|  |  |  |
| --- | --- | --- |
|  | УНИВЕРЗИТЕТ У НОВОМ САДУ  **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ** |  |

Милош Рибар

**ВИШЕАГЕНТСКО ОКРУЖЕЊЕ У КЛАСТЕРУ РАЧУНАРА СА БАЛАНСИРАЊЕМ ОПТЕРЕЋЕЊА СЕРВЕРА**

ДИПЛОМСКИ РАД

- Основне академске студије -

Нови Сад, 2018

|  |  |  |
| --- | --- | --- |
|  | УНИВЕРЗИТЕТ У НОВОМ САДУ ⚫ **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**  21000 НОВИ САД, Трг Доситеја Обрадовића 6 | Датум: |
|  |
| **ЗАДАТАК ЗА ИЗРАДУ ДИПЛОМСКОГ (BACHELOR) РАДА** | Лист/Листова: |
| 1/1 |

*(Податке уноси предметни наставник - ментор)*

| Врста студија: | Основне академске студије |
| --- | --- |
| Студијски програм: | **Рачунарство и аутоматика** |
| Руководилац студијског програма: | **др. Поповић Мирослав** |

|  |  |  |  |
| --- | --- | --- | --- |
| Студент: | **Рибар Милош** | Број индекса: | **RA 171/2014** |
| Област: | **Примењене рачунарске науке и информатика** | | |
| Ментор: | **др. Видаковић Милан** | | |
| НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА  ИЗДАЈЕ СЕ ЗАДАТАК ЗА ДИПЛОМСКИ (Bachelor) РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА:   * проблем – тема рада; * начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна; * литература | | | |

**НАСЛОВ ДИПЛОМСКОГ (BACHELOR) РАДА:**

|  |
| --- |
| **ВИШЕАГЕНТСКО ОКРУЖЕЊЕ У КЛАСТЕРУ РАЧУНАРА СА БАЛАНСИРАЊЕМ ОПТЕРЕЋЕЊА СЕРВЕРА** |

**ТЕКСТ ЗАДАТКА:**

|  |
| --- |
| Задатак рада је имплементација вишеагентског окружења у кластеру рачунара са балансирањем оптерећења. Серверски део апликације ће бити реализован у Java програмском језику, а клијентски део уз помоћ Angular 6 платформе. Апликација омогућава креирање и брисање агената, као и њихово покретање путем слања порука. Такође, апликација омогућава комуникацију између више инстанци апликације које су спојене у кластер. О томе којој инстанци апликације ће стићи конкретан захтев зависи од балансера оптерећења. Агенти комуницирају преко JMS подсистема, имплементираног унутар Wildfly сервера. |

|  |  |  |
| --- | --- | --- |
| Руководилац студијског програма: | Ментор рада: | |
|  |  | |
| Примерак за:  - Студента;  - Ментора | | |

Образац **Q2.НА.15-04** - Издање 2

# Sadržaj

[Sadržaj 1](#_Toc525395825)

[1. Uvod 3](#_Toc525395826)

[2. Opis korišćenih tehnologija i koncepata 4](#_Toc525395827)

[2.1 Agentsko okruženje i agenti 5](#_Toc525395828)

[2.2 Klaster računara 5](#_Toc525395829)

[2.3 Balanser opterećenja 6](#_Toc525395830)

[2.4 MapReduce 6](#_Toc525395831)

[2.5 Java Platform, Enterprise Edition (JavaEE) 8](#_Toc525395832)

[2.5.1 Java Naming and Directory Interface (JNDI) 8](#_Toc525395833)

[2.5.2 Enterprise Java Beans (EJB) 10](#_Toc525395834)

[2.5.1 Java Message Service (JMS) 10](#_Toc525395835)

[2.6 Angular 12](#_Toc525395836)

[2.7 Wildfly aplikacioni server 13](#_Toc525395837)

[3. Specifikacija aplikacije 16](#_Toc525395838)

[4. Implementacija aplikacije 16](#_Toc525395839)

[5. Zaključak 16](#_Toc525395840)

[6. KLJUČNA DOKUMENTACIJSKA INFORMACIJA 17](#_Toc525395841)

[7. KEY WORDS DOCUMENTATION 18](#_Toc525395842)

# Uvod

Zadaci softverskog inženjerstva godinama postaju sve kompleksniji, i zahteva se sve brže vreme odziva sistema. U postizanju dobrih performansi, može nam pomoći primena distribuiranih višeagentskih okruženja. Primene ovakvih okruženja su ogromne, na primer, odbrana od potencijalnih ***Denial of Service*** ***(DDoS)*** napada i njihova analiza, pa do inteligentnih rešenja i oporavka servisa od otkaza. Softverska rešenja za ovakve probleme često zahtevaju mnogo procesorsorske snage, radne memorije i mnogo slobodnog prostora u skladištu podataka. Paralelno programiranje može da pruži dobre performanse, ali uključivanjem više mašina u rešavanje problema dobijamo mnogo bolje performanse.

Današnji softverski sistemi najčešće nisu okrenuti ka jednom korisniku, već ka većoj grupi korisnika, koji mogu da koriste sistem u isto vreme. Ovakav zahtev stvara jedan ozbiljan problem, koji se tiče dostupnosti sistema i scenarija otkazivanja istog. Sistem mora biti dostupan u svako doba dana i noći, i ne sme da otkaže zbog previše zahteva koji stižu na server koji ga opslužuje. Zbog ovoga, uvodimo termin visoke dostupnosti u softverski inženjering.

Visoka dostupnost (***High Availability)*** softverskog sistema podrazumeva postojanje klastera računara, koji će se ponašati kao serveri softverskog sistema. Oni bi trebalo da opslužuju zahteve korisnika u isto vreme. Ovim izbegavamo postojanje jedinstvene tačke otkaza softverskog sistema. Ako dođe do otkaza servera na jednom računaru, ostali računari u klasteru preuzimaju posao palog servera. Međutim, pored svega ovoga, potrebno je ograditi se od otkaza bilo kog servera u klasteru. Posao treba nekako podeliti na jednake delove. U distribuiranim sistemima, ovaj posao obavljaju balanseri opterećenja ***(Load Balancer)*** - serveri koji se ponašaju kao dispečeri zahteva. Čak ni otkaz balansera ne mora biti problem. Možemo postaviti više balansera i iskoristiti balansiranje opterećenja na klijentskoj strani, i ovime sprečiti, ili barem smanjiti verovatnoću otkaza servera.

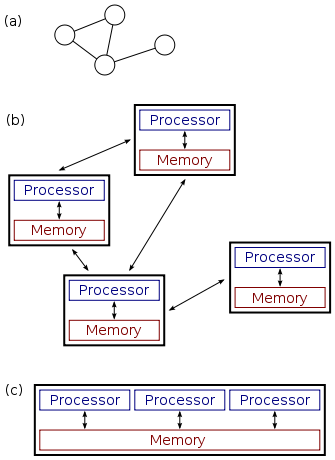
Agentsko okruženje koje je kičma ovog rada je napisano u programskom jeziku ***Java***. Za pravljenje klastera i balansiranje opterećenja koristimo ***Wildfly*** aplikacioni server. Napravljen je od strane softverske kompanije JBoss, a trenutno ga razvija softverska kompanije ***Red Hat***. Napisan je u programskom jeziku ***Java***, i implementira ***JavaEE*** specifikaciju. Radi na više operativnih sistema. Baziran je na Open-Source projektima, kao što su ***Hibernate***, ***RESTEasy***, ***HornetQ***, ***ApacheCXF*** i mnogi drugi.

Prvo poglavlje je uvodno, dok drugo poglavlje sadrži detaljniji opis korišćenih tehnologija i koncepata za potrebe izrade diplomskog rada. Treće poglavlje sadrži detaljnu specifikaciju aplikacije. Četvrto poglavlje sadrži opis konkretne implementacije agentskog okruženja, i korake za podešavanje visoke dostupnosti sistema koristeći Wildfly aplikacioni server.

# Opis korišćenih tehnologija i koncepata

Problemi koje softverski sistem treba da reši, vremenom se mogu menjati i proširivati novim problemima, za čije je rešenje potreban adekvatan mehanizam. Jedan od pogodnih koncepata za ovakvu problematiku, čine agentska okruženja. Agentska paradigma takođe predstavlja dosledan pristup u implementaciji distribuiranih sistema.

Distribuirani sistemi su sistemi koji se sastoje od više samostalnih računara koji komuniciraju preko računarske mreže. Prednost distribuiranih sistema u odnosu na paralelno programiranje je u tome što je dostupno onoliko procesorske moći, koliko je računara uključeno u sistem. Ovakav sistem može vremenom i u zavisnosti od potreba sistema da se proširi ili smanji. Termini „Paralelna obrada” i „Distribuirano izračunavanje” imaju mnogo preklapanja. Ipak, moguća je gruba klasifikacija. Ilustraciju razlike između ova dva termina možemo videti na slici 1.1.



Slika 1.1: Ilustracija razlike između distribuiranih (a, b) i paralelnih (c) sistema

Najbitnija razlika između distribuiranih i paralelnih sistema je ta što, kod distribuiranih sistema, svaki čvor ima svoju lokalnu memoriju, a komunikacija između drugih računara se uspostavlja razmenom poruka preko mreže. Kod paralelnih sistema memorija je deljena i svaki procesor ima direktan pristup deljenoj memoriji.

Paralelni sistemi koriste paralelne algoritme, a distribuirani sistemi koriste raspodeljene algoritme.

## Agentsko okruženje i agenti

Agent je softverski sistem koji je autonoman, drugim rečima, deluje bez inicijative ostalih sistema, i imaju kontrolu nad akcijama i unutrašnjim stanjem. Jedan agent komunicira sa drugim agentima preko nekog komunikacionog jezika, razmenom poruka. Jedan takav jezik je FIPA jezik. Poruke koje se razmenjuju između agenata napisane u ovom jeziku su FIPA ACL poruke. Agenti reaguju na promene u sistemu i odgovaraju na njih u konačnom vremenu.

Broj oblasti u kojima agenti nalaze svoju primenu su ogromne. Neke od primena agenata su upravljanje računarskom mrežom, nadgledanje pacijenata u bolnici, kontrola u energetici i rekonfiguracija veb servisa.

Agentsko okruženje predstavlja proširivo programsko okruženje koje upravlja životnim tokom agenata, i obezbeđuje mu sve potrebne mehanizme za realizaciju određenog zadatka. Ono obezbeđuje servise za rešavanje problema, komunikaciju između agenata, mobilnost agenata i sigurnosne mehanizme. Jedno takvo okruženje može podržavati mehanizme za rešavanje više problema. Problemi koje agenti u agentskom okruženju rešavaju, mogu, a ne moraju biti srodni. U ovom radu, agentsko okruženje podržava realizaciju ***MapReduce*** programerskog modela, koji se koristi za prebrojavanje reči u tekstualnim dokumentima. Takođe, podržan je i numerički algoritam ***Regresija*** koji predviđa rezultate fudbalskih utakmica. Ovo su dva problema koja nisu srodna, a agenti za njihovo rešavanje su prisutni unutar agentskog okruženja.

## Klaster računara

Klaster računara je grupa računara koji su povezani u lokalnu mrežu. Ovako povezani računari mogu se posmatrati kao jedan sistem. Svaki čvor u klasteru je zadužen za isti zadatak kao i svi ostali članovi klastera. Primena klastera računara je nebrojena. Koriste se, na primer, za povećanje dostupnosti softverskih sistema i izračunavanja za koja je potrebna ogromna procesorska moć. Svaki klaster ima svoju verziju operativnog sistema. Preporuka je da svaki čvor klastera poseduje isti operativni sistem. Njihova veličina varira od malih poslovnih klastera, čija brojnost nije velika, do ogromnih superkompjutera. Posrednik u komunikaciji između računara u klasteru se zove klaster middleware. To je najčešće aplikacija preko koje računari u klasteru razmenjuju poruke.

Postoje različite arhitekture klastera računara.

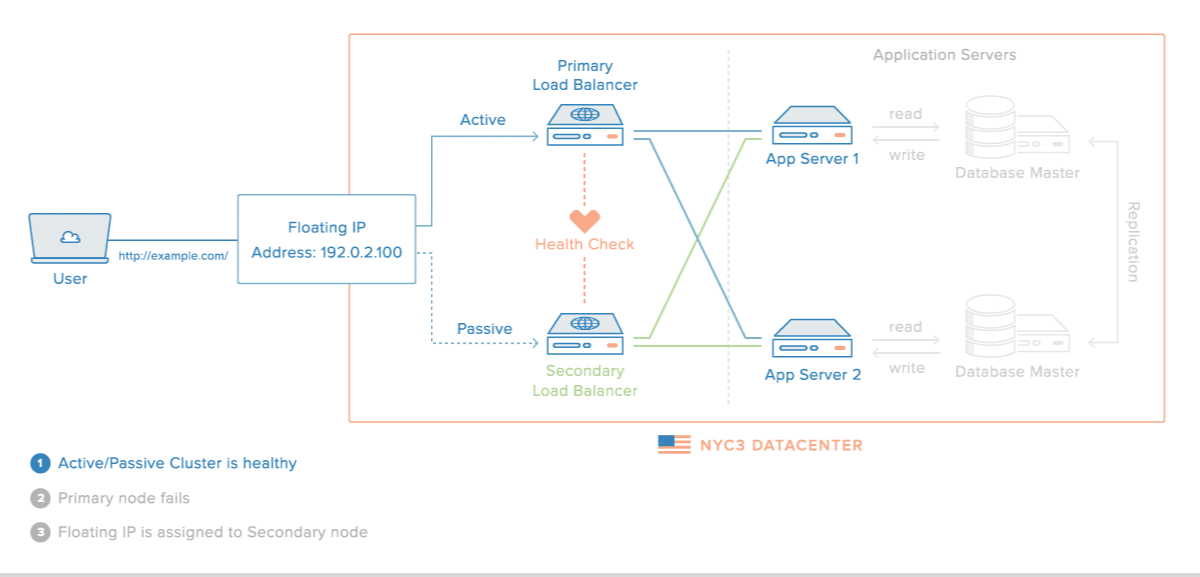
U svakom klasteru je neophodan i klasterizovan fajl sistem, koji omogućava da se jedan fajl sistem koristi od strane više servera. Postoje deljeni fajl sistemi, i distribuirani fajl sistemi.

## Balanser opterećenja

Balanser opterećenja je server čiji je zadatak da obezbedi dobru distribuciju posla između povezanih čvorova u mreži. Ovim se optimizuje korišćenje resursa, izbegava se preopterećenost jednog čvora, obezbeđuje se minimalno vreme odgovora i izbegava se postojanje ***Single Point of Failure***.

Postoje razne tehnike za balansiranje opterećenja, kao što su Round-Robin, Least Connections, Least Traffic, Least Latency, Server-Side Load Balancing, Client-Side Random Load Balancing.

Ilustracija malog klastera sa balansiranjem opterećenja je data u slici 1.2.1.



Slika 1.2: Visoka Dostupnost i Load Balancing u klasteru računara

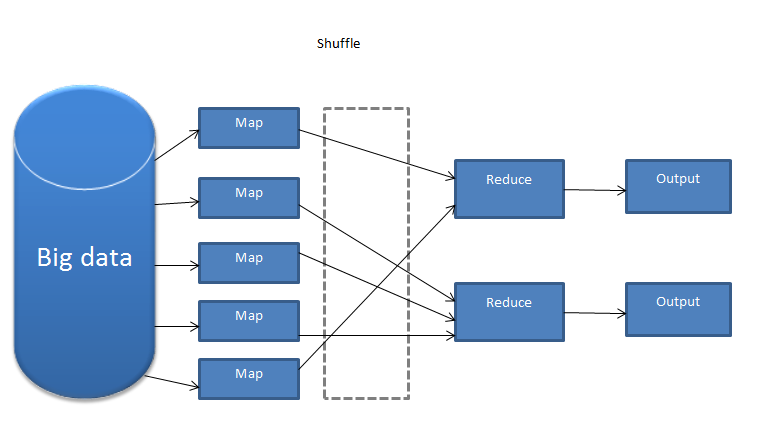
## MapReduce

MapReduce je model programiranja, a kako samo ime modela kaže, MapReduce model se sastoji od dva glavna dela, Map dela, i Reduce dela. U agentskom okruženju, map deo posla obavlja map agent, a reduce deo posla obavlja reduce agent. Ovaj model programiranja je prisutan u distribuiranim sistemima. Jedna od implementacija MapReduce algoritma je ***Apache Hadoop*** koji je nastao iz Google-ovog projekta. Objasnićemo MapReduce model programiranja na problemu prebrojavanja reči u tekstualnom dokumentu.

