telefoane mobile, televizoare inteligente sau ceasuri inteligente [6]. Ca rezultat, s-a înregistrat o cerere semnificativă pentru resursele de rețea și servere dedicate procesării datelor, ceea ce a condus la aparitia unor paradigme inovatoare, precum Edge Computing și Fog Computing. Aceste concepte avansate propun o apropiere a procesării și capacității de stocare de utilizator, fenomen ce contribuie la reducerea latenței și la creșterea disponibilității datelor, formându-se astfel domeniul de cercetare cunoscut sub numele de Edge Computing prin intermediul unei game largi de caracteristici și tehnologii ce contribuie la procesarea și stocarea datelor de la marginea rețelei.

Astfel, paradigma Fog Computing constituie un model stratificat, având ca obiectiv primordial conservarea avantajelor oferite de cloud pentru desfășurarea aplicațiilor sau serviciilor capabile să proceseze și să stocheze date pentru utilizatorii finali în noduri distribuite. Această structură, corelată cu nodurile edge, facilitează diminuarea timpului de răspuns necesar pentru sarcinile sau procesele generate în timpul utilizării.

Dintre cele trei paradigme, cloud computing-ul s-a distins ca fiind modelul cel mai răspândit și cu un grad de maturitate superlativ, fiind utilizat atât în mediul academic, cât și în sectorul privat, în mod implicit instrumentele de simulare[4] dobândind o importanță tot mai mare de-a lungul ultimelor decenii în tot ceea ce înseamnă cercetările dedicate acestei tehnologii aflate într-o continuă dezvoltare. Continuând această idee, prin intermediul proceselor de simulare devine posibilă o evaluare prealabilă a configurațiilor de servicii, precum și a strategiilor de alocare și gestionare a resurselor, ceea ce facilitează îmbunătățirea performanței sistemelor, testarea riguroasă a ipotezelor tehnice și comerciale, respectiv obținerea de rezultate fiabile, ușor de reprodus, cu un timp de execuție bine optimizat, și cu un risc considerabil diminuat printr-un necesar de investiții minime.

În contextul cercetărilor actuale, simularea se prezintă ca fiind metoda predominantă la nivel global, având capacitatea de a configura medii virtuale complexe ce integrează aplicații, sarcini, dispozitive și condiții de rețea specifice. Aceste simulări produc date semnificative referitoare la performanța unei infrastructuri anume, incluzând metrici de tipul utlizării procesorului central (CPU), lățimea de bandă disponibilă și latența, facilităndu-se astfel procedura de luare a unor decizii fundamentate cu privire la implementarea în mediile reale.

1. Platforme de simulare

Nu poate fi omis faptul că există o plajă variată de simulatoare dedicate mediilor de tip Cloud, Fog sau Edge[4], fiecare având propriile trăsături distinctive și funcționalități specifice. Aceste instrumente permit configurarea de diverși parametri sistemici, inclusiv arhitectura rețelei, tipurile de dispozitive implicate, cerințele aplicațiilor ori caracteristicile proceselor în sine, oferindu-se astfel o platformă de testare care se dovedește a fi atât flexibilă, cât și accesibilă. Totuși, trebuie menționat faptul că niciun simulator nu reușește să satisfacă în mod amănunțit toate cerințele posibile ale unui scenariu complex, întrucât această diversitate de instrumente generează un obstacol suplimentar, și anume selecția simulatorului adecvat în funcție de obiectivele specifice pe care cercetarea le urmărește.

În literatura de specialitate, au fost formulate numeroase clasificări și analize comparative ale simulatoarelor, dintre care se remarcă, în primul rând, CloudSim[8], întrucât acesta este cel mai utilizat simulator specific infrastructurilor cloud, concetrându-se pe alocarea eficientă de resurse și pe programarea sarcinilor(scheduling), fiind facilitată analiza performanței în mediile virtualizate. În plus, avem și extensiile special concepute pentru abordarea particularităților mediilor Fog și Edge, și anume iFogSim și YAFS[5], acestea oferind suport pentru topologii dinamice și implicit pentru distribuții geografice, elemente esențiale unei gestionări eficiente a resurselor unor arhitecturi emergente. Alte categorii de simulatoare ce trebuie menționate sunt LEAF și EdgeCloudSim[9], care sunt specializate pe simularea de aplicații de tip Internet of Things sau ,,IoT’’, respectiv evaluării performanței în condiții de latență scăzută și având un rol important în optimizarea interacțiunilor unor dispozitive conectate. Nu în ultimul rând, SimGrid[10] reprezintă un instrument robust pentru simulările distribuite, fiind extins recent prin intermediul inițiativelor precum ENIGMA, având ca scop principal integrarea mobilității și vizualizării geografice, fiind urmărită îmbunătățirea capacității de modelare a sistemelor complexe. Așadar, perspectiva efectuării unor comparații între aceste simulatoare devine un factor de o importanță deosebită atât din punct de vedere tehnic, cât și din punct de vedere practic datorită perspectivei scenariilor reale ce urmează a fi replicate.

În continuarea aspectelor prezentate anterior, tabelul atașat mai jos oferă o sinteză comparativă a simulatoarelor utilizate cel mai frecvent în literatura de specialitate, subliniindu-se paradigma abordată, limbajul de programare aplicat, suportul pentru mobilitate și gestionarea energiei, precum și câteva observații esențiale.

|  |  |  |  |
| --- | --- | --- | --- |
| Simulator | Paradigma | Limbaj | Observatii principale |
| CloudSim[] | Cloud | Java | Simulator clasic, neavând mobilitate dinamică |
| iFogSim[] | Cloud + Fog | Java | Cu topologie fixă, utilizat pentru IoT |
| YAFS | Fog + Edge | Python | Mobilitate reală, având grafuri dinamice |
| LEAF | Edge | Java | Folosit pentru evenimente discrete, IoT distribuit |
| EdgeCloudSim | Cloud + Edge | Java | Simplu, folosit pentru aplicații de tip edge |
| SimGrid | Distribuit | C/C++ | Foarte greu de personalizat, folosit pentru arhitecturile distribuite |
| ENIGMA | Fog + Edge | C/C++ | Are mobilitate IoT, implicit vizualizare geospatială bazată pe SimGrid |

Tabel 1. Comparație între cele mai utilizate simulatoare [5].

În urma unei analize cuprinzătoare a simulatorilor disponibili, am optat pentru utilizarea CloudSim ca fundament pentru simulările prezentate în această lucrare, având în vedere popularitatea, maturitatea și flexibilitatea remarcabile pe care acesta le are. În literatura de specialitate, CloudSim se evidențiază printr-o serie de versiuni care au fost dezvoltate de-a lungul timpului, fiecare dintre acestea aducând îmbunătățiri semnificative și extensii funcționale, și anume: CloudSim, versiunea de bază, CloudSim Plus, care este orientat pe obiect și modular, și nu în ultimul rând, varianta cea mai recentă, CloudSim 7G, extinsă cu funcționalități moderne. //data release in table + paper.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Versiune | Anul apariției | Limbaj | Extensibilitate | Funcționalitate |
| CloudSim | 2009 | Java | Bună | Simplu, oferă modelare pentru VMs implicit datacenters |
| CloudSimPlus | 2017 | Java | Ridicată | Imbunătățire la modularitate, ușor de utilizat, cod mai clar si mai bine structurat |
| CloudSim7G | 2023 | Java | Foarte bună | Oferă flexibilitate aplicațiilor distribuite, suportă simularea și managementul rețelelor în cloud, include funcționalități de management al energiei pentru simularea consumului de energie, integrarea IoT, etc. |

Tabel 2.Comparație între simulatoarele CloudSim [7], [11]

Dintre multiplele variante existente ale simulatorului CloudSim, CloudSim 7G a fost selectat pentru această cercetare, întrucât este cea mai sofisticată și cuprinzătoare iterație din această serie de instrumente. Spre deosebire de versiunile anterioare, caracterizate prin module disparate, adesea dificil de integrat și de întreținut, CloudSim 7G introduce o arhitectură reconfigurată și standardizată, capabilă să sprijine simulări complexe și multi-modulare într-o manieră coezivă și eficientă.

Unul dintre cele mai notabile beneficii ale acestei versiuni o reprezintă suportul său robust pentru modelarea consumului energetic, realizat printr-o refactorizare substanțială a modulului dedicat energiei, precum și a altor componente relevante, inclusiv cele asociate containerelor și rețelelor. În contextul în care eficiența energetică a centrelor de date constituie o preocupare centrală atât pentru cercetare, cât și pentru sectorul industrial [12], CloudSim 7G facilitează o evaluare realistă a strategiilor de conservare a energiei și a politicilor de scalare automată a consumului energetic. Acesta oferă oportunitatea simulării suprapunerilor de containere pe mașini virtuale, gestionarea supraîncărcării, migrarea sarcinilor de lucru și analiza penalizărilor rezultate din procesul de virtualizare.

În plus, CloudSim 7G integrează funcționalități derivate din modulele istorice, precum NetworkCloudSim, ContainerCloudSim și CloudSimSDN, reunite sub o arhitectură unificată, care se distinge prin compatibilitate și extensibilitate. Această sinteză complexă elimină dificultățile de compatibilitate care au fost întâmpinate anterior, facilitându-se conceperea unor scenarii hibride. În cadrul acestor scenarii, pot coexista mașini virtuale (VM-uri), containere, funcții serverless și servicii distribuite, toate acestea operând într-un mediu de simulare singular.

