**UNIVERSITATEA DIN BUCUREȘTI**

**FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ**

**LUCRARE DE DISERTAȚIE**

|  |  |
| --- | --- |
| Coordonator științific  **Conf. Dr. Cristian  KEVORCHIAN** | Absolvent  **Alexandru Mihai GEOROCEANU** |

**București   
2015**

**UNIVERSITATEA DIN BUCUREȘTI**

**FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ**

**Platformă WEB pentru Jocuri în Timp Real**

|  |  |
| --- | --- |
| Coordonator științific  **Conf. Dr. Cristian KEVORCHIAN** | Absolvent  **Alexandru Mihai GEOROCEANU** |

**București   
2015**

**DECLARAŢIE PRIVIND ORIGINALITATEA LUCRĂRII**

Prin prezenta declar că lucrarea de disertație cu titlul “*Platformă WEB pentru Jocuri în Timp Real*” este elaborată de mine şi este rezultatul propriei mele activităţi de documentare şi cercetare. Declar de asemenea că ea nu a mai fost prezentată parțial sau integral la o altă facultate sau instituţie de învăţământ superior din ţară sau străinătate.

De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare:

* toate fragmentele de text reproduse exact, chiar dacă sunt traduceri proprii din altă limbă, sunt scrise între ghilimele şi deţin referinţa precisă a sursei;
* reformularea în cuvinte proprii a textelor scrise de către alţi autori deţine referinţa precisă la sursă;
* codul sursă, imagini etc. preluate din proiecte *open*-*source* sau alte surse sunt utilizate cu respectarea drepturilor de autor şi deţin referinţe precise;
* rezumarea ideilor altor autori precizează referinţa precisă la textul original.

București *2015*

Absolvent *Alexandru Mihai GEOROCEANU*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Cuprins**

Introducere 2

CAPITOLUL 1. Dezvoltarea pentru cloud 3

1.1 Introducere în cloud computing 3

1.2 Tipurile de servicii în Cloud Computing 4

1.3 Clasificarea UCSB-IBM 4

1.4 Avantaje și dezavantaje 5

CAPITOLUL 2. Tehnologii folosite 7

2.1 JavaScript – limbajul de programare 7

2.1.1 Scurt istoric 7

2.1.2 Generalități ale limbajului 8

2.2 Node.js – serverul de aplicație 9

2.3 Redis – serverul de structuri de date 10

2.4 Nginx – serverul web (proxy, load balancer) 11

2.5 Docker – platforma pentru aplicații distribuite 12

2.5.1 Introducere 12

2.5.2 Problema mediului 13

2.5.3 Actualizarea software-ului 14

2.5.4 Eliminarea surprusului virtualizării 14

CAPITOLUL 3. Arhitectura infrastructurii 18

3.1 Prezentare de ansamblu 18

3.2 Instalarea aplicației 18

3.3 Securitatea mediului 18

3.4 Scalabilitatea aplicației 18

CAPITOLUL 4. Prezentarea aplicației 19

4.1 Arhitectura software 19

4.2 Funcționalități server 19

4.3 Funcționalități client 19

Bibliografie 20

# Introducere

Prezenta lucrare demonstrează eficacitatea noilor tehnologii web – WebSockets, JavaScript pe server – și posibilitatea facilă de a utiliza o soluție ce poate scala orizontal fie pe mașini proprii, fie pe soluții *cloud* comerciale precum Amazon Web Services, IBM SoftLayer, Microsoft Azure și altele.

S-a ales utilizarea unui mediu gol pe mașina de dezvoltare, neinstalând niciun program din cele necesare rulării aplicației – server web, de structuri de date sau server de aplicație – și s-a folosit *Docker* cu imagini oficiale atât pentru mediul local cât și de producție. În acest fel se obține rapid o suită de programe instalate independent (fiecare într-un container separat), ușor de descărcat și de actualizat. Mai mult, instalările sunt verificate și actualizate periodic de către producătorii aferenți.

Global, tot mai multe persoane au acces la Internet. Un joc online, de exemplu, poate deveni oricând *viral* și, negândind de la bun început o arhitectură ce se poate scala orizontal, sistemul se poate supra-aglomera rapid și chiar închide din cauza traficului. Soluția prezentată în această lucrare este de tip *3-tier*: unul sau mai multe servere cu rol de *proxy* și *load-balancer* (nginx), unul sau mai multe servere care rulează aplicația (node.js) și unul sau mai multe servere de structuri de date (redis) prin care se face legatura între nodurile de aplicație.

Aplicația rulează în toate navigatoarele moderne de Internet (Chrome, Safari, Firefox) folosind cea mai eficientă (și nouă) tehnologie de comunicare client-server, cu posibilitatea serverului de a trimite mesaje clienților (server push) fără supraîncărcarea generată de metodele clasice – periodic pooling – și cu un plus major de performanță, nerenunțând nici la securitate. Conexiunea folosită se numește WebSocket și este un protocol nou *full-duplex* peste o singură conexiune TCP. Aceasta este o conexiune persistentă spre deosebire de conexiunile HTTP prin care se transportă, spre exemplu, resursele statice precum fișiere JavaScript, CSS, HTML.

