**RAPORT CECBID – Robert Poenaru**

**Mai 2022**

**1.Implementarea unui serviciu de administrare al pachetelor pentru sistemele de calcul virtualizate (VM) dintr-o retea**

Scopul acestui serviciu este de a putea asigura utilizatorului o interfata web care sa permita afisarea unor liste cu pachetele software instalate pe instantele de computing ce ruleaza in reteaua Openstack [1] si asupra carora utilizatorul are drept de acces. In versiunea curenta, acest serviciu ofera posibilitatea de a putea vizualiza pachetele, de a verifica existenta unor versiuni noi comparativ cu cele deja instalate, si de a putea aduce fiecare pachet/serviciu la versiunea cea mai noua disponibila.

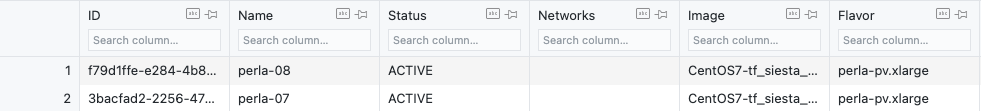
Pentru dezvoltarea sistemului, s-a folosit in mod principal modulul Flask [3]. Flask permite realizarea de aplicatii web prin intermediul limbajului de programare Python, ceea ce ofera un grad foarte mare de compatibilitate intre diferitele arhitecturi hardware dar si sisteme de operare. Aplicatia *principala* dezvoltata in cadrul proiectului contine mai multe functii, iar una dintre acestea este managementul de pachete de pe resursele de calcul, al carui mecanism de functionare va fi descris in cele ce urmeaza. Aceasta va rula in permanenta pe un *server principal*, i.e., un nod de calcul deschis pentru accesul web de catre public (in continuare se va utiliza doar denumirea de server pentru a se face referire la nodul pe care este dezvoltat serviciul web Flask).

In procesul vizualizare al pachetelor de pe VM-urile disponibile, exista atat o parte de *backend* (modul de executie al functiilor dezvoltate in Python) dar si *frontend* (interfata grafica efectiva pe care utilizatorul o acceseaza/utilizeaza). La fiecare conectare, practic, utilizatorul va putea vizualiza informatii care au fost create sau procesate in partea de backend, prin executarea unui intreg grup de functii si module Python.

*1.1 Backend*

Toate informatiile cu privire la masinile virtuale disponibile pentru utilizator (sau denumit in continuare *user*) sunt preluate dintr-o baza de date ce exista in prealabil, pe unul din agentii principali ai retelei Openstack. Baza de date cu masinile virtuale a fost creata in cadrul unei implementari separate ce *interogheaza* constant infrastructura OpenStack, verificand ce resurse sunt atribuite utilizatorului. Baza de date este obtinuta prin executarea a unei comenzi de tip terminal, iar rezultatul comenzii va fi captat, parsat si stocat in userID.openstack.VM.list.db.

Numele bazei de date este configurat in asa fel incat sa se poata asigura o diferentiere a masinilor virtuale disponibile pe OpenStack si pentru ca interfata web sa acceseze fisierul ce corespunde in mod corect utilizatorului logat. Modulul sqlite3 [4] asigura fluxul de citire/modificare a bazei de date, iar structura acesteia poate fi vazuta in figura de mai jos:



Campul ID contine un lant de caractere (numit *hash*) care va fi atribuit in mod unic fiecarei masini virtuale. Acest lucru va facilita controlul individual al fiecarui VM, intrucat user-ul va putea avea instante cu nume identic (campul Name din figura), insa ID-ul va fi diferit pentru fiecare.

In momentul in care utilizatorul va accesa pagina de management de pachete din cadrul aplicatiei principale Flask, va avea la dispozitie un buton prin care poate vizualiza intreaga lista de VM-uri (aceasta listava fi prezentata la partea de frontend). Prin apasarea butonului se va interoga baza de date userID.openstack.VM.list.db , proces realizat cu ajutorul functiei refresh\_instances().

Legatura (i.e., modul de comunicare) dintre client si server se realizeaza prin schimbul constant de evenimente asigurat de serviciul socketIO [5]. Acest modul este necesar atat pe aplicatia Flask dezvoltata in Python:

from flask\_socketio import SocketIO

from flask\_socketio import emit

from flask import Flask, render\_template

cat si pe interfata web (HTML + JavaScript + CSS):

<!-- use socketIO -->

<script

src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/4.0.1/socket.io.js"

crossorigin="anonymous">

</script>

Un eveniment socketIO definit in interfata web fi configurat sa trasmita (denumirea tehnica este de a *emite*) o anumita comanda catre server, dar si viceversa. Astfel, prin apasarea butonului „instance-refresher”, un eveniment cu comanda „refresh\_instances” va fi emis catre server:

$("#instance-refresher").click(() => {

console.log("Refreshing instances");

sio.emit("refresh\_instances");

});

Serverul va raspunde acestui mesaj tot prin intermediul unui eveniment socketIO, executand un set de instructiuni in cadrul functiei refresh\_instances()si finalizand cu transmiterea raspunsului catre client (aplicatia web):

@socketio.event

def refresh\_instances():

print('User requested VM list')

active\_vms = [(vm[0], vm[1]) for vm in vm\_db.get\_user\_vms(VM\_DB)]

emit('instances', {'vms': active\_vms})

In codul de mai sus, variabilele vm[0] si vm[1] reprezinta ID-ul si numele masinilor virtuale ce sunt preluate din baza de date. Lista de VM-uri se transmite catre interfata web cu ajutorul comenzii *emit* din modulul socketIO. Odata ce lista este primita, aceasta va fi redata ca si continut HTML pe pagina activa. Intrucat editarea continua a sursei „.html” nu este o metoda eficienta, se va folosi jQuery [6]:

<!-- use jQuery -->

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js">

</script>

pentru a putea accesa doar anumite campuri (tag-uri) din pagina HTML. In cazul afisarii masinilor virtuale, acestea vor fi asezate sub forma unei liste selectabile (in termeni tehnici aceasta se numeste *dropdown-*list) in momentul in care instantele VM sunt primite de la serverul principal:

//save the instances list from the server as an array

sio.on("instances", (data) => {

$("#vm-drp-list").empty();

data.vms.forEach((element) => {

let vm\_name = element[1];

let vm\_id = element[0];

$("#vm-drp-list").append(

'<a class="dropdown-item" href="#">' + vm\_name + "</a>"

);

vm\_id\_list.push(vm\_id);

vm\_name\_list.push(vm\_name);

});

});

In codul de mai sus, prima comanda se asigura ca tabelul este gol inainte de initializare, iar variabilele vm\_id\_list, vm\_name\_list sunt utilizate pentru memorarea fiecarui ID si nume de VM, oferind posibilitatea de utilizare ulterioara a acestora. Prin comanda „append” (specifica JavaScript / jQuery) se adauga in dropdown-list numele ficarei masini virtuale, iar prin parametrul HTML „<a>” elementul devine selectabil in pagina. Mai departe, utilizatorul va putea selecta din lista de VM-uri, una specifica (afisarea se face pe baza campului Name din baza de date initiala). Selectarea prorpiu-zisa a unei masini virtuale este pasul necesar pentru afisarea listei de pachete instalate pe acel sistem. Deoarece intregul set de pachete ce sunt instalate in mod standard (*by default*) pe o masina virtuala tip Linux (mai precis cele de tip RHEL [7]) contine si multe servicii care nu sunt de interes utilizatorului, aplicatia va selecta doar acele pachete de dezvoltare (development tools) cu care utilizatorul isi va putea controla calculele si simularile dorite. Codul de mai jos este responsabil cu selectarea unui VM din dropdown:

//actions when the user selects an item from vm-drp-list

$("#vm-drp-list").on("click", "a", (e) => {

selected\_vm = e.target.text;

let selected\_vm\_index = $("#vm-drp-list a").index(e.target) + 1;

sio.emit("vm\_selected", {

vm\_id: vm\_id\_list[selected\_vm\_index - 1],

vm\_name: vm\_name\_list[selected\_vm\_index - 1],

});

Alegerea de catre utilizator a unui VM specific din lista se face dupa numele acestuia, insa aplicatia va identifica automat si ID-ul (prin cele doua variabile de tip vector despre care s-a discutat mai sus: vm\_id\_list, vm\_name\_list). In momentul selectarii propriu-zise a unui element din lista, un alt eveniment socketIO va fi emis catre server, prin care se va cere (sau in termeni tehnici *request*) informatia cu privire la pachetele masinii respective. Evenimentul „vm\_selected” din codul de mai sus va trimite catre server un mesaj care va contine numele si ID-ul masinii, iar functia „sio.emit” este responsabila de crearea acestui eveniment. Ulterior, serverul va raspunde evenimentului prin functia urmatoare:

@socketio.event

def vm\_selected(data):

vm\_id = data['vm\_id']

vm\_name = data['vm\_name']

vm\_packages = pack.get\_vm\_packages(USER\_ID, vm\_id) <--- sends command to VM for package retrieval

...