În ceea ce privește performanța, versiunea 7G realizează optimizări semnificative ale codului sursă, diminuându-l cu peste 13.000 de linii, concomitent cu o îmbunătățire considerabilă a timpului de execuție și a eficienței utilizării memoriei, reușind să aloce cu până la 25% mai puțină memorie heap, fără compromiterea funcționalității. Această versiune introduce, de asemenea, concepte avansate, inclusiv suport pentru virtualizare imbricată și modelarea penalizărilor de latență asociate interfețelor virtuale, aspecte esențiale pentru simularea realistă a infrastructurilor cloud de nouă generație.

Prin urmare, CloudSim 7G asigură un echilibru optim între realismul simulării, extensibilitate și eficiență computațională, reprezentând alegerea ideală pentru obiectivele prezentei lucrări, care se focalizează pe analiza performanței și consumului energetic în scenarii avansate de tip cloud.

1. Algoritmi de planificare a sarcinilor

În contextul sistemelor de cloud computing, gestionarea programării sarcinilor constituie o componentă esențială, având o influență semnificativă asupra calității serviciilor oferite utilizatorilor, eficienței economice și performanței la nivel global. În acest context, Al-Arasi și Saif[13] analizează o problemă esențială în domeniul programării sarcinilor în cloud computing, focalizându-se în mod special asupra eficienței tehnicilor meta-euristice. Cloud computing-ul, definit prin resurse abundente și dinamice, necesită implementarea unor metode optime de programare pentru alocarea sarcinilor către mașinile virtuale (VM) într-un mod care să îndeplinească o varietate de obiective de optimizare. Aceste obiective includ, printre altele, reducerea timpului total de execuție (makespan), minimizarea costurilor operaționale, asigurarea unei distribuții echilibrate a sarcinilor și îmbunătățirea eficienței energetice.

Există o structurare complexă a sarcinilor în cadrul celor trei straturi fundamentale ale arhitecturii cloud[13]: stratul software, concentrat pe maximizarea experienței utilizatorului prin minimizarea timpului de procesare și a costurilor aferente, stratul platformă, dedicat virtualizării resurselor și optimizării gestionării sarcinilor, respectiv stratul infrastructură, care se focalizează pe implementarea coordonată a resurselor și facilitarea colaborării între diversele entități cloud. Procesul de planificare a sarcinilor este, de asemenea, segmentat în trei etape distincte[13]: identificarea resurselor disponibile, selecția adecvată a acestora și alocarea sarcinilor, de regulă, mediată de un broker de centre de date (DataCenter Broker), care este susținut de servicii de informații cloud.

Este bine documentat în literatura de specialitate faptul că planificarea sarcinilor în domeniul cloud computing este clasificată ca NP-hard (Not Polynomial hard), ceea ce indică faptul că, în contextul instanțelor de mari dimensiuni, obținerea unor soluții exacte devine o întreprindere impracticabilă. Urmărindu-se aproximarea soluțiilor optime într-un interval de timp acceptabil, cercetătorii din sfera academică au propus o varietate de abordări euristice și metaeuristice, incluzând metodele tradiționale de programare, cum ar fi „First Come First Serve” (FCFS), „Round Robin” (RR) și „Shortest Job First” (SJF), care s-au dovedit însă ineficiente în furnizarea unor performanțe optime în cadrul complexității NP-hard asociate sistemelor cloud. Deși abordările euristice, precum Min-Min și Max-Min, oferă anumite îmbunătățiri, acestea sunt adesea constrânse de capcana optimilor locali. În contrast, algoritmii meta-euristici, inspirați de fenomenele naturale și structurile biologice, propun soluții care manifestă o scalabilitate mai mare și se apropie de optimalitate. Astfel, aceste tehnici sunt catalogate în două categorii fundamentale, și anume cele bio-inspirate (de exemplu, algoritmii genetici și algoritmii memetici) și cele bazate pe inteligența colectivă a roiului (de exemplu, „PSO” - Particle Swarm Optimization și „ACO” - Ant Colony Optimization).

O contribuție deosebită a acestei analize[13] rezidă în efectuarea unei evaluări comparative riguroase a unui număr semnificativ de algoritmi meta-euristici, bazată pe o varietate diversificată de factori, printre care se numără mediul de simulare dominant (CloudSim), metricile de optimizare corespunzătoare (inclusiv durata de realizare, costurile, termenele limită, consumul de energie etc.), amploarea experimentală (volumul sarcinilor și resurselor implicate) și natura interdependenței sarcinilor, fie că sunt dependente sau independente. Printr-o clasificare sistematică și detaliată, studiul subliniază că durata de realizare se prezintă ca fiind criteriul cel mai frecvent vizat în procesele de optimizare, fiind menționată în 33% din cercetările analizate, urmată de costuri (18%), echilibrarea sarcinilor (16%) și alte dimensiuni relevante, precum eficiența energetică și debitul. Agoritmii de Optimizare a Roiului de Particule (PSO) și Algoritmul Genetic (GA) se evidențiază ca fiind cele mai frecvent aplicate metode metaeuristice în domeniile inteligenței în roi, respectiv al algoritmilor bio-inspirați.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algoritm | Tipul algoritmului | Optimizare principala | Avantaje | Dezavantaje |
| FCFS[] | Tradițional | Primul venit primul servit | Ușor de implementat | Timpul de execuție și costul ridicat |
| RoundRobin | Tradițional | Distribuție egală a resurselor | Potrivit pentru medii interactive | Creste overhead-ul, nu e bun pentru sarcini cu timp de execuție mare |
| Random Scheduler | Tradițional | Aleatoriu | Ușor de implementat | Lipsa predictibilității, controlului și foarte ineficient |
| LJF | Tradițional | Se prioritizează sarcinile lungi | Se imbunătățeste utilizarea resurselor | Sarcinile scurte sunt blocate de cele lungi |
| Min-Min | Euristic | Se reduce makespan-ul | Timp scurt la finalizarea sarcinilor | Sarcini distribuite inegal, având in vedere resursele |
| Max-Min | Euristic | Incărcare echilibrată | Utilizează foarte bine resursele | Sarcinile scurte sunt blocate de cele lungi |
| ACO | Meta-euristic | Se balansează incărcarea, are makespan | Adaptiv | Timp de convergență mare |
| Genetic Algorithm(GA) | Meta-euristic | Are makespan, cost | Foarte flexibil, eficient în privința optimizărilor | Sensibil la parametri |
| PSO | Meta-euristic | Se balansează incărcarea, are makespan | Optimizări globale bune | Poate rămâne blocat intr-o soluție locală, care pare a fi optimă, însă se nu este. |

Tabel3.Comparație între Algoritmii de Planificare a sarcinilor

1. Alocarea resurselor pentru eficiența energetică

Utilizarea tehnologiilor de cloud computing a cunoscut o evoluție semnificativă, depășind limitele conceptelor tradiționale de tipul calculului paralel, grid computing-ului sau a calculului distribuit. În prezent, această paradigmă tehnologică integrează virtualizarea serverelor cu o diversitate de resurse și infrastructuri emergente. În calitate de sistem contemporan, cloud computing-ul se confruntă cu o gamă extinsă de provocări, în special în contextul optimizării alocării resurselor. Eficiența în alocarea acestor resurse este de o importanță deosebită, având în vedere impactul său asupra modului în care infrastructura virtualizată este distribuită pentru satisfacerea cerințelor utilizatorilor, conform acordurilor de nivel de serviciu (SLA), urmărind înn paralel minimizarea costurilor, precum și reducerea consumului de energie.

Alocarea resurselor constituie un proces complex ce implică distribuirea capacității de procesare, a memoriei și a lățimii de bandă între diverse aplicații și utilizatori în cadrul mediilor cloud. Obiectivul fundamental al acestui demers este de a asigura o alocare atât echitabilă, cât și eficientă a acestor resurse, garantând, în consecință, desfășurarea activităților într-un interval temporal adecvat, concomitent cu optimizarea consumului energetic și a cheltuielilor operaționale. Centrele de date se confruntă cu provocarea de a menține un echilibru delicat între disponibilizarea resurselor necesare pentru operațiuni cu prioritate înaltă și riscul de a consuma aceste resurse în mod ineficient, prin alocarea lor către procese de o prioritate inferioară. Conform analizei efectuate, se estimează că peste 20% din energia utilizată de centrele de date provine dintr-o utilizare ineficientă a resurselor, fiind astfel evidențiată necesitatea stringentă de a spori eficiența energetică în acest context[14].

Printre problemele fundamentale identificate se regăsesc dificultățile inerente prognozării cerințelor aplicațiilor utilizatorilor, asigurarea capacității echipamentelor fizice de a susține, în mod simultan, multiple instanțe de mașini virtuale, gestionarea eficientă a programării sarcinilor asociate proceselor de lungă durată, precum și atingerea unui nivel optim de eficiență energetică, în ciuda creșterii volumului de muncă. Eseul subliniază importanța primordială a armonizării perspectivelor și intereselor atât ale furnizorilor de servicii, cât și ale consumatorilor acestora, în scopul realizării unor servicii cloud cu un grad înalt de productivitate.