Arhitectura aplicației se bazează pe evenimente – coechipierul s-a conectat, a făcut o mutare, a scris un mesaj pe chat ș.a.m.d. – care sunt captate și interpretate de server și clienți. Toate acțiunile sunt validate în continuare de către serverul de aplicație, așa încât nu sunt permise mutări ilegale sau alte acțiuni prin care un jucător să poată trișa.

# Dezvoltarea pentru cloud

## Introducere în cloud computing

Începuturile serviciilor informatice au fost dominate de computere tip *mainframe* create de către IBM și serveau în principal clienților de tip corporație sau băncilor, deoarece aceștia aveau nevoie să suporte un număr mare de clienți concurenți conectați și/sau tranzacții concurente.

Potrivit începuturilor serviciilor informatice, când tehnologia costa foarte mult, un asemenea *mainframe* era deseori doar închiriat pentru niște sume exorbitante de bani, cumpărarea unui asemenea hardware fiind o opțiune chiar cu mult mai costisitoare și oricum mai puțin rentabilă deoarece progresele rapide în tehnologie ar fi făcut ca scumpa mașină să fie depășită relativ rapid.

Legea lui Moore a rămas validă de-a lungul vremii, însă performanța nu a crescut liniar în viteză a procesorului – așa cum era inițial de așteptat. În schimb, producătorii au început să creeze procesoare cu mai multe nuclee și să păstreze o frecvență relativ mică a acestora. Această nouă abordare a dus la dezvoltarea accelerată a *parallel computing*, inginerii software punând mai nou accentul pe programele ce folosesc mai multe fire de execuție în paralel. Bineînțeles, creșterea pe orizontală a puterii de procesare a implicat și posibilitatea sporita de multi-tasking, utilizatorii fiind mult mai confortabili în ziua de astăzi să navigheaza de exemplu pe Internet în timp ce pe fundal se petrece o operație de durată ce solicită intensiv procesorul.

Cu același principiu s-a trecut în ziua de astăzi la un nivel mai mare: scalarea cu totul a mașinilor fizice pe orizontală. Dacă o aplicație sau un server suportă acest tip de scalare, atunci indiferent de numărul de utilizatori, se vor putea adăuga oricâte servere de același tip, nefiind nevoie de o așteptare în creștere a performanței produselor hardware.

Primul și cel mai important actor (chiar și astăzi) pe piața distribuitorilor de servicii cloud computing este Amazon. Acesta are o filozofie „pay per usage” așa încât utilizatorul nu are nevoie de abonamente sau contracte, ci pur și simplu plătește în funcție de volumul de resurse consumate. Resursele pot fi mașini virtuale deschise în funcție de bunul plac al utilizatorului – atât ca specificații tehince (memorie RAM, procesor, spațiu pe disk) cât și servicii specializate precum baze de date preconfigurate, servere DNS, servere de email și așa mai departe.

## Tipurile de servicii în Cloud Computing

SaaS (Software as a Service) este un model de serviciu cloud computing unde a un distribuitor licențiază o aplicație către utilizatorii săi pentru a fi utilizată ca și serviciu „on demand”. Un exemplu de SaaS este aplicația CRM (content relationship management) Salesforce.com.

IaaS (Infrastructure as a Service) este oferirea unei infrastructuri computerizate (de regulă o platformă pentru virtualizare de mediu) ca și serviciu. În loc de a cumpăra servere, software, spațiu în data center sau echipament de rețea, clienții cumpără în schimb aceste resurse ca și serviciu externalizat. Un asemenea distribuitor de servicii IaaS este Amazon web services.

PaaS (Platform as a Service) este oferirea unei platforme de calcul și stivă de soluții ca serviciu. Acesta facilitează lansarea aplicațiilor fără costul și complexitatea de a cumpăra și administra straturile hardware și software din spate. PaaS oferă resursele necesare întregului ciclu de viață a dezvoltării și lansării aplicațiilor și serviciilor web. Un exemplu de PaaS este GoogleApps.

## Clasificarea UCSB-IBM

Clasificarea UCSB-IBM (Figura 1**)** a fost realizată prin efortul coroborat dintre academie (Universitatea din California, Santa Barbara) și industrie (IBM T.J. Watson Research Center) într-o încercare de a înțelege imaginea cloud computing-ului. Țelul acestui efort a fost de a facilita explorarea ariei cloud computing și de asemenea de a promova eforturile educaționale în predarea și adoptarea ariei cloud computing.

În clasificarea USCB-IBM autorii au folosit principiul de compunere de la Service-Oriented Architecture (SOA) pentru a clasifica diferitele straturi ale cloud-ului. Compunerea în SOA este modul de a coordona și asambla o colecție de servicii pentru a forma servicii compozite. În acest fel, serviciile cloud pot fi de asemenea compuse din unul sau mai multe diferite servicii cloud.

