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

Cuprins

Introducere

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

Global, tot mai multe persoane au acces la Internet. Un joc online, de exemplu, poate deveni oricând *viral* și, neanticipând problemele de performanță în a servi mulți utilizatori concurenți de la bun început sau o arhitectură ce 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 dintre nodurile de aplicație.

Lucrarea prezintă o platformă gândită pentru a veni în ajutorul creării sau implementării de jocuri între doi sau mai mulți jucători umani. Aplicația implementată pe această platformă a fost Tic Tac Toe, însă abstractizarea creată în spatele platformei permite adăugarea de alte jocuri. Pentru fiecare sesiune de joc se poate introduce un nume prin care jucătorul va fi prezentat celorlalți vizitatori. În fereastra de joc se poate folosi interfața de chat pentru discuții cu ceilalți jucători conectați. Pe ecran este afișat de asemenea și jocul Tic Tac Toe cu controalele aferente.

Aplicația rulează în toate browserele moderne de Internet (Chrome, Safari, Firefox) folosind cea mai eficientă 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 *full-duplex* peste o singură conexiune TCP. Aceasta este o conexiune persistentă spre deosebire de conexiunile HTTP prin care se transportă resurse precum fișiere JavaScript, CSS, HTML.

Am ales utilizarea exclusiv a *Docker* pentru mediul de dezvoltare cât și pentru cel 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.

CAPITOLUL 1. Dezvoltarea pentru cloud

1.1 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 utilizatori concurenți conectați și/sau tranzacții concurente.

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

Legea lui Moore a rămas validă de-a lungul vremii, însă performanța procesorului nu a crescut liniar în viteză, așa cum era inițial de așteptat. În schimb, producătorii au început să creeze procesoare cu mai multe nuclee și să păstreze frecvența acestora relativ mică. 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 sporită de multi-tasking, utilizatorii fiind mult mai confortabili în ziua de astăzi să navigheze pe Internet în timp ce pe fundal se execută o operație de durată ce solicită intensiv procesorul.

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

Primul și cel mai important actor (chiar și în prezent) pe piața distribuitorilor de servicii cloud computing este Amazon. Acesta are o filozofie „pay per usage” ceea ce înseamnă că pentru a folosi aceste servicii nu este nevoie de abonament sau contract, ci pur și simplu se plătește în funcție de volumul de resurse consumate. Resursele pot fi mașini virtuale pornite în funcție de nevoile 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 ș.a.m.d.

1.2 Tipurile de servicii în Cloud Computing

SaaS (Software as a Service) este un model de serviciu cloud computing unde 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 un tip de serviciu care oferă o infrastructură computerizată (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 aceste resurse ca și serviciu externalizat. Un asemenea distribuitor de servicii IaaS este Amazon Web Services.

PaaS (Platform as a Service) oferă o platformă de calcul și stivă de soluții ca serviciu. Acesta facilitează lansarea aplicațiilor fără costul și complexitatea de a cumpăra și administra componentele 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.

1.3 Clasificarea UCSB-IBM

Clasificarea UCSB-IBM (Figura 1**)** a fost realizată prin colaborarea între mediul academic (Universitatea din California, Santa Barbara) și industrie (IBM T.J. Watson Research Center) într-o încercare de a avea o imagine asupra cloud computing-ului. Țelul acestui efort a fost de a facilita explorarea ariei cloud computing și de asemenea de a promova educarea și adoptarea acestei noi arii.

În clasificarea UCSB-IBM autorii au folosit principiul de compunere din SOA (Service Oriented Architecture) pentru a clasifica diferitele componente ale cloud-ului. Principiul de construire a serviciilor, compunerea, reprezintă modul de a coordona și asambla o colecție de servicii pentru a forma servicii compozite. În acest fel, serviciile cloud pot fi compuse din unul sau mai multe servicii diferite de 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.

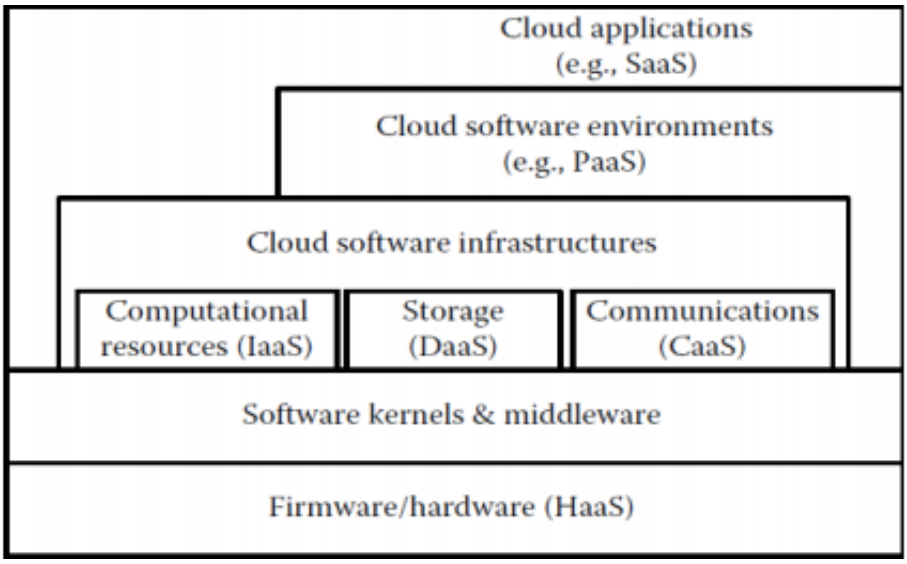

Figura 1: Clasificarea UCSB-IBM

1.4 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 acelui cod sursă care funcționează și a fost validat doar pentru a-l 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.

CAPITOLUL 2. Tehnologii folosite

2.1 JavaScript – limbajul de programare

2.1.1 Scurt istoric

Limbajul de programare JavaScript a fost creat în 10 zile, în luna mai anul 1995 de către Brendan Eich, angajat la acea vreme la Netscape. Numele original, *Mocha*, a fost ales de către 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 browsere să o poată implementa, pornind de la dezvoltările Netscape. Așa a apărut ECMA 262 Ed. 1: ECMAScript – numele standardului oficial, iar JavaScript fiind cea mai bine cunoscută implementare. ActionScript 3 este o altă bine cunoscută implementare, cu extensiile aferente, folosită la început în special pentru applet-urile Flash. La data curentă, ActionScript este folosit cu Adobe AIR (Adobe Integrated Runtime) ceea ce permite aplicațiilor să ruleze ca și aplicații native sub Microsoft Windows, Apple OS X, Google Android și Apple iOS.

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 la data scrierii acestui document.

Următoarele evenimente notabile s-au petrecut în 2005: în 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 colaborarea 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 browserului 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 timpul acestor evenimente, comunitatea de dezvoltare open source a început să lucreze la revoluționarizarea utilității limbajului. Rezultatul efortului comunității a fost prezentat 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 eterogene 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 un proiect denumit Harmony.

Toate acestea ne aduc în ziua de astăzi, cu JavaScript intrând într-un complet nou și incitant ciclu al evoluției, inovației și standardizării, cu noutăți precum platforma Node.js, care permite folosirea 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.

2.1.2 Generalități ale limbajului

JavaScript este un limbaj complet orientat obiect, cu natură funcțională. Acesta 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*, care este reprezentat printr-un container de perechi cheie-valoare. Deși în implementările uzuale se regăsesc și tipurile *Date* și *RegEx*, acestea sunt de asemenea 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` fiind falsă (rezultatul expresiei din stânga este exact 0,30000000000000004 și nu 0,3).

Obiectele sunt similare cu tipul dict din Python sau HashMap din Java și conțin perechi cheie-valoare. Cheile 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 acestor tabele – precum funcțiile de dispersie sau metode de re-dispersie. Tablourile și funcțiile sunt implementate ca și obiecte.

Declararea funcțiilor arată similar declarării funcțiilor în limbajul 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. O importantă diferență față de limbajele clasice de programare este scopul variabilelor: în JavaScript, scopul unei variabile este dat de funcția în care aceasta apare, în contrast cu Java sau C, în care scopul variabilei este dat de blocul de cod în care este declarată. Compilatorul are un comportament de „hoisting”, adică mută declarațiile variabilelor la începutul funcției din care fac parte, indiferent de locul din cod în care dezvoltatorul a scris declarația.