În vederea sistematizării concluziilor, cercetarea [14] îmbrățișează o taxonomie care facilitează clasificarea modalităților de alocare a resurselor în cinci categorii bine definite, și anume strategice, bazate pe resurse țintă, orientate spre optimizare, fundamentate pe programare, precum și tehnici axate pe consumul de energie.

În consecință, strategiile fundamentate pe consumul de energie își concentrează eforturile asupra reducerii utilizării energetice în centrele de date. Aceste strategii pot fi ulterior clasificate în două categorii distincte, respetiv tehnici axate pe eficiența energetică și tehnici orientate spre managementul temperaturii. Tehnicile eficiente din perspectiva consumului energetic urmăresc optimizarea plasării mașinilor virtuale și coordonarea distribuției sarcinilor de lucru, având ca obiectiv principal diminuarea consumului energetic. În acest context, se recurge la metaeuristici, cum ar fi algoritmii de optimizare inspirați de comportamentul coloniilor de furnici sau de coloniile artificiale de albine, cu scopul de a spori eficiența energetică, deși această abordare poate conduce la o prelungire a timpului necesar execuției. Pe de altă parte, tehnicile bazate pe temperatură se concentrează pe evaluarea influenței distribuției volumului de lucru asupra temperaturii componentelor hardware, având ca ambiție prevenirea supraîncălzirii prin echilibrarea încărcărilor termice. Cu alte cuvinte, aceste strategii nu contribuie doar la creșterea fiabilității sistemului, ci și la diminuarea necesităților de răcire, însă trebuie menționat faptul că aceste tehnici neglijează adesea considerente esențiale,cum ar fi costurile asociate sau alinierea optimă a volumului de lucru.

Totdată, nu putem subestima nici importanța componentelor esențiale ale analizei tehnicilor de alocare strategică a resurselor, care joacă un rol fundamental în facilitarea adaptabilității atât pentru furnizorii de servicii cloud, cât și pentru consumatorii acestora, în contextul caracterului dinamic și evolutiv al mediilor de cloud computing. Aceste strategii sunt elaborate cu scopul de a facilita luarea de decizii informate și anticipative în ceea ce privește distribuția resurselor, ele contribuind la armonizarea cerințelor consumatorilor cu resursele disponibile, într-un mod care nu doar optimizează performanța, ci și contribuie la reducerea costurilor, sporind astfel satisfacția generală a utilizatorilor.

Tehnicile de alocare strategică a resurselor sunt, în general, divizate în două categorii fundamentale, și anume alocarea dinamică a resurselor și abordările bazate pe inteligența artificială. Aceste metodologii permit furnizorilor de servicii cloud efectuarea de analize riguroase și comparative între diversele resurse disponibile, prezicerea cu acuratețe a cerințelor de resurse ale consumatorilor, în funcție de specificitatea sarcinii respective, respectiv determinarea infrastructurii virtuale care este cea mai adecvată pentru ducerea la bun sfârșit a respectivei sarcini în cel mai eficient mod posibil.

În ultimii ani, centrele de date au evoluat în calitate de consumatori semnificativi de energie electrică[14], ceea ce a determinat comunitatea academică să dezvolte strategii mai eficiente pentru gestionarea distribuției resurselor, necesitate fundamentată prin faptul că în baza unei administrări inteligente a resurselor, se poate obține nu doar diminuarea consumul energetic, dar și reducerea costurile operaționale globale asociate infrastructurii cloud. În acest context, lucrarea [14] propune o clasificare a strategiilor de alocare a resurselor, având ca obiectiv eficiența energetică, în două categorii esențiale, respectiv tehnicile conștiente de energie, care subliniază importanța minimizării utilizării energiei pentru a promova atât eficiența economică, cât și sustenabilitatea ecologică, și tehnicile conștiente de temperatură, care examinează efectele termice asociate implementării resurselor și caută să atenueze acumularea de căldură în interiorul unităților fizice.

Alocarea resurselor pe baza consumului energetic este orientată înspre atingerea unor standarde de excelență operațională, în conformitate cu constrângerile inerente consumului și distribuției energiei. Aceste strategii au devenit tot mai cruciale în asigurarea eficienței operațiunilor centrelor de date, atât din perspectiva performanței, cât și a sustenabilității ecologice. O varietate de cercetători au avansat metode inovatoare pentru a diminua consumul energetic, prin optimizarea plasării mașinilor virtuale (VM). Putem lua drept exemplu dezvoltarea unei tehnici bazate pe principiile inteligenței colective, care facilitează migrarea dinamică a VM-urilor, urmărind echilibrarea sarcinii sistemului concomitent cu reducerea consumului energetic. Această abordare a demonstrat o eficiență notabilă în minimizarea timpului de răspuns, însă nu a abordat în mod explicit problema timpului de execuție. Similar, a fost implementată optimizarea coloniilor de furnici în vederea abordării provocării alocării VM-urilor, având ca obiectiv reducerea risipei energetice și implicit a consumului total de energie. Deși această metodă a reușit să îndeplinească obiectivele urmărite, nu a fost luată în considerare variabila timpului de răspuns.

Aceste tehnici orientate spre energie constituie un avans remarcabil în direcția unui cloud computing sustenabil, însă este impetuos necesar de a fi meționat faptul că acesta tinde de cele mai multe ori să privilegieze eficiența energetică în detrimentul altor parametri relevanți, cum ar fi latența sau timpul de execuție.

4. Analiză și fundamentare Teoretică

1. Arhitectura abstracta a aplicatiei

Fundamentarea structurii logice a aplicației descrise în această lucrare se regăsește pe arhitectura de tip CloudSim 7G, un cadru evoluat ce se dedică simulărilor în mediile de tip cloud, conceput pentru experimentarea de scenarii complexe, de natura alocării de resurse, de planificare a sarcinilor sau de optimizare a consumului de energie. Astfel, simulatorul CloudSim 7G dezvoltă versiunea de bază CloudSim, prin intermediul unei abordări modulare, cu o flexibilitate aparte și o eficiență ridicată, bazată pe performanță, un suport bine dezvoltat pentru modelele energetice detaliate și tototdată o scalabilitate ridicată. Cu o arhitectură stratificată, conține și oferă oportunitatea folosirii unei platforme teoretice robuste, care ulterior se constituie într-o bază solidă pentru posibilitatea efectuării analizei comparative a algoritmilor de planificare.

Conceput cu scopul de a generaliza, respectiv unifica capabilitățile fragmentate ale extensiilor CloudSim anterioare, CloudSim 7G reușește să integrese sub forma unei singure arhitecturi funcționalități avansate pentru cloud, edge sau fog computing. Oferind o structură de tip modular, stratificată logic, CloudSim 7G este potrivit pentru efectuarea de analize comparative și experimente controlate, întrucât permite modelarea detaliată a resurselor, a comportamentului energetic sau a proceselor de planificare.

A screenshot of a computer screen

AI-generated content may be incorrect.

Figura 4.2 Arhitectura CloudSim[15]

1. Fundamentele arhitecturii CloudSim 7G

Simulatorul CloudSim 7G funcționează în baza orientării pe evenimente discrete, în contextul în care entitățile principale, de tipul cloudlet-urilor, mașinilor virtuale, host-urilor sau brokerilor interacționează prin intermediul unor evenimente planificate și gestionate cu ajutorul unui motor de simulare centralizat, model ce permite simularea unui mediu real fără a necesita costuri de execuție în mediile fizice. Principiile esențiale pe care se bazează arhtiectura CloudSim 7G sunt reprezentate, în primul rând, de modularitate și reconfigurabilitate, întrucât componentele principale sunt independente logic, putând fi înlocuite ori extinse fără ca restul sistemului să fie afectat. În plus, trebuie să vorbim și despre abstracizare completă a infrastructurii cloud, pornind de la nivelul fizic, ce implică datare și host-uri, la nivelul virtual, prin mașini vitruale și mergând până la cel al aplicației, folosindu-se de cloudet-uri, respectiv despre separarea clară între alocarea de mașini virtuale și planificarea sarcinilor, întrucât fiecare este controlată prin intermediul politicilor independente, testabile în mod izolat. Nu în ultimul rând, suportul extins pe care îl prezintă când vine vorba despre consumul energetic și politicile energetice, cu posibilitatea încorporării modelelor personalizate specific nivelului de utilizare al resurselor.

1. Structura teoretică a simulatorului CloudSim7G

Simulatorul de tip CloudSim 7G are abilitatea de a reproduce arhitectura stratificată prin intermediul mai multor componente abstracte. Astfel, pentru a începe cu începutul, trebuie să vorbim despre stratul de modelare a infrastructurii, întrucât în cadrul acestuia sunt definite entități fundamentale, precum host-urile fizice, reprezentând unități de procesare configurabile, te tipul CPU, RAM, BW, adică lațimea de bandă, ce pot fi ocupate concomitent de mai multe mașini virtuale. În acest context, mașinile virtuale reprezintă entități de nivel intermediar ce simulează instanțte virtualizate, iar cloudlet-urile sunt definite drept sarcini de calcul ce rulează pe mașini virtuale în mod independent. Nu putem omite nici datacenter-ul ca element de bază, cu o structură logică cu rol de a grupa host-urile într-un spațiu de simulare unitar. Nu în ultimul rând, trebuie menționat faptul că fiecare entitate este descrisă teoretic printr-un set de parametri, de tip MIPS, adica un milion de instrucțiuni pe secundă, memorie sau număr de nuclee, cu relații modelate logic, fără dependență în ceea ce privește implementarea.