După principiul compunerii, modelul UCSB-IBM a clasificat cloud-ul în cinci straturi. Fiecare strat încapsulează unul sau mai multe servicii cloud. Serviciile cloud aparțin aceluiași strat dacă au un nivel echivalent de abstractizare, evidențiat de către utilizatorii vizați de către acestea. De exemplu, toate mediile software de cloud (cunoscute și ca platforme cloud) vizează programatorii, în timp ce aplicațiile cloud vizează utilizatorii finali. Așadar, mediile software cloud vor fi clasificate într-un alt strat decât aplicațiile cloud.

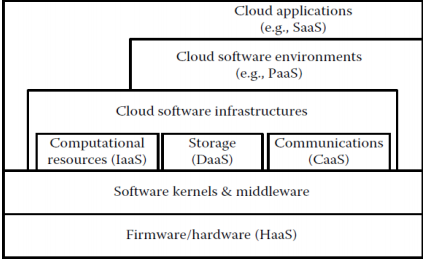


Figura : Clasificarea UCSB-IBM

## Avantaje și dezavantaje

Principalele avantaje ale cloud computing-ului:

* Prin folosirea virtualizării, calculul în cloud deschide infrastructura către un număr mare de aplicații. Într-adevăr, mașinile virtuale și sistemele de operare specializate oferă un mediu ideal pentru executarea aplicațiilor vechi. Aceste aplicații sunt de cele mai multe ori foarte sensibile în ceea ce privește mediul de execuție și nimeni nu dorește modificarea acelor coduri-sursă care funcționează și au fost validate doar pentru a le adapta pentru executarea pe o anumită platformă.
* Virtualizarea folosită de cloud oferă de asemenea un mediu configurabil și reproductibil pentru a ținti o aplicație anume, așadar utilizatorul poate folosi imediat aplicația și/sau o poate refolosi la o dată viitoare. Acesta poate fi un element important în favoarea calculului în cloud pentru un utilizator ce dorește să poată avea posibilitatea de a reproduce analiza de-a lungul timpului sau care se confruntă cu aplicații vechi, neîntreținute, care sunt greu de portat pe medii noi.

Există, desigur și dezavantaje:

* Deși producătorii de hardware și de sisteme de operare au făcut un efort imens de a îmbunătăți virtualizarea, performanţa încă depinde de modul în care hardware-ul din spate, rețeaua și mașina virtuală au fost configurate. Deși performanța procesorului virtual este apropiată de performanța procesorului fizic, performanța rețelei virtuale încă rămâne în urma capabilităților de performanță ale interconectării fizice.
* Există mai mulți actori și furnizori pe piața cloud-ului, însă nu există un standard adoptat (Docker adreseaza această problemă însă este adoptat în prezent drept tehnologie *beta*) și, chiar mai rău, unele tehnologii se bazează pe interfețe proprietare. Ca și consecință, nu există compatibilitate între furnizori diferiți. O dată ce un utilizator dezvoltă o aplicație pentru un anumit cloud, e posibil să fie nevoie de ceva efort pentru a porta acea aplicație pe un cloud diferit.
* Modelul de taxare dictează nevoia de a evalua și cuantifica nevoile computaționale, de stocare și de conectare la rețea ale unei aplicații. Aceste valori sunt adesea foarte greu de previzionat și pot rezulta costuri neanticipate.

# Tehnologii folosite

## JavaScript – limbajul de programare

### Scurt istoric

Limbajul de programare JavaScript a fost creat în 10 zile, în luna mai anul 1995 de către Brendan Eich, angajat atunci al Netscape. Numele original a fost *Mocha* ales de fondatorul Netscape. În septembrie numele i-a fost schimbat în LiveScript iar apoi, în decembrie al aceluiași an, după ce a primit o licență trademark de la Sun, a fost adoptat numele JavaScript. Aceasta a fost o mișcare de marketing la acel timp, Java devenind deja populară pe atunci.

În 1996 – 1997 JavaScript a fost dusă la ECMA (o organizație privată non profit pentru standardizare) pentru a elabora o specificație standard, pe care și alți producători de navigatoare web să o poată implementa, pornind de la dezvoltările Netscape. Așa a apărut ECMA 262 Ed. 1: ECMAScript fiind numele standardului oficial, iar JavaScript fiind cea mai bine cunoscută implementare. ActionScript 3 este o altă bine cunoscută implementare, cu extensii.

Procesul de standardizare a continuat în cicluri, cu elaborarea ECMAScript 2 în 1998 și apoi ECMAScript 3 în 1999, acesta din urmă fiind baza pentru limbajul JavaScript de astăzi.

Următoarele eveninmente notabile s-au petrecut în 2005: primul, Brendan Eich și Mozilla s-au realăturat ECMA ca și membri non-profit și s-a început lucrul la E4X, ECMA-357, care a originat de la foști angajați Microsoft la BEA. Aceasta a dus la lucrul împreună cu Macromedia, care pe atunci implementa E4X în ActionScript 3. Al doilea eveniment a fost începerea lucrului la ECMAScript 4 cu țelul de a standardiza ce era în ActionScript 3 și de a implementa noul standard în SpiderMonkey (motorul JavaScript al navigatorului Mozilla Firefox).