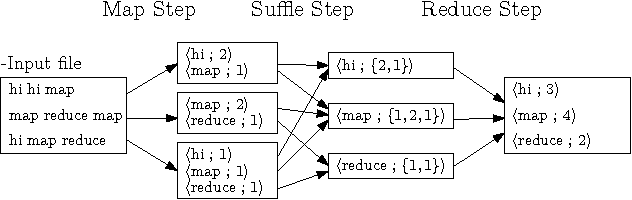
Svaki maper dobija neki deo datoteke kao ulaz. Zadatak mapera je da naprave tokene od ulaznih podataka i da ih namapiraju na neku vrednost. U našem slučaju, ta vrednost je broj pojavljivanja neke reči u datom delu, tekstualnog dokumenta. Između Map i Reduce faze, nalazi se i ***Shuffle*** faza koja sortira kreiranu mapu tokena i vrednosti.

Svaki maper šalje podatke reducerima. Zadatak reducera je da pronađe preklapajuće parove tokena i vrednosti sakupljenih od mapera, i da odrade operaciju reducinga nad njima. U našem slučaju, treba da saberu broj pojavljivanja primljenih tokena koji su stigli kao izlaz iz mapera. Svaki reducer generiše svoj izlaz. Mi smo u našem primeru ovaj izlaz posmatrali kao konačan broj ponavljanja reči u tekstualnom dokumentu.

Jednostavan primer šeme MapReduce programerskog modela dat je na slici 1.2.2.

  
Slika 1.3: Šema MapReduce programerskog modela

Jednostavna ilustracija primene MapReduce programerskog modela na naš problem je data na slici 1.2.3.

  
Slika 1.4: Jednostavna ilustracija primene MapReduce programerskog modela za prebrojavanje ponavljanja reči u tekstualnom fajlu

Primene MapReduce programerskog modela su velike. Neke od njih su klasterovanje dokumenata, distribuirano sortiranje i mašinsko učenje. MapReduce programerski model se u industriji koristi u svojim sofisticiranijim implementacijama, ali navedeni koncepti ostaju isti.

## Java Platform, Enterprise Edition (JavaEE)

JavaEE [1] je standard koji je razvijan od strane Java Community Procesa, uz doprinos stručnjaka iz industrije, komercijalnih i otvorenih organizacija, Java korisničkih grupa i bezbroj pojedinaca. Svako izdanje definiše nove funkcije koje se prilagođavaju potrebama industrije, poboljšavaju prenosivost aplikacija i povećavaju produktivnost programera.

Koristeći JavaEE standard napisano je bezbroj API-ja koje programeri koriste u svojim projektima.   
U ovom radu, korišćeni su razni podsistemi napisani u JavaEE, i biće opisani u narednim poglavljima.

### Java Naming and Directory Interface (JNDI)

JNDI [1] je Java API za servis imenika (eng.directory service) koji omogućava Java softverskim klijentima da pronađu podatke ili resurse u obliku Java objekata navodeći njihovo ime. Sun Microsystems je prvi izbacio JNDI specifikaciju 1997-e godine. Kasnije je i kompanija JBoss izbacila svoju implementaciju JNDI API-ja. Koristeći JNDI izbegavamo pisanje „boilerplate” koda.

JNDI je nezavisan od osnovne implementacije. Omogućava pristup različitim servisima kroz identičan interfejs. Možemo pristupiti konfiguracionim fajlovima za bazu podataka, mail sesijama, JMS fabrikama, koristeći koncepte servisa imenika (***Directory Service***). Pored toga, on određuje ***Service Provider Interface*** (SPI) koji obezbeđuje vezu sa konkretnim servisom.

Sve operacije podržane u servisima imenika su prisutne u JNDI API:

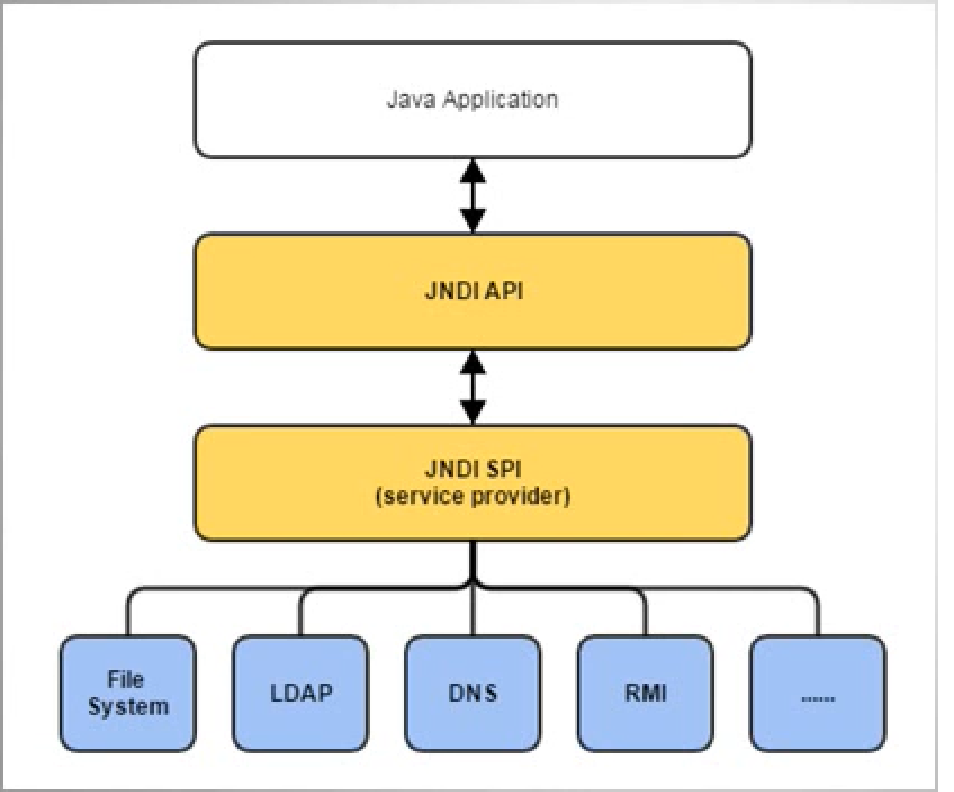
* **Binding** – operacija dodavanja veze ime – objekat
* **Unbinding** – operacija brisanja veze ime – objekat
* **Rebinding** – operacija modifikacije veze ime – objekat
* **Lookup** – operacija pretrage objekta po imenu
* **Pretraga pomoću filtera atributa**

Sve ovo se odigrava preko interfejsa Context, koji predstavlja kontekst imenika. On se sastoji od skupa veza ime – objekat. Preko njega komuniciramo sa JNDI API, koji komunicira sa JNDI SPI, preko kog možemo pristupiti različitim servisima.

Java Development Kit sadrži različite implementacije SPI koji omogućavaju rad servisima imenovanja kao što su servisi navedeni u tabeli koja sledi.

|  |  |  |
| --- | --- | --- |
| Skraćenica | Pun naziv | Opis |
| LDAP | **Lightweight Directory Access Protocol** | Omogućava pristup i održavanje distribuiranih servisa imenovanja preko IP protokola. |
| DNS | **Domain Name System** | Omogućava JNDI API pristup podacima smeštenih unutar DNS sistema. |
| RMI | **Java Remote Method Invocation** | Java API koji koristi Java Remote Method Protocol za pozivanje metoda nekog drugog koda u JVM kontekstu. Da bi programeri mogli da koriste ne-JVM kontekste, kasnije je razvijen CORBA sistem. |
| CORBA | **Common Object Request Broker Architecture** | Omogućava komunikaciju između Java i bilo kog drugog sistema. |

Komunikacija između Java klijenta i JNDI API je ilustrovana na slici 1.2.4.



Slika 1.5: Komunikacija između java aplikacije i JNDI API

### Enterprise Java Beans (EJB)

EJB [1] je java API koji se koristi za izradu poslovnog softvera. EJB je serverska komponenta koja enkapsulira poslovnu logiku aplikacije. Sadrži „runtime enviroment” za veb servise, kao što su servisi za softversku bezbednost i komponente za upravljanje životnim ciklusom servleta.

Tipovi EJB Session Bean-ova su:

* **Session Bean** – Stateful, Stateless i Singleton Beanovi kojima se može pristupiti preko Lokalnog ili Remote interfejsa. Svi beanovi ovog tipa podržavaju asinhrono izvršavanje.
* **Message Driven Bean** – Bean koji reaguje na poruke, koristeći paradigmu razmene poruka.

Kako agenti razmenjuju određene ACL poruke, Message Driven Bean-ovi su pogodni za njihovu implementaciju. Koristeći JNDI lookup možemo pristupiti EJB Session Beanovima.

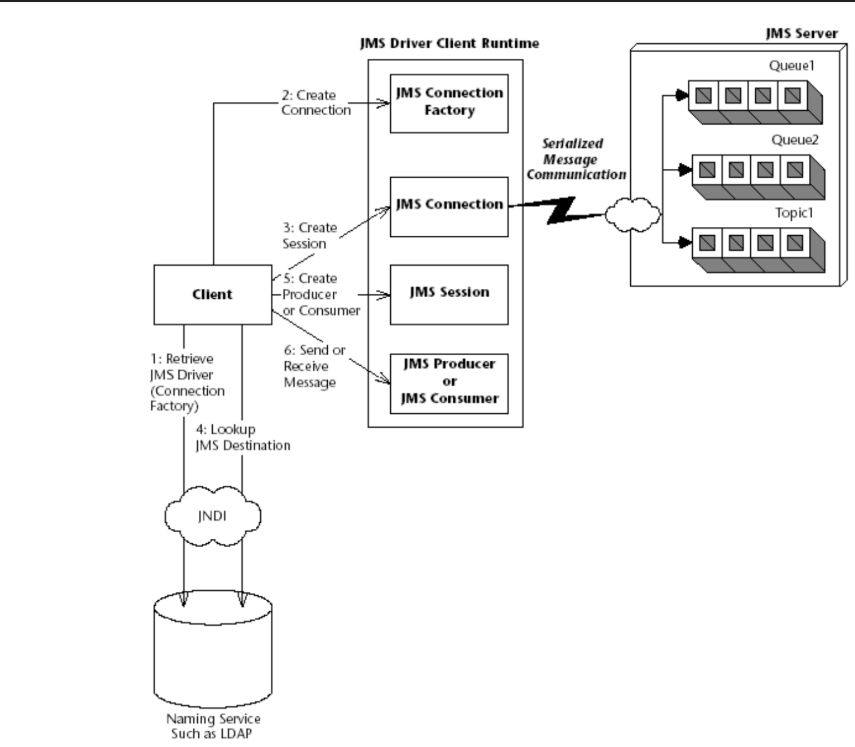
### Java Message Service (JMS)

Komunikacija između agenata uspostavljena je upotrebom Java Message Service middleware-a. Služi za transport poruka između dva ili više klijenata. Ovo je standard za razmenu poruka baziran na JavaEE specifikaciji, koja omogućava slanje, primanje, i čitanje poruka. Razvijen je od strane Sun Microsystems. Omogućava da komponente sistema u kojima JMS radi, ne moraju da znaju ništa o ostalim komponentama sistema koje koriste JMS. Ovo znači da agenti u agentskom okruženju ne moraju da znaju ništa o implementaciji i strukturi ostalih agenata unutar sistema, i sa njima mogu da uspostave indirektnu komunikaciju.

Komponente JMS API [1] su prikazane u sledećoj tabeli.

|  |  |
| --- | --- |
| Naziv | Opis |
| JMS Provider | Implementacija JMS interfejsa za message-oriented middleware (MOM). Implementirani su kao Java JMS implementacija ili kao adapteri ka ne-Java MOM. |
| JMS Client | Aplikacija ili proces koja šalje ili prima poruke. |
| JMS Producer/Publisher | JMS Klijent koji pravi i šalje poruke. |
| JMS Consumer/Subscriber | JMS Klijent koji čita poruke. |
| JMS Message | Objekat koji sadrži podatke koji se prenose između JMS klijenata. |
| JMS Queue | Prostor u kom se čuvaju poruke. Ove poruke može da pročita samo jedan JMS Consumer. Posle čitanja, nestaju iz reda. Garantuje da će se poruka pročitati, ali ne garantuje obradu poruka redosledom kojim su poslate. |
| JMS Topic | Mehanizam za slanje poruka koje se isporučuju svim JMS Consumer-ima. |

JMS API možemo iskonfigurisati koristeći JNDI lookup. Ilustracija korišćenja JNDI lookup-a za konfiguraciju JMS API, prikazana je na slici 1.2.5.



Slika 1.6: Korišćenje JNDI lookup za konfiguraciju JMS API

## Angular

Angular [1] je platforma otvorenog koda za razvoj veb aplikacija koja je bazirana na korišćenju TypeScript programskog jezika. Platformu razvija kompanija Google i često izbacuje nove verzije platforme. Isti tim koji je bio zadužen za razvoj AngularJS, zadužen je za razvoj Angular platforme, i predstavlja prerađenu AngularJS. Angular je sačinjen od skupa API-ja koji olakšavaju posao razvijanja veb aplikacije.

Angular u svojoj arhitekturi koristi hijearhiju komponenti. Svaka komponenta je sačinjena od HTML dela, koji predstavlja šablon za prikaz određene komponente, i TypeScript dela, koji predstavlja logiku komponente. Ovakva arhitektura komponenti stvara održivu arhitekturu čitave veb aplikacije. Angular komponente koriste servise koji u suštini predstavljaju logiku za komunikaciju sa serverom. Oni mogu biti injektovani u same komponente, i komponente mogu direktno pristupiti njihovim metodama. Komunikaciju sa serverom omogućavaju nam API-ji iz paketa common/http koj je dostupna kao deo skupa Angular paketa.

Angular koristi koncepte vezivanja atributa, reagovanja na događaje i delegiranje događaja na komponente veb aplikacije, ako je to potrebno. Koristeći operator "[ ]" koristimo vezivanje atributa za HTML elemente. Koristeći operator "( )"  reagujemo na događaje koji su se desili nad HTML elementima. Ovi događaji mogu biti klik na dugme, preuzimanje fokusa komponente i gubljenje fokusa.

Angular platforma se koristi za kreiranje Single Page veb aplikacija. Aplikacija ima stanje koje je predefinisano rutom na kojoj se aplikacija nalazi. Svaka ruta je vezana za neku komponentu, koja unutar sebe može sadržati druge komponente. Odlaskom na neku rutu koristeći Router API, stanje aplikacije će biti promenjeno. Dobra strana Angular platforme je ta što ne dovlači nove stranice sa servera. Ovo omogućava neometano korišćenje aplikacije, i omogućava ponašanje nalik desktop aplikacijama.



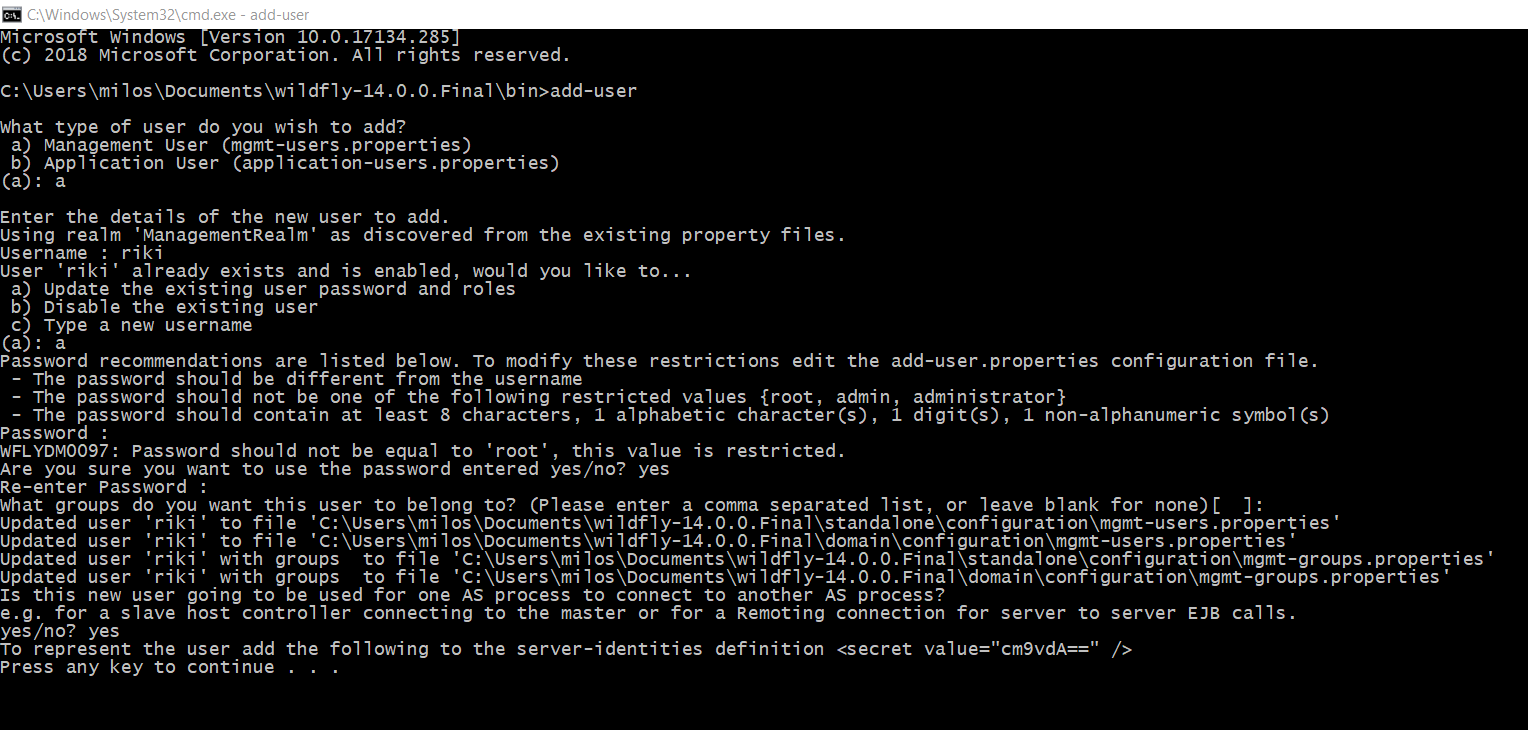
Slika 1.7: Arhitektura Angular veb aplikacije

## Wildfly aplikacioni server

Današnji poslovni softverski sistemi su često jako veliki, i potreban je veliki broj programera da bi se takav jedan sistem implementirao. Programeri često imaju mnogo posla oko kreiranja poslovne logike sistema, i potrebno je održati fokus na njenom razvoju. Aplikacioni server [1] je softverski radni okvir koji pruža različite mehanizme za opsluživanje veb aplikacije. Oni pružaju serversko okruženje koje je potrebno svakoj veb aplikaciji da bi pravilno radila. Mehanizmi koje aplikacioni serveri pružaju su razni, a neki od njih su klasterovanje i balansiranje opterećenja. Omogućavaju programerima da se fokusiraju na kreiranje poslovne logike softverskog sistema.

Wildfly aplikacioni server [3] je sačinjen od mnogo manjih podsistema. Pružaju servise imenika, razne servise koji pomažu u upravljanju životnog ciklusa aplikacije, servise za logovanje, i mnoge druge. Detaljan pregled podsistema koje Wildfly pruža je na oficijalnom sajtu Wildfly servera. Ovo omogućava serversku modularnost, i ovi servisi se mogu uključivati i isključivati po želji. Konfiguracija samog servera je centralizovana i jednostavna. Server pruža administraciju preko administratorske konzole, koja je predstavljena u obliku veb aplikacije.

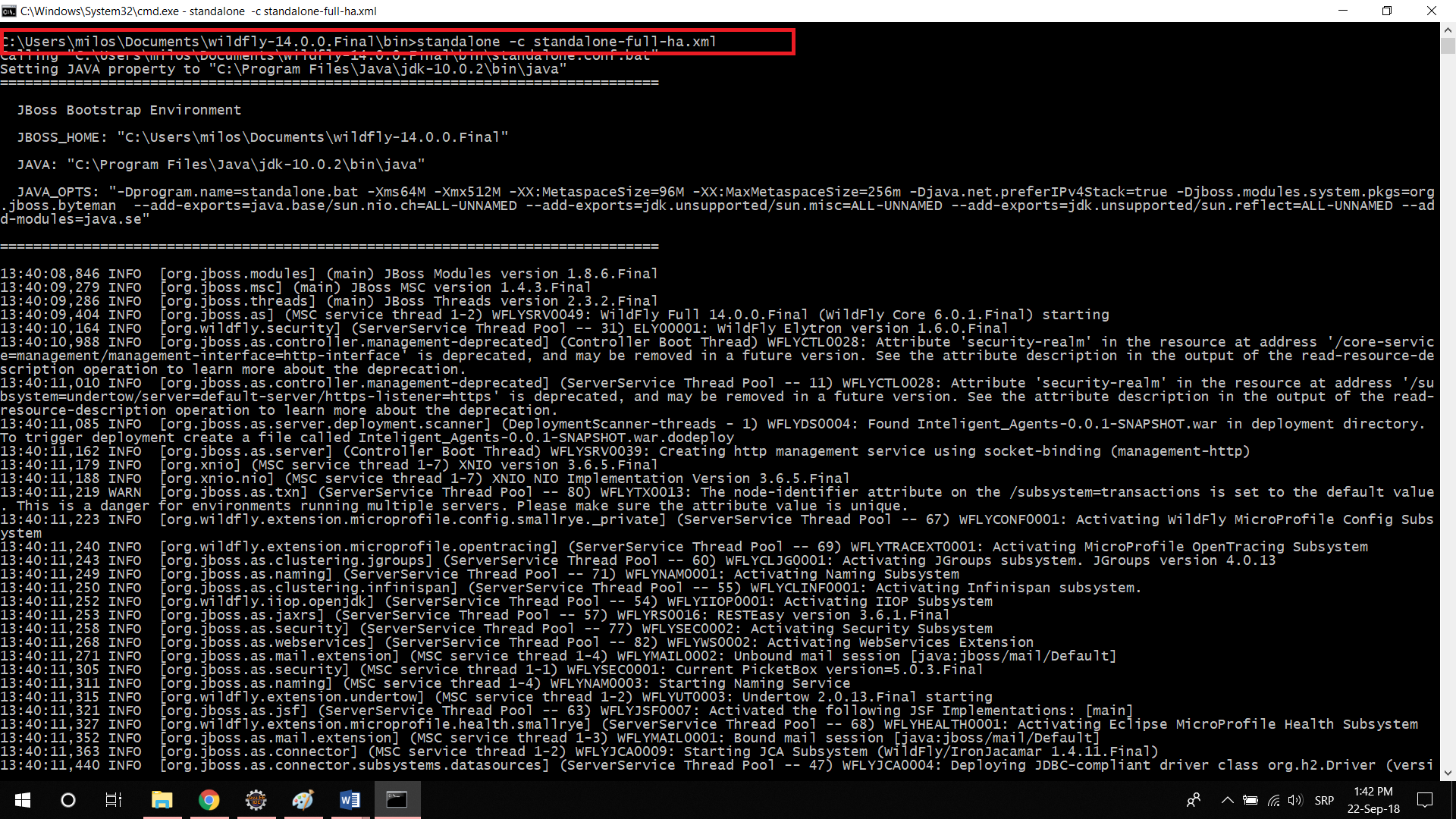
Da bi smo mogli da koristimo administratorsku konzolu, potrebno je dodati korisnički profil korisnika koji je zadužen za administraciju. Ovo možemo da uradimo preko ***add-user*** programa koji dolazi uz Wildfly server i nalazi se u bin folderu. Postupak za kreiranje administratorskog korisničkog profila je prikazano na slici 1.8.

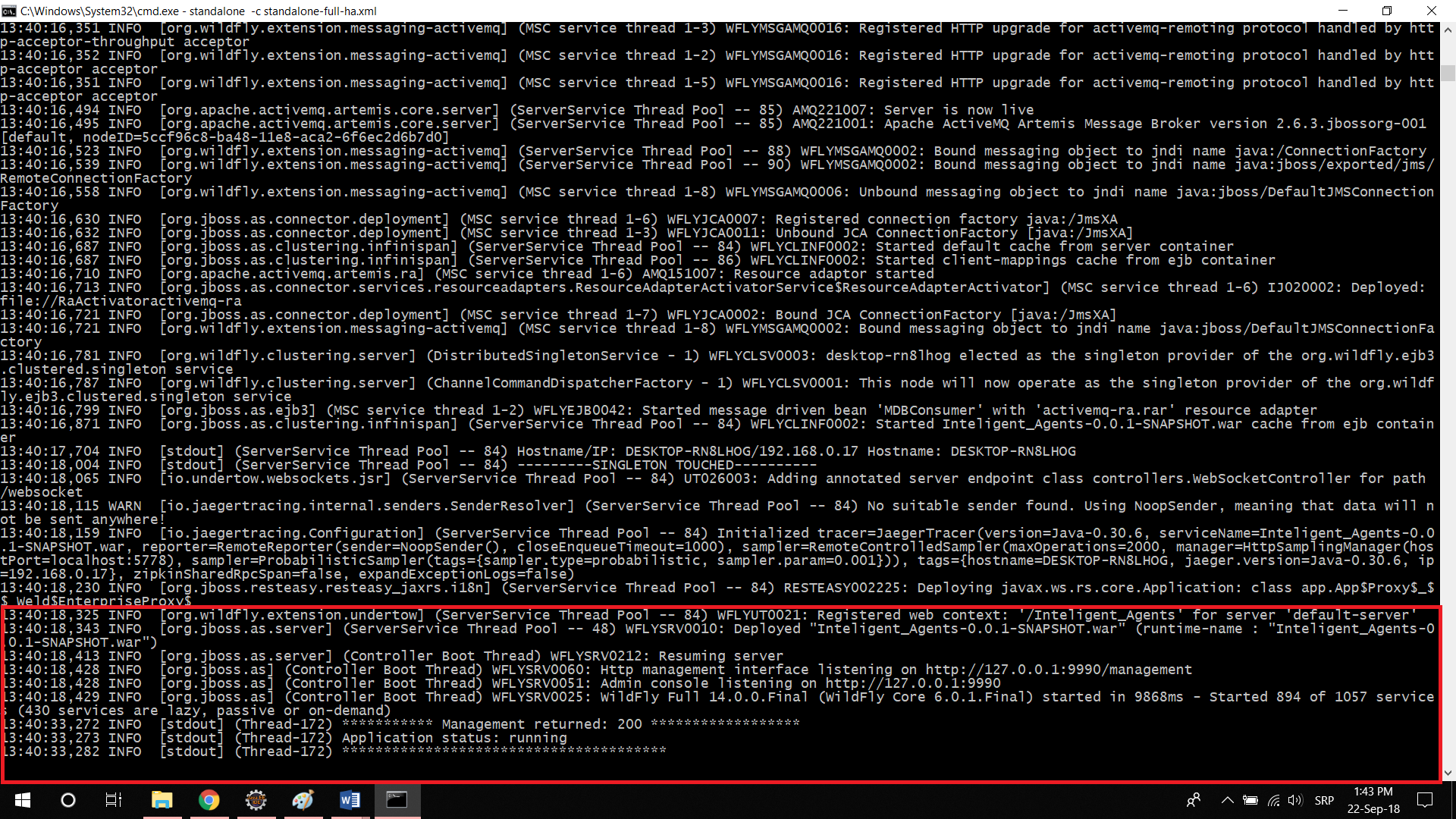
  
Slika 1.8: Kreiranje admin korisničkog profila preko add-user programa

Da bi smo mogli da počnemo da koristimo administratorsku konzolu, potrebno je pokrenuti server. Pokretanje se obavlja preko ***standalone*** programa. Kao ulazni parametar možemo uneti IP adresu i port na kom želimo da pokrenemo server. Možemo uneti i u kom režimu želimo da pokrenemo server. Režimi rada su predstavljeni preko standalone profila. Napisani su u obliku XML fajla, i svo konfigurisanje ovih profila mora da se uradi direktno preko njihovih XML fajlova. Ovi XML fajlovi se nalaze u standalone/configuration folderu.

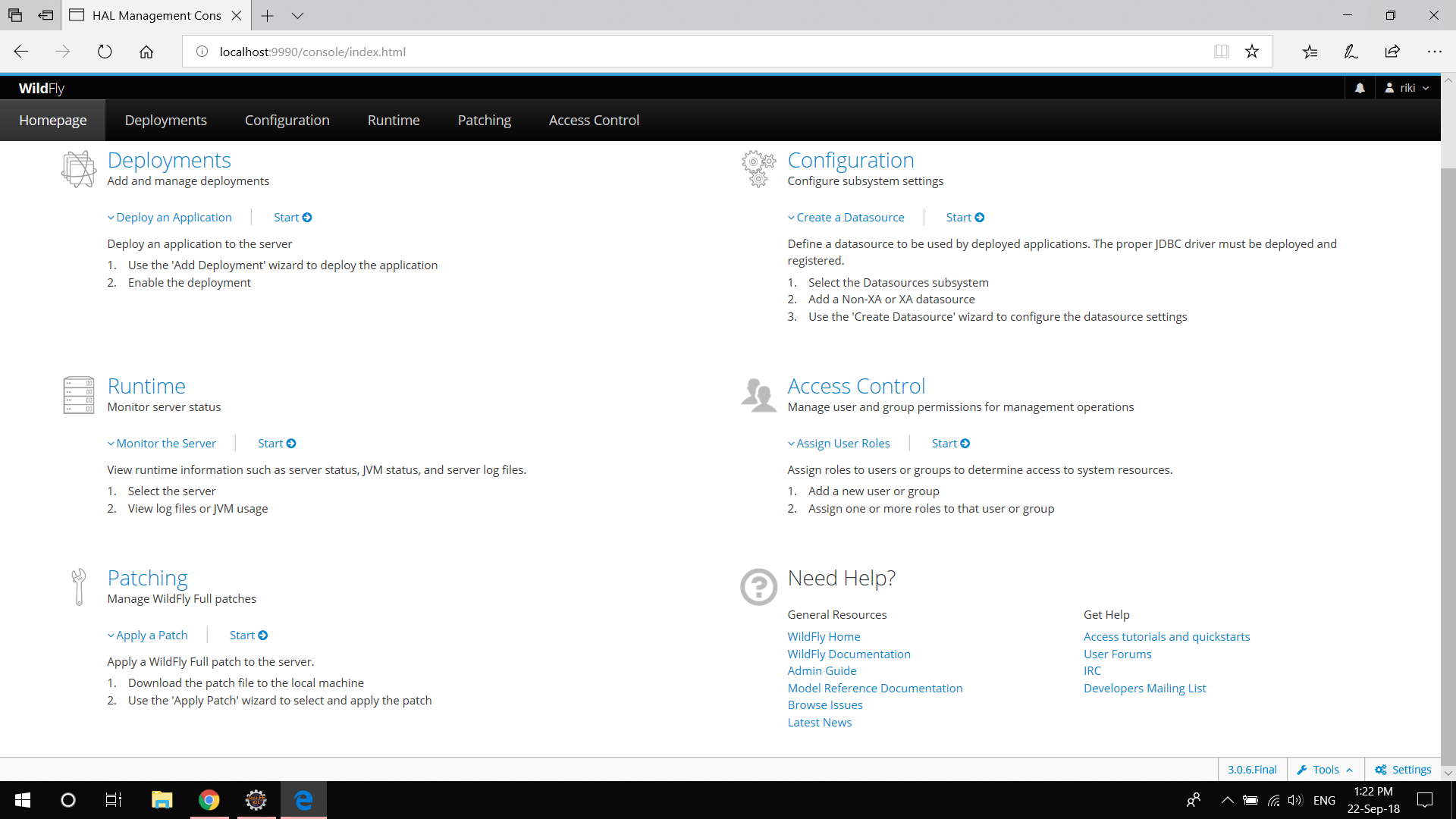
* **Standalone** – podrazumevani profil u kom nisu iskonfigurisani podsistemi kao JMS, i nije dostupan mehanizam za visoku dostupnost deploy-ovane aplikacije.
* **Standalone-full** – profil u kom su uključeni podsistemi, ali mehanizam za visoku dostupnost deploy-ovane aplikacije nije dostupan.
* **Standalone-ha** – profil u kom je dostupan mehanizam visoke dostupnosti, ali podsistemi su isključeni.
* **Standalone-full-ha –** profil u kom su uključeni i podsistemi i mehanizmi za visoku dostupnost.
* **Standalone-load-balancer** – profil koji nam omogućava da se Wildfly server ponaša kao balanser opterećenja.

Pokretanje Wildfly servera u **Standalone-full-ha** režimu možemo obaviti tako što pozovemo standalone program preko konzole sa ulaznim parametrom -c standalone-full-ha.xml. Pokretanje standalone aplikacije u ovom režimu predstavljeno je na slici 1.9. Uspešno pokretanje standalone u full-ha režimu je predstavljeno na slici 1.10.

  
Slika 1.9: Izgled konzole pri uspešnom pokušaju pokretanja standalone programa u nekom od režima rada

  
Slika 1.10: Izgled konzole pri uspešnom pokretanju standalone programa u nekom od režima rada. Na slici je primer pokretanja standalone programa kada je aplikacija deploy-ovana na server

Administratorska konzola je prikazana na slici 1.11.

  
Slika 1.11: Wildfly administratorska konzola

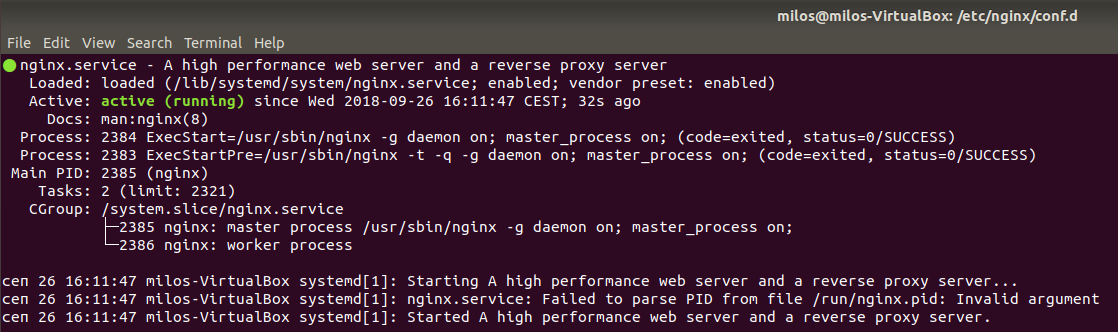
## NGINX

NGINX [4] je veoma brz HTTP i Proxy server, kog koriste najveće svetske softverske kompanije. Neke od tih kompanija su Netflix, Hulu, Pinterest, GitHub, Soundcloud, i mnoge druge.

Za razliku od ostalih servera, ne oslanja se na računarske niti da bi rukovao zahtevima, već koristi skalabilnu, događajima vođenu asinhronu arhitekturu. Koristi jako malo resursa. Zbog ovoga, može da opslužuje sve vrste klastera, od malog poslovnog klastera do ogromne farme računara.

U ovom radu, zbog svoje jednostavne konfiguracije, NGINX je iskorišćen kao balanser opterećenja za WebSocket komunikaciju.

Koristeći neki od menadžera paketa, instalacija NGINX servera se svodi na jednu komandu. Pod operativnim sistemom Ubuntu, ta komanda je: sudo apt install nginx. Dalje, koristeći komandu sudo systemctl status ngnix, možemo proveriti da li je nginx server upaljen. Primer je dat na slici 1.12.

  
Slika 1.12: NGINX server je upaljen i sluša na adresi IP:Port. Konfiguracija servera može da se promeni u nginx.conf datoteci.

Konfiguracija servera može da se promeni kreiranjem konfiguracione datoteke u folderu /etc/nginx/conf.d/\*.conf.

NGINX može da se koristi i kao balanser opterećenja samog agentskog okruženja, ali taj posao će u ovom radu odraditi Wildfly aplikacioni server.

# Specifikacija aplikacije

# Implementacija aplikacije

Aplikacija je implementirana u Java programskom jeziku, verzije 1.8. Za kompletan razvoj aplikacije i kreiranje klastera i balansera opterećenja, korišćeni su sledeći alati i tehnologije:

* Eclipse JEE Oxygen IDE
* EJB 3.2
* JDK 10 – JNDI, JMS
* Angular 6 i Bootstrap 4
* Wildfly 14 aplikacioni server
* Oracle VM VirtualBox
* Host mašina sa operativnim sistemom Windows 10
* Virtuelne mašine sa operativnim sistemom Ubuntu 18.04
* NGINX 1.15.4 web server

Arhitektura aplikacije zasnovana je na Agentskoj arhitekturi. Agenti su Bean-ovi koji reaguju na poruke dobijene od Message Driven Bean-a za rukovanje porukama. Oni se obraćaju servisima koji su im namenjeni. Klijentska aplikacija komunicira sa serverom preko Web Socketa, i tako prati promene u klasteru. Čvorovi klastera komuniciraju međusobno preko REST poziva. Serverska aplikacija je podeljena u sledeće slojeve:

* Controller – omogućavaju komunikaciju između čvorova preko Handshake i Heartbeat protokola
* Model – predstavljaju modele podataka
* Agents – Agenti koji reaguju na poruke
* Services – Servisi kojima se agenti obraćaju
* Configurators – EJB Beanovi koji rukuju JMS, JDK i Handshake i Heartbeat mehanizmima
* Request Senders – Omogućavaju slanje HTTP zahteva iz koda, koristeći RESTEasy

Front-end je realizovan uz pomoć Angular 6 platforme. Arhitektura Angular 6 veb aplikacije se sastoji iz sledećeg:

* Components – komponente koje čine izgled Angular 6 aplikacije. Ovaj deo aplikacije je jednostavan, postoje samo četiri komponente: Konzola, Home, Toolbox i Lista.
* Services – klase koje su zadužene za komunikaciju sa serverom. Postoje REST servis i WebSocket servis. Injektovani su u komponente.
* Model – model podataka koji se koristi za komunikaciju sa serverom.

## Serverska aplikacija

Serverski deo aplikacije rađen je po ugledu na Agentski arhitektonski šablon. Podeljena je na tri celine, Model-View-Controller. View čini Angular 6 veb aplikacija. Više reči o njoj će biti u poglavlju 4.2. Kontroleri se sastoje od metoda koje reaguju na HTTP ili Web Socket zahteve. Zbog bolje čitljivosti i održavanja koda, kontroleri su podeljeni u više klasa:

* AgentsController –služi za CRUD operacije nad agentima
* HandshakeController –služi za uspostavljanje veze između čvorova
* HeartBeatController – služi za proveru konekcije sa ostalim čvorovima
* WebSocketController – služi za otvaranje i zatvaranje WebSocket sesije
* HeartbeatController – kontroler sa jednom metodom koja vraća HTTP 200 OK odgovor. Koristi se u Heartbeat protokolu.

Kontroleri koriste istoimene servise, u kojima se krije sva logika. Servisi dalje koriste Request Sendere da bi mogli da šalju HTTP zahteve drugim čvorovima aplikacije. Servisi i Request Senderi su Stateful, Stateless i Singleton EJB Beanovi, i mogu da se injektuju u druge klase preko anotacije @Inject.

Kofiguratori su klase koje su zadužene za JNDI lookup, konfiguraciju JMS podsistema, dobavljanje informacija o Hostu kada se server u potpunosti pokrene i pokretanje Handshake protokola. Oni su većinom EJB Stateful beanovi. Sadrže request sendere preko kojih komuniciraju sa drugim čvorovima klastera.

Agenti su Bean-ovi koji komuniciraju međusobno u sistemu preko JMS podsistema. Upravljanje porukama vrši MDBConsumer klasa. Jezik i poruke preko kojih komuniciraju su pravljeni po ugledu na FIPA ACL poruke. Oni koriste servise koji su istog naziva kao i sami agenti, i koji su za njih namenjeni. Sistem poseduje sedam vrsta agenata, među kojima su MapAgent i ReduceAgent.

## Controller

Kontroleri su realizovani preko JAX-RS API-ja u kom se nalaze potrebne anotacije za njihovo kreiranje. Postoji pet kontrolera:

* **HandshakeController** –zadužen za rukovanje zahtevima za sinhronizaciju čvorova pri ulasku u mrežu
* **AgentsController** – zadužen za rukovanje operacijama koji se tiču agentskog okruženja
* **WebSocketController** – zadužen za otvaranje i zatvaranje WebSocket konekcije
* **ClientCommunicationController** – zadužen za parsiranje i dobavljanje podataka JNDI stabla
* **HeartbeatController** – kontroler sa jednom metodom koja vraća HTTP 200 OK odgovor. Koristi se u Heartbeat protokolu.

AgentsController je jedini kontroler u kom se direktno krije logika. On šalje odgovor websocket-u koji je poslaozahtev. AgentsService servis služi za obavljanje dodatnog posla AgentsController kontrolera. Ostali kontroleri su jednostavni, i koriste istoimene servise kako bi rukovali zahtevima. Implementacija AgentsController kontrolera data je na slici 4.1.

@Path("/app")

**public** **class** AgentsController {

@Inject

**private** AgentsService agentsService;

@Inject

**private** ClientRequestSender clientRequestSender;

@Inject

**private** JMSTopic jmsTopic;

**private** ObjectMapper mapper = **new** ObjectMapper();

@GET

@Path("/node")

@Produces(MediaType.***TEXT\_PLAIN***)

**public** String node() {

**return** "ACTIVE";

}

@GET

@Path("/agents/running")

@Produces(MediaType.***APPLICATION\_JSON***)

**public** List<AID> getRunningAgents(){

**return** agentsService.getAllRunningAgents();

}

@PUT

@Path("/agents/running/{type}/{name}")

@Consumes(MediaType.***APPLICATION\_JSON***)

@Produces(MediaType.***APPLICATION\_JSON***)

**public** String putNewAgent(@PathParam(value = "type") String type, @PathParam(value = "name") String name) **throws** JsonProcessingException, IOException{

String retStr = "";

Host myHostData = agentsService.getMyHostInfo();

**boolean** shouldAddAgent = agentsService.getAllRunningAgents()

.stream().filter( aid ->

aid.getName().equals(name) &&

aid.getHost().getHostAddress().equals(myHostData.getHostAddress()) &&

aid.getHost().getAlias().equals(myHostData.getAlias())

).findFirst() != **null**;

**if**(!shouldAddAgent) {

retStr = "Agent with the given name already exists on this node!";

} **else** {

**boolean** isMainNode = agentsService.getMainNode().getHostAddress().equals(myHostData.getHostAddress())

&& agentsService.getMainNode().getAlias().equals(myHostData.getAlias());

**for** (AgentTypeDTO item : agentsService.getAllSupportedAgentTypes()) {

**if**(item.getName().equals(type)) {

//add agent to my list

AID aid = **new** AID(name, myHostData, **new** AgentType(item.getName(), item.getModule()));

AgentInterface myAgent = AgentsFactory.*createAgent*(aid,jmsTopic,agentsService);

agentsService.getMyRunningAgents().add(myAgent);

agentsService.getAllRunningAgents().add(aid);

//initialize WEB-SOCKET

Iterator<Session> iterator = WebSocketController.*sessions*.iterator();

**while**(iterator.hasNext()) {

Session s = iterator.next();

s.getBasicRemote().sendText(mapper.writeValueAsString(**new** Message("startAgent", mapper.writeValueAsString(aid))));

}

**for**(Host h : agentsService.getSlaveNodes())

clientRequestSender.postNewRunningAgent(aid, h.getHostAddress());

**if**(!isMainNode)

clientRequestSender.postNewRunningAgent(aid, agentsService.getMainNode().getHostAddress());

retStr = "SUCCESS";

**break**;

}

}

}

**return** retStr;

}

@DELETE

@Path("/agents/running/{aid}")

@Consumes(MediaType.***APPLICATION\_JSON***)

@Produces(MediaType.***APPLICATION\_JSON***)

**public** String deleteRunningAgent(@PathParam(value = "aid") String aid) **throws** ParseException,

InstantiationException, IllegalAccessException, ClassNotFoundException, IOException {

String retStr = "";

String[] splits = aid.split("\_\_");

String agentName = splits[0];

String hostAddress = splits[1];

String alias = splits[2];

String agentType = splits[3];

AID queryAid = agentsService.getAllRunningAgents().stream()

.filter(o -> o.getHost().getHostAddress().equals(hostAddress)

&& o.getHost().getAlias().equals(alias)

&& o.getName().equals(agentName))

.findFirst().orElse(**null**);

**if**(queryAid == **null**)

retStr = "Can't find the agent by given AID.";

**else**{

agentsService.getAllRunningAgents().remove(queryAid);

//WEBSOCKET GOES HERE

Iterator<Session> iterator = WebSocketController.*sessions*.iterator();

**while**(iterator.hasNext()) {

Session s = iterator.next();

s.getBasicRemote().sendText(mapper.writeValueAsString(**new** Message("stopAgent", mapper.writeValueAsString(queryAid))));

}

//check if this is my Agent and delete from that list

**boolean** isMyAgent = **false**;

**if**(agentsService.getMyHostInfo().getHostAddress().equals(hostAddress)

&& agentsService.getMyHostInfo().getAlias().equals(alias))

isMyAgent = **true**;

**if**(isMyAgent) {

**for** (Iterator<AgentInterface> i = agentsService.getMyRunningAgents().iterator(); i.hasNext();) {

AgentInterface item = i.next();

**if**(item.getClass().isInstance(Class.*forName*("beans." + agentType).~~newInstance~~())) {

AgentClass agentObj = (AgentClass) Class.*forName*("beans." + agentType).cast(item);

AID myAid = agentObj.getAid();

**if**(myAid.getName().equals(agentName)) {

agentsService.getMyRunningAgents().remove(item);

**break**;

}

}

}

}

**boolean** isMainNode = **false**;

**if**(agentsService.getNodeType().equals(NodeType.***MASTER***))

isMainNode = **true**;

//send delete request to all (other) slaves

**for**(Iterator<Host> i = agentsService.getSlaveNodes().iterator(); i.hasNext();) {

String resp = clientRequestSender.deleteRunningAgents(i.next(), queryAid);

System.***out***.println(resp);

}

//if I am a slave, send to the main node also

**if**(!isMainNode) {

String resp = clientRequestSender.deleteRunningAgents(agentsService.getMainNode(), queryAid);

System.***out***.println(resp);

}

retStr = aid;

}

**return** retStr;

}

@POST

@Path("/messages")

@Consumes(MediaType.***APPLICATION\_JSON***)

@Produces(MediaType.***TEXT\_PLAIN***)

**public** String sendMessage(ACLMessage aclMessage) **throws** JsonProcessingException, IOException{

Iterator<Session> iterator = WebSocketController.*sessions*.iterator();

**while**(iterator.hasNext()) {

Session s = iterator.next();

aclMessage.setContent(aclMessage.getContent());

s.getBasicRemote().sendText(mapper.writeValueAsString(**new** Message("aclMessage", mapper.writeValueAsString(aclMessage))));

}

jmsTopic.send(aclMessage);

**return** "Success";

}

@GET

@Path("/messages")

@Produces(MediaType.***APPLICATION\_JSON***)

**public** Performative[] getMessages() {

**return** Performative.*values*();

}

@POST

@Path("/agents/running")

@Consumes(MediaType.***APPLICATION\_JSON***)

@Produces(MediaType.***APPLICATION\_JSON***)

**public** **boolean** postRunningAgents(AID aid){

**boolean** success = **true**;

**try** {

**if** (!agentsService.getAllRunningAgents().contains(aid)) {

agentsService.getAllRunningAgents().add(aid);

}

Iterator<Session> iterator = WebSocketController.*sessions*.iterator();

**while**(iterator.hasNext()) {

Session s = iterator.next();

s.getBasicRemote().sendText(mapper.writeValueAsString(**new** Message("startAgent", mapper.writeValueAsString(aid))));

}

} **catch**(Exception e) {

success = **false**;

e.printStackTrace();

}

**return** success;

}

}

Slika 4.1: Implementacija AgentsController kontrolera. Služi za rukovanje operacijama nad agentima.

## Configurator

Kao što je već rečeno, konfiguratori su koje su zadužene za JNDI lookup, konfiguraciju JMS podsistema, dobavljanje informacija o Hostu kada se server u potpunosti pokrene i pokretanje Handshake protokola. Glavna uloga im je održavanje veze između čvorova klastera i pristup informacijama o računaru na kom je aplikacija pokrenuta.

**JndiTreeParser** je klasa koja služi za pronalaženje svih tipova agenata koji postoje u nekom od čvorova klastera. Koristi se JBoss-ova implementacija JNDI podsistema. Informacija o tipovima agenata će biti poslata dalje svim članovima klastera u Handshake fazi. Implementacija klase je prikazana na slici 4.1.

**GetHostDataService** je klasa koja služi za inicijalizaciju informacija o samoj mašini na kojoj je aplikacioni server pokrenut. Daje nam informacije o imenu računara, na kom je portu aplikacija pokrenuta, koji je IP na kom je server pokrenut, i sve to smešta u promenljivu Host koji je jako bitan za dalji rad aplikacije. Takođe, ova klasa priprema HeartBeat protokol, koji svakih 15 sekundi, nakon povezivanja sa glavnim čvorom klastera, proverava da li su svi čvorovi klastera živi. Implementacija ove klase je data na slici 4.2.

**JMSTopic** klasa je zadužena za konfiguraciju JMS podsistema i kreiranje JMS Topic-a preko kog će svi agenti u klasteru razmenjivati poruke. Koristimo JBoss JNDI da bi smo došli do instance JMS Topic-a. Upravljanje članovima JMS podsistema je automatsko, i o tome se brine ActiveMQ Artemis podsistem. Dovoljno je kreirati instancu JMS Topica u kodu, i čvor klastera je povezan u JMS mrežu. Implementacija klase je data na slici 4.3.

**HandshakeService** klasa je između ostalog zadužena za prijavu čvorova u klaster. Tokom prijave, čvor sve svoje tipove agenata šalje master čvoru, gde posle toga glavni čvor šalje svim ostalim čvorovima nove tipove agenata. Posle ovoga, glavni čvor šalje novoprijavljenom čvoru listu svih podržanih agenata u klasteru i listu svih živih agenata. Ovime se obavlja sinhronizacija sistema. Implementacija metode za handshake je data na slici 4.4.

@Stateless

**public** **class** JndiTreeParser {

**private** String EXP = "java:jboss/exported/";

**private** String INTF = "!" + AgentInterface.**class**.getName();

**private** Context context;

**public** JndiTreeParser() {

Hashtable<String, Object> env = **new** Hashtable<>();

env.put(Context.***URL\_PKG\_PREFIXES***, "org.jboss.ejb.client.naming");

**try** {

**this**.context = **new** InitialContext(env);

} **catch** (NamingException e) {

e.printStackTrace();

}

}

**public** List<AgentType> parse() **throws** NamingException {

List<AgentType> result = **new** ArrayList<>();

NamingEnumeration<NameClassPair> moduleList = context.list(EXP);

**while** (moduleList.hasMore()) {

String module = moduleList.next().getName();

processModule(module, result);

}

**return** result;

}

**private** **void** processModule(String module, List<AgentType> result) **throws** NamingException {

NamingEnumeration<NameClassPair> agentList = context.list(EXP + "/" + module);

**while** (agentList.hasMore()) {

String ejbName = agentList.next().getName();

AgentType agentType = parseEjbNameIfValid(module, ejbName);

**if** (agentType != **null**) {

result.add(agentType);

}

}

}

**private** AgentType parseEjbNameIfValid(String module, String ejbName) {

**if** (ejbName != **null** && ejbName.endsWith(INTF)) {

**return** parseEjbName(module, ejbName);

}

**return** **null**;

}

**private** AgentType parseEjbName(String module, String ejbName) {

ejbName = extractAgentName(ejbName);

**return** **new** AgentType(ejbName, module);

}

**private** String extractAgentName(String ejbName) {

**int** n = ejbName.lastIndexOf(INTF);

**return** ejbName.substring(0, n);

}

}

Slika 4.2: Korišćenje JNDI Lookup-a za pretragu svih podržanih tipova agenata na jednom čvoru klastera.

**public** **class** GetHostDataService **implements** Runnable {

**private** HandshakeRequestSender requestSender;

**private** HeartbeatInvoker heartbeat;

**private** JndiTreeParser jndiTreeParser;

**private** AgentsService agentsService;

**private** Host host;

**private** String mainNodeDetails;

@SuppressWarnings("unused")

**private** String ip;

**private** String hostname;

**private** **int** portOffset;

**private** AdminConsoleRequestSender adminRequestSender;

@SuppressWarnings("unused")

**private** ResultPredictionService resultPredictionService;

**public** GetHostDataService (String ip, String hostname, AgentsService as,

JndiTreeParser jtp, HeartbeatInvoker hbi, HandshakeRequestSender rhs) {

**this**.hostname = hostname;

**this**.ip = ip;

**this**.mainNodeDetails = "";

**this**.host = **null**;

**this**.portOffset = 0;

**this**.agentsService = as;

**this**.jndiTreeParser = jtp;

**this**.heartbeat = hbi;

**this**.heartbeat.init(**this**.agentsService);

**this**.requestSender = rhs;

}

@Override

**public** **void** run() {

**try** {

Thread.*sleep*(15000);

} **catch**(Exception e) {

e.printStackTrace();

}

**try** {

**this**.host = getHostData();

} **catch** (InstanceNotFoundException | AttributeNotFoundException | MalformedObjectNameException

| ReflectionException | MBeanException | CommandLineException e) {

e.printStackTrace();

}

**this**.mainNodeDetails = getMainNodeDetails();

**boolean** isRunning = sendAdminRequest(**this**.portOffset);

**if**(isRunning) {

**if**(!**this**.mainNodeDetails.equals(**this**.host.getHostAddress())) {

setSlaveData();

requestSender.registerSlaveNode(**this**.mainNodeDetails, **this**.host);

}

**else** {

setMasterData();

}

ScheduledExecutorService executor = Executors.*newScheduledThreadPool*(1);

executor.scheduleAtFixedRate(heartbeat, 180, 180, TimeUnit.***SECONDS***);

}

**else** {//kill the app

System.*exit*(1);

}

}

**private** **void** setMasterData() {

**this**.agentsService.setMainNode(**this**.host);

**this**.agentsService.setMyHostInfo(**this**.host);

**this**.agentsService.setNodeType(NodeType.***MASTER***);

ArrayList<AgentType> myAgentTypes = **new** ArrayList<AgentType>();

**try** {

myAgentTypes = (ArrayList<AgentType>)jndiTreeParser.parse();

} **catch** (NamingException e) {

e.printStackTrace();

}

**this**.agentsService.setMySupportedAgentTypes(myAgentTypes);

List<AgentTypeDTO> dtos = **new** ArrayList<AgentTypeDTO>();

**this**.agentsService.getMySupportedAgentTypes().stream().forEach(type -> {

AgentTypeDTO dto = **new** AgentTypeDTO();

dto.convertToDTO(type, **this**.host);

});

**this**.agentsService.setAllSupportedAgentTypes(dtos);

**for**(Iterator<AgentType> i = myAgentTypes.iterator(); i.hasNext();) {

AgentTypeDTO listItem = **new** AgentTypeDTO();

listItem.convertToDTO(i.next(), **this**.host);

**this**.agentsService.getAllSupportedAgentTypes().add(listItem);

}

}

**private** **void** setSlaveData() {

**this**.agentsService.setMyHostInfo(**this**.host);

**this**.agentsService.setNodeType(NodeType.***SLAVE***);

Host mainNode = **new** Host(**this**.mainNodeDetails, "mainNode");

**this**.agentsService.setMainNode(mainNode);

ArrayList<AgentType> myAgentTypes = **new** ArrayList<AgentType>();

**try** {

myAgentTypes = (ArrayList<AgentType>)jndiTreeParser.parse();

} **catch** (NamingException e) {

e.printStackTrace();

}

**this**.agentsService.setMySupportedAgentTypes(myAgentTypes);

List<AgentTypeDTO> dtos = **new** ArrayList<AgentTypeDTO>();

**this**.agentsService.getMySupportedAgentTypes().stream().forEach(type -> {

AgentTypeDTO dto = **new** AgentTypeDTO();

dto.convertToDTO(type, **this**.host);

});

**this**.agentsService.setAllSupportedAgentTypes(dtos);

**for**(Iterator<AgentType> i = myAgentTypes.iterator(); i.hasNext();) {

AgentTypeDTO listItem = **new** AgentTypeDTO();

listItem.convertToDTO(i.next(), **this**.host);

**this**.agentsService.getAllSupportedAgentTypes().add(listItem);

}

}

**public** **boolean** sendAdminRequest(**int** portOffset) {

**boolean** isUpAndRunning = **false**;

**int** currentManagementPing = 0;

adminRequestSender = **new** AdminConsoleRequestSender();

isUpAndRunning = adminRequestSender.isWildflyRunning(**this**.host.getHostAddress(), portOffset);

**try** {

**while**(!isUpAndRunning && currentManagementPing < 50) {

Thread.*sleep*(500);

isUpAndRunning = adminRequestSender.isWildflyRunning(**null**, portOffset);

}

**if**(!isUpAndRunning) {

System.***out***.println("Application start failed.\nReason 1: You must have a management user.\nReason 2: Deploy failed");

System.*exit*(1);

}

} **catch**(Exception e) {

e.printStackTrace();

}

**return** isUpAndRunning;

}

**public** Host getHostData() **throws** CommandLineException, InstanceNotFoundException, AttributeNotFoundException, MalformedObjectNameException, ReflectionException, MBeanException {

Host ret = **new** Host();

String port;

String host;

**int** portOffset;

port = ManagementFactory.*getPlatformMBeanServer*()

.getAttribute(**new** ObjectName("jboss.as:socket-binding-group=standard-sockets,socket-binding=http"), "port")

.toString();

host = ManagementFactory.*getPlatformMBeanServer*()

.getAttribute(**new** ObjectName("jboss.as:interface=public"), "inet-address")

.toString();

portOffset = Integer.*parseInt*(ManagementFactory.*getPlatformMBeanServer*()

.getAttribute(**new** ObjectName("jboss.as:socket-binding-group=standard-sockets"), "port-offset")

.toString());

**int** portValue = Integer.*parseInt*(port) + portOffset;

**this**.portOffset = portOffset;

**this**.agentsService.setPortOffset(portOffset);

String address = host.split(":")[0] + ":" + portValue;

String alias = host + "|" + **this**.hostname;

ret.setAlias(alias);

ret.setHostAddress(address);

**return** ret;

}

**public** String getMainNodeDetails() {

String resourcePath = "mainNodeData/mainNodeInfo.txt";

URL url = **this**.getClass().getClassLoader().getResource(resourcePath);

System.***out***.println("READING MAIN NODE DATA FROM: " + url.getPath());

InputStream resource = **this**.getClass().getClassLoader().getResourceAsStream(resourcePath);

String line = **null**;

**try** {

BufferedReader bufferedReader = **new** BufferedReader(**new** InputStreamReader(resource));

line = bufferedReader.readLine();

} **catch**(Exception e) {

e.printStackTrace();

}

**if**(line == **null**) {

System.***out***.println("ERROR WHLE READING THE MAIN NODE DATA!");

**return** "ERROR";

}

**else**

**return** line;

}

}

Slika 4.3: Klasa za dobavljanje informacija o serveru i računaru na kom je server pokrenut, odlučivanju da li je čvor master ili slave. Takođe, posao ove klase je da sačuva ove informacije u klasi Host.

@Singleton

@LocalBean

**public** **class** JMSTopic **implements** MessageListener {

**private** **final** String REMOTE\_CONNECTION\_FACTORY = "java:jboss/exported/jms/RemoteConnectionFactory";

**private** **final** String JMSTopic\_NAME = "java:jboss/agentTopic";

**private** **final** String USER = "guest";

**private** Topic topic;

**private** MessageConsumer consumer;

**private** MessageProducer producer;

**private** Connection connection;

**private** Session session;

@PostConstruct

**public** **void** construct() {

**try** {

Context context = **new** InitialContext();

ConnectionFactory cf = (ConnectionFactory) context.lookup(REMOTE\_CONNECTION\_FACTORY);

topic = (Topic) context.lookup(JMSTopic\_NAME);

context.close();

connection = cf.createConnection(USER, USER);

session = connection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);

connection.start();

consumer = session.createConsumer(topic);

consumer.setMessageListener(**this**);

producer = session.createProducer(topic);

} **catch** (JMSException ex) {

**throw** **new** IllegalStateException(ex);

} **catch** (NamingException e) {

**throw** **new** IllegalStateException(e);

}

}

@PreDestroy

**public** **void** closeAllConnections() {

**try** {

consumer.close();

producer.close();

connection.close();

session.close();

} **catch** (JMSException ex) {

**throw** **new** IllegalStateException(ex);

}

}

@Override

**public** **void** onMessage(Message msg) {}

**public** **void** send(ACLMessage msg) {

ObjectMessage objMsg;

**try** {

objMsg = **this**.session.createObjectMessage(msg);

**this**.producer.send(objMsg);

} **catch** (JMSException e) {

e.printStackTrace();

}

}

}

Slika 4.4: JMSTopic konfiguraciona klasa za dobavljanje konkretne instance klase JMSTopic. Koristi se JNDI Lookup za ove potrebe. Priključuje čvor u JMS klaster.

@Stateful

**public** **class** HandshakeService {

@Inject

**private** HandshakeRequestSender requestSender;

@Inject

**private** AgentsService agentsService;

**private** List<AgentTypeDTO> supported = **null**;

**public** List<Host> startHandshake(Host newSlave) {

**boolean** isSuccess = **true**;

List<Host> slaves = agentsService.getSlaveNodes();

**for** (Host slave : slaves) {

**if** (slave.equals(newSlave)) {

isSuccess = **false**;

**break**;

}

}

**if** (isSuccess) {

**if**(agentsService.getNodeType().equals(NodeType.***MASTER***)) {

isSuccess = tryHandshake(newSlave);

}

**if**(isSuccess) {

agentsService.getSlaveNodes().add(newSlave);

} **else** {

System.***out***.println("ERROR: Handshake failed. Initializing rollback.");

rollback(newSlave);

}

}

**return** agentsService.getSlaveNodes();

}

**private** **boolean** tryHandshake(Host newSlave) {

**boolean** isSuccess = **true**;

**int** currentHandshakeAttempt = 0;

**int** option = 0;

**if**(agentsService.getNodeType().equals(NodeType.***MASTER***)) {

System.***out***.println("Initializing handshake for slave: " + newSlave.getAlias());

**while**(currentHandshakeAttempt < 3 && option <= 5) {

**try** {

**switch**(option) {

**case** 0: {

supported = requestSender.fetchAgentTypeList(newSlave.getHostAddress());

agentsService.addNewAgentTypes(supported);

**break**;

}

**case** 1: isSuccess = sendRegisteredSlaveToSlaves(newSlave); **break**;

**case** 2: isSuccess = sendNewAgentTypesToAllSlaves(supported); **break**;

**case** 3: isSuccess = sendSlaveListToNewSlave(newSlave.getHostAddress()); **break**;

**case** 4: isSuccess = sendAgentTypesToNewSlave(newSlave.getHostAddress(), agentsService.getAllSupportedAgentTypes()); **break**;

**case** 5: isSuccess = sendRunningAgentsToNewSlave(newSlave.getHostAddress(), agentsService.getAllRunningAgents()); **break**;

}

} **catch**(Exception e) {

e.printStackTrace();

isSuccess = **false**;

}

**if**(!isSuccess) {

currentHandshakeAttempt++;

**try** {

System.***out***.println("Retrying handshake in 1.5s");

Thread.*sleep*(1500);

isSuccess = **true**;

} **catch**(Exception e) {

System.***out***.println("Error in handshake, at sleeping");

e.printStackTrace();

}

**continue**;

} **else** {

option++;

}

}

}

**return** isSuccess && currentHandshakeAttempt < 3;

}

**private** **void** rollback(Host newSlave) {

String alias = newSlave.getHostAddress();

List<AgentTypeDTO> toDelete = **new** ArrayList<AgentTypeDTO>();

agentsService.getAllSupportedAgentTypes().forEach(type -> {

**if**(type.getAlias().equals(alias) || type.getHostAddress().equals(alias)) {

toDelete.add(type);

}

});

agentsService.getAllSupportedAgentTypes().removeAll(toDelete); //Delete supported agents

agentsService.getSlaveNodes().removeIf(x -> x.getHostAddress().equals(alias)); //Delete node

agentsService.getSlaveNodes().forEach(slave -> {

requestSender.deleteBadNode(slave.getHostAddress(), alias);

});

}

**private** **boolean** sendRunningAgentsToNewSlave(String hostAddress, List<AID> allRunningAgents) {

**return** requestSender.sendAllRunningAgentsToNewSlave(hostAddress, allRunningAgents);

}

**private** **boolean** sendAgentTypesToNewSlave(String hostAddress, List<AgentTypeDTO> allSupportedAgentTypes) {

**return** requestSender.sendAgentTypesToNewSlave(hostAddress, allSupportedAgentTypes);

}

**private** **boolean** sendSlaveListToNewSlave(String hostAddress) {

**return** requestSender.sendExistingSlavesToNewSlave(hostAddress, **this**.agentsService.getSlaveNodes());

}

**public** **boolean** sendRegisteredSlaveToSlaves(Host newSlave) {

**boolean** isSuccess = **true**;

**if**(agentsService.getNodeType().equals(NodeType.***MASTER***)) {

**for**(Host slave : agentsService.getSlaveNodes()) {

isSuccess = requestSender.registerSlaveNode(slave.getHostAddress(), newSlave) != **null**;

**if**(!isSuccess) {

**break**;

}

}

}

**return** isSuccess;

}

**public** List<AgentType> fetchAgentTypeList() {

**try** {

**return** agentsService.getMySupportedAgentTypes();

} **catch**(Exception e) {

**return** **null**;

}

}

**public** **boolean** sendNewAgentTypesToAllSlaves(List<AgentTypeDTO> newTypes) {

**boolean** success = agentsService.getNodeType().equals(NodeType.***MASTER***);

**if**(success) {

List<AgentType> toSend = **new** ArrayList<AgentType>();

newTypes.stream().forEach(t -> toSend.add(**new** AgentType(t.getName(), t.getModule())));

**for**(Host slave : agentsService.getSlaveNodes()) {

success = requestSender.sendNewAgentTypesToSlave(slave.getHostAddress(), toSend);

**if**(!success) {

System.***err***.println("Error at sending agent types to slave with host: " + slave.getHostAddress());

**break**;

}

}

}

**return** success;

}

**public** **boolean** addNewAgentTypes(List<AgentTypeDTO> agentTypes) **throws** JsonProcessingException, IOException {

List<AgentTypeDTO> ret = agentsService.addNewAgentTypes(agentTypes);

**return** ret != **null**;

}

**public** ArrayList<AgentInterface> sendRunningAgents(String myHostAddress){

ArrayList<AgentInterface> retList = **new** ArrayList<AgentInterface>();

//add my running agents

**for** (Iterator<AgentInterface> i = agentsService.getMyRunningAgents().iterator(); i.hasNext();)

retList.add(i.next());

//only sending to other slaves (if I am the main node I will skip the slave who initiated the call)

**for** (Iterator<Host> h = agentsService.getSlaveNodes().iterator(); h.hasNext();) {

Host item = h.next();

**if**(!item.getHostAddress().equals(myHostAddress)) {

List<AgentInterface> respAgents = requestSender.getRunningAgents(item.getHostAddress());

**for**(Iterator<AgentInterface> ra = respAgents.iterator(); ra.hasNext();)

retList.add(ra.next());

}

}

//i am a slave node, send also to the main node

**if**(agentsService.getNodeType().equals(NodeType.***SLAVE***)) {

Host main = agentsService.getMainNode();

List<AgentInterface> respAgents = requestSender.getRunningAgents(main.getHostAddress());

**for**(Iterator<AgentInterface> ra = respAgents.iterator(); ra.hasNext();)

retList.add(ra.next());

}

**return** retList;

}

**public** **boolean** deleteNode(String alias){

**boolean** success = **true**;

**try** {

success = agentsService.deleteNode(alias);

} **catch**(Exception e) {

success = **false**;

e.printStackTrace();

System.***out***.println("Error deleting node. :(");

}

**return** success;

}

}

Slika 4.5: Implementacija prijave čvora na klaster.

## Agent

Agenti su realizovani kao Remote Beanovi, što omogućava CORBA podsistemu da pozove njihove metode za rukovanje porukama. MDBConsumer klasa reaguje na poruke koje su dostavljene u JMSTopic pod namespace-om java:jboss/agentTopic. Ova klasa dalje delegira poruku agentima u okruženju. Svaki agent ima svoj AID, u kom se dalje kriju potrebni podaci o samom agentu, i čvoru na kom agent trči. U zavisnosti od naziva performative koja je dostavljena u poruci, agenti mogu da ignorišu ili da odreaguju na dobijenu poruku. U celom agentskom okruženju, postoji sedam agenata, među kojima su nama zanimljivi MapAgent i ReduceAgent.

Implementacija MDBConsumer klase koja reaguje na poruke, data je na slici 4.6.

@MessageDriven(name = "MDBConsumer", activationConfig = {

@ActivationConfigProperty(propertyName = "destinationLookup", propertyValue = "topic/agentTopic"),

@ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Topic"),

@ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-acknowledge") })

**public** **class** MDBConsumer **implements** MessageListener{

@Inject

**private** AgentsService agentsService;

@Override

**public** **void** onMessage(Message msg) {

**try** {

processMessage(msg);

} **catch** (JMSException e) {

e.printStackTrace();

//WEBSOCKET MESSAGE?

} **catch** (InstantiationException e) {

e.printStackTrace();

//WEBSOCKET MESSAGE?

} **catch** (IllegalAccessException e) {

e.printStackTrace();

//WEBSOCKET MESSAGE?

} **catch** (ClassNotFoundException e) {

e.printStackTrace();

//WEBSOCKET MESSAGE?

}

}

**private** **void** processMessage(Message msg) **throws** JMSException, InstantiationException,

IllegalAccessException, ClassNotFoundException {

ACLMessage acl = (ACLMessage) ((ObjectMessage) msg).getObject();

ArrayList<AID> receiversList = acl.getReceivers();

**for**(Iterator<AID> i = receiversList.iterator(); i.hasNext();)

deliverMessage(acl, i.next());

}

@SuppressWarnings("deprecation")

**private** **void** deliverMessage(ACLMessage msg, AID aid) **throws** InstantiationException,

IllegalAccessException, ClassNotFoundException {

**for** (AgentInterface agent : agentsService.getMyRunningAgents()) {

**if**(agent.getClass().isInstance(Class.*forName*("beans." + aid.getType().getName()).~~newInstance~~())) {

AgentClass agentObj = (AgentClass) Class.*forName*("beans." + aid.getType().getName()).cast(agent);

AID myAid = agentObj.getAid();

**if**(myAid.getHost().getAlias().equals(aid.getHost().getAlias())

&& myAid.getHost().getHostAddress().equals(aid.getHost().getHostAddress())

&& myAid.getName().equals(aid.getName())) {

agent.handleMessage(msg);

**break**;

}

}

}

}

}

Slika 4.6: MDBConsumer klasa. Metoda deliverMessage šalje poruku onim agentima koji su navedeni u ACLMessage poruci.

### MapAgent

MapAgent je agent koji odrađuje map deo posla u MapReduce programerskom modelu. Prima ACLMessage poruku u kojoj stoji određena performativa. Ako je ta performativa CALL\_FOR\_PROPOSAL, MapAgent će odreagovati na poruku i iskoristiće MapService servis. U ACLMessage poruci stoji naziv fajla treba da se pročita, i koji sve MapAgent-i treba da budu uključeni u posao. U zavisnosti od pozicije u listi Map Agenata, i broja Map Agenata, Map Agent će odrediti početnu i krajnju tačku obrade fajla. Obrada se sastoji u tokenizaciji čitavog fajla, prebrojavanju tokena i smeštanja rezultata u asocijativnu mapu. Ovu asocijativnu mapu šalje Reduce Agentu, o kom će biti više reči u narednom poglavlju. Po završetku Map posla, agent šalje ACLMessage poruku Reduce agentu sa performativom PROPAGATE, na koju on reaguje.

Pomenuta asocijativna mapa se sastoji od

Implementacija MapAgent agenta je data na slici 4.7. On šalje sve potrebne podatke MapService servisu, koji je dalje zadužen za obradu traženog fajla.

@Stateful

@Remote(AgentInterface.**class**)

**public** **class** MapAgent **extends** AgentClass{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** AID aid;

**private** JMSTopic jmsTopic;

@SuppressWarnings("unused")

**private** AgentsService agentsService;

**private** MapService mapService;

**private** HashMap<String, Integer> counts;

@Override

**public** **void** handleMessage(ACLMessage message) {

**int** mapNumber = countMappers(message.getReceivers());

**if**(message.getPerformative().equals(Performative.***CALL\_FOR\_PROPOSAL***) && mapNumber > 0) {

**try** {

System.***out***.println(**this**.aid.getName() + " Invoked on : " + **this**.agentsService.getMyHostInfo().getHostAddress());

**int** positionInArray = positionInList(message.getReceivers());

**if**(positionInArray > -1) {

System.***out***.println("Turning on Mapper: " + **this**.aid.getName());

mapService = **new** MapService();

mapService.init();

mapService.createKeyValuePairs(message.getContent(), positionInArray, mapNumber);

ACLMessage reducerMsg = **new** ACLMessage();

reducerMsg.setConversationId(message.getConversationId());

reducerMsg.setPerformative(Performative.***PROPAGATE***);

reducerMsg.setReceivers(message.getReceivers());

reducerMsg.setSender(**this**.aid);

reducerMsg.setReplyTo(message.getSender());

HashMap<String, Object> userArgs = createMapReduceDetails(mapService.getCounts(), message);

reducerMsg.setUserArgs(userArgs);

jmsTopic.send(reducerMsg);

}

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

**private** **int** positionInList(ArrayList<AID> receivers) {

**int** ret = 0;

**for**(**int** i = 0; i < receivers.size(); i++) {

**if**(receivers.get(i).getName().equals(**this**.aid.getName())) {

**break**;

} **else** **if**(receivers.get(i).getType().getName().equals("MapAgent")) {

ret++;

}

}

**return** ret;

}

**private** HashMap<String, Object> createMapReduceDetails(HashMap<String, Integer> details, ACLMessage message) {

HashMap<String, Object> ret = **new** HashMap<String, Object>();

**for**(**int** i = 0; i < message.getReceivers().size(); i++) {

**if**(message.getReceivers().get(i).getType().getName().equals("ReduceAgent")) {

ret.put(message.getReceivers().get(i).getName(), details);

}

}

**return** ret;

}

**private** **int** countMappers(ArrayList<AID> receivers) {

**int** mapNumber = 0;

**for**(AID aid : receivers) {

System.***out***.println(aid.getName() + " : " + aid.getType().getName());

**if**(aid.getType().getName().equals("MapAgent")) {

mapNumber++;

}

}

System.***out***.println("Number of mappers: " + mapNumber);

**return** mapNumber;

}

}

Slika 4.7: Implementacija MapAgent agenta. Agent mora biti Stateful bean zbog pravilnog pozivanja metoda JMSTopic.

Fajlovi koje Map agenti mogu da pročitaju su tekstualni fajlovi pod nazivom big.txt, virus.txt i funny.txt. Oni su upakovani u .WAR datoteku. Da bi smo pročitali sadržaj ove datoteke, moramo da iskoristimo kod na slici 4.8 za lociranje fajla:

Path temp = Files.*createTempFile*(filename, ".txt");

Files.*copy*(**this**.getClass().getClassLoader().getResourceAsStream(fileLocation + filename), temp, StandardCopyOption.***REPLACE\_EXISTING***);

Slika 4.8: Kod za pravilno čitanje fajla upakovanog u .WAR datoteku.

Implementacija MapService servisa data je na slici 4.9.

@Stateful

**public** **class** MapService {

**private** **final** String fileLocation = "largeTextFiles/";

**private** HashMap<String, Integer> counts;

@PostConstruct

**public** **void** init() {

counts = **new** HashMap<String, Integer>();

}

**public** **void** createKeyValuePairs(String filename, **int** mapperPosition, **int** totalMapperNumber) **throws** URISyntaxException, IOException {

Path temp = Files.*createTempFile*(filename, ".txt");

Files.*copy*(**this**.getClass().getClassLoader().getResourceAsStream(fileLocation + filename), temp, StandardCopyOption.***REPLACE\_EXISTING***);

**try**(RandomAccessFile raf = **new** RandomAccessFile(temp.toFile(), "r")) {

String text = readFile(raf, mapperPosition, totalMapperNumber);

System.***out***.println(text);

String[] words = text.trim().split("[\\r\\n\\t\\s]+");

**for**(**int** i = 0; i < words.length; i++) {

addToMap(words[i]);

}

} **catch**(Exception e) {

System.***err***.println("Cant read file: " + temp.toFile().getAbsolutePath());

**return**;

}

}

**private** String readFile(RandomAccessFile raf, **int** mapperPosition, **int** totalMapperNumber) **throws** IOException {

**boolean** isLastMapper = (mapperPosition + 1) == totalMapperNumber;

**byte**[] buffer = createTextBuffer(raf, isLastMapper, mapperPosition, totalMapperNumber);

String text = **new** String(buffer);

**boolean** lastCharIsLetter = !Character.*isWhitespace*(text.charAt(text.length() - 1));

text = lastCharIsLetter ? text : text.trim();

**try** {

**while**(lastCharIsLetter && !isLastMapper) {

Character c = (**char**)raf.read();

lastCharIsLetter = !Character.*isWhitespace*(c);

text += c;

}

} **catch**(Exception e) {

System.***out***.println("Stopping seek, probably EOF");

}

**return** text;

}

**private** **byte**[] createTextBuffer(RandomAccessFile raf, **boolean** isLastMapper, **int** mapperPosition, **int** totalMapperNumber) **throws** IOException {

**byte**[] buffer;

Long fileLengthInBytes = raf.length();

Long nthOfFile = Math.*floorDiv*(fileLengthInBytes, totalMapperNumber);

Long seekToPosition = (**long**)0;

**if**(mapperPosition > 0) {

seekToPosition = nthOfFile \* mapperPosition;

raf.seek(seekToPosition);

**try** {

**char** c = (**char**)raf.read();

**while**(!Character.*isWhitespace*(c) && seekToPosition <= fileLengthInBytes) {

seekToPosition++;

c = (**char**)raf.read();

}

} **catch**(Exception e) {

e.printStackTrace();

System.***out***.println("Something went wrong while seeking, method : createTextBuffer");

}

**if**(isLastMapper) {

nthOfFile = fileLengthInBytes - (nthOfFile \* mapperPosition);

}

} **else** **if**(mapperPosition == 0 && isLastMapper) {

nthOfFile = fileLengthInBytes;

}

buffer = **new** **byte**[nthOfFile.intValue()];

raf.read(buffer, 0, nthOfFile.intValue());

**return** buffer;

}

**private** **void** addToMap(String word) **throws** IOException {

word = word.trim().replaceAll("[^A-Za-z0-9]", "");

**if**(word.length() > 0 && word != **null** && !word.equals("") && !word.equals(" ")) {

Integer count = counts.get(word);

**if**(count == **null**) {

count = 1;

} **else** {

count += 1;

}

counts.put(word, count);

}

}

**public** String getFileLocation() {

**return** fileLocation;

}

**public** HashMap<String, Integer> getCounts() {

**return** counts;

}

}

Slika 4.8: Implementacija MapService servisa, zaduženog za obradu tražene tekstualne datoteke

### ReduceAgent

ReduceAgent je agent koji obavlja Reduce deo posla u MapReduce programerskom modelu. Regauje na poruku sa performativom PROPOSE koju dobija isključivo od MapAgent-a. Njegov posao je da uzme izlaz MapAgent-a u vidu asocijativne mape, i da kreira novu asocijativnu mapu od svih MapAgent asocijativnih mapa. Reduce asocijativna mapa se sastoji od konačnog broja tokena u datoteci. Tek kada svi MapAgent-i budu poslali svoj izlaz ReduceAgent-u, i on sve te izlaze obradi, on će konačnu asocijativnu mapu vratiti korisniku kroz aktivni WebSocket u vidu Message klase.

Message klasa će kao svoj sadržaj imati asocijativnu mapu reduce agenta prebačenu u string, radi bolje čitljivosti poruke u pretraživaču. AgentClass već ima metodu za vraćanje odgovora korisniku preko WebSocketa, ali parsiranje ove poruke bi bilo teže na klijentskoj strani aplikacije. Za obradu mape koja je izlaz Map agenta, zadužen je ReduceService servis.

Implementacija ReduceAgent agenta je data na slici 4.9.

@Stateful

@Remote(AgentInterface.**class**)

**public** **class** ReduceAgent **extends** AgentClass{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** AID aid;

**private** JMSTopic jmsTopic;

**private** AgentsService agentsService;

**private** ReduceService reduceService;

**private** ObjectMapper mapper = **new** ObjectMapper();

@Override

**public** **void** handleMessage(ACLMessage message) {

**if**(message.getPerformative().equals(Performative.***PROPAGATE***)

&& message.getSender().getType().getName().equals("MapAgent")) {

**boolean** shouldReset;

System.***out***.println(**this**.aid.getName() + " Invoked on : " + **this**.agentsService.getMyHostInfo().getHostAddress());

**if**(reduceService == **null**) {

reduceService = **new** ReduceService();

}

System.***out***.println("Reduce invoked on: " + agentsService.getMyHostInfo().getHostAddress() + " by: " + message.getSender().getName());

HashMap<String, Object> userArgs = message.getUserArgs();

**for**(String key : userArgs.keySet()) {

Object fromMap = userArgs.get(key);

**if**(fromMap **instanceof** HashMap) {

HashMap<String, Integer> mapperOutput = (HashMap<String, Integer>) fromMap;

**try** {

reduceService.countOccurences(mapperOutput, message.getSender());

} **catch**(Exception e) {

e.printStackTrace();

}

}

}

shouldReset = reduceService.areAllMappersProcessed(getMappers(message));

System.***out***.println("Should Reset: " + shouldReset);

**if**(shouldReset) {

String content = reduceService.createHugeString().trim();

System.***out***.println(content);

**if**(content != "" && content != **null**) {

Message msg = **new** Message("aclMessage", "\*\*\*\*\*\*\*\*\*\*\*\*RESPONSE\*\*\*\*\*\*\*\*\*\* " + content);

Iterator<Session> iterator = WebSocketController.*sessions*.iterator();

System.***out***.println(agentsService.getMyHostInfo().getHostAddress() + " is sending a response!");

**int** i = 0;

**while**(iterator.hasNext()) {

System.***out***.println(agentsService.getMyHostInfo().getHostAddress() + " iterator: " + i++);

Session s = iterator.next();

**try** {

s.getBasicRemote().sendText(mapper.writeValueAsString(msg));

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

reduceService.resetAll();

}

}

}

}

**private** List<AID> getMappers(ACLMessage message) {

List<AID> mappers = **new** ArrayList<AID>();

**for**(AID mapper : message.getReceivers()) {

**if**(mapper.getType().getName().equals("MapAgent")) {

mappers.add(mapper);

}

}

**return** mappers;

}  
}

Slika 4.9: Implementacija ReduceAgent agenta. Posle obrade svih Map agenata, potrebno je resetovati stanje Reduce agenta u inicijalno stanje, zbog toga što je ReduceAgent Stateful bean.

Implementacija ReduceService servisa, koji je zadužen za prebrojavanje tokena, data je na slici 4.10.

@Stateful

**public** **class** ReduceService {

**private** HashMap<String, Integer> counts;

**private** List<AID> processedMappers;

**public** ReduceService() {

processedMappers = **new** ArrayList<AID>();

counts = **new** HashMap<String, Integer>();

}

**public** **void** countOccurences(HashMap<String, Integer> words, AID mapper) **throws** IOException {

**if**(mapper.getType().getName().equals("MapAgent")) {

**for**(String key : words.keySet()) {

Integer total;

Integer mapOccurrences = words.get(key);

Integer occurrences = counts.get(key);

**if**(mapOccurrences != **null** && occurrences != **null**) {

total = mapOccurrences + occurrences;

} **else** {

total = mapOccurrences;

}

counts.put(key, total);

}

**if**(!processedMappers.contains(mapper)) {

processedMappers.add(mapper);

}

}

}

**public** **boolean** areAllMappersProcessed(List<AID> mappers) {

**return** processedMappers.size() == mappers.size();

}

**public** **void** resetAll() {

counts = **new** HashMap<String, Integer>();

processedMappers = **new** ArrayList<AID>();

}

**public** **void** writeValues() {

**for**(String key : counts.keySet()) {

Integer count = counts.get(key);

System.***out***.println(key + ": " + count);

}

}

**public** String createHugeString() {

String ret = "";

**for**(String key : counts.keySet()) {

ret += key + ":" + Integer.*toString*(counts.get(key)) + " ";

}

**return** ret;

}

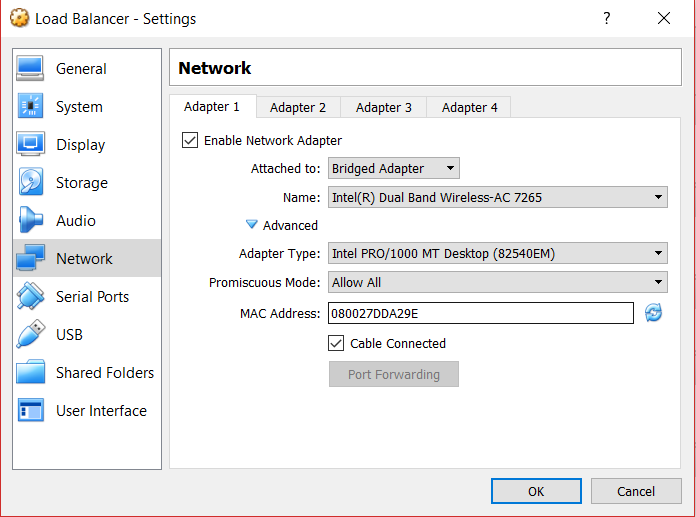
}

Slika 4.10: Implementacija ReduceService servisa, zaduženog za obradu asocijativne mape dobijene iz Map agenta.

## Konfiguracija klastera i balansera opterećenja

Napravićemo klaster koristeći virtuelne mašine na kojima je instaliran Ubuntu 18 operativni sistem. Da bi mašine imale pristup internetu, moramo da iskonfigurišemo mrežni adapter. Moguće je osposobiti četiri adaptera, među kojima treba da se nađu NAT i Bridged Adapter. NAT će virtuelnim mašinama omogućiti pristup internetu, a bridged adapter će omogućiti komunikaciju između Host (Windows 10) i virtuelne mašine. Ako uključimo promiscuous mode, biće moguća komunikacija u oba smera.

Primer konfiguracije mrežnog adaptera je dat na slici 4.11. Komandom ipconfig na Windows 10, a ip add show na Ubuntu operativnom sistemu, možemo da saznamo IP adresu mašine.

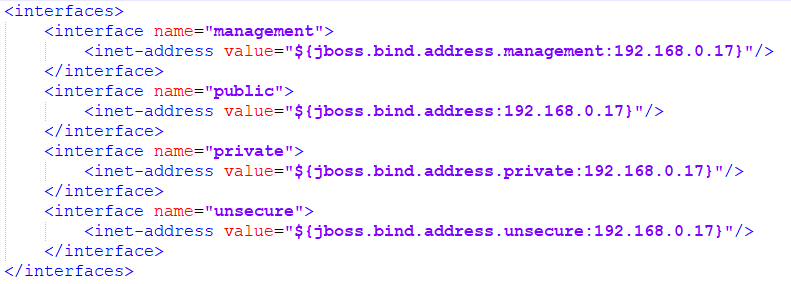
  
Slika 4.11: Konfiguracija Bridged Adaptera na mrežnom adapteru virtuelne mašine

Da bi Wildfly aplikacioni serveri mogli da komuniciraju međusobno koristeći JMS, moramo prvo da dodamo korisničke profile. U aplikacionom serveru mora da postoji jedan management, i jedan application korisnički profil. Management profil će nam omogućiti pristup Wildfly administracionoj konzoli, a Application user će nam omogućiti autentifikaciju za ArtemisQ klaster, koji je kičma JMS podsistema.

Kada dodamo korisnike, moramo da iskonfigurišemo standalone-full-ha profil. Konfiguracioni fajl se nalazi na sledećoj putanji u Wildfly server folderu:

* standalone/configuration/standalone-full-ha.xml

Unutar njega, moramo da promenimo na kojoj IP adresi će biti dostupni interfejsi. Umesto standardne IP adrese 127.0.0.1:8080 (localhost:8080) treba da unesemo IP adresu mašine. Primer konfiguracije interfejsa je dat na slici 4.12. Ovo treba da uradimo za svaku virtuelnu mašinu na kojoj će biti pokrenut aplikacioni server i deployovana aplikacija.

  
Slika 4.12: Konfiguracija interfejsa za rad u klasteru u standalone\*.xml fajlovima

Da bi smo iskonfigurisali balanser opterećenja, menjamo IP adrese interfejsa na isti način u kofiguracionom fajlu standalone-load-balancer.xml. Load balancer je podešen tako da pronađe sve Wildfly aplikacione servere koji su pokrenuti u standalone-full-ha profilu koristeći UDP protokol. Možemo da podesimo i port na kom će aplikacioni server biti pokrenut koristeći XML element koji je prikazan na slici 4.13. Ako je vrednost atributa port-offset=0 to znači da će aplikacioni server biti pokrenut na portu 8080. Ako je port-offset = 1, server će biti pokrenut na portu 8081. Svi ostali portovi (Administraciona i management konzola) biće pomereni za port-offset vrednost.

  
Slika 4.13: Konfiguracija porta na kom će server biti pokrenut

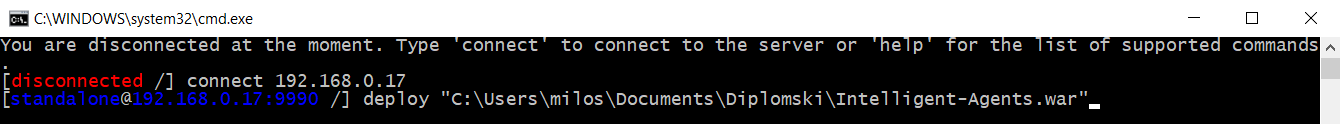
Upotreba JMS podsistema je moguća ako namestimo šifru klastera. Šifra se podešava u konfiguracionom fajlu standalone-full-ha. Primer podešavanja ove šifre je dat na slici 4.14.

  
Slika 4.14: Konfiguracija šifre klastera za korišćenje JMS podsistema

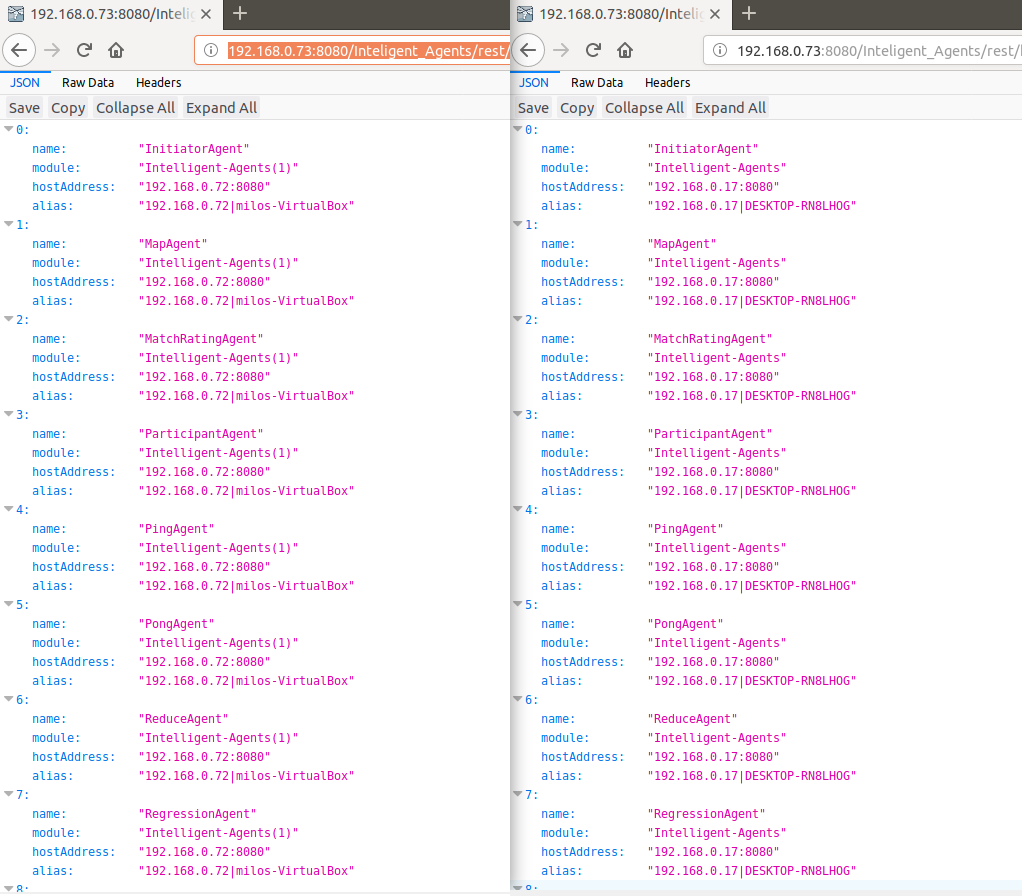
Kada podesimo ove fajlove, treba da uradimo deploy aplikacije na server. Ovo je moguće preko jboss-cli programa. Potrebno je pokrenuti ga iz terminala. Moramo da se konektujemo na server preko komande connect <ip:port>. Kada se konektujemo, zadamo sledeću komandu:

* deploy “lokacija do war fajla” – ovom komandom, .WAR fajl će biti sačuvan u standalone-full-ha.xml fajlu pod deployments XML elementom. Takođe, sadržaj .WAR fajla će biti prisutan na lokaciji standalone/deployments.

Primer deployovanja aplikacije preko ove aplikacije je dat na slici 4.15. Ako komanda prođe uspešno, u terminalu u kom je pokrenut server, možemo videti da je aplikacija uspešno pokrenuta.

  
Slika 4.15: Deploy .WAR fajla u standalone server

Deploy aplikacije radimo na onim čvorovima na kojima je pokrenut standalone-full-ha profil servera. Balanser opterećenja nam služi samo kao proxy zahteva. Pri pokretanju balansera opterećenja, videćemo ispis u konzoli da su se serveri sa full-ha profilom povezali sa njim. Ako pokušamo da pošaljemo zahtev balanseru opterećenja par puta, videćemo da se zahtevi ravnopravno prosleđuju serverima koji su sa njim povezani. Primer agentskog okruženja koje je povezano sa balanserom opterećenja je dato na slici 4.16. Šaljemo zahtev za dobavljanje svih tipova agenata koji se nalaze u agentskom okruženju.

  
Slika 4.16: Slanje jednog istog zahteva ka balanseru opterećenja daje nam različite rezultate

Pored ovoga, potrebno je iskonfigurisati balansiranje opterećenja WebSocket konekcije. U ovome nam Wildfly ne može pomoći, jer ne podržava balansiranje HTTP i WebSocket zahteva u isto vreme.

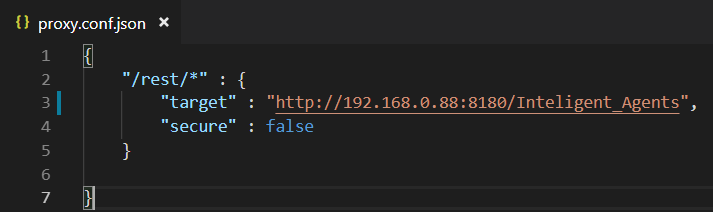
Za potrebe balansiranja opterećenja WebSocket konekcije koristimo NGINX veb server. Posle instalacije, treba ga iskonfigurisati. Potrebno je navesti upstream-ove, koji predstavljaju adrese servera na kojima trče naša agentska okruženja. Takođe, potrebno je iskonfigurisati server da sluša na adresi virtuelne mašine, a zahteve ka NGINX serveru treba prebaciti jednom od upstreamova.

Primer konfiguracije balansiranja opterećenja WebSocket konekcije koristeći NGINX server, dat je na slici 4.17. Konfiguracija NGINX servera ne mora da se odnosi samo na WebSocket konekciju. Sam NGINX server može da služi kao balanser opterećenja i HTTP i WebSocket zahteva. Potrebno je samo iskonfigurisati ga na pravi način.

|  |
| --- |
| upstream agent\_nodes { #Ime upstreamova nije bitno  ip\_hash; #Vrati odgovor WebSocketu koji je poslao zahtev  server 192.168.0.17:8080; #Host mašina (Windows 10)  server 192.168.0.89:8080; #Virtuelna mašina (Ubuntu 18)  }  server {  listen 192.168.0.88:8080; #Na kojoj adresi i portu sluša NGINX server  server\_name intelagents.io; #Alias NGINX servera  proxy\_set\_header Host $http\_host;    location / { #Redirektovati zahtev za ovaj URL  proxy\_http\_version 1.1;  proxy\_set\_header Upgrade $http\_upgrade;  proxy\_set\_header Connection "Upgrade";    proxy\_pass "http://agent\_nodes/Inteligent\_Agents/websocket";  } #agent\_nodes u proxy\_pass URL-u je ustvari ime upstreama  } |

Slika 4.17: Konfiguracija balansiranja opterećenja NGINX servera

Uspostavljanje konekcije Front-end aplikacije sa Wildfly balanserom opterećenja radi se tako što u korenu aplikacije napravimo proxy.conf.json fajl i u njemu napišemo koji URL ustvari treba da gađamo kada pravimo HTTP zahteve. Primer je dat na slici 4.18. Potrebno je da upalimo front end aplikaciju koristeći komandu npm start umesto ng serve da bi proxy.conf.json fajl bio pročitan.

  
Slika 4.18: Podešavanje konekcije ka Wildfly balanseru opterećenja

Da bi smo uspostavili konekciju sa WebSocket balanserom opterećenja, potrebno je samo da navedemo IP adresu na kojoj NGINX server sluša. Uspostavljanje konekcije sa WebSocket balanserom opterećenja prikazano je na slici 4.19.