Prin intermediul acestui simulator se oferă o separare clară între două tipuri de politici, element esențial în cadrul acestei lucrări, și anume politicile de alocare a VM-urilor pe host-uri, sau în termeni tehnici, VMAllocationPolicy, ce decid plasarea fiecărei mașini virtuale, strat cunoscut sub denumirea de strat de planificare și alocare. Ulterior, avem Cloudletscheduler-ul, sau politicile de planificare a cloudlet-urilor pe VM-uri, ce determină în mod esențial ordinea, respectiv modul de execuție al sarcinilor. Nu în ultimul rând, cu ajutorul acestei diviziuni putem să testăm individual performanța fiecărei politici, fără a avea parte de interferențe logice, fiind asigurate condiții controlate pentru facilitarea evaluării de algoritmi analizați în lucrare, situații întâlnite în cazurile FCFS, RR sau Min-Min.

Prin intermediul stratului execuției simulate, executarea este dusă la bun sfârșit printr-un eveniment discret, temporizat și gestionat de un Scheduler central, fie că vorbim despre alocare, începutul sarcinii sau finalizarea acesteia, aici fiind calculat consumul energetic estimativ, aplicat pe fiecare host, în funcție utilizarea sa curentă, utilizarea resurselor de tip CPU, RAM sau BW penru fiecare entitate, respectiv timpul calculat al fiecărei sarcini.

Nu în ultimul rând, stratul de analiză a performanței și consumului este esențial în realizarea comparației dintre consumul de energie raportat la performanță prin folosirea datelor generate ori procurate ca urmare a unui scenariu de simulare, utilizând diferite politici de scheduling, implicit algoritmi cu preocuparea consumului de energie.

1. Avantajele arhitecturii CloudSim 7G

Multiple avantaje teoretice fundamentează alegerea CloudSim 7G pentru simularea realizată în cadrul acestei lucrări, iar cea mai eficientă modalitate de a le expune este enumerarea lor. Astfel, prin utilizarea acestui simulator, se reduce redundanța și refactorizarea codului cu 67% și pastrându-se totodată acoperirea funcțională completă. În plus, spre deosebire de alte simulatoare, are capacitatea de a simula medii cloud, edge și fog într-un cadru unificat, și nu trebuie omis nici suportul pentru politici dinamice, inclusiv în ceea ce privește consolidarea energetică și scalarea de tip dinamic, iar nu în utlimul rând, interoperabilitatea cu modele personalizate și module prestabilite. Așadar, caracteristicile specifice oferă un cadru teoretic ideal studiilor comparative în ceea ce privește algoritmii de alocare și scheduling, fiind oferită o platformă robustă și foarte extensibilă în ceea ce privește cercetarea academică.

1. Modele abstracte utilizate

Simulatorul CloudSim 7G, prin intermediul abstractizării entităților principale și a standardizării acestora, implementează o abordare îmbunătățită asupra modelării infrastructurilor de tip cloud, fog sau edge. Astfel, acest simulator utilizează interfețe generalizate în detrimentul claselor rigide, specifice versiunilor anterioare, permițând astfel coabitarea unor entități de tip virtual și fizic în contextul unei singure simulări. Prin intermediul acestui lucru, se subliniază modelele abstracte moderne, folosite pentru reprezentarea entităților și a comportamentelor dinr-un scenariu simulativ.

1. Modelul HostEntity

În simulatorul CloudSim 7G, comportamentele asociate unui host, și anume alocare, monitorizare, executarea entităților virtuale, precum și evaluarea consumului energetic sunt definite prin intermediul HostEntity, care totodată modelează resursele fizice. Astfel, se reduce nevoia de utilizare a unor clase de tipul ContainerHost sau PowerHost, întrucât clasa menționată anterior se poate extinde înspre funcționalități specifice, precum energie, rețea sau virtualizare imbricată, cu ajutorul unor interfețe suplimentere, precum PowerHostEntity. Un host poate fi definit, teoretic, prin capactățile hardware, fie ele PEs, RAM sau BW, prin politicile de alocare a resurselor, consumului energetic prin prisma funcției de utilizare CPU, respectiv capabilitățile de partajare prin intermediul GuestEntity.

1. Modelul GuestEntity

Fie că vorbim despre mașini virtuale, containere sau funcții serverless, toate entitățile virtuale care au nevoie de un host pentru a rula se modelează prin intermediul interfeței GuestEntity, definind comportamentele standard asociate, fie ele execuția sarcinilor, gestionarea de resurse sau interacțiunea cu entitățile gazdă. În acest fel, VM-urile și containerele nu sunt percepute drept entități de sine, independente, cum erau în versiunile anterioare, ci drept implementări ale aceleiași interfețe, reducându-se astfel definitoriu complexitatea sistemului și permițând totodată scenarii de tip hibrid. Dacă vorbim despre exemplificarea acestor aspecte, trebuie să menționăm notificarea evenimentelor către HostEntity, actualizarea stării interne, fe că vorbim despre idle, running sau waiting, respectiv gestionarea cozii de cloudlet-uri în baza scenariilor alese, sau mai bine zis, CloudletScheduler.

1. Modelul VirtualEntity

VirtalEntity, un model introdus prin intermediul CloudSim 7G, poate fi perceput atât ca un host, cât și ca un guest, întrucăt poate găzdui și totodată poate fi găzduită de alte entități, extinzând funcționalitatea arhitecturii pentru a putea permite simularea de virtualizări imbricate. Așadar, acest model permite rularea de containere într-un VM, permite simularea edge device-urilor care funcționează ca mini-centre de date și chiar permite definirea unor ierarhii flexibile de entități virtuale și fizice. Un exemplu concludent al acestor lucruri poate fi situația în care un VirtualEntity are rolul de container pe un VM, însă găzduiește guest-uri de nivel inferior la rândul său.

1. Modelul Cloudlet și ciclul de viață generalizat

Prin termenul de Cloudlet, înțelegem sarcina de lucru sau task-ul, în termeni specifici, ce urmează afi executată în baza unui scenariu simulativ, iar în CloudSim 7G, cloudlet-uril au fost generalizată pentru a permite existența mai multor categorii, respectiv execuții simple, sau MI, execuții în etape sau workflow-uri, comportamente reconfigurabile prin CloudletScheduler și nu în ultimul rând, deadline-uri și QoS personalizate. Dezvoltând informațiile prezentate până acum, ciclul de viață al unui cloudlet este gestionat de CoudletScheduler prin intermediul a trei metode, și anume update logic handler, ce actualizează progresul execuției, stop condition handler-ul, care definește momentul opririi și unpause logic handler, ce reia execuția unui cloudlet în așteptare. Astfel, se induce o flexibilitate ce avantajează suportul simultan pe mai multe tipuri e cloudlet-un în contextul unui singur scenariu, fără a apărea conflicte în acest sens.

1. Politica de selecție

Nu putem omite prezentarea modelului abstract „Selection Policy”, adică politica de selecție, care joacă un rol esențial în gestionarea eficientă a resurselor în mediile de tip cloud. Aceste politici definesc mecanismele folosite pentru selecția entităților relevante cum ar fi host-uri, mașini virtuale sau unități de procesare, în funcție de diverse criterii sau strategii de optimizare.

În cadrul CloudSim, politicile de selecție sunt implementate sub forma unor clase abstracte sau interfețe, care permit extinderea și personalizarea comportamentului de selecție pentru diverse scenarii de simulare, acestea oferind flexibilitate în modelarea proceselor decizionale ce apar în alocarea si migrarea resurselor.

Un exemplu relevant este clasa VmAllocationPolicy, care reprezintă logica de alocare a mașinilor virtuale pe host-uri, dar și politici mai avansate precum VmAllocationPolicyMigrationStaticThreshold, utilizate în contexte ce implică optimizarea consumului energetic. De asemenea, CloudSim include politici speciale pentru selectarea mașinilor virtuale în contextul de migrare, grupate sub clasa abstractă VmSelectionPolicy. Dintre strategiile implementate, se remarcă VmSelectionPolicyMinimumUsage, care selecteaza mașina virtuală cu cel mai redus nivel de utilizare, sau VmSelectionPolicyMaximumCorrelation, care identifica mașina virtuală a cărei utilizare influențează cel mai mult corelarea sarcinilor pe acel host.

Astfel, cu alte cuvinte, politica de selecție este un model abstract care definește modul în care o resursă este selectată dintr-un set de resurse disponibile, de exemplu ce procesor va fi ales pentru executarea unui anumit task, ce mașină viruala va fi selectată pentru procesul de migrare sau ce host este cel mai potrivit pentru găzduirea unei noi mașini virtuale.

1. Modelul de simulare cu eveimente discrete CloudSim 7G

Modelul de simulare determinist bazat pe evenimente discrete, unde fiecare acțiune se asociază cu un timp și este procesată crogologic de către un motor central numit Simulațion engine este menținut în cadrul CloudSim 7G, îmntrucât principalele avantaje sunt reprezentate de repetabilitate și control asupra stării, de către decuplarea dintre entități și execuție, respectiv simularea realistă a fluxurilor de execuție, care conține, în ordinea enumerării, deployment, scheduling, execuțion și completion.