Din cauza agitației create de diverși actori din scena limbajului – cum ar fi Douglas Crockford – care în 2007 și-a unit forțele cu Microsoft și s-a opus ECMAScript 4, următorul standard a fost ECMAScript 3.1.

În timp ce toate aceste lucruri se petreceau, comunitatea de dezvoltare open source a început să lucreze la revoluționarizarea utilității limbajului. Acest efort al comunității a luat loc în 2005 când Jesse James Garrett a publicat un articol în care a inventat termenul Ajax și a descris un set de tehnologii, dintre care JavaScript era coloana vertebrala, folosite pentru a crea aplicații web unde datele se pot încărca pe fundal, eliminând necesitatea de a reîncărca paginile în întregime și rezultând în aplicații mai dinamice. Aceasta a dus la perioada renașterii a utilizării JavaScript, accelerată de bibliotecile open-source și de comunitățile formate în jurul acestora, cu lansarea bibliotecilor precum Prototype, jQuery, Dojo, Mootools și altele.

În iulie 2008 părțile heterogene s-au reunit la Oslo. Aceasta a dus la eventualul acord la începutul lui 2009 de a redenumi ECMAScript 3.1 în ECMAScript 5 și de a continua dezvoltarea limbajului folosind o agendă denumită Harmony.

Toate acestea ne aduc în ziua de astăzi, cu JavaScript intrând un complet nou și incintant ciclu al evoluției, inovației și standardizării, cu noutăți precum platforma Nodejs, care ne permite să folosim JavaScript pe partea de server și API-uri HTML5 pentru a controla obiecte media, WebSockets pentru comunicații în timp real, metode de a afla coordonatele geografice ale utilizatorului sau folosirea senzorilor dispozitivelor (precum accelerometrul) și multe altele.

### Generalități ale limbajului

JavaScript este un limbaj complet orientat obiect, cu natură funcțională. Are un set foarte mic de tipuri de date – trei primitive: *boolean, number, string* și valorile speciale *null* și *undefined*. Orice alt obiect este o variațiune al tipului *object*. Deși în implementările se găsesc și tipul *Date* sau *RegEx*, acestea sunt tot variațiuni ale tipului *object*. Nu există tipuri întregi, toate numerele sunt reprezentate pe 64 biți în format IEEE 754, ceea ce înseamnă că aritmetica este puțin diferită față de C sau Java, de exemplu expresia 0.1 + 0.2 === 0.3 este falsă. Rezultatul expresiei din stânga este 0,30000000000000004 și nu 0,3.

Obiectele sunt similare cu tipul dicționar din Python sau HashMap din Java; acestea sunt containere de perechi nume/valoare. Numele sunt de tip *string* (sau alte elemente precum numere care sunt convertite la *string*). Valorile pot fi de orice tip, inclusiv alte obiecte. Deși obiectele sunt implementate ca și tabele de dispersie nu sunt vizibile metode specifice – precum funcțiile de dispersie sau metode de re-dispersie. Tablourile și funcțiile sunt implementate ca și obiecte.

Declarația funcțiilor arată ca funcțiile din C cu excepția faptului că sunt declarate cu ajutorul cuvântului cheie *function* în locul tipului de date returnat. Apelul unei funcții nu obligă la introducerea unui număr fix de parametri. Parametrii în exces sunt ignorați, iar cei lipsă iau valoarea *undefined* ceea ce face ușoară scrierea de funcții cu număr arbitrar de parametri.

În majoritatea limbajelor de programare se găsesc clase și obiecte, unde clasele moștenesc alte clase. În JavaScript, moștenirea este bazată pe prototip; asta înseamnă că nu există clase, iar obiectele moștenesc alte obiecte.

## Node.js – serverul de aplicație

Node.js este o platformă construită pe motorul JavaScript Google Chrome V8 și a fost creată pentru dezvoltarea facilă de aplicații rapide și eficiente. Node.js folosește un model bazat pe evenimente, nonblocant, ceea ce îi oferă lejeritate și eficiență în folosirea resurselor.

V8 este un proiect open source create de Google și reprezintă nucleul navigatorului Google Chrome. Prima apariție publică a fost în septembrie 2008, o dată cu prima apariție a navigatorului. A reprezentat un pas mare în față cu privire la performanța navigatoarelor și a împins tehnologia acestora la un alt nivel (în momentul de față V8 este în urma motorului creat de Mozilla pentru Firefox – SpiderMonkey). Este scris în C++ și inovația sa a fost precompilarea codului sursă JavaScript în cod mașină în locul unei simple interpretări și apoi aplicarea unui proces JIT pentru a îmbunătăți dinamica execuției codului.

În anul 2009 Ryan Dahl încerca să rezolve o problemă grea în acele zile: să furnizeze navigatorului cât timp dintr-un upload a rămas. Inspirat de serverul web Ruby – Mongrel și de recenta lansare a Chrome și V8 a decis să dea o șansă JavaScript-ului, creând primele iterații ale Node.js. Proiectul a fost dezvoltat și sponsorizat de Joyent, compania unde Ryan lucrează. A delegat în 2012 poziția de supervizor unui coleg – Isaac Schlueter în ianuarie 2012.