Odata ce serverul primeste informatiile cu privire la masina virtuala, prin intermediul protocolului (modulului) MQTT [8] va fi transmisa o comanda de tip terminal pe un canal (sau topic) unic de comunicare intre serverul principal si VM. In mod normal, un canal de comunicare mqtt are forma „nume\_canal\_principal/nume\_canal\_secundar/...”. Comanda de terminal ce va fi executata are forma urmatoare: yum list --installed | more | grep devel, iar canalul mqtt este creat special pentru acel utilizator si acea masina virtuala. Un exemplu test de topic mqtt pentru un utilizator denumit „user69” si pentru un VM selectat va fi:

/openstack/cloudifin/servers/user69/f79d1ffe-e284-4b86-926a-c6a6b23859d1

cu structura (retea openstack)/(ID-ul unic al utilizatorului)/(ID-ul masinii virtuale). In final, masina virtuala va primi un mesaj (sub forma unui strig binar ce va fi decodat in format UTF-8) care contine comanda de terminal. VM-ul va executa aceasta comanda, iar rezultatul din terminal va fi re-trimis catre serverul principal tot prin mqtt (via un topic diferit). Serverul va stoca mesajul (dupa decodarea si parsarea acestuia) intr-un fisier de tip „.db”. Structura fisierului se poate vedea in schema de mai jos:

Graphical user interface, text, application, email

Description automatically generated

Aceste coloane reprezinta de fapt: numele pachetului instalat, versiunea curenta, si detalii suplimentare cu privire la acesta (daca este cazul). Tot acest fir de executie este implementat in functia vm\_packages = pack.get\_vm\_packages(USER\_ID, vm\_id) din codul sursa. Din baza de date cu pachete disponibile pe VM, urmeaza sa fie emis de catre server un eveniment socketIO cu intreg continutul acesteia (ca si raspuns la request-ul initial vm\_selected primit de la client):

vm\_packages = pack.get\_vm\_packages(USER\_ID, vm\_id)

emit('vm\_packages', {'vm\_packages': vm\_packages})

Mai departe, clientul va primi lista de pachete si o va afisa (ca si continut HTML) sub forma unui tabel. Implementarea pe partea de client care asigura acest procedeu este schitata mai jos:

//wait for the server to send the packages on the VM that was previously selected

sio.on("vm\_packages", (data) => {

// tuple list which contains the packages

vm\_packages = data["vm\_packages"];

$("#vm-name-title").html(vm\_name\_list[selected\_vm\_index - 1]);

$("#vm-id-title").html(vm\_id\_list[selected\_vm\_index - 1]);

//make the table visible

$("#vm-table").css("display", "block");

for (let i = 0; i < vm\_packages.length; i++) {

$("#vm-table > tbody").append(...)

Remarcabil in versiunea curenta a serviciului este faptul ca fiecare element din tabel este selectabil, iar utilizatorul va putea vedea (per fiecare pachet) pe langa detaliile din baza de date afisata mai sus, inca doua optiuni suplimentare:

1) Verificarea existentei unei versiuni de software mai noua decat cea curenta

2) Actualizarea unui pachet software la versiunea cea mai noua disponibila din biblioteca de pachete RHEL oficiala.

Cele doua functii se asigura prin intermediul a doua coloane aditionale la tabelul ce contine lista de pachete, mai exact un buton “Check” si un buton “Update”. Prin selectarea celor doua butoane de catre utilizator, doua evenimente declansatoare socketIO pot fi transmise catre serverul principal:

//check when the user clicks on the "check-button"

$("#vm-table").on("click", "#check-update-button", (e) => {

sio.emit("check\_update", { vm\_id: vm\_id, package: package\_name });

});

//check when the user clicks on the "update-button"

$("#vm-table").on("click", "#update-button", (e) => {

sio.emit("update", { vm\_id: vm\_id, package: package\_name });

});

Ca si pas urmator, serverul va comunica prin mqtt doua posibile comenzi de tip terminal ce vor ajunge sa fie executate pe VM-ul respectiv:

def execute\_check\_update(userID, vm\_id, package\_name):

print(f'From: {userID}')

print(f'will check update for {package\_name} on VM: {vm\_id}')

# shell command to be executed on the selected VM

command = f'yum check-update | grep {package\_name}'

publish\_command(userID, vm\_id, command)

def execute\_update(userID, vm\_id, package\_name):

print(f'From: {userID}')

print(f'will update {package\_name} on VM: {vm\_id}')

# shell command to be executed on the selected VM

command = f'yum update {package\_name}'

publish\_command(userID, vm\_id, command)

unde functia „publish\_command()” este implementarea mqtt ce va transmite fie comanda „f'yum check-update | grep {package\_name}'” sau „f'yum update {package\_name}'“, inspre a fi rulate pe masina virtuala. Aceste comenzi vor verifica starea unui anume pachet sau vor porni un procedeu de actualizare tipic sistemului de operare Linux.

Intregul proces de executie si comunicare dintre partea de client si server descrisa mai sus poate fi encapsulata sub forma unei diagrame precum cea de mai jos: