Univerzitet u Novom Sadu
Fakultet tehničkih nauka
Katedra za računarstvo i automatiku

Milan Vidaković

PROŠIRIVO AGENTSKO OKRUŽENJE BAZIRANO NA JAVA TEHNOLOGIJI

— doktorska disertacija —

Kandidat:
mr Milan Vidaković

Mentor
prof. dr Zora Konjović

Novi Sad, 2003. godine

SADRŽAJ

Predgovor

Agentska paradigma predstavlja najprirodniji i najdosledniji postojeći pristup implementaciji distribuiranih sistema. Uz pomoć agenata moguće je u potpunosti realizovati koncept distribuiranih softverskih komponenti, koje će, osim rešenja zadatka na distribuiranom nivou, pružiti i određenu količinu autonomnosti i inteligencije da bi se zadati cilj ostvario.

Agenti, dakle, predstavljaju softverske komponente koje su u stanju da obrađuju velike količine informacija, a da pri tom pruže izvesnu količinu inteligencije, autonomnosti i sposobnosti komunikacije.

Da bi agenti bili u stanju da reše postavljene zadatke, osim inteligencije, moraju biti u stanju da komuniciraju sa drugim agentima, da koriste postojeće resurse i druge sisteme, kao i da budu u stanju da svoj posao nastave i na drugom računaru.

Sve navedene osobine agenata implementiraju se u agentskim okruženjima. Agentska okruženja predstavljaju programska okruženja koja upravljaju životnim ciklusom agenta i obezbeđuju mu sve navedene mehanizme za realizaciju zadatka (komunikacija, mobilnost, pristup resursima i drugim podsistemima). Bitan element arhitekture agentskog okruženja predstavlja sigurnosni podsistem, koji omogućuje zaštitu podataka u procesu komunikacije, zaštitu agenata od neautorizovanog korišćenja, zaštitu agentskog okruženja od malicioznih agenata i obezbeđenje integriteta agenata u procesu prenosa sa jednog računara na drugi.

Za realizaciju agentskih okruženja trenutno se ističu sledeće tehnologije distribuiranih komponenti: CORBA (*Common Object Request Broker Architecture*) i J2EE (*Java 2 platform – Enterprise Edition*). Obe tehnologije podržavaju model distribuiranih komponenti i poseduju napredne mehanizme za komunikaciju i pretraživanje i smeštaj agenata.

Postoje mnogobrojne implementacije agentskih okruženja, kako u akademskim institucijama, tako i u industriji. Osim toga, razvoj agentske tehnologije je na međunarodnom nivou usaglašen FIPA specifikacijom. FIPA (*Foundation for Intelligent Physical Agents*) je neprofitna organizacija koja definiše standarde na polju saradnje među heterogenim agentskim okruženjima.

Tema doktorske disertacije predstavlja jedan predlog načina upotrebe Java tehnologije za realizaciju agentskog okruženja koje podržava FIPA specifikaciju. Za implementaciju distribuiranih komponenti upotrebljena

je J2EE tehnologija. J2EE tehnologija je odabrana zato što ona obuhvata širok spektar tehnologija distribuiranih informacionih sistema, a omogućuje skalabilnost, pouzdanost i ima veliku podršku u aplikacionim serverima.

Cilj disertacije je proširivi model agentskog okruženja baziran na FIPA specifikaciji. Ovaj model omogućuje upotrebu zamenljivih softverskih komponenti (*plug-in*), kojima su povereni pojedinačni poslovi u agentskom okruženju. Na ovaj način moguće je prilagoditi konfiguraciju sistema potrebama vezanim za rešavanje konkretnih problema i jedinstveno implementirati specifične algoritme za konkretan problem.

Tekst doktorske disertacije sastoji se iz sedam poglavlja.

Prvo poglavlje sadrži sledeća uvodna razmatranja: prikaz definicija osnovnih pojmova i problema u oblasti agentskih sistema, opise reprezentativnih implementacionih tehnologija (CORBA i J2EE) i detaljan opis i analizu FIPA specifikacije.

Drugo poglavlje se bavi detaljno agentskim okruženjima. Dat je prikaz radova koji se bave agentskim okruženjima, a zatim detaljan prikaz i analiza šest odabranih reprezentativnih agentskih okruženja. Svako okruženje je analizirano sa stanovišta najbitnijih faktora agentskog okruženja: razmena poruka, mobilnost agenata, sigurnost, servisi i FIPA kompatibilnost.

Treće poglavlje razmatra mogućnosti primene J2EE tehnologije za implementaciju agentskog okruženja. U njemu je dat prikaz i analiza tehnika iz J2EE tehnologije koje se mogu upotrebiti u agentskoj tehnologiji. Koncepti koji se žele ostvariti su asinhrona razmena poruka, mobilnost agenata, direktorijumi, servisi i sigurnosni mehanizmi, a tehnologije potrebne za realizaciju su JMS (*Java Messaging Service*) sistem poruka, serijalizacija, JNDI tehnologija i J2EE sigurnosni mehanizmi. Analizirani su odabrani J2EE aplikativni serveri (Orion i JBoss).

Četvrto poglavlje obuhvata formalnu specifikaciju proširivog agentskog okruženja, korišćenjem objedinjenog jezika modeliranja. Svi potrebni koncepti za realizaciju agentskog okruženja su modelirani u skladu sa FIPA specifikacijom. Identifikovani su elementi FIPA specifikacije koji se modeluju kao *plug-in* komponente. Najbitnije su: direktorijum agenata, direktorijum servisa, podsistem za razmenu poruka i bezbednosni podsistem.

U petom poglavlju dat je opis implementacije predloženog modela korišćenjem J2EE tehnologije. Agenti i servisi su modelirani kao EJB komponente, komunikacija između agenata se ostvaruje upotrebom JMS podsistema, direktorijumi su implementirani upotrebom JNDI podsistema, a mobilnost agenata i bezbednosni mehanizmi su modelovani postojećim rešenjima unutar J2EE tehnologije (serijalizacija i Java Cryptography Extension).

Sadržaji četvrtog i petog poglavlja predstavljaju centralni deo disertacije.

Šesto poglavlje je posvećeno verifikaciji predloženog agentskog okruženja u bibliotečkom informacionom sistemu BISIS. Na početku je predstavljena funkcionalna struktura BISIS-a i analiza funkcija sa stanovišta pogodnosti i opravdanosti implementacije primenom agentske tehnologije. Na osnovu ove analize, odabran je i implementiran skup funkcija BISIS-a korišćenjem predloženog modela agentskog okruženja. Implementacijom su obuhvaćeni svi koncepti predloženog modela, a sama implementacija je izvršena u programskom jeziku Java.

Sedmo poglavlje sadrži zaključna razmatranja, analizu rezultata i doprinosa doktorske disertacije, kao i analizu pravaca daljih istraživanja.

Zahvaljujem se svim članovima Komisije na korisnim sugestijama koje su doprinele da tekst bude jasniji i pregledniji.

Posebnu zahvalnost dugujem mentoru dr Zori Konjović i dr Dušanu Surli na nesebičnoj podršci u toku izrade disertacije.

Zahvaljujem se Jovani i porodici na razumevanju i podršci.

Novi Sad, septembar 2003.

MilanVidaković

Poglavlje 1

Uvodna razmatranja

1.1 Istorijat

Agentska paradigma predstavlja nov pogled na razvoj softvera. Prema nekim autorima [Schelde93, Lubar93, Ford95], koreni agentske paradigme sežu do prve polovine dvadesetog veka, do prvog pojma robota, kao uređaja koji je u stanju da obavlja poslove za ljude, da bi se razvojem računarske tehnike sredinom prošlog stoleća došlo do prvih hardverskih uređaja koji poseduju osobine agenata: servo mehanizmi, kontroleri i automatski piloti [Norman97]. U to vreme se nije koristio pojam *agent*, niti su postojale definicije agenata.

Od sredine šezdesetih godina prošlog veka, težište je prebačeno sa hardvera na softver, što je dovelo do pojave novih disciplina, kao što su veštačka inteligencija [Russel95], objektno programiranje [Agha86, Agha93] i HCI (*Human Computer Interface*) [Maes94]. Sve ove discipline, kao i mnoge druge, razvijane su pojedinačno, ali je tek početkom devedesetih godina prošlog veka uočeno da se ove discipline mogu posmatrati i kao celina, tj. da predstavljaju osnovne elemente agentske tehnologije [Jennings98].

Iako u oblasti veštačke inteligencije agentska tehnologija zauzima istaknuto mesto, sve do početka devedesetih godina prošlog veka nije bilo značajnih napora u proučavanju agenata [Jennings98]. Objašnjenje leži u činjenici da su istraživanja u veštačkoj inteligenciji imala tendenciju izučavanja različitih oblasti (učenje, razumevanje, rešavanje problema, itd.) odvojeno.

Začeci agentske tehnologije u oblasti veštačke inteligencije leže u oblasti planiranja (*AI planning*) [Allen90]. Ova oblast se bavi određivanjem akcija koje će biti izvršene u okruženju. Prema ovoj tehnologiji, agent je sistem koji izvršava akcije u nekom okruženju, koje je predstavljeno predikatskom logikom prvog reda. Akcije su reprezentovane PDA listama (*pre-condition*, *delete*, *add*) koje definišu uslove pod kojim će akcija biti izvedena. Algoritam planiranja uzima okruženje i reprezentaciju cilja, i proizvodi plan – program, koji specificira kako će agent postići cilj. Prvi ovakvi sistemi su bili GPS [Newel61] i STRIPS [Fikes71], kao i njegovi naslednici [Chapman87].

Osim tehnike planiranja, razvijena je i arhitektura inkorporiranja (*Subsumption architecture*) [Brooks86, Brooks91a, Brooks91b], gde je agent kolekcija ponašanja (*behaviors*) koje dovode do rešenja zadatka. Svako ponašanje je konačni automat stanja koji neprestano mapira ulazne informacije u izlazne akcije. Ovo je mapiranje postignuto pravilima "situacija-akcija", koje određuju koja će akcija biti izvedena na osnovu trenutnog stanja agenta.

Konačno, u oblasti veštačke inteligencije razvijena je i arhitektura agenata praktičnog rezonovanja (*Practical reasoning agents*) [Bratman88]. Ova arhitektura je inspirisana praktičnim rezonovanjem ljudi. U pitanju je praktično rasuđivanje bazirano na stavovima kao što su verovanja, želje i namere. Tipičan primer ove arhitekture je BDI (*belief-desire-intention*) sistem [Bratman88, Georgeff87]. Agenti u ovom sistemu su okarakterisani "mentalnim stanjem", definisanom sledećim karakteristikama: verovanja, želje i namere. Verovanja predstavljaju informacije koje agent ima o okruženju. Želje predstavljaju različite mogućnosti koje su na raspolaganju agentu. Namere predstavljaju opcije koje je agent odabrao. Najpoznatija implementacija ovog modela je PRS (*Procedural Reasoning System*) [Georgeff87].

Objektno programiranje igra veoma važnu ulogu u agentskoj tehnologiji. Iako se većina agentskih implementacija bazira na objektnoj paradigmi, postoji bitna razlika između objekata i agenata [Jennings98]:

- za razliku od objekata, agenti ne izvršavaju metode međusobno, već zahtevaju akcije (koje ne moraju biti izvršene na pozivajućoj strani);
- objekti ne poseduju karakteristike inicijative, reakcije i komunikacije.

HCI daje agentima nov domen upotrebe – umesto klasičnog izvršavanja programa kao reakcije na akcije korisničkog interfejsa, agentski orijentisane aplikacije imaju inicijativu. Ovakve aplikacije već postoje u obliku aktivnih *news* i *web* čitača [Maes94, Lieberman95].

1.2 Definicije agenata

Sve do početka devedesetih godina prošlog veka nije bilo konsenzusa oko definicije agenata. Pionir na polju veštačke inteligencije, Nikolas Negroponte je u knjizi "Being Digital" [Negroponte97] predstavio viziju agenata:

- "Agent odgovara na telefonski poziv, prepoznaje osobu koja je zvala, prosleđuje poziv do vas ili čak laže u vašu korist."
- "Ako imate nekoga ko vas dobro poznaje i ima pristup velikoj količini informacija, tada takva osoba može da dela u vaše ime veoma efikasno... Ovde nije u pitanju inteligencija. U pitanju je deljeno znanje i mogućnost njegovog korišćenja u vašem interesu."
- "Kao što komandant šalje izviđače... vi ćete slati agente da skupljaju informacije za vas. Agenti će slati druge agente. Proces će se umnožavati."

Osim navedenog opisa agenata, u radovima iz istog perioda, pojavile su se i prve definicije. Najčešće citirane definicije su:

"Autonomni agent je sistem koji je deo nekog okruženja i koji oseća to okruženje, dela unutar njega tokom vremena, a u cilju ispunjenja sopstvenog cilja." [Franklin96]

"Inteligentni agenti su softverski entiteti koji izvršavaju određeni skup operacija u korist korisnika ili drugog programa sa izvesnim stepenom nezavisnosti, a za to koriste izvesno znanje ili reprezentaciju korisnikovih želja ili ciljeva. Svakog agenta, dakle, karakterišu njegova autonomija i inteligencija. Autonomija je mogućnost samostalnog delovanja agenta i može se meriti njegovom interakcijom sa drugim entitetima. Inteligencija je stepen razumevanja i učenja." [Knapik98]

Agent je:
"... hardverski (ili češće softverski) sistem koji zadovoljava sledeće uslove:
*– **autonomija***
agenti delaju bez direktne intervencije ljudi ili drugih sistema i imaju izvesnu kontrolu nad akcijama i unutrašnjim stanjem;
*– **komunikacija***
agenti imaju interakciju sa drugim agentima (i, po mogućnosti, sa ljudima) uz pomoć nekakvog agentskog komunikacionog jezika;
*– **reakcija***
agenti osećaju svoje okruženje (koje može biti fizički svet, korisnici preko korisničkog interfejsa, drugi agenti, Internet ili kombinacija svega navedenog) i odgovaraju u konačnom vremenu na promene koje se tu odvijaju;
*– **inicijativa***

agenti ne reaguju samo na okolinu – oni su u stanju da demonstriraju ponašanje inicirano nekakvim ciljem." [Wooldridge95]

Potreba za inteligencijom agenata pruža mogućnost implementacije mnogobrojnih metoda veštačke inteligencije. Na primer, ekspertni sistemi mogu da pomognu agentima da interpretiraju podatke iz okruženja, predviđaju posledice ili trendove, planiraju ili zakazuju određene radnje i sl. Fazi logika može pomoći agentu da se snađe sa nepreciznim, dvosmislenim ili nekompletnim informacijama. Neuralne mreže se mogu iskoristiti da omoguće agentima da uče, da probaju da donesu odluku na osnovu nekompletnog znanja i sl. Inteligentni agenti se više ne gledaju kao posebna vrsta agenata – podrazumeva se da su svi agenti inteligentni.

Potreba za inteligencijom je direktno povezana sa potrebom za autonomnošću agenta. Autonomnost agenta podrazumeva da on može da funkcioniše bez direktne intervencije bilo operatera, bilo drugih sistema.

Prema [Bradshaw97], agentska tehnologija pruža sledeće mogućnosti:

- za rešavanje problema koji su izvan opsega klasičnih procesa automatizacije, ili zato što ne postoji tehnologija koja bi rešila problem, ili zato što bi klasične metode bile skupe;
- za rešavanje problema na način koji je značajno bolji (jevtiniji, prirodniji, efikasniji ili brži) od postojećih.

Na osnovu navedenih mogućnosti, do sada su razvijene mnoge konkretne implementacije agentskih sistema, koje se koriste, kako u akademskim uslovima, tako i u industriji. Ove implementacije se koriste u sledećim oblastima:

- agenti za pretragu i preuzimanje znanja [Bowman94],
- agenti za upravljanje računarskom mrežom [Head94],
- finansijski agenti [Maes99, Papazoglou01],
- agenti za nadgledanje pacijenata u bolnicama [Hayes-Roth89],
- agenti za industrijsku automatiku i kontrolu [Jennings96],
- agenti za kontrolu u energetici [Gustavsson99] i
- agenti za personalizaciju web servisa [Kuno02].

1.3 Značajne karakteristike agentske tehnologije

Agentska tehnologija predstavlja skup tehnologija koje omogućuju realizaciju agentske paradigme. Ove tehnologije su zasebno postojale i pre pojave agenata, ali je njihovo postojanje danas neophodno za realizaciju agenata. To su: mobilnost agenata, razmena poruka, servisi, direktorijumi i sigurnosni mehanizmi.

1.3.1 Mobilnost agenata

Agenti moraju da budu mobilni da bi u potpunosti realizovali zadatak. Mobilnost [Gilbert95, Bellavista00a, Fukuta01] predstavlja mogućnost da agent izvršava svoj zadatak na više računara, odnosno da može da bude u stanju da se "seli" sa računara na računar. Prelaz sa jednog računara na drugi podrazumeva prenos podataka i koda, odnosno podrazumeva da agent po prelasku na drugi računar bude u stanju da nastavi sa izvršenjem programa. Mobilnost agenata je proširenje koncepta distribuiranih komponenti i serijalizacije.

1.3.2 Komunikacija između agenata

Bitna osobina agenata je njihova sposobnost da komuniciraju. Komunikacija može biti između dva agenta, kao i između agenta i okruženja. Komunikacija se svodi na razmenu poruka. Kao de facto standard za međuagentsku komunikaciju pojavio se KQML jezik.

KQML [Labrou97, Neches91] (*Knowledge Query and Manipulation Language*) je jezik za komunikaciju između agenata. Zasniva se na razmeni tekstualnih poruka (*performatives*) između agenata. Sadržaj poruka može biti KQML poruka ili poruka na nekom drugom jeziku.

Po KQML standardu, u svakoj oblasti agentskog rada mora da postoji bar jedan agentski centar (*Facilitator*) koji se bavi obradom poruka i komunikacijom sa drugim agentima. Ostali agenti komuniciraju sa agentskim centrima da bi se njihove poruke prosleđivale svim drugim prijavljenim agentima, kao i da bi dobili adrese određenih prijavljenih agenata radi direktne komunikacije.

KQML poruke se sastoje od sledećih elemenata:

- blok za identifikaciju pošiljaoca i primaoca (**sender, receiver, from** i **to** polja),
- blok za identifikaciju poruka (**reply-with** i **in-reply-to** polja) i
- blok sa porukom (**language**, **ontology** i **content** polja).

Blok za identifikaciju služi za ispravnu identifikaciju učesnika u komunikaciji. Polja **sender** i **receiver** su obavezna. Blok za identifikaciju poruka služi za određivanje redosleda poruka. Svaka poruka mora da ima jedinstven identifikator, a svaki odgovor na tu poruku mora da sadrži taj identifikator, da bi agenti bili u stanju da pravilno povežu poruke. Blok sa porukom se sastoji iz polja sa nazivom jezika koji se koristi u telu poruke, polja sa nazivom ontologije i polja sa samom porukom.

Polje sa nazivom jezika označava u kom jeziku je napisan sadržaj poruke.

Ontologije predstavljaju specifikaciju konceptualizacije. Naime, formalna reprezentacija znanja se zasniva na konceptualizaciji – objektima, konceptima i ostalim entitetima koji postoje u zadatoj oblasti interesovanja, kao i relacijama između njih [Genesereth87]. Ontologije se sastoje iz definicija objekata i skupa formalnih aksioma koje ih opisuju. Ontologije praktično predstavljaju rečnik po kojem se upiti i izrazi vezani za razmenu znanja postavljaju.

KQML poruka sadrži samo naziv ontologije koja se koristi. Pretpostavlja se da je agent u stanju da interpretira poruku samo na osnovu naziva ontologije koja se koristi.

KQML poruke se dele u tri grupe:

- govorna grupa (**discourse performatives**),
- grupa za intervencije i manipulaciju (**intervention and mechanics performatives**) i
- grupa za obradu i mrežnu komunikaciju (**networking and facilitation performatives**).

Govorna grupa obuhvata poruke koje su najbliže govornom jeziku po semantici. To su poruke kojima agenti pitaju druge agente, odgovaraju na pitanja, oglašavaju svoje sposobnosti, manipulišu bazom znanja i dr.

Grupa za intervencije i manipulaciju obuhvata poruke sistemskog tipa a koje se odnose na intervencije u normalnom sledu poruka (prijavu grešaka, sinhronizaciju akcija i druge).

Grupa za obradu i mrežnu komunikaciju služi za razmenu specifičnih poruka vezanih za pronalaženje drugih agenata, prosleđivanje poruka, upućivanje javnih poruka i dr.

1.3.3 Servisi

Servisi su od suštinskog značaja za agente [Gilbert95, Lewis96]. Agenti koriste servise da bi pristupili određenim resursima ili realizovali kompleksne zadatke. Drugim rečima, servisi omogućavaju agentima da implementiraju samo osnovne algoritme za rešavanje zadatka, a ne i sve potrebne rutine i implementacije protokola potrebne za njegovo rešavanje.

1.3.4 Direktorijumi agenata i servisa

Direktorijumi su takođe bitan deo agentske tehnologije [Lewis96]. Direktorijumi su sistemi koji omogućavaju skladištenje i pronalaženje komponenti na osnovu njihovih opisa. U agentskoj tehnologiji se koriste dva tipa direktorijuma: direktorijum agenata i direktorijum servisa. Direktorijum agenata obezbeđuje registraciju i pretragu agenata, dok direktorijum servisa pruža iste usluge, ali za servise.

1.3.5 Sigurnosni mehanizmi

Razmena poruka i mobilnost agenata pre svega impliciraju upotrebu sigurnosnih mehanizama u agentskoj tehnologiji [He98, Yuh-Jong01, Kim02, Lupu96]. Sigurnost predstavlja veoma bitan segment ove tehnologije i mora mu se posvetiti pažnja. U trećem, četvrtom i petom poglavlju će biti analizirani sigurnosni mehanizmi neophodni za realizaciju agentskih okruženja.

1.4 Definicija agentskog okruženja (Agent Framework)

Prema [d'Inverno97] agentsko okruženje čini kolekcija entiteta, objekata, agenata i autonomnih agenata. Entiteti grupišu atribute bez funkcionalnog sloja. Objekti predstavljaju entitete sa sposobnostima uticanja na okolinu. Agenti su proširenje objekata sa skupom ciljeva koje on pokušava da postigne. Autonomni agenti su agenti koji generišu svoje ciljeve na osnovu motivacija. Osim spomenutih elemenata, agentsko okruženje karakteriše mogućnost ostvarivanja međuagentskih veza.

Agentska okruženja u [Maamar00] predstavljaju skup usluga koje zahtevaju korisnici ili druga okruženja. Nosioci usluga su agenti.

Iz svega navedenog vidi se da su agenti programske celine koje nisu sposobne da se izvršavaju samostalno. Agentima je potrebno programsko okruženje unutar koga će biti kreirani i unutar koga će izvršavati zadatke [Kautz94, Brugali00, Blixt00, d'Inverno97]. Osim programskog okruženja koje upravlja životnim ciklusom agenta, za efikasno rešavanje

zadataka potrebni su još servisi koje će agenti koristiti bilo za pristup resursima, bilo za realizaciju kompleksnih algoritama. Skup svih navedenih pojmova čini agentsko okruženje. Agentsko okruženje takođe omogućuje još i razmenu poruka, mobilnost agenata i potrebnu zaštitu, kako od neaturoizovanog pristupa agentima i servisima, tako i zaštitu od malicioznih agenata.

Razmena poruka se odvija preko agentskih okruženja. To znači da agenti ne komuniciraju direktno, već se poruke upućuju agentskom okruženju, a ovo poseduje mehanizme za pronalaženje odredišnog agentskog okruženja, u kom se nalazi odredišni agent. Mobilnost agenata podrazumeva da će agentsko okruženje prebaciti agenta u odredišno okruženje, koje će se pobrinuti da prebačeni agent nastavi svoj rad tamo. O svim pojedinostima vezanim za ovaj postupak brinuće se agentsko okruženje. Agentska okruženja obezbeđuju sve navedene usluge upotrebom direktorijuma. Direktorijumi omogućavaju lociranje željenog agenta ili servisa na osnovu njegovog opisa. Bezbednosni podsistem agentskog okruženja pruža usluge zaštite svim ostalim podsistemima.

Agentsko okruženje, dakle, predstavlja programsko okruženje koje upravlja životnim ciklusom agenata i obezbeđuje mu sve potrebne mehanizme za realizaciju zadatka.

1.5 Implementacione tehnike za inteligentne agente

Inteligentni agenti se mogu implementirati raznim tehnikama, od kojih se kao jedna od najkorisnijih pokazala objektno-orijentisana tehnika. Upravo zbog njenih osobina kao što su enkapsulacija, nasleđivanje, polimorfizam, dinamičko vezivanje i perzistencija, moguće je jednostavno i efikasno implementirati agente. Činjenica da se u OO tehnici podaci vezuju uz kod, čini izbor ove tehnike još smislenijom, jer bolje modelira stvarni svet.

Kao jedan od najkompletnijih predstavnika objektno-orijentisanih jezika, pojavio se programski jezik Java [Java]. On realizuje sve standardne koncepte OO tehnike. Za inteligentne agente je posebno važan Javin koncept serijalizacije, koji omogućuje da se objekat u memoriji prevede u binarni niz koji se može snimiti u datoteku ili prebaciti računarskom mrežom na drugi računar, a zatim tamo rekonstruisati i startovati ponovo. Ova osobina je veoma važna za mobilnost agenata, jer omogućuje agentima da se "presele" sa jednog računara na drugi.

Osim serijalizacije, za mobilnost agenata potreban je i koncept izvršavanja koda na udaljenim računarima. Ovo je u programskom jeziku

Java omogućeno upotrebom RMI (*Remote Method Invocation*) sistema [Wong99]. Ovaj sistem omogućuje Java programima da izvršavaju metode klasa na drugim računarima.

RMI i serijalizacija ne pružaju dovoljan broj servisa za naprednije zadatke, kao što su imenovanje i pretraga objekata, podrška za sigurnost podataka i transakcije.

Postoje mnogobrojne tehnologije koje obezbeđuju navedene servise, ali su se trenutno istakle dve: CORBA i J2EE.

1.5.1 CORBA

CORBA (*Common Object Request Broker Architecture*) [CORBA, DiPippo99, Benech97] predstavlja standard koji je kreirao konzorcijum *Object Management Group* (OMG). Ovaj standard definiše okvir u kojem se prave objekti koji se izvršavaju na serverskoj strani, a definiše i same servere. Standard se zasniva na *Internet Inter-ORB Protocol (*IIOP*)* protokolu. Sistem čine CORBA aplikacioni serveri, CORBA komponente (koje se izvršavaju na serverima), IIOP protokol za komunikaciju između servera i klijenata, i CORBA klijenti (koji izazivaju izvršenje CORBA komponenti na serverima). Ovaj standard predviđa izvršavanje komponenti napisanih u svim programskim jezicima koji ga podržavaju. Osim navedenih koncepata, CORBA podržava transakcije i svoj sistem imenovanja i pretraživanja komponenti (*Object Naming Service* - *COS Naming*). Za potrebe implementacije FIPA specifikacije svi potrebni koncepti se mogu realizovati postojećim elementima CORBA arhitekture. Agenti i servisi se mogu implementirati kao CORBA komponente. Direktorijumi se mogu implementirati upotrebom *COS Naming* sistema, a za potrebe komunikacije može se iskoristiti IIOP protokol.

1.5.2 J2EE

J2EE (*Java 2 platform – Enterprise Edition*) [J2EE] predstavlja kolekciju tehnologija koje omogućavaju rad sa sistemima velikog obima. Najbitniji element ove tehnologije je EJB (*Enterprise JavaBeans*) [EJB, Roman99]. EJB predstavlja kolekciju klijenata, aplikacionih servera i objekata na serverskoj strani, koje klijenti dobijaju na korišćenje od strane servera. Mehanizam rada podrazumeva da klijent zatraži od aplikacionog servera komponentu i radi sa njom koliko treba, a zatim inicira njeno oslobađanje. J2EE tehnologija uključuje i podršku za baze podataka (*Java Database Connectivity* - JDBC), sistem imenovanja i pretrage

objekata (*Java Naming and Directory Interface* - JNDI), transakcije (*Java Transaction API* - JTA), asinhronu komunikaciju između objekata (*Java Messaging Service* - JMS) i podršku za elektronsku poštu (*Java Mail*). Navedeni elementi J2EE tehnologije se u potpunosti mogu iskoristiti za implementaciju agentske tehnologije po FIPA specifikaciji. Agenti i servisi se mogu implementirati kao EJB komponente, direktorijumi se mogu implementirati upotrebom JNDI sistema, a komunikacija upotrebom JMS sistema.

1.6 FIPA specifikacija

FIPA (*Foundation for Intelligent Physical Agents*) [FIPA, Lyell02] je neprofitna organizacija koja definiše standarde na polju saradnje među heterogenim agentskim okruženjima. Ova organizacija je obezbedila širok spektar specifikacija koje definišu mnoge aspekte agentske tehnologije, uključujući tu i:

1. distribuirane računarske platforme i programske jezike,
2. sisteme za razmenu poruka,
3. sigurnosne sisteme i
4. direktorijume.

FIPA je predložila apstraktnu arhitekturu agentskog okruženja [FIPA00001], koja obuhvata heterogene sisteme, i ne bavi se specificiranjem segmenata na nivou postojećih tehnika, već nudi apstraktnu specifikaciju, koja ne isključuje postojeće tehnike. Ova arhitektura nudi:

1. model servisa i model direktorijuma servisa koji je dostupan agentima i drugim servisima [FIPA00023]
2. sistem razmene poruka [FIPA00067],
3. opis sadržaja poruke [FIPA00007] i
4. podršku različitim agentskim jezicima [FIPA00061].

Apstraktna arhitektura definiše sledeće koncepte:

1. transportni sistem za razmenu poruka (*Transport Message*),
2. direktorijum agenata (*Agent Directory*),
3. direktorijum servisa (*Service Directory*),

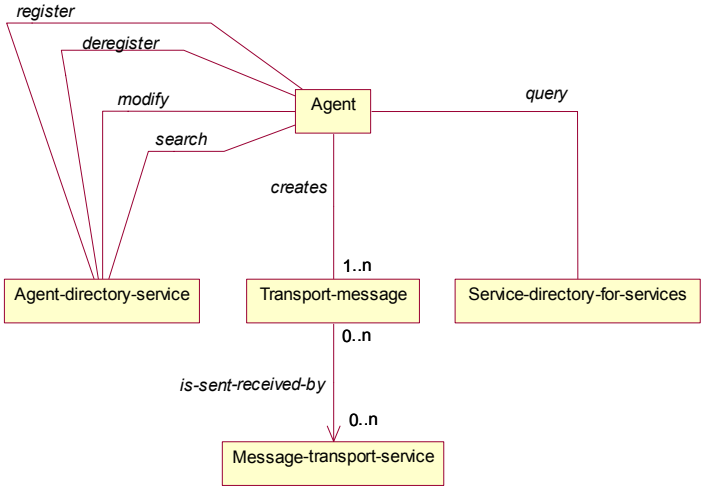
4. jezik za međuagentsku komunikaciju (ACL – *Agent Communication Language*).

Ova arhitektura se zatim može realizovati nekom od postojećih softverskih tehnologija.

1.6.1 Agenti i servisi

Agenti i servisi su osnovni element apstraktne arhitekture. Agenti obavljaju zadate poslove, pri tom komunicirajući jedni sa drugima upotrebom agentskog jezika za komunikaciju (ACL). Servisi obezbeđuju podršku za agente. Oni obezbeđuju sve potrebne funkcionalnosti za rad agenata, uključujući direktorijumske servise za agente i servise za komunikaciju. Sami servisi mogu biti implementirani kao agenti, ali to otežava pronalaženje i komunikaciju sa takvim servisima i nije preporučeno od strane FIPA.

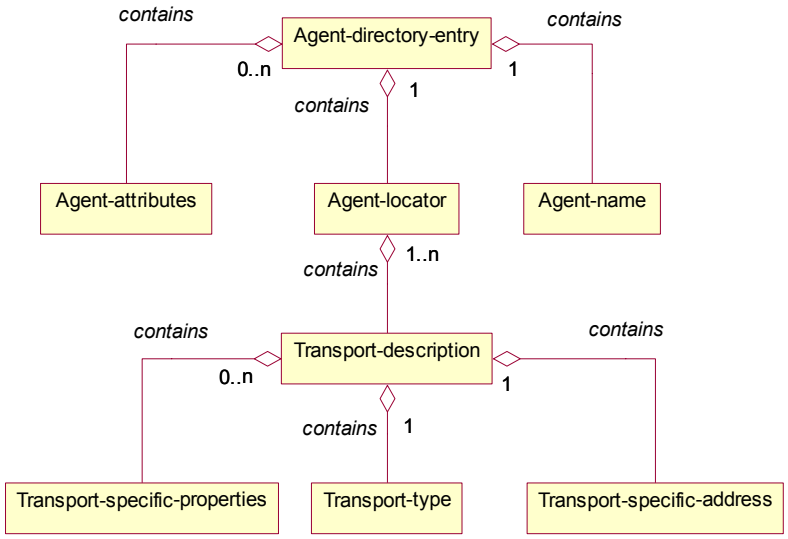
Prilikom startovanja agenta, mora biti obezbeđen pristup korenskom servisu (*service-root*). Ovaj servis obezbeđuje osnovni skup servisa koji su potrebni za funkcionisanje agenta, kao što su servis za transport poruka (*Message-transport-service*), direktorijum agenata (*Agent-directory-service*) i direktorijum servisa (*Service-directory-for-services*). Slika 1.1 prikazuje dijagram klasa potrebnih servisa za funkcionisanje agenata.



Slika 1.1: Dijagram klasa osnovnih elemenata potrebnih za funkcionisanje agenata.

1.6.2 Direktorijum agenata

Osnovna uloga direktorijuma agenata je da obezbedi lokaciju gde će se agenti registrovati. Agenti se registruju u ovom servisu ostavljanjem **direktorijumskih stavki za agente** (*Agent-directory-entry*) koje se sastoje najmanje iz para (ime-agenta, lokator-agenta). Ime agenta je jedinstveno ime za pretraživanje, a lokator agenta (*Agent-locator*) je složena struktura (*Transport-description*) koja definiše tip transporta (*Transport-type*), adresu (*Transport-specific-address*) i nula ili više karakteristika (*Transport-specific-properties*) koje se koriste za komunikaciju sa agentom. Osim ovoga, poželjno je da direktorijumska stavka sadrži i atribute agenata (*Agent-attributes*), kao što su spisak servisa koje agent nudi, cena upotrebe, ograničenja i sl. Slika 1.2 prikazuje dijagram klasa direktorijumske stavke za agenta.



Slika 1.2: Dijagram klasa za direktorijumsku stavku za agenta

Direktorijum agenata nudi sledeće usluge:

1. registracija agenata, što dovodi do ubacivanja stavke u repozitorijum,
2. modifikaciju stavki,
3. brisanje stavki i
4. pretraživanje.

Agent se registruje tako što direktorijumu agenata prosledi direktorijumsku stavku sa svojim opisom. Modifikaciju i brisanje stavki obavlja agent koji se prijavio (registrovao odgovarajućom stavkom). Pretraživanje agenata se svodi na to da agent kome treba usluga drugog agenta, pretražuje direktorijum agenata i kao rezultat pretrage dobija nula ili više direktorijumskih stavki. Pošto direktorijumska stavka sadrži lokator agenta, agent je u stanju da stupi u kontakt sa drugim agentom.

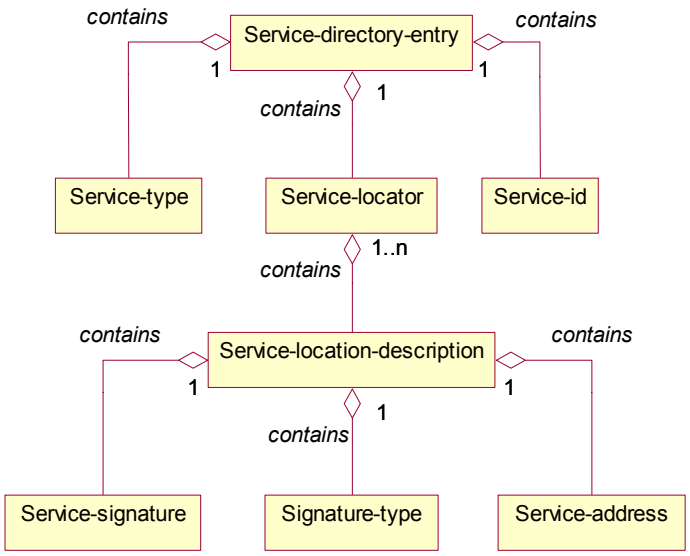
1.6.3 Direktorijum servisa

Osnovna uloga direktorijuma servisa je da obezbedi sredstva kojima agenti i drugi servisi mogu da pronađu odgovarajuće servise. Direktorijum servisa se sastoji iz **direktorijumskih stavki za servise** (*Service-directory-entry*), koje se sastoji iz sledećih elemenata:

1. id servisa (globalno jedinstveno ime servisa – *Service-id*),
2. tip servisa (kategorizovan tip servisa – *Service-type*) i
3. lokator servisa – *Service-locator*. To je složena struktura (*Service-location-description*) koja se sastoji iz signature tipa – *Signature-type*, signature servisa – *Service-signature* i adrese servisa – *Service-address*.

Osim navedenih, moguće je smestiti i atribute servisa koji sadrže dopunske informacije o servisu, kao što su cena servisa, ograničenja i sl. Slika 1.3 prikazuje dijagram klasa direktorijumske stavke za servise.

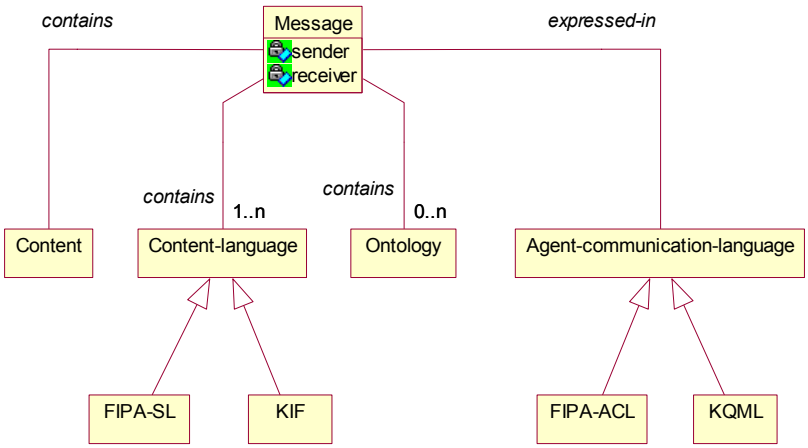
Servisi se registruju kod direktorijumskog sistema za servise. Pri tom im se dodeljuje odgovarajuća stavka (*Service-directory-entry*). Servis ovu stavku može obrisati ili modifikovati, a servisi ili agenti mogu da pretražuju stavke od interesa.



Slika 1.3: Dijagram klasa za direktorijumsku stavku za servise

1.6.4 Agentske poruke

U FIPA agentskim okruženjima, agenti komuniciraju razmenom poruka. Poruke se sastoje iz parova (ključ, vrednost) i napisane su u nekom od agentskih komunikacionih jezika (slika 1.4). Svaka poruka je napisana u nekom od agentskih jezika (*Agent-communication-language*, čije konkretne implementacije mogu biti FIPA-ACL ili KQML). Sastoji se iz naziva pošiljaoca (*sender*) i primalaca (*receiver*) i sadržaja (*Content*). Sadržaj može biti uslovljen ontologijom (*Ontology*) koja se posebno navodi, a opisan je jezikom za opis sadržaja (*Content-language*, čije konkretne implementacije mogu biti FIPA-SL [FIPA00008] ili KIF – *Knowledge Interchange Format* [Genesereth92]). Poruka u svom sadržaju može da sadrži druge poruke.

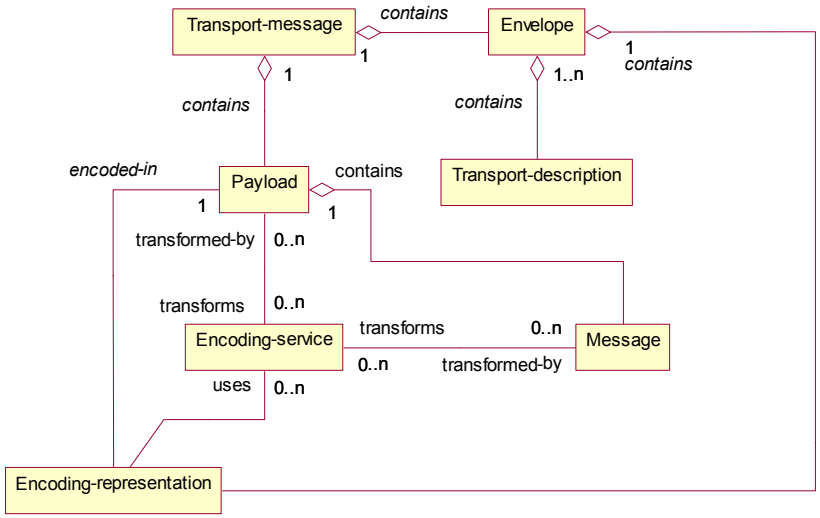


Slika 1.4: Dijagram klasa poruka

1.6.5 Transport poruka

Svaka poruka (*Message*) se transformiše u transportnu poruku (*Transport-message*) koja je zapravo druga reprezentacija poruke, prilagođena transportnom medijumu. Originalna poruka je sadržana u paketu (*Payload*), koji zajedno sa opisom čini transportnu poruku. Opis poruke (*Envelope*) [FIPA00073] definiše transportne atribute (*Transport-description*), kao što su adresa pošiljaoca i primaoca, transportni protokol i dr.

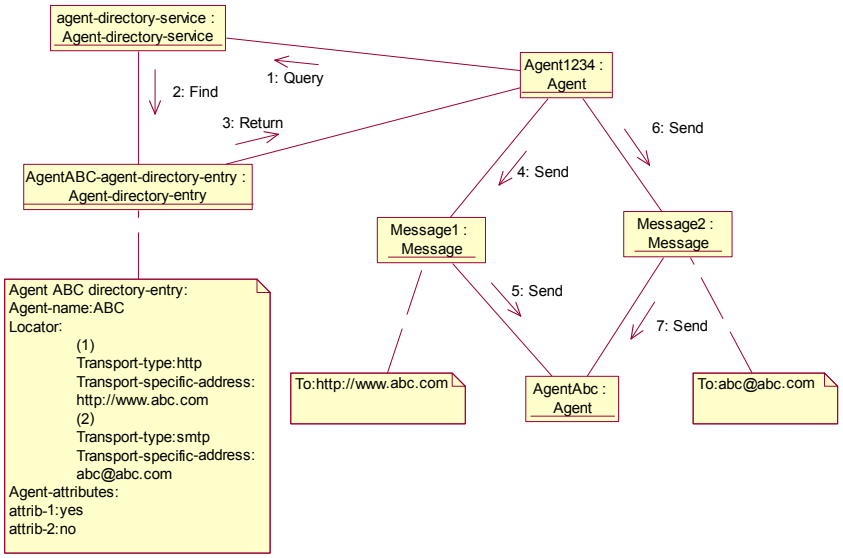
Proces transformacije poruke se svodi na transformaciju paketa u transportni oblik (*Encoding-representation*, predstavljen XML-om, serijalizovanim Java objektom i sl.) upotrebom servisa za transformacije (*Encoding-service*).



Slika 1.5: Dijagram klasa za transport poruka

1.6.6 Razmena poruka

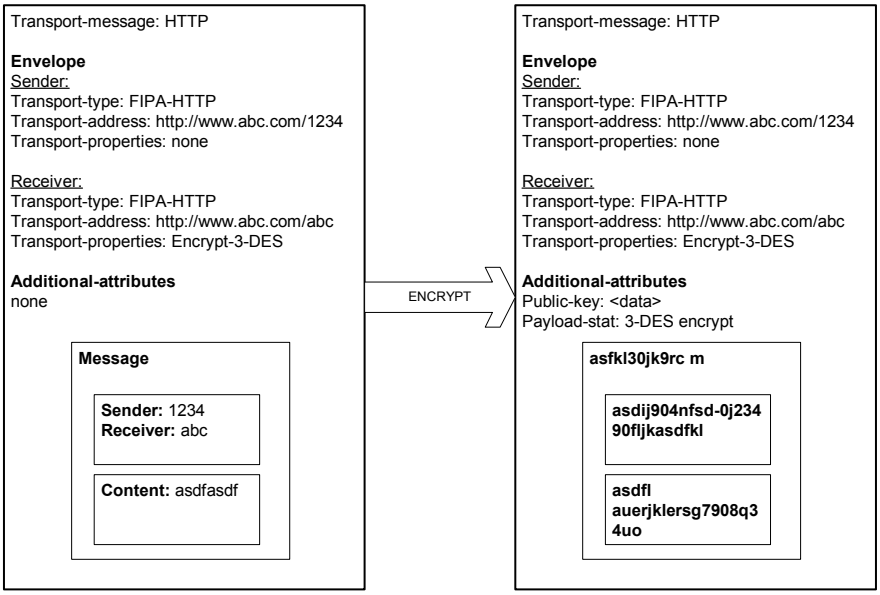
Za razmenu poruka neophodno je da polazni agent, na osnovu agentskog imena odredišnog agenta, dobije sve potrebne transportne parametre, kao što su tip transporta (*Transport-type*) i transportna adresa (*Transport-specific-address*). Za dobijanje ovih parametara, agent se obraća direktorijumu agenata. Ovi parametri se nalaze u lokatoru agenta (*Agent-locator*). Slika 1.6 prikazuje dijagram saradnje za komunikaciju između dva agenta upotrebom dva različita transportna protokola. Komunikacija počinje pretragom direktorijumskog servisa za agente (*1:Query*). Na osnovu imena odredišnog agenta, polazni agent iz direktorijuma agenata dobija direktorijumsku stavku agenta (*Agent-directory-entry*) u kojoj se nalaze svi potrebni parametri za komunikaciju (*Transport-type*, *Transport-specific-address*, *Transport-properties*). Polazni agent može da pošalje poruku po prvom tipu transporta (u ovom slučaju, HTTP protokolom), a zatim da promeni tip transporta (na SMTP protokol). Odredišni agent je u mogućnosti da komunicira upotrebom oba tipa transporta, što je i naglasio u svojoj direktorijumskoj stavci.



Slika 1.6: Dijagram saradnje za komunikaciju između dva agenta

1.6.7 Validacija poruka i kriptovanje

Sigurnosni mehanizmi u FIPA specifikaciji podrazumevaju validaciju poruke u smislu detekcije izmene sadržaja i kriptovanje poruke. Ako se primeni bilo koji od ovih mehanizama, parametri vezani za njegovu primenu se moraju smestiti u kovertu (*envelope*), što je prikazano na slici 1.7, za slučaj kriptovanja poruke.



Slika 1.7: Kriptovanje poruke

Slika 1.7 prikazuje proces kriptovanja poruke. Poruka je već transformisana u transportni oblik i nalazi se unutar paketa (*Payload*), a svi transportni parametri se nalaze u opisu poruke (*Envelope*). Ako je potrebno izvršiti kriptovanje poruke, tada se poruka kriptuje, a javni ključ i naziv algoritma za kriptovanje se smeštaju u opis. Na osnovu naziva algoritma iz opisa, prijemna strana je u stanju da odabere odgovarajući algoritam, a na osnovu javnog ključa da dekriptuje poruku.

Poglavlje 2

Agentska okruženja

Prvi radovi o agentskim okruženjima su se zasnivali na prikazu rešenja konkretnog problema [Palaniappan92, Chauhan98]. Ovo je dovelo do pojave namenski razvijanih sistema koji koriste agentsku tehnologiju za realizaciju zadataka [Nardi98, Wilson00, Kendall00, Bellavista00b].

Sa druge strane, kao prirodna posledica uopštavanja problema, pojavila su se agentska okruženja [JAF, Bigus, Aglets, JAT, Bellifemine99] koja nisu pravljena za realizaciju posebnog problema, već pružaju podršku proizvoljnim agentima. Ova okruženja predstavljaju sisteme koji upravljaju životnim tokom agenata, omogućuju međuagentsku komunikaciju i mobilnost. Sigurnosni aspekti su takođe uzeti u obzir.

Sa stanovišta razvoja agentskih okruženja značajan rad predstavlja [Shehory01], gde je dat pregled tehnika modelovanja agentskih sistema. Ove tehnike su bitne u identifikaciji osnovnih elemenata agentske tehnologije i njihovih međusobnih veza.

Značajan rad na polju agentskih okruženja predstavlja [Aridor98], koji definiše šeme (*design pattern*-e) za poznate probleme i situacije u razvoju agentskih okruženja. Postoje tri osnovne šeme: *Traveling*, *Task* i *Interaction*. Prva se bavi problemima prenosa agenata, druga vezom između agenata i zadataka koji se rešavaju, a treća se bavi komunikacijom.

Sa tehnološke tačke gledišta, agentska okruženja se baziraju ili na sopstvenim rešenjima ili na tehnologiji distribuiranih komponenti. Agentska okruženja kao što su JAF (*Java Agent Framework*) [JAF] i JAT (*Java Agent Template*) [JAT] realizuju sve elemente agentskog okruženja sopstvenim kodom. Sa druge strane, JADE (*Java Agent DEvelpment framework*) [Bellifemine99] koristi RMI i CORBA tehnologije za realizaciju većine elemenata agentskog okruženja.

U poslednje vreme, velik broj radova na temu sigurnosnih aspekata agentske tehnologije se pojavljuje [Vuong01, Binder02, Varadharajan00, Kim02, Yuh-Jong01, He98]. Sigurnosni aspekti koji se razmatraju su: obezbeđivanje integriteta poruka, zaštita koda tokom prenosa agenata, zaštita agentskog okruženja od agenata i zaštita agenata od agentskih okruženja.

Sa stanovišta analize nekog agentskog okruženja, postoji određen broj zahteva koje to okruženje mora da ispunjava. To je pre svega, mogućnost izvršenja agenata u kontrolisanom okruženju, uz primenu sigurnosnih mehanizama. Takođe, mobilnost agenata je bitan koncept, koji povlači implementaciju tehnika distribuiranih informacionih sistema. Osim izvršavanja koda, okruženja moraju da poseduju mogućnost za razmenu poruka, posebno KQML poruka. Pronalaženje odgovarajućih agenata i servisa (direktorijum agenata i servisa) takođe predstavlja bitan zahtev koji se postavlja ispred agentskog okruženja. Postojeća agentska okruženja ispunjavaju većinu ovih zahteva. U ovom poglavlju će biti analizirano šest reprezentativnih agentskih okruženja: *Java Agent Framework*, *Agent Building and Learning Environment*, *Voyager*, *Aglets*, *Java Agent Template* i *Java Agent DEvelopment framework*.

2.1 JAF – Java Agent Framework

JAF [JAF, Vincent01, Horling98] je agentski sistem razvijen na MIT-u. Prevashodno je razvijen za MASS [Horling00] sistem (*Multi Agent System Simulator*). MASS predstavlja okruženje za multiagentsku koordinaciju i komunikaciju. Ovo okruženje podrazumeva distribuiranu simulaciju agentskog okruženja koja je vođena događajima (*event-driven*). MASS poseduje mehanizam vremenskog prozivanja agenata, kao i komunikacije razmenom poruka između agenata. MASS sistem koristi TAEMS [Horling02] (*Task Analysis, Environment Modeling and Simulation*) okvir za modelovanje rešenja problema. Ovaj okvir se koristi za rešavanje problema kada su specifični rokovi zadati, kada su informacije potrebne za rešavanje nedostupne, kada se rezultati rada agenata moraju uklopiti u generalno rešenje zadatka, kao i kada se rezultati rada agenata paralelno koriste za više ciljeva.

JAF se zasniva na komponentnom agentskom modelu. Komponentni model se zasniva na softverskim celinama – komponentama, što u ovoj implementaciji predstavlja Java klase. Ove komponente su pisane tako da se njihovim kombinovanjem mogu rešavati složeni problemi. Ovo implicira poštovanje određene konvencije kreiranja i imenovanja metoda, koje omogućuje rešavanje problema prostim "uključivanjem" ovih softverskih komponenti u sistem. Sam sistem se sastoji iz komponenti prikazanih u tabeli 2.1.

Naziv	Opis
Control	Obezbeđuje inicijalizaciju i kontrolu u toku izvršavanja.
State	Koristi se kao skladište podataka. Koristi se i kod pronalaženja agenata.
Log	Obezbeđuje dnevnik aktivnosti.
LogViewer	Grafički interpretira dnevnike aktivnosti.
Execute	Upravlja aktivnostima agenata. Aktivnosti su pokrivene odgovarajućim događajima. Sve strane uključene u izvršenju mogu biti informisane o izvršenju implementiranjem odgovarajućih *Listener-a*.
Communicate	Omogućuje TCP/IP komunikaciju, uz podršku heterogenih sistema poruka.
PreprocessTaemsReader	Procesira tekstualne TAEM-e i na osnovu njih kreira objektnu strukturu.
Observe	Omogućava periodično ili događajima vođeno nadgledanje događaja.
ResourceModeller	Nadgleda izvršenje da bi izgradio odgovarajući model upotrebe resursa.

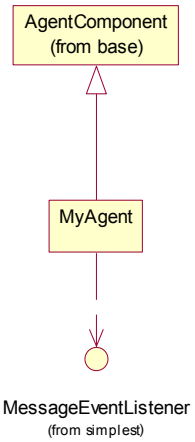
Tabela 2.1: Spisak komponenti JAF sistema

Sistem je otvoren za dodavanje novih softverskih komponenti [Horling98], pod uslovom da su one u stanju da se spoje na sistem događaja **`ActionEvent`** komponente *Execute*.

2.1.1 Arhitektura

JAF se sastoji iz softverskih komponenti, od kojih su neke prisutne u sistemu, a neke korisnik kreira. Komponente variraju od najjednostavnijih klasa koje rešavaju jedan jednostavan problem (*utility class*), preko onih koje reaguju na ponašanja drugih klasa, do klasa koje poseduju sopstvene ciljeve i načine realizacije. Na najnižem nivou apstrakcije svaka komponenta koja nasleđuje **`agent.base.AgentComponent`** je JAF komponenta. Ova klasa implementira osnovni skup funkcionalnosti koje je potreban za rad agenta u JAF sistemu

(inicijalizaciju agenta – metoda **`init()`**, početno podešavanje – metoda **`begin()`**, periodično izvršavanje – metoda **`pulse()`** i završne akcije – metoda **`end()`**). Slika 2.1 prikazuje jednostavnu JAF komponentu, a listing 2.1 prikazuje izvorni kod za tu komponentu.



Slika 2.1: Dijagram klasa jednostavne JAF komponente

```
import agent.base.AgentComponent;
import agent.base.AgentEvent;
import agent.base.ListenerVector;
import agent.simplest.*;
import agent.simplest.ActionEvent;
import utilities.Connection;
import utilities.Message;
import utilities.KQMLMessage;

public class MyAgent extends AgentComponent implements
MessageEventListener {
  protected Log log;
  protected Communicate communicate;

  public MyAgent() {
    super();
    addDependency("Log");
    addDependency("Communicate");
  }
```

Listing 2.1: Izvorni kod jednostavne JAF komponente

```
  public void init() {
    super.init();
    log = (Log)agent.simplest.State.findComponent("Log");
    communicate = (Communicate)
agent.simplest.State.findComponent("Communicate");
    communicate.addMessageEventListener(this);
    log.log("Zavrsetak init funkcije", Log.LOG_INFO, 0);
  }

  public void begin() {
    super.begin();
  }

  public void pulse() {
    super.pulse();
  }

  public void end() {
    super.end();
  }

  public void messageReceived(MessageEvent me) {
    Message m = (Message)me.getMessage();
    if (m instanceof KQMLMessage &&
me.getConnection().getType() != Connection.KQML) {
      KQMLMessage km = (KQMLMessage)m;
      String perf = km.getPerformative();
      if (perf.equalsIgnoreCase("tell")) {
        String word = m.contentWord();
        String data = m.contentData();
        if (!word.equals("pulse")) {
           log.log("Primljena poruka: " + data,
Log.LOG_INFO, 0);
        }
      }
    }
  }
  public void messageSent(MessageEvent me) {}
}
```

Listing 2.1: Izvorni kod jednostavne JAF komponente – nastavak

Listing 2.1 ilustruje kod jednostavnog agenta koji svu inicijalizaciju obavlja u `init()` metodi, a u stanju je da prima KQML poruke preko metode `messageReceived()`. Unutar `init()` metode, pronalaze se dve komponente – *Log* i *Communicate*. *Log* komponenta će služiti za vođenje dnevnika aktivnosti, a *Communicate* komponenta za prijavljivanje ovog agenta sistemu za manipulaciju porukama. Tek po prijavljivanju agenta

(metodom **`addMessageEventListener()`**), agent je u stanju da prima KQML poruke.

2.1.2 Inicijalizacija i tok izvršavanja

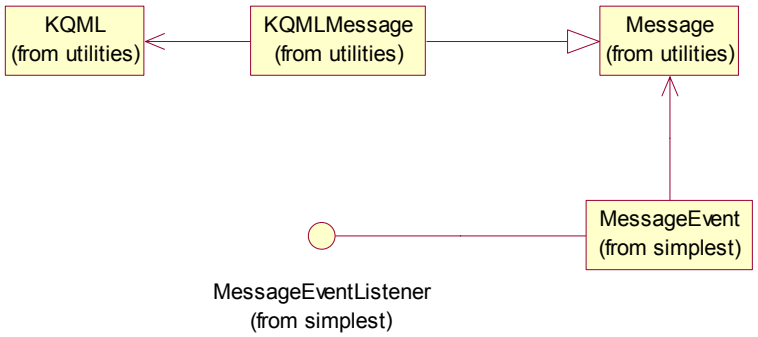
Control komponenta je zadužena za izvršenje zadataka ostalih komponenti. Svaka komponenta prolazi kroz sledeće faze životnog ciklusa:

- Kreiranje – komponenta se kreira i poziva se konstruktor.
- Inicijalizacija – poziva se metod **`init()`** u koji se smešta inicijalizacioni kod.
- Početak – poziva se metoda **`begin()`**. Ovim se završava inicijalizacija komponente.
- Impuls – poziva se metoda **`pulse()`**. Ova metoda se poziva periodično od strane *Control* komponente, pa se tu može smestiti kod koji je potrebno periodično izvršavati.
- Kraj – poziva se metoda **`end()`** kojom se završava životni ciklus komponente.

2.1.3 Komunikacija

Komunikacija između komponenti se zasniva na dva koncepta – konceptu razmene poruka i konceptu događaja, kojim komponente bivaju obaveštene o zbivanjima u okruženju (okruženje predstavlja MASS sistem). Poruke se šalju uz pomoć *Communicate* komponente, koja omogućuje slanje proizvoljnih poruka između komponenti. Poruke se šalju na zadati računar i zadati port. Ako komponenta želi da primi poruku, mora da implementira **`MessageEventListener`** interfejs. Ovaj interfejs za svaku pristiglu poruku poziva metodu **`messageReceived()`**. Parametar ove metode je objekat klase **`MessageEvent`**, koji reprezentuje pristizanje poruke.

KQML poruke su podržane klasom **`KQMLMessage`**, koja nasleđuje opštu klasu **`Message`**. Slika 2.2 ilustruje razmenu poruka.



Slika 2.2: Dijagram klasa razmene KQML poruka

Za potrebe parsiranje KQML poruka, postoji klasa KQML (paket **`utilities`**), koja sadrži metodu **`parseMessage().`** Ova metoda vrši parsiranje KQML poruke i iz nje izvlači sve potrebne elemente i smešta u asocijativnu listu. Klasa **`Message`** predstavlja osnovnu klasu za poruke i sadrži polja zajednička za sve tipove poruka (**`sourceaddr`**, **`destaddr`**, **`sourceport`**, **`destport`**, **`sendtime`**, **`receivetime`** i **`data`**). Klasa **`KQMLMessage`** nasleđuje klasu **`Message`** i specijalizuje je za razmenu KQML poruka. Osim nasleđenih atributa, sadrži i atribut **`kqmlmess`**, koji sadrži KQML poruku. Klasa **`MessageEvent`** predstavlja tip događaja vezan za prispeće ili uspešno slanje KQML poruke. Ova dva događaja se obrađuju upotrebom **`MessageEventListener`** interfejsa, koji sadrži dve odgovarajuće metode **`messageSent()`** i **`messageReceived()`**.

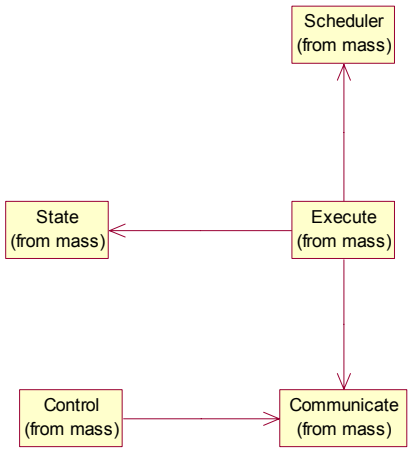
Po prispeću poruke na računar, komponenta *Communicate* poziva metodu **`messageReceived()`** svih prijavljenih komponenti. Ovo nije jedini tip događaja na koji komponenta može da reaguje. Na raspolaganju su sledeći događaji:

- **`ActionEvent`** – generički **`Action`** događaj,
- **`LogEvent`** – događaj unosa stavke u dnevnik (log),
- **`MessageEvent`** – događaj slanja, odnosno prijema poruke,
- **`PropertyEvent`** – događaj koji se javlja kada se promeni neki od sistemskih atributa (property),
- **`ScheduleEvent`** – događaj koji se javlja prilikom izvršavanja zakazanog zadatka i
- **`SensorEvent`** – generički **`Sensor`** događaj.

Za svaki od navedenih događaja postoji odgovarajući *Listener* koji se mora implementirati da bi se događaji mogli registrovati.

2.1.4 Izvršenje zadataka

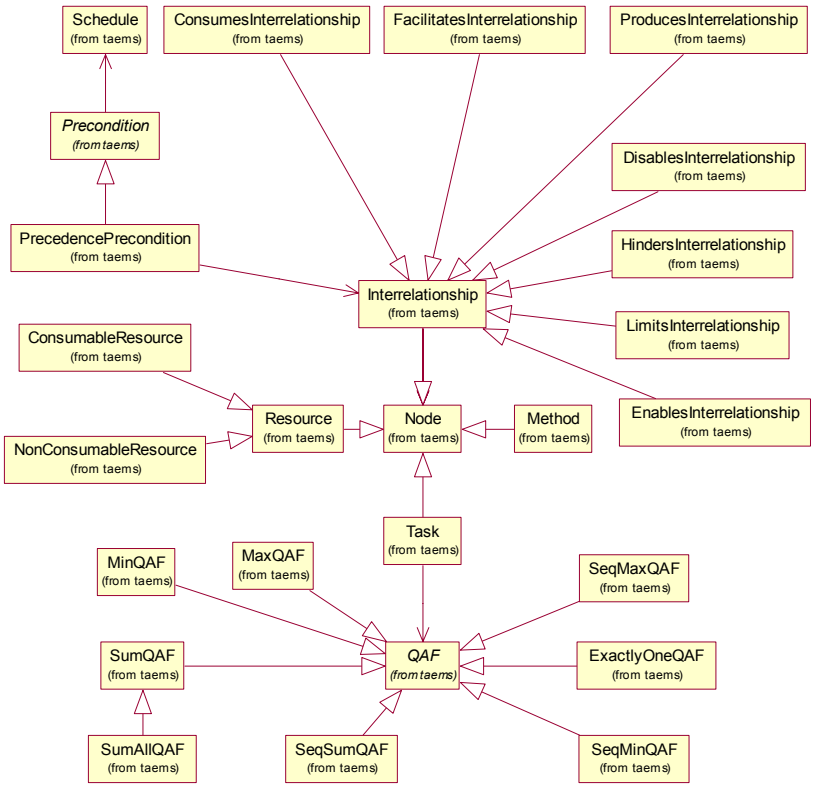
Izvršenje zadatka je povereno komponenti *Control*, koja u saradnji sa *Execute* i *Scheduler* komponentama upravlja izvršenjem zadataka (slika 2.3).



Slika 2.3: Dijagram klasa izvršenja zadataka

Klasa **`Execute`** sadrži metodu **`pulse()`**, koja je zadužena za izvršenje zadataka zakazanih **`Scheduler`** komponentom. Osim toga, ova klasa je zadužena i za izvršenje zadataka poverenih agentima. Svaka poruka koja se šalje prolazi kroz **`Control`** i **`Communicate`** klase, a ove je prosleđuju **`Execute`** klasi preko njenih metoda **`messageReceived()`** i **`messageSent()`**.

Izvršenje zadatka može biti programirano u samoj komponenti, ali za kompleksnije zadatke postoji paket **`taems`**, koji omogućuje realizaciju složenih zadataka upotrebom TAEMS okvira. Kao što je ranije rečeno, TAEMS okvir služi za modelovanje rešenja problema. To znači da se svaki zadatak predstavlja u obliku grafa čiji čvorovi (klasa **`Node`**) mogu biti: metod, zadatak, relacija između entiteta i resurs (slika 2.4). Osnovni čvor je grupa zadataka, što je, u stvari, zadatak koji nema nadzadatak. On se deli na podzadatke, koji mogu imati još podzadataka ili konkretne metode koje se moraju izvršiti.



Slika 2.4: Dijagram klasa elemenata TAEMS strukture

Metod (klasa **`Method`**) predstavlja osnovni element u TAEMS strukturi. On definiše šta agent može da učini. On sadrži različite rezultate koje metod može da generiše, kao i karakteristike svakog pojedinačnog rezultata (raspodelu kvaliteta, trajanja i cene). Kvalitet predstavlja apstraktnu karakteristiku izvršenja metode koju bi agent trebalo da maksimizuje. Cena predstavlja količinu sredstava koja su potrebna da se metoda realizuje. Cena može biti izražena u novčanim jedinicama, ali i u procesorskom vremenu ili nečemu drugom što bi trebalo da se minimizuje tokom izvršenja metode. Trajanje definiše vreme izvršenja metode.

Metode se organizuju u zadatke (**`Task`** klasa). Zadaci predstavljaju grupu poslova koja treba da se obavi. Kombinacija ovih podzadataka ili metoda se reprezentuje funkcijom kvaliteta akumulacije (QAF – *Quality*

Accumulation Function) [Wagner01], koja definiše kvalitet podzadataka ili metoda u odnosu na ukupan zadatak. Ona definiše međusobnu zavisnost podzadataka, redosled izvršavanja i način izračunavanja ukupnog kvaliteta. Postoji konačan broj funkcija:

- *Min* – ekvivalent AND operatora; ukupan kvalitet je jednak najmanjem kvalitetu, a neuspešno završen zadatak ima kvalitet 0,
- *Max* – ekvivalent OR operatora; ukupan kvalitet je jednak najvećem kvalitetu, bez obzira da li postoje neuspešno završeni zadaci (njihov kvalitet je 0),
- *Sum* – suma svih kvaliteta,
- *All* (*Sum All*) – suma svih kvaliteta, ali pod uslovom da su svi uspešno završeni; ako neki podzadatak nije uspešno završen, ukupan rezultat je nula,
- *Seq Min* (*Sequenced Min*) – kao i *Min*, samo što je neophodno da svi podzadaci budu uspešno završeni, kao i da se poštuje redosled,
- *Seq Max* (*Sequenced Max*) – kao i *Max*, samo što je neophodno da svi podzadaci budu uspešno završeni, kao i da se poštuje redosled,
- *Seq Sum* (*Sequenced Sum*) – kao i *Sum*, samo što je neophodno da svi podzadaci budu uspešno završeni, kao i da se poštuje redosled,
- *Exactly One* – ekvivalent XOR operatora; samo jedan podzadatak sme da bude uspešno završen) i
- *Sigmoid* (nije još podržano).

Resursi (**`Resource`** klasa) predstavljaju entitete čije stanje menjaju metode. Postoje dva tipa resursa: potrošni i nepotrošni. Potrošnim resursima opada stanje na nulu i moguće je samo pravljenje novih resursa (osim potrošnje). Nepotrošni resursi trenutno menjaju svoje stanje, ali po završetku dejstva, stanje im se vraća na zadato.

Relacije između entiteta definišu međusobni uticaj. Postoje sledeće vrste relacija:

- Uslovljavanje (**`EnablesInterrelationship`** klasa); jedan zadatak ili metod ne može da bude izvršen ako drugi nije uspešno završen (ovo se može simulirati i uz pomoć sekvencijalne QAF),

- Blokiranje (`DisablesInterrelationship` klasa); izvršenje jednog zadatka ili metode direktno blokira izvršenje druge,
- Olakšavanje (`FacilitatesInterrelationship` klasa); "soft" podešavanje, koje nije tipa uključenje/isključenje, već se definiše faktor koji utiče na kvalitet, trajanje i cenu izvršenja zadatka ili metode na koju se ova relacija odnosi.
- Sprečavanje (`HindersInterrelationship` klasa); "soft" podešavanje, koje je slično Olakšavanje relaciji, samo što se kvalitet smanjuje, a trajanje i cena se povećavaju.
- Proizvodnja (`ProducesInterrelationship` klasa); ova relacija uvek povezuje metod i resurs. Svako izvršenje metode će "proizvesti" povezani resurs.
- Potrošnja (`ConsumesInterrelationship` klasa); za razliku od prethodne relacije, ova relacija će "trošiti" povezani resurs prilikom izvršenja metode,
- Ograničenje (`LimitsInterrelationship` klasa); nije još podržano.

Klase `Precondition`, `PrecedencePrecoddition` i `Schedule` obezbeđuju izvršenje zadataka u vremenu i po zadatom redosledu.

2.1.5 Parametri i State komponenta

State komponenta omogućuje smeštanje i obradu proizvoljnih parametara. Pod parametrima se podrazumevaju, kako proizvoljni parovi (naziv, vrednost), tako i parovi (naziv_komponente, referenca_na_komponentu). Zapravo, *State* komponenta se može koristiti kao direktorijum servisa.

2.1.6 Karakteristike JAF agentskog okruženja

JAF predstavlja napredno okruženje za rad sa agentima. Dobre osobine su: rad sa TAEMS okvirom, implementacija sistema za razmenu poruka koji nije eksplicitno vezan za KQML, podrška za *real-time* sisteme i ugrađena podrška za rad sa dnevnikom aktivnosti. Manu predstavlja činjenica da ne postoji sistem razmene direktnih poruka od jednog agenta do drugog, već se sami agenti moraju dogovoriti kako da se njihove poruke razlikuju u mnoštvu poruka koje cirkulišu od jednog do drugog računara. Ovo je direktna posledica implementirane komunikacije između komponenti, koja uključuje samo adresu i port računara, ali ne i nekakav ID kojim bi se razlikovale komponente. Ovako, svaka

komponenta prima sve poruke koje su pristigle na računar i sama mora da se pobrine da izdvoji onu poruku koja je za nju. Osim ovoga, ne postoji direktna podrška za mobilnost agenata, kao ni za direktorijum agenata. Sistem ne podržava FIPA specifikaciju.

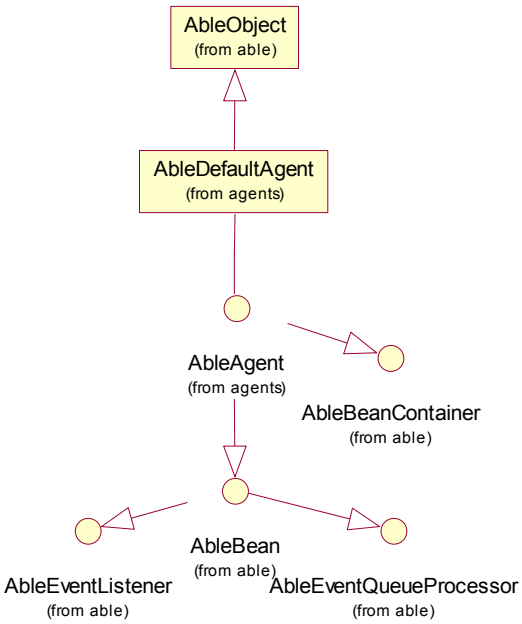
2.2 ABLE agentsko okruženje

ABLE (*Agent Building and Learning Environment*) [Bigus, Bigus00] je okruženje za izgradnju agenata i primenu algoritama za mašinsko učenje i rezonovanje. Napisan je u programskom jeziku Java i, osim agentskog okruženja, predstavlja i biblioteku komponenti i razvojno okruženje za implementaciju mašinskog učenja. ABLE je projekat *IBM T.J.Watson Research Center* istraživačkog centra.

2.2.1 Arhitektura

ABLE agentsko okruženje se sastoji iz Java interfejsa i klasa koji omogućavaju izgradnju JavaBeans komponenti, nazvanih *AbleBeans*. Ove komponente pre svega implementiraju zaključivanje na osnovu pravila (*Rule Set Inferencing*) koristeći tehnike Bulove i fazi logike, mašinsko učenje bazirano na nekoliko osnovnih modela poput neuralnih mreža i stabla odlučivanja, kao i osnovne manipulacije podacima, kao što su pisanje i čitanje tekstualnih podataka, rad sa bazama podataka, kao i obradu podataka. U distribuciju je uključen poseban paket koji mapira FIPA specifikaciju na ABLE agentsko okruženje.

Osnovna komponenta ovog okruženja je *AbleAgent* komponenta. Ova komponenta predstavlja okvir za implementaciju agenata. Agenti sadrže *AbleBean* komponente i kontrolišu njihov rad. Slika 2.5 prikazuje dijagram klasa koji opisuje bitne elemente *AbleAgent* arhitekture.



Slika 2.5: Dijagram klasa *AbleAgent* komponente

`AbleAgent` je interfejs koji je naslednik **`AbleBean`** interfejsa. Ova dva interfejsa definišu osnovni skup metoda koje svaki agent mora da implementira. Te metode su:

- **`process()`** – metoda koja izvršava zadatak. Obično ova metoda otvara ulazni tok podataka, preuzima iz njega parametre, izvršava zadatak, otvara izlazni tok i preko njega šalje rezultate rada;
- **`getInputBuffer()`** i **`getOutputBuffer()`** - vraćaju reference na ulazni i izlazni tok podataka;
- **`setTimerEventProcessingEnabled()`** – uključuje/isključuje periodično procesiranje zadataka.

`AbleBean` interfejs implementira **`AbleDefaultAgent`** klasa, koja predstavlja osnovnu agentsku komponentu ovog agentskog okruženja. Svi agenti nasleđuju ovu klasu. **`AbleDefaultAgent`** nasleđuje **`AbleObject`** klasu. **`AbleObject`** klasa je ključna klasa u ovoj arhitekturi pošto sadrži sve potrebne metode za rad sa okruženjem i *AbleBean* komponentama (serijalizaciju komponente u tok ili datoteku, povezivanje

sa odgovarajućim ulaznim i izlaznim tokovima podataka, procesiranje zadatka dodeljenog ovoj komponenti i dr.). Za rad *AbleBean* komponenti zadužena je klasa koja implementira **`AbleBeanContainer`** interfejs.

Za komunikaciju među komponentama podržan je model događaja (*Event driven model*). Ovaj model omogućuje iniciranje softverskih događaja, kao reakciju na spoljašnje ili unutrašnje događaje. Interfejs **`AbleEventQueueProcessor`** definiše metode koje upravljaju događajima (**`processAbleEvent()`** i **`processTimerEvent()`**). Komponenta koja se registruje za određeni tip događaja dobija objekte odgovarajuće klase događaja, kao reakciju na njihovo pojavljivanje. Reakcija na događaje je definisana interfejsom **`AbleEventListener`**, odnosno njegovom metodom **`handleAbleEvent()`**.

2.2.2 Zaključivanje bazirano na pravilima

Able agentsko okruženje omogućuje zaključivanje bazirano na pravilima. Ova pravila se pišu u ARL jeziku (*Able Rule Language*). Jezik omogućuje pisanje širokog spektra pravila, počev od jednostavnih *if-then* pravila, pa do složenih pravila predikatske logike (napisanih u Prologu). Sintaksa ARL je slična Javi i podržava neke osnovne Java koncepte, kao što su kreiranje Java objekata i Java logičke izraze. ARL izrazi se prevode u **`AbleRuleSet`** objekte koji izvode definisano zaključivanje kada se prozovu.

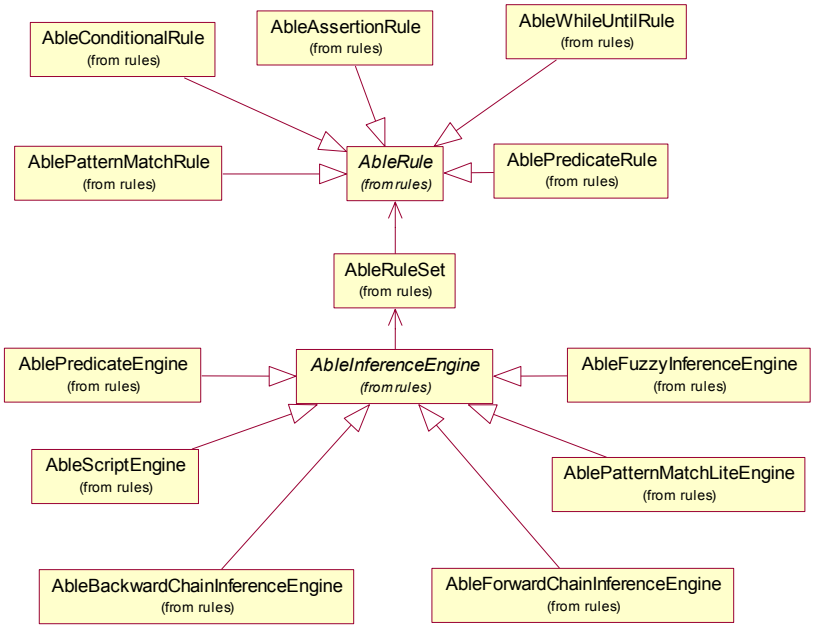
Za rad sa pravilima, napravljen je *RuleSet Editor*, editor pravila koji omogućuje kako pisanje, tako i proveru i izvršavanje pravila. Sva pravila se prevode u serijalizabilne Java objekte (objekte **`AbleRuleSet`** klase), koji se mogu učitavati od strane Able okruženja. Osim prevođenja tekstualnih pravila u Java objekte i njihovog snimanja/učitavanja, Able agentsko okruženje omogućava i njihovo izvršavanje.

Osnovna klasa koja reprezentuje pravilo je **`AbleRule`** klasa. Ova klasa reprezentuje generičko pravilo i ne koristi se direktno. Konkretna pravila su opisana klasama naslednicama ove klase. Dozvoljeno je proizvoljno kombinovanje pravila, dokle god se to ne kosi sa odabranim sistemom zaključivanja. Za proveru ispravnosti koda koristi se ugrađeni validator koda.

Svi sistemi zaključivanja se baziraju na **`AbleRule`** klasi, odnosno njenim klasama naslednicama: **`AbleAssertionRule`** (dodela, poput "age = 56"), **`AbleConditionalRule`** (definiše uslov, listu pravila koja se pozivaju ako je uslov zadovoljen (*if*) i listu pravila koje se pozivaju ako uslov nije

zadovoljen (*else*), **`AbleWhileUntilRule`** (petlja, definisana uslovom i listom pravila koja se pozivaju dokle god je uslov zadovoljen), **`AblePatternMatchRule`** (pravila za *pattern matching*) i **`AblePredicateRule`** (pravila definisana klasom **`AblePredicateEngine`**).

Slika 2.6 prikazuje dijagram klasa koji opisuje **`AbleRuleSet`** memorijsku strukturu.



Slika 2.6: Dijagram klasa **`AbleRuleSet`** memorijske strukture

Na slici 2.6 prikazani su podržani sistemi zaključivanja. To su: *Script* (izrazi koji se izvršavaju u sekvenci, bez dodatnih kontrola), *Backward chaining*, *Forward chaining*, *Pattern Matching*, Fazi logika i predikatski račun.

2.2.3 AbleRuleSet struktura

Struktura pravila je prikazana u listingu 2.2:

```
ruleset <nameOfRuleSet> {

<import package.class;>*  // Zero or more statements
<library package.class;>* // Zero or more statements

variables { // Global variable declaration section
<Variable Declaration Statement>+ // One or more statements
}
inputs { <variableName>* } // Exactly one statement, zero
or more names
outputs{ <variableName>* } // Exactly one statement, zero
or more names

functions { <name/arity>* }* // Zero or more statements,
zero or more names

void init() { <rule>+ };

void main() using <Inference Engine> {

<rule>+ // One or more statements
}

void idle(){

}
```

Listing 2.2: Struktura AbleRuleSet zapisa

Svaki zapis počinje **`ruleset`** ključnom reči. Učitavanje spoljašnjih klasa je omogućeno ključnim rečima **`import`** i **`library`**. Njihova sintaksa je ista kao u Javi i označava spisak klasa iz zadatih paketa, čiji će se objekti kreirati i čije će se metode i atributi koristiti.

Promenljive se deklarišu upotrebom ključne reči **`variables`**. Ova ključna reč označava zonu u kojoj se deklarišu varijable, koje mogu biti sledećeg tipa:

- *Boolean*
- *Categorical* - sadrži listu stringova,
- *Continuous* - sadrži raspon brojeva od donje do gornje granice,
- *Discrete* - sadrži listu brojeva,
- *Fuzzy* - sadrži Fazi promenljivu, koja se sastoji iz donje granice, gornje granice vrednosti promenljive, kao i niza Fazi skupova definisanih nad ovom varijablom,

- *List* - liste: kompletan skup metoda za rad sa listama je implementiran u **`com.ibm.able.beans.rules.AbleListLib`** paketu,
- *Numeric* - sadrži numeričku vrednost, koja se interno čuva kao **`double`** vrednost,
- *Object* - sadrži proizvoljan Java objekat,
- *String* - sadrži Java String i
- *<userType>* - korisnički definisana varijabla, kao instanca korisnički definisane klase.

Za komunikaciju sa sistemom, omogućen je ulazni i izlazni tok podataka upotrebom ključnih reči **`inputs`** i **`outputs`**. U ovim sekcijama se navode one varijable, čija će se vrednost podešavati iz ulaznog toka, odnosno čija će se vrednost prebacivati u izlazni tok.

Korisnički definisane funkcije se nalaze u **`functions`** bloku. Ovaj blok sadrži deklaracije funkcije koje se pozivaju iz ostatka koda, a definisane su spolja. Ove funkcije je potrebno programski uključiti u sistem pozivom metode **`addUserDefinedFunction()`** klase **`AbleRuleSet`**.

Poslednja sekcija je **`AbleRuleBlock`** sekcija. Ova sekcija se sastoji iz **`init()`**, **`main()`** ili **`idle()`** bloka, kao i iz korisnički definisanih blokova (koji se moraju pozvati **`invokeRuleBlock()`** ugrađenom funkcijom). Blok **`init()`** služi za inicijalizaciju varijabli, **`main()`** blok počinje izvršenje algoritma (koji će se sistem zaključivanja koristiti definisano je ključnim rečima "**`using <inference engine>`**"), a **`idle()`** blok se poziva nakon završetka algoritma. Blok **`idle()`** ne mora da sadrži kod, ako nema potrebe za posebnom akcijom nakon završetka algoritma.

2.2.4 Korišćenje Able Rule Language programa

ARL program se može proveriti sintaksno, startovati i debagirati iz *RuleSetEditor*-a. Ovaj editor predstavlja integrisano okruženje koje obuhvata editor, interpreter i debager. Osim ovoga, moguće je i dodeliti ARL program odgovarajućem AbleBean-u, čime će se omogućiti njegovo izvršavanje u Able okruženju. Poslednja mogućnost je da se željeni ARL program startuje programski, iz proizvoljnog Java koda. Listing 2.3 ilustruje programsko uključenje ARL programa.

```
// Kreiranje RuleSet-a
AbleLogger  lclTracerP = Able.TraceLog;
AbleRuleSet lclRuleSet = new AbleRuleSet();
// Kreiranje fazi RuleSet osluškivača
AbleFuzzyChgListener lclFuzzyListener  = new
AbleFuzzyChgListener();
lclRuleSet.addRuleSetChangeListener(lclFuzzyListener);
// Povezivanje datoteke sa izvornim kodom i RuleSet-a.
String lclPath =
Able.ProductDirectory+"examples"+File.separator+"rules"+Fil
e.separator;
lclRuleSet.instantiateFrom(lclPath +
"SampleSensorEffector.arl", lclTracerP, false);
// Definisanje eksternih funkcija i njihovo povezivanje sa
RuleSet-om.
// Funkcije su metode ove klase (tstLockMethod i
setLockMethod).
Class lclClass     = this.getClass();
Method lclSenMethod= lclClass.getMethod("tstLockMethod",
new Class[] {Object.class});
Method lclEffMethod= lclClass.getMethod("setLockMethod",
new Class[] {Object.class});
AbleUserDefinedFunction lclSensor   = new
AbleUserDefinedFunction("tstLock", this, lclSenMethod);
AbleUserDefinedFunction lclEffector = new
AbleUserDefinedFunction("setLock", this, lclEffMethod);
lclRuleSet.addUserDefinedFunction(lclSensor);
lclRuleSet.addUserDefinedFunction(lclEffector);
lclRuleSet.init();
// Smeštanje podataka u ulazni bafer.
Object[] lclInpBuffer =
(Object[])lclRuleSet.getInputBuffer();
lclInpBuffer[0] = new Boolean(true);
lclInpBuffer[1] = new Double(9.9);
lclInpBuffer[2] = "Ghi";
lclInpBuffer[3] = new Double(88.88);
// Procesiranje fazi RuleSet-a.
lclRuleSet.process();
// Prikupljanje rezultata iz izlaznog bafera.
Object[] lclOutBuffer =
(Object[])lclRuleSet.getOutputBuffer();
for (int i=0; i<lclOutBuffer.length; i++) {
  if (trace.isLogging()) trace.text(
    Able.TRC_HIGH,this,"runSample","!! Output element[" +
      i + "]: <" + lclOutBuffer[i] + ">");
}
```

Listing 2.3: Primer programskog uključenja ARL programa

Listing 2.3 takođe ilustruje i uključenje eksternih funkcija koje će biti pozvane iz ARL programa (listing 2.4). U listingu 2.3 su uključene dve funkcije (**`lclSensor`** i **`lclEffector`**) sa parametrima, a u listingu 2.4, su deklarisane i pozvane iz ARL programa.

```
//-------------------------------------------------------
// External user-defined functions are declared below.
//-------------------------------------------------------
functions {setLock/1, tstLock/1 };

void main()  using FuzzyAdd () {
  //-------------------------------------------------------
  // Set some values by calling user-defined functions.
  //-------------------------------------------------------
  S1: vB2 = tstLock(vB3);
  S2: vN2 = tstLock(vN3);

  //-------------------------------------------------------
  // Some conditional rules making use of user-defined
  // functions.
  //-------------------------------------------------------
  R1: if (vB2 == true)
    then  vB2 = setLock(vB3);

  R2: if (vB2 == true)
    then  vN2 = setLock(vN3);
```

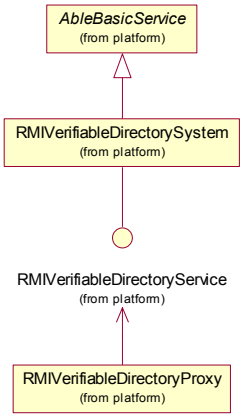
Listing 2.4: Deo ARL programa koji deklariše i poziva spoljašnje funkcije

2.2.5 Mobilni agenti

Mobilni agenti su implementirani u paketu **`com.ibm.able.platform`**. Ovaj paket sadrži sledeće elemente potrebne za implementaciju mobilnih agenata:

- direktorijum agenata (**`RMIVerifiableDirectorySystem`**),
- sistem imenovanja agenata (**`RMIVerifiableNamingSystem`**) i
- sistem za prenos poruka (**`AbleJasMessageSystemAdapter`**).

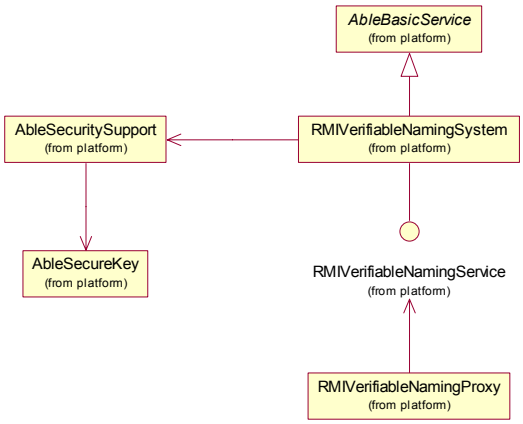
Direktorijum agenata je implementiran klasama predstavljenim na slici 2.7.



Slika 2.7: Dijagram klasa direktorijuma agenata

Klasa **`RMIVerifiableDirectorySystem`** implementira funkcionalnost direktorijuma agenata. Ova klasa služi za registrovanje, deregistrovanje i pretragu direktorijuma agenata. Ova klasa implementira interfejs **`RMIVerifiableDirectoryService`**, gde su specificirane metode za rad sa direktorijumom agenata, a nasleđuje klasu **`AbleBasicService`**, čime je automatski povezana sa podsistemom za generisanje događaja i vođenje dnevnika aktivnosti. Klasa **`RMIVerifiableDirectoryProxy`** povezuje RMI sistem sa direktorijumom agenata.

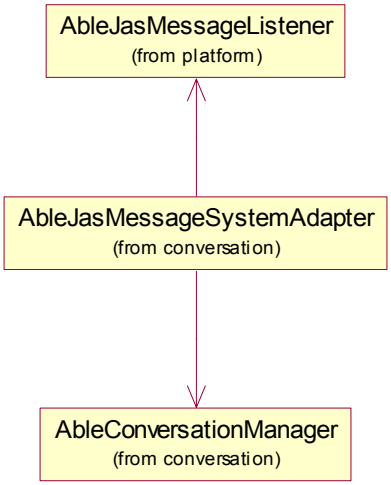
Sistem imenovanja agenata je prikazan na slici 2.8.



Slika 2.8: Dijagram klasa sistema imenovanja

Sistem imenovanja agenata obezbeđuje jedinstveno imenovanje agenata na nivou celog sistema i implementira sigurnosni podsistem. Imenovanje agenata je definisano **`RMIVerifiableNamingService`** interfejsom, a implementirano u klasi **`RMIVerifiableNamingSystem`**, koja nasleđuje **`AbleBasicService`**. Povezivanje ovog sistema sa RMI sistemom je implementirano u klasi **`RMIVerifiableNamingProxy`**. Sigurnosni mehanizmi su svedeni na kriptografsku zaštitu komunikacije, a implementirani u klasama **`AbleSecuritySupport`** i **`AbleSecureKey`**. Ove dve klase omogućavaju kreiranje privatnih i javnih ključeva, enkripciju, dekripciju, kao i potpisivanje i verifikaciju poruka.

Sistem za transport poruka je prikazan na slici 2.9.



Slika 2.9: Dijagram klasa sistema za transport poruka

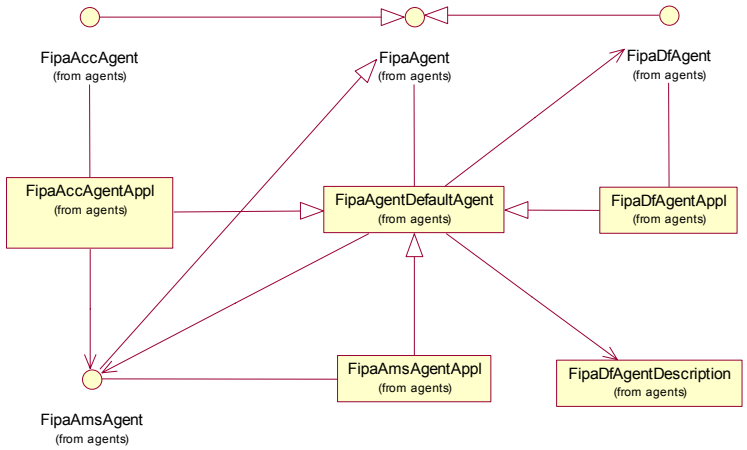
Centralna klasa sistema za razmenu poruka je **`AbleJasMessageSystemAdapter`** klasa. Ova klasa koristi klasu **`AbleConversionManager`** za implementaciju svih potrebnih metoda za razmenu poruka. Klasa **`AbleJasMessageListener`** predstavlja klasu kojom se agent registruje za prijem poruka.

2.2.6 Implementacija FIPA specifikacije

Poseban paket (**`com.ibm.able.platform.agents`**) je uključen u distribuciju za podršku implementaciji FIPA specifikacije [FIPA]. Ovaj paket implementira sledeće koncepte:

- Agent Communication Channel (ACC),
- ACL message,
- Agent Management System (AMS),
- Agent Platform,
- Directory Facilitator (DF),
- FIPA letter,
- FIPA "message-envelope" i
- FIPA Service description.

Slika 2.10 prikazuje dijagram klasa koje implementiraju FIPA specifikaciju.



Slika 2.10: Dijagram klasa FIPA implementacije

Osnovna klasa koja implementira FIPA specifikaciju je **`FipaAgentDefaultAgent`**. Ona implementira **`FipaAgent`** interfejs. Nju nasleđuju svi provajderi usluga (**`FipaAccAgentAppl`**, **`FipaDfAgentAppl`** i **`FipaAmsAgentAppl`**). Ovi provajderi usluga su napravljeni kao Java aplikacije i svi koriste RMI tehnologiju. Klasa **`FipaAccAgentAppl`** (implementira **`FipaAccAgent`** interfejs) je provajder komunikacionih usluga. Namena mu je da prenosi poruke između registrovanih agenata. Klasa **`FipaDfAgentAppl`** (implementira **`FipaDfAgent`** interfejs) je provajder usluga direktorijuma agenata. Omogućava registraciju, odjavljivanje, modifikaciju i pretragu agenata. Informacija o agentu (koja

se smešta u direktorijum) se nalazi u klasi `FipaDfAgentDescription`. Klasa `FipaAmsAgentAppl` implementira `FipaAmsAgent` interfejs i predstavlja menadžerski servis koji kreira prethodna dva servisa (ako već ne postoje) i odgovoran je za kreiranje, izvršavanje i uništenje agenata.

2.2.7 Karakteristike Able agentskog okruženja

Kada je u pitanju implementacija mašinskog učenja, Able agentsko okruženje predstavlja dobru implementaciju. Podržan je širok spektar metoda, počev od Fazi logike, preko neuralnih mreža, pa do predikatskog računa. Takođe, obezbeđen je kompletan podsistem za interpretaciju ARL jezika, koji omogućuje implementaciju navedenih metoda. Osim ovoga, sistem omogućuje programsko uključenje ARL programa u proizvoljan Java program, čime se korisnik ne vezuje za integrisano okruženje Able agentskog okruženja. Takođe, dobra osobina je to da je implementirana FIPA specifikacija. Mana ovog sistema je da se koristi RMI umesto naprednijih sistema za distribuirane komponente.

2.3 Voyager

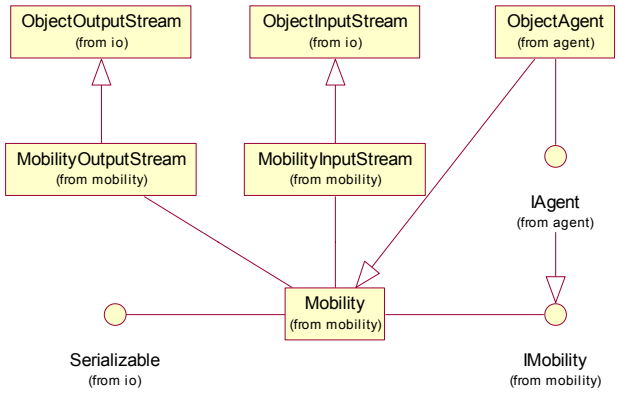
Voyager [Voyager] je aplikacioni server firme Objectspace. Voyager podržava EJB, CORBA, RMI, DCOM, JTA, JDBC i SSL standarde. Paket se sastoji iz sledećih elemenata:

1. ***Voyager Application Server*** – EJB server, koji podržava i *Java Server Pages* (JSP).
2. ***Voyager ORB*** - *Object Request Broker* (ORB) koji omogućava komunikaciju između Voyager-a, SOAP, CORBA, RMI i DCOM objekata. Sadrži takođe i Distributed Naming Service, podsistem za perzistenciju objekata i dr.
3. ***Voyager ORB Professional*** – izgrađen na Voyager ORB sistemu, pruža grafičku konzolu za menadžment, *Java Naming and Directory Interface* (JNDI) servis, *Load Balancing* servis, *CORBA naming service* i dr.
4. ***Voyager Security*** – sistem za autentifikaciju i autorizaciju, *Secure Socket Layers* (SSL) i *Firewall Tunneling* upotrebom SOCKS protokola, kao i HTTPs protokol.
5. ***Voyager Transactions*** – podržava Object Transaction Service (OTS) i JTA.

2.3.1 Podrška za agentsku tehnologiju

Voyager nije samo aplikacioni server. On podržava u izvesnoj meri i agentsku tehnologiju [Ganguly00].

Paketi: **`com.objectspace.voyager.mobility`** i **`com.objectspace.voyage.agent`** omogućavaju proizvoljnim klasama da implementiraju agentski i mobilni interfejs, koji im omogućava da proizvoljne klase učine mobilnim u smislu da one mogu da se "presele" na drugi računar i tamo nastave izvršavanje. Slika 2.11 prikazuje dijagram klasa sistema za podršku mobilnosti i agenata.



Slika 2.11: Dijagram klasa mobilnosti koda i agenata

Ključna klasa za mobilnost koda je klasa **`Mobility`**. Ova klasa omogućuje proizvoljnoj klasi (koja za minimum mora da implementira interfejs **`java.io.Serializable`** – da bi mogla da se serijalizuje, i, time da se prenese preko mreže) da se "prenese" na drugi računar. Ključne metode za postizanje ovog cilja su metode **`of(Object obj)`** i **`moveTo()`**. Prva metoda "promoviše" proizvoljni objekat u mobilni objekat, a druga takav mobilni objekat prenosi na drugi računar. Klasa **`ObjectAgent`** je samo okvir za **`IAgent`** interfejs koji nasleđuje **`IMobility`** interfejs i proširuje ga dodatnim metodama vezanim za agentsku tehnologiju (metode **`preDeparture()`**, **`postDeparture()`**, **`preArrival()`** i **`postArrival()`** koje se pozivaju pre, odnosno posle odlaska i dolaska objekta na drugi računar). Prenos konteksta agenta je implementiran dvema klasama: **`MobilityOutputStream`** i **`MobilityInputStream`**. Ove

klase obavljaju snimanje odnosno učitavanje agenta na takav način da je sistem svestan nove lokacije preseljenog agenta, tako pozivi njegovih metoda završavaju na drugom računaru, gde se on zapravo nalazi.

2.3.2 Karakteristike Voyager aplikacionog servera

Voyager aplikacioni server poseduje sistem mobilnosti, kao jedan od najbitnijih obeležja agentske tehnologije, ali ne poseduje ni jedan od ostalih bitnih agentskih koncepata, kao što su direktorijum agenata, razmena poruka (KQML pogotovo), kao ni mogućnost za međuagentsko uređenje odnosa. Sistem ne podržava FIPA specifikaciju.

2.4 Aglets

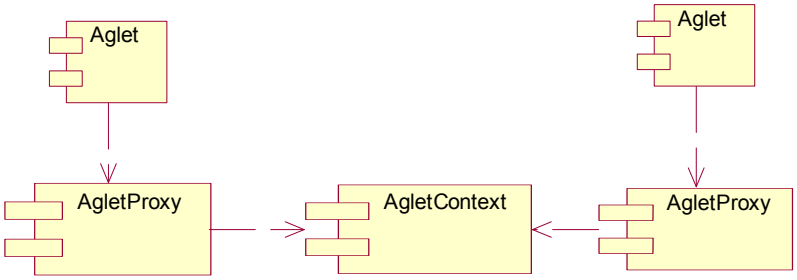
Aglets [Aglets, Tai99, Thai03] je agentsko okruženje nastalo u IBM-ovoj istraživačkoj laboratoriji u Tokiju (*Tokyo Research Labaratory*). Sistem je napisan u programskom jeziku Java i podržava mobilnost agenata, komunikaciju između agenata na nivou razmene poruka (klasa **`Message`**) i bezbednosne mehanizme [Varadharajan00, Wang01] na nivou pristupa fajl sistemu, mrežnim resursima i programskim nitima.

2.4.1 Arhitektura

Aglets sistem je višeslojni sistem, koga čine sledeći slojevi:

1. *AgletsRuntime* sloj – sloj zadužen za izvršavanje agenata,
2. ATCI sloj (*Agent Transport and Communication Interface*) – sloj zadužen za komunikaciju između agenata,
3. ATP sloj (*Agent Transfer Protocol*) – sloj zadužen za transport agenata preko mreže i
4. Transportni sloj (TCP/IP mreža).

AgletsRuntime sloj se sastoji iz agenata (nazvanih Agleti), agentskih proksija i agentskih konteksta (slika 2.12).



Slika 2.12: Dijagram komponenti *Aglet Runtime* sloja

Aglet komponenta (klasa **`com.ibm.aglet.Aglet`**) predstavlja osnovnu klasu za reprezentaciju mobilnih agenata. Mobilni agenti koji se prave moraju da naslede ovu klasu. *AgletProxy* komponenta (klasa **`com.ibm.aglet.AgletProxy`**) predstavlja međusloj između *Aglet* komponente i drugih komponenti. Ona implementira bezbednosne mehanizme kojima se sistem štiti od malicioznih *Aglet* komponenti, a štiti i *Aglet* komponentu od drugih komponenti. Osim ovoga, *AgletProxy* implementira osnovne mehanizme za mobilnost agenata. Komponenta *AgletContext* predstavlja vezu između *Aglet* komponente i *AgletsRuntime* sloja.

ATCI sloj enkapsulira konekcije između Aglets agentskih okruženja. Uz pomoć njega, ATP sloj šalje i prima mobilne agente preko mreže. ATP sloj podržava sledeće operacije:

1. *Dispatch* – slanje agenta na udaljeni server;
2. *Retract* – vraćanja agenta sa udaljenog sistema nazad na polazni;
3. *Fetch* – ekvivalent GET komande HTTP protokola, tj. služi za slanje i prijem informacija;
4. *Message* – slanje poruka preko mreže.

2.4.2 Životni ciklus Agleta

Aglet komponente započinju životni ciklus ili kreiranjem (**`createAglet()`** metoda) ili kloniranjem (**`clone()`** metoda). Klonirani Aglet je identičan originalnom, samo što ima drugačiji **`AgletID`** objekat koji ga identifikuje. Aglet može da pošalje sebe na drugi server metodom **`dispatch()`**, da se deaktivira, tj. da se prebaci u skladište podataka (serijalizuje i smesti u skladište) metodom **`deactivate()`**, kao i da se ukloni sa servera metodom **`dispose()`**.

Po kreiranju Agleta, poziva se metoda **`onCreation()`**. Ova metoda se poziva tačno jednom i tu se može staviti inicijalizacioni kod koji zavisi od Aglet API-ja (u konstruktoru ovaj API nije dostupan).

Ako se Aglet kreira, klonira ili rekonstruiše iz skladišta podataka, nakon inicijalizacije poziva se metod **`run()`**. Ovo je odgovarajuće mesto za kod koji treba da obavlja namenski zadatak agenta.

Po uklanjanju Agleta, poziva se **`onDisposing()`** metoda. Ovde se može staviti kod koji oslobađa resurse.

2.4.3 Događaji u Aglets sistemu

Aglets sistem podržava model događaja. Događaji (*Events*) predstavljaju objekte određenih klasa (**`XxxEvent`**) koji bivaju prosleđeni odgovarajućem *listener*-u, po pojavi događaja. Ovim je omogućena asinhrona reakcija na događaje. Postoji konačan broj događaja koje sistem može da prati: *CloneEvent* (događaj kloniranja Agleta), *MobilityEvent* (kada se Aglet šalje preko mreže) i *PersistencyEvent* (kada se deaktivira). Za ove događaje su razvijene odgovarajuće klase događaja i *Listener*-a (tabela 2.2).

Događaj	**Klasa događaja**	***Listener***	**Metod *listener*-a**
pre kloniranja	**`CloneEvent`**	**`CloneListener`**	**`onCloning`**
klon agleta je kreiran	**`CloneEvent`**	**`CloneListener`**	**`onClone`**
nakon kloniranja	**`CloneEvent`**	**`CloneListener`**	**`onCloned`**
pre slanja Agleta	**`MobilityEvent`**	**`MobilityListener`**	**`onDispatching`**
pre vraćanja Agleta	**`MobilityEvent`**	**`MobilityListener`**	**`onReverting`**
nakon prispeća na odredište	**`MobilityEvent`**	**`MobilityListener`**	**`onArrival`**
pre deaktiviranja	**`PersistencyEvent`**	**`PersistencyListener`**	**`onDeactivating`**
nakon aktiviranja	**`PersistencyEvent`**	**`PersistencyListener`**	**`onActivation`**

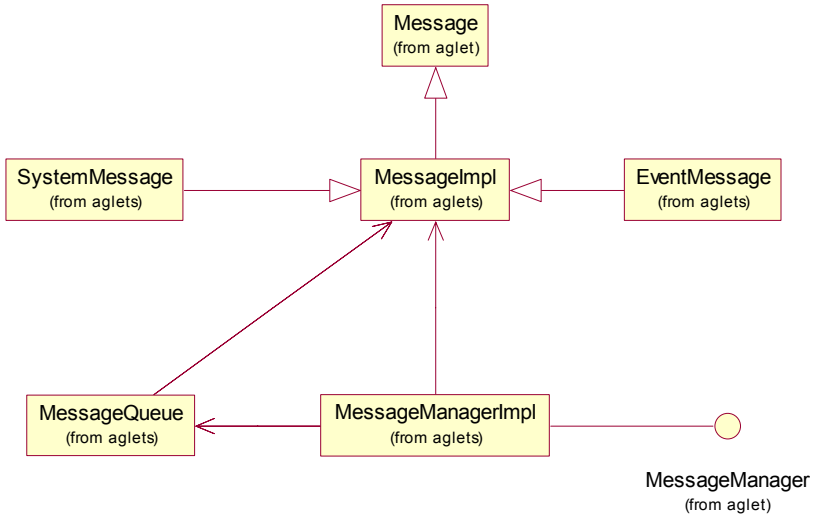
Tabela 2.2: Događaji u Aglets sistemu

2.4.4 Poruke u Aglets sistemu

Sistem podržava sledeće tipove poruka:

1. sinhrone poruke koje blokiraju izvršenje dok se ne dobije odgovor (**`sendMessage()`** metoda);
2. asinhrone poruke koje ne blokiraju izvršenje (**`sendAsyncMessage()`** metoda);

Za prijem poruke potrebno je redefinisati **`handleMessage()`** metodu. Slika 2.13 prikazuje dijagram klasa mehanizma za razmenu poruka.



Slika 2.13: Dijagram klasa sistema za razmenu poruka

`MessageManagerImpl` klasa je osnovna klasa, koja implementira **`MessageManager`** interfejs. Ona koristi **`MessageQueue`** klasu za smeštanje poruka u red. Poruke su klase naslednice klase **`Message`**. Postoje tri tipa poruka: opšte poruke implementirane klasom **`MessageImpl`**, sistemske poruke implementirane klasom **`SystemMessage`** i poruke inicirane događajima (klasa **`EventMessage`**).

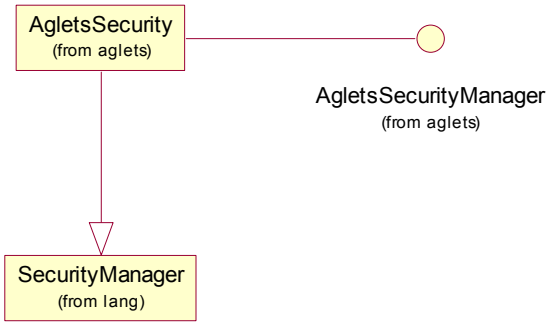
2.4.5 Bezbednost u Aglets sistemu

Bezbednosni mehanizam pokriva tri bitna aspekta: pristup fajl sistemu, pristup mrežnim resursima i pristup programskim nitima. Prema ovom sistemu, Agleti se dele na proverene i neproverene (*trusted* i *untrusted*).

Prvi su Agleti učitani sa sigurnog sistema, tj. lokalnog fajl sistema; svi ostali Agleti su *untrusted*. Postoje posebna podešavanja za oba tipa Agleta i nalaze se u sledećim datotekama:

- **`$HOME/.aglets/security/trusted`** i
- **`$HOME/.aglets/security/untrusted`**.

Pristup lokalnom fajl sistemu, mrežnim resursima i ostalim komponentama je regulisan *AgletsSecurity* podsistemom (slika 2.14).



Slika 2.14: Dijagram klasa *AgletsSecurity* podsistema

AgletsSecurity podsistem se zasniva na klasi **`AgletsSecurity`** koja nasleđuje **`java.lang.SecurityManager`** klasu i implementira interfejs **`AgletsSecurityManager`**. Ova klasa se postavlja kao sistemski *SecurityManager*. To ima za posledicu da se svaki pristup navedenim resursima proverava preko odgovarajućih metoda ove klase (na primer, svaki pokušaj pristupa lokalnoj datoteci za pisanje se proverava pozivom metode **`checkWrite()`**).

2.4.6 Karakteristike Aglets agentskog okruženja

Aglets sistem se odlikuje temeljnim implementacijama mobilnosti i bezbednosti. Mobilnost je podržana funkcijama **`onDispatching()`**, **`onReverting()`** i **`onArrival()`**. Bezbednost je implementirana **`AgletsSecurity`** klasom, čiji rad se zasniva na ugrađenim Java bezbednosnim mehanizmima, a koji omogućuje kompletnu bezbednosnu kontrolu proizvoljnih Agleta. Sistem poruka nije do kraja objektno realizovan jer se koristi metoda **`handleMessage()`** a ne odgovarajući *listener*. Osim ovoga, nisu posebno podržane KQML poruke. Nije podržan ni direktorijum agenata, kao ni mogućnost za međuagentsko uređenje odnosa. Velika mana ovog sistema predstavlja zahtev da se

koristi Java JRE verzija 1.1, a ne novije. Sistem ne podržava FIPA specifikaciju.

2.5 JAT Lite

JAT Lite (*Java Agent Template*) [JAT, Jeon00] je agentsko okruženje napravljeno na Stenford univerzitetu. Sistem je zamišljen kao okruženje u kojem agenti izvršavaju svoje zadatke i razmenjuju poruke. Sistem se sastoji iz agenata i rutera poruka.

2.5.1 Arhitektura

JAT Lite agentsko okruženje se sastoji iz pet slojeva:

1. Apstraktni sloj (*Abstract Layer*)
2. Osnovni sloj (*Base Layer*)
3. KQML sloj (*KQML Layer*)
4. Sloj za rutiranje (*Router Layer*)
5. Sloj za protokole (*Protocol Layer*)

Apstraktni sloj sadrži apstraktne klase koje će biti implementirane u višim slojevima. Ovaj sloj se sastoji iz sledećih apstraktnih klasa: **`AgentAction`**, **`ServerThread`**, **`ReceiverTherad`**, **`MessageBuffer`**, **`ConnectionTable`**, **`Address`**, **`AddressTable`** i **`Security`**.

Osnovni sloj obezbeđuje osnovnu TCP/IP komunikaciju (bez specifikacije viših nivoa protokola).

KQML sloj obezbeđuje podršku za razmenu KQML poruka.

Sloj za rutiranje obezbeđuje rutiranje poruka.

Sloj za protokole obezbeđuje više protokole poput SMTP i FTP protokola. Ako agenti moraju da razmenjuju poruke preko SMTP ili FTP protokola, ovaj sloj sadrži rutine potrebne za rad sa spomenutim protokolima.

2.5.2.Agenti u JAT Lite sistemu

`AgentAction` apstraktna klasa je osnovna klasa za implementaciju agenata. Tabela 2.3 sadrži spisak metoda koje je potrebno redefinisati da bi se agent mogao implementirati.

Naziv	Opis
`createServerThread()`	Kreira serversku nit koja ima za zadatak da čeka na određenom portu konekciju sa drugim agentima.
`createReceiverThread()`	Nakon uspostavljanja veze sa drugim agentom, objekat ove klase se kreira za komunikaciju između agenata.
`sendMessage()`	Šalje zadatom agentu poruku. Agent se identifikuje po ID-u.
`broadCast()`	Šalje svim konektovanim agentima poruku.
`processMessage()`	Procesira pristiglu poruku.
`Act()`	Poziva se kada god stigne poruka od drugog agenta.
`run()`	Glavna metoda za implementaciju. Ovde se smešta sva logika koju je potrebno implementirati.
`endAction()`	Poziva se prilikom prestanka rada agenta.

Tabela 2.3: Metode za redefiniciju prilikom nasleđivanja `AgentAction` klase

Svaki agent implementira svoju funkcionalnost u `run()` metodi. Uobičajeno je da se ovde smešta kod koji proverava red poruka i koji za svaku poruku poziva `Act()` metodu.

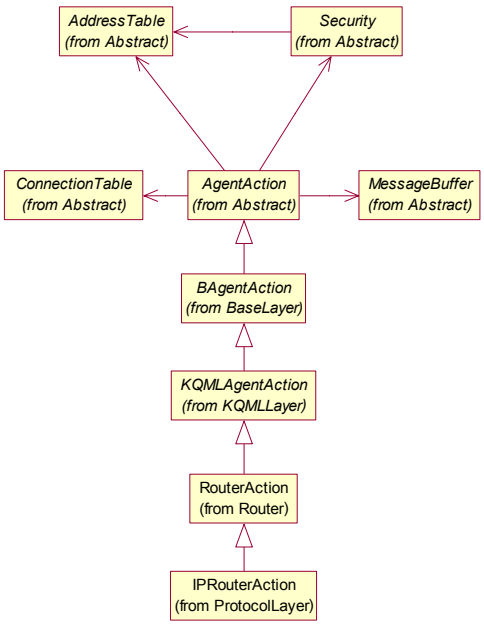
Osim `AgentAction` apstraktne klase, postoji i `BAgentAction` apstraktna klasa koja implementira sve navedene metode osim `processMessage()` i `Act()` metode. Nasleđivanjem ove klase nije potrebno implementirati metode koje se bave komunikacijom, već je dovoljno samo implementirati metode koje se bave izvršenjem zadatka i procesiranjem pristigle poruke. Sve preostale metode su implementirane tako da se komunikacija odvija upotrebom TCP/IP protokola.

Slika 2.15 prikazuje klase koje nasleđuju `AgentAction` klasu i implementiraju svoju zadatu funkcionalnost.

Klasa **KQMLAgentAction** nasleđuje **BAgentAction** klasu i takođe je apstraktna klasa, ali je prilagođena obradi KQML poruka.

Klasa **RouterAction** nasleđuje **KQMLAgentAction** i omogućuje odloženo rutiranje poruka između agenata, u smislu da, ako odredišni agent nije aktivan, poruku snima u skladište i kada se agent aktivira, šalje mu poruku iz skladišta. Takođe, poruke se mogu slati u zadato vreme. Ova klasa održava listu on-line agenata (agenata koji su aktivni). Ovu listu mogu da dobiju svi prijavljeni agenti.

IPRouterAction nasleđuje **RouterAction** i obezbeđuje podršku još za SMTP i FTP protokole.



Slika 2.15: Dijagram klasa koje nasleđuju **AgentAction** apstraktnu klasu

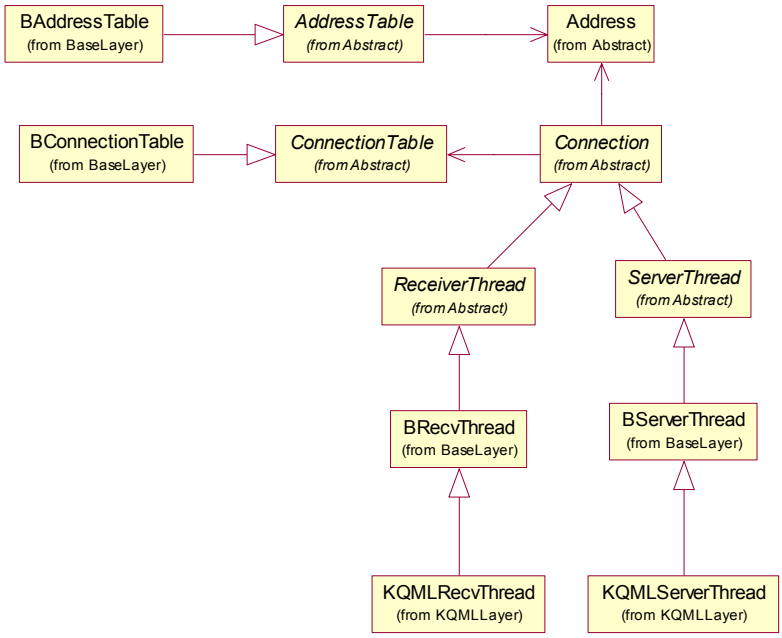
Klasa **AgentAction** sadrži veze sa osnovnim mehanizmima koji su potrebni za funkcionisanje agenta. To su: **ConnectionTable** (za održavanje liste konekcija sa agentima koji komuniciraju sa ovim agentom), **AddressTable** (za održavanje liste agenata koji komuniciraju sa ovim agentom), **Security** (za implementaciju bezbednosnih mehanizama) i **MessageBuffer** (za komunikaciju razmenom poruka).

2.5.3 Komunikacija

Komunikacija između agenata se odvija razmenom poruka. Ove poruke se prenose IP, SMTP i FTP protokolom. U osnovi svake komunikacije su apstraktne klase **`AddressTable`** i **`ConnectionTable`** (slika 2.16).

`AddressTable` definiše podsistem za povezivanje agenata. Ona specificira osnovne metode za manipulaciju listom agenata sa kojima je agent u vezi (dodavanje, uklanjanje i pribavljanje adrese na osnovu ID-a). **`AddressTable`** čuva informacije o pojedinom agentu u objektima klase **`Address`**. **`BAddressTable`** klasa naslednica implementira listu agenata i sadrži identifikator, adresu, port, tip i opis. Ovo je potrebno kada se želi poslati poruka nekom agentu. U tom slučaju, svi potrebni podaci se dobijaju na osnovu ID-a agenta.

Apstraktna klasa **`ConnectionTable`** specificira listu ostvarenih konekcija među agentima. Konekcije su opisane objektima klase **`Connection`**. Klasa **`ConnectionTable`** definiše osnovne metode za manipulaciju listom konekcija (dodavanje, uklanjanje i provera konekcije). Klasa naslednica **`BConnectionTable`** implementira ovu listu.



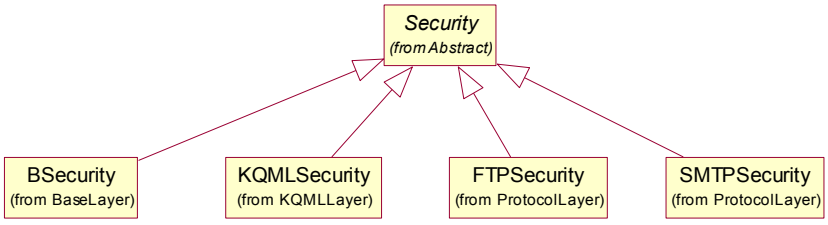
Slika 2.16: Dijagram klasa komunikacije u JAT Lite sistemu

Za uspostavljanje veze između agenata potrebne su dve programske niti: **`ServerThread`** i **`ReceiverThread`**. Prva nit čeka na otvaranje komunikacije sa drugim agentom, a druga biva kreirana kada ovaj agent otvara konekciju sa drugim agentom. Klase **`BRecvThread`** i **`BServerThread`** implementiraju komunikaciju na nivou TCP/IP protokola, a klase **`KQMLRecvThread`** i **`KQMLServerThread`** implementiraju komunikaciju na nivou KQML poruka.

2.5.4 Sigurnosni mehanizmi

Sigurnosni mehanizmi su podržani apstraktnom klasom **`Security`** (slika 2.17). Ova klasa sadrži sledeće metode koje implementiraju sigurnosne mehanizme:

1. **`processServerLogin()`** – ova metoda se poziva prilikom spajanja drugog agenta. Ona po protokolu šalje potvrdu konekcije i smešta tog agenta u tabelu adresa (**`AddressTable`**);
2. **`processClientLogin()`** – drugi kraj veze prilikom konektovanja sa drugim agentom. Prima potvrdu drugog agenta za konekciju i smešta njegovu adresu (adresu agenta na koga se spaja) u tabelu adresa (**`AddressTable`**);
3. **`processClientLogout()`** – ova metoda se poziva prilikom prekidanja veze sa drugim agentom. Ona po protokolu šalje poruku o raskidanju veze;
4. **`isValidAgent()`** – vraća *true* ako agent ima pravo da se konektuje.



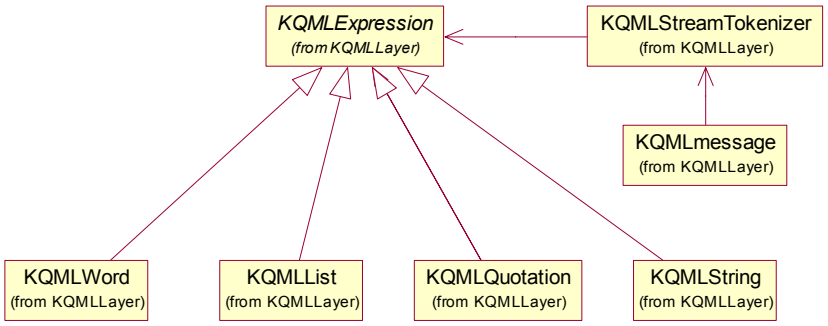
Slika 2.17: Dijagram klasa sigurnosnih mehanizama u JAT Lite sistemu

Bezbednosni sistem dozvoljava implementaciju proizvoljnog sigurnosnog algoritma. Za potrebe implementacije sigurnosnog mehanizama, potrebno je naslediti klasu **`Security`** i redefinisati nabrojane metode. U okviru standardne instalacije implementirane su klase koje redefinišu nabrojane metode na nekoliko nivoa protokola

(TCP/IP, KQML, FTP i SMTP). Konkretno, klasa **`BSecurity`** implementira elementaran protokol uspostavljanja veze, gde se šalje string "Connected" po uspešnom uspostavljanju veze. Klasa **`KQMLSecurity`** realizuje kompleksniji algoritam uspostave veze, gde se razmenjuju KQML poruke za autentifikaciju pozivajućeg agenta kod pozvanog agenta. Klase **`FTPSecurity`** i **`SMTPSecurity`** implementiraju sigurnosne mehanizme spajanja na SMTP i FTP server.

2.5.5 KQML poruke

KQML poruke su posebno podržane paketom **`KQMLLayer`**. Ovaj paket sadrži klase potrebne za rad sa KQML porukama. Slika 2.18 prikazuje dijagram klasa ovog paketa.



Slika 2.18: Dijagram klasa **`KQMLLayer`** paketa

U osnovi ovog paketa je klasa **`KQMLExpression`** koja predstavlja KQML izraz. Klase **`KQMLWord`**, **`KQMLList`**, **`KQMLQuotation`** i **`KQMLString`** predstavljaju elemente KQML izraza. Klasa **`KQMLStreamTokenizer`** služi kao parser KQML poruka, a klasa **`KQMLMessage`** predstavlja jednu KQML poruku koja sadrži KQML izraz.

2.5.6 Karakteristike JAT Lite agentskog okruženja

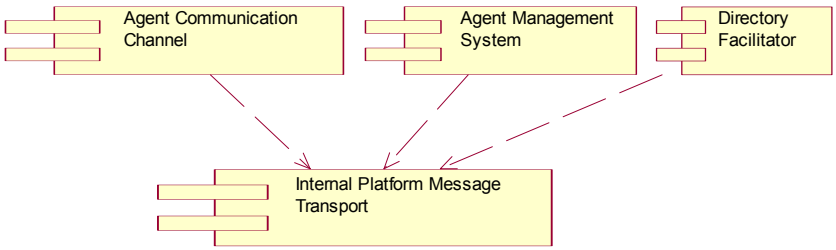
JAT Lite implementira osnovne elemente agentske tehnologije. Podržana je međuagentska komunikacija na nivou razmene poruka, a KQML poruke su posebno podržane. Dobra osobina podsistema za razmenu poruka je ta da se poruke šalju agentima, a ne host računaru, tako da svaki agent dobija samo one poruke koje su namenjene njemu. Sistem ne podržava mobilnost agenata. Nije podržan ni direktorijum agenata, kao ni mogućnost za međuagentsko uređenje odnosa. Sistem ne podržava FIPA specifikaciju.

2.6 JADE

JADE (*Java Agent DEvelopment Framework*) je agentsko okruženje razvijeno na Univerzitetu u Parmi [Bellifemine99]. Odlikuje se FIPA kompatibilnošću i implementacijom svih koncepata potrebnih za razvoj agentske tehnologije (razmena poruka, mobilnost, direktorijumi i sigurnost). Sistem je napisan na programskom jeziku Java i koristi RMI i CORBA tehnologije za implementaciju distribuiranih softverskih koncepata.

2.6.1 Arhitektura

JADE agentsko okruženje se sastoji iz sledećih podsistema, kako je prikazano na slici 2.19:

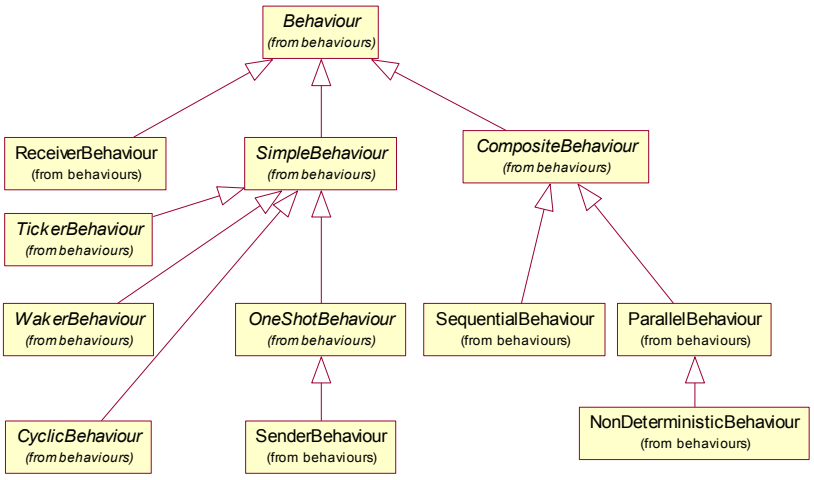


Slika 2.19: Dijagram komponenti JADE agentskog okruženja

Agent Communication Channel (ACC) je komponenta koja omogućuje komunikaciju između agentskih okruženja, kao i komunikaciju unutar agentskog okruženja. *Agent Management System* (AMS) je komponenta koja obavlja nadzornu ulogu, odnosno kontroliše pristup i upotrebu ostalih komponenti unutar okruženja. Ona obavlja autentifikaciju agenata i kontroliše njihovu registraciju. *Directory Facilitator* (DF) je komponenta koja služi za smeštaj i pretragu informacija o agentima. *Internal Platform Message Transport* (IPMT) je komponenta koja povezuje pojedinačne komponente ovog agentskog okruženja i omogućuje njihovu međusobnu komunikaciju.

2.6.2 Model izvršavanja agentskih zadataka

JADE agentsko okruženje koristi klase koje nasleđuju apstraktnu klasu **`Behaviour`** za realizaciju zadataka. Ova klasa definiše dve ključne metode koje će biti pozvane tokom izvršenja zadatka: **`action()`** i **`done()`**. Metoda **`action()`** se poziva da bi se zadatak izvršio, a metoda **`done()`** se poziva da bi se utvrdilo da li je izvršenje zadatka gotovo. Ako vrati *true*, zadatak je gotov. JADE agentsko okruženje sadrži unapred definisane klase naslednice, koje definišu širok spektar zadataka koji se mogu upotrebom ovog mehanizma realizovati, kao što je prikazano na slici 2.20:



Slika 2.20: Dijagram klasa naslednica apstraktne klase **`Behaviour`**

Klasa **`ReceiverBehaviour`** se može koristiti za prijem poruka. Ona po pozivu **`action()`** metode kopira poruku u atribut, tako da se može dobiti pozivom metode **`getMessage()`**.

Klasa **`SimpleBehaviour`** modeluje jednostavne zadatke koji se ne mogu prekidati. Ona je apstraktna klasa, a njene klase naslednice precizno definišu podvrste jednostavnih zadataka koji se mogu rešavati: **`OneShotBehaviour`**, **`CyclicBehaviour`**, **`TickerBehaviour`** i **`WakerBehaviour`**. Klasa **`OneShotBehaviour`** definiše zadatak koji se izvršava samo jednom. Klasa naslednica ove klase (**`SenderBehaviour`**) reprezentuje zadatak slanja poruke. **`CyclicBehaviour`** klasa definiše zadatke koji se izvršavaju ciklično, u beskonačnoj petlji. Klasa

TickerBehaviour definiše izvršenje koje se odvija periodično. Programer mora da implementira metodu **onTick()** u kojoj će se nalaziti kod za izvršenje. **WakerBehaviour** klasa definiše izvršenje definisano u zadatom vremenskom periodu. Programer mora da implementira metodu **handleElapsedTimeout()**, koja će biti pozvana po isteku zadatog vremena.

Klasa **CompositeBehaviour** definiše složena izvršavanja. Sadrži listu podređenih **Behaviour** klasa, koje će biti izvršavane po politici zadatoj implementiranjem metoda: **scheduleFirst()** i **scheduleNext()**. Programer mora da implementira i metodu **checkTermination()** koja će biti automatski pozvana po izvršenju svakog podređenog objekta da bi se utvrdilo da li treba prekinuti ukupno izvršenje. Postoje dve klase naslednice ove klase: **ParallelBehaviour** i **SequentialBehaviour**. Klasa **ParallelBehaviour** izvršava paralelno podređene zadatke, a klasa **SequentialBehaviour** izvršava zadatke jedan po jedan. Klasa **NonDeterministicBehaviour** je zastarela klasa i ne koristi se više.

2.6.3 Razmena poruka

Osnovu razmene poruka čini slanje objekata klase **ACLMessage**. Ova klasa reprezentuje poruku po FIPA specifikaciji. Ona sadrži asocijativnu listu parova (ključ, vrednost), koji definišu poruku. JADE razlikuje tri tipa komunikacije:

1. komunikacija unutar JADE okruženja,
2. komunikacija između dva JADE okruženja i
3. komunikacija između JADE i ne-JADE okruženja.

U prvom slučaju, sva komunikacija se svodi na prenos objekta poruke direktno do primaoca, bez enkapsulacije poruke u transportni oblik. Komunikacija između dva JADE okruženja se svodi na upotrebu RMI mehanizma, gde se poruka serijalizuje i prenosi kao parametar udaljenoj metodi na prijemnoj strani. U slučaju komunikacije između JADE i nekog drugog agentskog okruženja, koristi se IIOP protokol, a poruka se transformiše iz objekta **ACLMessage** klase u objekat klase **java.lang.String**, a zatim iz ovog oblika u IIOP bajt tok (*IIOP byte stream*). Na prijemnoj strani se takođe odvija dvostruka transformacija.

JADE agentsko okruženje omogućuje jednostavno filtriranje pristiglih poruka. Ovo filtriranje se realizuje upotrebom klase **MessageTemplate**. Ova klasa definiše metodu **match()**, čijom redefinicijom se može

implementirati proizvoljno filtriranje pristiglih poruka. Osim ove metode, postoje još i metode **`and()`**, **`or()`** i **`not()`**, koje obezbeđuju logičke operacije kojima se omogućuje pravljenje složenih pravila filtriranja.

2.6.4 Mobilnost

JADE agentsko okruženje podržava mobilnost agenata. Mobilnost je implementirana u osnovnoj agentskoj klasi – klasi Agent u dvema metodama: **`doMove()`** i **`doClone()`**. Prva inicira prenos agenta na odredišnu adresu, a druga pravi kopiju agenta na odredišnoj adresi. Za reakciju na prenos i kopiranje agenta zadužene su sledeće četiri metode:

- **`beforeMove()`** – poziva se pre početka prenosa agenta,
- **`afterMove()`** – poziva se nakon prenosa na odredišnu adresu,
- **`beforeClone()`** – poziva se pre pravljenja kopije agenta i
- **`afterClone()`** – poziva se nakon pravljenja kopije agenta.

Mobilnost agenta je realizovana u klasama koje implementiraju **`MobilityManager`** interfejs. U JADE okruženju postoje dve klase koji implementiraju ovaj interfejs: **`DummyMobilityManager`** i **`RealMobilityManager`**. Prva klasa se koristi ako se mobilnost agenata ne želi podržati, a druga u potpunosti omogućuje mobilnost agenata.

2.6.5 Sigurnost

JADE poseduje sigurnosni podsistem [Poggi01], koji omogućuje sledeće bezbednosne mehanizme: autentifikaciju, autorizaciju i kriptovanje komunikacije.

Autentifikacija se zasniva na prijavljivanju korisnika uz pomoć korisničkog imena i šifre (koji se čuvaju u *JADE password* datoteci). Agenti se identifikuju sertifikatima koji sadrže njihovo ime i ime vlasnika agenta.

Autorizacija je definisana ulogama koje kontrolišu pristup resursima ili akcijama. Autorizacija se zasniva na listi dozvoljenih uloga, sadržanih u datoteci *PolicyFile*. Uloge obuhvataju: kreiranje i deaktiviranje agenata, suspendovanje i nastavljanje rada agenata, kopiranje i prenošenje na druge lokacije, gašenje celog agentskog okruženja (*shut down*), kontakt sa CA zbog rada sa sertifikatima i registracija/deregistracija agenata.

Kriptovanje komunikacije se uključuje odgovarajućim parametrom u konfiguracionoj datoteci. Na taj način, umesto obične tekst-orijentisane komunikacije, koristi se SSL/TLS komunikacija.

Rukovanje sertifikatima je implementirano tako da se jednom agentskom okruženju poveri uloga CA (*Certificate Authority*) i sve operacije sa sertifikatima se obavljaju preko njega. To znači da ovi sertifikati važe samo unutar mreže JADE agentskih okruženja.

2.6.6 Karakteristike JADE agentskog okruženja

JADE predstavlja agentsko okruženje koje je u najvećoj meri implementiralo sve potrebne koncepte prema FIPA specifikaciji. Implementirana je efikasna razmena poruka, mobilnost agenata, direktorijumi i sigurnosni mehanizmi. Za komunikaciju se, na transportnom nivou, koriste RMI, odnosno IIOP, čime je omogućena ne samo komunikacija između JADE agentskih okruženja, već i komunikacija sa drugim agentskim okruženjima (pod uslovom da poštuju FIPA specifikaciju). Mana ovog okruženja je što nije orijentisano ka J2EE tehnologiji (što je pokušano da se ispravi u projektu BlueJADE [Cowan02], ali se to svelo na smeštanje celog JADE agentskog okruženja na J2EE aplikacioni server kao servis).

2.7. Uporedne karakteristike analiziranih agentskih okruženja

Tabela 2.4 sadrži uporedne karakteristike postojećih agentskih okruženja. Agentska okruženja su poređena po pet karakteristika: mobilnost agenata, razmena poruka, direktorijum agenata, direktorijum servisa i sigurnosni mehanizmi. Takođe, razmatrana je upotreba specijalizovanih tehnologija kao što su CORBA ili EJB. Od svih analiziranih agentskih okruženja, jedino Voyager i JADE koriste CORBA tehnologiju.

Ni jedno od razmatranih agentskih okruženja ne podržava koncept međuagentskog uređenja odnosa. Bez ovog koncepta, sva agentska okruženja su ravnopravna i nalaze se u jedinstvenom prostoru. Ni jedno okruženja nije svesno svog okruženja i sva međusobna komunikacija se izvodi ručno, u smislu da programer mora zna lokacije svih agentskih servisa od interesa.

Agentsko okruženje	Mobilnost agenata	Razmena poruka	Direktorijum agenata	Direktorijum servisa	Sigurnosni mehanizmi
JAF	ne	da, implementirana bez razrađenog sistema za identifikaciju primaoca poruka i bez upotrebe specijalizovanih servisa	ne	ne	ne
Able	da, implementirana upotrebom RMI	da, implementirana upotrebom RMI	da, implementiran upotrebom RMI	da, samo je specificirana stavka servisa; servis nije implementiran	da, implementirano u `AbleSecuritySupport` klasi
Voyager	da, implementirana upotrebom CORBA, RMI i COM	ne	ne	ne	da, implementirano upotrebom SSL i HTTPS protokola
Agleti	da, implementirana upotrebom RMI	da, implementirana bez upotrebe specijalizovanih servisa	ne	ne	da, implementirano u `AgletsSecurity` klasi
JAT	ne	da, implementirana bez upotrebe specijalizovanih servisa	ne	ne	ne
JADE	da, implementirana upotrebom RMI i CORBA	da, implementirana upotrebom RMI i CORBA	da, podaci se čuvaju u asocijativnim mapama	da, samo je specificirana stavka servisa; servis nije implementiran	da, implementirana autentifikacija, autorizacija, kriptovanje komunikacije i sertifikati.

Tabela 2.4: Uporedne karakteristike agentskih okruženja

Poglavlje 3

Java tehnologija i agentsko okruženje

Za rad agentskog okruženja mogu se koristiti tehnologije dostupne po J2EE specifikaciji. Koncepti koji se žele ostvariti su asinhrona razmena poruka, mobilnost agenata, direktorijumi i sigurnosni mehanizmi, a tehnologije potrebne za realizaciju su JMS sistem poruka, serijalizacija, JNDI tehnologija i sigurnosni mehanizmi ugrađeni u J2EE aplikacione servere.

3.1 Asinhrona razmena poruka

Asinhrona razmena poruka je osnovni cilj koji mora da realizuje svaki sistem zasnovan na agentima. Učesnici u komunikaciji ne smeju biti blokirani dok komunikacija traje. U J2EE tehnologiji postoji sistem koji ovo omogućuje. To je JMS (*Java Messaging Service*). JMS je skup Java interfejsa koji omogućavaju komunikaciju sa proizvoljnim sistemom za razmenu poruka (*Messaging System*). JMS podržava dva načina rada:

1. *Peer-to-peer* komunikacija i
2. *publish/subscribe* komunikacija.

Prvi način rada definiše komunikaciju između dva ravnopravna entiteta upotrebom reda (*queue*) za razmenu poruka, dok drugi način omogućuje slanje poruka širokom spektru klijenata koji se "pretplate" (*subscribe*) na određene teme.

Ova tehnologija se može upotrebiti za razmenu poruka zato što omogućava neblokirajuću razmenu proizvoljnih poruka. Pošto su KQML poruke standard za međuagentsku komunikaciju, potrebno je integrisati ove poruke u JMS sistem. JMS sistem će zapravo predstavljati dodatni sloj koji će vršiti transportnu ulogu, a učesnici u komunikaciji će iz JMS poruka izdvajati KQML poruke.

3.2 Mobilnost agenata

Mobilnost agenata predstavlja mogućnost da agent kopira sebe i nastavi izvršenje na drugom računaru. Ova mogućnost je direktno uslovljena mrežnom infrastrukturom agentskog okruženja. EJB okruženje direktno omogućuje prenošenje objekata preko mreže, čime se omogućuje i

prenos programa i podataka. Ovim je koncept mobilnih agenata u potpunosti podržan.

3.3 Direktorijum agenata i servisa

Direktorijum agenata i servisa je u J2EE tehnologiji podržan preko JNDI sistema. Ovaj sistem omogućava mapiranje imena na objekte. Zamišljen je kao biblioteka opšteg tipa koja omogućava pristup različitim servisima za imenovanje kroz identičan interfejs. Kao elemente standardne biblioteke, isporučuju se odgovarajući moduli za pristup sledećim servisima:

- CORBA *Common Object Services (COS) Name service*,
- RMI registry i
- LDAP (*Lightweight Directory Access Protocol*).

Ovako zamišljen JNDI sistem se može iskoristiti za agentsku tehnologiju, i to na primer, u oblasti direktorijuma. Primer upotrebe JNDI servisa za direktorijum servisa je data u listingu 3.1.

```
/**
 * JNDIServiceManager smešta servise u JNDI stablo.
 * Zapravo čuva uz svaki servis i njegov identifikator, na
 * osnovu koga ga vraća korisniku.
 */
public class JNDIServiceManager implements ServiceManager {
  private Context ctx;
  private EJBObject facilitator;
  /** JNDI putanja gde se smeštaju servisi. */
  private String path = "java:comp/agent/services/";

  public JNDIServiceManager() {
    try {
      this.ctx = new InitialContext();
    } catch (NamingException ex) {
      ex.printStackTrace();
    }
  }

  /** Inicijalizuje ServiceManager. */
  public void init(EJBObject facilitator) {
    this.facilitator = facilitator;
  }
```

Listing 3.1: Direktorijum servisa implementiran JNDI sistemom

```
/** Vraća servis na osnovu id-a.
 *  @param id Identifikator servisa.
 */
public Service getService(Object serviceId) {
  Service retVal = null;
  try {
    Object serviceObject = ctx.lookup(path+serviceId);
    retVal = ((ServiceHolder)PortableRemoteObject.narrow(
       serviceObject, ServiceHolder.class)).getService();
  } catch(Exception e) {
    e.printStackTrace();
  }
  return retVal;
}

/** Vraća servis u direktorijum. */
public void returnService(Object serviceId) {
  try {
    Object serviceHolderObject =
      ctx.lookup(path+serviceId);
    ServiceHolder serviceHolder =
      (ServiceHolder)PortableRemoteObject.narrow(
                  serviceHolderObject,
                  ServiceHolder.class);
    serviceHolder.remove();
    // Uklonimo serviceHolder na osnovu uniqueId-a.
    ctx.unbind(path+serviceId);
  } catch (Exception ex) {
    ex.printStackTrace();
  }
}

/** Traži zadati servis u direktorijumu. */
public Object findService(String serviceName)
  throws RemoteException {
  Object retVal = null;
  // XML parser će vratiti ime klase servisa.
  String serviceClassName =
    XMLAgentParser.getServiceClassName(
      serviceName);
  if (serviceClassName != null) {
    ServiceHolder serviceHolder = null;
    try {
      InitialContext context = new InitialContext();
        Object homeObject = context.lookup(
          "java:comp/env/ServiceHolder");
```

Listing 3.1: Direktorijum servisa implementiran JNDI sistemom – nastavak

```
          ServiceHolderHome home =
          (ServiceHolderHome)PortableRemoteObject.narrow(
             homeObject, ServiceHolderHome.class);
           serviceHolder =
(ServiceHolder)PortableRemoteObject.narrow(
             home.create(serviceClassName),
ServiceHolder.class);
      } catch(RemoteException e) {
        throw new RemoteException(
          "EJBServiceManager fatal error: " +
          e.getMessage());
      } catch(NamingException e) {
        throw new RemoteException(
          "EJBServiceManager fatal error: " +
          e.getMessage());
      } catch(CreateException e) {
        e.printStackTrace();
        throw new RemoteException(
          "EJBServiceManager fatal error: " +
           e.getMessage());
      }
      // Ubacimo servis u stablo.
      String serviceUniqueId = null;
      try {
        serviceUniqueId =
((Facilitator)facilitator).getUniqueId();
        retVal = serviceUniqueId;
        ctx.rebind(path+retVal, serviceHolder);
      } catch (Exception ex) {
        ex.printStackTrace();
      }
    }
    return retVal;
  }

  /** Lista sve servise. */
  public Enumeration listServices() {
    try {
      return ctx.listBindings(path);
    } catch (Exception ex) {
      ex.printStackTrace();
      return null;
    }
  }
}
```

Listing 3.1: Direktorijum servisa implementiran JNDI sistemom - nastavak

Listing 3.1 ilustruje upotrebu JNDI sistema u direktorijumskom sistemu servisa. Pojedinačni servisi se povezuju sa svojim nazivom, tj. servis se smešta u repozitorijum, a identifikuje se svojim nazivom. Metoda `getService()` vraća servis na osnovu njegovog identifikatora. Servis se nalazi u JNDI stablu, a identifikator jedinstveno određuje njegovu lokaciju (time što je deo putanje u stablu). Metoda `returnService()` uklanja servis tako što ga briše iz stabla. Metoda `findService()` pronalazi servis na osnovu simboličkog imena. Metoda `listServices()` lista sve servise tako što lista sva podstabla dela JNDI stabla zaduženog za servise.

Identičan pristup se može usvojiti i za implementaciju direktorijuma agenata. Jedina razlika je što se u JNDI stablo ne stavljaju servisi, već agenti.

3.4 Sigurnosni mehanizmi

Većina postojećih aplikacionih servera poseduje nekakav oblik sigurnosnih mehanizama. Osnovni mehanizam predstavlja autentifikaciju upotrebe komponenti, što omogućuje kontrolu pristupa i izvršavanja komponenti na serveru.

Osim kontrole pristupa komponentama, većina današnjih aplikacionih servera podržava kriptografsku zaštitu. Konkretno, kod aplikacionog servera WebSphere [WebSphere] se bezbednosni parametri podešavaju u `sas.server.props` konfiguracionoj datoteci:

```
com.ibm.CORBA.loginUserid=<userid>
com.ibm.CORBA.principalName=<DOMAIN/userid>
com.ibm.CORBA.loginPassword=<password>
com.ibm.CORBA.securityEnabled=true
com.ibm.CORBA.authenticationTarget=<localos/ltpa>
```

Prva četiri parametra definišu zaštitu pristupa komponentama aplikacionog servera na nivou autentifikacije korisnika, dok peti parametar definiše kriptografsku zaštitu komunikacije.

Ako je vrednost `com.ibm.CORBA.authenticationTarget` parametra `localos`, tada se sledeći parametri podešavaju:

```
com.ibm.CORBA.SSLTypeIClientAssociationEnabled=true
com.ibm.CORBA.LocalOSClientAssociationEnabled=true
com.ibm.CORBA.LTPAClientAssociationEnabled=false
```

Ako je vrednost **com.ibm.CORBA.authenticationTarget** parametra **ltpa**, tada se sledeći parametri podešavaju:

```
com.ibm.CORBA.SSLTypeIClientAssociationEnabled=true
com.ibm.CORBA.LocalOSClientAssociationEnabled=false
com.ibm.CORBA.LTPAClientAssociationEnabled=true
```

Navedeni parametri definišu upotrebu kriptovanih konekcija između klijenata i servera, kao i između svih komponenti koje komuniciraju izvan servera. Na ovaj način, omogućena je kriptografska zaštita komunikacije, a implementirana je u samom aplikacionom serveru.

3.4.1 Implementacija kriptografske zaštite upotrebom JCE

Ako aplikacioni server ne podržava kriptografsku zaštitu, ona se može implementirati unutar agentskog okruženja. Jedan primer kriptografske podrške podrazumeva implementaciju metoda za kriptovanje i dekriptovanje poruka upotrebom JCE (*Java Cryptography Extension*) biblioteke, koja je deo standardne implementacije J2EE. Ovaj sistem se bazira na asimetričnom metodu kriptovanja upotrebom javnog i privatnog ključa, gde se razmena javnih ključeva odvija preko sertifikata.

Kriptografska zaštita komunikacije u agentskom okruženju se može implementirati upotrebom JCE biblioteke na sledeći način:

Inicijalizacija sistema zaštite počinje kreiranjem javnog i privatnog ključa lokalnog agentskog okruženja.:

```
AlgorithmParameterGenerator paramGen =

AlgorithmParameterGenerator.getInstance("DH");
paramGen.init(512);
AlgorithmParameters params = paramGen.generateParameters();
DHParameterSpec dhParamSpec =
(DHParameterSpec)params.getParameterSpec(DHParameterSpec.cl
ass);
KeyPairGenerator keyGen =
KeyPairGenerator.getInstance("DH");
keyGen.initialize(dhParamSpec);
KeyPair pair = keyGen.generateKeyPair();
pubKey = pair.getPublic();
privKey = pair.getPrivate();
```

Nakon kreiranja javnog i privatnog ključa, lokalno agentsko okruženje mora da pošalje svoj javni ključ udaljenom agentskom okruženju. Za to će kreirati odgovarajući sertifikat u koji će smestiti javni ključ. Za tu svrhu će upotrebiti početni sertifikat izdat od strane CA (*Certificate Authority*). Ovaj sertifikat će služiti za autentifikaciju svih sertifikata kreiranih od strane lokalnog okruženja. Početni sertifikat se nalazi u datoteci **.keystore**:

```
Security.addProvider(new BouncyCastleProvider());
FileInputStream fis = new FileInputStream(".keystore");
BufferedInputStream bis = new BufferedInputStream(fis);
KeyStore ks = KeyStore.getInstance("JKS");
// pripremimo šifru za pristup tajnom ključu
char[] c = {'t','e','s','t','1','2','3'};
// učitamo sertifikat iz datoteke u memoriju
ks.load(fis, c);
// izdvojimo sertifikat u objekat klase Certificate
java.security.cert.Certificate cs =
ks.getCertificate("myalias");
// izdvojimo privatni ključ upotrebom šifre
PrivateKey privKey = (PrivateKey)ks.getKey("myalias", c);
```

Sertifikat iz datoteke **.keystore** sadrži i privatni ključ kojim je potpisan sertifikat. Do ovog ključa se može doći samo ako se zna šifra za pristup. Ovaj ključ će se iskoristiti za kreiranje novog sertifikata – X.509 sertifikata koji će sadržati javni ključ lokalnog okruženja, i koji će se poslati udaljenom okruženju:

```
// pripremimo identifikacione parametre sertifikata
Hashtable  attrs = new Hashtable();
attrs.put(X509Principal.C, "YU");
attrs.put(X509Principal.O, "FTN");
attrs.put(X509Principal.L, "Novi Sad");
attrs.put(X509Principal.ST, "Yugoslavia");
attrs.put(X509Principal.CN, "XJAF");
// kreiramo generator X.509 sertifikata
X509V1CertificateGenerator certGen = new
X509V1CertificateGenerator();
// verzija
certGen.setSerialNumber(BigInteger.valueOf(1));
// izdavač sertifikata (čiji privatni ključ se koristi za
potpis)
certGen.setIssuerDN(new X509Principal(attrs));
// datum pocetka validnosti
certGen.setNotBefore(new Date(System.currentTimeMillis()-
10));
// datum kraja validnosti
```

```
certGen.setNotAfter(new Date(System.currentTimeMillis() +
10001 * 60 * 60 * 24 * 365));
// subjekat (čiji javni ključ se šalje ovim sertifikatom)
attrs.put(X509Principal.CN, getName());
certGen.setSubjectDN(new X509Principal(attrs));
// ubacimo javni kljuc
certGen.setPublicKey(this.pubKey);
certGen.setSignatureAlgorithm("SHA1withDSA");
// i potpišemo privatnim ključem iz sertifikata
X509Certificate cert =
certGen.generateX509Certificate(privKey);
```

Udaljeno okruženje prima lanac sertifikata koji sadrži sertifikat sa javnim ključem lokalnog okruženja, i osnovni sertifikat koji može poslužiti za utvrđivanje autentičnosti. Po prijemu ovih sertifikata, izdvaja se javni ključ lokalnog agentskog centra i on se koristi za kreiranje para javni-privatni ključ, kao i za kreiranje tajnog ključa. Javni ključ se zatim ugrađuje u novi sertifikat i šalje nazad podređenom agentskom centru, koji ga izdvaja i koristi za generisanje svog tajnog ključa (kao atribut `otherPartyPubKey`):

```
KeyAgreement keyAgree = KeyAgreement.getInstance("DH");
keyAgree.init(privKey);
keyAgree.doPhase(otherPartyPubKey, true);
secKey = keyAgree.generateSecret("DES");
byte[] raw = secKey.getEncoded();
SecretKeySpec sKeySpec = new SecretKeySpec(raw, "DES");
decoder = Cipher.getInstance("DES");
decoder.init(Cipher.DECRYPT_MODE, sKeySpec);
encoder = Cipher.getInstance("DES");
encoder.init(Cipher.ENCRYPT_MODE, sKeySpec);
```

Nakon inicijalizacije, obe strane kriptuju i dekriptuju poruke pozivom metode `doFinal()` klase `Cifer`, koja obavlja poslove kriptovanja i dekriptovanja, u zavisnosti od moda inicijalizacije (`Cipher.DECRYPT_MODE ili Cipher.ENCRYPT_MODE`).

3.5 J2EE Aplikacioni serveri

J2EE aplikacioni serveri predstavljaju programske sisteme koji omogućuju upotrebu svih elemenata J2EE tehnologije. Aplikacioni serveri pre svega omogućuju upotrebu distribuiranih komponenti u obliku EJB komponenti. Tipičan razvoj aplikacija uz upotrebu J2EE aplikacionih servera se svodi na definisanje softverskih komponenti koje će biti korišćene unutar aplikacionog servera, njihovo kreiranje i

isporuku serveru. EJB komponenta se koristi tako što se identifikuje unutar J2EE servera (upotrebom JNDI podsistema), nakon čega se dobija lokalni reprezent te klase (*stub* klasa), a po završetku rada se komponenta vraća sistemu. Upravljanje životnim ciklusom komponente, raspodela opterećenja i ostali implementacioni detalji sakriveni su od programera i o njima se brine sam aplikacioni server.

Analizirana su i isprobana dva aplikaciona servera:

- Orion aplikacioni server [Orion] i
- JBoss aplikacioni server [JBoss].

Aplikacioni server Orion je proizvod IronFlare kompanije i besplatan je za razvoj. Proizvod se isporučuje u vidu zip arhive koju je potrebno samo raspakovati. Ovaj server podržava J2EE specifikaciju i podržava naprednije tehnike kao što su raspodela opterećenja, XML/XSLT procesiranje i automatska isporuka aplikacija (auto-deployment).

Aplikacioni server JBoss je aplikacioni server kompanije JBOSS GROUP LLC. Proizvod se isporučuje u vidu zip arhive koju je potrebno samo raspakovati. Ovaj aplikacioni server je razvijen kao open-source projekat. Server u potpunosti podržava J2EE specifikaciju i nudi dodatne mogućnosti kao što su microkernel (omogućuje razvoj aplikacija bez restartovanja servera) i Aspect Oriented Framework.

Aplikacioni server Orion se pokazao kao jednostavniji za razvoj, zato što nije potrebno projekat isporučiti serveru (*deployment*), već je prilikom ponovnog kompajliranja klasa dovoljno restartovati server (što se odvija veoma brzo). Za razliku od njega, kod JBoss aplikacionog servera potrebno je projekat nakon ponovnog kompajliranja isporučiti, ali zato nije potrebno restartovanje servera (koje desetak puta sporije od restartovanja Orion aplikacionog servera).

Poglavlje 4

Specifikacija proširivog agentskog okruženja

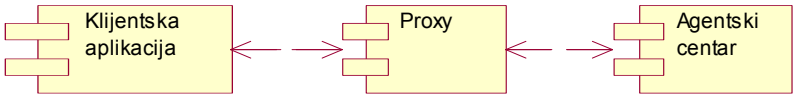
Pregled FIPA specifikacije dat je u poglavlju 1, a u poglavlju 2 opisano je nekoliko reprezentativnih agentskih okruženja. Iz ova dva poglavlja se vidi da je prilikom razvoja agentskog okruženja neophodno obezbediti sledeće mogućnosti:

1. mogućnost izvršenja specijalizovanih agentskih komponenti, što povlači jasnu definiciju agentskih interfejsa, odnosno klasa;
2. mogućnost međuagentske komunikacije, kako na lokalnom nivou (unutar jednog agentskog centra), tako i na globalnom nivou (između različitih agentskih centara);
3. mogućnost upotrebe direktorijumskog sistema agenata i servisa, čime se mogu odabirati agenti i servisi za rešavanje problema;
4. mogućnost obrade zadataka;
5. mogućnost definisanja sigurnosnih mehanizama kojima bi se sprečila zloupotreba agentskog okruženja i pripadajućih agenata.

Na osnovu navedenih mogućnosti modelovano je agentsko okruženje koje se zasniva na tehnologiji distribuiranih komponenti, ali ne zavisi od konkretne implementacije. Model agentskog okruženja prikazan u ovom poglavlju obuhvata osnovne elemente agentskog okruženja, ali i definiše dodatne mogućnosti, kao što su mobilni zadaci i međusobno uređenje odnosa između agentskih centara.

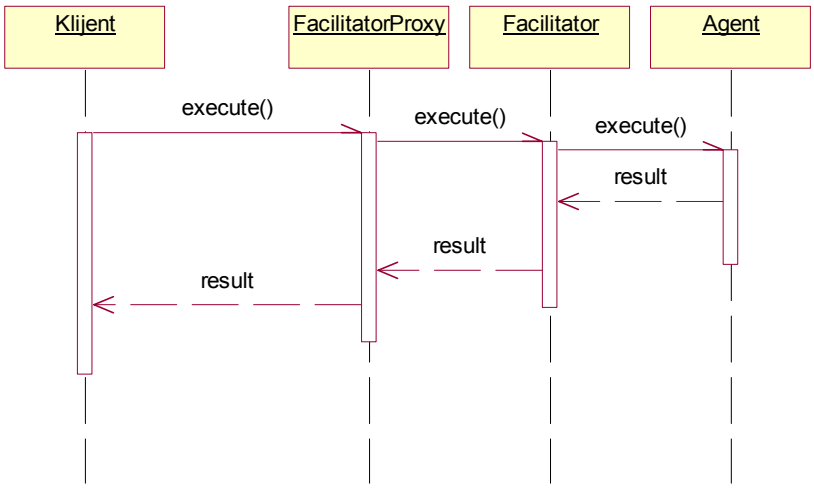
4.1 Globalna arhitektura agentskog okruženja

Sistem se sastoji od klijenata i agentskih centara. Klijenti se obraćaju agentskim centrima za rešavanje zadatka. Zadatak rešavaju agenti koje angažuje agentski centar. Slika 4.1 ilustruje globalnu arhitekturu sistema.



Slika 4.1: Dijagram komponenti sistema

Klijent zadaje zadatak agentskom centru, agentski centar angažuje agenta za izvršenje zadatka i vraća klijentu rezultat rada. *FacilitatorProxy* komponenta obezbeđuje pristup klijentskoj aplikaciji do agentskog centra i sakriva od klijenta sve potrebne tehnike za rad sa agentima. Sa stanovišta klijenta, njemu je potrebno samo da kreira *FacilitatorProxy* komponentu i da joj prosledi zadatak – sve ostale pojedinosti odrađuje *FacilitatorProxy* komponenta. Slika 4.2 ilustruje ciklus realizacije zadatka.



Slika 4.2: Dijagram sekvenci za izvršenje zadatka

Izvršenje zadatka se realizuje na dva načina: programski ili slanjem KQML poruke agentu. Izvršavanje zadatka je asinhrono, odnosno ne blokira se izvršenje klijentskog koda dok se ne realizuje zadatak, već po realizaciji informiše klijentsku aplikaciju o rezultatu. Na ovaj način, klijentska aplikacija može da nastavi da izvršava kod, a o rezultatima rada biće upoznata kasnije. Angažovani agent rešava zadatak unutar matičnog agentskog okruženja, ili prelazi u drugo agentsko okruženje, u zavisnosti od prirode zadatka.

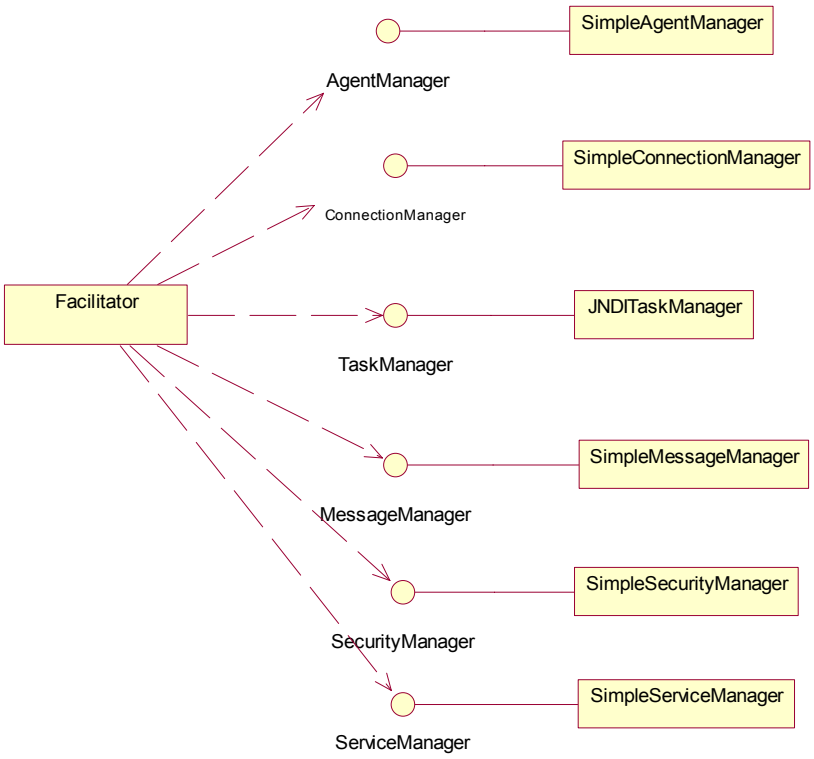
Klijentska aplikacija i *FacilitatorProxy* komponenta se izvršavaju na klijentskoj virtuelnoj mašini. Agentski centar (*Facilitator* komponenta), agent i sve ostale klase se izvršavaju u virtuelnoj mašini aplikacionog servera. To znači da su agentski centar i agenti angažovani kao serverske komponente, pa se o upravljanju životnim ciklusom ovih komponenti, raspodeli opterećenja i ostalim implementacionim detaljima (*session fail-over*, *clustering*, itd.) stara aplikacioni server.

4.1.1 Agentski centar

Agentski centar (*Facilitator* komponenta) je centralna komponenta ovog agentskog okruženja. Zadatak agentskog centra je upravljanje agentskim sistemom. Ovo obuhvata sledeće poslove:

1. obezbeđenje radnog okruženja agentima, posredstvom odgovarajućeg API-ja;
2. manipulacija zadacima koje treba rešiti;
3. regrutacija odgovarajućeg agenta za zadati posao;
4. rukovanje servisima dostupnim agentima i agentskom centru;
5. obezbeđenje međuagentske komunikacije i
6. obezbeđenje komunikacije među agentskim okruženjima.

Navedene poslove može da obavlja sama komponenta, ali je bolje da se taj posao raspodeli na odgovarajuće podsisteme. Ti podsistemi se nazivaju menadžeri. Menadžeri su instance klasa koje implementiraju odgovarajući menadžerski interfejs. Ovakav pristup pruža mogućnost da se menadžeri mogu zameniti pogodnijim, odnosno, da se mogu implementirati različiti algoritmi za konkretne menadžere. To znači da se menadžeri mogu birati prilikom konfigurisanja sistema. Ovakav koncept se zove koncept plaginova (*plug-in*), tj. modula u prevedenom obliku, koji se nezavisno instaliraju. Ovo takođe znači da zadati menadžeri nisu neophodni prilikom kompajliranja (*compile-time*), već je samo neophodno njihovo postojanje prilikom inicijalizacije sistema. Slika 4.3 prikazuje koncepciju *plug-in*-ova u ovom agentskom okruženju.



Slika 4.3: Funkcionalnost pojedinih delova je poverena menadžerima

Za rad sa agentima nadležan je **`AgentManager`** interfejs. Za rad sa zadacima potreban je **`TaskManager`** interfejs. Za komunikaciju je zadužen **`MessageManager`**, a za međuagentsko povezivanje i međusobno uređenje **`ConnectionManager`** interfejs. Za bezbednosne aspekte odgovoran je **`SecurityManager`** interfejs, a za manipulaciju servisima **`ServiceManager`**.

Klase koje implementiraju spomenute interfejse zapravo implementiraju odgovarajuće algoritme za pojedine funkcije. Menadžeri imaju referencu na *Facilitator* komponentu, što omogućuje komunikaciju sa drugim menadžerima, samom *Facilitator* komponentom, kao i agentima. Sistem je koncipiran tako da se može prilikom konfigurisanja odabrati proizvoljan menadžer pod uslovom da implementira zadati interfejs.

Osnovna namena *Facilitator* komponente je da izvršava zadate zadatke. Zadaci se poveravaju *TaskManager* komponenti, odgovarajući agent se regrutuje uz pomoć *AgentManager* komponente, a poruke mu se šalju uz pomoć *MessageManager* i *CommunicationManager* komponente. Za bezbednost izvršavanja brine se *SecurityManager* komponenta, a za rukovanje servisima *ServiceManager* komponenta.

Metode *Facilitator* komponente su nabrojane u tabeli 4.1.

Naziv	**Opis**
`String getUniqueId()`	Vraća jedinstven broj.
`void execute(Object taskId)`	Izvršava agenta identifikovanog po njegovom ID-u.
`AgentResult execute(Object agentId, AgentTask task)`	Izvršava agenta identifikovanog po njegovom ID-u, ali agentu prosleđuje zadati zadatak.
`void executeUsingKQML(Object taskId)`	Izvršava agenta tako što mu šalje KQML poruku sa zadatkom u *content* polju KQML poruke.
`void moveTo(String address, Object agentId, AgentTask task)`	Prebacuje zadatog agenta na zadatu adresu, sa zadatkom koji tamo treba da izvrši.
`Object acceptAgent(Agent agent, AgentTask task, String senderAddr)`	Prihvata prosleđenog agenta i vraća njegov novi ID.

Tabela 4.1: Spisak metoda koje mora da implementira *Facilitator* komponenta

Metode `moveTo()` i `acceptAgent()` se koriste kod prenosa agenata.

Ovakva arhitektura je u potpunosti u saglasnosti sa FIPA specifikacijom. Predloženi menadžeri sadrže svu funkcionalnost predloženu FIPA standardom. Direktorijum agenata i servisa je pokriven menadžerima agenata i servisa, razmena poruka *MessageManager* komponentom, a sigurnost *SecurityManager* komponentom.

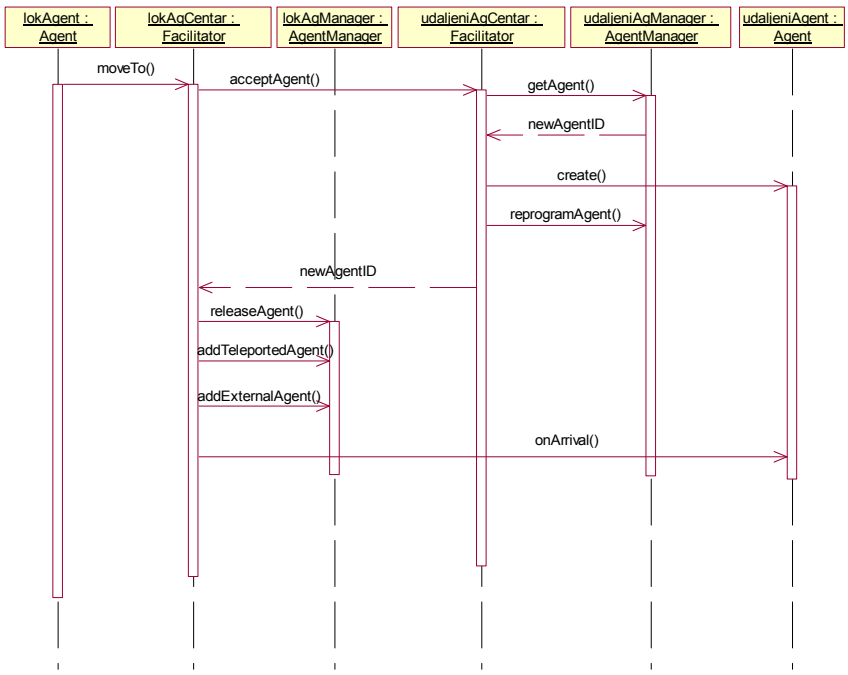
4.1.2 Mobilnost agenata

Mobilni agenti mogu da se premeste iz jednog agentskog okruženja u drugo i da tamo nastave svoj rad. Proces prebacivanja agenata podrazumeva prebacivanje podataka i programa koji agent izvršava. Koncept serijalizacije omogućuje upravo to. Kada je potrebno prebaciti agenta u odredišno agentsko okruženje, dovoljno je serijalizovati agenta i prebaciti ga na odredište.

Po prelasku na odredište, potrebno je obezbediti da agent može da nastavi sa radom. Agentsko okruženje po prijemu agenta obavlja deserijalizaciju i poziva metodu **`onArrival()`** u klasi agenta, čime se agentu signalizira da je uspešno završen prenos i da može da nastavi sa izvršenjem zadatka.

Mobilnost agenata podrazumeva da agent, po prispeću na odredište nastavlja započeti rad. Postavlja se pitanje šta uraditi sa njegovim originalom, u polaznom agentskom okruženju? Usvojen koncept da se originalni agent deaktivira, ali se čuvaju sve potrebne reference na njega, tako da se sva komunikacija preusmeri na novonastalog agenta u odredišnom agentskom okruženju.

Slika 4.4 prikazuje dijagram sekvence za prenos agenta.



Slika 4.4: Prenos agenata u drugi agentski centar

Prenos agenta počinje pozivom metode **`moveTo()`** agentskog centra. Lokalni agentski centar kontaktira udaljeni agentski centar i poziva njegovu metodu **`acceptAgent()`**.

Ova metoda regrutuje jednog agenta za prosleđeni zadatak i vraća njegov ID. Osim ovoga, vrši se prenošenje konteksta lokalnog agenta u udaljeni. Za to se koristi metoda **`reprogramAgent()`** *AgentManager* komponente.

Ova metoda prenosi kontekst prosleđenog agenta u novoodabranog udaljenog agenta. To je izvodljivo zbog serijalizacije agenta, kojom se kompletan agent prenosi sa svim atributima i metodama. Preneseni agent se preslikava preko udaljenog i time je kontekst prenet u potpunosti.

Po prijemu ID-a udaljenog agenta, lokalni agentski centar oslobađa lokalnog agenta koji je zatražio premeštanje i ažurira lokalne tabele:

1. tabelu eksternih agenata (**`externalAgentAddress`**) – premešteni agent je sada sa stanovišta lokalnog agentskog centra eksterni agent i u ovoj tabeli se nalazi par (*spoljašnjiAgentID*, *adresaUdaljenogAgentskogCentra*) i

2. tabelu premeštenih agenata (**`teleportedAgents`**) – svako obraćanje premeštenom lokalnom agentu mora da završi u udaljenom agentskom centru. Za ove potrebe se koristi tabela premeštenih agenata koja sadrži parove (*lokalniAgentID*, *spoljašnjiAgentID*). Na osnovu *lokalniAgentID*-a se dolazi do *spoljašnjiAgentID*-a, a on se koristi za pronalaženje udaljenog agentskog centra u tabeli eksternih agenata.

Nakon ažuriranja tabela, poziva se metoda **`onArrival() Agent`** interfejsa, čime se agent obaveštava da je prenos uspešno završen. To znači da će agent biti obavešten o uspešno završenom prenosu pozivom ove metode, a ako je prenos neuspešan, biće generisan izuzetak.

Prilikom deaktiviranja lokalnog agenta koji je prenesen, utvrđuje se da li je agent prenesen, a zatim se pronalazi agentski centar do kojeg je prenet i tamo se inicira deaktiviranje. Agent se pronalazi u tabeli premeštenih agenata, i odatle se dobija njegov novi ID na odredištu. Na osnovu novog ID-a, pronalazi se odredišni agentski centar (u tabeli eksternih agenata), i njemu se zadaje da deaktivira agenta.

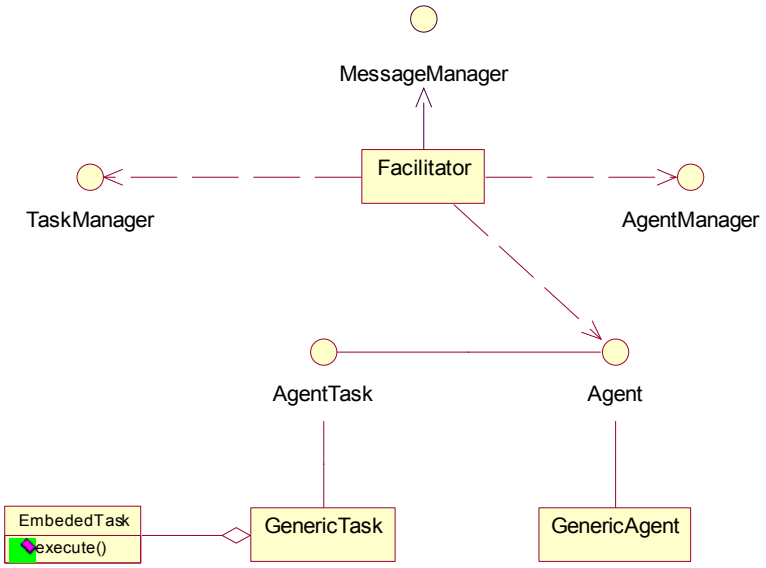
Po uspešnom deaktiviranju agenta, obe tabele se ažuriraju brisanjem stavki koje su se odnosile na deaktiviranog agenta. Ako se agent preselio i sa te lokacije, inicira se postupak na novoj lokaciji. Postupak se ponavlja dok se agent ne pronađe i deaktivira.

4.1.3 Mobilnost zadataka

Mobilnost koda osim agentske mobilnosti pruža još jednu mogućnost – mogućnost da agent izvršava proizvoljan kod prosleđen kao objekat preko mreže. Ova mogućnost znači da nije potrebno pisati namenske agente, već bi bilo poželjno da se omogući izvršavanje proizvoljnog koda u agentima. Ovim bi se agenti mogli svesti na okvire za izvršavanje zadataka, a zadatak bi sadržavao u sebi i program za svoje rešavanje. Mobilnost zadatka, znači, predstavlja mogućnost da zadatak u sebi sadrži i program za svoje rešavanje.

Klasa **`AgentTask`** u sebi sadrži metodu **`execute()`**, kojom se omogućava zadatku da se izvršava unutar agenta. Time sam zadatak preuzima upravljanje nad agentom. Zadatak ima pristup svim metodama i atributima koje "vidi" agent. Zadatak može da zatraži od agentskog centra druge agente i da sebe prosledi njima. Zadatak u tim agentima nastavlja svoje izvršenje, a režim izvršenja je diktiran prosleđenim atributima.

Za tu svrhu su razvijene dve klase: **`GenericTask`** i **`GenericAgent`**. Ove dve klase omogućuju proizvoljnom zadatku da preuzme kontrolu nad agentima, tako što se ugradi u objekat klase **`GenericTask`**, a objekat klase **`GenericAgent`** automatski izvršava ugrađeni kod.



Slika 4.5: Mobilnost zadatka

Na slici 4.5 se nalazi primer realizacije mobilnog zadatka. Objekat klase **`EmbededTask`** predstavlja zadatak koji će u sebi sadržati i program koji će ga rešiti. Ovaj zadatak se ugrađuje u **`GenericTask`**, a klasa **`GenericAgent`** ga izvršava po automatizmu. Time je omogućeno da se program sadržan u klasi **`EmbededTask`** izvršava unutar generičkog agenta. Ovaj kod ima mogućnost pristupa agentskom centru, može da šalje i prima KQML poruke, i samim tim je u stanju da se premešta sa jednog na drugi računar. Premeštanje koda sa jednog računara na drugi se svodi na regrutaciju generičkog agenta na udaljenom računaru i prosleđivanje instance zadatka. Ovaj agent na udaljenom računaru dobija instancu zadatka koja nastavlja izvršenje na udaljenom serveru.

4.1.4 Upravljanje agentima

Za upravljanje agentima zadužen je **`AgentManager`** interfejs, odnosno odgovarajuća klasa koja ga implementira. Ova komponenta predstavlja direktorijum agenata i služi za regrutaciju agenata za odgovarajući posao.

Osim regrutacije, ova komponenta i deaktivira agente. Klasa koja implementira ovaj interfejs mora da implementira metode nabrojane u tabeli 4.2.

Naziv	**Opis**
`Object getAgent(AgentTask id)`	Vraća ID agenta na osnovu ID-a zadatka.
`void releaseAgent(AgentTask id, Object agentId)`	Deaktivira agenta.
`Agent getAgent(Object agentId)`	Vraća referencu na agenta na osnovu ID-a.
`boolean exists(Object agentId)`	Vraća *true* ako agent postoji u lokalnom agentskom centru.
`void addExternalAgentAddress (Object agentId, String addr)`	Dodaje adresu agenta na spisak spoljašnjih agenata (adresu njegovog agentskog centra).
`boolean isExternalAgent(Object agentId)`	Vraća *true* ako je agent iz spoljašnjeg agentskog centra.
`String getExternalAgent(Object agentId)`	Vraća adresu spoljašnjeg agentskog centra kojem pripada zadati agent.
`void reprogramAgent(Object agentId, Agent agent)`	Menja kontekst zadatog agenta kontekstom prosleđenog agenta (podrška mobilnim agentima).
`void addTeleportedAgent( Object agentId, Object remoteAgentId)`	Dodaje agenta u asocijativnu listu prenetih agenata. Sadržaj liste su parovi (lokalni ID agenta, udaljeni ID agenta nakon prenosa). Ova metoda služi za podršku mobilnim agentima.
`boolean isTeleportedAgent( Object agentId)`	Vraća *true* ako je agent prenet na drugu lokaciju. Ova metoda služi za podršku mobilnim agentima.
`Object getTeleportedAgentId( Object agentId)`	Vraća ID agenta na novoj lokaciji. Ova metoda služi za podršku mobilnim agentima.

Tabela 4.2:. Spisak metoda koje je potrebno implementirati za rad *AgentManager* komponente

Ovaj interfejs specificira upravljanje agentima na sledeći način:

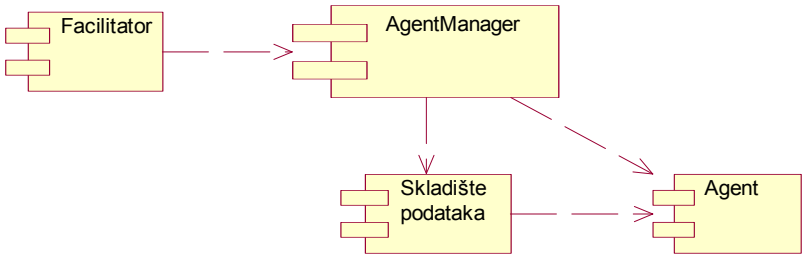
1. dodaje i uklanja agenta iz direktorijumskog servisa;
2. vodi poseban direktorijum za agente koji ne pripadaju lokalnoj *Facilitator* komponenti, a zatraženi su preko ovog servisa i
3. vodi poseban direktorijum prenesenih agenata na nove lokacije.

Direktorijum lokalnih agenata služi za manipulaciju agentima koji su kreirani lokalno. Svaka stavka ovog servisa sadrži ID agenta i lokalnu referencu. Prilikom deaktiviranja lokalnog agenta, uklanja se i odgovarajuća stavka u ovom servisu. Prilikom slanja poruka ili deaktiviranja agenta, na osnovu ID-a agenta dobija se referenca koja se koristi za pristup samom agentu. Prilikom regrutacije agenta, ubacuje se stavka u ovu listu, a prilikom deaktiviranja se uklanja.

Direktorijum onih agenata koji ne pripadaju lokalnoj *Facilitator* komponenti se koristi prilikom rada sa spoljašnjim agentima (agentima koji pripadaju spoljašnjem agentskom centru). Ovaj servis čuva ID agenta i adresu udaljenog agentskog centra kojem pripadaju. Kada je potrebno poslati poruku spoljašnjem agentu ili ga deaktivirati, prvo se proverava da li je agent spoljašnji (pretragom po direktorijumskom servisu). Ako je spoljašnji, odgovarajuća poruka se šalje njegovom agentskom centru. Ako nije, obavlja se posao u lokalu. Prilikom deaktiviranja udaljenog agenta, ova stavka se uklanja iz liste na udaljenom agentskom centru.

Direktorijum prenesenih agenata se koristi prilikom prenosa agenta na novu lokaciju. Svaki agent koji se prenese na novu lokaciju oslobađa se u lokalnom agentskom centru, ali se njegov ID i nova adresa ubacuju u ovu listu. Time je omogućeno da se komunikacija sa agentom nastavi i nakon prenosa na novu lokaciju. Prilikom deaktiviranja agenta, njegova stavka se uklanja i iz ove liste.

Klasa koja implementira ovaj interfejs mora da implementira specificirane metode i da održava sve liste. Mesto čuvanja podataka nije specificirano i može biti asocijativna lista, baza podataka, *EntityBean* komponenta, JNDI servis i dr. (slika 4.6).



Slika 4.6: Dijagram komponenti direktorijumskog servisa agenata

Metoda **`reprogramAgent()`** obavlja punjenje konteksta lokalnog agenta na osnovu prosleđenog agenta. Time je obezbeđeno da lokalni agent preuzme identitet i kontekst agenta koji je prešao u novi agentski centar.

4.1.5 Upravljanje zadacima i njihovo izvršenje

TaskManager komponenta omogućuje izvršenje zadataka. Ovu komponentu realizuje klasa koja implementira **`TaskManager`** interfejs. Ovaj interfejs sadrži metode nabrojane u tabeli 4.3.

Naziv	**Opis**
`AgentTask getTask(Object id, boolean remove)`	Vraća zadatak na osnovu id-a.
`void setTask(AgentTask task, Object id)`	Smesta zadatak u repozitorijum.
`void removeTask(Object id)`	Uklanja zadatak iz repozitorijuma.

Tabela 4.3. Spisak metoda **`TaskManager`** interfejsa.

Svaki zadatak se pre izvršenja smešta u repozitorijum. Nakon izvršenja zadatka, on se uklanja iz repozitorijuma.

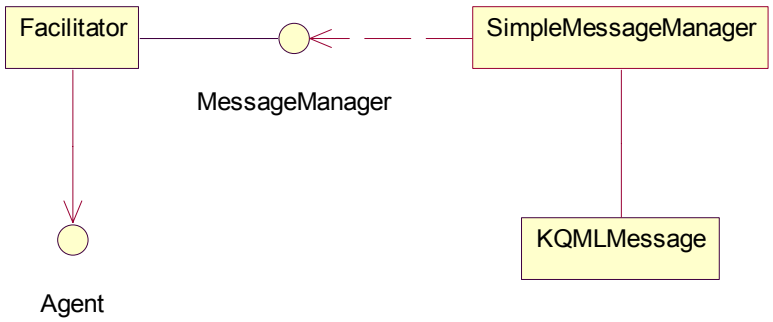
Izvršenje zadatka se realizuje na dva načina: programski (izvršenjem **`execute()`** metode agenta) ili slanjem KQML poruke agentu. Zadaci se definišu kao instance klasa koje implementiraju **`AgentTask`** interfejs. Instanca klase koja implementira **`AgentTask`** interfejs se prosleđuje kao parametar metodi **`execute()`** interfejsa **`Facilitator`**. Ova metoda angažuje agenta, prosleđuje mu parametre i vraća klijentu rezultat rada.

Kod izvršenja zadatka upotrebom KQML poruka, klijent šalje KQML poruku sa formulacijom zadatka agentskom centru. Agentski centar na

osnovu zadatka regrutuje odgovarajućeg agenta i prosleđuje mu istu KQML poruku sa formulacijom zadatka.

4.1.6 Razmena poruka između agenata

Razmena poruka između agenata se svodi na razmenu KQML poruka. Poruke se razmenjuju upotrebom *MessageManager* komponente. Ova komponenta implementira **`MessageManager`** interfejs i koristi usluge *Messaging* sistema aplikacionog servera [CORBAMessaging, JMS], tako da dodatno programiranje sistema za razmenu poruka nije potrebno. Slika 4.7 prikazuje dijagram klasa sistema za razmenu poruka.



Slika 4.7: Dijagram klasa sistema za razmenu poruka.

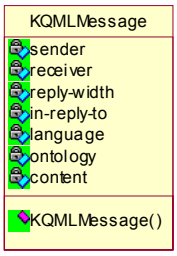
Interfejs **`MessageManager`** definiše osnovni skup metoda koje moraju da se implementiraju za međuagentsku komunikaciju. Klase koje implementiraju ovaj interfejs (klasa **`SimpleMessageManager`** na primer) zapravo implementiraju odgovarajući algoritam za komunikaciju. Ovaj interfejs definiše osnovni skup metoda koje moraju da budu implementirane za obezbeđenje funkcionalnosti komunikacije. Metode su nabrojane u tabeli 4.4.

Naziv	Opis
`void onRemove()`	Poziva se prilikom uklanjanja menadžera.
`Object putKQMLMessage(KQMLMessage message)`	Smešta poruku u skladište (*repository*).
`KQMLMessage getKQMLMessage(Object id)`	Vadi poruku iz skladišta.
`void sendKQMLMessage(KQMLMessage message)`	Šalje poruku.
`void sendKQMLMessage(Object msgId)`	Šalje poruku koja je prethodno smeštena u skladište.

Tabela 4.4. Spisak metoda koje moraju da se implementiraju za rad *MessageManager*-a.

Kada agent šalje KQML poruku drugom agentu, tada se KQML poruka šalje ka agentskom centru koji sadrži agenta kome je poruka upućena. Ova poruka se prosleđuje agentu pozivanjem njegove metode `onKQMLMessage()`.

Poruke su enkapsulirane u osnovnu klasu – klasu `KQMLMessage` prikazanu na slici 4.8.



Slika 4.8: Klasa *KQMLMessage*

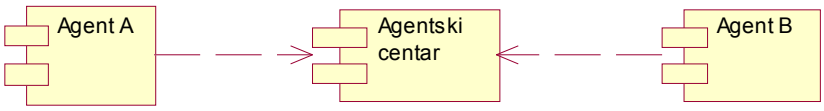
Poruke se razmenjuju prosleđivanjem objekta klase `KQMLMessage` između učesnika dijaloga. Razmena KQML poruka je asinhrona. To je takođe postignuto upotrebom *Messaging* servisa aplikacionog servera. Rutiranje poruka obavlja *ConnectionManager*, uz pomoć *AgentManager*-a. Ova dva menadžera su u stanju da prepoznaju da li je poruka namenjena

agentu unutar agentskog centra, a ako nije, u stanju su da odrede koji agentski centar sadrži agenta primaoca poruke. Oni su u stanju da odrede adresu primaoca na osnovu asocijativne liste adresa agentskih centara, čiji je ključ ID agenta kome se šalje poruka.

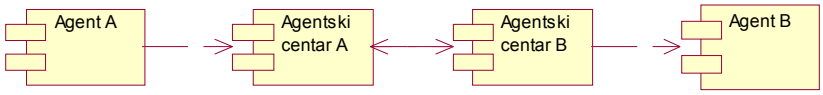
Postoje tri tipa komunikacije između agenata:

1. komunikacija agent – agentski centar. Svaki agent poseduje referencu na svoj agentski centar tako da se razmena poruka svodi na pozivanje zadatih metoda u agentskom centru i agentu;
2. komunikacija agent A – agent B, gde oba agenta pripadaju istom agentskom centru. Agenti međusobno komuniciraju preko agentskog centra, koji obezbeđuje direktnu vezu između agenata i
3. komunikacija agent A – agent B, gde su agenti iz različitih agentskih centara.

Slike 4.9 i 4.10 ilustruju međuagentsku komunikaciju u okviru istog i različitog agentskog centra.

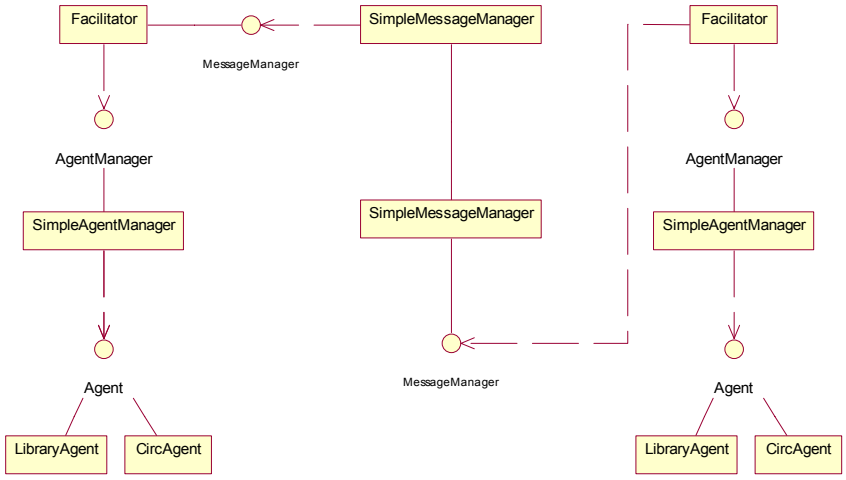


Slika 4.9: Dijagram komponenti za komunikaciju agenata iz istog agentskog centra



Slika 4.10: Dijagram komponenti za komunikaciju agenata iz različitih agentskih centara

Agentski centar obavlja komunikaciju sa agentima preko menadžera poruka, čiji je zadatak prosleđivanje i rutiranje poruka. U slučaju komunikacije sa agentima iz drugih agentskih centara, koriste se usluge *ConnectionManager* komponente. Slika 4.11 prikazuje klase koje učestvuju u komunikacionom procesu.



Slika 4.11: Dijagram klasa koje učestvuju u komunikaciji

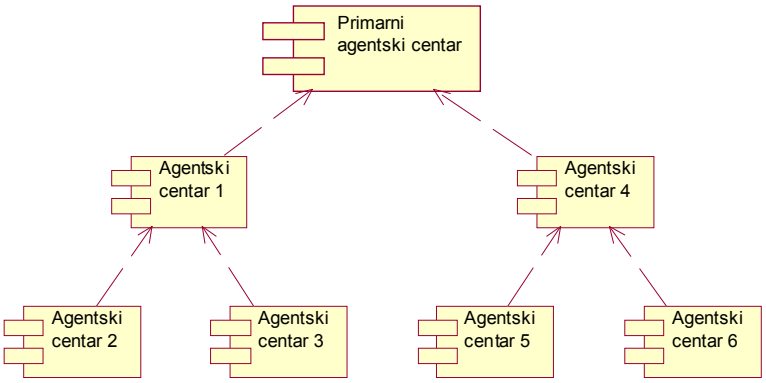
Ako je potrebno kriptovati i/ili potpisati poruku, pozivaju se odgovarajuće metode *SecurityManager*-a. U tom slučaju, kriptovana poruka postaje deo nove poruke, koja predstavlja novi sloj u komunikaciji. Za ovaj sloj je zadužen *SecurtyManager*. Po prijemu poruke, odvija se proces dekriptovanja i/ili verifikacije, i nakon toga, poruka se u svom originalnom obliku prosleđuje *MessageManager*-u.

4.1.7 Uređenje međusobnih odnosa između agentskih centara

Neophodno je da se uspostavi način međuagentskog uređenja odnosa. Ovo se pre svega odnosi na agentske centre. Agentski centri moraju biti u stanju da međusobno komuniciraju, a da za to nije potrebno da programer zna adrese centara, već da oni formiraju određenu strukturu.

Motiv za uređenje međusobnih odnosa može biti i raspodela opterećenja, kao što je prikazano u [Lyell02]. Umesto da se veliki broj agenata potreban za rešenje problema kreira u jednom agentskom centru, može se formirati hijerarhijska struktura servera, čime se smanjuje opterećenje pojedinačnog agentskog centra. U knjizi [Ferber99] se uvodi definicija agentske organizacije (*Multi-Agent Organization*) kao sredstva koje omogućuje međusobna dejstva između agenata. Agentska organizacija predstavlja način na koji se zadaci, podaci, resursi i koordinacija akcija mogu distribuirati.

Agentski centri se mogu urediti, na primer, u strukturu stabla, u čijem korenu se nalazi primarni agentski centar. Slika 4.12 ilustruje ovakvu organizaciju.

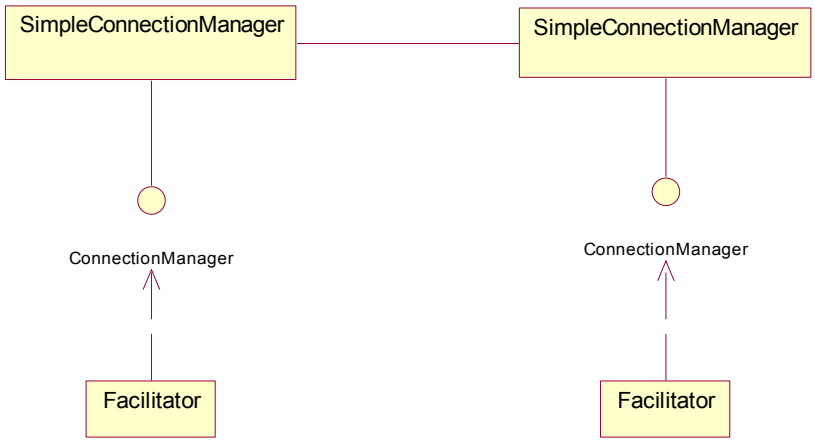


Slika 4.12: Dijagram komponenti hijerarhijske organizacije

Osnovni zadatak agentskih centara je da se po inicijalizaciji prijave svom nadređenom centru, tako da se u svakom trenutku zna stanje u mreži centara. Sve ovo je neophodno da bi agentski centri bili u stanju da uslužuju klijentske programe i agente (kako sopstvene, tako i iz drugih centara). Pružanje usluga agentima se svodi na registraciju, prenos poruka i pronalaženje drugih agenata. Za sve ove usluge potrebna je komunikacija sa drugim agentskim centrima. Posao komunikacije sa drugim agentskim centrima obavljaju klase koje implementiraju `ConnectionManager` interfejs (slika 4.13). Ovaj interfejs definiše osnovni skup metoda koje moraju da se implementiraju da bi agentski centri mogli da komuniciraju međusobno (navedenih u tabeli 4.5).

Naziv	Opis
`void onRemove()`	Poziva se prilikom uklanjanja komponente.
`String getMyAddr()`	Vraća adresu agentskog centra.
`void register(String Facilitator)`	Registruje adresu prosleđenog agentskog centra u svom domenu.
`void unregister(String child)`	Otkazuje registraciju.
`Facilitator getRoot()`	Vraća korenski agentski centar.
`Enumeration getAllFacilitators()`	Vraća sve agentske centre koji postoje u listi.

Tabela 4.5: Spisak metoda `ConnectionManager` interfejsa



Slika 4.13: Dijagram klasa komunikacije između agentskih centara

Prilikom inicijalizacije, agentski centar ima konfigurisan roditeljski agentski centar i njemu se prijavljuje (registruje). Svaki agentski centar održava dinamičku listu agentskih centara ispod sebe. Ako je agentskom centru potreban spisak svih aktivnih agentskih centara, on traži od svakog podređenog agentskog centra spisak, kao i od roditeljskog agentskog centra.

4.1.8 Sigurnosni mehanizmi

Sigurnost je veoma važan aspekt rada agentskog okruženja. Sigurnost se ne odnosi samo na komunikaciju između agenata, već i na sigurnost podataka unutar agenata, sigurnost njihovog prenosa iz jednog agentskog okruženja u drugi, sigurnost agenata od neautorizovane upotrebe, kao i sigurnost agentskog okruženja od malicioznih agenata. Svi ovi koncepti se mogu ostvariti upotrebom sigurnosnih mehanizama unutar postojećih aplikacionih servera. Najmanje što se može ostvariti je autentifikacija korisnika komponenti. Time je omogućena kontrola pristupa komponentama na serveru, a samim tim i kontrola pristupa agentima, tj. njihova zaštita od neautorizovane upotrebe. Sigurnost podataka unutar agenata i sigurnost njihovog prenosa se ostvaruju upotrebom kriptografskih tehnika. Sigurnost agentskog okruženja od malicioznih agenata se ostvaruje implementacijom sigurnosnih mehanizama unutar konkretnog programskog jezika.

Većina današnjih aplikacionih servera podržavaju kriptovanu komunikaciju između korisnika i servera, kao i između dva servera. Digitalno potpisivanje i verifikacija uz upotrebu sertifikata je takođe kod nekih servera podržana [WebSphere]. Ako korisnik nije zadovoljan postojećim nivoom zaštite u aplikacionom serveru ili želi da uvede alternativni sistem zaštite, to mu je omogućeno implementacijom *SecurityManager* komponente. Sigurnosni mehanizmi [He98, Yuh-Jong01] se implementiraju u *SecurityManager* komponenti definicijom metoda koje su u njoj specificirane (tabela 4.6).

Naziv	**Opis**
`byte[] encrypt(byte[] source)`	Vrši kriptovanje prosleđenog niza bajtova.
`byte[] decrypt(byte[] source)`	Vrši dekriptovanje prosleđenog niza bajtova.
`byte[] sign(byte[] source)`	Vrši potpisivanje prosleđenog niza bajtova.
`boolean verify(byte[] source, byte[] signature)`	Vrši proveru validnosti potpisanog niza bajtova.

Tabela 4.6. Spisak metoda **`SecurityManager`** interfejsa.

Metode **`encrypt()`** i **`decrypt()`** obezbeđuju kriptovanje i dekriptovanje proizvoljnog niza bajtova. Metode **`sign()`** i **`verify()`** obezbeđuju digitalni potpis i verifikaciju digitalnog potpisa. Ova dva para metoda

nije neophodno implementirati. Ako u sistemu tajnost podataka nije obavezna, već je samo potrebno obezbediti integritet podataka, dovoljno je implementirati metode **`sign()`** i **`verify()`**.

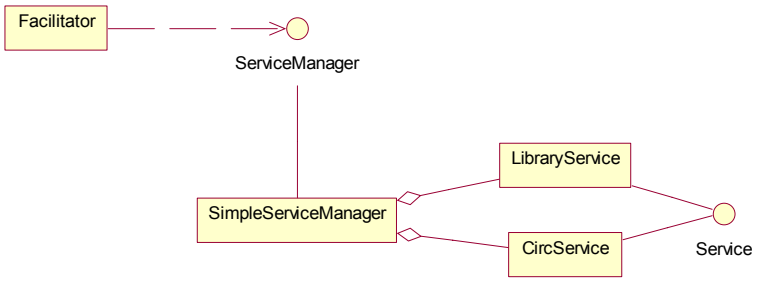
Navedene metode **`SecurityManager`** interfejsa rade nad proizvoljnim nizom bajtova. Time se upotreba ovog menadžera ne ograničava na razmenu poruka, već se može koristiti i kod prenosa agenata, zadavanja zadataka, prenosa rezultata rada i dr.

Kriptografski sistem nije ovim interfejsom specificiran i na autoru menadžera je da odabere potrebne algoritme i biblioteke. Ako se koristi asimetrično kriptovanje [Menzes97], ukazaće će se potreba za razmenom javnih ključeva. U tom slučaju, mogu se upotrebiti sertifikati, kao mehanizam koji omogućuje dodatnu sigurnost, odnosno potvrđuje identitet onoga koji publikuje svoj javni ključ. Mehanizam razmene sertifikata nije preciziran i na korisniku je da izabere odgovarajući metod.

Kao alternativa gornjem pristupu, moguće je i da *SecurityManager* otvori SSL *socket* ka drugom serveru i time se ostvaruje automatska razmena sertifikata i kriptovanje podataka. Ovo podrazumeva da sva komunikacija između servera ide preko SSL konekcije, a ne regularnim vezama.

4.1.9 Direktorijum servisa

Direktorijum servisa (*ServiceManager* komponenta) predstavlja podsistem koji upravlja spiskom servisa dostupnih agentima i agentskim centrima (slika 4.14). Servisi predstavljaju programske podsisteme sposobne da obavljaju specijalizovane zadatke. Umesto da se razvijaju agenti za rešavanje nekih zajedničkih zadataka, agentski centar ima na raspolaganju servise koji obavljaju te zajedničke zadatke.



Slika 4.14: Dijagram klasa direktorijuma servisa

Servisi prijavljeni ovom sistemu se mogu pretraživati, dodavati, uklanjati i iznajmljivati. Metode potrebne za implementaciju ovog menadžera su nabrojane u tabeli 4.7.

Naziv	Opis
`Object addService(Service service)`	Dodaje servis u direktorijum.
`void removeService(Object serviceID)`	Uklanja servis iz direktorijuma.
`Service getService(Object serviceID)`	Iznajmljuje servis korisniku.
`void returnService(Object serviceID)`	Vraća servis u direktorijum.
`Enumeration listServices()`	Vraća spisak svih servisa u direktorijumu.
`Object findService(String desc)`	Traži zadati servis u direktorijumu.

Tabela 4.7: Metode koje se moraju implementirati za *ServiceManager* komponentu

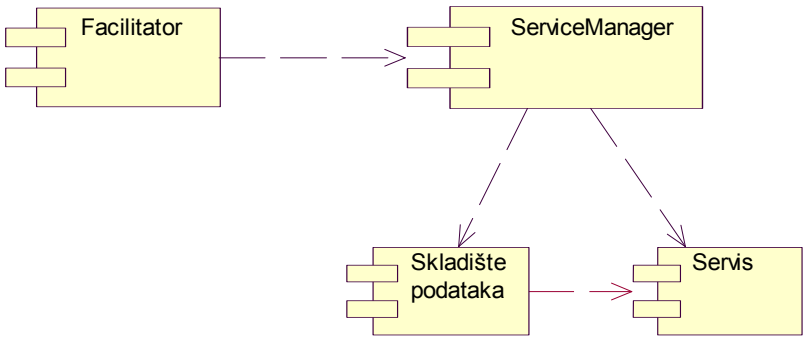
Servisi su predstavljeni interfejsom **`Service`**. Ovaj interfejs specificira metode koje svaki servis mora da implementira (tabela 4.8).

Naziv	Opis
`void remove()`	Poziva se prilikom vraćanja servisa u direktorijum.

Tabela 4.8: Metode interfejsa **`Service`**

Servis se po prestanku upotrebe vraća u direktorijum, metodom **`returnService()`**. Ova metoda poziva metodu **`remove()`** servisa neposredno pre vraćanja u direktorijum. Time je omogućeno da servis obavi finalizacione radnje prilikom vraćanja u direktorijum.

Klasa koja implementira **`ServiceManager`** interfejs mora da implementira specificirane metode i da održava listu servisa. Mesto čuvanja podataka nije specificirano i može biti asocijativna lista, baza podataka, *EntityBean* komponenta, JNDI servis i dr. (slika 4.15).



Slika 4.15: Dijagram komponenti direktorijumskog sistema servisa

4.1.10 Agenti

Agenti su softverske komponente koji izvršavaju prosleđeni zadatak. Ove komponente su klase koje implementiraju **`Agent`** interfejs. Ovaj interfejs specificira minimalan broj metoda koje agent mora da implementira (tabela 4.9).

Naziv	**Opis**
`AgentResult execute(AgentTask task)`	Izvršava prosleđeni zadatak.
`void onKQMLMessage(KQMLMessage message)`	Po prijemu KQML poruke, ova metoda se izvršava.
`KQMLMessage waitKQMLMessage()`	Čeka na prispeće KQML poruke.
`Facilitator getFacilitator()`	Vraća *Facilitator* komponentu.
`void onArrival(Object agentId, String senderAddress, AgentTask task)`	Poziva se kada se agent preseli u drugi agentski centar.

Tabela 4.9: Spisak metoda koje agentska komponenta mora da implementira

Za programsko izvršenje zadatka zadužena je metoda **`execute()`**. Ova metoda je automatski pozvana od strane *Facilitator* komponente ako je specificirano programsko izvršenje zadatka. U tom slučaju, ova metoda mora da vrati objekat klase **`AgentResult`** kao rezultat rada. Ako je potrebno da se ovaj zadatak više puta izvrši, a da međurezultate prosledi

onome ko ga je zadao, tada će se u rezultatu rada naglasiti da zadatak nije završen (atribut **`finished`** interfejsa **`AgentResult`**), a agent će zapamtiti gde je stao, tako da prilikom ponovnog poziva može da nastavi izvršenje zadatka.

Ako je izvršenje zadatka zadato upotrebom KQML poruka, tada se početna KQML poruka sa definicijom zadatka šalje agentu, a po prispeću poruke, biće aktivirana metoda **`onKQMLMessage()`**. Svaka poruka koja stigne do agenta će izazvati izvršenje ove metode. Svako obraćanje agentskom centru se vrši preko parametra **`Facilitator`**, koji predstavlja referencu na agentski centar.

Metoda **`onArrival()`** se koristi prilikom prelaženja iz jednog agentskog centra u drugi. Ovu metodu nije neophodno implementirati ako mobilnost agenta neće biti obezbeđena.

Poglavlje 5

Implementacija proširivog agentskog okruženja u J2EE tehnologiji

Mana jednog broja postojećih agentskih okruženja je ta da se za implementaciju distribuiranih komponenti koristi RMI, koja ne pruža dovoljno mogućnosti. Osim ovoga, svi potrebni koncepti (prenos poruka, direktorijumi agenata i servisa, sigurnosni mehanizmi) su iznova programirani, zato što nisu korišćene tehnologije distribuiranih komponenti (CORBA, J2EE, DCOM). Ove tehnologije imaju razvijene podsisteme za razmenu poruka, direktorijumske servise, sigurnosne mehanizme i odgovarajuće standardizovane protokole za sve ove koncepte. Osim toga, ove tehnologije se zasnivaju na aplikacionim serverima [Milosavljević02] u kojima se izvršavaju distribuirane komponente.

Kao što je navedeno u poglavlju 2, agentsko okruženje BlueJADE [Bellifemine99, Cowan02] koristi J2EE tehnologiju, ali ne koristi sve koncepte iz nje. BlueJADE se bazira na HP-ovom aplikacionom serveru, i JADE agentskom okruženju, gde je JADE okruženje implementirano kao servis unutar aplikacionog servera. Sa druge strane, agentsko okruženje SAFT [Blixt00] koristi većinu elemenata J2EE tehnologije, ali ne koristi JMS za asinhronu razmenu poruka. Takođe, direktorijum agenata je implementiran u JNDI tehnologiji i ne postoji način da se upotrebi neka druga tehnologija. Nisu iskorišćeni svi sigurnosni mehanizmi aplikacionog servera, već samo autentifikacija korisnika komponenti.

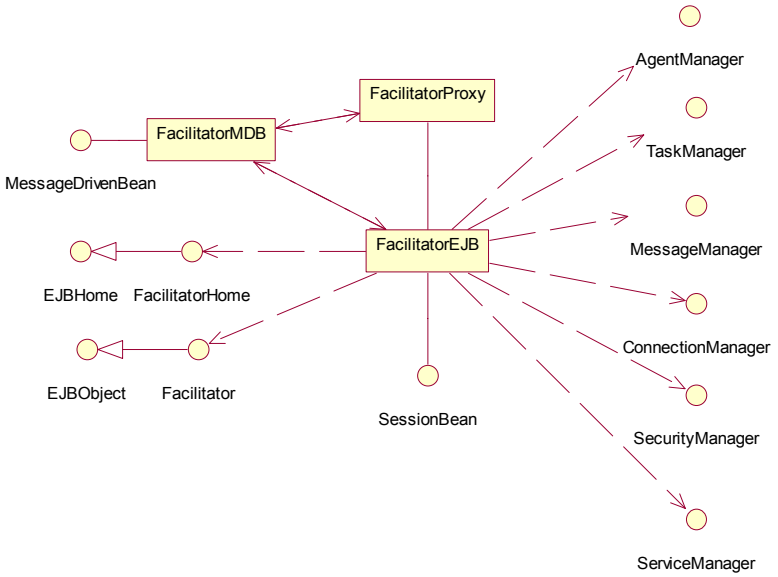
U ovom poglavlju biće predstavljen jedan predlog upotrebe J2EE tehnologije [Vidaković02a] za realizaciju agentskog okruženja. J2EE tehnologija je odabrana zato što ona obuhvata širok spektar tehnologija distribuiranih informacionih sistema, a omogućuje skalabilnost, pouzdanost i ima veliku podršku u aplikacionim serverima. Ono što je karakteristično za J2EE tehnologiju je jednostavna upotreba distribuiranih softverskih komponenti – EJB komponenti, koje se kreiraju, izvršavaju i uništavaju u aplikacionom serveru [Vidaković02b]. Komponente se izvlače iz skladišta komponenti, koriste i zatim vraćaju nazad u skladište. Količina komponenti, njihova raspodela po klijentima i upravljanje opterećenjem [Rana00] je prepušteno aplikacionim

serverima. Osim podrške distribuiranim komponentama, J2EE tehnologija sadrži sve potrebne sisteme za rad agentskog okruženja, uključujući tu JMS za razmenu poruka, JNDI podsistem za direktorijumske sisteme i bezbednosne mehanizme.

Implementacija agentskog okruženja je izvedena u programskom jeziku Java. Sistem se sastoji iz Java klijenta i aplikacionog servera na kojem se nalazi implementacija agentskog sistema. Aplikacioni server je J2EE aplikacioni server i za ovu priliku su odabrani Orion i JBoss aplikacioni serveri [Orion, JBoss]. Ovi serveri imaju implementirane sve J2EE koncepte (EJB, JNDI, JTS, i dr.). EJB komponente i ostale klase su implementirane prema specifikaciji iz poglavlja 4.

5.1 Agentski centar

Agentski centar (*Facilitator* komponenta) je centralna komponenta ovog agentskog okruženja i realizovana je kao EJB komponenta na J2EE aplikacionom serveru.



Slika 5.1. Dijagram klasa *Facilitator* komponente.

Facilitator komponenta je reprezentovana klasom **`FacilitatorEJB`**. Ova klasa implementira **`SessionBean`** interfejs. **`FacilitatorEJB`** zavisi od

dva interfejsa: **FacilitatorHome** (koji definiše metode za upravljanje životnim ciklusom komponente) i **Facilitator** (koja definiše *Remote* interfejs komponente, tj. sve *business* metode koje su dostupne drugim klasama). **FacilitatorProxy** klasa služi za skrivanje svih implementacionih tehnika od klijenta i služi kao veza između klijentske aplikacije i **FacilitatorEJB** komponente. **FacilitatorMDB** je *MessageDrivenBean* EJB komponenta koja služi za JMS komunikaciju između **FacilitatorEJB** komponente i ostatka sistema. Svaka JMS poruka ka agentskom sistemu prolazi kroz ovu komponentu.

5.2 Mobilnost agenata

Mobilnost agenata je implementirana prema specifikaciji iz prethodnog poglavlja. Za prebacivanje agenta potrebno je implementirati metodu **moveTo()** agentskog centra (listing 5.1).

```
public Object moveTo(String address, Object agentId,
                     AgentTask task)
                     throws RemoteException {
  Facilitator external = new ProxyConnector(address);
  try {
    Agent agent = Facilitator.getAgentManager().getAgent(
                  agentId);
    Object newAgentId = external.acceptAgent(agent, task,
                                             myAddr);
    Facilitator.releaseAgent(agentId);
    Facilitator.getAgentManager().addTeleportedAgent(
      agentId, newAgentId);
    Facilitator.getAgentManager().addExternalAgentAddress(
      newAgentId, address);

    return external.onArrival(newAgentId,
      Facilitator.getConnectionManager().getMyAddr(),
      task);
  } catch (RemoteException ex) {
    String s = "FATAL ERROR: Could not send agent to the
remote Facilitator at: " + address;
    ex.printStackTrace();
    throw new RemoteException(s);
  }
}
```

Listing 5.1: Prebacivanje agenta u drugi agentski centar

Metoda **moveTo()** počinje spajanjem na odredišni agentski centar. Za to služi klasa **ProxyConnector**, koja prilikom konstrukcije prima adresu odredišnog agentskog centra. Ova klasa implementira **Facilitator**

interfejs, tako da sa stanovišta korisnika, ona predstavlja udaljeni agentski centar.

Od odredišne strane se zahteva jedan agent koji je u stanju da reši zadati zadatak (metoda **`acceptAgent()`**). Ako takav agent postoji, on se regrutuje na odredišnoj strani i u njega se premešta kontekst polaznog agenta. To znači da je u agenta na odredišnoj strani preneseno kompletno stanje agenta sa polazne strane.

Nakon prenosa konteksta na odredište, polazni agent se oslobađa, ali se njegov ID smešta u listu prebačenih agenata (**`addTeleportedAgent()`** metoda). Na ovaj način, svako obraćanje agentu sa polazne strane će biti preusmereno na odredišnu stranu.

ID agenta sa odredišne strane se ubacuje u listu spoljašnjih agenata (metoda **`addExternalAgentAddress()`**) da bi se znala adresa na koju je premešten agent.

Metoda **`onArrival()`** se poziva nakon uspešnog prenosa agenta na odredište.

5.3 Mobilnost zadataka

Mobilni zadaci se implementiraju dvema klasama: **`GenericTask`** i **`GenericAgent`**. Klasa **`GenericTask`** implementira **`AgentTask`** interfejs i predstavlja generički zadatak koji u sebi sadrži stvarni zadatak. Klasa **`GenericAgent`** implementira **`Agent`** interfejs i predstavlja generičkog agenta koji je u stanju da reši generički zadatak. Pošto generički zadatak sadrži u sebi stvarni zadatak, generički agent neće rešavati stvarni zadatak, već će stvarni zadatak preuzeti kontrolu nad generičkim agentom i preko njega će rešiti zadatak (listing 5.2).

```
public AgentResult execute(AgentTask task, Object agentId)
{
  return task.execute(this, agentId);
}
```

Listing 5.2: Preuzimanje kontrole nad agentom

Iz listinga 5.2 se vidi da će generički agent prilikom izvršenja zadatka zapravo pozvati metodu **`execute()`** samog generičkog zadatka, koji će pozvati metodu **`execute()`** stvarnog zadatka (listing 5.3).

```
public AgentResult execute(Agent agent, Object agentId) {
  return embeddedTask.execute(agent, agentId);
}
```

Listing 5.3: Izvršenje stvarnog zadatka

Ovim je omogućeno da stvarni zadatak preuzme kontrolu nad generičkim agentom i da kroz njega realizuje zadatak.

5.3.1 Dinamičko učitavanje klasa

Koncept mobilnosti zadataka uvodi nov problem u postupak implementacije – problem učitavanja proizvoljnih klasa. Ovaj problem se javlja zato što se prilikom izvršenja mobilnog zadatka javlja mogućnost da se zadatak preseli na aplikacioni server koji nema definiciju klase u svom okruženju (***.class** datoteku u **CLASSPATH**-u). Standardna Java serijalizacija prevodi interno stanje objekta u niz bajtova. Ovo stanje agenta podrazumeva samo stanja svih atributa. Definicije metoda se nalaze u ***.class** datoteci. Ova datoteka se ne prenosi na odredište i podrazumeva se da na odredištu postoji njena kopija. To znači da je u specifičnim uslovima moguća situacija da se na odredištu pojavi serijalizovan objekat klase čija definicija ne postoji. U tom slučaju, potrebno je obezbediti mehanizam koji će, po potrebi, uz serijalizovane objekte prenositi i njihove ***.class** datoteke na odredište. Ovaj koncept se zove Dinamičko učitavanje klasa (*Dynamic Class Loading*) [Java, Liang98, Krumel98].

Dinamičko učitavanje klasa je podržano u osnovnoj instalaciji programskog jezika Java. Podržano je `RMIClassLoader` klasom. Ova klasa omogućuje prenos objekata proizvoljne klase od strane klijenta ka serveru, i to samo ako se ti objekti koriste kao parametri udaljenih metoda ili kao povratne vrednosti. To znači da se prilikom pozivanje udaljenih metoda na serverskoj strani prenose serijalizovani objekti proizvoljnih klasa, a na serverskoj strani se ti objekti deserijalizuju na sledeći način:

- ako je objekat klase koja je već učitana, ili definicija klase (***.class** datoteka) postoji, koristi se sistemski *class loader*;
- ako definicija klase ne postoji, kontaktira se pozivna strana (klijentska strana), preuzima se ***.class** datoteka i objekat se deserijalizuje.

Preuzimanje ***.class** datoteke se odvija upotrebom standardnog *http* protokola. To znači da na klijentskoj strani mora da postoji web server koji je u stanju da prosledi odgovarajuća ***.class** datoteka (da se osnovni web direktorijum poklapa sa korenskim paketom klijentske aplikacije). Sistemski parametar `java.rmi.server.codebase` sadrži adresu web servera koji će `RMIClassLoader` kontaktirati. Mana ovog sistema je ta da ovakav sistem učitavanja klasa podržan samo kod RMI izvršavanja

udaljenih objekata. Postojeći J2EE aplikacioni serveri se ne zasnivaju na RMI sistemu.

U agentskom okruženju koje se razmatra u ovoj tezi, potreba za dinamičkim učitavanjem klasa se javlja prilikom izvršenja mobilnih zadataka. Mobilni zadaci predstavljaju proizvoljne klase koje implementiraju **`AgentTask`** interfejs. To znači da svaka klasa koja implementira ovaj interfejs može biti prosleđena agentskom okruženju na rešavanje. Tu se javljaju dva problema:

- definicija klase ne mora biti prisutna u agentskom okruženju i
- bezbednost agentskog okruženja može biti ugrožena zato što se u zadatku nalazi proizvoljan kod [Binder02].

Oba problema se rešavaju implementacijom specijalizovanog *class loader*-a, koji je u stanju da kontaktira klijentsku stranu, prenese ***.class** datoteku i rekonstruiše kompletan objekat. Osim toga, ovaj *class loader* će koristiti usluge *SecurityManager* komponente da bi se sistem zaštitio od malicioznog koda unutar klase koja se rekonstruiše. *SecurityManager* komponenta omogućuje zaštitu sistema na nivou pristupa sistemskim resursima, programskim nitima, mreži i dr. Osim toga, objekat koji se rekonstruiše može da sadrži sertifikat koji će potvrditi da njegov kod nije maliciozan. Listing specijalizovanog *class loader*-a je prikazan u listingu 5.4.

```
public class HttpClassLoader extends ClassLoader {
  private String address;
  private int port;

  /** Konstruktor. */
  public HttpClassLoader() {
    super(Thread.currentThread().getContextClassLoader());
  }

  /** Podešava parametre za komunikaciju sa
   *  HTTP serverom.
   */
  public void setComm(String address, int port) {
    this.address = address;
    this.port = port;
  }
```

Listing 5.4: Specijalizovan *class loader* za dinamičko učitavanje klasa

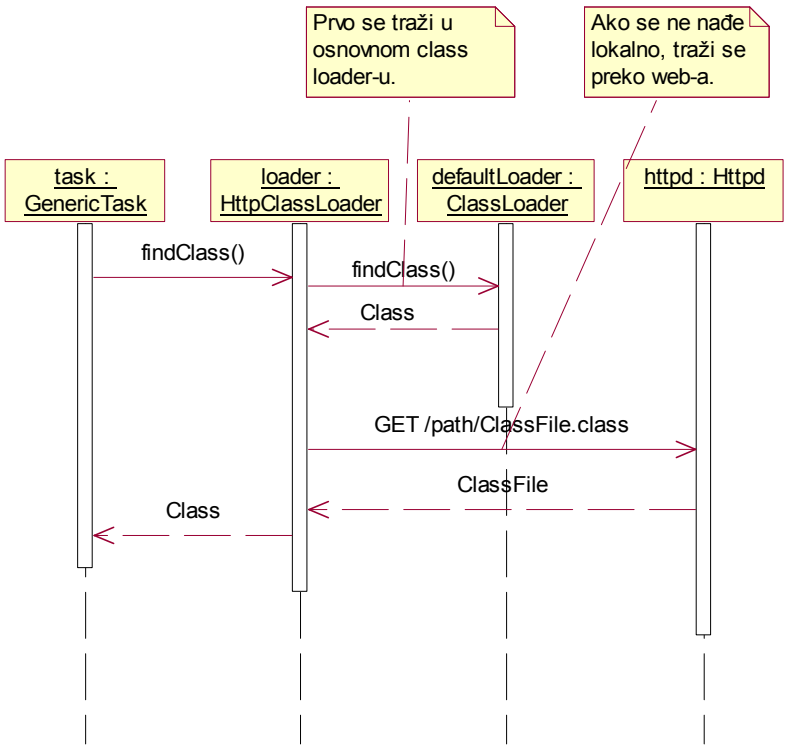
```
  /** Vraća niz stringova, čiji elementi su delovi paketa.
    * @param s Puna naziv klasa sa paketom kome pripada
    * @return Niz potpaketa završno sa nazivom klase.
    */
  private String[] getPathElements(String s) {
    String[] retVal;
    StringTokenizer st = new StringTokenizer(s, ".");
    int len = st.countTokens();
    if (len == 0) {
      retVal = new String[1];
      retVal[0] = s;
    } else {
      retVal = new String[len];
      int i = 0;
      while(st.hasMoreTokens()) {
        retVal[i++] = st.nextToken();
      }
    }
    return retVal;
  }

  /** Rekonstruiše punu putanju do *.class fajla
   *  na osnovu niza potpaketa i naziva klase.
   */
  private String getElementPath(String[] elements) {
    String retVal = "";
    for (int i = 0; i < elements.length; i++)
      retVal += "/" + elements[i];
    return retVal;
  }

  /** Redefinisana metoda za učitavanje klase.
   * Ako ne nadje definiciju klase lokalno,
   * preko http protokola će preuzeti definiciju klase.
   */
  public Class findClass(String name) throws
ClassNotFoundException {
    int len = 0;
    try {
      Class ret = super.findClass(name);
      return ret;
    } catch (ClassNotFoundException e) {
      // ako klasa nije pronađena lokalno
      String[] pathElements = getPathElements(name);
      String elementPath = getElementPath(pathElements);
      String finalPath = elementPath + ".class";
```

Listing 5.4: Specijalizovan *class loader* za dinamičko učitavanje klasa – nastavak

```
      try {
        // spoji se sa serverom
        InetAddress addr = InetAddress.getByName(address);
        Socket s = new Socket(addr, port);
        PrintWriter out = new PrintWriter(
          new BufferedWriter(
            new OutputStreamWriter(
                    s.getOutputStream())), true);
        DataInputStream in = new DataInputStream(
          s.getInputStream());
        out.println("GET " + finalPath + " HTTP/1.1");
        out.println();
        // pokupimo zaglavlje
        in.read(responseBytes, 0, responseBytes.length);
        String response = new String(responseBytes);
        // ako je klasa pronađena na klijenskoj strani
        if (isOk(response))
          len = in.read(classBytes, 0, classBytes.length);
        else
          throw new ClassNotFoundException(
                    "Class definition was not found: " +
                     name);
        in.close();
        out.close();
        s.close();
      } catch (IOException ex) {
        ex.printStackTrace();
      }
      return defineClass(name, classBytes, 0, len);
    }
  }
  /** Vraća <b>true</b> ako je odgovor pozitivan
   * (postoji tražena datoteka.
   */
  private boolean isOk(String response) {
    StringTokenizer st = new StringTokenizer(response);
    st.nextToken();
    String s = st.nextToken();
    if (s.equals("200"))
      return true;
    else
      return false;
  }
  private byte[] classBytes = new byte[100000];
  private byte[] responseBytes = new byte[19];
}
```

Listing 5.4: Specijalizovan *class loader* za dinamičko učitavanje klasa – nastavak

Dijagram sekvence sa slike 5.2 ilustruje način rada klase **`HttpClassLoader`**.



Slika 5.2: Dijagram sekvence dinamičkog učitavanja klase

Prilikom učitavanja generičkog zadatka, koristi se *class loader* klase **`HttpClassLoader`**. Ova klasa nasleđuje standardnu klasu **`ClassLoader`** i redefiniše metodu **`findClass(String name)`**, koja na osnovu punog naziva klase (uključujući i naziv paketa kome pripada) učitava klasu u virtualnu mašinu. Ova metoda je redefinisana tako da prvo potraži definiciju klase u lokalnoj virtuelnoj mašini. Ako je ne nađe, kontaktira web server preko zadate adrese i porta i zahteva definiciju klase (u vidu ***.class** datoteke). Zahtev se upućuje kao standardan *http* zahtev tipa:

```
GET /puna_putanja/ImeKlase.class HTTP/1.1
```

Ovaj zahtev se procesira u web serveru, i ako datoteka postoji, ona se vraća uz odgovarajuću povratnu informaciju:

- ako datoteka postoji, pre njenog sadržaja, šalje se string:

`"HTTP/1.0 200 OK\r\n\r\n"`, nakon čega sledi sama datoteka kao niz bajtova;

- ako datoteka ne postoji, šalje se string:

  `"HTTP/1.0 404 File not Found\r\n\r\n"`.

Svaki mobilni zadatak sadrži adresu web servera preko kojeg se može doći do definicije klase. To omogućuje potpunu mobilnost zadatka, jer je u svakom trenutku i u svakom aplikacionom serveru moguće rekonstruisati prosleđeni zadatak.

5.4 Upravljanje agentima

Upravljanje agentima se svodi na njihovo kreiranje, oslobađanje i pronalaženje odgovarajućeg agenta za zadati zadatak. Agenti su implementirani kao EJB komponente, ali na način da se njihov kontekst implementira kao obična klasa, koja se inicijalizuje prilikom kreiranja EJB komponente. Time je omogućeno da se agent nalazi unutar EJB komponente i da je ona odgovorna za njegov rad, ali da sami agenti ne zavise od konkretne implementacije EJB standarda.

Primer ugradnje agenta u EJB komponentu je prikazan na listingu 5.5.

```
public Object getAgent(AgentTask id)
              throws AgentNotFoundException {
  Object retVal = null;
  // Na osnovu klase zadatka, dobićemo ime klase agenta.
  // Ako ntakav agent ne postoji, bacice
  // AgentNotFoundException, koji ce biti prosledjen na
  // gore.
  String agentClassName;
  try {
    agentClassName = XMLAgentParser.getAgentProperty(
                   "class", id.getClass().getName());
  } catch (AgentNotFoundException ex) {
    // Ako ga nema u konfiguracionom fajlu,
    // onda ovaj čvor nema takvog agenta koji bi rešio
    // zadati posao.
    ex.printStackTrace();
    throw new AgentNotFoundException(ex.getMessage());
  }
  // Kreiramo jedan AgentHolder i ubacimo mu klasu.
  AgentHolder agentHolder = null;
```

Listing 5.5: Ugradnja agenta u EJB komponentu.

```
  try {
    InitialContext context = new InitialContext();
    Object homeObject = context.lookup(
                                        "ejb/AgentHolder");
    AgentHolderHome home =
       (AgentHolderHome)PortableRemoteObject.narrow(
          homeObject, AgentHolderHome.class);
    // Ovim se kreira AgentHolder EJB komponenta.
    // Iniciramo holder imenom klase agenta kojeg
    // treba da sadrži.
    agentHolder = (AgentHolder)PortableRemoteObject.narrow(
             home.create(agentClassName),
             AgentHolder.class);
  } catch(Exception e) {
    throw new AgentNotFoundException("AgentManager error: "
+ e.getMessage());
  return agentHolder;
}
```

Listing 5.5: Ugradnja agenta u EJB komponentu – nastavak

Listing 5.5 prikazuje pronalaženje odgovarajućeg agenta za zadati zadatak. Na osnovu prosleđenog zadatka, traži se naziv klase agenta koji je u stanju da reši zadati zadatak. Ako takvog agenta nema, baca se izuzetak. Ako takav agent postoji, kreira se EJB komponenta koja će čuvati agenta. Prilikom kreiranja EJB komponente, njoj se prosleđuje naziv klase agenta kojeg će sadržati. Ova komponenta prilikom svoje konstrukcije kreira i objekat agenta, kao što je prikazano na listingu 5.6.

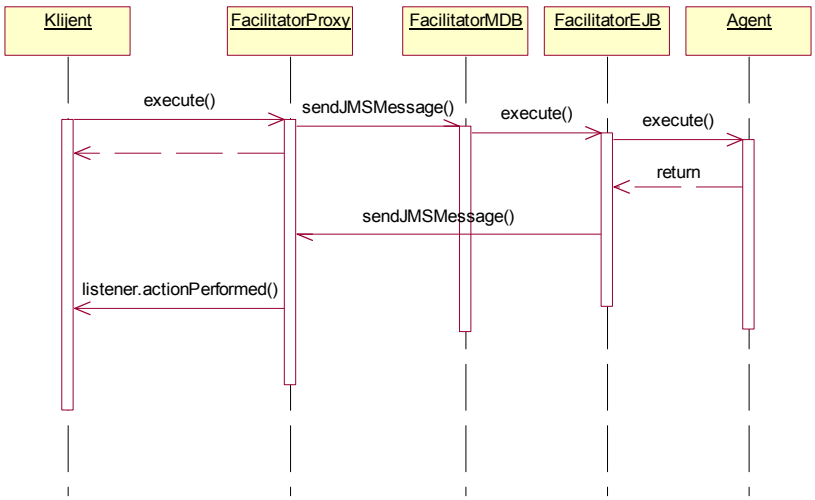
```
public void ejbCreate(String agentClassName)
            throws CreateException {
  // Kreiramo Agenta koji je dodeljen ovoj komponenti.
  try {
    agent = (Agent)Class.forName(
                        agentClassName).newInstance();
  } catch (Exception ex) {
    ex.printStackTrace();
    throw new CreateException("Could not create agent: "+
      agentClassName + ": " + ex.getMessage());
  }
}
```

Listing 5.6: Kreiranje objekta agenta unutar EJB komponente

5.5 Upravljanje zadacima i njihovo izvršenje

Za izvršenje zadataka zadužena je *TaskManager* komponenta. Ovu komponentu realizuje klasa koja implementira **`TaskManager`** interfejs. Izvršenje zadatka se realizuje na dva načina: programski (izvršenjem **`execute()`** metode agenta) ili slanjem KQML poruke agentu. Zadaci se definišu kao instance klasa koje implementiraju **`AgentTask`** interfejs. Instanca klase koja implementira **`AgentTask`** interfejs se prosleđuje kao parametar metodi **`execute()`** klase **`FacilitatorEJB`**. Ova metoda angažuje agenta, prosleđuje mu parametre i vraća klijentu rezultat rada.

Ako usvojimo da klijent ne mora da komunicira sa agentom preko JMS-a (jer se time komplikuje klijentski kod), onda se za komuniciranje od agenta ka klijentu mora usvojiti koncept *listener*-a. *Listener* je Java klasa koju jedna klasa prosledi drugoj, tako da druga po završetku posla obaveštava prvu klasu pozivom odgovarajućih metoda *listener*-a. *Listener*-i se masovno koriste u korisničkom interfejsu u Javi, gde je potrebno da komponente korisničkog interfejsa obaveste glavni program da je izvršena neka akcija nad njima. Slika 5.3 prikazuje dijagram sekvenci izvršenja zadatka.



Slika 5.3: Dijagram sekvenci asinhronog izvršenja zadatka

Klijent samo treba da kreira klasu koja opisuje zadatak i da kreira odgovarajući *listener*. Sav ostali posao sakriven je u klasi **`FacilitatorProxy`**. Ova klasa zapravo kreira JMS poruku koja sadrži

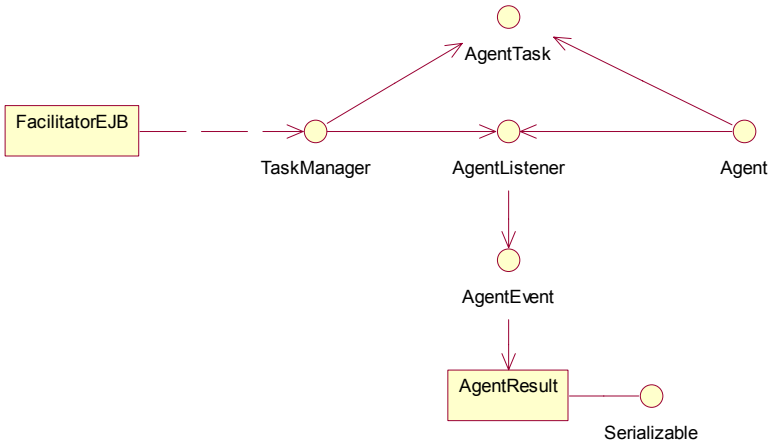
zadatak i šalje je *Facilitator* komponenti. Ovu JMS poruku prima *FacilitatorMDB* komponenta, koja predstavlja primaoca JMS poruka u J2EE aplikacionim serverima. Ova komponenta prima JMS poruku, izdvaja zadatak iz nje, i prosleđuje ga *Facilitator* komponenti. *Facilitator* komponenta regrutuje odgovarajućeg agenta i zadaje mu zadatak. Sva komunikacija od agenta prema klijentu se odvija preko JMS poruka, koje se u **`FacilitatorProxy`** klasi pretvaraju u odgovarajuće pozive *listener* metoda. Ovim je izvedeno asinhrono izvršavanje zadataka, a za tu svrhu je iskorišćen JMS sistem.

Metoda **`execute()`** se izvršava asinhrono, odnosno ne blokira izvršenje klijentskog koda dok se ne realizuje zadatak, već po realizaciji poziva instancu klase naslednice interfejsa **`AgentListener`** koju kreira klijentska aplikacija. Na ovaj način, klijentska aplikacija može da nastavi da izvršava kod, a preko *listener*-a biće upoznata sa rezultatima rada agenta.

Zadaci se zadaju kreiranjem objekata onih klasa koje reprezentuju zadatak. Klase koje reprezentuju zadatak moraju da implementiraju interfejs **`AgentTask`**. Ovaj interfejs definiše zadatak, parametre zadatka i neophodne metode za realizaciju mobilnosti zadatka. Ovaj interfejs nasleđuje **`java.io.Serializable`** interfejs, što omogućuje prenos zadatka preko mreže.

Rezultat rada se dojavljuje klijentu tako što se aktivira odgovarajuća *listener* klasa. Ove klase moraju da implementiraju **`AgentListener`** interfejs.

Svaki *listener* prosleđuje klijentu rezultat rada preko klase koja implementira **`AgentEvent`** interfejs. Slika 5.4 daje pregled klasa i interfejsa potrebnih za izvršenje zadatka.



Slika 5.4: Dijagram klasa podsistema za upravljanje zadacima

Ako je izvršenje zadatka programsko, tj. izvršenjem **`execute()`** metode, onda postoje tri tipa događaja koje se automatski šalju klijentu. To su:

1. **`AgentEvent.START_EVENT`** – pre izvršenja zadatka,
2. **`AgentEvent.PERFORMING_EVENT`** – u toku izvršenja zadatka i
3. **`AgentEvent.PERFORMED_EVENT`** – po završetku izvršenja zadatka.

Ako agent pošalje poruku tipa **`AgentEvent.PERFORMING_EVENT`** u toku svog izvršenja, to znači da će njegov zadatak više puta izvršavati. U tom slučaju, potrebno je da agent pamti dokle je stigao u izvršenju. Za to je predviđen mehanizam pamćenja stanja i signalizacije završetka posla – atribut **`finished`** u klasi **`AgentResult`** (čiji objekti se prosleđuju kao rezultat izvršenja zadatka).

Kod izvršenja zadatka upotrebom KQML poruka, agent šalje KQML poruke klijentu, odnosno ne postoje predefinisani događaji, zato što KQML specifikacija definiše slobodnu razmenu poruka koje ne spadaju u posebne tipove događaja.

5.5.1 Primer izvršenja zadatka

Klijentska aplikacija počinje interakciju sa agentskim sistemom kreiranjem **`FacilitatorProxy`** klase:

```
FacilitatorProxy fp = new FacilitatorProxy();
```

Zatim se kreira objekat odgovarajuće klase zadatka i odgovarajući *listener* i prosleđuju se **`execute()`** metodi klase **`FacilitatorProxy`**:

```
SimpleAgentListener al= new SimpleAgentListener(this);
fp.execute(task, al);
```

Nakon zadavanja zadatka, aplikacija nastavlja sa radom, a može i da sačeka rezultat izvršenja:

```
synchronized (this) {
  try {
    wait();
  } catch(InterruptedException ex) {
    System.err.println("Thread interrupt error: " +
                       ex.getMessage());
  }
}
```

Klasa koja implementira **`AgentListener`** interfejs služi za osluškivanje rezultata rada (listing 5.7).

```
class SimpleAgentListener implements AgentListener {
  private TestAgent ta;

  public SimpleAgentListener(TestAgent ta) {
    this.ta = ta;
  }

  public void actionStarted(AgentEvent e) {
    System.out.println("%%% Startovao posao: " +
                       e.getResult().getContent());
  }
  public void actionPerforming(AgentEvent e) {
    System.out.println("%%% Radim posao: " +
                       e.getResult().getContent());
  }
  public void actionPerformed(AgentEvent e) {
    System.out.println("%%% Uradio posao: " +
                       e.getResult().getContent());
    synchronized (ta) {
      ta.notify();
    }
  }
}
```

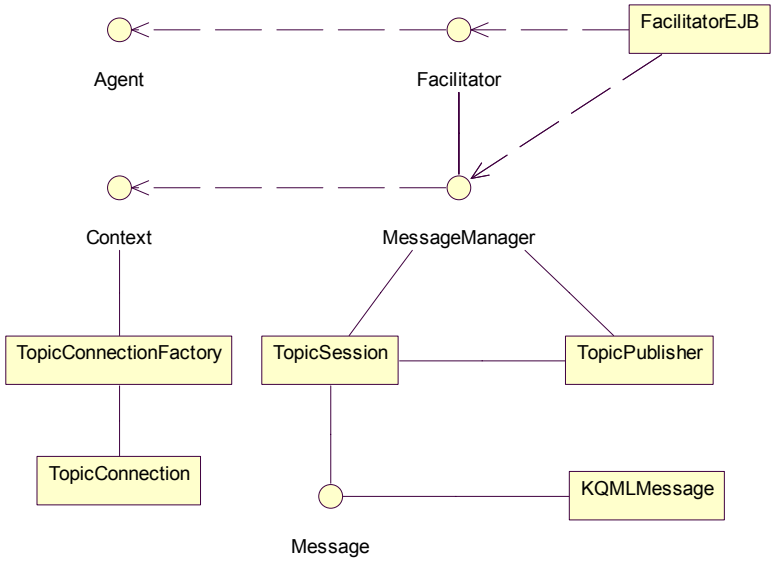
Listing 5.7: Klasa koja implementira **`AgentListener`** interfejs

U listingu 5.7 se vidi da se klijentska aplikacija po završetku zadatka deblokira metodom **`notify()`**. Ovaj koncept nije neophodan, i klijentska aplikacija ne mora da čeka izvršenje agentskog zadatka **`wait()`** metodom, već može da nastavi izvršenje, a rezultate rada će prikupljati uz pomoć *listener*-a, asinhrono.

Klasa **`FacilitatorProxy`** kreira objekat klase **`FacilitatorEJB`**, koja predstavlja EJB komponentu koja implementira agentski centar. Ova klasa regrutuje odgovarajućeg agenta za izvršenje zadatka i prosleđuje mu zadatak, a zatim vraća klijentu rezultate rada preko objekata klase **`AgentEvent`**.

5.6 Razmena poruka između agenata

Razmena poruka između agenata se svodi na razmenu KQML poruka. Poruke se razmenjuju upotrebom *MessageManager* komponente. Ova komponenta koristi usluge JMS sistema [JMS], tako da dodatno programiranje nije potrebno, a sva komunikacija je standardizovana kao što je prikazano na slici 5.5.



Slika 5.5: Razmena KQML poruka upotrebom JMS sistema

Slika 5.5 ilustruje primenu JMS tehnologije za razmenu KQML poruka. Kada agent šalje KQML poruku drugom agentu, tada se KQML poruka ugrađuje u JMS poruku, uz potrebne informacije o odredišnom agentu. JMS poruka se šalje ka agentskim centrima koji su se prijavili na ovaj servis, a agentski centar koji sadrži agenta kome je poruka upućena izdvaja iz tela JMS poruke ugrađenu KQML poruku. Ova poruka se prosleđuje agentu pozivanjem njegove metode **`onKQMLMessage()`**.

Poruke se razmenjuju prosleđivanjem objekta klase **`KQMLMessage`** između učesnika dijaloga. Razmena KQML poruka je asinhrona. To je takođe postignuto upotrebom JMS servisa, tako što se KQML poruka smešta unutar JMS poruke.

Ako je potrebno slati KQML poruke, tada se poziva metoda **`sendKQMLMessage()`** interfejsa **`Facilitator`**:

```
Facilitator.sendKQMLMessage(new KQMLMessage(
         "recommend-one",
         message.getReceiver(),  // sender
         message.getSender(),    // receiver
          "", "",                // from, to
         message.getReplyWith(), // in-reply-to
         // reply-with

         "id"+((Facilitator)Facilitator).getUniqueId(),
         "java",                 // language
         "math",                 // ontology
         new FactorielTask(new Integer(value-1))));
```

Ova metoda je asinhrona – program nastavlja sa izvršenjem i ne čeka rezultat slanja. Poruka se smešta u red za slanje, a odatle se JMS sistemom prenosi do odredišta. Na odredištu se identifikuje primalac poruke i proziva njegova metoda **`onKQMLMessage():`**

```
public void onKQMLMessage(KQMLMessage message) {
  System.out.println("AGENT " + message.getReceiver() +
                     " received a message: " +
                     message.getCommand() +
                     ", :" + message.getContent() +
                     " from: " + message.getSender());
}
```

Slanje KQML poruka je implementirano unutar *MessageManager* komponente (listing 5.8).

```
public void sendKQMLMessage(Object msgId) throws
KQMLMessageException {
  try {
    KQMLMessage message = getKQMLMessage(msgId);
    if (message.getReceiver().equals(// ovom Facilitatoru
    "Facilitator" + Facilitator.getConnectionManager().getMyAddr()
                                    )) {
      parse(message);
    } else if (message.getReceiver().equals(
              "proxy"+
              Facilitator.getConnectionManager().getMyAddr())) {
      try { // ka FacilitatorProxy
        ObjectMessage oMessage = session.createObjectMessage();
        oMessage.setStringProperty("sender", "Facilitator");
        oMessage.setStringProperty("receiver",
                  "FacilitatorProxy"+
                  Facilitator.getConnectionManager().getMyAddr());
        oMessage.setIntProperty("type",
        AgentEvent.PERFORMED_EVENT);
        String id = message.getInReplyTo();
        oMessage.setStringProperty("id", id);
        oMessage.setObject((Serializable)message.getContent());
        publisher.publish(oMessage);
      } catch (JMSException ex) {
        ex.printStackTrace();
        throw new KQMLMessageException(
          "FATAL ERROR. Could not send JMS message:" +
          ex.getMessage());
      }
    } else if (Facilitator.getAgentManager().exists(
                                        message.getReceiver())) {
      // lokalni agent
      Facilitator.getAgentManager().getAgent(
      message.getReceiver()).onKQMLMessage(message, Facilitator);
    } else {
      // kod drugog Facilitator-a
      Facilitator.getConnectionManager().sendKQMLMessage(message);
    }
  } catch (RemoteException e) {
    e.printStackTrace();
    throw new KQMLMessageException(
      "FATAL ERROR. Could not communicate with FACILITATOR: " +
      e.getMessage());
    }
  // uklonimo poruku
  messageRepository.remove(msgId);
}
```

Listing 5.8: Implementacija dispečera poruka u *MessageManager* komponenti

Svaka poruka se prilikom slanja stavlja u red čekanja, a *MessageManager* izvlači poruke iz reda i šalje ih na odredište. Pri tom proverava da li se poruka šalje klijentu, samom *Facilitator*-u, lokalnom agentu ili udaljenom agentu. U sva četiri slučaja, poruka se smešta u JMS poruku i šalje na odredište. Tamo se izvlači iz JMS poruke i distribuira dalje.

5.7 Uređenje međusobnih odnosa između agentskih centara

Komponenta *ConnectionManager* se stara o vezi agentskog centra sa drugim agentskim centrima. Svaki agentski centar poseduje spisak svih podređenih agentskih centara. Kreiranje spiska je prikazano na listingu 5.9.

```
private void putInCollection(Vector collection,
                             Enumeration values) {
  while (values.hasMoreElements())
    collection.addElement(values.nextElement());
}

public Vector getAllFacilitators() {
  Vector collection = new Vector();
  try {
    Enumeration list;
    // svi podređeni agentski centri
    list = subagents.elements();
    while (list.hasMoreElements()) {
      String addr = (String)list.nextElement();
      Facilitator external = new ProxyConnector(addr);
      putInCollection(collection,
        external.getAllFacilitatorsExceptThis(
          myAddr).elements());
    }
    // sada root
    if (rootAddr != null && !rootAddr.equals("") &&
        (connected))
      putInCollection(collection,
        root.getAllFacilitatorsExceptThis(
          myAddr).elements());
    } catch (Exception ex) {
      ex.printStackTrace();
    }
    return collection;
  }
```

Listing 5.9: Kreiranje spiska podređenih agentskih centara

Svaki agentski centar ima spisak podređenih agentskih centara i jedan roditeljski agentski centar. Agentski centar na vrhu hijerarhije nema roditeljski agentski centar. Prijavljivanje roditeljskom agentskom centru je prikazano na listingu 5.10.

```
public SimpleConnectionManager() {
  rootAddr = XMLAgentParser.getFacilitatorSubtagProperty(
             "address", "root").trim();
  if (rootAddr != null && !rootAddr.equals("") &&
      (!connected) ) {
    root = new ProxyConnector(rootAddr);
    myAddr = XMLAgentParser.getFacilitatorProperty(
             "addr").trim();
    try {
      root.register(myAddr);
      connected = true;
    } catch (Exception ex) {
      System.out.println("Connection manager failed.");
      ex.printStackTrace();
    }
  }
}
```

Listing 5.10: Registracija kod roditeljskog agentskog centra

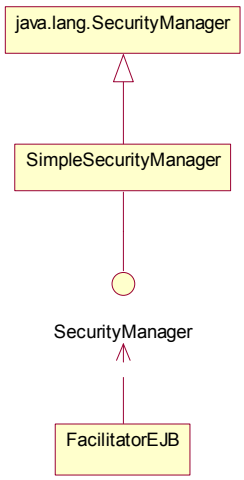
Prilikom inicijalizacije, agentski centar kontaktira roditeljski centar i prijavljuje se kod njega kao podređeni. Ovim je formirana hijerarhijska organizacija agentskih centara.

5.8 Sigurnosni mehanizmi

Sigurnost je veoma važan aspekt rada agentskog okruženja. Kao što je već rečeno, sigurnost se ne odnosi samo na komunikaciju između agenata, već i na sigurnost podataka unutar agenata, sigurnost njihovog prenosa iz jednog agentskog okruženja u drugi, sigurnost agenata od neautorizovane upotrebe, kao i sigurnost agentskog okruženja od malicioznih agenata. Svi ovi koncepti se mogu ostvariti upotrebom sigurnosnih mehanizama unutar postojećih aplikacionih servera.

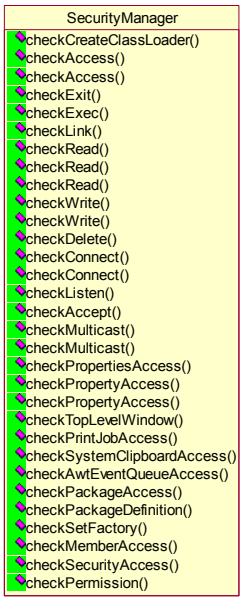
Poseban problem predstavlja zaštita od malicioznih agenata. Problem zaštite od malicioznih agenta se svodi na zaštitu lokalnih resursa i fajl sistema. Sigurnosni menadžer, implementiran u J2EE aplikacionom serveru, može biti dizajniran tako da može da vrši proveru pristupa lokalnom fajl sistemu i lokalnim resursima. Da bi ovo postigao, sigurnosni menadžer mora da nasledi klasu `java.lang.SecurityManager`, koja omogućuje kontrolu pristupa resursima pre njihovog korišćenja. Ova

klasa omogućuje kontrolu pristupa mrežnim resursima, kreiranju objekata (konstrukcija *Class Loader*-a), kontrolu pristupa lokalnom fajl sistemu, pristupa sistemskim atributima, klipbordu i programskim nitima (slika 5.6). Ovaj pristup je opcion. Dovoljno je samo naslediti **`java.lang.SecurityManager`** klasu.



Slika 5.6: Dijagram klasa sigurnosnog menadžera

Nasleđivanjem **`java.lang.SecurityManager`** klase, potrebno je redefinisati odgovarajuće metode za pristup resursima. Slika 5.7 prikazuje sve metode ove klase.



Slika 5.7: Metode klase **`java.lang.SecurityManager`**

Metode od interesa za zaštitu lokalnog fajl sistema su: **`checkRead()`**, **`checkWrite()`** i **`checkDelete()`**. Metode od interesa za zaštitu pristupa mrežnim resursima su: **`checkAccept()`**, **`checkConnect()`**, **`checkListen()`** i **`checkSetFactory()`**.

5.9 Direktorijum servisa

Upravljanje servisima podrazumeva da odgovarajući menadžer kreira, oslobađa i pronalazi odgovarajući servis. Servisi su implementirani kao elementi EJB komponenti, u smislu da se objekti koji reprezentuju servise smeštaju u odgovarajuću EJB komponentu. Time je omogućeno da servisi ne zavise od konkretne implementacije EJB standarda.

Primer ugradnje servisa u EJB komponentu je prikazan na listingu 5.11.

```
public Object getService(String serviceName)
              throws RemoteException {
  Object retVal = null;
  String serviceClassName =
    XMLAgentParser.getServiceClassName(
       serviceName);
  if (serviceClassName != null) {
    ServiceHolder serviceHolder = null;
    try {
      InitialContext context = new InitialContext();
      Object homeObject =
        context.lookup("ejb/ServiceHolder");
      ServiceHolderHome home =
(ServiceHolderHome)PortableRemoteObject.narrow(
      homeObject, ServiceHolderHome.class);
      // Ovim se kreira ServiceHolder EJB komponenta.
      serviceHolder =
(ServiceHolder)PortableRemoteObject.narrow(
      home.create(serviceClassName), ServiceHolder.class);
    } catch(Exception e) {
      e.printStackTrace();
      throw new RemoteException("FATAL ERROR: " +
        e.getMessage());
    }
  }
  return retVal;
}
```

Listing 5.11: Ugradnja servisa u EJB komponentu

Listing 5.11prikazuje pronalaženje odgovarajućeg servisa. Na osnovu naziva servisa, traži se naziv klase servisa. Ako takav servis postoji, kreira se EJB komponenta koja će čuvati servis. Prilikom kreiranja EJB komponente, njoj se prosleđuje naziv klase servisa kojeg će sadržati. Ova komponenta prilikom svoje konstrukcije kreira i objekat servisa, kao što je prikazano na listingu 5.12

```
public void ejbCreate(String serviceClassName)
            throws CreateException {
  // Kreiramo servis koji je dodeljen ovoj komponenti.
  try {
    service = (Service)Class.forName(
                        serviceClassName).newInstance();
  } catch (Exception ex) {
    ex.printStackTrace();
    throw new CreateException("Could not create service: "+
      service ClassName + ": " + ex.getMessage());
  }
}
```

Listing 5.12 Kreiranje objekta servisa unutar EJB komponente

5.10 Agenti

Agenti su klase koje implementiraju **`Agent`** interfejs. Kreirani agent se smešta u EJB komponentu, čime su omogućene dve stvari: implementacija agenata ne zavisi od implementacije EJB komponenti i moguće je lako prenošenje konteksta agenta. Prenošenje konteksta agenta se odvija tokom prenosa agenta iz jednog agentskog centra u drugi. To se izvodi tako što se objekat koji reprezentuje agenta serijalizuje i prebaci na odredišni agentski centar. Tamo se poziva metoda **`reprogramAgent()`**, koja je prikazana u listingu 5.13

```
public void reprogramAgent(Object agentId, Agent agent) {
  AgentHolder agentHolder = get(agentId);
  try {
    agentHolder.setAgent(agent);
  } catch (Exception ex) {
    ex.printStackTrace();
  }
}
```

Listing 5.13 Prenos konteksta agenta

Prilikom prenosa agenta, na odredišnoj strani se regrutuje jedan agent koji je u stanju da reši zadatak. U EJB komponentu koja sadrži tog agenta, smešta se kontekst prenesenog agenta. Pošto je kontekst serijalizabilan, on je rekonstruisan na odredištu i prosleđen kao parametar metodi **`reprogramAgent()`**. Ova metoda smešta rekonstruisan objekat agenta u svoj atribut i time je prenos okončan, a agent u potpunosti rekonstruisan.

5.11 Karakteristike proširivog agentskog okruženja

Proširivo agentsko okruženje predstavlja agentsko okruženje bazirano na tehnologiji distribuiranih komponenti koje podržava FIPA specifikaciju. Opisana implementacija ovog okruženja je bazirana na J2EE tehnologiji. Ovako implementirano okruženje ima sledeće karakteristike:

- podržava mobilnost agenata upotrebom EJB tehnologije,
- omogućuje razmenu KQML poruka upotrebom JMS tehnologije,
- podržava direktorijume agenata i servisa upotrebom JNDI tehnologije,
- omogućava upotrebu sigurnosnih mehanizama po JCE tehnologiji.

Osim navedenih karakteristika, ovo okruženje podržava koncept *plug-in* menadžera, čime je omogućeno da se pojedinačna zaduženja agentskog centra delegiraju pojedinim menadžerima. Menadžeri su softverske komponente čije postojanje i upotreba nisu statički određeni, već se mogu dinamički menjati, a odabir konkretnog menadžera se svodi na podešavanje unutar konfiguracione datoteke.

Ovo okruženje nudi koncept mobilnih zadataka koji omogućavaju da se u same zadatke ubaci programski kod, a da agenti u tom slučaju predstavljaju okruženja za rešavanje zadataka.

U ovom okruženju koristi se koncept međusobnog uređenja odnosa između agentskih okruženja. Ovim konceptom omogućeno je formiranje agentske mreže, čiji čvorovi su agentski centri, a agentima je omogućena međusobna komunikacija i mobilnost.

Predloženi model agentskog okruženja može se implementirati različitim tehnologijama distribuiranih softverskih komponenti. Takođe, upotrebom *plug-in*-ova omogućeno je da se određeni segmenti agentskog okruženja mogu implementirati u različitim softverskim okruženjima. Sa druge strane, trenutno nije podržan koncept agentskog jezika. Agentski jezik bi omogućio programiranje agenata u jeziku višeg nivoa apstrakcije, čime bi se programer rasteretio od implementacionih detalja. Proširenje agentskog okruženja konceptom agentskog jezika bi se svelo na dodavanje samo još jednog menadžera.

Poglavlje 6

Verifikacija agentskog okruženja na Bibliotečkom Informacionom Sistemu BISIS

Verifikacija agentskog okruženja će biti izvedena u Bibliotečkom informacionom sistemu BISIS [Surla00]. Softverski sistem BISIS služi za podršku bibliotečkom poslovanju, a projektovan je i implementiran primenom objektnog pristupa, što obuhvata UML za specifikaciju i programski jezik Java za implementaciju. Sistem u potpunosti podržava Unicode [Unicode] standard, što ga čini posebno pogodnim za korišćenje u višejezičnim okruženjima. Koristi se relacioni sistem za upravljanje bazama podataka. Sistem je tako koncipiran da ne zavisi od konkretnog SUBP. Softverski sistem BISIS je baziran na klijent-server arhitekturi i čine ga BISIS serveri, korisnici preko Interneta i BISIS klijenti.

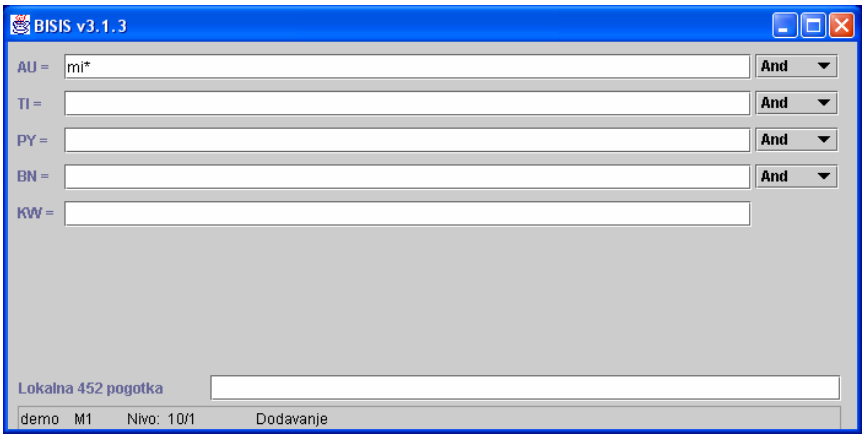
BISIS serveri omogućavaju pretragu, izmenu i unos novih zapisa bibliografske građe. Skup BISIS servera povezanih računarskom mrežom čini bibliotečku mrežu.

Korisnici se dele na dve kategorije:

1. korisnici u pretraživanju građe i
2. bibliotekari.

Korisnici u pretraživanju građe mogu samo da pretražuju bazu zapisa i za tu svrhu koriste svoj web brauzer.

Bibliotekari održavaju bazu zapisa i za tu svrhu koriste BISIS klijent [Vidaković98b, Vidaković98a, Vidaković02c]. BISIS klijent je Java aplikacija napisana za potrebe bibliotekara. Ova aplikacija omogućava pretragu, modifikovanje i unos novih bibliografskih jedinica. BISIS klijent je povezan na lokalnu bazu BISIS sistema kojem pripada. To znači da se svaki zadati upit izvršava unutar lokalne baze zapisa. Primer rezultata pretrage lokalne baze je dat na slici 6.1.



Slika 6.1: Rezultat lokalne pretrage

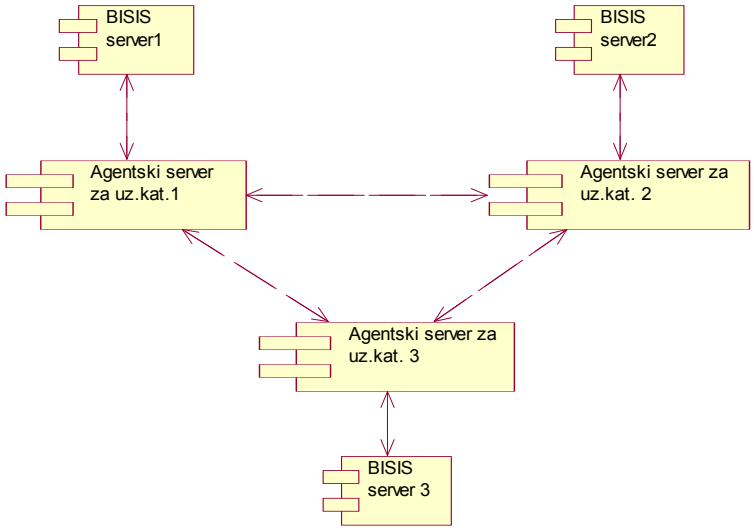
Osim pretraživanja lokalne baze podataka, BISIS sistem podržava uzajamnu katalogizaciju [Milosavljević00a]. Uzajamna katalogizacija predstavlja mogućnost razmene podataka između dve biblioteke na nivou zapisa. Posao obrade i unosa bibliografske građe obavljaju bibliotekari na osnovu međunarodnog standarda UNIMARC [UNIMARC]. Cilj uzajamne katalogizacije je da bibliotekari mogu da preuzimaju zapise iz drugih biblioteka u cilju kompletiranja sopstvene baze zapisa u kontrolisanim uslovima. To znači da treba obezbediti bibliotekaru mogućnost da pretražuje druge baze zapisa, odabere zapise i memoriše ih u sopstvenoj lokalnoj bazi, modifikovane u skladu sa specifičnostima biblioteke kojoj lokalna baza pripada.

