

UNIVERSITATEA TEHNICĂ „Gheorghe Asachi” din IAȘI  
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE  
DOMENIUL: Calculatoare și Tehnologia Informației  
SPECIALIZAREA: Tehnologia Informației

## **LUCRARE DE DIPLOMĂ**

**Coordonator științific:**  
**ș.l.dr.ing. Cristian-Mihai Amarandei**

**Absolvent:**  
**Eduard-Vasilică Pușcașu**

**Iași, 2022**



UNIVERSITATEA TEHNICĂ „Gheorghe Asachi” din IAȘI  
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE  
DOMENIUL: Calculatoare și Tehnologia Informației  
SPECIALIZAREA: Tehnologia Informației

# **Planificarea si monitorizarea executiei aplicatiilor intensiv computationale in arhitecturi distribuite**

LUCRARE DE DIPLOMĂ

**Coordonator științific:**  
**ș.l.dr.ing. Cristian-Mihai Amarandei**

**Absolvent:**  
**Eduard-Vasilică Pușcașu**

**Iași, 2022**



**DECLARAȚIE DE ASUMARE A AUTENTICITĂȚII  
PROIECTULUI DE DIPLOMĂ**

Subsemnatul POSCASU EDUARD-VASILICA,  
legitimat cu CI seria ZC nr. 336491, CNP 5000704045370  
autorul lucrării PLANIFICAREA SI MONITORIZAREA  
EXECUTIEI APLICATIILOR INTENSIV COMPUTATIONALE  
IN ARHITECTURI DISTRIBUITE

elaborată în vederea susținerii examenului de finalizare a studiilor de licență, programul de studii TEHNOLOGIA INFORMAȚIEI organizat de către Facultatea de Automatică și Calculatoare din cadrul Universității Tehnice „Gheorghe Asachi” din Iași, sesiunea UNI-E-IULIE a anului universitar 2023, luând în considerare conținutul Art. 34 din Codul de etică universitară al Universității Tehnice „Gheorghe Asachi” din Iași (Manualul Procedurilor, UTI.POM.02 - Funcționarea Comisiei de etică universitară), declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române (legea 8/1996) și a convențiilor internaționale privind drepturile de autor.

Data

06.07.2023

Semnătura





# Cuprins

<b>Introducere</b>	<b>1</b>
<b>1 Fundamentarea teoretică și documentarea bibliografică</b>	<b>3</b>
1.1 Scopul temei alese . . . . .	3
1.2 Domeniul aplicației propuse . . . . .	4
1.3 Analiza aplicațiilor similare . . . . .	4
1.3.1 Mle-scheduler . . . . .	4
1.3.2 Kubernetes . . . . .	5
1.3.3 Slurm . . . . .	6
1.3.4 Jenkins . . . . .	6
1.4 Elaborarea specificațiilor aplicației . . . . .	7
<b>2 Proiectarea aplicației</b>	<b>9</b>
2.1 Funcționalități propuse . . . . .	9
2.2 Arhitectura software a aplicației . . . . .	9
2.2.1 Nivelul logic . . . . .	11
2.2.2 Interacțiunea utilizatorului cu aplicația . . . . .	11
2.3 Resurse software utilizate . . . . .	11
2.3.1 Interconectarea componentelor principale . . . . .	15
2.4 Informații generale despre implementarea aleasă . . . . .	16
2.4.1 Limitele funcționale . . . . .	16
2.4.2 Analiza SWOT a aplicației dezvoltate . . . . .	16
<b>3 Implementarea aplicației</b>	<b>19</b>
3.1 Comunicarea utilizatorului cu aplicația . . . . .	19
3.2 Descrierea generală a implementării . . . . .	20
3.2.1 Structura aplicației . . . . .	20
3.2.2 Gestionarea resurselor alocate . . . . .	22
3.2.2.1 Obiectul de tip Server . . . . .	22
3.2.3 Stocarea datelor . . . . .	23
3.2.4 Receptionarea unui job . . . . .	24
3.2.5 Mod de utilizare Supervisord . . . . .	25
3.2.6 Implementarea Supervisord in proiect . . . . .	26
3.2.6.1 Crearea mediului pentru rularea joburilor în Supervisord . . . . .	27
3.2.7 Obiectul de tip Job . . . . .	28
3.3 Dificultăți întâmpinate și soluții . . . . .	28
<b>4 Testarea aplicației și rezultate experimentale</b>	<b>31</b>
4.1 Punerea în funcțiune a aplicației . . . . .	31
4.1.1 Pornirea aplicațiilor auxiliare . . . . .	31
4.1.2 Configurare server . . . . .	31

4.1.2.1	Configurație VirtualBox . . . . .	31
4.1.2.2	Instalarea sistemului de operare . . . . .	33
4.1.3	Instalarea Supervisor . . . . .	35
4.2	Rezultate obținute . . . . .	35
<b>Concluzii</b>		<b>37</b>
<b>Bibliografie</b>		<b>39</b>
<b>Anexe</b>		<b>41</b>
1	Structura fișierelor . . . . .	41



# Planificarea si monitorizarea executiei aplicatiilor intensiv computationale in arhitecturi distribuite

Eduard-Vasilică Pușcașu

## Rezumat

Scopul proiectului prezentat este de a adresa o provocare contemporană în domeniul aplicațiilor intensiv computationale, dar și al aplicațiilor clasice care rulează în arhitecturi distribuite. Justificarea alegerii temei vine din nevoia de a eficientiza modul în care sarcinile sunt repartizate pe o infrastructură distribuită, oferind totodată o reprezentare cât mai clară a evenimentelor care se produc în timpul proceselor respective.

Pornind de la această idee, s-a ajuns la dezvoltarea unei aplicații care are sarcina de a primi executabile și de a le distribui către calculatorul corespunzător pentru rulare, acesta fiind ales utilizând un algoritm de sortare. În acest context, am optat ca joburile să fie gestionate conform principiului FIFO și în funcție de disponibilitatea serverelor pe care le avem în dotare.

Proiectul se bazează pe o arhitectură eficientă, care preia serverele din cozi de mesaje pentru a fi utilizate în rularea joburilor viitoare, dar și configurația executabilelor pe care dorim să le rulăm. Configurația implică comanda de rulare și doi indicatori, unul dintre ei specificând dacă acesta trebuie să repornească în caz de eroare, precum și numele fișierului de pe serverul gazdă. După preluarea și validarea mesajului de configurație, și verificarea disponibilității unui server, se pregătește "mediul" pentru rulare, în principal un fișier cu configurația acestuia și două fișiere de jurnal.

În privința implementării proiectului, pentru segmentul de monitorizare, s-au utilizat fire de execuție "watchdog" pentru a asigura urmărirea joburilor în derulare și a furniza un feedback de înaltă calitate în timp real. De asemenea, s-au aplicat cele mai bune practici pentru a dezvolta un cod cât mai corect, durabil și ușor de înțeles. (Folosirea șablonului de proiectare Singleton, OOP etc).

Decizia luată în structurarea proiectului a inclus utilizarea unor aplicații care au facilitat monitorizarea și repornirea joburilor, precum și utilizarea unor cozi de mesaje pentru a adera la principiul FIFO (First In, First Out).

Testarea și monitorizarea aplicației implică utilizarea de jurnale care afișează mesaje din secțiunile critice ale aplicației, precum și o bază de date NoSQL care facilitează accesul la datele generate în timpul rulării, dar și ridicarea unui număr de mașini virtuale pentru a se putea simula arhitectura distribuită.



## Introducere

În ritmul accelerat al progresului tehnologic, de la an la an apar pe piață dispozitive cu o putere de procesare și capacitate de stocare tot mai mare, această progresie generând o necesitate de a dezvolta și implementa metode și instrumente care să administreze eficient aceste resurse abundente.

Una dintre provocările din sfera tehnologiei informației este, fără îndoială, optimizarea utilizării resurselor hardware în sistemele care rulează aplicații de o varietate diferită și complexitate semnificativă. Prin eficiența în acest context, avem în vedere implementarea unei proceduri de administrare a memoriei și a capacității de procesare care să maximizeze performanța fără a discredita integritatea sau funcționarea sistemului.

În contextul acestei provocări contemporane, proiectul propus se axează pe elaborarea unei aplicații, capabile să interacționeze dinamic cu o multitudine de servere. Această soluție software oferă funcționalitatea de a transmite către aceste sisteme diverse executabile cu scopul de a le rula. În plus, aplicația are capacitatea de a monitoriza îndeaproape întregul proces de execuție, oferind o perspectivă detaliată și o gestionare eficientă a resurselor utilizate. Prin implementarea aplicației, utilizatorii vor putea gestiona și monitoriza eficient sarcinile și resursele alocate, maximizând în același timp capacitatea de utilizare a sistemelor.

În urma finalizării proiectului, ne așteptăm să dezvoltăm un produs software capabil să asigure repartizarea și monitorizarea eficientă a aplicațiilor pe o arhitectură distribuită. Prin monitorizare înțelegem capacitatea programului de a oferi o imagine clară a proceselor în derulare, ceea ce presupune afișarea unor informații referitoare la joburile ce se afla în coada sau care rulează în prezent, joburile care au eșuat sau, din contră, care au fost finalizate cu succes. Pe lângă acestea, pot fi afișate și alte informații, precum erorile apărute, cât timp a durat rularea joburilor sau cât de multe resurse sunt disponibile în prezent.

Cu privire la tehnologiile folosite, acestea au fost selectate cu o atenție meticuloasă, implicând o comparație detaliată cu alternativele disponibile pe piață. Prin această abordare, am reușit să evităm utilizarea de aplicații sau tehnologii care sunt fie insuficiente pentru proiect, fie nejustificat de complexe.

S-a optat pentru utilizarea unei aplicații numite Supervisor pentru partea de supraveghere, aceasta devenind din ce în ce mai populară în ultimii ani, fiind utilizată în special în contextul containerelor. Rolul său vital în lucrarea propusă a fost în monitorizare, repornire în caz de întrerupere și furnizarea feedback-ului proceselor în timp real. În afară de Supervisor, RabbitMQ a devenit o ustensilă evidentă în gestionarea mesajelor care urmau să fie procesate, iar MongoDB a furnizat o soluție eficientă pentru monitorizarea proceselor care trebuiau repornite indiferent de eroarea întâmpinată. Astfel, prin utilizarea acestor aplicații principale și a altor instrumente, cum ar fi SSH, proiectul a putut fi finalizat conform cerințelor dorite.

Lucrarea este organizată în patru secțiuni principale, care urmăresc procesul de producție specific pentru documentația unui produs software: documentația, proiectare, implementare și testare. Capitolul 1 acoperă o mică parte din domeniul teoretic din care face parte proiectul, tot aici incluzând și analiza aplicațiilor cu obiective similare. Capitolul 2 se focalizează pe designul aplicației și posibilele scenarii, prezentând și tehnologiile utilizate și scopul pentru care au fost alese în detrimentul altora. Capitolul 3 este o continuare a capitolul 2, explicând implementarea aplicației și abordările posibile pentru problemele propuse. În final, capitolul 4 se ocupă de testare și de rezultatele obținute, dar acoperă și etapa de configurare a proiectului, adică pașii necesari înainte de a lansa aplicația.



## Capitolul 1. Fundamentarea teoretică și documentarea bibliografică

### 1.1. Scopul temei alese

Scopul acestei teme este de a adresa o provocare contemporană în domeniul aplicațiilor intensiv computaționale, dar și al aplicațiilor clasice care rulează în arhitecturi distribuite. Alegerea acestei teme este justificată de necesitatea de a optimiza procesul de distribuire a sarcinilor pe o infrastructură distribuită, iar în același timp de a oferi o monitorizare eficientă a execuției acestora.

În infrastructurile distribuite, aplicațiile pot fi rulate pe mai multe servere, fie ele fizice sau virtuale. Distribuirea și administrarea eficientă a executabilelor pe sistemele puse la dispoziție reprezintă un aspect critic pentru obținerea performanței maxime și utilizarea optimă a resurselor disponibile. Prin urmare, dezvoltarea unei aplicații specializate care să faciliteze repartitia și monitorizarea acestor joburi devine indispensabilă în contextul actual.

Scopul temei este, în primul rând, de a ușura utilizatorul în procesul de distribuire a executabilelor pe infrastructura distribuită. Prin intermediul aplicației propuse, utilizatorul va avea la dispoziție un mecanism simplu și intuitiv pentru a încărca și rula aplicații pe serverele disponibile. Astfel, se va reduce complexitatea acestui proces și se va economisi timpul necesar pentru configurarea și administrarea manuală a fiecărui server în parte.

În al doilea rând, un astfel de proiect aduce numeroase beneficii utilizatorilor prin capacitatea sa de a distribui și monitoriza joburile computaționale. Prin intermediul aplicației, utilizatorii vor putea beneficia de:

- **Eficiență în utilizarea resurselor:** Aplicația permite monitorizarea detaliată a execuției joburilor, furnizând informații precum timpul de execuție, consumul de resurse, erori sau avertismente. Aceasta facilitează identificarea și remedierea problemelor legate de performanță sau erori în timp real, permițând astfel o utilizare mai eficientă a resurselor disponibile.
- **Simplificarea procesului de planificare:** Aplicația propusă ajută utilizatorii să planifice și să distribuie sarcinile computaționale în mod automat sau manual, în funcție de preferințele și cerințele specifice ale fiecărui proiect. Aceasta reduce efortul manual necesar pentru a configura și distribui joburile pe servere, asigurând o planificare mai rapidă și mai precisă a resurselor disponibile.
- **Creșterea scalabilității:** O infrastructură distribuită permite extinderea orizontală a resurselor prin adăugarea de servere suplimentare. Aplicația dezvoltată facilitează procesul de distribuire și monitorizare a joburilor pe aceste servere noi, asigurând astfel scalabilitate și adaptabilitate în funcție de cerințele fluctuante ale aplicațiilor intensiv computaționale.

Astfel, prin dezvoltarea acestei aplicații, se urmărește nu doar simplificarea procesului de distribuire și monitorizare a joburilor computaționale, ci și optimizarea utilizării resurselor disponibile în infrastructurile distribuite, oferind utilizatorilor un instrument puternic pentru a gestiona și maximiza eficiența execuției aplicațiilor intensiv computaționale.

## 1.2. Domeniul aplicației propuse

În ultimii ani, clusterelor<sup>1</sup> și cloud-ul<sup>2</sup> au devenit din ce în ce mai predominante și adoptate în industrie, în special în domeniul infrastructurii IT. Acest trend este rezultatul unor factori cheie care au contribuit la popularitatea și utilizarea extinsă a acestor tehnologii.

În primul rând, eficiența operațională reprezintă un aspect crucial în adoptarea clusterelor și a cloud-ului. Prin consolidarea resurselor de calcul și stocare pe mai multe noduri și prin utilizarea lor distribuită, aceste tehnologii permit o utilizare mai eficientă a infrastructurii și a capacităților disponibile. Astfel, se reduce inactivitatea resurselor și se maximizează performanța și productivitatea sistemelor IT.