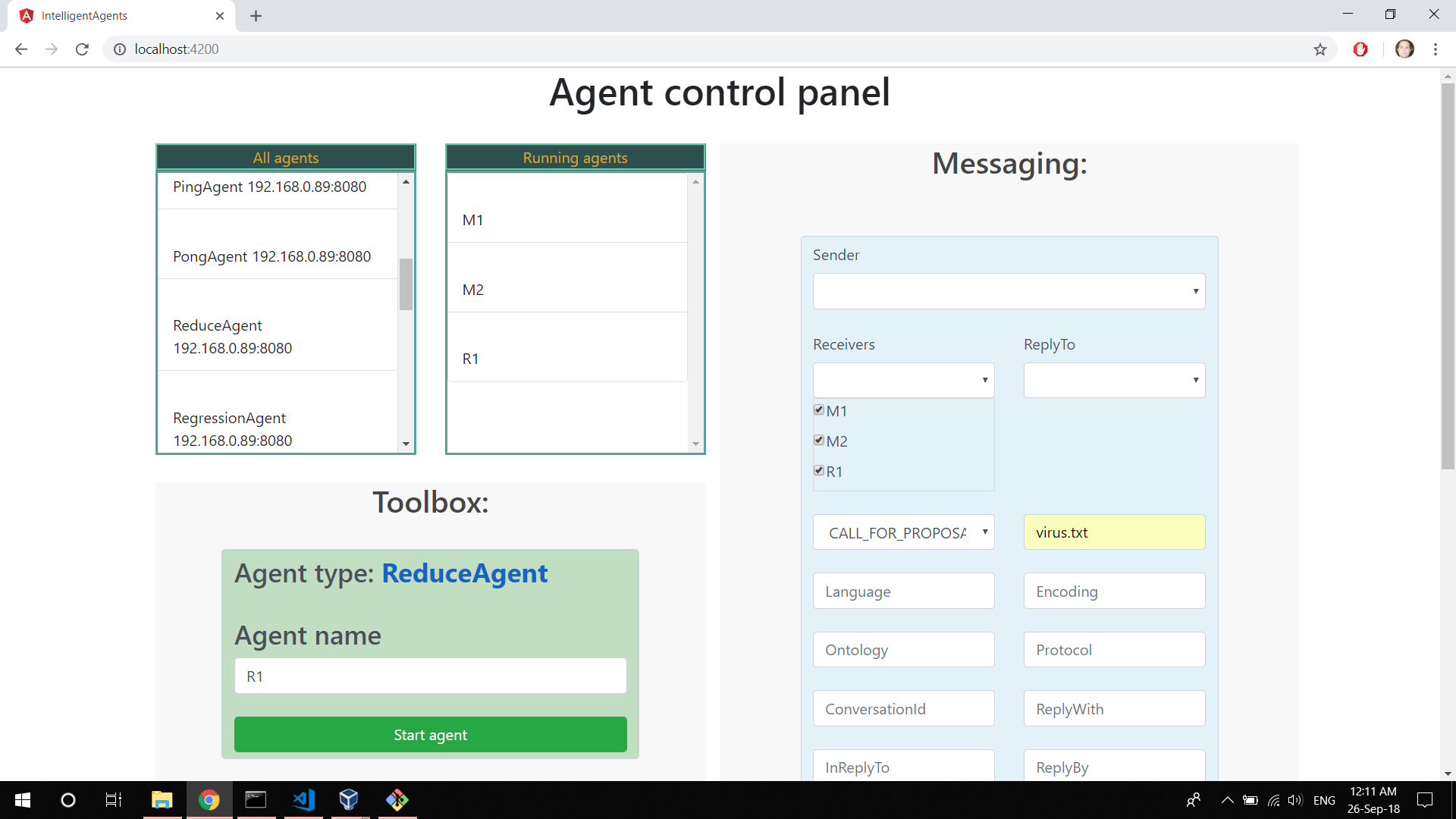
  
Slika 4.19: Podešavanje konekcije ka NGINX WebSocket balanseru opterećenja

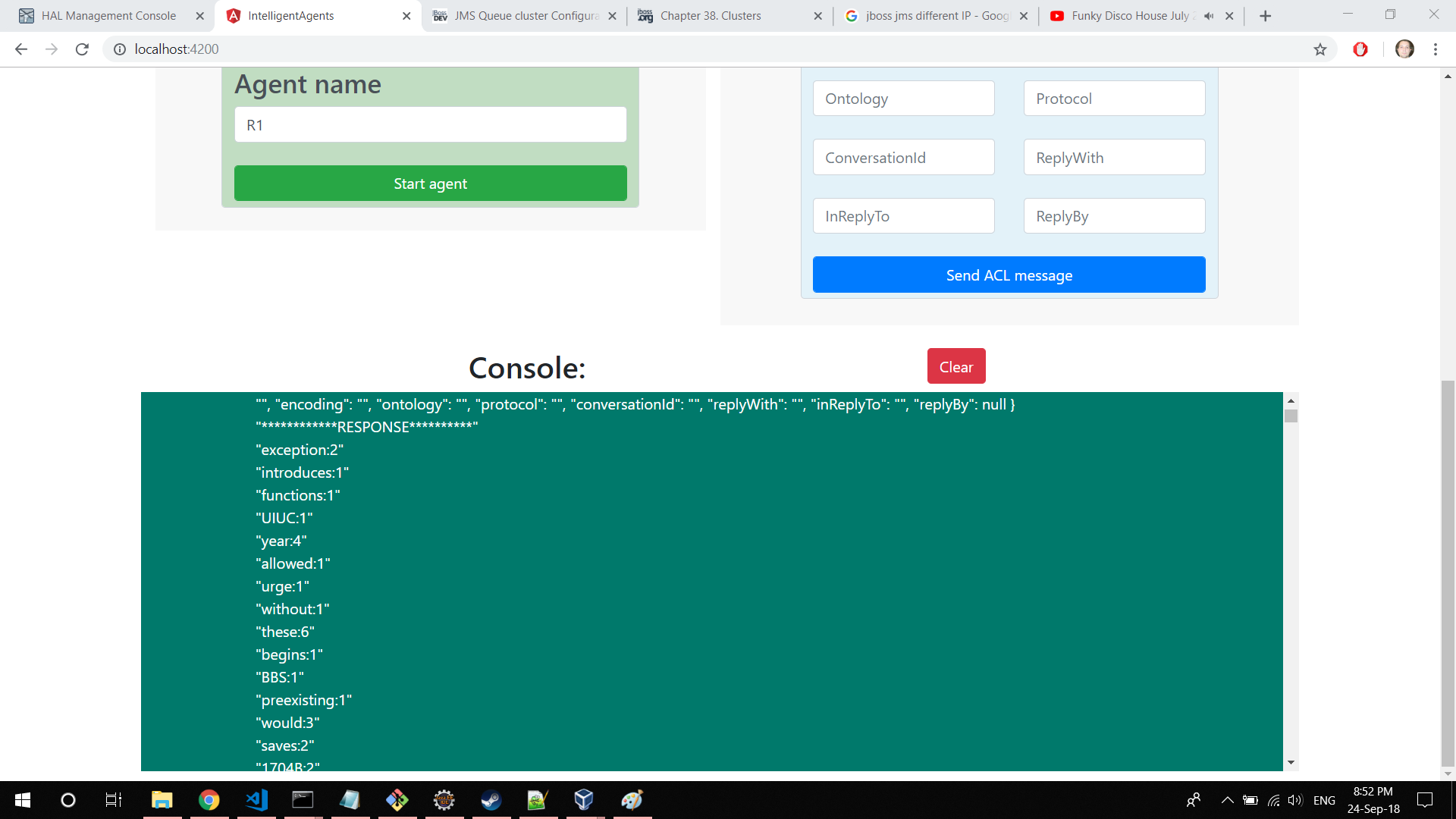
Koristeći korake za konfiguraciju klastera i balansera opterećenja, upalićemo front-end aplikaciju i pripremiti poruku za MapAgent i ReduceAgent. Napravićemo jednog MapAgent-a na host mašini, i jednog na virtuelnoj mašini. Reduce agenta ćemo napraviti na host mašini.

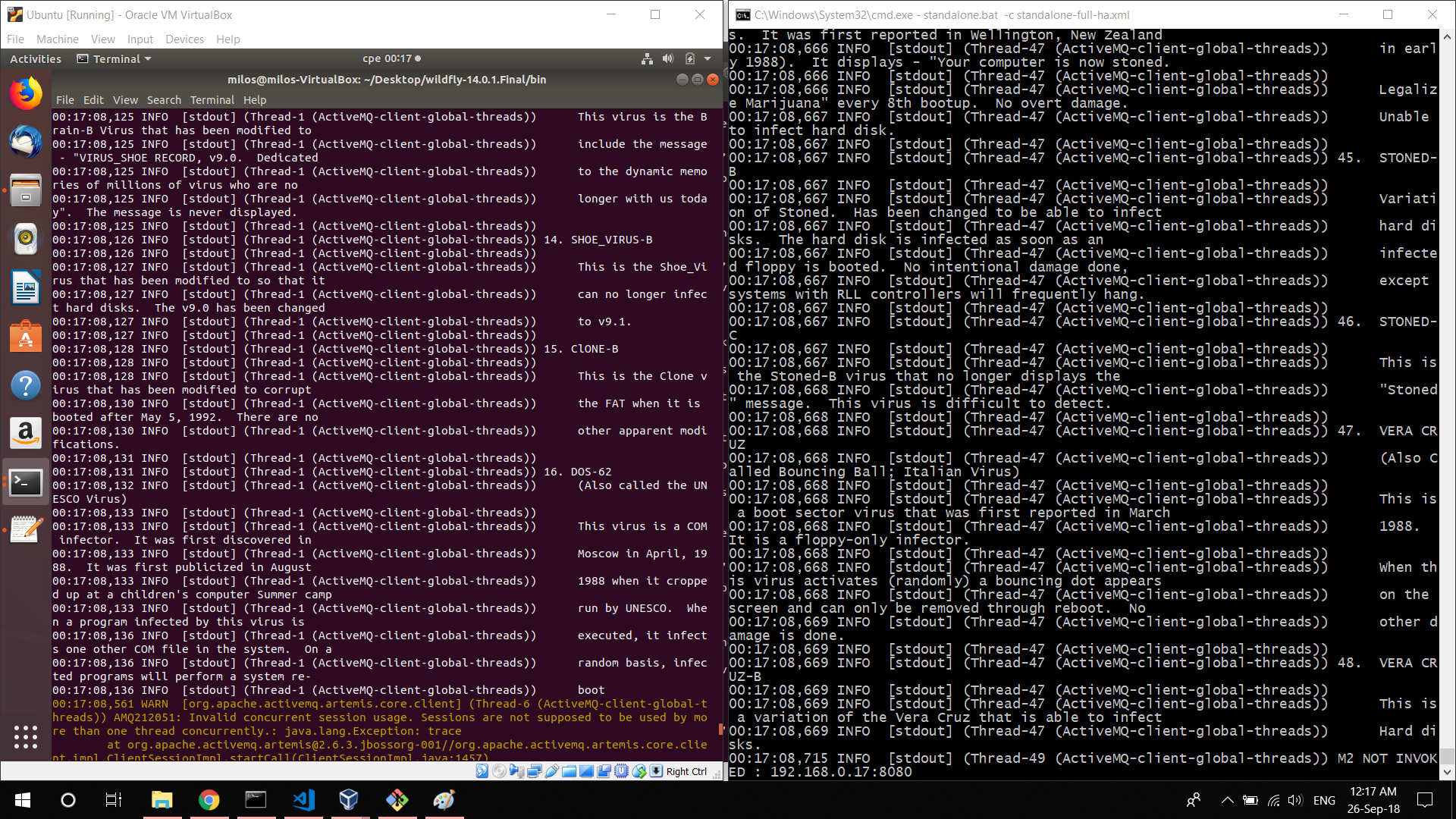
Adrese aplikacionih servera u primeru su sledeće:

* 192.168.0.17:8080 – Wildfly aplikacioni server na host mašini
* 192.168.0.89:8080 – Wildfly aplikacioni server na Ubuntu virtuelnoj mašini
* 192.168.0.88:8180 – Wildfly balanser opterećenja
* 192.168.0.88:8080 – NGINX balanser opterećenja

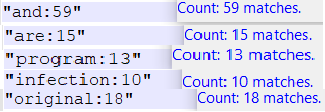
Primer korišćenja front end aplikacije za pravljenje agenata i slanje poruka je dat na slici 4.20.   
Odgovor na poslatu poruku je poslao Reduce agent posle obrade svih asocijativnih mapa. Izlaz iz Reduce agenta je prikazan na slici 4.21. Šta se dešava na aplikacionim serverima je prikazano na slici 4.22.

  
Slika 4.20: Pravljenje agenata i slanje CALL\_FOR\_PROPOSAL poruke koje osposobljava Map agente. Obrađujemo datoteku virus.txt koja se nalazi u .WAR fajlu aplikacije

  
Slika 4.21: Odgovor na poslatu poruku. Ovakav izlaz je izgenerisan iz Reduce agenta.

  
Slika 4.22: Prikaz konzole nakon slanja poruke agentskom okruženju. Leva konzola prikazuje kraj prvog dela fajla. Druga slika pokazuje kraj drugog dela fajla.

Da bi smo potvrdili da naša implementacija MapReduce programerskog modela radi kako treba u ovako iskonfigurisanom okruženju, možemo da uporedimo izlaz Reduce agenta koji smo dobili u konzoli front end aplikacije. Potvrda pravilog rada naše implementacije je data na slici 4.23. Koristimo Notepad++ Count opciju za prebrojavanje reči u tekstualnoj datoteci.

  
Slika 4.23: Potvrda pravilnog rada aplikacije za neke od reči u celom tekstualnom fajlu virus.txt

# Zaključak

# KLJUČNA DOKUMENTACIJSKA INFORMACIJA

|  |  |  |  |
| --- | --- | --- | --- |
| Редни број, **РБР**: | |  | |
| Идентификациони број, **ИБР**: | |  | |
| Тип документације, **ТД**: | | Монографска публикација | |
| Тип записа, **ТЗ**: | | Текстуални штампани документ | |
| Врста рада, **ВР**: | | Дипломски рад | |
| Аутор, **АУ**: | | Рибар Милош | |
| Ментор, **МН**: | | др. Видаковић Милан, редовни професор | |
| Наслов рада, **НР**: | | Вишеагентско окружење у кластеру рачунара са балансирањем оптерећења сервера | |
| Језик публикације, **ЈП**: | | Српски/латиница | |
| Језик извода, **ЈИ**: | | Српски/енглески | |
| Земља публиковања, **ЗП**: | | Република Србија | |
| Уже географско подручје, **УГП**: | | АП Војводина | |
| Година, **ГО**: | | 2018. | |
| Издавач, **ИЗ**: | | Ауторски репринт | |
| Место и адреса, **МА**: | | Факултет техничких наука (ФТН), Д. Обрадовића 6, 21000 Нови Сад | |
| Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога) | |  | |
| Научна област, **НО**: | | Електротехника и рачунарство | |
| Научна дисциплина, **НД**: | | Примењене рачунарске науке и информатика | |
| Предметна одредница/Кqучне речи, **ПО**: | | Вишеагентско окружење, кластер, балансирање оптерећења | |
| **УДК** | |  | |
| Чува се, **ЧУ**: | | Библиотека ФТН, Д. Обрадовића 6, 21000 Нови Сад | |
| Важна напомена, **ВН**: | |  | |
| Извод, **ИЗ**: | | У овом раду описан је рад и примена вишеагентских окружења у кластеру са балансирањем оптерећења, користећи ***MapReduce*** програмерски модел. Овај рад пружа информације о конфигурисању Wildfly апликационог сервера тако да се рачунари повежу у кластер, и како да се исконфигурише да би се понашао као балансер оптерећења. | |
| Датум прихватања теме, **ДП**: | |  | |
| Датум одбране, **ДО**: | |  | |
| Чланови комисије, **КО**: | Председник: |  |
|  | Члан: |  | Потпис ментора |
|  | Члан, ментор: | др. Видаковић Милан, редовни професор |  |

# KEY WORDS DOCUMENTATION

|  |  |  |  |
| --- | --- | --- | --- |
| Accession number, **ANO**: | |  | |
| Identification number, **INO**: | |  | |
| Document type, **DT**: | | Monographic publication | |
| Type of record, **TR**: | | Textual material, printed | |
| Contents code, **CC**: | | Bachelor Thesis | |
| Author, **AU**: | | Ribar Miloš | |
| Mentor, **MN**: | | dr. Vidaković Milan, professor, Ph. D | |
| Title, **TI**: | | Multi-Agent Clustered Framework With Load Balancing | |
| Language of text, **LT**: | | Serbian | |
| Language of abstract, **LA**: | | Serbian/English | |
| Country of publication, **CP**: | | Republic of Serbia | |
| Locality of publication, **LP**: | | AP Vojvodina | |
| Publication year, **PY**: | | 2018 | |
| Publisher, **PB**: | | Author’s reprint | |
| Publication place, **PP**: | | Faculty of Technical Sciences, D. Obradovića 6, 21000 Novi Sad | |
| Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes) | |  | |
| Scientific field, **SF**: | | Electrical and computer engineering | |
| Scientific discipline, **SD**: | | Applied computer sciences and informatics | |
| Subject/Key words, **S**/**KW**: | | Multi-Agent Framework, Cluster, Load Balancing | |
| **UC** | |  | |
| Holding data, **HD**: | | Library of Faculty of Technical Sciences, D. Obradovića 6, 21000 Novi Sad | |
| Note, **N**: | |  | |
| Abstract, **AB**: | | This thesis describes how one multi-agent framework works in a cluster with a load balancer, using ***MapReduce*** programming model. This thesis provides information on how to configure Wildfly application server to create a computer cluster, and how to configure it to act as a load balancer. | |
| Accepted by the Scientific Board on, **ASB**: | |  | |
| Defended on, **DE**: | |  | |
| Defended Board, **DB**: | President: |  |
|  | Member: |  | Menthor's sign |
|  | Member, Mentor: | dr. Vidaković Milan, professor |  |

Biografija

Милош Рибар, рођен 11.09.1995, - Београд, општина Звездара, Србија. Завршио основну школу „Добросав Радосављевић Народ” и „Митровачку гимназију”.

Контакт електронска пошта:

[milosribar@yahoo.com](mailto:milosribar@yahoo.com)

Контакт мобилни телефон:

061/158-68-02