În majoritatea limbajelor de programare se găsesc clase și obiecte, unde clasele pot moșteni alte clase. În JavaScript, moștenirea este bazată pe prototip, ceea ce înseamnă că nu există clase, iar obiectele moștenesc alte obiecte. Moștenirea prototipală are o paradigmă diferită față de moștenirea ce folosește clase și trebuie înțeleasă pentru a scrie corect aplicații. În cazul în care se dorește folosirea unei moșteniri bazată pe clase, se poate simula comportamentul moștenirii prin clase cu ajutorul unor metode ce prelucreaza atributul prototip.

2.2 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, care îi oferă flexibilitate și eficiență în folosirea resurselor, facându-l ideal pentru proiecte ce implică comunicații în timp real.

V8 este un proiect open source creat de Google și reprezintă nucleul browserului Google Chrome [7]. Prima apariție publică a fost în septembrie 2008, odată cu lansarea primei versiuni a browserului. Proiectul V8 a reprezentat un pas mare în față cu privire la performanța browserelor și a împins tehnologia acestora la un alt nivel (în momentul de față

V8 este mai puțin performant față de motorul creat de Mozilla pentru Firefox, *SpiderMonkey*). Motorul V8 este scris în C++ și inovația pe care a adus-o a fost introducerea precompilării codului sursă JavaScript în cod mașină (inițial motoarele doar interpretau codul, fără precompilare) și apoi aplicarea unui proces JIT (Just In Time) pentru a îmbunătăți dinamica execuției codului.

În anul 2009 Ryan Dahl încerca să rezolve o problemă grea în acelei perioade: să furnizeze browserului informații despre timpul rămas până la terminarea operației de upload. Inspirat de serverul web Ruby denumit *Mongrel* și de recenta lansare a Chrome cu V8, a decis să dea o șansă JavaScript-ului, creând primele iterații ale Node.js. Proiectul a fost dezvoltat și sponsorizat de Joyent, compania a cărui angajat este Ryan Dahl. În ianuarie 2012 poziția de supervizor al Node.js a fost delegată unui coleg, Isaac Schlueter.

Cu ajutorul Node.js s-a început dezvoltarea accelerată a WebSockets. Aceștia reprezintă o modalitate eficientă, rapidă și modernă de comunicare bidirecțională în timp real între browser și server. Iterațiile inițiale ale Node.js au fost o oportunitate pentru dezvoltatori de a experimenta cu WebSockets și de se pregăti pentru protocolul final. Totodată, le-au fost permise sugestiile timpurii pentru perfecționarea acestei tehnologii noi.

În prezent, Node.js este adoptat de dezvoltatorii din toată lumea, având o comunitate numeroasă, activă și un manager de pachete – npm (nameless packet manager / npmjs.com) – unde aceștia să poată partaja bibliotecile sau aplicațiile dezvoltate. Instalarea (și compilarea, unde este cazul) de biblioteci pentru Node.js 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 și problemele deschise. Orice dezvoltator poate aduce contribuții la proiectele găzduite pe *npm* prin trimiterea unui *pull request*. Aceasta este o acțiune specifică GitHub în care dezvoltatorul trimite o cerere de *merge* a unui *commit* creat de el. Pentru a crea legătura către acel *commit*, dezvoltatorul care dorește să contribuie își creează o clonă a *repository*-ului, și apoi publică pe GitHub *commit*-ul în care se află modificările sale. Modificările sunt revizuite apoi de către contribuitorii direcți ai *repository*-ului original și, dacă aceștia sunt de acord, le integrează în proiect.

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.

Suportul nu este restrâns doar la anumite platforme – Node.js se poate instala pe toate sistemele de operare majore: Linux, Apple OS X, Microsoft Windows (în urma colaborării proiectului cu Microsoft în iulie 2011), IBM AIX, BSD și Solaris.

Node.js combinat cu un browser, o bază de date de documente (precum MongoDB sau CouchDB) și JSON (JavaScript Object Notation – format pentru schimb de date „descoperit" de Douglas Crockoford) oferă un mediu complet de dezvoltare în JavaScript. Folosind biblioteci pentru clientul din browser și adaptări de tipare de dezvoltare precum MVC, MVP, MVVM, Node.js oferă posibilitatea reutilizării modelelor de date și interfețelor de servicii de pe client pe partea de server.

Numeroase companii au început să folosească această tehnologie în ultimii ani, demonstrând fiabilitatea acesteia chiar și în domeniul financiar, cel mai elocvent exemplu fiind PayPal. Această corporație care este un proxy pentru plăți online a reușit să rescrie o aplicație Java în jumătate din timpul original necesar și cu mai puțini dezvoltatori. Mai mult, noua versiune scrisă în Node.js a mărit performanța, dublând capacitatea numărului de cereri pe secundă și scăzând timpul de răspuns per cerere cu 35%. Compania a continuat rescrierea aplicațiilor existente, ajungând la 12 aplicații după șase luni.

2.3 Redis – serverul de structuri de date

2.3.1 Generalități

Redis a apărut din necesitatea de a procesa în timp real date cu o bază de date SQL care folosește o stocare pe disc. Serverul SQL nu a putut face față la volumul mare de date din cauza operațiilor intense de scriere. Problema întâmpinată de Salvatore Sanfilippo – creatorul Redis – a fost la procesarea de date legate de analiză web: numărul în timp real de vizualizări pe secundă ale unor site-uri, care trebuia prezentat utilizatorilor conectați la aplicația sa și apoi salvate temporar (se dorea un istoric detaliat doar pentru ultimele minute).

Încercând mai multe baze de date SQL și diverse arhitecturi de baze de date, Salvatore nu a gasit o soluție pentru a procesa câteva mii de pagini pe secundă din cauza resurselor hardware scăzute de care dispunea. Deoarece nu dispunea nici de fondurile financiare pentru a cumpăra hardware mai performant, dezvoltatorul a decis să creeze un prototip de bază de date cu stocare în memoria RAM, care să suporte multe operații de adăugare și ștergere concomitente într-o listă cu două capete.

Deși Redis a pornit ca și proiect al unei singure persone, acesta are acum 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 actualizate 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 la restartarea sistemelor. Restartarea poate fi necesară pentru adăugarea mai multor noduri.

2.3.2 Tipuri de date și operații

Redis nu reprezintă o stocare simplă cheie-valoare, ci este un server de structuri de date, oferind mai multe tipuri de date ale valorilor, nu doar șiruri de caractere. Tipurile de valori ce se pot folosi sunt următoarele:

- Șiruri de caractere
- Liste: colecții de șiruri de caractere ordonate în funcție de ordinea în care au fost introduse (liste înlănțuite)
- Seturi: colecții de șiruri de caractere unice, neordonate
- Seturi ordonate: fiecărui șir de caractere îi este asociat un număr fracționar, denumit *scor*. Elementele sunt întotdeauna accesate în funcție de scor, ceea ce duce la posibilitatea de a accesa o colecție de elemente (de exemplu accesarea primelor 10 elemente sau ultimelor 10 elemente din set)
- Tabele de dispersie: câmpuri cheie-valoare similare cu *dict* din Python
- Tablouri de biți: se pot folosi operații precum numărarea biților de 1, căutarea primului bit nenul sau nul ș.a.m.d.
- HyperLogLogs: o structură de date probabilistică, folosită pentru estimarea cardinalității unui set

Cheile Redis sunt accesate și setate folosind metode sigure binar (verificate să nu încerce a folosi mai multă memorie decât au alocată, verificare de securitate), ceea ce înseamnă că valoarea unei cheie poate fi orice valoare binară, de la un șir de caractere la conținutul unei poze JPEG. Șirul vid de caractere este de asemenea o cheie validă, iar dimensiunea maximă pe care o poate avea o cheie este de 512MB.

Cheile foarte lungi sunt însă nerecomandate, deoarece pentru o cheie de 1024 biți costul căutării cheii în setul de date poate necesita mai multe comparații costisitoare. Este recomandată folosirea unei funcții de dispersie în cazul cheilor mai lungi, spre exemplu SHA1 sau SHA2.

Cele mai importante operații sunt *GET* – accesarea unui scalar, INCR – incrementarea scalarului cu 1, și *INCRBY* – similar cu INCR, însă primește un parametru pentru a incrementa cu un număr întreg arbitrar. Există de asemenea funcțiile analoage *DECR* și *DECRBY*. În cazul în care cheia pe care se face operația nu există, este creată automat.

Pentru operațiile pe liste se pot folosi funcțiile *RPUSH* – pentru adăugarea la sfârșitul listei de valori, *RPOP* – pentru ștergerea și accesarea ultimului element din listă împreună cu funcțiile analoage *LPUSH* și *LPOP* care acționează la capătul din stânga (de început) al listei. Pentru accesarea unui grup de valori din listă se poate folosi *LRANGE* care primește pe lângă parametrul cheie încă doi: ordinalul de start și de stop.

În cazul în care se dorește efectuarea mai multor operații în mod secvențial, se poate folosi comanda *MULTI* care adaugă comenzile într-o coadă de execuție, iar la apelul comenzii *EXEC* le execută pe toate secvențial. Redis nu suportă operații de „rollback" precum serverele de baze de date clasice, atât din motive de performanță cât și din cauza faptului că o comandă eșuează doar în cazul în care a fost folosită o sintaxă greșită. Așadar există posibilitatea de a programa o aplicație în care comenzile să nu eșueze din cauza Redis.

O comandă utilă în practică este *EXPIRE*, aceasta oferind un mecanism automat de ștergere a datelor după o anumită perioadă de timp. De exemplu, pentru a păstra un istoric al cheii *joc:1234* pentru o perioadă de o zi (86400 secunde), vom folosi (doar după crearea în prealabil a cheii) comanda *EXPIRE joc:1234 86400*, iar Redis va șterge cheia după expirarea acestei perioade de timp cu o eroare între 0 și 1 milisecunde. Ștergerea unei chei temporare se face fie în mod activ – când un utilizator încearcă să o acceseze și aceasta deja a expirat, fie în mod pasiv iterând următorii pași de 10 ori pe secundă:

1. Se testează 100 chei arbitrare din setul de chei temporare
2. Se șterg toate cheile găsite expirate
3. Dacă mai mult de 25 chei erau expirate, se pornește din nou pasul 1.

Acesta este un algoritm probabilistic trivial, în care presupunerea este că setul de date extras este reprezentativ pentru tot spațiul cheilor și se continuă procesul de ștergere până

când probabilitatea existenței de chei pentru ștergere este mai mică de 25%. Cu alte cuvinte, în orice moment, numărul de chei ce trebuie șterse este mai mic sau egal cu numărul maxim de operații de scriere împărțit la 4.

2.3.3 Utilizarea practică

Serverul folosește foarte multă memorie RAM și scrie pe disc asincron. Acest lucru nu îl face recomandat pentru date importante deoarece există o probabilitate mare de pierdere de date în cazul căderii serverului (tensiune electrică, sursă defectuoasă, probleme de rețea etc.). Pe de altă parte, acest lucru face ca Redis să fie unul dintre cele mai rapide servere din domeniul său și utilizabil pentru datele comune „mai puțin importante” care se regăsesc în aplicațiile web curente, în special în rețelele de socializare. În rarele cazuri în care serverul cade, nu va fi atât de importantă salvarea unei postări sau etichetarea unei persoane într-o poză. Este însă vital pentru rețeaua de socializare să asigure o viteză de utilizare plăcută fără ca experiența să fie încetinită de operațiuni pe baze de date tradiționale.

Ca și alte servere de structuri de date, Redis implementează operații de inserare, ștergere și căutare. Include operații pe liste și seturi, iar operațiile de actualizare sunt blocante, modificările fiind replicate pe celelalte noduri Redis în mod asincron. 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 ale Redis este suportul pentru limbaje de programare, precum următoarele:

- 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

2.4 Nginx – serverul web (proxy, load balancer)

2.4.1 Generalități

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, diferită 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, unde traficul servit începe să împingă resursele utilizate peste 85%.

Pentru proiectul curent am avut nevoie de server care să îndeplinească atribuțiile unui load balancer și ale unui proxy. Mai mult, pentru a asigura cea mai eficientă transmisie de date, serverul trebuie trebuie să poată prelucra 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 putem folosi 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 serverul de aplicație va servi doar cereri pe protocoalele http:// și ws://, însă serverul web 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 *sticky sessions* – 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, acest tip de sesiuni era implementat cu ajutorul unui Cookie, însă în cazul WebSocket – unde protocolul nu este încă standardizat în întregime – folosirea Cookies poate fi periculoasă, deoarece browserul poate să implementeze diferit comportamentul WebSocket.

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.

2.4.2 Arhitectura

Nginx folosește un singur proces „master” și mai multe procese „worker”, fiecare dintre acestea rulând cu un singur fir de execuție (single threaded) și fiind proiectate să manevreze mii de conexiuni simultan. Pentru a răspunde rapid la cereri, serverul se folosește de mecanismul pentru evenimente al sistemului de operare.

Procesul „worker” al Nginx este executat într-o buclă strânsă de evenimente pentru a manevra conexiunile sosite. Fiecare modul Nginx este inclus în procesul „worker”, ceea ce rezultă în faptul că oricare dintre operațiile de procesare a cererii, filtrare, manevrare ale conexiunilor proxy și multe altele sunt executate în procesul „worker”. Datorită acestei arhitecturi, sistemul de operare poate manevra fiecare proces separat și planifica procesul să fie executat optim pe fiecare nucleu al procesorului. Dacă există vreun proces care ar bloca un „worker”, cum ar fi de exemplu o operație pe disc, se poate configura folosirea mai multor asemenea procese „worker” decât nuclee de procesor pentru a manevra încărcarea.

Există de asemenea un număr mic de procese ajutătoare pe care procesul Nginx „master” le creează pentru a manevra activități dedicate. Printre acestea se numără procesul care încarcă cache-ul și cel care manevrează cache-ul. Procesul care încarcă cache-ul este responsabil pentru pregătirea metadatelor pentru ca procesele „worker” să îl poată folosi. Procesul care manevrează cache-ul este responsabil pentru verificarea elementelor din cache și expirarea celor invalide.

Procesul „master” furnizează fundația peste care fiecare modul își poate efectua scopul. Fiecare protocol și handler este implementat în propriul modul. Modulele sunt legate între ele în lanț pentru a manevra conexiuni și procesa cereri. După ce o cerere este manevrată aceasta este apoi trecută printr-o serie de filtre prin care răspunsul este procesat. Unul dintre aceste filtre este responsabil pentru procesarea sub-cererilor, una dintre cele mai puternice facilități ale Nginx.

Sub-cererile sunt modul în care Nginx poate returna rezultatele unei cereri care diferă de resursa (URI) pe care clientul a trimis-o. În funcție de configurație, acestea pot fi imbricate și accesa alte sub-cereri. Filtrele pot colecta răspunsurile mai multor sub-cereri și le pot grupa

pe acestea într-un singur răspuns pentru client. Răspunsul este apoi finalizat și trimis către client.

2.5 Docker – platforma pentru aplicații distribuite

2.5.1 Introducere

În mare, **Docker [8]** 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. Aceasta din urmă poate fi o mașină fizică sau un server virtual (KVM, XEN, Virtualbox ș.a.m.d.) 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 timp foarte scurt 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 la 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 *background*, ci se lansează în *foreground*, facilitând astfel și investigarea rezultatelor cu comanda `docker logs`.

2.5.2 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, astfel încât să nu mai existe vechea scuză în cazul erorilor de aplicație „pe calculatorul meu funcționează corect”.

Problemele acestea se pot rezolva și cu ajutorul mașinilor virtuale, însă acestea 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, ocupă foarte mult spațiu pe disc. De asemenea, 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 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 unei aplicații complete (inclusiv a sistemului de operare) 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).

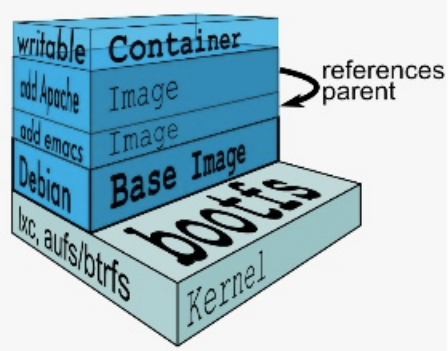

Figura 2: Sistemul de fișiere folosit de Docker

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.

2.5.3 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 mentenanța proiectelor. Aceste imagini sunt actualizate periodic, create și optimizate de către dezvoltatorii fiecărui proiect, fiind gata de a rula atât pe mediile de dezvoltare cât și în mediile de producție.

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, astfel încât utilizatorul să poată rula aplicația cu noua versiune de software și, dacă întâmpină probleme sau nu este validată de testele

automate, să rămână la imaginea cu care aplicația funcționează corect până la rezolvarea problemelor pe noua platformă.

2.5.4 Eliminarea surplusului 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 rolul predefinit. 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 a 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 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 a vizat comportamentul aplicației când aceasta este folosită de un număr din ce în ce mai mare 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.

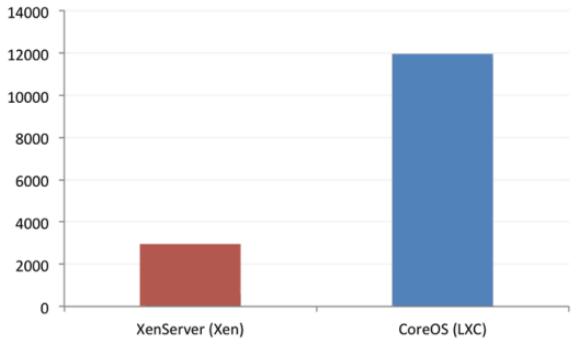

Figura 3: Numărul de cereri procesate în 800s (Mai mult înseamnă mai bine)

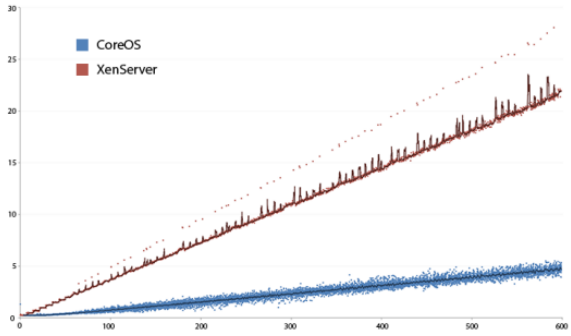

Figura 4: Progresia timpului de răspuns pentru primele 600s (mai puțin e mai bine)

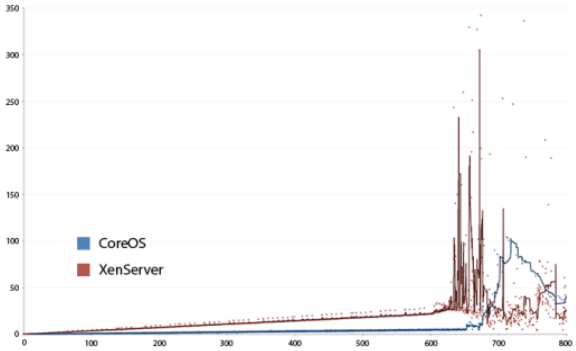

Figura 5: Progresia timpului de răspuns pentru primele 800s (mai puțin e mai bine)

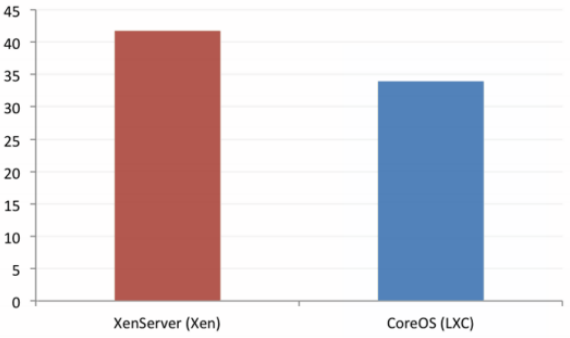

Figura 6: Timpul în ms pentru a executa o interogare SQL Select (mai puțin e mai bine)

Testul de aplicație este un marco-benchmark efectuat folosind un blog WordPress ce conține 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 0 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 0 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 cu o instalare nouă de CoreOS care folosește 161MB, diferența este de 745MB și aceasta 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	+	0
Descoperirea de servicii	+	0
Configurarea în *cluster*	0	+
Disponibilitate înaltă	+	+
Timp de pornire	+	0
Migrarea a mașinilor	-	+

Tabelul 1: Comparație de flexibilitate operațională

CAPITOLUL 3. Arhitectura infrastructurii

3.1 Prezentare de ansamblu

Infrastructura aplicației a fost proiectată pentru a oferi maxim de performanță, scalabilitate și siguranță pentru vizitatori. Accesarea aplicației implică folosirea unui browser, preferabil unul cu suport pentru noua tehnologie WebSocket, pentru a se bucura de performanță și consum redus de trafic între client și server.

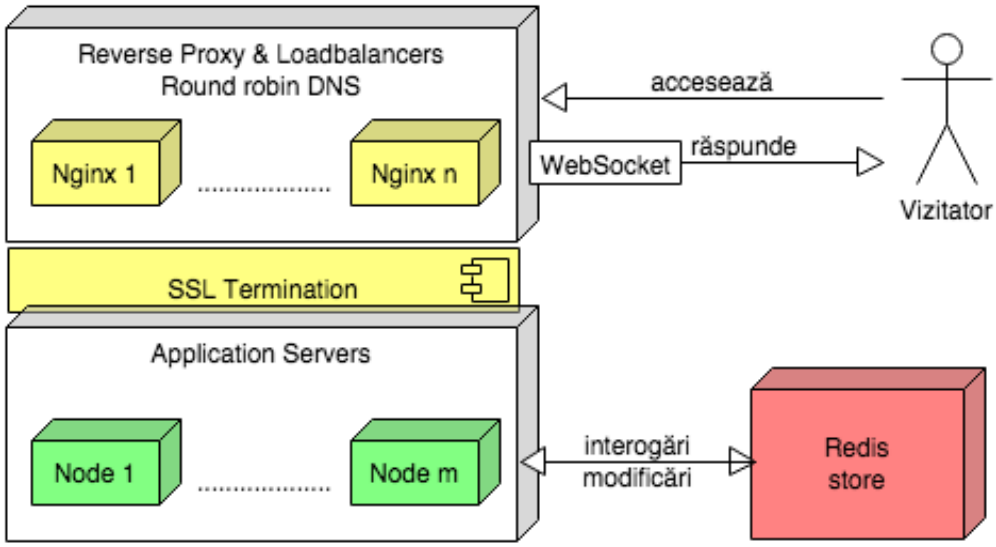

Figura 7: Prezentare de ansamblu a arhitecturii

În primă fază, cererea vizitatorului către domeniul pe care este găzduită aplicația (domeniu.com de exemplu) începe printr-o interogare DNS a nameserverelor domeniului. Pentru a folosi sistemul round-robin DNS, se pot introduce mai multe înregistrări de tip A, fiecare conținând adresa IP a unuia dintre serverele Nginx ale aplicației.

Browserul alege aleator una dintre adresele IP și creează o conexiune către serverul web Nginx. Dacă cererea a fost făcută prin protocolul HTTP, atunci nu se transmit alte date în afară de cele necesare instrucțiunii de redirecționare către versiunea securizată prin protocolul HTTPS. În cazul în care browserul cunoaște și protocolul experimental SPDY, acest protocol este folosit în locul protocolului HTTPS, deoarece este mai rapid (urmărește să devină standardul HTTP/2).

Serverul web este de asemenea un SSL Termination Proxy, decriptând traficul venit dinspre utilizatori spre serverele de aplicație și invers, criptând la retur datele trimise de serverele de aplicație către utilizatori. Acest lucru nu doar ușurează administrarea certificatelor de securitate, ci și îmbunătățește modularitatea infrastructurii, separând două elemente cu roluri diferite. Singurele două porturi publice, expuse la Internet sunt porturile 80 (implicit pentru HTTP) și 443 (implicit pentru HTTPS/SPDY).

Mai departe, traficul de date generat de cererea vizitatorului se mută în rețeaua privată a Docker, pe un server Node.js ales cu ajutorul unei funcții de dispersie a serverului Nginx. Funcția de dispersie garantează faptul că unui vizitator îi va fi desemnat întotdeauna același server de aplicație, algoritmul luând în calcul adresa IP a acestuia. Toate serverele Node.js expun aplicația pe portul 3000, însă au adrese IP private diferite.

Tot în rețeaua privată Docker se află și serverul Redis, locul central în care se persistă datele aplicației. Acesta rulează pe portul 6379, fără autentificare și fără criptare (fiind parte dintr-o rețea izolată de exterior, am ales configurația cea mai performantă).

3.2 Instalarea și rularea aplicației

Codul sursă al aplicației este menținut într-un *repository* Git public disponibil la adresa git@github.com:toxik/disertatie.git. Pentru a instala aplicația este nevoie doar de un mediu (calculator personal sau server) care are deja instalate pachetele Git și Docker. Apoi, după executarea comenzilor de mai jos, se va obține rapid un mediu complet funcțional:

```
# descărcarea imaginilor Node.js, Redis și Nginx
docker pull node:slim && docker pull redis && docker pull nginx
# descărcarea surselor aplicației
git clone git@github.com:toxik/disertatie.git && cd disertatie
# instalarea și compilarea bibliotecilor node
docker run -v $(pwd)/src:/usr/src/myapp -w /usr/src/myapp node:slim npm install
```

Pentru a porni aplicația este suficientă rularea unei singure instanțe Redis împreună cu o instanță a serverului Node.js:

```
# pornirea serverului Redis (cu persistență pe disc)
docker run --name redis -v $(pwd)/redis:/data -d redis redis-server
# adăugarea unui server de aplicație legat de containerul redis
docker run -it -d -v $(pwd)/src:/usr/src/myapp --link redis:redis -w \
      /usr/src/myapp node:slim node socket.js
```

Serverul web Nginx este pornit pentru a adăuga facilitățile de securitate și de asemenea pentru a fi proxy în cazul în care pornim mai multe instanțe ale serverului Node.js:

```
docker run --name proxy -v $(pwd)/certs:/etc/nginx/certs -v \
      $(pwd)/nginx/conf.d:/etc/nginx/conf.d -p 80:80 -p 443:443 -d nginx
```

Aplicația funcționează, desigur, și dacă pachetele sunt instalate pe mașina locală. Această metodă este însă nerecomandată, fiind mai anevoioasă: implică instalarea serverului Redis, instalarea Node.js cu dependențele sale și a serverului Nginx care, pe site-ul oficial, are disponibile pachetele de instalare doar pentru Linux și Windows (pentru Apple OS X de exemplu, metoda de instalare implică instalarea în prealabil a unui manager de pachete neoficial care apoi să compileze pachetul de instalare Nginx cu ajutorul unei rețete dezvoltate de comunitate).

3.3 Securitatea mediului

Am acordat o atenție deosebită securității aplicației și a mediului deoarece în ultima perioadă tot mai multe conturi au fost sparte, în mare parte din cauza implementării sau configurării defectuoase a unor mecanisme de securitate sigure.

Implicit, protocoalele HTTP și WS (WebSocket) nu sunt securizate, unui atacator fiindu-i foarte ușor să intercepteze mesajele transmise către și dinspre calculatorul unei victime. Social Engineering este metoda prin care atacatorii află informații private despre victime și le folosesc apoi în mod abuziv în conjuncție cu mecanismele pentru recuperearea accesului la serviciile de email, bancare și altele. Marea majoritate a persoanelor nu au o conduită strictă pe Internet, așadar devine foarte importantă securitatea conversațiilor pe care aceștia le poartă în spațiul virtual, mai ales că atacatorii de cele mai multe ori impersonează persoane cunoscute în vederea convingerii victimelor, folosind informații obținute ilegal prin spionarea mesajelor acestora.

Aplicația refuză comunicarea printr-un protocol nesecurizat, în cazul HTTP folosind o redirecționare către HTTPS, iar pentru protocolul WebSocket (ws://) se foloseşte extensia protocolului ce beneficiază de criptare, wss://. Ambele protocoale folosesc în vederea criptării și identificării același certificat, în cazul curent emis de către COMODO, un distribuitor de certificate digitale de încredere.

La data scrierii documentului, testul de securitate de pe site-ul Qualys SSL Labs a indicat scorul A+ (cel mai înalt) pentru configurația serverului a cărui subiect este lucrarea de față (Figura 8).

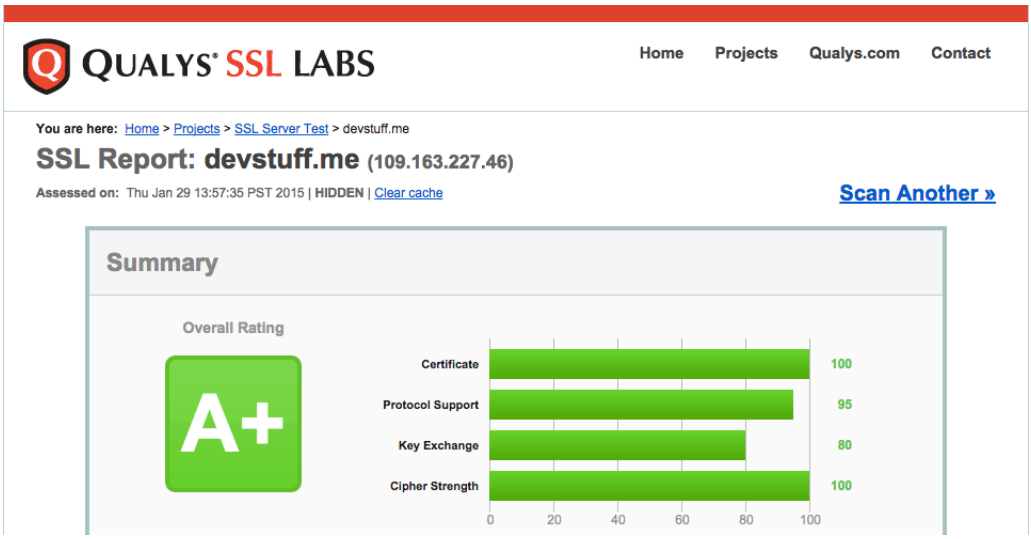

Figura 8: Scorul A+ al configurației de securitate

Certificatul a fost semnat cu o cheie RSA 2048 și folosește algoritmul SHA256 (256 biți) pentru semnătură în vederea păstrării proprietăților de securitate mai mult timp. Certificatele care folosesc semnătura RC4 (40 biți) sunt deja considerate nesigure și pot fi sparte relativ rapid cu uneltele potrivite.

Protocoalele suportate sunt TLS 1.0, TLS 1.1, TLS 1.2. Am renunțat la suportul pentru SSL 2 și SSL 3 deoarece acestea sunt considerate depășite și s-au găsit vulnerabilități care cu greu se pot rezolva. Cea mai bună cale de protecție este de a renunța la aceste ultime protocoale complet, deși asta înseamnă că unii clienți nu vor putea accesa site-ul, precum Internet Explorer 6, 8 sub Windows XP.

Serverul nu este vulnerabil la atacul POODLE (Padding Oracle On Downgraded Legacy Encyption), o exploatare *man-in-the-middle* care încearcă să retrogradeze conexiunea la SSL 3 (CVE-2014-3566). Acest atac a fost la scurt timp implementat și pe protocolul TLS (CVE-2014-8730), ceea ce a condus analiștii de securitate la căutarea unei modalități de prevenire a modului de atac, nu a protocolului. Soluția a fost implementarea TLS_FALLBACK_SCSV (Fallback Signaling Cipher Suite Value).

Atacul Heartbleed (CVE-2014-0160) a fost datorat unui bug în implementarea OpenSSL a extensiei heartbeat (RFC6520) pentru protocoalele TLS/DTLS. Când este exploatat duce la o scăpare a conținutului memoriei de la server către client și de la client către server. Acest atac a expus multe chei private, compromițând certificatele serverelor atacate. Soluția recuperării după acest atac nu este doar înlocuirea software-ului vulnerabil ci

trebuie făcută revocarea certificatului compromis (ceea ce de regulă nu este un proces ușor și nici gratuit) și recrearea cheii private împreună cu generarea unui nou certificat.

Configurația curentă nu este vulnerabilă nici la atacul CCS Injection (CVE-2014-0224) în care biblioteca OpenSSL nu restricționează corect procesarea mesajelor *ChangeCipherSpec*, astfel permițând atacatorilor *man-in-the-middle* folosirea unei chei „master" de dimensiune zero în anumite comunicații OpenSSL-OpenSSL și oferindu-le acestora acces la informații private printr-un *handshake* TLS special construit.

Forward secrecy sau *key erasure* este proprietatea protocoalelor de schimb de chei de a se asigura că o cheie de sesiune derivată dintr-un set de chei pe termen lung nu poate fi compromisă dacă una dintre cheile pe termen lung este compromisă în viitor. Cheia folosită pentru a proteja transmisia de date nu trebuie sa fie folosită în vederea derivării nici unei alte chei, iar dacă cheia folosită pentru a proteja transmisiunea de date este derivată din alte date legate de chei, atunci acele date nu trebuie să fie folosite la alte chei. În acest fel, compromiterea unei singure chei permite acesul doar la datele protejate de acea cheie.

Proprietatea *Perfect Forward Secrecy (PFS)* este implementată când sistemul:

- generează chei publice aleatoare per sesiune în privința agreerii de chei și
- nu folosește nici un fel de algoritm determinst în generarea de mai sus.

Această proprietate asigură că un singur mesaj compromis nu poate duce la compromiterea altora și de asemenea asigură că nu există o singură valoare secretă care poate duce la compromiterea mesajelor. Configurația actuală a serverului implementează proprietatea PFS cu majoritatea browserelor.

OSCP Stapling este o abordare alternativă la OCSP (Online Certificate Status Protocol) pentru verificarea stării revocării certificatelor digitale X.509. Această extensie îi permite serverului să preia costurile resurselor folosite în vederea prezentării răspunsurilor OSCP, în loc de a apela direct autoritatea care a emis certificatele (CA). Serverul aplicației este configurat să implementeze și OSCP Stapling.

HTTP Strict Transport Security (HSTS) este o îmbunătățire a securității specificată de aplicația web prin folosirea unui header special de răspuns. Când un browser care implementează HSTS primește acest header, acesta va preveni toate comunicațiile trimise nesecurizat și va redirecționa toate comunicațiile spre HTTPS. În configurația Nginx acest lucru este implementat prin adăugarea liniei următoare:

```
add_header Strict-Transport-Security "max-age=63072000; includeSubDomains";
```

În ceea ce privește securitatea containerelor Docker, fiecare container primește o stivă proprie de rețea, așadar un container nu va primi acces privilegiat la *socket* sau interfețele altor containere. Desigur, dacă sistemul gazdă este configurat de așa manieră, containerele pot interacționa între ele prin interfețele de rețea respective – așa cum pot interacționa și cu sisteme externe. Specificând porturi publice pentru containere sau folosind *link*-uri, traficul IP este lăsat să treacă între containere. Din punct de vedere arhitectural al rețelei, toate containerele pe un sistem Docker stau pe interfețe *bridge*. Acest lucru le face să emuleze perfect o topologie fizică între mașini fizice conectate în același switch Ethernet.

3.4 Scalabilitatea aplicației

Arhitectura de față a fost proiectată de la început cu posibilitatea scalării pe orizontală a resurselor care o compun. Pe scurt, arhitectura infrastructurii este separată în trei puncte de încărcare: nodul de persistență (Redis), nodul de aplicație (Node.js) și nodul de balansare a traficului care este de asemenea răspunzător cu criptarea datelor (Nginx).

Pentru a reuși separarea încărcării dintre sistemul logic de balansare și cel de aplicație am folosit conceptul de SSL Termination: tot traficul din spatele serverelor de balansare este izolat într-o rețea privată și datele transmise necriptat interiorul acestei rețele sunt necriptate, eliminând surplusul de procesare al criptării datelor de mai multe ori pentru aceeași cerere.

În cazul în care încărcarea serverelor Nginx rezultată din criptarea și decriptarea datelor este mai mare decât încărcarea procesării efective a cererilor – validarea mutărilor jucătorilor în jocuri, procesarea mesajelor de comunicație și alte opreații efectuate pe serverul de aplicație, atunci se poate mări doar numărul de servere Nginx astfel încât să se mențină performanța oferită de serverele de aplicație. Adăugarea de servere Nginx se va face împreună cu modificări în serverul DNS, deoarece pentru a putea fi folosite de către vizitatori, adresele IP ale acestor noi servere trebuie publicate într-un sistem round-robin: pentru același nume de domeniu *exemplu.com* se vor adăuga mai multe adrese IP prin înregistrări A, browserele alegând la întâmplare una dintre adrese. Configurația Nginx relevantă pentru balansarea conexiunilor către serverele de aplicație, care se ocupă și de cazurile în care un nod de aplicație nu mai răspunde este:

```
proxy_pass              http://io_nodes;
proxy_next_upstream     error timeout invalid_header http_500;
proxy_set_header        Upgrade $http_upgrade;
proxy_set_header        Connection "upgrade";
proxy_http_version      1.1;
proxy_connect_timeout   2;
```

În cazul în care procesarea datelor de către serverele de aplicație începe să devină lentă, se pot adăuga noi servere de aplicație Node.js, lăsând numărul de servere de balansare constant dacă acesta nu are încărcare destul de mare. Pentru a adăuga servere de aplicație se pornește încă un container Docker ce conține aplicația, însă acesta nu va fi vizibil automat pentru serverele de balansare. Am rezolvat această problemă realizând un mic script folosind BASH, `updatelist.sh`, care să interogheze serverul Docker despre containerele Node.js active (cu comanda `docker ps`) și, după extragerea adresei IP private a fiecăruia dintre acestea, să recompileze fișierul de configurare `_io-nodes.conf`, trimițând la final un semnal procesului Nginx pentru a reîncărca și aplica configurația nouă:

```
#!/bin/bash
PORT=3000
echo "upstream io_nodes { ip_hash;" > ../nginx/conf.d/_io-nodes.conf
docker ps | grep "node " | awk '{print $1}' | while read cont; do echo
"server \
       $(docker inspect -f '{{.NetworkSettings.IPAddress}}' $cont):$PORT;"
      >> ../nginx/conf.d/_io-nodes.conf ; done
echo } >> ../nginx/conf.d/_io-nodes.conf
docker exec -it proxy nginx -s reload
```

Manevrarea conexiunilor Socket.IO folosește implicit o zonă de memorie RAM. Acest lucru împiedică pornirea mai multor servere Socket.IO care să deservească aceeași aplicație deoarece acestea vor avea date diferite despre ce conexiuni active există la un moment dat. Rezolvarea problemei este folosirea unui adaptor pentru stocare în Redis a datelor de sesiune astfel încât toate nodurile să dispună de aceeași imagine a sesiunilor. Această modificare deschide posibilitatea trimiterii unui mesaj de la un client conectat pe serverul de aplicație A către un client conectat pe serverul de aplicație B.

Scalarea Redis este non trivială. Cum Redis folosește memoria RAM pentru stocarea datelor, simpla adăugare de servere nu va genera rezultatele dorite. Soluția este fragmentarea datelor – *sharding*. Fiecare server Redis va opera pe anumite fragmente din datele aplicației – în cazul nostru avem nevoie de o operație (de tipul modulo) care să impartă șirurile de date *game:sha256hash* în mod egal către serverele pornite. Pentru simplitatea configurării serverelor de aplicație se poate folosi proiectul *Redis Cluster*, o aplicație care promite scalabilitate liniară până la 1000 de noduri. Funcția de dispersie este `HASH_SLOT = CRC16(key) mod 16384`. În aplicația prezentată nu s-a folosit Redis Cluster deoarece proiectul este încă în faza Alpha de dezvoltare, iar configurația curentă (folosind un singur nod) poate susține (pe laptopul de dezvoltare) mai mult de 30.000 de operații pe secundă, asta însemnând că poate susține aproximativ 10.000 de jucători concurenți cu un timp de răspuns sub 50 ms.

CAPITOLUL 4. Prezentarea aplicației

4.1 Arhitectura software

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, prevenindu-se mutările ilegale sau alte acțiuni prin care un jucător să poată trișa.

În general, comunicarea pe web se face cu cereri GET la care serverul trimite un răspuns. Acest răspuns este de regulă în format HTML (HyperText Markup Language), iar browserul interpretează instrucțiunile găsite în conținutul paginii, încărcând resursele de stilizare CSS (Cascading Style Sheet) referențiate și scripturile JavaScript ce pot fi incluse. Toate aceste resurse – fișierele CSS, JS, imaginile – generează la rândul lor câte o cerere GET la care serverul răspunde și, după ce transferul s-a efectuat, conexiunea se încheie.

Pentru a comunica în timp real este nevoie de conexiuni cu durată mai lungă. Fiecare cerere individuală vine cu un surplus de timp și de resurse consumate, browserul fiind nevoit de fiecare dată să verifice în cache-ul DNS adresa IP a serverului, apoi să inițieze o conexiune TCP, în cazul protocolul HTTPS trebuie de asemenea să efectueze în plus un *handshake* cu serverul pentru a interschimba cheile publice ș.a.m.d.

Am dorit să evit folosirea conexiunilor de scurtă durată, prin urmare am ales să folosesc o bibliotecă open-source care să ajute la comunicarea în timp real, Socket.IO. Aceasta este folosită în acest proiect și standardizează comunicarea prin evenimente dintre client și server, oferind suport pentru următoarele tipuri de transport:

- WebSocket
- FlashSocket
- jsonp-polling
- xhr-polling

Primul transport (WebSocket) este cel preferat deoarece este cel mai eficient atât pentru client cât și pentru server. Dacă browserul clientului este mai vechi și nu are suport pentru el, se va încerca folosirea FlashSocket, care însă necesită disponibilitatea plugin-ului Flash pe browser. Acest lucru nu este posibil pe platformele mobile iOS sau Android deoarece producătorii au renunțat la plugin-urile Flash din cauza problemelor de performanță

care afectau și consumarea bateriei într-un ritm mai accelerat decât normal. Transportul FlashSocket este destinat folosirii în special cu versiunile anterioare Internet Explorer 10. Dacă nici transportul FlashSocket nu poate fi inițializat, se va încerca folosirea jsonp-polling sau xhr-polling, adică cereri GET de lungă durată care se întrerup când informația este transmisă sau atunci când clientul vrea să trimită informație către server (folosind o cerere POST). Aceste cereri sunt foarte dezavantajoase din punct de vedere al performanței, însă fac posibilă folosirea unor browsere mai vechi.

Partea grafică a proiectului (*GUI*-ul – Graphical User Interface) a fost construită folosind HTML și CSS, iar acțiunile dinamice de schimbare a conținutului sunt implementate cu ajutorul limbajului de programare JavaScript. Aplicația reacționează la evenimentele trimise de server, în cazul jocului Tic Tac Toe desenând starea sesiunii de joc într-un tabel HTML, eventual descriind și textual starea jocului (vezi Figura 9).

Figura 9: Câștigarea unui joc urmat de o cerere de rejucare

Caracterul robust al aplicației este oferit de arhitectura bazată pe evenimente – în loc de compararea periodică a unor date de intrare cu cele anterioare pentru a vedea dacă s-a schimbat starea aplicației, se apelează funcțiile *callback* doar cu setul de date relevant și doar atunci când acesta se schimbă. De exemplu, atunci când clientul emite evenimentul *new message* trimite și un obiect în care este stocat mesajul trimis. Serverul ascultă trimiterea acestor evenimente și face *broadcast* acestui mesaj către canalul din care face parte trimițătorul cu un eveniment denumit tot *new message*, la care ascultă clienții (browserele).

Aplicația are nevoie de chei unice pentru toate sesiunile de jocuri, fiind de preferat ca acestea să fie nontriviale (astfel încât un vizitator să nu poată „ghici” codul unei sesiuni arbitrare). Soluția a fost folosirea funcției de dispersie SHA256 (creată de NSA – National

Security Agency din Statele Unite ale Americii), aceasta având o rată de coliziune neglijabilă. Comparația favorită a criptanaliștilor despre probabilitatea obținerii unei coliziuni folosind SHA256 este cu probabilitatea ca un asteroid să se lovească de Pământ în următoarea secundă, obliterând civilizația și omorând câteva miliarde de oameni.

Proiectul de față este un motor pentru crearea de jocuri în timp real pentru mai mulți jucători, așadar am creat o abstractizare a acestui tip de joc, după cum urmează:

```
var AbstractGame = function (session) {
	this.id 			= session || 'invalidHash';
	this.playersNo 		= -1;
	this.state 			= {
		gametype: 	'generic', game: 'new',
		players: 	[ ], spectators: [ ],
		board: 		[ ], rematch: [ ],
		currPlayer: null, winner: null,
		lastmove: 	{}
	},
	this.nextPlayer 	= function() { return 0 };
	this.makeMove 		= function(move) {
		var lastmove = null;
		if (this.rematch(move)) { this.state.lastmove = move; return true; }
		if (this.state.game !== 'finished' && this.checkAndMakeMove(move)) {
			lastmove = move;
			if (!this.checkEnd()) {
				this.state.currPlayer = this.nextPlayer();
			}
		}
		this.state.lastmove = lastmove;
		return lastmove !== null;
	};
	this.addPlayer = function(sid) {
		if (this.state['players'].length < this.playersNo) {
			this.state.players.push(sid);
		} else {
			this.state.spectators.push(sid);
		}
		return this.state;
	}
	this.start = function() {
		if (this.playersNo === this.state.players.length
			&& this.state.game === 'new') {
			this.state.game = 'progressing';
			this.state.currPlayer = this.nextPlayer();
		}
		return this.state;
	};
	this.checkEnd = function() { return false };
	this.checkAndMakeMove = function() { return false };
	this.rematch = function(move) { return false };
};
```

După cum se poate vedea mai sus, un obiect de tipul abstract are în compoziție tablouri de jucători, spectatori și tabla de joc. Funcția *makeMove* nu trebuie să fie

implementată în jocurile ce implementează acest prototip, aceasta asigurând trecerea la următorul jucător prin apelul *nextPlayer* și verificarea condiției de oprire a jocului. De asemenea, implementările jocurilor nu au de ce să implementeze nici adăugarea de jucători la o sesiune sau pornirea acesteia.

Persistența sesiunilor de joc se face în Redis, sub chei de forma game:*hash-sha256*, hash-ul fiind generat cu ajutorul datei și orei curente concatenate cu cheia de identificare a socket-ului conectat. Adresa la care se desfășoară o sesiune de joc este de forma *exemplu.com/g/hash-sha256*.

4.2 Funcționalități server

Aplicația rulează pe unul sau mai multe servere de aplicație Node.js și folosește două din modulele care se găsesc preinstalate – *http* și *crypto*. Sunt utilizate de asemenea și module externe, precum *redis, express, socket.io* și *socket.io-redis*.

Biblioteca folosită pentru prelucrarea cererilor de tip HTTP este *express*. Aceasta acționează ca un nivel peste modulul intern *http*, oferind flexibilitate și ușurința descrierii acțiunilor uzuale în cod împreună cu o gamă largă de funcționalități utile. Cu Express putem scrie rapid o funcție *middleware* în care să tratăm sau măcar să înregistrăm toate mesajele de eroare, de exemplu:

```
app.use(function(err, req, res, next){
      console.error(err.stack);
      res.status(500).send('body-parser');
});
```

Pentru a defini rute URL pentru diverse metode HTTP (precum *GET, POST, DELETE, UPDATE* ș.a.m.d.) se pot crea funcții care trebuie date ca și parametru metodelor aplicației pe care dorim să le expunem. De exemplu, pentru a defini răspunsurile unei cereri GET către pagina rădăcină a aplicației și a unei cereri POST către pagina *disertație*, se poate folosi codul de mai jos, înlocuind bineînțeles funcțiile date ca și parametru cu comportamentul dorit:

```
app.get('/', function (req, res) {
   res.send('Răspunsul cererii GET către rădăcină');
});
app.post('/disertație', function (req, res) {
   res.send('Răspunsul unei cereri POST pe pagina disertație');
});
```

Express vine și cu o metodă ajutătoare cu scopul de a servi fișiere statice localizate în directoare. Aceasta este implementată ca o metodă a modulului – *express.static* – ce poate fi

configurată cu diverși parametri în funcție de necesități precum: listarea extensilor de servit, trimiterea câmpului *header* ETag (o metodă de a controla dacă resursa statică a fost modificată), opțiunea *maxAge* (pentru specificarea către browsere a duratei recomandate de păstrare în *cache* a resursei) și altele. În aplicația prezentată am folosit codul de mai jos pentru a servi directorul `public` în rădăcina (fiindcă primul parametru este funcția) aplicației:

```
app.use(express.static(__dirname + '/public', {
                maxAge: '1y',
                etag: false,
                setHeaders: function (res, path, stat) {
                        res.set('x-hostname', process.env.HOSTNAME )
                }
        })
);
app.get('/g/:gameid', function (req, res) {
        gameid = req.params.gameid;
        res.sendFile(__dirname + '/public/index.html');
});
```

Sub directiva de servire a directorului se poate observa și directiva prin care se tratează vizitatorii care încearcă să acceseze un anumit joc. Acestora li se salvează identificatorul (rezumatul funcției de dispersie *sha256* reprezentat în baza 64) în vederea încărcării ulterioare a datelor din Redis despre jocul cerut și servirea acestor date prin conexiunea WebSocket. Mai jos este prezentată definiția funcției de dispersie *createId*, folosită în vederea generării de identificatori unici ale sesiunilor de joc, plecând de la un *seed* format din identificatorul socket-ului jucătorului care a inițiat sesiunea, concatenat cu data și ora curente:

```
var createId = function(socketId) {
        return crypto.createHash('sha256')
                .update(socketId + new Date().getTime())
                .digest('base64');
};
```

Definiția funcției de mai sus face parte din modulul pe care l-am creat în vederea separării apelurilor către Redis, denumit *game-dao* (DAO fiind prescurtarea de la *Data Access Object*). Modulul este inițializat cu un singur parametru, instanța clientului redis, folosit global în toată aplicația și expune patru metode publice: *create* – pentru crearea unei sesiuni noi de joc fără salvare, *load* – pentru încărcarea unui joc existent (primește doi parametri – identificatorul jocului și o funcție *callback* pentru manipularea datelor despre joc), *loadMoves* – pentru recuperarea din Redis a mutărilor efectuate într-o sesiune a unui joc și *save*, care primește ca prim parametru obiectul jocului și îl persistă în Redis.

Validarea mutărilor jucătorilor și verificarea terminării jocului sunt verificate, din motive de securitate – pentru a evita jucătorii care doresc să trișeze trimițând date false, pe partea de server. Bineînțeles pentru ca acest lucru să fie posibil, fiecare joc suportat de platformă va implementa cele două metode definite în modelul *abstract*, *checkAndMakeMove* și *checkEnd*.

Pentru jocul implementat de Tic Tac Toe, *checkAndMakeMove* verifică dacă poziția în care se dorește marcarea este între limitele 0 și 8 inclusiv, dacă este rândul jucătorului care a inițiat mutarea și dacă poziția nu este deja ocupată:

```
this.checkAndMakeMove = function(move) {
       // make sure we're dealing with integers
       move.index = parseInt(move.index, 10);

       // disregard if move is outside of bounds
       if (move.index < 0 || move.index > 8) return false;

       // disregard move if it doesn't come from the current player
       if ( this.state.currPlayer !==
         this.state.players.indexOf(move.actor) )
         return  false;

       // disergard move if the index is currently occupied
       if (this.state.board[move.index] !== -1) return false;

       // everything fine, change the state and score
       this.state.game = 'progressing';
       this.state.board[ move.index ] = this.state.currPlayer;
       this.state.score[ this.state.currPlayer ] += Math.pow(2,
                                                    move.index);
       return true;
}
```

Condiția de terminare este verificată după fiecare mutare, după cum urmează:

```
var wins = [7, 56, 448, 73, 146, 292, 273, 84];
for (i in wins) {
     if ((wins[i] & this.state.score[this.state.currPlayer]) === wins[i]) {
      this.state.game = 'finished';
      this.state.winner = this.state.currPlayer;
      break;
     }
 }
```

Am ales cel mai succint și eficient mod de a verifica condiția, astfel: întâi, pe parcursul jocului, scorul fiecărui jucător este incrementat cu 2 la puterea poziției marcate. Codul de mai sus verifică apoi dacă scorul jucătorului curent se află în tabloul configurațiilor de final (7 reprezentând completarea primului rând, 56 al doilea rând, 73 prima diagonală ș.a.m.d.) folosind operația de conjuncție pe biți.

4.3 Funcționalități client

Clientul dezvoltat pentru această aplicație rulează într-un browser ce respectă noile standarde HTML5 și JavaScript ECMA-262, versiunea 3. S-a folosit biblioteca jQuery în implementarea clientului pentru a actualiza mai facil elementele din pagină, comunicarea cu serverul realizându-se exclusiv printr-un WebSocket deschis la inițializarea paginii. Folosirea WebSocket asigură eficiența aplicației, deoarece aceasta comunică cu serverul doar când și cât are nevoie, nemaifiind nevoie de a reîncărca o pagină întreagă cu resursele ei aferente atunci când se face o modificare foarte mică (de exemplu, o mutare a unuia dintre jucători). Prin această metodă se poate modifica entitatea vizată, atât pe server (după validări), cât și în interfața utilizatorului.

La prima accesare a aplicației este necesară introducerea unui nume de utilizator pentru sesiunea respectivă. În funcție de adresa URL pe care vizitatorul o folosește, acesta va intra fie pe calea rădăcină / (așa zisul *lobby*) unde va putea să vorbească cu ceilalți vizitatori, fie pe o sesiune de joc existentă – unde va avea rol de participant sau de spectator.

Când vizitatorul este in *lobby*, acesta nu face parte din nicio sesiune de joc. O sesiune nouă poate fi creată apăsând pe butonul *New game* din suita de controale (Figura 10).

Figura 10: Starea inițială a jocului

Altă funcționalitate a aplicației este aceea de a crea un joc nou și de a-l juca cu același partener din jocul precedent. Apăsând pe butonul *Rematch* din suita de controale (Figura 10), se trimite o cerere către al doilea jucător, iar dacă acesta acceptă (apăsând tot pe butonul *Rematch* după ce vede notificarea) se vor interschimba rolurile (al doilea jucător va juca drept X, iar primul va juca drept 0) și un nou joc va fi pornit cu tabla de joc reinițializată.

Pentru modificarea stării jocului a fost creată pe client funcția *handleGameState*, care este apelată când este primit evenimentul *game state*. Această funcție actualizează valoarea celor nouă celule ale tabelului cu X, 0 sau spațiu, în funcție de datele primite de la server. Tot aici se interpretează și metadatele primite, precum al cui rând este la mutare, dacă există un câștigător sau dacă oponentul a cerut o rejucare a meciului. În această funcție sunt generate mesaje conform rolului pe care clientul îl are (jucătorul care este la mutare, cel care așteaptă mutarea sau spectator).

Butonul *Replay* cere de la server un tablou cu toate stările jocului de până atunci. Le trece apoi, cu pauze de câte o secundă, prin funcția handleGameState. Este o soluție simplă de a revedea toate mișcările efectuate de jucători, probabil mult mai interesante la un joc de șah de exemplu. Atunci când unul dintre jucătorii activi – X sau 0 – face o nouă mutare sau inițiază o cerere de rejucare a meciului, mișcările precedente nu mai sunt afișate, aplicația afișând pe ecran ultima configurație în timp real.

Aplicația funcționează și dacă browserul folosit este unul de pe un dispozitiv mobil. Mai mult, dispozitivele mobile (telefoanele, tabletele) dispun de regulă de o cameră capabilă să focalizeze automat și să fotografieze. Codul QR, folosit cu un program de citire de coduri de bare duce la posibilitatea obținerii rapid de informații pe dispozitiv. Un vizitator odată ce face face parte dintr-o sesiune de joc poate împărtăși acea sesiune prin codul QR prezentat pe ecran. Un vizitator cu dispozitiv mobil poate deschide aplicația de citire de coduri, îndrepta camera dispozitivului către ecranul în care se desfășoară sesiunea curentă și în două secunde să se alăture sesiunii de pe propriul dispozitiv ca și cum ar fi introdus manual adresa URL în browser (Figura 11 prezintă codificarea textului corspunzător adresei URL a sesiunii de joc https://devstuff.me/g/soBycG9%2BvkUHoWVRCWRmEZo%2FdKwOYpyerPxMZDvNY60%3D).

Figura 11: Cod QR corespunzător unei sesiuni de joc

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.

[3] Dimitri Aivaliotis. *Mastering NGINX*. Birmingham, 2013

[4] John Rittinghouse, James Ransome. *Cloud Computing: Implementation, Management and Security*. CRC Press, 2009

[5] Tim Mather, Subra Kumaraswamy, Shahed Latif. *Cloud Security and Privacy*. O'Reilly, 2009

[6] George Reese. *Cloud Application Architectures*. O'Reilly, 2009

[7] https://code.google.com/p/v8/, accesat Ianuarie 2015

[8] https://docs.docker.com/, accesat Ianuarie 2015