De obicei, calculatoarele care sunt incluse în structura sistemelor distribuite utilizează sisteme de operare bazate pe arhitectura UNIX. Alegerea unui astfel de sistem se datorează mai multor motive, printre care se numără stabilitatea și fiabilitatea sa, precum și capacitatea sa de a tolera defecțiuni și a se recupera rapid din situații neprevăzute.

Sistemele de operare UNIX, în special cele derivate din sistemul de operare Unix original oferă un mediu de dezvoltare robust și flexibil pentru construirea și gestionarea sistemelor distribuite. Acest lucru se datorează structurii lor modulare și conceptelor fundamentale, cum ar fi gestionarea proceselor, a fișierelor și a rețelei.

Zona din care face parte proiectul propus se axează pe dezvoltarea și implementarea de strategii eficiente pentru planificarea și monitorizarea sarcinilor de calcul într-un mediu distribuit, cum ar fi un cluster de calculatoare sau un mediu de cloud computing.

În literatura de specialitate lucrarea de licență se poate defini și ca un "job scheduler" sau planificator de sarcini. "Scheduler"-ul este o componentă a sistemelor informatice care se ocupa cu gestionarea și executarea unor sarcini sau procese într-un mediu de calcul distribuit. Este utilizat în mai multe domenii, inclusiv în sistemele de operare, calculul distribuit, cloud computing, calcul de înaltă performanță și gestionarea resurselor în centrele de date.

Planificarea executabilelor joacă un rol crucial în sistemele distribuite, deoarece crearea unui trebuie să abordeze și să rezolve mai multe provocări în același timp. Dificultățile cele mai întâlnite sunt: resursele necesare pentru rularea jobului, diversitatea calculatoarelor care sunt folosite în rezolvarea problemei, sistemul de operare pe care rulează aplicația și tipul jobului. În acest caz, abordarea pentru rezolvarea problemei trebuie să aibă în vedere toate detaliile care pot buna funcționare a acesteia.

## 1.3. Analiza aplicațiilor similare

Există mai multe unelte care se ocupă cu această problemă, inclusiv Jenkins, Kubernetes, Slurm și Mle-scheduler.

### 1.3.1. Mle-scheduler

MLE Scheduler[3] este un planificator de sarcini pentru arhitecturi distribuite care optimizează performanțele sistemului alocând resurse dinamic pentru îndeplinirea joburilor aflate într-o coadă. Acesta poate avea ca workeri mașini virtuale oferite de Google Compute Engine sau servere SSH. Una dintre abilitățile esențiale ale MLE-Scheduler-ului constă în capacitatea de a învăța din datele istorice, prin intermediul algoritmilor de învățare automată. Prin analiza datelor isto-

<sup>1</sup>Clusterelor de calculatoare[1] sunt folosite pentru a rezolva probleme complexe și a procesa volume mari de date într-un timp mai scurt decât ar putea face un calculator performant. Acestea sunt configurate pentru a funcționa împreună și a împărți sarcinile între componente, cum ar fi procesoarele, memoria RAM și memoria de stocare, pentru a obține o eficiență sporită și capacitate de scalare.

<sup>2</sup>Cloud computing[2] sunt servicii furnizate de diverse companii de IT, cele mai populare fiind AWS (Amazon Web Services) și Azure oferit de Microsoft. Prin intermediul acestora utilizatorii poate acceseze și să utilizeze resurse prin intermediul internetului. Infrastructura lor este formată din mai multe cluster de calculatoare unde utilizatorii beneficiază de resurse extinse și de capacitate de procesare pentru a-și îndeplini sarcinile.

rice referitoare la sarcini, Scheduler-ul poate face predicții precise și coerente în ceea ce privește sarcinile viitoare. Acest aspect este deosebit de important în contextul planificării și optimizării resurselor de timp și energie într-un mediu dinamic și complex.

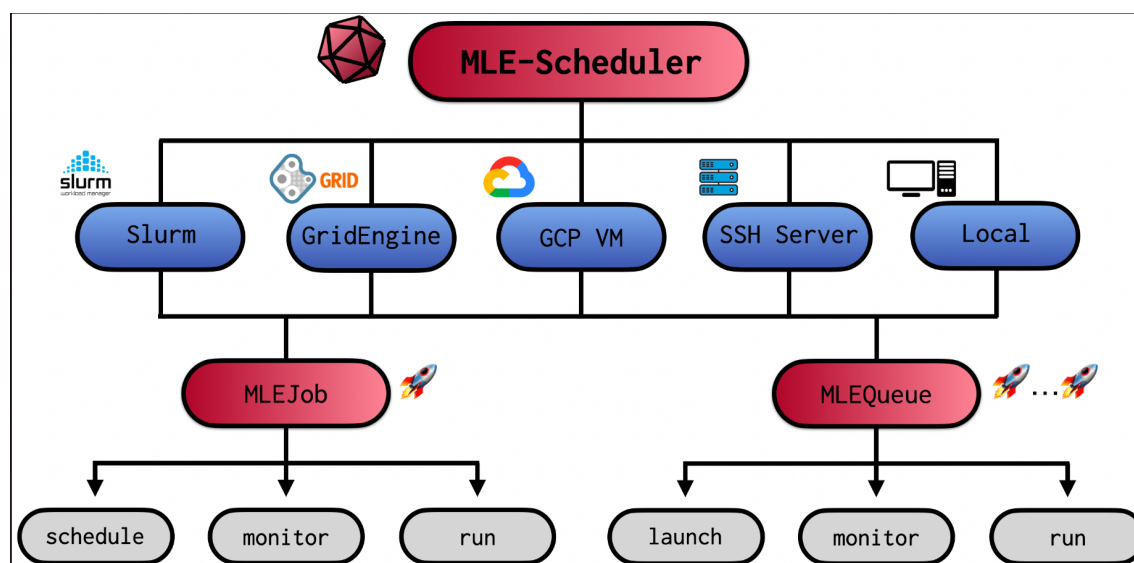


Figura 1.1. MLE-Scheduler

### 1.3.2. Kubernetes

Kubernetes[4] este un sistem open-source de orchestrare a containerelor, proiectat pentru a gestiona și a automatiza sarcinile de implementare, scalare și gestionare a aplicațiilor în medii distribuite. Unul dintre aspectele cheie ale funcționalității Kubernetes este abilitatea sa avansată de a planifica și de a distribui eficient sarcinile și resursele în infrastructurile de calcul distribuite. În această prezentare, vom explora modul în care Kubernetes abordează planificarea joburilor într-un mod academic și cum optimizează distribuția acestora în nodurile de calcul disponibile.

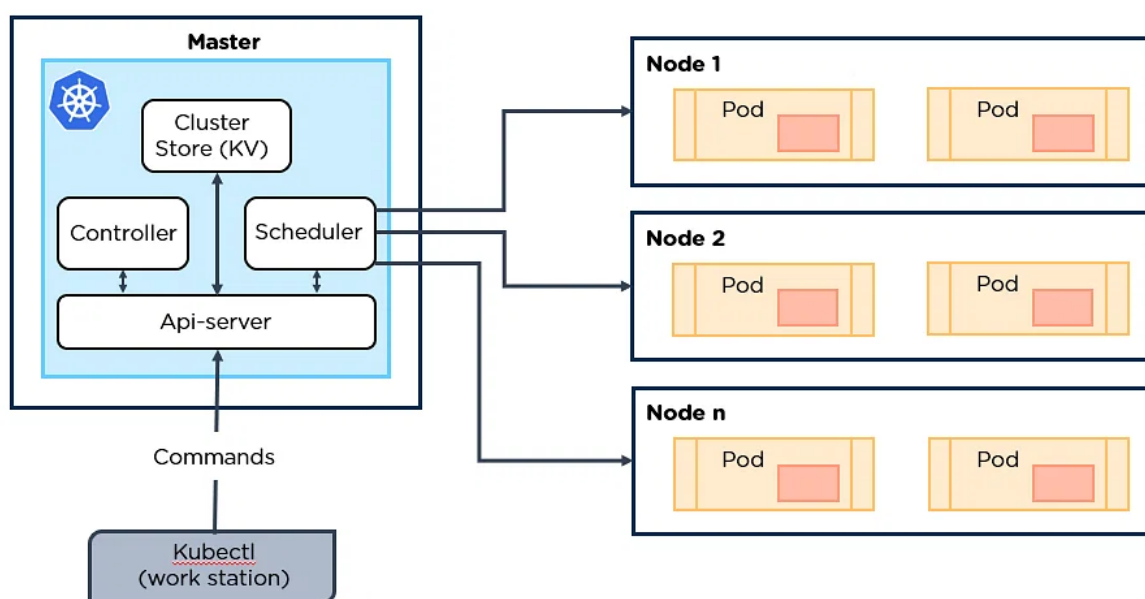


Figura 1.2. Arhitectura Kubernetes<sup>3</sup>

<sup>2</sup>Poza preluată de pe : <https://github.com/mle-infrastructure/mle-scheduler>

<sup>3</sup>Poza preluată de pe : <https://www.simplilearn.com/tutorials/kubernetes-tutorial/kubernetes-architecture>

Arhitectura și concepte cheie:

- Kubernetes utilizează o arhitectură distribuită și scalabilă, formată din clustere de noduri de calcul în care se afla un Master Node ce are rolul de a supraveghea și controla operațiunile de planificare și administrare în timp ce *Worker Nodes* executa sarcinile și furnizează resursele necesare pentru rularea aplicațiilor.

Descrierea joburilor și cerințele de resurse:

- Utilizatorii definesc joburile prin intermediul fișierelor de configurare sau prin intermediul interfeței de programare (API<sup>4</sup>) unde descrierea joburilor include cerințe de resurse precum CPU, memorie, spațiu de stocare și alte dependențe necesare pentru executarea sarcinilor

Planificarea și distribuția joburilor[5]:

- Kubernetes utilizează un planificator avansat pentru a alege nodurile de calcul potrivite pentru a rula fiecare job, planificarea ține cont de cerințele de resurse ale joburilor și de starea curentă a clusterului. Acest planificator analizează disponibilitatea și capacitatea nodurilor asigurând distribuția echilibrată a încărcării pe noduri.

Replanificarea și autoscaling:

- K8s monitorizează constant starea și disponibilitatea nodurilor și a resurselor, dar în cazul în care un nod eșuează sau devine indisponibil, acesta va planifica și redistribui automat sarcinile pe alte noduri disponibile,

### 1.3.3. Slurm

SLURM[6] (Simple Linux Utility for Resource Management) este un sistem open-source de gestionare a resurselor în mediile de calcul de înaltă performanță HPC (High-Performance Computing) se referă la utilizarea de sisteme și tehnologii specializate pentru a realiza calcule și procesări complexe și intensive din punct de vedere computațional. HPC se caracterizează prin abilitatea de a furniza putere de calcul extinsă și performanță superioară, comparativ cu sistemele de calcul tradiționale.

Aplicația utilizează un model de resurse partajate, permițând mai multor utilizatori să aibă acces la resursele clusterului și să ruleze sarcini în mod concurent. Acesta oferă suport pentru gestionarea priorităților, politici de acces și controale fine-grained pentru a asigura utilizarea echitabilă și eficientă a resurselor disponibile.

Utilizatorii pot programa și gestiona sarcinile lor într-un mod eficient, beneficiind de o distribuție echilibrată a resurselor și de un control precis asupra alocării acestora. Sistemul permite, de asemenea, administrarea flexibilă a resurselor, permițând adăugarea sau eliminarea nodurilor de calcul în funcție de nevoile utilizatorilor și de cerințele aplicațiilor.

### 1.3.4. Jenkins

Jenkins[7] este un sistem open-source de automatizare a construirii, testării și distribuiri aplicațiilor software. Este un server bazat pe Java care oferă un cadru flexibil și extensibil pentru automatizarea ciclului de viață al dezvoltării software.

Prin intermediul Jenkins, dezvoltatorii pot defini și configura fluxuri de lucru personalizate, cunoscute sub numele de "joburi", care pot include etape de construire, testare, analiză statică a codului, integrare continuă și distribuire automată. Aceste joburi pot fi programate pentru a rula în mod regulat, pentru a răspunde la evenimente sau pentru a fi declanșate manual.

<sup>4</sup>API (Application Programming Interface) reprezintă o interfață de programare a aplicațiilor, care permite comunicația și interacțiunea între diferite componente software. În mod specific, API-ul definește setul de metode, funcții și protocoale prin intermediul cărora alte aplicații sau servicii pot accesa și utiliza funcționalitățile unei aplicații sau sistem.



#### ***1.4. Elaborarea specificațiilor aplicației***

Scopul aplicației constă în posibilitatea utilizatorului de a executa programe în mod dinamic în cadrul unei arhitecturi distribuite, iar extinderea arhitecturii să fie la fel de dinamică ca și trimiterea oricărui executabil. Astfel, se realizează o aplicație care utilizează un cluster ce poate fi extins pe orizontală într-un mod simplu și flexibil.

Proiectul va fi integrat cu o bază de date, prin intermediul căreia se vor trimite în timp real informații relevante despre joburile executate și condițiile în care acestea s-au încheiat, astfel încât utilizatorul să poată obține o imagine clară asupra executabilelor.

În sinteză, scopul aplicației este de a permite utilizatorului să execute programe în mod dinamic într-o arhitectură distribuită, cu extensibilitate flexibilă. Prin integrarea cu o bază de date, se furnizează informații în timp real despre joburile executate, asigurând o înțelegere clară a rezultatelor. Această soluție eficientă și extensibilă facilitează gestionarea proceselor și obținerea rezultatelor dorite.



## Capitolul 2. Proiectarea aplicației

Proiectarea unei aplicații este o etapă crucială în dezvoltarea produselor software, iar realizarea sa corectă și detaliată din stadiul inițial este decisivă. O proiectare bine făcută încă de la început aduce numeroase beneficii pe parcursul întregului ciclu de dezvoltare și în utilizarea ulterioară a aplicației. Pe lângă beneficiile pe termen scurt, dacă aceasta este bine realizată încă de la început asigură și o scalabilitate mai ușoară a aplicației. Prin anticiparea nevoilor viitoare și adoptarea unei arhitecturi flexibile, se facilitează adăugarea de funcționalități suplimentare sau extinderea capacității aplicației fără a necesita modificări majore sau revizuirea întregii structuri.

În dezvoltarea proiectului, s-a avut în vedere aplicarea cât mai strictă a principiilor OOP[8]<sup>5</sup> și a altor bune practici de programare, precum SOLID[9]<sup>6</sup>, "KISS" (Keep It Simple, Stupid) etc. Așadar, s-a urmărit să se respecte structurile și relațiile obiectelor, promovând o abordare modulară și extensibilă. În egală măsură, s-a căutat să se simplifice soluțiile și să se evite complexitatea inutilă, păstrându-se accentul pe simplitate și claritate în implementare.

### 2.1. Funcționalități propuse

Aplicația presupune trei funcționalități de baza:

- rularea unui job pe calculatorul cel mai potrivit pentru configurația dorită
- trimiterea executabilului pe mașina dorită
- monitorizarea procesului și afișarea informațiilor utile despre acesta
- captarea rezultatului sau al erorilor în caz de eroare

Aplicația utilizează o arhitectură de tipul master-slave, în care rolul de master a fost central în procesul de dezvoltare, iar rolul de slave a fost preluat de calculatoarele subordonate acesteia.

### 2.2. Arhitectura software a aplicației

<sup>5</sup>Programarea orientată pe obiecte (OOP) este o paradigmă de programare care se concentrează pe organizarea datelor și comportamentului în obiecte autonome. OOP se bazează pe încapsulare, moștenire și polimorfism, facilitând reutilizarea și extensibilitatea codului. Prin OOP, se obține un cod modular și eficient, favorizând proiectarea flexibilă a aplicațiilor.

<sup>6</sup>Scopul principal al principiilor SOLID este de a promova o proiectare software modulară, flexibilă și ușor de întreținut. Iată o descriere scurtă a fiecărui principiu SOLID:

Principiul Responsabilității Unice (Single Responsibility Principle - SRP): O clasă ar trebui să aibă o singură responsabilitate și să fie responsabilă de un singur aspect al aplicației. Acest principiu promovează coeziunea și separarea responsabilităților în cadrul sistemului.

Principiul Deschis/Închis (Open/Closed Principle - OCP): O entitate software (clasă, modul sau componentă) ar trebui să fie deschisă pentru extensie, dar închisă pentru modificare. Asta înseamnă că ar trebui să poată fi extinsă prin adăugarea de noi funcționalități, fără a modifica codul existent.

Principiul Înlocuirii Liskov (Liskov Substitution Principle - LSP): Obiectele dintr-o ierarhie de clase ar trebui să poată fi înlocuite cu obiectele din clasele lor derivate, fără a afecta corectitudinea programului. În esență, acest principiu se referă la folosirea corectă a tipurilor de date și la menținerea comportamentului adecvat al obiectelor.

Principiul Separării Interfețelor (Interface Segregation Principle - ISP): Mai degrabă decât să existe o interfață mare și complexă, este mai bine să avem mai multe interfețe mai mici și specializate. Astfel, clienții nu vor fi obligați să depindă de metode pe care nu le folosesc și vor avea acces doar la ceea ce le este necesar.

Principiul Inversării Dependințelor (Dependency Inversion Principle - DIP): Modulele de nivel superior nu ar trebui să depindă direct de modulele de nivel inferior, ci de abstracțiuni. Acest principiu promovează decuplarea și inversarea dependențelor, astfel încât să fie ușor să se facă schimbări și să se obțină flexibilitate în proiectarea software.

Aceste principii SOLID sunt considerate bune practici în dezvoltarea software și contribuie la crearea unui cod modular, flexibil, ușor de întreținut și extensibil.

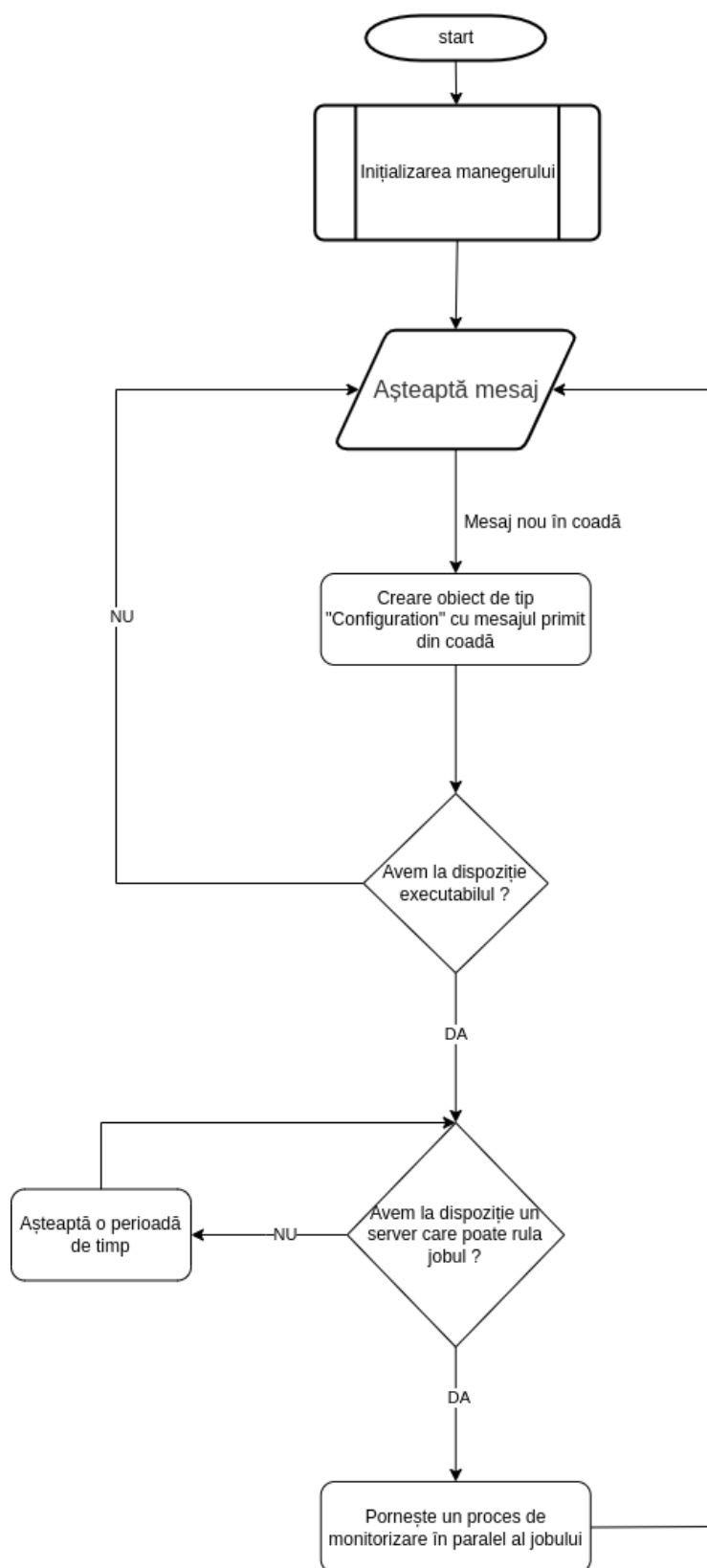


Figura 2.1. Modelul general de funcționare al aplicației

### 2.2.1. Nivelul logic

În cadrul stratului logic, se realizează o analiză a cerințelor funcționale, prin care se extrag acțiunile pe care utilizatorul le poate efectua, precum și serviciile pe care aplicația trebuie să le ofere. Diagrama logică prezentată în figura 2.1 evidențiază acțiunile și fluxul acestora atunci când aplicația este lansată pe mașina gazdă. Este observabil faptul că aplicația oferă o singură funcționalitate, și anume preluarea și prelucrarea joburilor. În vederea implementării acestor funcționalități la nivel de cod, este necesară o clasă care va îndeplini rolul de job și va stoca toate informațiile necesare pentru monitorizarea, rularea și monitorizarea acestuia pe server. De asemenea, un aspect esențial în implementare îl reprezintă serviciul care va aștepta comenzile efectuate de către utilizator.

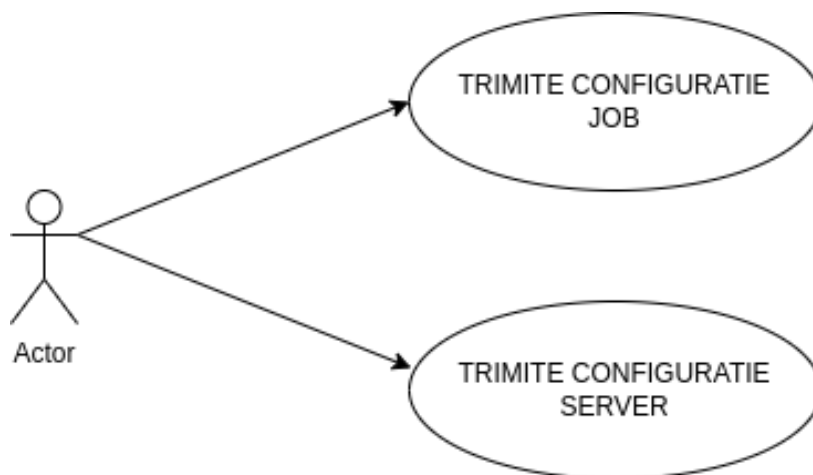


Figura 2.2. Interacțiunea utilizatorului

### 2.2.2. Interacțiunea utilizatorului cu aplicația

Pentru a facilita înțelegerea interacțiunii utilizatorului cu aplicația, în figura 3.2 sunt prezentate toate dialogurile posibile între aceștia. Deoarece aplicația este destinată unui utilizator cu cunoștințe minime în programare, s-a decis să nu aibă o interfață și nici o metodă de autentificare. Până în acest moment, programatorul poate trimite două tipuri de mesaje către aplicație:

- primul tip de mesaj constă în trimiterea configurației serverului și a informațiilor necesare pentru a stabili o conexiune stabilă la server.
- al doilea tip de mesaj constă în trimiterea configurației jobului împreună cu informațiile necesare pentru rularea acestuia.

## 2.3. Resurse software utilizate

Implementarea proiectului s-a bazat pe utilizarea sistemului de operare Linux Ubuntu, având în vedere numeroasele beneficii pe care acesta le oferă. Ubuntu se remarcă prin stabilitate și securitate, asigurând un mediu fiabil și protejat pentru desfășurarea aplicațiilor critice. Interfața prietenoasă și ușurința de utilizare a acestui sistem de operare contribuie la accesibilitatea proiectului, adaptându-se atât utilizatorilor cu experiență, cât și celor mai puțin familiarizați cu sistemele de operare Linux. De obicei, distribuțiile Linux vin cu interpretorul Python preinstalat, reprezentând unul dintre factorii decizionali în alegerea acestui limbaj de programare pentru implementarea proiectului. Pe lângă acestea, am identificat și alte beneficii care au consolidat decizia luată:

- Simplitate și ușurință de învățare: Python este cunoscut pentru sintaxa sa clară și concisă, care face codul ușor de citit și de înțeles. Aceasta face ca Python să fie un limbaj accesibil chiar și pentru programatorii începători și permite o curbă de învățare rapidă și eficientă.

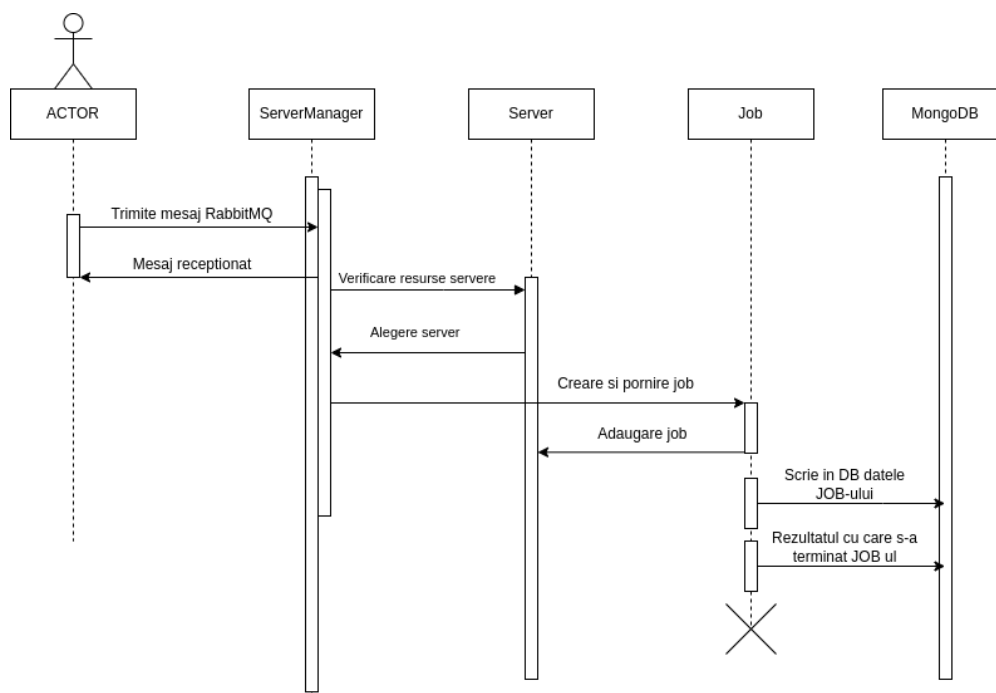


Figura 2.3. Diagrama de secvențe în crearea și rularea unui job

- Numeroase biblioteci
- Portabilitate și interoperabilitate: Python este un limbaj portabil care rulează pe diverse platforme, inclusiv pe sistemele de operare Linux, Windows și macOS. Acest aspect facilitează implementarea și rularea proiectului pe diferite medii de dezvoltare și producție. De asemenea, Python oferă suport pentru interoperabilitate cu alte limbaje de programare, permițând integrarea cu module sau componente scrise în alte limbaje.
- Comunitate activă și suport extins.

După stabilirea limbajului de programare pentru partea de comunicare cu utilizatorul, s-a luat decizia de a utiliza cozi de mesaje. Această alegere a fost determinată de necesitatea de a face față unui volum mare de cereri și de a scala aplicația pe orizontală. Astfel, prin trimiterea serverului dorit în a fi utilizat către un cluster de calculatoare deja existent, se obține o soluție eficientă de extindere a capacității de calcul.

Pentru gestionarea cozilor de mesaje, am avut opțiunea de a utiliza două aplicații, RabbitMQ[10] sau Apache Kafka[11]. Având în vedere numărul redus de cozi și nivelul scăzut de complexitate, am decis să utilizăm RabbitMQ, deoarece este o soluție mai ușor de utilizat și configurat. Alegerea acestei platforme ne-a oferit flexibilitatea necesară și a permis implementarea eficientă a sistemului de cozi de mesaje în cadrul aplicației. O altă notabilă diferență a fost fiabilitate și durabilitatea, deoarece aceasta oferă un mecanism solid de stocare a mesajelor și asigură livrarea acestora în mod fiabil chiar și în cazul întreruperilor de rețea sau eșecurilor hardware, acest aspect fiind util într-o aplicație critică sau într-un mediu în care integritatea datelor este crucială.

În vederea furnizării utilizatorului unui instrument de monitorizare în timp real, s-a considerat necesară adăugarea unei baze de date pe lângă logul aplicației. Această bază de date va fi utilizată pentru a stoca toate job-urile care au fost rulate sau sunt în desfășurare, împreună cu informații suplimentare relevante, care facilitează identificarea acestora într-un mod mai eficient. Astfel, utilizatorul va avea posibilitatea de a accesa rapid istoricul job-urilor și de a analiza datele relevante, permițând o monitorizare și o evaluare mai profundă a executabilului. Prin adăugarea

<sup>7</sup>Poza preluată de pe: <https://ilegra.com/blog/what-is-rabbitmq/>

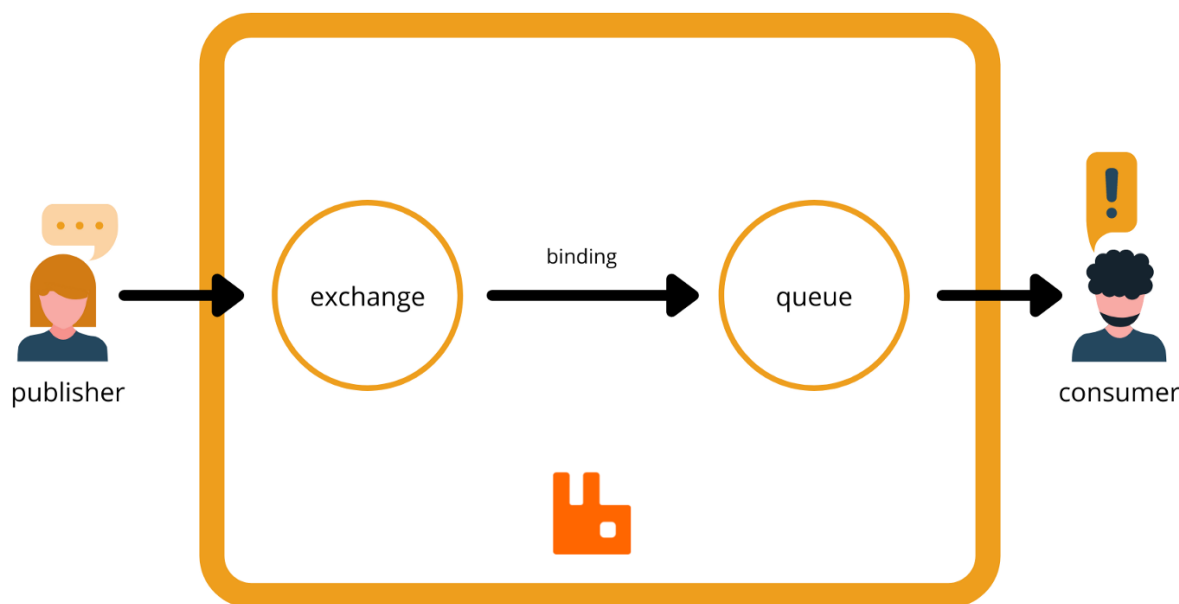


Figura 2.4. Modul în care funcționează RabbitMQ<sup>7</sup>

acestei baze de date, se asigură o abordare mai completă și riguroasă în monitorizarea în timp real a datelor și a job-urilor.

Similar cu alegerea pentru sistemul de cozi de mesaje, am avut opțiunea de a alege între două tipuri de baze de date: SQL și NoSQL. Având în vedere natura și structura viitoarelor informații ce urmau să fie stocate, am decis să utilizăm o bază de date NoSQL, în special MongoDB. Alegând baza de date MongoDB am obținut o flexibilitate în schema datelor, adică stocarea datelor se poate face într-un mod flexibil, fără a fi necesară definirea unui model de date rigid în avans, cum aveam în cazul datelor în SQL. Acest aspect fiind util în cazul în care vom avea schimbări sau adăugări frecvente în structura datelor.

Pentru a putea simula un mediu de dezvoltare similar cu cel din producție, s-a ajuns la concluzia că utilizarea Docker[12] și Docker-Compose[13] este cea mai bună opțiune. Această decizie se bazează pe avantajele oferite de aceste tehnologii. Prin utilizarea containerelor Docker, atât RabbitMQ, cât și serverul MongoDB pot fi încapsulate în containere izolate, asigurând astfel o izolare adecvată și portabilitatea aplicației. Docker-compose permite definirea și gestionarea întregului mediu de dezvoltare printr-un singur fișier de configurare, asigurând reproductibilitatea mediului și coerența între diferitele mașini sau echipe de dezvoltare. Aceasta simplifică procesul de gestionare a resurselor și permite scalabilitatea orizontală prin definirea și configurarea facilă a mai multor instanțe ale aceluiași serviciu. Pe lângă aceste avantaje, utilizarea Docker și Docker-compose aduce flexibilitate prin disponibilitatea unui ecosistem bogat de imagini predefinite și simplifică ciclul de dezvoltare prin crearea unui mediu de dezvoltare consistent și reproductibil.

În ceea ce privește comunicarea dintre aplicație și workeri, cea mai potrivită opțiune a fost folosirea protocolului de comunicare SSH[14]

SSH (Secure Shell) este un protocol criptografic de rețea ce furnizează o metodă securizată de acces și gestionare a sistemelor remote prin intermediul unei rețele nesecurizate. A fost conceput pentru a înlocui protocolul mai puțin securizat Telnet, oferind mecanisme puternice de criptare

și autentificare.

Unul dintre avantajele cheie ale SSH constă în caracteristicile sale de securitate robuste. Protocolul utilizează diverse algoritmi de criptare, precum RSA, DSA și ECC, pentru a asigura confidențialitatea și integritatea datelor transmise între client și server. Prin criptarea comunicării, SSH previne interceptarea și manipularea de către entități malefice.

SSH oferă, de asemenea, metode sigure de autentificare, inclusiv autentificarea bazată pe parole și autentificarea cu chei publice. Autentificarea cu chei publice este în mod deosebit avantajoasă, deoarece elimină necesitatea transmiterii parolelor prin rețea, reducând astfel riscul interceptării parolelor sau al atacurilor de forță brută. Prin autentificarea cu chei publice, utilizatorii generează un set de chei constând într-o cheie privată (păstrată în siguranță pe client) și o cheie publică (încărcată pe server). Serverul verifică autenticitatea clientului solicitându-i să demonstreze posesia cheii private corespunzătoare.

În plus, SSH suportă redirecționarea de porturi și tunelarea, permițând transferul securizat de date între sisteme. Această funcționalitate permite utilizatorilor să stabilească tuneluri criptate pentru servicii precum desktop remote, transfer de fișiere sau conexiuni la baze de date, îmbunătățind astfel securitatea generală a comunicațiilor în rețea.

Un alt avantaj al SSH este independența sa de platformă. Protocolul este disponibil pe diferite sisteme de operare, inclusiv sistemele de tip Unix (de exemplu, Linux, macOS) și Windows. Această compatibilitate cross-platform face din SSH un instrument versatil pentru administrarea securizată remote și transferul de fișiere în diferite medii.



Figura 2.5. Conexiune SSH<sup>8</sup>

După o cercetare extinsă care a inclus și problema comunicării directe cu serverele prin intermediul SSH pentru interogarea sistemului de operare pentru verificarea stadiilor fiecărui proces, s-a concluzionat că cea mai bună soluție pentru monitorizarea joburilor care rulează pe mașinile subordonate este utilizarea Supervisor<sup>d</sup>[15].

Supervisor<sup>d</sup>[15], o aplicație open-source utilizată pe scară largă în industrie, oferă o abordare simplă și eficientă pentru gestionarea și monitorizarea proceselor în mediul Unix. Cu funcționalitatea sa de management centralizat, repornire automată, opțiuni flexibile de configurare și facilități avansate de monitorizare, Supervisor<sup>d</sup> aduce beneficii precum stabilitate îmbunătățită, continuitate a serviciilor și supervizare eficientă a proceselor. De cele mai multe ori în industrie aplicația este folosită pentru monitorizarea proceselor din containerele Docker.

Configurația Supervisor<sup>d</sup> prezentată în Figura 2.6 se regăsește pe fiecare mașină. Este evidentiat faptul că aceasta utilizează portul 9001 pentru funcționare, iar un alt aspect deosebit de semnificativ este reprezentat de câmpul numit "[rpcinterface:supervisor]", care specifică API-ul prin intermediul căruia se va realiza comunicarea cu aplicația.

<sup>8</sup>Poza preluată de pe : <https://www.ssh.com/academy/ssh/protocol>



```

server@server:/etc/supervisor$ cat supervisord.conf
; supervisor config file

[unix_http_server]
file=/var/run/supervisor.sock ; (the path to the socket file)
chmod=0700 ; socket file mode (default 0700)

[inet_http_server]
port=*:9001

[rpcinterface:supervisor]
supervisor.rpcinterface_factory=supervisor.rpcinterface:make_main_rpcinterface

[supervisord]
logfile=/var/log/supervisor/supervisord.log ; (main log file;default $CWD/supervisord.log)
pidfile=/var/run/supervisord.pid ; (supervisord pidfile;default supervisord.pid)
childlogdir=/var/log/supervisor ; ('AUTO' child log dir, default $TEMP)

; the below section must remain in the config file for RPC
; (supervisorctl/web interface) to work, additional interfaces may be
; added by defining them in separate rpcinterface: sections
[rpcinterface:supervisor]
supervisor.rpcinterface_factory = supervisor.rpcinterface:make_main_rpcinterface

[supervisorctl]
serverurl=unix:///var/run/supervisor.sock ; use a unix:// URL  for a unix socket

; The [include] section can just contain the "files" setting. This
; setting can list multiple files (separated by whitespace or
; newlines). It can also contain wildcards. The filenames are
; interpreted as relative to this file. Included files *cannot*
; include files themselves.

[include]
files = /etc/supervisor/conf.d/*.conf

```

Figura 2.6. Configurație supervisord

VirtualBox[16], o aplicație adițională, a fost utilizată pentru a recrea un mediu de testare și a genera un număr adecvat de mașini virtuale pentru a verifica funcționalitatea aplicației. Această soluție aduce cu sine o serie de beneficii semnificative. Cel mai important beneficiu este ca facilitează realizarea de teste repetabile și consistente, datorită abilității sale de a clona rapid mașinile virtuale. În plus, aceasta oferă posibilitatea de a testa într-o varietate de sisteme de operare și configurații hardware fără a necesita achiziția sau accesul la echipamente fizice specifice. Nu în ultimul rând, VirtualBox contribuie la economia de resurse prin utilizarea eficientă a capacității de calcul existente, diminuând astfel costurile asociate cu testarea.

### 2.3.1. Interconectarea componentelor principale

Componentele hardware reprezintă elementele fizice fundamentale necesare pentru a susține și a asigura funcționalitatea unei aplicații software. Aceste componente includ servere, stații de lucru, echipamente de stocare, rețele și alte dispozitive esențiale. Ele ocupă un rol crucial datorită următoarelor aspecte. În primul rând, componentele hardware influențează direct performanța aplicației, oferind resurse necesare pentru o execuție rapidă și fluentă. În al doilea rând, ele asigură scalabilitatea aplicației, permițând adaptarea la cerințele în creștere și manipularea volumului sporit de trafic sau de date. În plus, componente hardware fiabile și robuste contribuie la asigurarea unei funcționări constante și la minimizarea erorilor, rezultând o experiență satisfăcătoare pentru utilizatori. În ceea ce privește securitatea, unele componente hardware pot integra funcționalități de protecție împotriva amenințărilor cibernetice.

În ceea ce privește testarea proiectului, aceasta va fi efectuată prin intermediul a două mașini care vor simula arhitectura distribuită. Această abordare permite verificarea funcționalității și performanței sistemului în condiții similare cu cele din mediile de producție.

Abordăm cazul în care avem doar două mașini virtuale:

- mașina nr. 1 va avea 1 CPU și 2 GB ram
- mașina nr. 2 va avea 2 CPU și 4 GB ram (diferențele dintre cele două mașini sunt proiectate

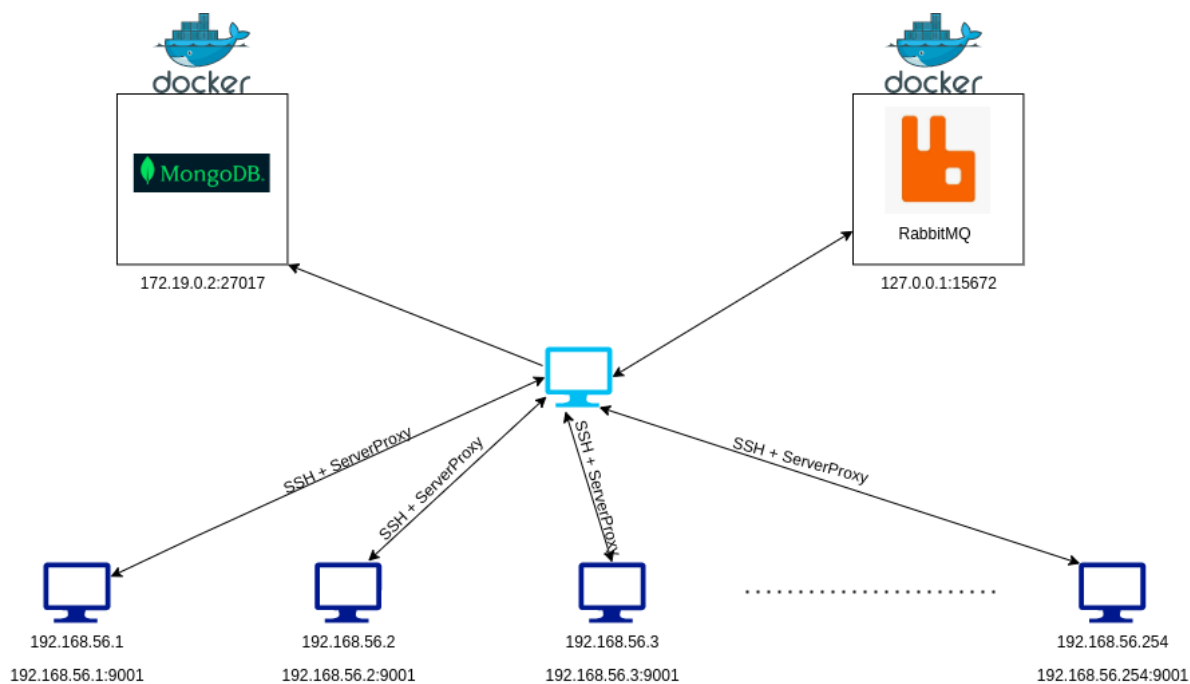


Figura 2.7. Schema aplicației

pentru a permite redirectionarea sarcinilor către o mașină mult mai eficientă în realizarea acestora)

- localhost-ul va avea nevoie de 1 CPU și 4 GB ram
- spațiul de stocare pentru fiecare mașină este de 10 GB și 25 GB pentru localhost

În cazul în care se dorește o testare mai amănunțită a aplicației, numărul de mașini virtuale poate fi crescut. Cu toate acestea, pentru a efectua o testare minimă a proiectului, resursele menționate sunt suficiente.

## 2.4. Informații generale despre implementarea aleasă

### 2.4.1. Limitele funcționale

În stadiul actual al implementării și având în vedere resursele disponibile pentru testarea aplicației, aceasta poate rula pe orice distribuție de Linux sau Windows. Cu toate acestea, este important de menționat că cele două servere disponibile nu pot rula pe sistemul de operare Windows, întrucât acestea trebuie să ruleze într-un mediu în care Supervisorul este implementat.