Cu ajutorul Node.js s-a început dezvoltarea accelerată a WebSockets. Aceștia din urmă reprezintă o modalitate eficientă, rapidă și modernă de comunicare bilaterală în timp real între navigator și server. Încă dinainte de a se stabiliza standardul, Node.js având suport pentru WebSockets a dat dezvoltatorilor posibilitatea de a experimenta cu iterațiile inițiale, pregătindu-i pentru protocolul final și de asemenea lăsându-i să-și exprime opiniile despre implementările intermediare pentru a perfecționa tehnologia.

În prezent, node este îmbrățișat de dezvoltatorii din toată lumea, având o comunitate vibrantă și un manager de pachete – npm (nameless packet manager / npmjs.com). Instalarea (și compilarea, unde este cazul) de biblioteci pentru node se face de regulă cu ajutorul acestui manager de pachete, software-ul publicat fiind actualizat și întreținut. Fiecare pachet are o referință către un *repository* pe GitHub (github.com) unde se poate vedea codul sursă, lăsa comentarii, vedea problemele deschise și, de ce nu, trimiterea unui pull request pentru îmbunătățirea codului.

Există mai mult de 120.000 de pachete publicate pe npm și peste 200.000 de descărcări pe săptămână, dovadă clară a comunității dinamice și a pasiunii dezvoltatorilor pentru această arie nouă de tehnologie.

## Redis – serverul de structuri de date

Redis a pornit ca și proiect al unei singure persone însă acum are contribuitori multipli ca și proiect open-source sub licență BSD. Este scris în limbajul de programare C. Un server Redis este accesat printr-un protocol implementat în diverse biblioteci client (care trebuie aduse la zi când protocolul se schimbă). Clientul realizează distribuirea dispersată către servere. Acestea din urmă stochează datele în RAM, dar datele pot fi copiate pe disc pentru rezervă sau restartarea sistemelor. Restartarea poate fi necesară pentru adăugarea mai multor noduri.

Serverul folosește foarte multă memorie RAM și scire pe disc asincron. Acest lucru îl face nerecomandat pentru date importante deoarece există o probabilitate mare de pierdere de date în cazul căderii serverului (tensiune electrică, sursă defectuoasă ...). Pe de altă parte, acest lucru face Redis unul dintre cele mai rapide servere din domeniul său și utilizabil pentru datele comune „mai puțin importante” ce se regăsesc în aplicațiile web curente, în special în rețelele de socializare de exemplu. În rarele cazuri în care serverul chiar cade, nu va fi atât de importantă salvarea unei singure actualizări a unei postări sau o singură etichetare a unei persoane într-o singură poză. Este însă vital pentru rețeaua de socializare să distribuie o viteză de utilizare plăcută fără baze de date care să încetinească experiența.

Ca și alte servere de structuri de date, Redis implementează operații de inserare, ștergere și căutare. Include operații pe liste și pe seturi, iar operațiile de actualizare le face prin blocare, urmată de replicare asincronă. Este estimat să suporte în jur de 100.000 de operații de recuperare/actualizare pe secundă folosind un server cu 8 nuclee și 50 de clienți concurenți.

Una dintre caracteristicile impresionante este suportul pentru limbaje de programare:

|  |  |  |
| --- | --- | --- |
| * ActionScript * C, C#, C++, Clojure * Common Lisp * D, Dart * Erlang | * GNU Prolog, Go * Haskell, haXe * Java * Lua * Matlab | * Node.js * Objective-C * Perl, PHP, Python * Ruby, Rust * Scala, Scheme |

## Nginx – serverul web (proxy, load balancer)

Nginx este un server HTTP și proxy inversat, cât și server proxy de email, scris de Igor Sysoev. Servește de multă vreme site-uri rusești foarte încărcate precum Yandex, Mail.Ru, VK și Rambler. Potrivit Netcraft, nginx a fost în spatele celor mai ocupate 21.09% site-uri în ianuarie 2015. Asemenea Node.js, Nginx folosește o abordare asincronă bazată pe evenimente diferit de de modelul serverului Apache HTTP care implicit folosește o abordare bazată pe fire de execuție sau procese. Arhitectura modulară Nginx poate produce o mai bună estimare a performanței la încărcări mari.

Pentru proiectul curent am avut nevoie de un load balancer și proxy cu condiția de a avea în lista de protocoale suportate conexiunile WebSocket și implicit HTTP/1.1 Upgrade. Un plus a reprezentat și prezența suportului pentru protocolul experimental SPDY, care la standardizare va deveni HTTP/2 și va deservi în viitor drept noul protocol standard pentru servit aplicații web.

Pe partea de securitate există protocolul de securitate TLS/SSL, Nginx oferind suport pentru SNI (Server Name Indication) în cazul în care se dorește servirea mai multor domenii HTTPS pe aceeași adresă IP și OSCP stapling (extensie pentru verificarea revocării certificatelor digitale X.509). Nginx poate fi folosit și ca un proxy *SSL Termination*, în sensul că aplicația va servi doar cereri pe protocoalele http:// și ws://, însă serverul le va securiza pentru utlizatorii finali trecând la https:// și, pentru WebSocket, wss:// incluzând certificatul de securitate.