Astfel, în baza tuturor informațiilor prezentate în acest subcapitol, putem concluziona că CloudSim 7G introduce o arhitectură actualizată, unificată a entităților simulate, clasele specializate redundante fiind înlocuite de către interfețe abstracte puternic generalizate, iar prin modele cum sunt HostEntity, GuestEntity sau VirtualEntity, împreună cu politicile de selecție și planificare, procedura cercetării devine una flexibilă, extensibilă și totodată precisă. Cu alte cuvinte, simulatorul CloudSim 7G reușește să creeze o bază teoretică solidă și totodată modulară, cu un rol esențial în analiza algoritmilor de scheduling și a scenariilor de optimizare în ceea ce înseamnă cloud computing.

1. Algoritmi de planificare analizați

Planificarea de sarcini în domeniul cloud computing-ului reprezintă o procedură esențială în vederea utilizării eficiente a resuselor și a optimizării performanței în general, prin intermediul selectării unei strategii potrivite alocării sarcinilor sau a cloudlet-urilor către Mașinile Virtuale sau VM-uri, în baza disponibilității, pentru atingerea unui obiectiv, fie el reprezentat de către reducerea timpului total de execuție, echilibrarea sarcinilor sau minimizarea consumului de energie. Astfel, această aplicație simulează trei categorii centrale de algoritmi de scheduling, respectiv algoritmi clasici sau basic, euristici și metaeuristici, acompaniați de o strategie privind un consum energetic mai redus, folosing un model de consolidare și alocare dinamică a VM-urilor.

1. FCFS sau First-Come First-Served

Momentul în care sarcinile au fost generate au ordinea de execuție stabilită de algoritmul FCFS, considerându-se că sarcinile sunt sortate în ordine cronologică, iar planificatorul le atribuie către primele VM-uri disponibile. Totuși, trebuie menționat faptul că acest algoritm nu ia în calcul nici dimensiunea, nici complexitatea sarcinii. Pentru exemplificarea acestor aspecte, avem pseudocodul de mai jos:

FCFS(C, V)

1. se sortează lista C dupa timpul de sosire
2. index\_vm 0
3. pentru fiecare cloudlet c in C:
4. vm V[index\_vm]
5. asociaza cloudlet c cu vm
6. index\_vm (index\_vm + 1) mod |V|
7. rulează simularea CloudSim
8. returnează atribuirea cloudlet-urilor la VM-uri

Unde avem C ce reprezintă lista de cloudet-uri, adică sarcinile ce trebuie executate, care ulterior vor fi sortate după timpul de sosire sau ID, respectiv lista de mașini virtuale disponibile reprezentată de V, acest scenariu evidențiind modul in care sarcinile sunt lansate secvențial și executate de îndată ce o mașină virtuală devine disponibilă.

1. Round Robin

Funcționând prin rotație, acest algoritm alocă un interval de timp fix fiecărei sarcini în parte, acest interval fiind cunoscut drept quantum, iar după fiecare execuție parțială, sarcinile ce au rămas nefinalizate sunt reintroduse în coadă. În tot acest timp, VM-urile execută ciclic cloudlet-urile, fără prioritizarea sarcinilor după un anumit criteriu, fie el dimensiune sau deadline. Cu titlu de exemplu, putem lua următoarea situație:

RoundRobin(C, V, quantum)

1. index\_vm 0
2. Pentru fiecare cloudet c in C:
3. vm V[index\_vm]
4. asociaza cloudlet c cu vm
5. index\_vm (index\_vm + 1) mod |V|
6. rulează simularea CloudSim
7. returnează atribuirea cloudlet-urilor la VM-uri

Unde avem C reprezentând lista de cloudet-uri, adică sarcinile ce trebuie executate, lista de mașini virtuale disponibile reprezentată de V, implicit quantum-ul, ce reprezintă timpul exact alocat unei cereri in fiecare iterație, care nu există in simulare, însă se poate folosi cu scopul de a planifica sarcini de intrerupere.

Dupa cum se vede mai sus, acest algoritm are o tehnică de planificare echitabilă, unde fiecare cerere primește o porțiune fixă de timp procesor, pe care am numit-o quantum, cererile fiind procesate ciclic, într-o coadă FIFO, altfel spus, primul venit, primul servit. În mod contrariu, dacă o cerere nu se finalizează în timpul alocat, este reintrodusă la finalul cozii pentru următoarea iterație.

1. Random Scheduler

Acest algoritm asociază fiecare sarcină în mod aleatoriu unui VM, fără a exista reguli de prioritate ori selecție, fiind o simplă distribuție aleatorie a respectivelor sarcini, demonstrându-și utlilitatea în crearea de scenarii de test ori stres în care nu este urmărită o optimizare explicită. Exemplificând, avem:

RandomScheduler(C, V)

1. pentru fiecare cloudlet c in C:
2. vm valoare\_random\_din(V)
3. asociaza cloudlet-ul c cu vm
4. rulează simularea CloudSim
5. returnează atribuirea cloudlet-urilor la VM-uri

Rezumând cele prezentate, C însumează lista de cloudlet-uri, respectiv V, lista de mașini virtuale. Astfel, această abordare este simplă si nedeterministă, iar fiecare cloudlet este atribuit aleatoriu unei mașini virtuale disponibile.

1. LJF sau Longest Job First

În acest algoritm, au regim prioritar sarcinile consumatoare de resurse, fiind sortate descrescător în baza lungimii acestora, cele mai lungi sarcini fiind alocate progresiv înspre VM-uri disponibile, sub forma:

LJF(C, V)

1. sortarea listei de C in ordine descrescătoare dupa lungimea cloudlet-ului
2. index\_vm 0
3. pentru fiecare c in C:
4. vm V[index\_vm]
5. se asociază cloudlet-ul c cu vm
6. index\_vm (index\_vm + 1) mod |V|
7. rulează simularea CloudSim
8. returnează atribuirea cloudlet-urilor la VM-uri

Cu alte cuvinte, C reprezintă lista de cloudlet-uri care sunt ordonate descrescător după durata estimată, adică lungimea lor, respectiv V care e lista de mașini virtuale, cele mai lungi sarcini sau cloudlet-uri fiind astfel programate primele.

1. Min-Min

Acest algoritm identifică timpul minim de execuție pentru fiecare VM în parte, iar ulterior efectuării unui calcul, este selectată sarcina ce se poate finaliza în cel mai scurt timp, dupa fiind alocată VM-ului corespunzător, proces reluat până la planificarea completă a tuturor sarcinilor existente. Transpunerea acestor informații sub formă de pseudocod se face astfel:

MinMin(C, V)

1. cât timp lista C nu este goală:
2. pentru fiecare cloudlet c in C:
3. pentru fiecare VM v in V:
4. T[c][v] timpul estimat de finalizare, dacă c ruleaza pe v
5. caută combinatia (c\_min, v\_min) cu cel mai mic T[c][v]
6. se alocă c\_min la v\_min
7. se elimina c\_min din lista C
8. rulează simularea CloudSim
9. returnează atribuirea cloudlet-urilor la VM-uri

Astfel, această abordare pune accentul pe estimarea timpilor de execuție, selectând mai exact, combinația în care un cloudlet se finalizează cel mai repede, dintre toate opțiunile posibile, unde C reprezintă lista de sarcini sau cloudlet-uri, respectiv V lista de mașini virtuale.

1. MAX-MIN

Cu o logică asemănătoare algoritmului prezentat anterior, există totuși o diferență prin faptul că în cadrul acestuialgoritm este selectată sarcina cu timpul de execuție cel mai mare dintre toate, sarcină ce se alocă VM-ului cu cea mai eficientă putere de procesare, urmărindu-se evitarea acumulării de sarcini costisitoare la final. În acest caz, avem:

MinMin(C, V)

1. cât timp lista C nu este goală:
2. pentru fiecare cloudlet c in C:
3. pentru fiecare VM v in V:
4. T[c][v] timpul estimat de finalizare, dacă c rulează pe v
5. pentru fiecare c din C, caută v\_min pentru care T[c][v\_min] este minim
6. c\_max cloudlet-ul cu cel mai mare T[c][v\_min]
7. se alocă c\_max la v\_min
8. se elimina c\_min din lista C
9. rulează simularea CloudSim
10. returnează atribuirea cloudlet-urilor la VM-uri

Reluând ideea de mai sus, se va căuta sarcina cu timpul minim estimat de finalizare pe o mașină virtuală, ulterior fiind selectată sarcina cu cel mai mare timp minim estimat de realizare ce poate fi alocată pe acea mașină virtuală, unde C reprezintă lista de cloudlet-uri sau sarcini, implicit V lista de mașini virtuale.

1. MinLenghtRoundRobin

Algoritm hibrid bazat pe sortarea sarcinilor în ordine crescătoare în funcție de lungime, acesta le ditribuie ulterior circlar pe VM-uri, combinând logica Min-Min cu cea de tip Round Robin, distribuind sarcinile scurte în mod uniform în cadrul sistemului.

MinLengthRR(C, V)

1. se sortează crescător lista C dupa lungimea cloudlet-ului
2. index\_vm 0
3. Pentru fiecare cloudet c in C:
4. vm V[index\_vm]
5. asociaza cloudlet c cu vm
6. index\_vm (index\_vm + 1) mod |V|
7. rulează simularea CloudSim
8. returnează atribuirea cloudlet-urilor la VM-uri

Cu alte cuvinte, se pune accentul pe prioritizarea sarcinilor de lungime scurtă, având o distribuție ciclică, ideală pentru evitarea supraîncărcării și reducerea timpului de execuție, unde C reprezintă lista de sarcini sau cloudlet-uri, respectiv V lista de mașini virtuale.

1. ACO sau Ant Colony Optimization

Algoritm metaeuristic inspirat din comportamentul furnicilor, fiecare dintre furnici clădește o soluție de alocare a sarcinilor pe VM-uri în baza unor probablități influențate de o matrice de feromoni alăturată unei funcții euristice. Ulterior fiecărei iterații, feromonii sunt actualizați în baza celei mai optime soluții identificate.

ACO(C, V, nrAnts, nrIter, ρ, α, β)

1. Pentru fiecare i din C, respectiv j din V:
2. Se inițializează feromonul τ[i][j] cu o valoare mică pozitivă
3. Pentru fiecare k în nrIter:
4. Pentru fiecare furnică a din nrAnts:
5. Pentru fiecare cloudet in C:
6. Pentru fiecare VM in V:
7. Calculează timpul estimat T[i][j] = cloudletLength() / vmMips()
8. Calculează probabilitatea P[i][j] = \* ( / toate VM-uri
9. Alege VM-ul cu probabilitate P[i][j]
10. Se asociază cu
11. Se calulează makespan
12. Selectează cea mai buna soluție
13. Pentru fiecare i din C, respectiv j din V:
14. Se evaporă feromonul: (1 – ρ) \*
15. Se adaugă feromon pentru solutia buna:
16. Pentru fiecare , din solutia aleasa:
17. + , unde / cost\_solutie
18. Returnează cea mai buna mapare cloudlet-VM

Unde avem parametrii α, β, care controlează cat de mult contează feromonul α, respectiv euristica β, ρ fiind rata de evaporare, ce variază intre 0 și 1 și reduce treptat feromonul vechi, τ[i][j] care reprezintă nivelul de feromon folosit pentru alegerea cloudletului pe mașina virtuală , T[i][j] care indică timpul estimat de execuție dacă cloudletul ruleză pe mașina virtuală , nu putem omite nici P[i][j] folosit pentru calcularea probabilitații influențată de feromon si euristica în alegerii mașinii virtuale pentru cloudletul .

Inspirat din comportamentul natural al coloniilor de furnici de a căuta cea mai scurtă rută posibilă către hrană, ACO transpune felul în care furnicile comunică prin feromoni pentru a permite altor furnici să urmeze traseul pe care acestea l-au avut anterior. Altfel spus, concentrația de feromon este mai mare cu cât calea parcursă este mai scurtă și implicit, crescând posibilitatea ca aceasta să fie parcursă mai frecvent.

Transpunând în practică, se urmează gradual mai multe etape, starea inițială constând în probabilitatea egală a tuturor căilor de a fi alese, ulterior furnicile explorând complet aleatoriu căile dintre cuib, respectiv nodul sursă, și hrană sau nodul destinație. Odată ajunse la respectiva destinație, primele sosite întăresc drumul cel mai scurt cu ajutorul feromonilor, iar în timp, rutele mai lungi își pierd parțial atractivitatea datorită evaporării feromonilor, colonia înreptându-se în acest fel înspre drumul cel mai scurt, sau mai bine zis, cel mai eficient, putându-se observa clar o analogie între acest proces și cel de optimizare a sarcinilor în cloud.

1. Genetic Algorithm

Algoritmul genetic sau GA reprezintă o metoda de optimizare stocastică inspirată de principiile selecției naturale și ale evoluției biologice, așa cum au fost formulate de Charles Darwin. Conceput inițial de John Holland în anii 1970, acest algoritm face parte din clasa tehnicilor de calcul evolutiv si este utilizat extensiv pentru rezolvarea problemelor de optimizare combinatorie, căutare euristică respectiv învațare automată. Astfel, procesul evolutiv simulat de acest algoritm se desfașoară pe baza unei populații de indivizi, fiecare reprezentând o posibilă soluție la o problemă dată. [16]

Este important sa menționăm etapele parcurse de acest algoritm, așadar, în primul rând, se pornește cu inițializarea aleatorie a unei populații inițiale de cromozomi, unde fiecare cromozom va codifica o soluție potențiala a spațiului de căutare. Următorul pas va fi reprezentat de asocierea unui scor de fitness fiecărui individ, calculat printr-o funcție obiectivă ce cuantifică calitatea unei soluții candidate, această funcție fiind diferită în funcție de domeniul problemei. Așadar, în contextul de cloud computing, am folosit urmatoarea funcție:

unde reprezintă mulțimea sarcinilor alocate vm-ului j, este lungimea sarcinii I, respectiv MIPS al mașinii virtuale. Este important să menționăm si selecția părinților pentru reproducere pe baza valorii de fitness, fiind favorabili cei cu adaptabilitate superioară, cele mai utilizate strategii fiind reprezentate de selecția prin ruletă, selecția prin turneu, implicit seleția ierarhică. Nu putem omite nici crossover-ul sau recombinarea, unde doi cromozomi părinți sunt combinați pentru generarea de descendenți, operație realizată printr-un singur punct de tăiere, prin doua sau uniform. Apoi pentru a menține diversitatea genetică a populației și pentru prevenirea convergenței, se va aplica o mutație aleatorie asupra unor gene individuale, introducând trăsături noi în populație, implicit contribuind la explorarea eficientă a spațiului de soluții. Nu în ultimul rând, în final, se va forma o nouă populație prin înlocuirea completă sau parțială a celei anterioare, utilizând o combinație între părinți si copii sau doar descendenți generați.

Așadar, acest algoritm evoluează prin selecție, crossover și mutație, folosind o populație de soluții, sau altfel spus, cromozomi, soluțiile fiind evaluate cu ajutorul unei funcții de fitness, cele mai bune fiind ulterior combinate pentru a se obține soluții noi, optime [17].

GA(C, V, N, G, Pc, Pm)

1. Se va inițializa populația: Populație Generează N cromozomi în mod aleatoriu, unde fiecare cromozom este un vector de n gene, adică VM-ul ales pentru fiecare cloudlet
2. Calcularea fitness-ului fiecărui cromozom: fitness(cromozom) = 1 / makespan(cromozom)
3. Pentru fiecare g din G:
4. Populație\_nouă
5. Repetă de N/2 ori:
6. Alege doi părinți din Populație
7. Dacă rand() < Pm atunci:
8. Copii crossover(părinte1, părinte2)
9. Altfel:
10. Copii Copii identici cu părinții
11. Pentru fiecare copil:
12. Pentru fiecare genă i:
13. Dacă rand() < Pm atunci:
14. Copil[i] alege aleatoriu un VM diferit
15. Se adaugă copiii in Populație\_nouă
16. Calculează fitness-ul fiecarui cromozom din Populație\_nouă
17. Populație selectează cei mai buni N cromozomi din Populație Populație\_nouă
18. Returnează cel mai bun cromozom din Populație, adică maparea optimă dintre cloudlet și VM

Folosind o exprimare diferită, avem C, lista de cloudlet-uri, V care este lista de mașini virtuale, N va fi dimensiunea populației, G numărul maxim de generații, Pm probabilitatea de corssover, respectiv Pm probabilitatea de mutație. Este important să menționăm și operația de crossover care reprezintă îmbinarea a două mapări pentru generarea uneia noi, mutația care, pentru menținerea diversității în populație, va schimba în mod aleatoriu o masină vituală pentru un task sau mai bine zis, cloudlet, respectiv funcția de fitness care evidentiază cât de bună este soluția la care s-a ajuns, ce implică și makespan-ul, respectiv timpul maxim de execuție al oricărei mașini virtuale.

Această abordare va lua în considerare starea curentă a sistemului, cât și datele găsite anterior, va anticipa impactul unei noi alocări de mașini virtuale asupra sistemului, urmând ca în final să fie selectată soluția ce constă în minimizarea impactului asupra echilibrului de încărcare.

1. PSO sau Particle Swarm Optimization

Particle Swarm Optimization este o metodă metaeuristică de optimizare bazat pe populație, dezvoltat de James Kennedy si Russ Eberhart in 1995, fiind inspirat de mișcarea în grup a păsărilor care caută hrană sau de alte comportamente colective din natură. Acest algoritm are la bază o paradigma de tip swarm intelligence, în care un ansamblu de agenți numiți particule explorează simultan spațiul soluțiilor în scopul identificării unei configurații optime a unei funcții obiectiv.

[18]

PSO(C, V, S, G, W, c1, c2) [18]

1. Se inițializează un roi de S particule, pentru fiecare particulă p:
2. Poziția p.pos, o mapare aleatorie dintre cloudlet și VM
3. Viteza p.vel, o listă de modificări posibile ale mapării anterioare
4. Personal best p.best, adică poziția cu fitness maxim atins de p
5. Global best gBest care e cea mai bună poziție gasită
6. Pentru fiecare generație g din G:
7. Pentru fiecare particula p din roi:
8. Se calculeză fitness(p.pos) = 1 / makespan(p.pos)
9. Dacă fitness(p.pos) > fitness(gBest):
10. p.best p.pos
11. Dacă fitness(p.best) > fitness(gBest):
12. Gbest p.best
13. Pentru fiecare particulă p:
14. Pentru fiecare cloudlet i din C:
15. Se calculează noua viteză:
16. p.vel[i] w \* p.vel[i] + c1 \* rand() \* (p.best[i] – p.pos[i]) + c2 \* rand() \* (gBest[i] – p.pos[i])
17. Se va actualiza poziția nouă: p.pos[i] p.pos[i] + p.vel[i]
18. Se returneză gBest ca soluție finală, adică cea mai bună mapare dintre cloudlet și VM

Astfel, avem C ca fiind lista de cloudlet-uri sau sarcini, V care reprezintă lista de mașini virtuale, S drept numărul de particule sau de soluții candidate, G numărul de iterații, respectiv c1 care este coeficientul de atracție cognitivă, adică tendința particulei de a se întoarce spre cea mai bună solutie găsită de către sine însăși, respectiv c2 coeficientul de atracție social, anume tendința unei particule de a se îndrepta spre cea mai bună soluție globală gasită de întregul roi, accentul punându-se pe colaborarea dintre particule.

Luând cunoștință de aspectele prezentate mai sus, termenul de particulă se definește ca fiind o posibilă mapare a cloudlet-urilor mașinilor vituale, poziția curentă sau soluția curentă împreună cu viteza particulei reprezentând carateristicile principale ale acestui algoritm. Formula generală a vitezei este un element cheie in acest algoritm, fiind reprezentat de următoarea formulă:

Unde va fi reprezentat de viteza curentă, indicând poziția curentă, adică mai clar, mașina virtuală alocată pentru cloudlet-ul sau sarcina i, folosite pentru introducerea variabilității, cu valori aleatoare cuprise între 0 si 1, respectiv w, care controlează cât de mult poate influența viteza anterioară.

Așadar, particula se va putea îndrepta spre coeficientul de atracție cognitivă, p.best, care va fi cea mai bună soluție pe care aceasta a gasit-o, sau spre celălalt coeficient, cel de atracție socială, care va fi reprezentat de cea mai bună soluție globală, gBest, din întregul roi. Într-un final, se vor actualiza toate soluțiile in mod iterativ, având ca scop o mapare cât mai bună.

1. Strategii propuse pentru reducerea consumului de energie

Pentru a reduce semnificativ consumul de energie din centrele de date, vom folosi două abordări principale reprezentate de scalarea verticală sau dinamică a mașinilor virtuale, respectiv consolidarea mașinilor virtuale slab utilizate.

1. Scalarea dinamică a mașinilor virtuale

În domeniul de cloud computing, adaptarea dinamică a resurselor este esențială pentru a asigura că sistemul poate răspunde provocărilor apărute pe parcurs fără a consuma inutil resurse, astfel fiind evitată supraîncarcarea sistemului.

scalareVMs(broker, V, th)

1. loadPerVM nrCloudlets / nrVMs
2. dacă loadPerVM > th :
3. nrVMsuplimentare ceil((loadPerVM / th – 1) \* nrVMs)
4. pentru i de la 1 la nrVmsuplimentare:
5. creează un nou VM standard
6. adaugă noul VM la V
7. actualizează V în broker

Unde avem valori de intare broker-ul, care este obiectul din CloudSim ce are ca scop principal trimiterea cloudlet-urilor către mașinile virtuale, adică face comunicarea dintre utilizator și centrul de date, V, reprezentând lista de mașini virtuale, implicit th ce este pragul limită care indică valoarea maximă de cloudlet-uri sau sarcini ce vor fi acceptate pe o mașină virtuală.

Astfel, pentru reducerea supraîncărcării de pe mașinile virtuale deja existente, în primul rând, se va analiza sarcina medie pe fiecare mașină virtuală, adică valoarea medie de cloudlet-uri care sunt alocate pe o mașină virtuală activă, ulterior verificându-se dacă încărcarea pe fiecare mașină virtuală depășește limita acceptată, situație în care ar fi adăugate mașini virtuale noi.

1. Consolidarea mașinilor vituale

Algoritmul de consolidare are ca scop principal reducerea radicală a consumului energetic, întrucât la fiecare iterație se calculează variația estimată a energiei pentru fiecare alocare posibilă, iar dacă se ajunge la situația identificării unui host ce nu este utilizat, se va declanșa consolidarea, adică migrarea mașinilor virtuale, respectiv închiderea host-urilor.

consolidare(C, V)

1. se mapează VM la o lista de cloudlet-uri
2. pentru fiecare v din V:
3. dacă v nu are cloudlet-uri alocate sau are utilizare < 30%:
4. găsește alt v’ cu utilizare < 70%
5. mută cloudlet-urile de pe v pe v’
6. marchează v pentru închidere
7. elimină VM-urile neutilizate din V
8. actualizează V în broker

Prin intermediul unei exprimări diferite, ca valori de intare vom avea C, reprezentând lista de cloudlet-uri sau sarcini, respectiv V care este lista de mașini virtuale. Astfel, pentru a vedea cât de încărcată este o mașină virtuală, vom avea nevoie de ajutorul unei structuri de tip dicționar, cu scopul de a mapa fiecărei mașini virtuale o listă de sarcini sau cloudlet-uri alocate. Ulterior, pentru a identifica mașinile virtuale candidate pentru consolidare, vor fi selectate doar cele care nu au sarcini alocate sau sunt inutilizate, respectiv slab utilizate, cu un procentaj de mai puțin de 30% din capacitatea de MIPS sau milioane de instrucțiuni pe secundă ocupată. Așadar, prin intermediul acestei proceduri, sarcinile vor fi grupate mai eficient, reducându-se semnificativ consumul de energie.

Pe parcursul acestui capitol, au fost prezentate în mod detaliat analiza conceptuală, analiza logică și analiza tehnică a soluției înaintate pentru reducerea consumului de energie prin intermediul programării sarcinilor în mediul cloud computing-ului, fiind desriși amănunțit algoritmii clasici precum FCFS, Round Robin sau Min-Min ori metaeuristici precum Genetic, ACO, PSO, împreună cu raționamentele ce au stat la baza selecției fiecăruia. În plus, au fost introduse două strategii complementare fundamentale în îmbunătățirea eficienței energetice, și anume scalarea dinamică a mașinilor virtuale și consolidarea acestora, două metode ce îmbină eficient reducerea timpului total de execuție și diminuarea consumului de resurse în exces, fiind totodată argumentată alegerea mediului de simulare CloudSim, ce permite modelarea realistă a interacțiunilor dintre componentele unui centru de date virtual, nefiind necesară infrastructura fizică reală. De asemenea, prin intermediul integrării unei baze de date relaționale și utilizarea protocolului JDBC, a fost asigurată persistența și posibilitatea reutilizării rezultatelor experimentale în mod scalabil și eficient. Având în vedere toate aspectele prezentate în rândurile de mai sus, putem determina că soluția înaintată oferă un cadru robust pentru testarea și evaluarea strategiilor de programare a sarcinilor în cloud computing, accentuând atât obiectivele funcționale ale proiectului, precum și cerințele de eficiență energetică.

Lipsesc referinte la pseudocod, text, figuri, typos

5. Proiectare de detaliu și implementare

1. 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. Prezentarea generală a funcționalităților
2. Cerințe funcționale

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.

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 5.1.1.1 Diagrama Use Case

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. Cerințe non-funcționale

În ceea ce privește cerințele non-funcționale, acestea însumează mai multe caracteristici esențiale pentru asigurarea bunei funcționări a aplicației, și anume performanța, întrucât afișarea de rezultate și salvarea acestora trebuie să se facă instant, și totodată timpul în care se execută simularea trebuie să fie unul rezonabil, ușurința în utilizare, întrucât interfața grafică trebuie sa aibă butoane clare și mesaje concrete, pentru a fi intuitivă și prietenoasă cu utilizatorul, portabilitatea, pentru asigurarea funcționării corecte indiferent de sistemul de operare folosit, scalabilitatea, pentru adăugarea facilă de noi algoritmi de scheduling sau parametri de simulare, respectiv fiabilitatea, oferind mesaje de eroare corecte în cazul introducerii parametrilor de intrare invalizi sau a unei conexiuni pierdute la baza de date. În plus, nu putem face abstracție nici de extensibilitate, în sensul în care codul sursă trebuie să fie modular pentru a permite introducerea de noi funcționalități, fără a le afecta pe cele deja introduse, respectiv de persistență, pentru a asigura disponibilitatea simulărilor salvate în baza de date, pentru consultarea ulterioară.

1. Arhitectura generală a aplicației
2. 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.

A diagram of a cloud computing system

AI-generated content may be incorrect.

Figura 5.2.1.1 Diagrama de secvență pentru pornirea simulării

Cu alte cuvinte, diagrama de secvență de mai sus ilustrează procesul de lansare a unei simulări ce nu folosește metode de a crește eficiența energetică în cadrul acestei aplicații, pornind de la interacțiunea utilizatorului și continuând cu apelurile interne dintre componentele aplicației. Astfel, utilizatorul interacționează cu interfața grafică reprezentată de clasa MainGUI prin apăsarea butonului „Start Simulation”, declanșând metoda clickStartSimulation(). Clasa MainGUI apelează metoda runSimulation() din componenta EnergySimulatorNormal, responsabilă cu gestionarea simulării în conditii normale, fără a utiliza metode de creștere a eficinței energetice. În interiorul EnergySimulatorNormal, sunt apelate două metode pentru inițializarea resurselor necesare și anume createVMs() din clasa VMManager, care creează lista de mașini virtuale, respectiv createCloudlets() din clasa CloudletManagerm care generează sarcinile ce urmează a fi executate. După ce s-au configurat resursele, metoda CloudSim.startSimulation() este invocată pentru a porni efectiv simularea, urmând ca, după finalizarea execuției, procesarea rezultatelor să fie făcută prin intermediul metodei printResults() din ResultsPrinter, care caculează si formatează datele aferente fiecărui cloudlet, iar în cele din urmă, datele vor fi transmise înapoi către MainGUI prin apelul displayResults(results), pentru a fi afișate în interfața grafică.

A diagram of a diagram

AI-generated content may be incorrect.

Figura 5.2.1.2 Diagrama de secvență pentru salvarea rezultatelor simulării

Această diagramă de secvență ilustează procesul de salvare a rezultatelor simulării în baza de date, imediat după rularea simulării, reflectând o arhitectură clar structurată pentru persistența datelor, separând logica de simulare de logica de stocare. Așadar, clasa EnergySimulatorNormal apelează metoda save(simulationSummary, cloudletsResults) din componenta SaveSimulation, trimițând datele rezultate din simulare. Save simulation deschide o conexiune la baza de date prin metoda connect(), utilizând clasa DatabaseManager. Înainte de inserarea datelor, este apelată metoda createTablesIfNotExist() pentru a verifica si crea dacă este cazul tabelele simulation\_summary, respectiv simulation\_results. După ce infrastructura este pregătită, se execută inserarea propriu-zisă, și anume insertSimulationSummary() pentru datele generale ale simulării, respectiv insertCloudletResults() pentru resultatele individuale ale fiecărui cloudlet, urmând ca la final, DatabaseManager să trimită o confirmare către SaveSimulation, semnalând salvarea cu succes.

A diagram of a data flow

AI-generated content may be incorrect.

Figura 5.2.1.3 Diagrama de secvență pentru încărcarea unei simulări salvate și afișarea graficului

Diagrama de secvența ilustrată mai sus prezintă procesul prin care utilizatorul încarcă o simulare salvată anterior în baza de date, selectează una dintre simulări și vizualizează rezultatele sub formă de grafic. Astfel, utilizatorul interacționează cu interfața MainGUI prin apăsarea butonului „Load” declanșând metoda clickLoad(). MainGUI apelează metoda loadSimulationSummaries() din componenta SimulationSummaryLoad, responsabilă cu preluarea informațiilor generale ale unei simulări din baza de date. SimulationSummaryLoad comunică cu DatabaseManager, solicitând lista informațiilor generale ale unei simulări prin fetchSummaryList(), urmând să returneze lista simulărilor salvate. După primirea listei, MainGUI apelează showTableToSelect() pentru a afișa utilizatorului un tabel cu simulările disponibile, urmând ca utilizatorul să selecteze o simulare, iar MainGUI să trimită identificatorul simulation\_id prin select(simulation\_id) și să apeleze loadCloudletResults(simulation\_id). Ulterior, SimulationSummaryLoad preia rezultatele individuale ale cloudlet-urilor de la DatabaseManager, care le returnează sub forma unui set de date cloudletResults, iar în final, MainGUI trimite aceste date către ResultsPrinter prin generateChart(cloudletResults), care generează și afișează graficul corespunzător simulării selectate. Această secvență subliniază modularitatea sistemului și claritatea cu care sunt separate responsabilitățile reprezentate de interfața utilizator, logica de încărcare, precum și logica de vizualizare.

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. 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 de clase

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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | 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 {  int numUsers = 1;  Calendar calendar = Calendar.getInstance();  CloudSim.init(numUsers, calendar, false);  Datacenter datacenter = createDatacenterNormal("Datacenter\_0", *numHosts*, *hostMIPS*, *hostRAM*);  DatacenterBroker broker = new DatacenterBroker("Broker");  List<Cloudlet> cloudletList = createCloudlets(broker.getId(), *numCloudlets*);  broker.submitCloudletList(cloudletList);  SchedulingAlgorithm algorithm;  switch (*algorithmName*) {  case "ACO":  algorithm = new ACO();  break;  //...  default:  algorithm = new MinLengthRoundRobin();  }  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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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.

|  |  |
| --- | --- |
| 1  2  3  4 | 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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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();  } } |

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | 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();  } } |

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | 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());  } } |

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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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().

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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().

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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.

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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.

|  |  |
| --- | --- |
| 1  2  3 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | 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());  } } |

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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | 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());  } } |

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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | 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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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.

|  |  |
| --- | --- |
| 1  2  3  4 | 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 5.3.3 Diagrama Bazei de date

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. 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.

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 5.3.4 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 vizualizare a graficelor

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.

[1] Van Dijk, J. *The Network Society*; Sage Publications: Thousand Oaks, CA, USA, 2012. [1]

[2] S. Yi, C. Li, and Q. Li, "Fog computing: A taxonomy, survey and future directions," in *Proc. 2015 Workshop on Mobile Big Data (Mobidata '15)*, pp. 1–6, ACM, 2015.

[3] Mahmud, R.; Kotagiri, R.; Buyya, R. Fog computing: A taxonomy, survey and future directions. In *Internet of Everything*; Springer: Berlin, Germany, 2018; pp. 103–130.

[4] E. Del-Pozo-Puñal, F. García-Carballeira, and D. Camarmas-Alonso, "A scalable simulator for cloud, fog and edge computing platforms with mobility support," *Future Generation Computer Systems*, vol. 144, pp. 117–130, 2023.

[5] M. Fahimullah, G. Philippe, S. Ahvar, and M. Trocan, "Simulation tools for fog computing: A comparative analysis," *Sensors*, vol. 23, no. 7, p. 3492, 2023.

[6] K. Kok, S. Karnouskos, D. Nestle, A. Dimeas, A. Weidlich, C. Warmer, *et al*., "Smart houses for a smart grid," in *Proc. 20th Int. Conf. and Exhibition on Electricity Distribution (CIRED 2009)*, Stevenage, UK: IET, Jun. 2009, p. 0751.

[7] R. Andreoli, J. Zhao, T. Cucinotta, and R. Buyya, "CloudSim 7G: An integrated toolkit for modeling and simulation of future generation cloud computing environments," *Software: Practice and Experience*, 2025. (în curs de apariție / in press, dacă nu are volum și pagini)

[8] M. A. Shahid, M. M. Alam, and M. M. Su’ud, "A systematic parameter analysis of cloud simulation tools in cloud computing environments," *Applied Sciences*, vol. 13, no. 15, p. 8785, 2023.

[9] M. Ashouri, F. Lorig, P. Davidsson, and R. Spalazzese, "Edge computing simulators for IoT system design: An analysis of qualities and metrics," *Future Internet*, vol. 11, no. 11, p. 235, 2019.

[10] P. Velho and A. Legrand, "Accuracy study and improvement of network simulation in the SimGrid framework," in *Proc. 2nd Int. Conf. on Simulation Tools and Techniques (SIMUTools '09)*, Mar. 2009.

[11] S. N. A. Jawaddi and A. Ismail, "Integrating OpenAI Gym and CloudSim Plus: A simulation environment for DRL agent training in energy-driven cloud scaling," *Simulation Modelling Practice and Theory*, vol. 130, p. 102858, 2024.

[12] R. Schmidt, D. Beaty, and J. Dietrich, "Increasing energy efficiency in data centers," *ASHRAE Journal*, vol. 49, no. 12, p. 18, 2007.

[13] Al-Arasi, R. A., & Saif, A. (2020). "Task Scheduling in Cloud Computing Based on Meta-Heuristic Techniques: A Review Paper." *EAI Endorsed Transactions on Cloud Systems*, vol. 6, no. 17, Jan. 2020.

[14] Abid, A., Manzoor, M. F., Farooq, M. S., Farooq, U., & Hussain, M. (2020). "Challenges and Issues of Resource Allocation Techniques in Cloud Computing." *KSII Transactions on Internet and Information Systems (TIIS)*, vol. 14, no. 7, pp. 2815–2839.

[15][CloudSim 7G: An Integrated Toolkit for Modeling and Simulation of Future Generation Cloud Computing Environments - Andreoli - 2025 - Software: Practice and Experience - Wiley Online Library](https://onlinelibrary.wiley.com/doi/full/10.1002/spe.3413)

[16] [Genetic Algorithms and Machine Learning | Machine Learning](https://link.springer.com/article/10.1023/A:1022602019183)

[17][Genetic-Algorithm-Guide-[Tom-Mathew].pdf](https://datajobstest.com/data-science-repo/Genetic-Algorithm-Guide-%5bTom-Mathew%5d.pdf)

[18] [Particle Swarm Optimization Algorithm and Its Applications: A Systematic Review | Archives of Computational Methods in Engineering](https://link.springer.com/article/10.1007/S11831-021-09694-4)

# Bibliography

|  |  |
| --- | --- |
| [1] | J. Van Dijk, The Network Society, CA, USA: Sage Publications: Thousand Oaks, 2012. |