### 2.4.2. Analiza SWOT a aplicației dezvoltate

Puncte tari:

- aplicația poate să ruleze pe orice sistem de operare
- foarte eficient dacă se dorește rularea joburilor într-o ordine FIFO<sup>9</sup>
- oferă informații în timp real

Puncte slabe:

<sup>9</sup>FIFO (First-In, First-Out) este un acronim folosit în domeniul gestiunii stocurilor și al gestionării datelor, care se referă la o metodă de ordonare și accesare a elementelor în ordinea în care au fost adăugate.

- în cazul în care unul dintre servere este oprit se pierd joburile care rulează pe el până la revenirea sa
- serverele în rolul de "slave" pot opera doar pe Linux
- un algoritm neperformant de luarea deciziilor în ceea ce privește alegerea gazdei pentru jobul ce urmează să ruleze
- neeficient dacă se dorește rularea joburilor într-o altă ordine în afară de FIFO

Oportunități:

- Piața serverelor este în continuă dezvoltare
- Design simplu ce poate fi schimbat pentru îndeplinirea sarcinilor dorite
- Aplicațiile computaționale, de cele mai multe ori, rulează pe servere proprii

Amenințări:

- Produse similare cu notorietate pe piață
- Opțiunea de a folosi cloud



## Capitolul 3. Implementarea aplicației

### 3.1. Comunicarea utilizatorului cu aplicația

The screenshot shows the RabbitMQ management UI. At the top, it indicates the version is 3.6.16 running on Erlang. The 'Queues' tab is selected, showing a list of queues. Below the table, there is a pagination section showing 'Page 1 of 1' and 'Displaying 2 items'. The table itself has columns for 'Name', 'Features', 'State', 'Ready', 'Unacked', 'Total', and 'Message rates' (incoming, deliver/get, ack). Two queues are listed: 'JOBS' and 'SERVERS', both with a state of 'idle' and 0 messages in each category.

Overview			Messages			Message rates		
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
JOBS		idle	0	0	0			
SERVERS		idle	0	0	0			

Figura 3.1. Cozi în RabbitMQ

Asa cum a fost prezentat și în capitolul 2 comunicarea dintre utilizator se va face prin intermediul a doua cozi de mesaje în RabbitMQ, una fiind folosită pentru procesarea job-urilor (JOBS), iar cealaltă pentru a putea scala pe orizontală aplicația (SERVERS).

Un exemplu de mesaj ce se afla în coada JOBS:

```
[
  {
    "filename": "program_test.py",
    "autostart": false,
    "autorestart": false,
    "command": "python3"
  }
]
```

Un exemplu de mesaj ce se afla în coada SERVERS:

```
[
  {
    "ip": "192.168.56.105",
    "user": "server",
    "password": "server"
  }
]
```

La nivel de cod, mesajele sunt preluate prin intermediul unui consumator implementat cu ajutorul bibliotecii Python pika. Pika este o bibliotecă populară pentru gestionarea comunicațiilor cu sisteme de mesagerie bazate pe protocoale de tip AMQP (Advanced Message Queuing Pro-

tocol), cum ar fi RabbitMQ. Prin intermediul funcționalităților oferite de pika<sup>10</sup>, se pot stabili conexiuni, să se declare cozi, să se publice și să se consume mesaje într-un mod avantajos și eficient. Biblioteca facilitează interacțiunea cu brokerii de mesagerie, permițând dezvoltatorilor să implementeze sisteme de mesagerie flexibile și scalabile în aplicațiile scrise în Python.

```

1  def start_rabbitmq_listener(servers_manager: ServerManager):
2      connection =
3          → pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
4      channel = connection.channel()
5      channel.queue_declare(queue='JOBS')
6      channel.queue_declare(queue='SERVERS')
7
8      def callback(ch, method, properties, body):
9          if method.routing_key == "JOBS":
10             jsons_configuration = json.loads(body)
11             jsons_number = len(jsons_configuration)
12             dl.logger.info(f"JOBS FROM RABBITMQ --> {jsons_number}
13                 → configuration : {jsons_configuration}")
14             if jsons_number != 0:
15                 if jsons_number > 1:
16                     for json_conf in jsons_configuration:
17                         servers_manager.start_job(json_conf)
18                 else:
19                     servers_manager.start_job(jsons_configuration[0])
20             if method.routing_key == "SERVERS":
21                 jsons_configuration = json.loads(body)
22                 jsons_number = len(jsons_configuration)
23                 dl.logger.info(f"SERVERS FROM RABBITMQ --> {jsons_number}
24                     → SERVERS : {jsons_configuration}")
25                 if jsons_number != 0:
26                     if jsons_number > 1:
27                         for json_conf in jsons_configuration:
28                             servers_manager.add_server(json_conf)
29                     else:
30                         → servers_manager.add_server(jsons_configuration[0])
31
32             channel.basic_consume(queue='JOBS', on_message_callback=callback,
33                 → auto_ack=True)
34             channel.basic_consume(queue='SERVERS',
35                 → on_message_callback=callback, auto_ack=True)
36             print(' [*] Waiting for messages. To exit press CTRL+C')
37             channel.start_consuming()

```

Listing 3.1. Implementarea consumatorului de mesaje

## 3.2. Descrierea generală a implementării

### 3.2.1. Structura aplicației

Aplicația a fost structurată în multiple pachete (figura 3.2), unele dintre ele fiind utilizate în scopul testării aplicației, în timp ce altele sunt folosite în implementarea propriu-zisă a aplicației.

<sup>10</sup><https://pika.readthedocs.io/en/stable/>

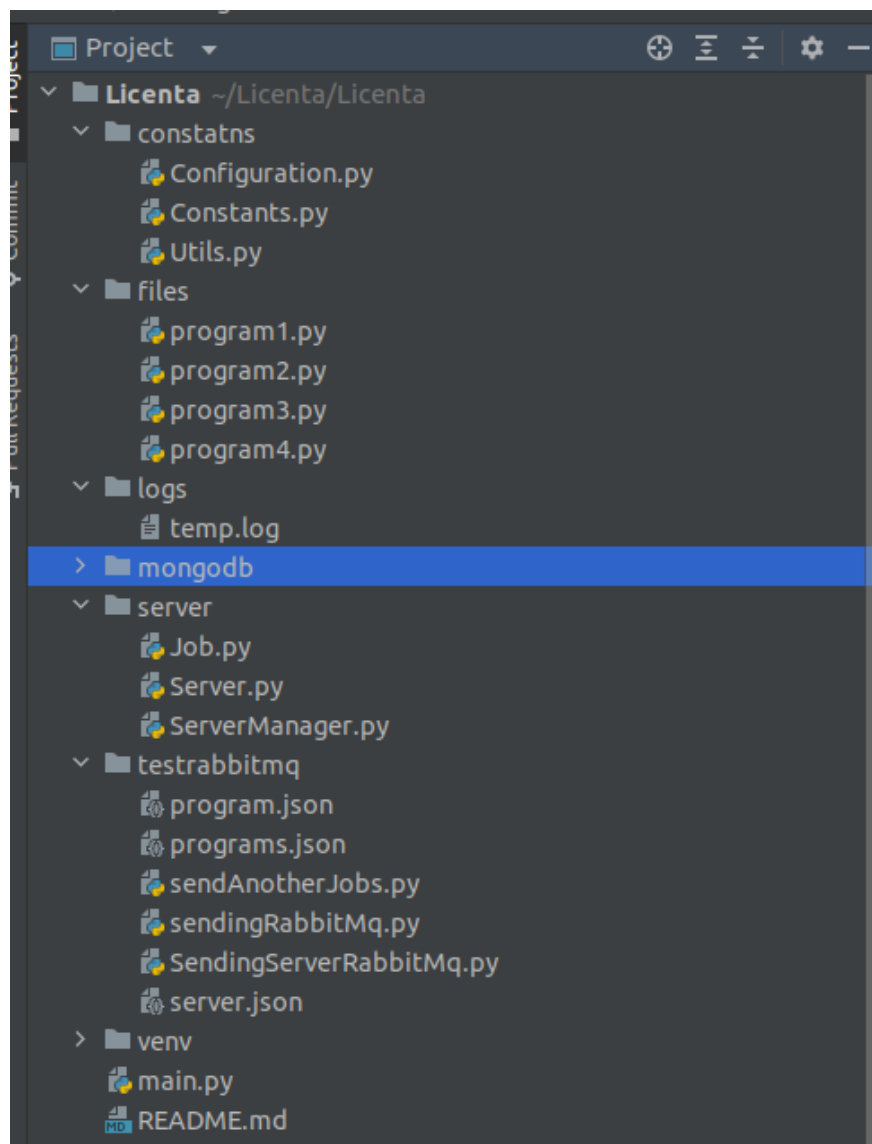


Figura 3.2. Pachetele și clasele proiectului

Pachetele implicate în implementarea proiectului sunt "server", "mongodb" și "constants". Pachetul "files" a fost dedicat stocării executabilelor aflate pe sistemul de operare. În ceea ce privește partea de testare, a fost creat pachetul "testrabbitmq" care conține două scripturi special concepute pentru a trimite mesaje în cozile de mesaje. Pentru gestionarea jurnalelor (logs), am optat să plasăm fișierul .log în directorul "logs".

La pornirea aplicației se vor crea două instanțe ale unor clase extrem de importante pentru funcționarea corespunzătoare a acesteia. Prima clasă se ocupă de stabilirea și gestionarea conexiunii cu baza de date utilizată, asigurând o comunicare corectă și eficientă cu sistemul de stocare. A doua clasă are responsabilitatea de a gestiona serverele aflate în subordine, având ca sarcină monitorizarea și administrarea acestora pentru a asigura o funcționare optimă a aplicației. Pentru a oferi un cod de calitate în ceea ce privește clasa care se ocupă de gestionarea serverelor, s-a luat decizia de a implementa pattern-ul de proiectare Singleton<sup>11</sup>. Astfel, se asigură că există o singură instanță a clasei, care poate fi accesată global în aplicație, evitând astfel posibile probleme de sincronizare sau gestionare inadecvată a resurselor.

### 3.2.2. Gestionarea resurselor alocate

Prin "resursa alocată" ne referim la puterea de calcul pe care utilizatorul o transmite prin intermediul cozii de mesaje numită SERVER. Mesajele din această coadă sunt utilizate pentru a construi obiecte de tip Server. Procesul de construire a obiectului are loc în interiorul managerului de server, iar odată ce obiectul este creat, acesta este adăugat într-o listă de elemente unice de același tip. Am decis să transmit direct JSON-ul<sup>12</sup>, deoarece în Python este interpretat ca o structură de date numită dicționar.

#### 3.2.2.1. Obiectul de tip Server

Așa cum am menționat anterior, în cadrul codului proiectului, logica este centrată în jurul obiectului numit Server. După preluarea datelor din coada de mesaje, se va crea un obiect care are trei câmpuri principale: conexiunea SSH, conexiunea realizată prin intermediul Supervisor, utilizând serverul proxy, și lista de joburi atribuite serverului, astfel avem o imagine clară a proceselor și a conexiunii făcute între aplicație și server.

```

1  def add_server(self, server_configuration):
2      server = Server(server_configuration["ip"],
3                      ↪ server_configuration["user"],
4                      ↪ server_configuration["password"])
5      self.servers.append(server)
6      # make here the list elements unique but use list for future
7      ↪ tasks
8      self.servers = list(tuple(self.servers))
9      dl.logger.info(f"Added new server {server.ip_address} in manager
10     ↪ ")

```

Listing 3.2. Stocarea mesajului primit din json

<sup>11</sup>Singleton este un pattern de proiectare în programare orientată pe obiecte care permite crearea unei singure instanțe a unei clase și furnizarea unui punct global de acces la acea instanță în cadrul unei aplicații. Scopul principal al Singleton-ului este de a restricționa instanțierea repetată a unei clase și de a asigura că există o singură instanță care poate fi accesată din întreaga aplicație.

<sup>12</sup>JSON (JavaScript Object Notation) este un format de text ușor de citit și de generat, utilizat pentru transmiterea datelor structurate între un client și un server. Acesta este bazat pe sintaxa obiectelor și listelor din limbajul de programare JavaScript, dar poate fi folosit într-o varietate de limbaje de programare. JSON este folosit adesea în aplicațiile web pentru a transmite și stoca date într-un format ușor de interpretat și de manipulat.



### 3.2.3. Stocarea datelor

Dupa ce datele au fost preluate acestea vor fi stocate în aplicație prin intermediul unei clase pentru a le putea accesa mult mai ușor pe parcursul întregului flux al aplicației.

```

1  class Configuration:
2      def __init__(self, json_file):
3          name = json_file["filename"]
4          try:
5              print(os.getcwd())
6              self.file_dimension = os.path.getsize(path_executable +
              ↪ name)
7          except FileNotFoundError:
8              raise FileNotFoundError(f"File {path_executable}{name}
              ↪ not found")
9          self.name, self.extension = os.path.splitext(name)
10         self.autostart = json_file["autostart"]
11         self.autorestart = json_file["autorestart"]
12         self.command = json_file["command"]

```

Listing 3.3. Stocarea mesajului primit din json

În vederea asigurării primirii unui job valid, este necesar ca fișierul executabil corespunzător să existe în partiția pe care funcționează aplicația. Aceasta reprezintă o măsură de verificare pentru a confirma că resursele necesare pentru executarea jobului sunt disponibile și accesibile. Prin verificarea prezenței fișierului executabil în locația specificată, se asigură că aplicația poate executa în mod corespunzător sarcina aferentă și poate continua fluxul de lucru fără probleme sau erori.

Pentru a asigura o conexiune cât mai flexibilă cu serverul și pentru a permite accesul la multiple resurse, am optat pentru crearea unei legături generice cu baza de date folosind biblioteca pymongo<sup>13</sup>. Astfel, utilizatorul are posibilitatea de a accesa colecția dorită. Această clasă operează în strânsă legătură cu clasa "Collection-DB", unde am definit în prealabil operațiile de actualizare, ștergere și inserare pentru joburi. Abordarea prezentată facilitează accesul la metodele menționate, asigurând astfel un design coerent și modular.

```

1  def create_job(self, ip_server, job_name, auto_restart, status, pid):
2      self.collection.insert_one(
3          {
4              "ip": ip_server,
5              "job_name": job_name,
6              "autor_restart": auto_restart,
7              "status": status,
8              "pid": [pid]
9          }
10     )
11
12  def update_job(self, ip, job_name, status, pid):
13      document = self.collection.find_one(
14          {"ip": ip, "job_name": job_name})
15
16      list_pids = document["pid"]
17      list_pids.append(pid)

```

<sup>13</sup><https://pymongo.readthedocs.io/en/stable/>

```

18
19     self.collection.update_one(
20         {"ip": ip, "job_name": job_name},
21         {"$set": {"status": status, "pid": list_pids}}
22     )

```

Listing 3.4. Metodele principale din Collection-DB

```

1  class MongoDB:
2      def __init__(self, database: str, connection_retries=3):
3          self.connection =
4              ↳ MongoClient("mongodb://root:example@172.19.0.3:27017")
5          self.database = self.connection[database]
6          connection_ok_flag = True if
7              ↳ self.database.command('ping')['ok'] == 1.0 else False
8
9          while connection_retries > 0 and connection_ok_flag is False:
10             self.connection =
11                 ↳ MongoClient("mongodb://root:example@172.19.0.3:27017")
12             connection_ok_flag = True if
13                 ↳ self.database.command('ping')['ok'] == 1.0 else False
14             connection_retries -= 1
15
16         print("Connection successful" if connection_ok_flag else
17             ↳ "Failed to connect")
18
19     def get_collection(self, collection_name: str) -> Collection:
20         return self.database[collection_name]

```

Listing 3.5. Clasa ce realizează conexiunea cu MongoDB

### 3.2.4. Receptionarea unui job

După ce aplicația a fost configurată, ceea ce înseamnă că a stabilit conexiunea cu baza de date și a primit cel puțin un server pe care poate executa sarcini, ea poate primi mesaje pentru a rula joburile.

Fluxul aplicației se desfășoară în următorul mod: consumatorul de mesaje preia mesajul primit din coada denumită JOBS, iar după ce obiectul de tip Configuration a fost instanțiat, acesta verifică existența fișierului executabil pe server și selectează cu grijă cel mai adecvat calculator pentru a-l utiliza în execuția acestuia.

```

1  def find_best_server_for_job(self, configuration) -> Server || None:
2      # store the list of good servers.
3      good_servers = self.servers
4      # TODO: check this part of free core-space
5      for server in self.servers:
6          if server.get_free_space() < configuration.file_dimension:
7              good_servers.remove(server)
8          if server.get_free_ram() < configuration.memory:
9              good_servers.remove(server)
10         if server.get_free_core() < 50.0:
11             good_servers.remove(server)
12     if good_servers:

```

```

13         return good_servers[0]
14     else:
15         time.sleep(15)
16     return None

```

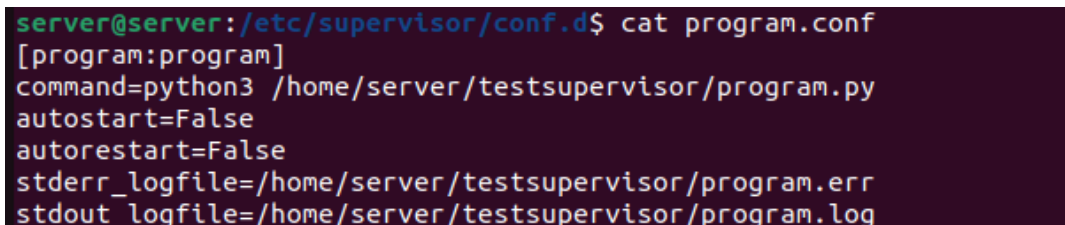
Listing 3.6. Algoritm pentru alegerea celui mai potrivit server

Algoritm selectat este relativ simplu, acesta verifică serverele pentru a se asigura că au suficient spațiu disponibil, memorie adecvată pentru a rula executabilul și un anumit procent liber din proces. În scopul simulării unui scenariu în care unul dintre servere nu ar fi potrivit pentru a executa jobul, s-a luat decizia de a utiliza ca criteriu faptul că serverul respectiv trebuie să aibă cel puțin 50% din capacitatea de procesor disponibilă.

### 3.2.5. Mod de utilizare Supervisor

După selectarea serverului cel mai potrivit, următorul pas constă în construirea jobului cu configurația cerută de utilizator. Așa cum s-a menționat și în capitolul 2, executabilele vor fi rulate în mediul furnizat de Supervisor. Pentru a putea obține o mai bună înțelegere a funcționării aplicației, se vor explica pașii necesari pentru a rula un program simplu:

- În primul rând, este necesară configurarea Supervisor, care se realizează prin modificarea fișierului găsit în directorul `/etc/supervisor/supervisor.conf`. În cadrul proiectului, nu avem nevoie de setări mult prea complexe.
- După ce Supervisor a fost setat și rulează După așteptările dorite, următorul pas constă în crearea configurației pentru executabilele pe care dorim să le rulăm. Până în prezent, cu implementarea actuală a proiectului, este posibil să setăm comanda prin care programul va fi rulat (de exemplu, `"python3"`), precum și doi indicatori (flag-uri): unul pentru a porni automat aplicația după configurare și unul pentru a reporni aplicația în caz de eroare. Pe lângă acestea, vor exista și fișierele de jurnal (log-uri) care vor fi utilizate pentru a înregistra rezultatele în caz de eroare sau rezultatele obținute în timpul rulării executabilelor (trebuie să se țină cont de faptul că fișierele de jurnal trebuie să existe înainte ca programul să fie rulat, dacă nu avem fișierele și aplicația este pornită va rezulta o eroare). Este important de menționat că fișierele de configurare ale programelor vor fi localizate în directorul `/etc/supervisor/conf.d`. Acest lucru este predefinit de Supervisor și, deși poate fi modificat, ar conduce la o scădere a securității, deoarece orice utilizator ar putea efectua modificări în aceste fișiere dacă acestea ar fi plasate într-un loc nepotrivit. Figura 3.3 este un exemplu de fișier `.conf` valid.



```

server@server:/etc/supervisor/conf.d$ cat program.conf
[program:program]
command=python3 /home/server/testsupervisor/program.py
autostart=False
autorestart=False
stderr_logfile=/home/server/testsupervisor/program.err
stdout_logfile=/home/server/testsupervisor/program.log

```

Figura 3.3. Fișier configurație executabil

- După ce fișierele de jurnal și cel cu configurarea au fost create cu succes, următorul pas constă în a anunța Supervisor că există noi configurații în fișierul destinat lor. Supervisor va relua citirea și va adăuga noile joburi pentru a fi pregătite pentru rulare.
- Pentru rularea aplicațiilor Supervisor oferă o interfață numită `supervisorctl`, care ne permite să obținem informații despre starea joburilor. Astfel, putem verifica dacă joburile pot fi

ruled (gata de rulare), dacă au fost deja rulate sau dacă nu au fost încă rulate. Supervisorul ne furnizează astfel un mecanism prin care putem monitoriza și gestiona starea și progresul joburilor în cadrul aplicației. Utilizând `supervisorctl`, putem accesa informații relevante referitoare la statusul joburilor, precum și alte detalii necesare pentru a asigura funcționarea corectă și eficientă a acestora.

```
server@server:/etc/supervisor/conf.d$ sudo supervisorctl
[sudo] password for server:
program                STOPPED    Not started
program_test           STOPPED    Not started
supervisor> help

default commands (type help <topic>):
=====
add      exit      open  reload  restart  start  tail
avail    fg          pid   remove  shutdown status update
clear    maintail  quit  reread  signal   stop   version
supervisor>
```

Figura 3.4. Interfața `supervisorctl`

### 3.2.6. Implementarea Supervisorul în proiect

Așa cum a fost prezentat în capitolul anterior, Supervisorul oferă un API care facilitează comunicarea cu aplicația. Aceasta se realizează prin intermediul protocolului XML-RPC.

XML-RPC este un protocol de comunicare bazat pe XML (Extensible Markup Language) și RPC (Remote Procedure Call). Protocolul permite aplicațiilor să comunice între ele prin transmiterea de mesaje XML structurate.

În cadrul Supervisorul, API-ul XML-RPC permite interacțiunea cu procesele și joburile gestionate de acesta. Prin intermediul apelurilor XML-RPC, se pot realiza operații precum pornirea sau oprirea proceselor, obținerea stării joburilor sau gestionarea configurațiilor.

Pentru a utiliza API-ul XML-RPC, este necesară configurarea și activarea acestuia în fișierul de configurare al Supervisorul. Apoi, se poate realiza comunicarea cu aplicația folosind librării sau clienți XML-RPC disponibili în diverse limbaje de programare.

La nivel de cod interacțiunea cu Supervisorul s-a realizat prin intermediul serverului proxy<sup>14</sup> din pachetul `xmlrpc`<sup>15</sup>. Pentru a simplifica procesul de conectare prin intermediul API-ului, am decis ca fișierul de configurare al Supervisorul să permită accesul tuturor. Astfel, conexiunea în Python va fi realizată doar prin utilizarea adresei IP și a portului corespunzător. Această abordare facilitează comunicarea cu Supervisorul și reduce posibilele obstacole în procesul de conectare.

Metodele principale care au fost folosite din API-ul citat sunt:

- "`getProcessInfo()`" a fost utilizată pentru a obține informații despre statusul unui proces, cum ar fi numele, portul și starea în care se află. Această metodă este utilă pentru a verifica dacă procesul s-a întrerupt sau nu, fapt ce se poate deduce prin comparația PID-ului primit în prezent cu PID-ul citit anterior. Dacă PID-urile sunt diferite, se poate presupune că procesul a fost întrerupt și a repornit din nou din cauza flag-ului de restart.

<sup>14</sup>Un server proxy este un intermediar între client și server. Acesta acționează în numele clientului, primind solicitările de la client și înaintându-le către server, apoi returnează răspunsurile primite de la server înapoi la client. Serverul proxy poate fi configurat pentru a efectua diverse funcții, cum ar fi filtrarea conținutului, ascunderea adresei IP a clientului, îmbunătățirea performanței prin cache sau balansarea sarcinii între mai mulți servere. De asemenea, serverul proxy poate fi utilizat pentru a asigura securitatea prin intermediul unor funcții precum autentificare, criptare sau protecție împotriva atacurilor de tipul Distributed Denial of Service (DDoS). În general, serverele proxy sunt utilizate pentru a îmbunătăți performanța, securitatea și confidențialitatea comunicațiilor între client și server.

<sup>15</sup><https://docs.python.org/3/library/xmlrpc.html>

- "startProcess()" pentru a porni aplicația dorită
- "stopProcess()" pentru a opri aplicația dorită

### 3.2.6.1. Crearea mediului pentru rularea joburilor în Supervisord

Asa cum a fost prezentat în subcapitolul 3.2.5 înainte de a porni un job trebuie sa avem niște pași premergători pentru a crea toate fișierele asa cum este specificat în documentația Supervisord.

În procesul de rezolvare a acestei probleme, utilizarea SSH-ului a fost deosebit de importantă. Prin intermediul SSH-ului, am putut executa comenzi în shell<sup>16</sup>, adică am putut interacționa și efectua operații în mediul liniei de comandă al sistemului respectiv.

Operațiile efectuate au inclus crearea fișierelor de jurnal în directorul specific utilizatorului, în acest caz, în fișierul destinat utilizatorului ("/home/server/testsupervisor/"), în cazul nostru acesta se numește "server".

Una dintre provocările majore întâmpinate în această parte a fost crearea fișierului .conf, care definește modul în care va fi rulat în mediul oferit de Supervisord. Am decis să creăm acest fișier de configurare pe partea locală și apoi să-l transferăm prin SFTP<sup>17</sup> către fișierul destinat utilizatorului pe server. După ce a ajuns în directorul userului, trebuie mutat în fișierul destinat configurațiilor, respectiv "/etc/supervisor/conf.d". Pentru a realiza acest scop, a trebuit să creăm o funcție care să mute fișierul și să furnizeze parola, deoarece modificările efectuate în directorul "/etc" necesită privilegii suplimentare. Codul de la 3.7 muta fișierul în directorul dorit din "/etc":

```

1  def move_file_in_etc(self):
2      stdin, stdout, stderr = self.ssh_server.exec_command(
3          command="sudo cp /home/server/testsupervisor/" +
4              ↪ self.configuration.name + ".conf /etc/supervisor/conf.d",
5              ↪ get_pty=True)
6      stdin.flush()
7      stdin.write(self.password + "\n")
8      if stderr.channel.recv_exit_status() != 0:
9          dl.logger.info(f"Can't execute the command :
10             ↪ {stderr.readlines()}")
11     else:
12         dl.logger.info(f"The following was produced: \n
13             ↪ {stdout.readlines()}")
14     self.ssh_server.exec_command("rm /home/server/testsupervisor/" +
15         ↪ self.configuration.name + ".conf")

```

Listing 3.7. Metoda ce mută fișierul .conf în directorul din /etc

<sup>16</sup>Shell-ul este un program de linie de comandă care oferă o interfață între utilizator și sistemul de operare. Acesta permite utilizatorului să introducă comenzi și să interacționeze cu sistemul de fișiere, programele și serviciile disponibile pe sistemul de operare.

Shell-ul interpretează comenzile introduse de utilizator și le execută în sistem. De asemenea, acesta oferă funcționalități suplimentare precum redirectionarea intrărilor și ieșirilor, gestionarea proceselor, crearea de script-uri pentru automatizarea sarcinilor și multe altele.

<sup>17</sup>SFTP este un acronim pentru "SSH File Transfer Protocol" (Protocol de transfer de fișiere SSH). Este un protocol criptat și securizat care permite transferul de fișiere între un client și un server prin intermediul unei conexiuni SSH. SFTP utilizează criptarea și autentificarea furnizate de protocolul SSH pentru a asigura confidențialitatea și integritatea datelor transferate.

SFTP oferă o interfață similară cu FTP (File Transfer Protocol), dar operează într-un mediu securizat. Prin SFTP, utilizatorii pot efectua operațiuni de transfer de fișiere, cum ar fi descărcarea (download), încărcarea (upload), ștergerea, redenumirea și navigarea prin structura de directoare de pe server.

### 3.2.7. Obiectul de tip Job

La fel de importantă ca și clasa "Server" 3.2.2.1, sau cu o importanță chiar mai mare este Job-ul, care reprezintă roțița ce pune în funcțiune scopul principal al proiectului, acela de a pregăti mediul în care vă rula aplicația și de a monitoriza rularea îndeaproape pe tot parcursul acesteia.

În ceea ce privește monitorizarea am optat pentru o idee simplă, anume aceea de a folosi un "whatchdog"<sup>18</sup> pentru fiecare proces ce urmează să fie rulat. În cod, putem găsi implementarea aceasta inițializată într-un fir de execuție (thread) separat, pentru a putea procesa simultan mai multe joburi în același timp.

```

1  def start_job(self):
2      self.create_enviroment_to_run_program_in_supervisor()
3      self.start_process()
4      self.update_job_info()
5      dl.logger.info(f"Started job on server {self.ip} with PID :
        ↳ {self.information['pid']}")
6      thr = threading.Thread(target=self.check_jos_is_running)
7      thr.start()

```

Listing 3.8. Metoda ce pregătește și porneste job-ul pe server

În codul prezentat în 3.8, se apelează trei metode pentru a crea mediul de funcționare al executabilului dorit ("create-enviroment-to-run-program-in-supervisor()"), precum și pentru a porni procesul cu "start-process". După ce ne-am asigurat că jobul a fost pornit, vom prelua datele furnizate de Supervisorul pentru a le utiliza în verificări ulterioare. Aceste verificări au scopul de a ne asigura că folosim același PID și ca programul nu a suferit erori neprevăzute.

### 3.3. Dificultăți întâmpinate și soluții

În cadrul implementării acestui proiect, am întâmpinat mai multe provocări tehnice care au necesitat eforturi semnificative pentru a fi depășite. Prima provocare întâmpinată a fost monitorizarea proceselor. În încercarea de a găsi o soluție, am luat în considerare inițial utilizarea comenzilor în linia de comandă. Cu toate acestea, am realizat că acest lucru ar fi o reinventare a roții și nu ar fi o soluție eficientă pe termen lung. Ca urmare, am decis să folosim Supervisorul pentru a rezolva această problemă. Supervisorul ne-a oferit un set complet de funcționalități pentru monitorizarea și controlul proceselor, inclusiv repornirea automată în caz de eșec, gestionarea jurnalului și multe altele.

Configurarea Supervisorul și a conexiunii SSH: A fost necesară o documentare intensivă și o înțelegere aprofundată a setărilor și opțiunilor disponibile în Supervisorul. De asemenea, am avut nevoie de cunoștințe avansate privind configurarea conexiunii SSH pentru a asigura securitatea și conectivitatea adecvată între aplicație și server.

După ce am început să utilizăm Supervisorul, o altă provocare a apărut în ceea ce privește realizarea unei conexiuni către un alt IP, nu doar către localhost. Acest lucru a generat mici probleme în configurarea aplicației și în asigurarea rulării acesteia conform setărilor personalizate dorite.

<sup>18</sup>Termenul "watchdog" (câine de pază) se referă la un mecanism de monitorizare sau o entitate software care are rolul de a monitoriza și verifica starea unui sistem sau a unei aplicații pentru a detecta și gestiona erori sau situații anormale.

Un "watchdog" este conceput pentru a asigura că sistemul funcționează în parametri normali și pentru a interveni în cazul în care apar probleme sau eșecuri. Acesta poate verifica periodic starea sistemului, cum ar fi disponibilitatea serviciilor, resursele utilizate, procesele în desfășurare sau alte criterii specifice. Dacă un watchdog detectează o situație anormală, poate iniția acțiuni corective, cum ar fi repornirea serviciului sau trimiterea unei alerte pentru a notifica administratorii sistemului.

Pentru a depăși această problemă, am revizuit și ajustat configurațiile aplicației noastre pentru a ne asigura că se face conexiunea corectă cu adresa IP specificată. Am efectuat teste riguroase pentru a verifica conectivitatea și funcționarea corectă în noua configurație, iar apoi am ajustat setările necesare în conformitate cu cerințele și specificațiile dorite. Astfel, s-a reușit stabilirea conexiunii cu IP-ul corespunzător și asigurarea unei funcționari corecte a aplicației în mediul dorit.

O provocare la fel de mare a fost crearea unor funcții care să execute comenzi SSH cu drepturi de administrator. În acest punct, principala problemă a constat în interpretarea mesajului returnat de linia de comandă după executarea comenzii.

Pentru a depăși această provocare, am implementat funcții speciale care să gestioneze execuția comenzilor SSH și să interpreteze rezultatele obținute. Am utilizat tehnici de analiză a șabloanelor (*parsing*) pentru a extrage informațiile relevante din mesajele returnate de linia de comandă. În plus, am dezvoltat mecanisme de gestionare a erorilor și de validare a rezultatelor pentru a ne asigura că interpretarea este corectă și că putem lua decizii adecvate în funcție de rezultatele obținute.





## Capitolul 4. Testarea aplicației și rezultate experimentale

### 4.1. Punerea în funcțiune a aplicației

Înainte de a porni aplicația, este necesar să efectuăm un set de pași premergători pentru a ne asigura că totul funcționează așa cum ne-am propus și că avem la dispoziție toate resursele necesare.

Pentru a ne asigura că aplicația poate fi executată pe mașina dorită, este necesar să avem limbajul de programare Python instalat. În sistemele de operare Unix, Python se poate instala folosind comanda:

Listing 4.1: Comanda bash pentru instalare Python

```
$ sudo apt-get install python
```

#### 4.1.1. Pornirea aplicațiilor auxiliare

Un alt pas necesar pentru a putea testa aplicația este de a inițializa RabbitMQ și MongoDB. Așa cum am prezentat în capitolul 2, aceste tehnologii trebuie să fie inițializate în containere Docker utilizând Docker-Compose.

Primul pas ar fi instalarea aplicațiilor Docker și Docker Compose cu ajutorul comenzilor:

Listing 4.2: Comanda instalare Docker și Docker Compose

```
$ sudo snap install docker
$ sudo apt install docker-compose
```

După ce aplicațiile au fost instalate cu succes putem crea fișierele de tip `.yaml`<sup>19</sup> ce vor conține configurația bazei de date și RabbitMQ. Am optat pentru a rula aplicațiile pe porturile sugerate de documentația oficială.

Având fișierele cu configurația docker, următorul pas e să le ridicăm rulând comanda:

Listing 4.3: Comanda pornire Docker

```
$ sudo docker-compose up
```

#### 4.1.2. Configurare server

##### 4.1.2.1. Configurație VirtualBox

Pentru a simula serverele care urmează să se conecteze la aplicație, am utilizat VirtualBox pentru a crea și configura mașini virtuale care reproduc un mediu similar cu cel real. Pentru partea de rețea, am utilizat două adaptoare de rețea pe mașinile virtuale. Unul dintre adaptoarele de rețea utilizate a fost "NAT" (Network Address Translation) permite mașinii virtuale să se conecteze la internet prin intermediul gazdei (host-ului) fizic, cu ajutorul acestei configurări, mașinile virtuale utilizează adresa IP a gazdei și conexiunea de rețea a acesteia pentru a accesa internetul.

Celălalt adaptor de rețea utilizat a fost "Host-only Adapter". Acesta creează o rețea virtuală între mașinile virtuale și gazdă, fără a le conecta la rețeaua fizică externă. Adaptorul "Host-only"

<sup>19</sup>Fișierele cu extensia `.yaml` sau `.yml` reprezintă fișiere de configurare utilizate pentru a defini structuri de date și setări într-un format ușor de citit pentru om și mașină. YAML (YAML Ain't Markup Language) este un limbaj de serializare a datelor, independent de limbajul de programare, care se bazează pe utilizarea spațiilor și indentării pentru a reprezenta ierarhiile și relațiile dintre obiecte.

Fișierele `.yaml` sunt adesea folosite pentru configurarea aplicațiilor și serviciilor, unde pot defini parametrii, setările și opțiunile necesare pentru funcționarea corespunzătoare a acestora. Aceste fișiere pot conține liste, dicționare, valori scalare și chiar referințe către alte obiecte sau fișiere.

```
puscasu@puscasu:~/mongodb$ cat docker-compose.yml
# Use root/example as user/password credentials
version: '3.1'

services:

  mongo:
    image: mongo
    restart: always
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: example

  mongo-express:
    image: mongo-express
    restart: always
    ports:
      - 8081:8081
    environment:
      ME_CONFIG_MONGODB_ADMINUSERNAME: root
      ME_CONFIG_MONGODB_ADMINPASSWORD: example
      ME_CONFIG_MONGODB_URL: mongodb://root:example@mongo:27017/
```

Figura 4.1. Configurație MongoDB

```
puscasu@puscasu:~/rabbitmq$ cat docker-compose.yml
version: "3.6"
# https://docs.docker.com/compose/compose-file/
services:
  rabbitmq:
    image: 'rabbitmq:3.6-management-alpine'
    ports:
      # The standard AMQP protocol port
      - '5672:5672'
      # HTTP management UI
      - '15672:15672'
    environment:
      # The location of the RabbitMQ server. "amqp" is the protocol;
      # "rabbitmq" is the hostname. Note that there is not a guarantee
      # that the server will start first! Telling the pika client library
      # to try multiple times gets around this ordering issue.
      AMQP_URL: 'amqp://rabbitmq?connection_attempts=5&retry_delay=5'
      RABBITMQ_DEFAULT_USER: "guest"
      RABBITMQ_DEFAULT_PASS: "guest"
    networks:
      - network
networks:
  # Declare our private network. We must declare one for the magic
  # Docker DNS to work, but otherwise its default settings are fine.
```

Figura 4.2. Configurație RabbitMQ

permite comunicarea directă între mașinile virtuale și gazdă, precum și între mașinile virtuale în sine.

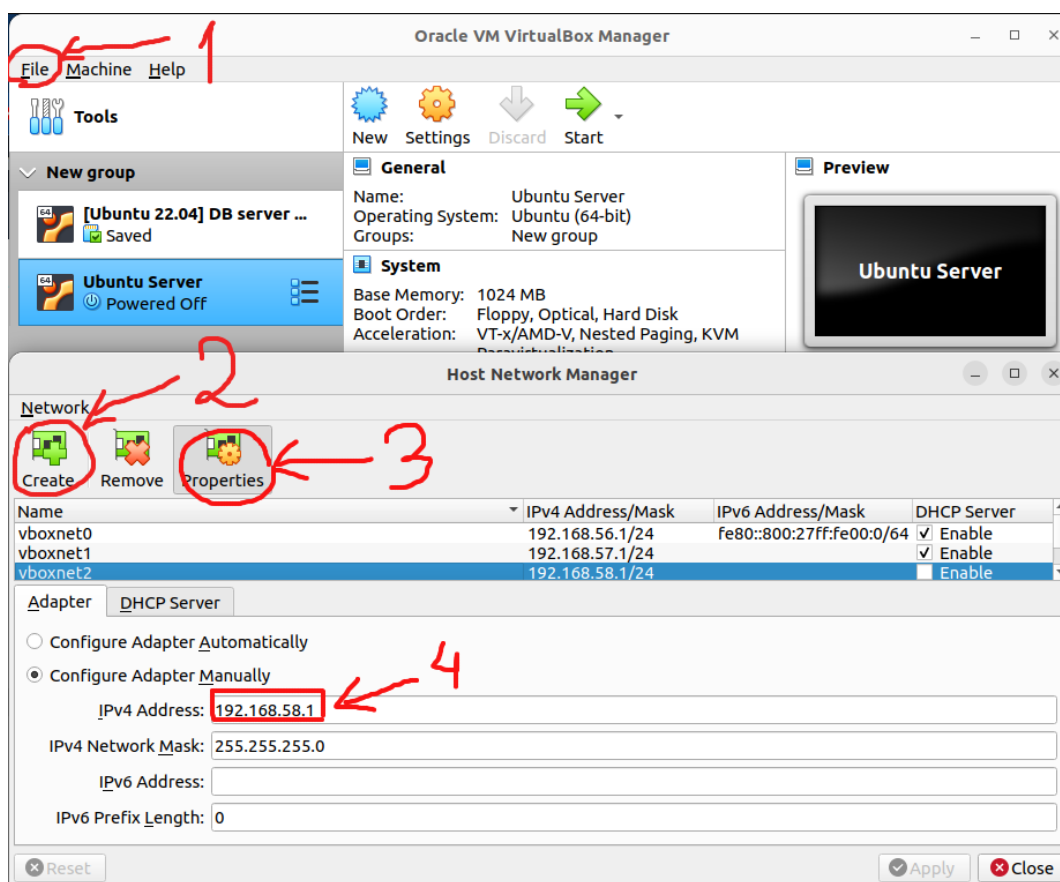


Figura 4.3. Crearea Host-only

Pentru a crea o rețea virtuală în VirtualBox și a o adăuga mașinii dorite, trebuie efectuați următorii pași:

- 1. Accesam din interfața principală: File -> Host Network Manager
- 2. După ce apare noua interfața se crează o nouă rețea folosind butonul "Create" și după din "Properties" se setează IP-ul dorit. (Figura 4.3)
- 3. Având rețeaua virtuală, se accesează setările mașinii pe care dorim să o configurăm și adăugăm rețeaua la câmpul "Host-only Adapter" la fel ca în figura 4.4

#### 4.1.2.2. Instalarea sistemului de operare

Un pas poate la fel de important ca cel de configurat mașinile virtuale din VirtualBox, dar care este obligatoriu a fost instalarea sistemului de operare ce va opera pe măsliniile virtuale. Am folosit un sistem de operare care poate rula cu puține resurse, Ubuntu Server 20.04<sup>20</sup>. Pentru a putea ridica o mașină care rulează folosind un astfel de sistem de operare avem nevoie de minim un *core*, 20 GB de spațiu și unul de memorie RAM.

<sup>20</sup>Ubuntu Server este o distribuție de sistem de operare bazată pe Linux, dezvoltată și distribuită de Canonical Ltd. Este o versiune optimizată și configurată special pentru utilizarea în medii de server, oferind un sistem de operare fiabil și sigur pentru a găzdui și administra diverse servicii și aplicații. Ubuntu Server se bazează pe distribuția principală Ubuntu, dar vine cu un set de pachete software și configurații predefinite orientate către nevoile și cerințele specifice ale unui mediu de server. Acesta oferă o platformă solidă pentru rularea serverelor web, serverelor de baze de date, serverelor de fișiere, serverelor de aplicații și multe altele.

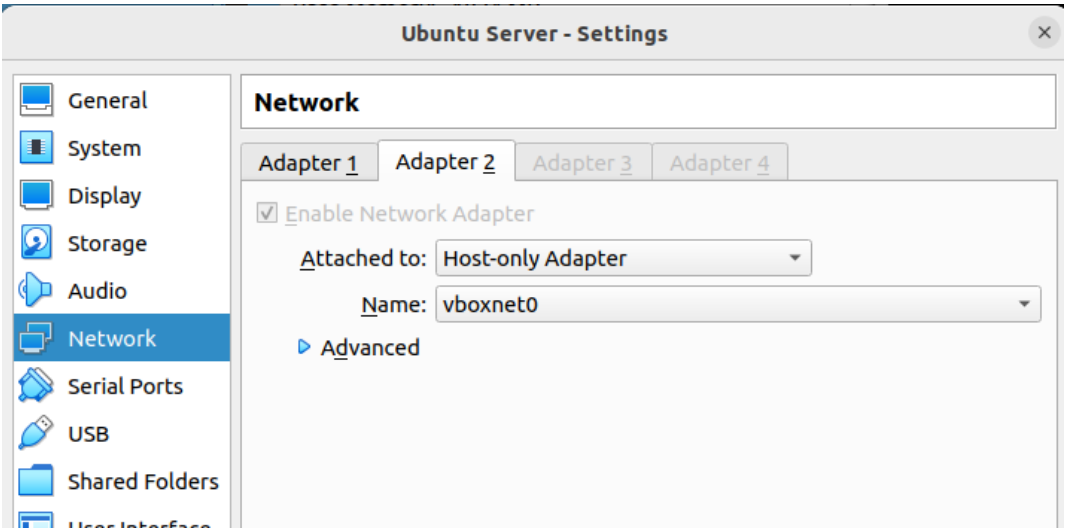


Figura 4.4. Setare Host-only Adapter

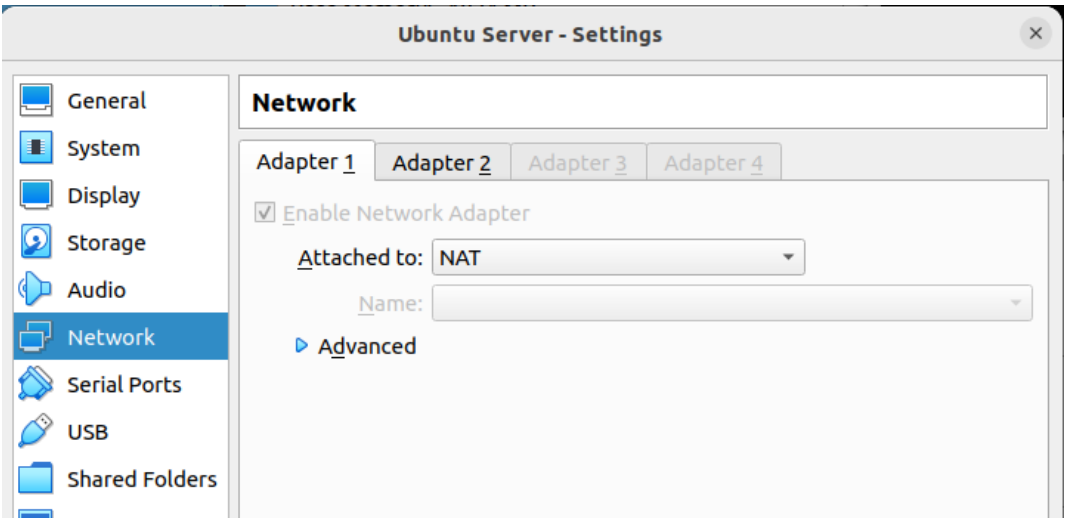


Figura 4.5. Configurație VirtualBox

### 4.1.3. Instalarea Supervisor

Având o conexiune stabilită între măsliniile ridicate în VirtualBox și host putem trece la următorul pas, acela fiind de a instala aplicația Supervisor pe serverele aflate în subordine. Instalarea este simplă aceasta făcându-se cu ajutorul unor simple comenzi<sup>21</sup>, iar dacă aplicația deja rulează statusul ei poate fi verificat cu comanda 4.1.3.

Listing 4.4: Comanda instalare Supervisor

```
$ sudo apt update && sudo apt install supervisor
```

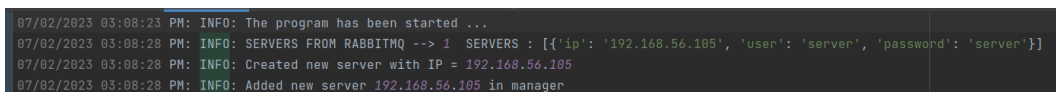
Listing 4.5: Verificare status Supervisor

```
$ sudo systemctl status supervisor
```

### 4.2. Rezultate obținute

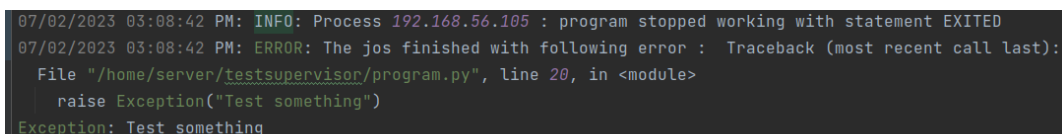
Pentru a asigura monitorizarea corectă a execuției aplicației, au fost utilizate log-uri cu scopul de a urmări în detaliu erorile care pot apărea, dar și modul în care funcționează aplicația.

Log-urile reprezintă înregistrări detaliate ale evenimentelor și activităților care apar în timpul rulării aplicației. Acestea oferă o perspectivă în timp real asupra funcționării aplicației, permițând identificarea și diagnosticarea erorilor, comportamentelor neașteptate sau a altor aspecte relevante. Câteva imagini cu logurile afișate de program în timpul rulării figura 4.6 și 4.7.



```
07/02/2023 03:08:23 PM: INFO: The program has been started ...
07/02/2023 03:08:28 PM: INFO: SERVERS FROM RABBITMQ --> 1 SERVERS : [{'ip': '192.168.56.105', 'user': 'server', 'password': 'server'}]
07/02/2023 03:08:28 PM: INFO: Created new server with IP = 192.168.56.105
07/02/2023 03:08:28 PM: INFO: Added new server 192.168.56.105 in manager
```

Figura 4.6. Log adaugare server



```
07/02/2023 03:08:42 PM: INFO: Process 192.168.56.105 : program stopped working with statement EXITED
07/02/2023 03:08:42 PM: ERROR: The job finished with following error : Traceback (most recent call last):
  File "/home/server/testsupervisor/program.py", line 20, in <module>
    raise Exception("Test something")
Exception: Test something
```

Figura 4.7. Log eroare server

Aplicația a fost dezvoltată cu succes și funcționalitatea sa a fost implementată conform cerințelor. Testele efectuate au evidențiat un mare succes, demonstrând că aplicația îndeplinește scopul și funcționează în conformitate cu așteptările.

Pentru a testa și evalua funcționalitatea aplicației, au fost utilizate două servere și mai multe job-uri care au fost distribuite pe aceste servere pentru rulare. Acest scenariu a permis testarea simultană a mai multor job-uri și evaluarea capacității de gestionare a aplicației într-un mediu realist.

În plus, a fost implementat și scenariul în care un job este pornit cu flag-ul de restart. Acest lucru a permis observarea comportamentului aplicației în ceea ce privește repornirea job-ului și schimbarea PID-ului pe care rulează. Prin monitorizarea bazei de date, am putut observa cum job-ul repornește în mod automat ori de câte ori este necesar, asigurând astfel continuitatea și stabilitatea aplicației.

Aceste teste au oferit oportunitatea de a evalua performanța, scalabilitatea și comportamentul aplicației în condiții reale, simulând situații complexe și variabile.

<sup>21</sup>Pașii pot fi găsiți și pe : <https://www.digitalocean.com/community/tutorials/how-to-install-and-manage-supervisor-on-ubuntu-and-debian-vps>

licenta.licenta

Documents Aggregations Schema Explain Plan Indexes

FILTER { field: 'value' }

ADD DATA VIEW

```
> {
  "_id": ObjectId("64a1686450c06a57bc00cb34"),
  "ip": "192.168.56.105",
  "job_name": "program.py",
  "autor_restart": true,
  "status": "RUNNING",
  "pid": Array
    0: 17872
    1: 17894
    2: 17894
    3: 17895
    4: 17896
  }

{
  "_id": ObjectId("64a1686650c06a57bc00cb35"),
  "ip": "192.168.56.105",
  "job_name": "program_test.py",
  "autor_restart": false,
  "status": "RUNNING",
  "pid": Array
    0: 17893
  }
```

Figura 4.8. Job-uri ce au rulat de mai multe ori

## Concluzii

În urma analizei detaliate a aplicației software descrise în capitolele anterioare, putem trage concluzia că am reușit să dezvoltăm o soluție tehnologică avansată, care poate fi aplicată cu succes în domeniul său specific. Această aplicație, construită pe baza unor tehnologii și soluții contemporane, se distinge prin robustețea și stabilitatea sa structurală, oferind în același timp un grad înalt de flexibilitate și scalabilitate. Aceste caracteristici o fac ideală pentru extindere și adaptare ulterioară, în vederea satisfacerii unor nevoi și cerințe similare.

În ceea ce privește procesul de implementare, acesta a reprezentat o oportunitate valoroasă de învățare și dezvoltare profesională. Am avut ocazia să înțelegem în profunzime utilitatea și funcționalitatea cozilor de mesaje, precum și importanța unei monitorizări precise și riguroase a sistemelor. De asemenea, am învățat să gestionăm și să tratăm eficient situațiile neprevăzute, cum ar fi erorile, ceea ce ne-a permis să îmbunătățim calitatea și fiabilitatea aplicației.

Un alt aspect important al procesului de implementare a fost familiarizarea cu tehnologii noi, precum Supervisor. Această experiență ne-a oferit oportunitatea de a ne extinde cunoștințele și competențele tehnice, contribuind astfel la dezvoltarea noastră profesională continuă.

În concluzie, dezvoltarea acestei aplicații a reprezentat un proces complex și provocator, dar în același timp extrem de valoros și instructiv. Rezultatul final este o soluție software robustă, flexibilă și scalabilă, care poate fi aplicată cu succes în domeniu și care reflectă cele mai recente tendințe și soluții tehnologice.

### *Direcții viitoare de dezvoltare*

Din punctul meu de vedere, aplicația are multiple direcții viitoare de dezvoltare, însă cele mai relevante în acest moment ar fi crearea unei interfeței, probabil web, care să implementeze funcționalitatea "drag and drop" pentru fișierele pe care dorim să le executăm. De asemenea, ar fi util să putem seta flag-uri din interfață pentru a rula aplicația, cum ar fi flag-ul de restart, și să avem opțiunea de a specifica comanda de executare, de exemplu, având variabile predefinite pentru a permite rularea în diferite limbaje.

Tot în interfață, ar trebui să avem și un sistem de feedback pentru joburile care rulează. De exemplu, joburile care au rulat cu succes ar putea fi afișate în verde, în timp ce joburile care au avut erori să fie afișate cu roșu.

Pe partea de backend, ar putea fi implementați algoritmi de clasificare diferiți, în funcție de joburile pe care dorim să le executăm. Un exemplu ar fi să avem mai mulți manageri de server care rulează folosind algoritmi diferiți, probabil specificați de utilizator în interfață. Un exemplu ar fi ca unul dintre acești manageri să funcționeze pe principiul FIFO (primul intrat, primul ieșit), în timp ce altul ar putea prioritiza joburile în funcție de anumite criterii. Astfel, atunci când pasăm un job nou, am putea specifica modul în care dorim să fie executat, ceea ce ar duce la o rulare mai rapidă sau mai lentă, în conformitate cu preferințele noastre.





## Bibliografie

- [1] IBM, “Computer-cluster,” <https://www.ibm.com/docs/en/zos-basic-skills?topic=sysplex-computer-cluster>.
- [2] ———, “Cloud computing,” <https://www.ibm.com/topics/cloud-computing>.
- [3] R. T. Lange, “Introducing mle-scheduler: A lightweight tool for cluster/cloud vm job management,” <https://roberttlange.com/posts/2021/11/mle-scheduler/>, 2021.
- [4] C. N. C. F. Google, Rancher Labs, “Kubernetes documentation,” <https://kubernetes.io/docs/home/>, 2021.
- [5] S. H. Perveez, “Understanding kubernetes architecture and its use cases,” <https://www.simplilearn.com/tutorials/kubernetes-tutorial/kubernetes-architecture>.
- [6] L. L. N. L. SchedMD, “Slurm,” <https://slurm.schedmd.com/documentation.html>.
- [7] K. Kawaguchi, “Jenkins,” <https://www.jenkins.io/>.
- [8] IBM, “Object-oriented programming in python,” <https://www.ibm.com/docs/en/watsonx?topic=language-object-oriented-programming>.
- [9] Y. Poirier, “Core design principles,” <https://blogs.oracle.com/java/post/core-design-principles>.
- [10] “Rabbitmq,” <https://www.rabbitmq.com/>.
- [11] “Apache kafka,” <https://kafka.apache.org/>.
- [12] IBM, “What is docker?” <https://www.ibm.com/topics/docker>.
- [13] A. Ligios, “Introduction to docker compose,” <https://www.baeldung.com/ops/docker-compose>, 2022.
- [14] R. Hill, “Getting started with ssh security and configuration,” <https://developer.ibm.com/articles/au-sshsecurity/>, 2011.
- [15] C. McDonough, “Supervisor,” <http://supervisord.org/introduction.html>.
- [16] Oracle, “Virtualbox,” <https://www.virtualbox.org/wiki/Documentation>.



## Anexe

### Anexa 1. Structura fișierelor

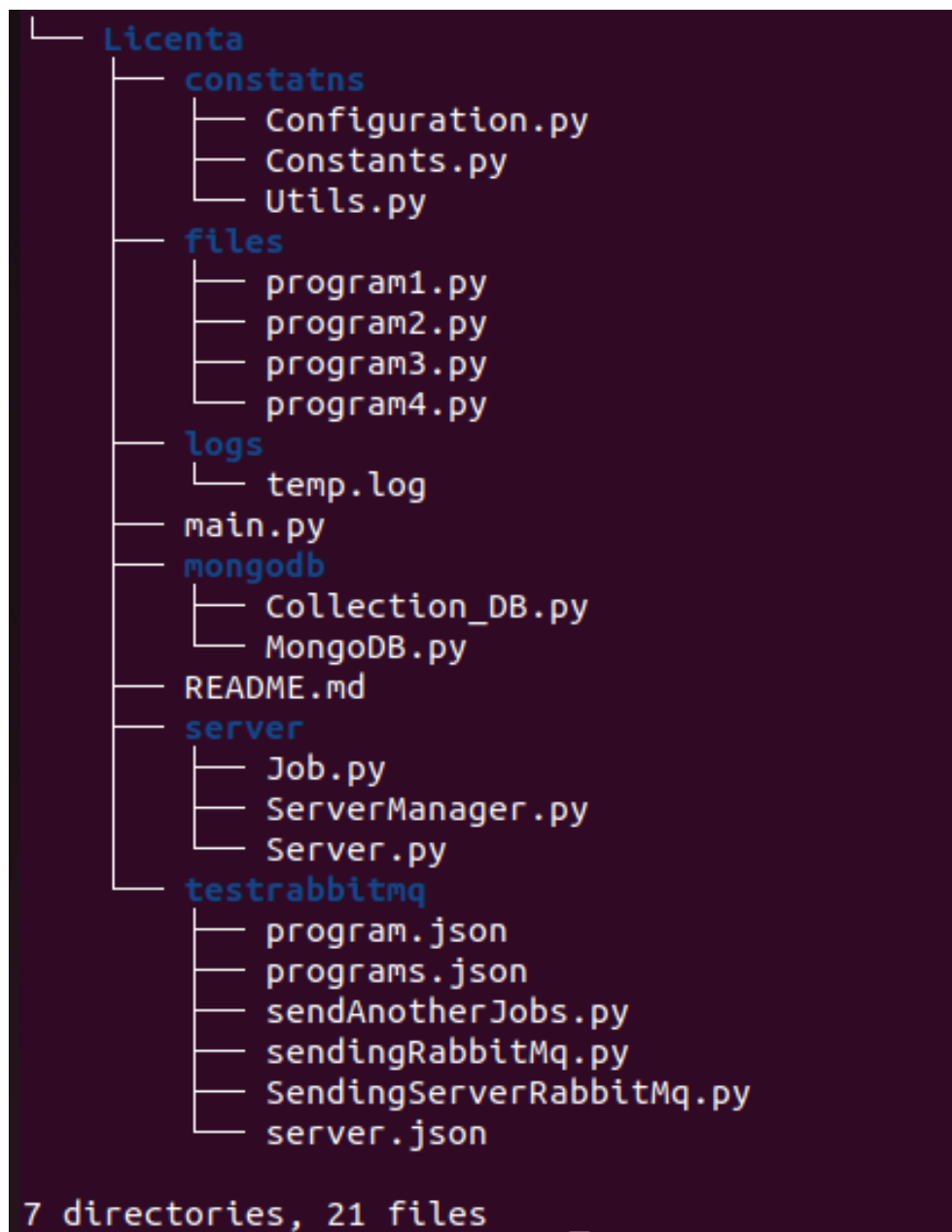


Figura A.1. Structura fișierelor