În cadrul aplicației folosite în acest proiect există biblioteca Socket.IO care necesită „sesiuni lipicioase” – cererile unui client trebuie să ajungă mereu la același server de aplicație. Nginx suportă acest tip de sesiune, folosind o funcție de dispersie în funcție de IP. Tradițional, în cazul altor proxy-uri, aceste sesiuni lipicioase erau implementate cu ajutorul unui Cookie, însă în cazul WebSocket – unde protocolul nu este încă standardizat în întregime – acest lucru poate fi periculos, deoarece navigatorul poate să implementeze diferit comportamentul.

Vital pentru arhitectura proiectului a fost și posibilitatea de a modifica fișierele de configurare în timpul rulării și reîncărcarea lor dinamic, fără a restarta serverul deoarece restartând serverul s-ar fi deconectat toate sesiunile WebSocket.

## Docker – platforma pentru aplicații distribuite

### Introducere

În mare, **Docker** funcționează ca o mașină virtuală cu o penalizare de performanță considerabil mai mică. E folosit pentru a rula aplicații într-un mediu izolat de restul sistemului. Fiecare container Docker poate rula propria distribuție Linux indiferent de distribuția de pe mașina gazdă. Mașina gazdă poate fi o mașină fizică sau un server virtual (KVM, XEN, Virtualbox ...) și pentru a instala Docker e nevoie de acces root pe respectiva mașină.

Definite printr-un singur fișier de configurare, containerele Docker pot fi refăcute de la zero în foarte scurt timp și vor funcționa identic indiferent de serverul pe care sunt refăcute. Este avantajoasă dezvoltarea aplicațiilor, local, folosind Docker pentru ca mai apoi să poată fi mutate în producție, fără surprize neprevăzute legate de instalare.

Imaginile noi pot fi create salvând modificările unei instanțe Docker cu ajutorul comenzii *docker commit*, sau prin execuția unui fișier Dockerfile. Execuția unui Dockerfile presupune pornirea unei imagini existente, executarea succesivă a unei liste de comenzi și definirea unor parametrii de execuție, totul într-un sistem automat. Un exemplu minimal de Dockerfile:

FROM ubuntu:latest   
MAINTAINER Alexandru Georoceanu <alex@navigheaza.ro>   
RUN apt-get update   
RUN apt-get -y upgrade   
ENTRYPOINT /bin/bash

Comanda *docker build -t="my-ubuntu" .* crează o imagine nouă numită my-ubuntu pornind de la ubuntu:latest și executând comenzile de update/upgrade. La execuția imaginii fără comandă explicită se va executa comanda de la ENTRYPOINT (în cazul nostru, o insanță bash).

Folosind Dockerfile se pot instala și configura automat servicii foarte complexe, inclusiv servicii distribuite și redundante pe un număr mare de instanțe.

Este important de notat faptul că instanțele Docker se opresc automat în momentul în care comanda inițială își încheie execuția. Pentru a rula servicii pe termen lung se folosește în mod uzual **supervisord**, un sistem de control al proceselor, sau pur și simplu nu se lansează serviciile în fundal, ci se lansează în *foreground*, facilitând astfel și investigarea rezultatelor cu comanda *docker logs*.

### Problema mediului

Fiecare mediu arată diferit – laptop dezvoltare 1, laptop dezvoltare 2, mediu de testare, producția, ș.a.m.d. De asemenea, fiecare dezvoltator are nevoie de lucruri diferite – de exemplu, dezvoltatorul ce lucrează la back-end are nevoie de alte unelte de dezvoltare față de cel care lucrează la partea de front-end a aplicației. Ideal, aceste medii ar trebui sa fie identice, așa încât să nu mai existe vechea scuză în cazul erorilor de aplicație „la mine funcționează”.

Problemele acestea se pot rezolva și cu ajutorul mașinilor virtuale, însă acestea din urmă prezintă alte costuri suplimentare în special în ceea ce privește performanța: au nevoie de foarte multe resurse și de timp pentru a porni sistemele de operare, au nevoie de regulă de foarte mult spațiu pe disc, partajarea de mașini virtuale nu este deloc facilă deoarece dimensiunile acestora tind să crească rapid către mai mulți gigaocteți.

Docker rezolvă problemele de performanță folosind o apariție relativ recentă în lumea Linux – containere LXC, suportul pentru acestea apărând în versiunea 3.8 de Kernel Linux. Acestea containere partajează kernel-ul existent cu mașina gazdă și rulează procese într-un mod izolat dar fără să deschidă sau să emuleze efectiv o mașină virtuală nouă cu absolut tot ceea ce presupune procesul de boot. Acest lucru scurtează pornirea „la rece” a unei aplicații complete de la câteva minute cu o mașină virtuală la câteva secunde cu containerele LXC.

Pentru a rezolva problema spațiului, Docker folosește AuFS (Advanced Multi-Layered Unification Filesystem), un sistem de fișiere format din „imagini” doar în citire, stivuite unul peste celălalt (ca în Figura 2).

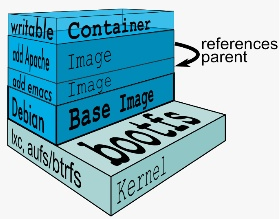


Figura : Sistemul de fișiere folosit de Docker

Spre deosebire de mașinile virtuale, o imagine se poate rula în oricâte instanțe – se va crea câte un container cu o imagine din vârf modificabilă și i se va aloca acesteia una sau mai multe adrese IP pentru accesul la rețea și comunicarea cu celelalte containere care rulează.

### Actualizarea software-ului

Pentru crearea unei imagini Docker, se poate folosi un fișier în format standard – Dockerfile. Cu comanda *docker build .* se poate crea o imagine nouă plecând de la instrucțiunile din fișierul Dockerfile din directorul curent. Aceasta se poate apoi partaja pe un repository public (hub.docker.com) sau unul privat.

Recrearea imaginii se face fie transmițând fișierul Dockerfile și rulând comanda de build, fie cu ajutorul repository-ului, de unde se pot descărca imaginile complete.

Actualizările vor conține întotdeauna doar modificările față de imaginea precedentă, astfel încât se vor face rapid și eficient. Mai mult decât atât, se poate vedea folosi comanda *docker diff* pentru a inspecta fișierele modificate în container.

Pentru cele mai populare resurse open-source (redis, ubuntu, MySQL, Postgres, CentOS, node.js și altele) există deja imagini oficiale urcate pe Docker Hub de către persoanele răspunzătoare cu mentenața proiectelor. Aceste imagini sunt actualizate periodic, create și optimizate de către dezvoltatori profesioniștii fiecărui proiect, fiind gata de a rula atât pe mediile de dezvoltare cât și în mediile productive deopotrivă.

Folosind comanda *docker pull nume-imagine* se vor căuta și descărca actualizările pentru imaginea dată ca parametru (exemple sunt redis:latest, node:0.11-slim, node:latest ș.a.). Imaginea curentă va rămâne pe disc, așa încât utilizatorul să poată rula aplicația cu noua versiune de software și, dacă întâmpină probleme sau testele automate nu validează, să rămână la imaginea cu care aplicația funcționează corect.

### Eliminarea surprusului virtualizării

Macro-benchmark-urile sunt cote de referință care se concentrează pe componente interconectate, cum ar fi infrastructura aplicației ca un întreg și nu doar executarea unui singur algoritm într-un anume limbaj de programare (micro-benchmark). Adevărata diferență dintre tehnologiile de virtualizare poate fi deseori găsită în felul în care mașinile virtuale comunică una cu cealaltă și cum încărcarea unei anumite mașini virtuale o poate influența pe alta.

Mașinile virtuale depind de infrastructura aplicației și îndeplinesc sarcina lor predefintiă. Această infrastructură poate fi construită peste un grup de mașini fizice, toate rulând sistemul de operare CoreOS sau XenServer.

Aceste implementări software „low level” sunt nucleul pe care un centru de calcul este construit. Apariția Docker a făcut folosirea containerelor LXC considerabil mai ușoară.

Pentru a compara Xen cu LXC au fost instalate XenServer 6.2 și CoreOS 324.3.0 cu Docker 0.11.1 pe două mașini identice echipate cu 4GB de RAM și un procesor Intel Xeon Quad core. Pe acestea au rulat sistemul de operare Ubuntu 12.04 în mașini virtuale sau containere.

Mașinile virtuale care au rulat XenServer folosesc un kernel paravirtualizat și au access la 2 nuclee ale procesorului. Prima mașină virtuală funcționează ca și un server de aplicație – are acces la 2GB de memorie RAM și rulează Apache 2.2, PHP 5.3 și WordPress 3.9. A doua mașină virtuală are acces la 1GB de RAM și rulează serverul de baze de date MySQL 5.5 având conținutul implicit pe care îl oferă WordPress.

Testul de performanță s-a axat pe performanța aplicației când este folosită de un număr în creștere de utilizatori. A fost folosit JMeter pentru a genera un număr mare de conexiuni simultane. Mașina gazdă a fost observată folosind top2, monitorizarea de la NewRelic și software XenServer.

|  |  |
| --- | --- |
| Figura : Numărul de cereri procesate în 800s (Mai mult înseamnă mai bine) | Figura : Progresia timpului de răspuns pentru primele 600s (mai puțin e mai bine) |
| Figura : Progresia timpului de răspuns pentru primele 800s (mai puțin e mai bine) | Figura : Timpul în ms pentru a executa o interogare SQL Select (mai puțin e mai bine) |

Testul de aplicație este un marco-benchmark. Aplicația noastră va prezenta un blog WordPress ce conținând datele implicite oferite la instalarea acestuia. Când o cerere este făcută către blog, software-ul WordPress trebuie să preia date din baza de date și să returneze un răspuns. S-a folosit JMeter pentru a trimite un număr din ce în ce mai mare de cereri concurente. Se folosește această tactică pentru a observa momentul în care serverul rămâne fără memorie și începe să aibă probleme severe de performanță.

Timp de 800 de secunde JMeter a încercat să producă cât de multe cereri a putut, cu un număr mereu în creștere de cereri concurente. Doar cererile care au reușit au fost numărate. La t = 0ms software-ul de testare a început cu o cerere concurentă și a trimis o nouă cerere după ce s-a terminat cea precedentă. Numărul de cereri concurente a fost crescut liniar, până când la 720 secunde a ajuns la 100 de conexiuni concurente.

Unul din motivele acestui benchmark a fost aflarea momentului în care performanța scade sever din cauza lipsei de resurse.

Figura 3 arată numărul de cereri reușite în 800 secunde: CoreOS a reușit să proceseze mult mai multe cereri în același interval de timp, de mai mult de patru ori decât Xen. Acest rezultat este neașteptat deoarece Sampathkumar [1] a arătat în micro-benchmarkurile făcute de el că LXC se comportă mai bine decât Xen cu 7%, nu cu 306%. Această diferență poate fi atribuită metodelor diferite în care izolarea procesorului este manevrată, unde Xen izolează per nucleu al procesorului, LXC folosește izolarea bazată pe prioritatea *cgroup*. Cu această strategie, LXC ar putea fi în stare să folosească toate resursele procesorului mai eficient și este cel mai probabil cauza acesti diferențe notabile.

Figura 5 arată că Xen are nevoie de mai mult timp pentru a răspunde la o singură cerere. Linia de tendință desenată între punctele de date ia valoarea medie de 30 de puncte date. O linie mai plată sugerează o performanță mai consistentă. În concluzie, figura arată că Xen nu se comportă la fel de consistent ca și LXC, nici chiar când are suficientă memorie fizică disponibilă.

Când numărul de conexiuni este crescut peste capabilitățle serverului în ceea ce privește memoria RAM, mașina gazdă începe să interschimbe memorie între hard disc și memoria RAM – Figura 5 arată clar acest efect. După 610 secunde (85 conexiuni concurente) Xen începe acest proces. LXC rămâne fără memorie după 655 secunde (91 conexiuni concurente), ceea ce înseamnă că surplusul de memorie introdus de Xen poate fi folosit pentru a procesa până la 6 cereri concurente în plus.

Mai mult, figura arată că LXC tratează lipsa de resurse într-un mod mai consistent. În schimb, Xen reușește să continue să servească răspunsuri mai repede și cu mai puține nereușite. Răspunsurile eșuate sunt identificate folosind coduri de status HTTP, de exemplu 502 Bad Gateway. Panta descendentă dupa 700 secunde este cauzată de cererile eșuate, deoarece graficul arată doar cererile reușite. După 707 secunde apliacația ce rulează sub LXC începe să arunce erori. Acest lucru este aștept deoarece Sampathkumar [1] a arătat deja că Xen este considerabil mai bun la izolare decât LXC, în special în situații unde resursele necesare le excedează pe cele disponibile.

Analizând mașina gazdă se poate vedea că XenServer folosește 906MB de RAM la pornire pentru a rula mașina virtuală domain0. Comparând asta cu o instalare nouă de CoreOS care folosește 161MB, diferența fiind de 745MB care ar putea fi folosită pentru servirea mai multor cereri.

Deși un țel nobil ar fi să luăm în calcul doar performanța și stabilitatea, în viața reală sunt la fel de importante ușurința de utilizare și setul de unelte disponibile. În special cu apariția noilor metodologii de dezvoltare software DevOps [2], care încurajează folosirea scripturilor și a automatizării în timpul planificării infrastructurii unei aplicații. Folosirea virtualizării și dezvoltarea API-urilor de infrastructură a înlesnit acest lucru. Atât LXC cât și Xen au diverse unelte și pot fi folosite la automatizarea lansării și creării de infrastructură, însă felul în care uneltele acestea funcționează diferă.

În Tabelul 1 este prezentată o imagine despre uneltele disponibile. Tabelul face diferența dintre CoreOS (care folosește Docker și LXC) și XenServer.

|  |  |  |
| --- | --- | --- |
|  | CoreOS | XenServer |
| Surplus înlăturat de tehnologie  Crearea de imagini  Descoperirea de servicii  Configurarea în *cluster*  Disponibilitate înaltă  Timp de pornire  Migrarea a mașinilor | +  +  +  0  +  +  - | -  0  0  +  +  0  + |

Tabelul : Comparație de flexibilitate operațională

# Arhitectura infrastructurii

## Prezentare de ansamblu

## Instalarea aplicației

## Securitatea mediului

## Scalabilitatea aplicației

# Prezentarea aplicației

## Arhitectura software

## Funcționalități server

## Funcționalități client

# Bibliografie

1. Sampathkumar. *Virtualizing intelligent river(r): A comparative study of alternative virtualization technologies*. Clemson University, 2013.
2. Michael Huttermann. *DevOps for Developers, volume 1*. Springer, 2012.