Server za uzajamnu katalogizaciju [Vidaković01] je aplikacija koja se instalira u okviru BISIS sistema i koja omogućuje svim prijavljenim klijentima pristup lokalnoj bazi podataka. Za prijavljene klijente, lokalna baza servera predstavlja udaljenu bazu zapisa. Iz bezbednosnih razloga, moguća je samo pretraga i preuzimanje zapisa preko servera za uzajamnu katalogizaciju. Ovim je omogućeno da klijenti pretražuju i preuzimaju zapise iz onih udaljenih baza na kojima je instaliran server za uzajamnu katalogizaciju. Međutim, da bi pronašli neki zapis unutar bibliotečke mreže, bibliotekari pre svega moraju da znaju na kojim je sve serverima bibliotečke mreže podignut server za uzajamnu katalogizaciju. Zatim, bibliotekari moraju da se prijave na svaki od poznatih servera i da tamo zadaju isti upit.

Ako neki zapis postoji na više servera, potrebno je pronaći najkvalitetniji i njega preuzeti. Određivanje kvaliteta se svodi na pojedinačno preuzimanje zapisa, njegov pregled i ocenjivanje od stane bibliotekara.

Iz navedenog se vidi da server za uzajamnu katalogizaciju pruža ogromne mogućnosti za bibliotekare, ali i da je potrebno sistem unaprediti na takav način da se proces uzajamne katalogizacije pojednostavi i automatizuje.

Za ovakve potrebe, kao predlog rešenja uzajamne katalogizacije, razvijen je **agentski server za uzajamnu katalogizaciju**. Na taj način, ilustrovano je pretraživanje bibliografskih zapisa u bibliotečkoj mreži i ocena kvaliteta zapisa. Ovaj agentski server za uzajamnu katalogizaciju je implementiran u okviru agentskog okruženja prikazanog u poglavlju 5. Mreža BISIS servera sa agentskim serverima za uzajamnu katalogizaciju predstavlja okruženje u kojem je verifikovano agentsko okruženje. Primer arhitekture mreže od tri agentska servera za uzajamnu katalogizaciju je prikazana na slici 6.2.



Slika 6.2: Dijagram komponenti mreže agentskih servera za uzajamnu katalogizaciju

Na slici 6.2 je prikazan primer mreže od tri agentska servera. Dodavanje novog servera u mrežu je automatizovano postupkom registracije agentskog centra u kome je server implementiran. Ovim se novi agentski server priključuje u mrežu bez dodatnih procedura.

Svaki agentski server je reprezentovan skupom bibliotečkih agenata kojima se delegiraju bibliotečki poslovi. Ovi agenti su u stanju da pristupe bibliotečkim zapisima preko bibliotečkog servisa za pristup BISIS serverima. Na ovaj način, agentski server je direktno povezan sa BISIS serverom. Agentska okruženja unutar kojih se izvršavaju bibliotečki agenti formiraju mrežu koja omogućuje ovim agentima da prelaze iz jednog agentskog servera u drugi, da pronalaze odgovarajuće agente na drugim serverima i da međusobno komuniciraju.

Bibliotečki agenti će demonstrirati sledeće funkcije: pretraživanje zapisa po bibliotečkoj mreži, ocenu kvaliteta zapisa i inteligentnu raspodelu opterećenja.

Pretraživanje zapisa po bibliotečkoj mreži se oslanja na mobilnost agenata. Agent koji se angažuje da pronađe zapis, obilazi bibliotečku mrežu i na svakom čvoru mreže pretražuje bazu zapisa upotrebom servisa za rad sa BISIS sistemom. Ovaj servis omogućuje agentima pretragu zapisa, njihovo preuzimanje i ocenjivanje.

Ocena kvaliteta zapisa se takođe zasniva na agentima. Na svakom serveru za uzajamnu katalogizaciju postoje specijalizovani agenti za ocenu kvaliteta koji će biti aktivirani KQML porukom koja sadrži identifikator zapisa čiji se kvalitet ocenjuje. Rezultat rada ovih agenata se putem KQML poruke prosleđuje klijentima koji su ih angažovali.

Konačno, agentski server za uzajamnu katalogizaciju poseduje mogućnost inteligentne raspodele opterećenja. To znači da postoji mogućnost izvršenja zadatka u distribuiranom okruženju gde će prvi agent koji pronađe zadovoljavajuće rešenje inicirati završetak pretrage kod drugih agenata, čime će se sprečiti dalje pretraživanje, a samim tim i dalje opterećivanje resursa na mreži.

Verifikacija agentskog okruženja agentskim serverom za uzajamnu katalogizaciju, dakle, demonstrira mobilnost agenata, razmenu KQML poruka, upotrebu servisa za pristup BISIS sistemu, sinhronizaciju rada agenata i mogućnost angažovanja agenata sa agentskog okruženja koje nije lokalno. Svi elementi implementirane arhitekture će biti detaljno opisani u narednim primerima. Takođe, biće prikazan i način povezivanja agentskog okruženja sa drugim Java aplikacijama (u ovom slučaju, BISIS klijent aplikacija).

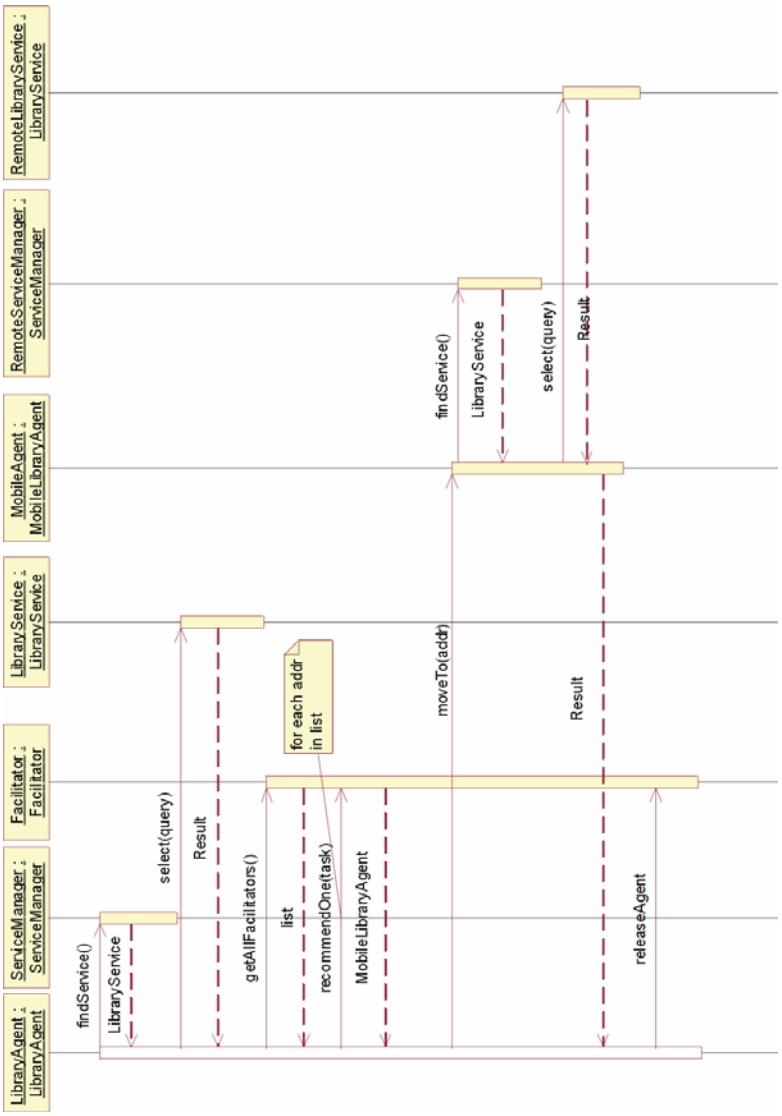
6.1 Pretraživanje bibliotečke mreže

Pretraživanje zapisa na bibliotečkoj mreži podrazumeva da se agentima zada upit koji će oni izvršavati na svim serverima bibliotečke mreže. Ovim se omogućava automatizovano pretraživanje svih servera bibliotečke mreže, a ne ručno pretraživanje svakog od njih, od strane bibliotekara. Za ovu priliku razvijena su dva agenta reprezentovana sledećim klasama:

1. **`LibraryAgent`** klasa i
2. **`MobileLibraryAgent`** klasa.

`LibraryAgent` klasa implementira funkcionalnost agenta koji prvo pretražuje lokalnu bazu zapisa, a zatim regrutuje **`MobileLibraryAgent`** agenta i zadaje mu redom servere koje ovaj mora da poseti i kod kojih mora da realizuje zadati upit. Slika 6.3 prikazuje dijagram sekvenci koji ilustruje postupak pretraživanja bibliotečke mreže.

Po zadavanju zadatka, **`LibraryAgent`** prvo pretražuje lokalnu bazu. To postiže tako što zahteva servis za pretraživanje biblioteke. Ako dobije traženi servis, njime pretražuje lokalnu bazu. Zatim od agentskog centra zahteva spisak svih agentskih centara u sistemu. Nakon toga, od agentskog centra traži jednog mobilnog agenta (klasa **`MobileLibraryAgent`**). Ovom agentu će zadati da obiđe sve bibliotečke servere iz spiska i da u svakom od njih pretraži bazu zapisa. Rezultat rada mobilnog agenta se akumulira, spaja sa rezultatom pretraživanja lokalne baze i vraća onome ko je angažovao agenta. Listing 6.1 prikazuje deo implementacije **`LibraryAgent`** klase, koji se odnosi na pretraživanje.



Slika 6.3: Dijagram sekvenci izvršenja pretrage biblioteče mreže

```
/**
 * Agent koji pretražuje biblioteke.
 */
public class LibraryAgent implements Agent {
  /** Spisak svih agentskih centara osim matičnog
   *  na koji ce se agent preneti da bi pronašao
   *  sve knjige koje zadovoljavaju
   *  zadati kriterijum (zadat u LibraryTask klasi).
   */
  Vector Facilitators;

  /** Vektor konačnih rezultata pretrage. */
  Vector results;

  /** Korak izvršenja, koji je jednak ukupnom
   *  broju agentskih centara
   *  u sistemu.
   */
  int step = -1;

  /** ID regrutovanog mobilnog agenta. */
  Object mobileId;

  /**
   * Izvršava zadati zadatak.
   * @param task Zadatak koji se izvršava.
   * @return Rezultat rada.
   */
  public AgentResult execute(AgentTask task,
    EJBObject Facilitator, Object agentId) {
    try {
      MobileLibraryTask t = new MobileLibraryTask(task.getCommand(),
        new MobileLibraryTaskParams("",
          agentId.toString()));
      step++;
      if (step == 0) { // prvi put pozvan
        // pripremimo rezultate rada;
        result = new AgentResult();
        // napravimo spisak dostupnih agentskih centara
        Facilitators =
((Facilitator)Facilitator).getAllFacilitators();
        // sada potrazimo kod lokalnog agentskog centra...
        Object serviceId =
((Facilitator)Facilitator).getServiceManager().findService(
 "LibraryService");
```

Listing 6.1: Deo **LibraryAgent** klase

```
        // uzmemo servis na osnovu njegovog id-a
        Service service =
((Facilitator)Facilitator).getServiceManager().getService(
  serviceId);
        // izvršimo pretragu biblioteke
        int hits = ((LibraryService)service).select(
          task.getCommand().toString());
        // oslobodimo servis posle upotrebe
((Facilitator)Facilitator).getServiceManager().returnService(
  serviceId);

        results = new Vector();
        // ubacimo rezultat pretrage lokalne baze
        results.addElement(
       new LibraryServiceReturnValue(
      ((Facilitator)Facilitator).getConnectionManager().getMyAddr(),
       hits));
      }
      if (step < (Facilitators.size()-1)) {
        // za svaki agentski centar u mreži
        // regrutujemo jednog mobilnog library agenta
        mobileId  = ((Facilitator)Facilitator).recommendOne(t);

        String destAddr = (String)Facilitators.elementAt(step);
        t.setParams(new MobileLibraryTaskParams(
          destAddr, agentId.toString()));
        // iniciramo izvršenje zadatka kod mobilnog agenta
        results.addElement(
          ((Facilitator)Facilitator).execute(mobileId,
            t).getContent());
        result.setFinished(false);
        // oslobodimo agenta
        ((Facilitator)Facilitator).releaseAgent(t,
         mobileId);
      } else {
          result.setFinished(true);
      }

      result.setContent(results);
      return result;
    } catch (Exception ex) {
      ex.printStackTrace();
    }
    return null;
  }
}
```

Listing 6.1: Deo **`LibraryAgent`** klase – nastavak

Zadatak za **MobileLibraryAgent** klasu sadrži adresu servera na koji agent mora da se prebaci i upit koji tamo treba da postavi. Po zadavanju zadatka, mobilni agent se prebacuje na zadatu adresu, tamo zahteva bibliotečki servis i izvršava pretragu. Rezultate rada prosleđuje matičnom **LibraryAgent** agentu, a ovaj ga zatim šalje na sledeću adresu. Postupak se ponavlja dok se ne iscrpe sve adrese. Listing 6.2 prikazuje deo listinga klase **MobileLibraryAgent**, koji se bavi pretragom:

```
/**
 *  Mobilni agent koji pretražuje biblioteke.
 */
public class MobileLibraryAgent implements Agent {
  /** Adresa agentskog centra na koji ce se agent preneti
   *  da bi pronašao sve zapise koji zadovoljavaju zadati
   *  kriterijum (zadat u LibraryTask klasi).
   */
  String destAddr;

  /**
   * Izvršava zadati zadatak.
   * @param task Zadatak koji se izvršava.
   * @return Rezultat rada.
   */
  public AgentResult execute(AgentTask task,
    EJBObject Facilitator, Object agentId) {
    try {
      // izvučemo odredišnu adresu iz zadatka
      destAddr=((MobileLibraryTaskParams)task.getParams()).destAddr;

      // preselimo se na zadatu adresu
     LibraryServiceReturnValue res =
(LibraryServiceReturnValue)((Facilitator)Facilitator).moveTo(destAddr,
agentId, task);

      // rezultat rada na odredišnoj adresi je u
      // parametru res,
      // pa ga smeštamo u AgentResult klasu
      AgentResult result = new AgentResult();
      result.setContent(res);
      result.setFinished(true);
      return result;
    } catch (Exception ex) {
      ex.printStackTrace();
    }
    return null;
  }
```

Listing 6.2: Listing dela **MobileLibraryAgent** klase

```
  /** Kada se agent prebaci na drugi server,
   *  ova metoda se izvrsava.
   */
  public Object onArrival(EJBObject Facilitator,
  Object agentId, String senderAddress, AgentTask task) {
    LibraryServiceReturnValue result = null;

    try {
      // potražimo servis za pretrazivanje biblioteke
      Object serviceId =
((Facilitator)Facilitator).getServiceManager().findService(
  "LibraryService");
      // uzmemo servis na osnovu njegovog id-a
      Service service =
((Facilitator)Facilitator).getServiceManager().getService(
  serviceId);
      // izvršimo pretragu biblioteke
      int hits =
((LibraryService)service).select(task.getCommand().toString());
      // oslobodimo servis posle upotrebe
((Facilitator)Facilitator).getServiceManager().returnService(
  serviceId);
      // vratimo rezultat pretrage
     result = new LibraryServiceReturnValue(
       ((Facilitator)Facilitator).getConnectionManager().getMyAddr(),
       hits);
    } catch (Exception ex) {
      ex.printStackTrace();
    }
    return result;
  }
}
```

Listing 6.2: Listing dela **MobileLibraryAgent** klase – nastavak

Rezultat rada **LibraryAgent** klase je objekat **AgentResult** klase, koji u sebi sadrži vektor objekata **LibraryServiceReturnValue** klase (listing 6.3).

```
public class LibraryServiceReturnValue implements
Serializable {
  public String host;
  public int hits;

  public LibraryServiceReturnValue(String host, int hits) {
    this.host = host;
    this.hits = hits;
  }

  public String toString() {
    return "Na ra\u010dunaru " + host + " ima " + hits + "
pogodaka.";
  }
}
```

Listing 6.3: Listing **LibraryServiceReturnValue** klase

BISIS klijent koji inicira agentsku pretragu dobija vektor ovakvih objekata kao rezultat pretrage. Svaki objekat sadrži adresu servera na kojem je rađena pretraga i broj pogodaka. Samu pretragu inicira korisnik zadavanjem upita. Po zadavanju upita sistem proverava da li će raditi lokalnu ili agentsku pretragu. Ako radi agentsku pretragu, kreira se objekat klase **FacilitatorProxy**, i njemu se prosleđuje objekat klase **LibraryTask**, koji sadrži upit koji će se izvršavati u bibliotečkoj mreži (listing 6.4).

```
public class SearchTask {
  MainFrame mf;
  public SearchTask(String query, MainFrame mf) {
    this.mf = mf;
    int hits;
    try {
      if (Environment.isAgent()) { // agentska pretraga
        doTheAgent(query);
      } else // ako je lokalna pretraga
        hits =  Environment.getReadTs().select(query);
    } catch (ExpressionException ex) {
      try {
        // ako nije uspela pretraga, resetuj broj pogodaka
        Environment.getReadTs().clearHits();
      } catch (Exception e) {}
      setErrorMessage(ex.toString().substring(
        ex.toString().indexOf(':')+1));
    } catch (Exception ex) {
      ex.printStackTrace();
    }
```

Listing 6.4: Klasa **SearchTask**, koja obavlja pretragu

```
  // ugasimo prozor sa 'odustani' dugmetom
  csd.setVisible(false);
  setFinished(true);
}
/ ** Metoda koja inicira agentsko pretraživanje. */
public void doTheAgent(String query) {
  // kreiramo FacilitatorProxy komponentu
  FacilitatorProxy fp = new FacilitatorProxy();
  // kreiramo zadatak na osnovu upita
  LibraryTask task = new LibraryTask(query, "");
  // pripremimo listener koji ce osluškivati
  // rezultate pretrage
  ServiceAgentListener al= new ServiceAgentListener(mf);
  // izvršimo pretragu
  fp.execute(task, al);

  // čekamo da se pretraga zavrsi...
  synchronized (mf) {
    try {
      mf.wait();
    } catch(InterruptedException ex) {
      System.err.println("Thread interrupt error: " +
        ex.getMessage());
    }
  }
  fp.stop(); // ugasimo FacilitatorProxy
}

/** Unutrašnja klasa koja implementira
 *  AgentListener interfejs.
 */
class ServiceAgentListener implements AgentListener {
  MainFrame mf;
  public ServiceAgentListener(MainFrame mf) {
   this.mf = mf;
  }
  // poziva se prilikom startovanja posla
  public void actionStarted(AgentEvent e) {
    System.out.println("%%% Startovao posao: " +
                       e.getResult().getContent());
  }
  // poziva se tokom izvršavanja posla
  public void actionPerforming(AgentEvent e) {
    System.out.println("%%% Radim posao: " +
      e.getResult().getContent());
  }
```

Listing 6.4: Klasa **SearchTask**, koja obavlja pretragu – nastavak

```
    // poziva se nakon završetka posla
    public void actionPerformed(AgentEvent e) {
      System.out.println("%%% Uradio posao: " +
                         e.getResult().getContent());
      // sačuvamo rezultat
      Environment.setAgentResult(e.getResult());
      synchronized (MainFrame.this) {
        // obavestimo klijentsku aplikaciju da je
        // gotova pretraga
        mf.notify();
      }
    }
  }
}
```

Listing 6.4: Klasa **SearchTask**, koja obavlja pretragu – nastavak

Unutrašnja klasa **ServiceAgentListener** implementira **AgentListener** interfejs i zadužena je da osluškuje rad agenta. Kada agent završi posao, on će izazvati izvršenje metode **actionPerformed()**, koja će u svom argumentu imati rezultat rada. Argument ove metode je objekat klase **AgentEvent**, koji ima metodu **getResult()** koja vraća objekat klase **AgentResult**. U ovom objektu se nalazi vektor objekata klase **LibraryServiceReturnValue**, koji sadrže adresu servera i broj pogodaka na njemu. Ovi podaci se koriste da se na ekranu prikaže rezultat rada agentskog pretraživanja (listing 6.5).

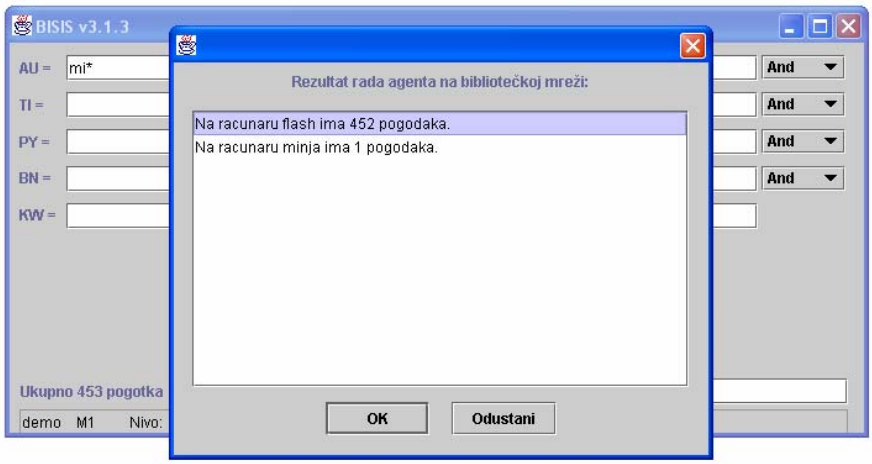
```
if (!Environment.isAgent()) { //obično pretraživanje
  try {
    hits = Environment.getReadTs().getHitCount();
  } catch (Exception ex) {}
  lComLin.setText(oblik_lokalni(hits)+" "+hits+
    " "+oblik_pogodak(hits));
} else { // agentsko pretraživanje
  // pokupimo vektor rezultata pretrage
  Vector results =
(Vector)Environment.getAgentResult().getContent();
  hits = 0;
  for (int i = 0; i < results.size(); i++) {
    // svaki pojedinačni element vektora sadrži i
    // broj pogodaka na udaljenom serveru
    LibraryServiceReturnValue item =
(LibraryServiceReturnValue)results.elementAt(i);
    hits += item.hits;
  }
```

Listing 6.5: Deo koda za prikazivanje rezultata pretrage u BISIS klijentu

```
  // u komandnoj liniji ispišemo ukupan broj pogodaka
  lComLin.setText("Ukupno "+hits+" "+oblik_pogodak(hits));
  // ispišemo pojedinačne rezultate na ekranu u vidu liste
  listDlg.setListData(results);
  listDlg.setTitle("Rezultat rada agenta na
  bibliote\u010dkoj mre\u017ei:");
  listDlg.setVisible(true);
}
```

Listing 6.5: Deo koda za prikazivanje rezultata pretrage u BISIS klijentu – nastavak

Iz BISIS klijenta se to vidi kao što je prikazano na slici 6.4:



Slika 6.4: Spisak pogodaka prilikom pretraživanja bibliotečke mreže

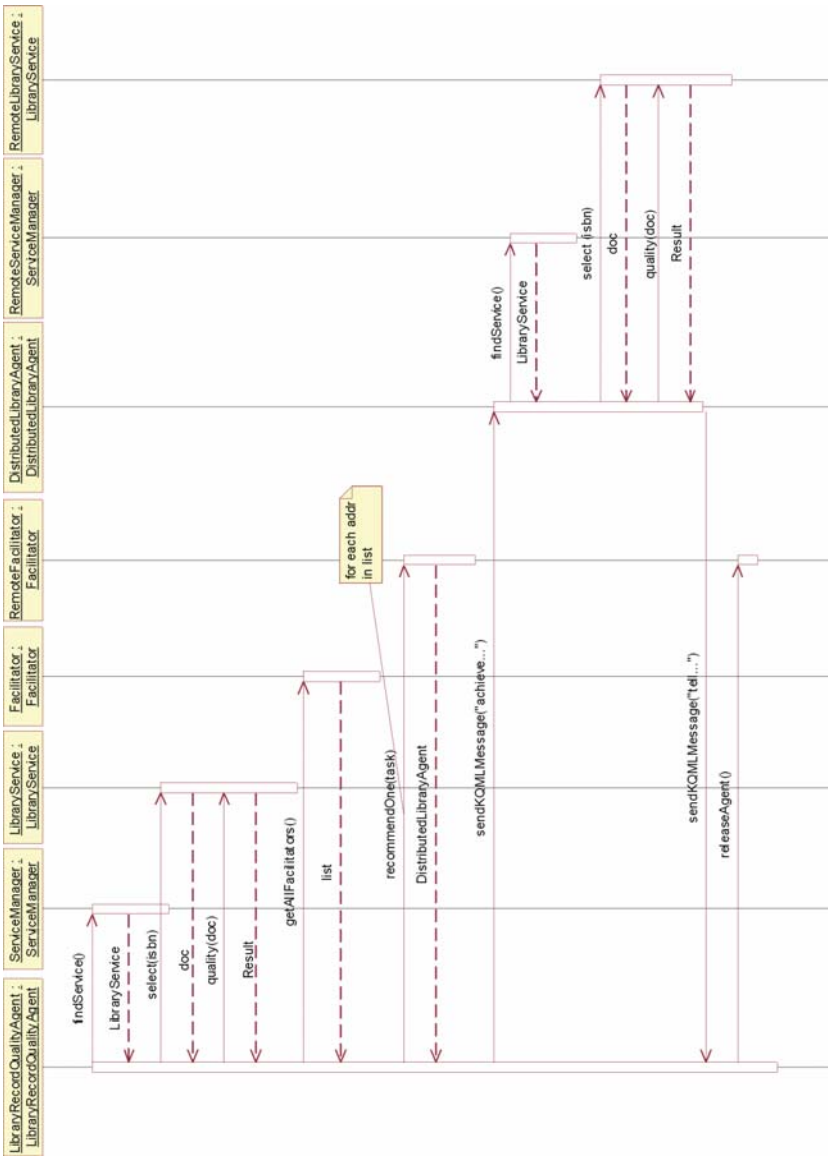
Lista pogodaka prikazuje broj pogodaka na svim prijavljenim čvorovima bibliotečke mreže. Bibliotekar može da se spoji na neki od izlistanih čvorova mreže i da nastavi pretraživanje, ili da izlista zapise koje je agent pronašao i preuzme željeni zapis. Ako se odluči za preuzimanje zapisa, poželjno je da odabere najbolji zapis. Za tu potrebu razvijen je podsistem za ocenu kvaliteta zapisa u bibliotečkoj mreži upotrebom agenata.

6.2 Ocena kvaliteta zapisa u bibliotečkoj mreži

Podsistem za ocenu kvaliteta zapisa u bibliotečkoj mreži bazira se na agentima. Osnovni zadatak je da se pronađe zapis sa najkvalitetnijom strukturom unutar bibliotečke mreže. Za realizaciju tog zadatka, angažovaće se agenti, koji će u svakom čvoru bibliotečke mreže pronalaziti zapis, ocenjivati njegov kvalitet i vraćati rezultate bibliotekaru.

Za ovu priliku su razvijena dva agenta i implementirana u klasama **`LibraryRecordQualityAgent`** i **`DistributedLibraryAgent`** klasa.

`LibraryRecordQualityAgent` klasa implementira agenta koji ocenjuje kvalitet zapisa na lokalnoj bazi podataka, a zatim pronalazi agente u celoj bibliotečkoj mreži i svakom od njih daje isti zadatak – ocenu kvaliteta zadatog zapisa. Angažovani agenti po čvorovima bibliotečke mreže su implementirani u **`DistributedLibraryAgent`** klasi. Slika 6.5 prikazuje dijagram sekvenci za ocenu kvaliteta zapisa u bibliotečkoj mreži.



Slika 6.5: Dijagram sekvenci ocenjivanja kvaliteta zapisa

Proces ocenjivanja kvaliteta započinje angažovanjem agenta koji je implementiran klasom **`LibraryRecordQualityAgent`**. Angažovanje agenta podrazumeva prosleđivanje zadatka u okviru kojeg se prosleđuje jedinstveni identifikator zapisa čiji kvalitet treba oceniti, na primer, ISBN broj. Ovaj broj je jedinstven za svaki zapis i jednoznačno ga određuje.

Agent započinje svoj rad pronalaženjem bibliotečkog servisa. Po pronalaženju bibliotečkog servisa, njemu se zadaje da pronađe zapis sa zadatim ISBN brojem i zatim se inicira ocena kvaliteta zapisa. Ocenu kvaliteta zapisa obavlja bibliotečki servis u skladu sa definisanim skupom UNIMARC polja i potpolja koje svaki zapis mora da ima. Ocena se vraća agentu, a ovaj od lokalnog agentskog centra traži listu svih agentskih centara u mreži. Agent zatim prolazi kroz listu i od svakog udaljenog agentskog centra zahteva jednog agenta za ocenjivanje kvaliteta zapisa (implementiran u **`DistributedLibraryAgent`** klasi). Ako ga dobije, šalje mu KQML poruku u kojoj mu zadaje da pronađe i oceni kvalitet zapisa, identifikovanog preko ISBN broja.

Udaljeni agent po prijemu KQML poruke pronalazi bibliotečki servis i uz pomoć njega pronalazi zapis. Po pronalaženju zapisa, agent inicira ocenjivanje kvaliteta zapisa i vraća rezultat glavnom agentu u vidu KQML poruke. Glavni agent prikuplja sve rezultate obrade i vraća klijentu konačan rezultat. Listing 6.6 ilustruje deo koda **`LibraryRecordQualityAgent`** klase koji se bavi ocenom kvaliteta zapisa.

```
/** Spisak agentskih centara u bibliotečkoj mrezi. */
Vector Facilitators;
/** Konačan rezultat će biti vektor pojedinačnih rezultata.
 */
Vector results;
/** Lokalni rezultat. */
AgentResult result;
/** Poruka na koju se čeka se snima u ovaj atrubut. */
KQMLMessage mess;
```

Listing 6.6: Deo koda **`LibraryRecordQualityAgent`** klase koji se bavi ocenjivanjem kvaliteta zapisa

```
/** Izvršava zadati zadatak. */
public AgentResult execute(AgentTask task,
  EJBObject Facilitator, Object agentId) {
  try {
      // pripremimo zadatak za udaljenog agenta
      DistributedLibraryTask t = new DistributedLibraryTask(
        task.getCommand());
      // pripremimo rezultate rada
      result = new AgentResult();
      // napravimo spisak dostupnih agentskih centara
      Facilitators = ((Facilitator)Facilitator).getAllFacilitators();
      // sada potražimo kod lokalnog agentskog centra...
      Object serviceId =
       ((Facilitator)Facilitator).getServiceManager().findService(
         "LibraryService");
      results = new Vector();
      // uzmemo servis na osnovu njegovog id-a
      Service service =
        ((Facilitator)Facilitator).getServiceManager().getService(
          serviceId);
      // izvučemo ISBN broj iz klase zadatka
      String isbn = task.getCommand().toString();
      // prilagodimo ISBN broj pretrazi bibliotečke mreze
      StringTokenizer st = new StringTokenizer(isbn, "-");
      String isbnQuery = "BN=";
      while (st.hasMoreTokens()) {
        String s = st.nextToken();
        if (st.hasMoreTokens())
          isbnQuery += s + " [W] BN=";
        else
          isbnQuery += s;
      }
      // izvršimo pretragu biblioteke
      int hits = ((LibraryService)service).select(isbnQuery);
      if (hits == 1) {
        // pokupimo ID zapisa
        int docId = ((LibraryService)service).getDocId(1);
        // pokupimo zapis
        String doc = ((LibraryService)service).getDoc(docId);
        // ocenimo kvalitet zapisa
        int quality = ((LibraryService)service).quality(doc);
        // ubacimo lokalni rezultat u konačan rezultat
       results.addElement(new RecordQualityReturnValue(
         task.getCommand().toString(),
       ((Facilitator)Facilitator).getConnectionManager().getMyAddr(),
         quality));
```

Listing 6.6: Deo koda **LibraryRecordQualityAgent** klase koji se bavi ocenjivanjem kvaliteta zapisa – nastavak

```
      } else
       results.addElement(new RecordQualityReturnValue(
         task.getCommand().toString(),
       ((Facilitator)Facilitator).getConnectionManager().getMyAddr(),
         -1));
      // oslobodimo servis posle upotrebe
      ((Facilitator)Facilitator).getServiceManager().returnService(
        serviceId);
      Object distributedId;
      // prođemo kroz listu agentskih centara
      for (int i = 0; i < Facilitators.size(); i++) {
        String addr = (String)Facilitators.elementAt(i);
        Facilitator f = new ProxyConnector(addr);
        // regrutujemo jednog agenta
        distributedId  = f.recommendOne(t);
        // dodamo ovog agenta u listu spoljnih
        //agenata udaljenog agentskog centra
        f.addExternalAgentAddress(agentId,
      ((Facilitator)Facilitator).getConnectionManager().getMyAddr());
        // dodamo udaljenog agenta u listu spoljnih
        // agentskih centara
(Facilitator)Facilitator).getAgentManager().addExternalAgentAddress(
          distributedId, addr);
        // pošaljemo udaljenom agentu KQML poruku sa
        // formulacijom zadatka
        ((Facilitator)Facilitator).sendKQMLMessage(
           new KQMLMessage(
             "achieve",
             agentId.toString(),    // sender
             distributedId,         // receiver
             "", "",                // from, to
             "",                    // in-reply-to
             // reply-with
             "id"+((Facilitator)Facilitator).getUniqueId(),
             "KQML",                    // language
             "kqml-ontology",           // ontology
             t                          // content
             ));

        // sačekamo odgovor udaljenog agenta
        KQMLMessage m = waitKQMLMessage();
        // ubacimo njegov rezultat u konačan rezultat
        results.addElement(m.getContent());
```

Listing 6.6: Deo koda **LibraryRecordQualityAgent** klase koji se bavi ocenjivanjem kvaliteta zapisa – nastavak

```
        // uklonimo ovog agenta sa spiska spoljnih
        // agenata u udaljenom agentskom centru
        f.removeExternalAgentAddress(agentId);
        // uklonimo udaljenog agenta sa spiska
        // spoljnih agenata u agentskom centru
((Facilitator)Facilitator).getAgentManager().removeExternalAgentAddres
s(distributedId);
        // oslobodimo agenta
        f.releaseAgent(t, distributedId);
      }
      // završili smo sa radom
      result.setContent(results);
      return result;
  } catch (Exception ex) {
    ex.printStackTrace();
  }
  return null;
}

/** Kada agent primi KQML poruku, ova metoda se izvrsava.
 */
public synchronized void onKQMLMessage(KQMLMessage message,
  EJBObject Facilitator) {
  // sačuvamo poruku
  mess = message;
  // oslobodimo agenta ako čeka na poruku
  notify();
}

/** Čeka na prispeće KQML poruke. */
public synchronized KQMLMessage waitKQMLMessage() {
  try {
    // čekamo da poruka stigne
    wait();
  } catch (Exception ex) {
    ex.printStackTrace();
  }
  // vratimo prispelu poruku
  return mess;
}
```

Listing 6.6: Deo koda **LibraryRecordQualityAgent** klase koji se bavi ocenjivanjem kvaliteta zapisa – nastavak

LibraryRecordQualityAgent agent aktivira udaljenog agenta (**DistributedLibraryAgent** klasa) slanjem KQML poruke. Po prijemu KQML poruke udaljeni agent započinje pretragu i ocenjivanje kvaliteta zapisa. Listing 6.7 ilustruje rad udaljenog agenta.

```
/** Kada agent primi KQML poruku, ova metoda se izvršava.
 */
public void onKQMLMessage(KQMLMessage message,
  EJBObject Facilitator) {

  AgentTask task = (AgentTask)message.getContent();
  RecordQualityReturnValue result = null;
  try {
      // potražimo servis
     Object serviceId =
((Facilitator)Facilitator).getServiceManager().findService(
  "LibraryService");
      // uzmemo servis na osnovu njegovog id-a
      Service service =
       ((Facilitator)Facilitator).getServiceManager().getService(
         serviceId);
      // izdvojimo ISBN broj iz zadatka
      String isbn = task.getCommand().toString();
      // prilagodimo ISBN broj za pretragu
      StringTokenizer st = new StringTokenizer(isbn, "-");
      String isbnQuery = "BN=";
      while (st.hasMoreTokens()) {
        String s = st.nextToken();
        if (st.hasMoreTokens())
          isbnQuery += s + " [W] BN=";
        else
          isbnQuery += s;
      }
      // izvršimo pretragu biblioteke
      int hits = ((LibraryService)service).select(isbnQuery);
      if (hits == 1) {
        // preuzmemo ID zapisa
        int docId = ((LibraryService)service).getDocId(1);
        // pruzmemo zapis
        String doc = ((LibraryService)service).getDoc(docId);
        // ocenimo kvalitet
        int quality = ((LibraryService)service).quality(doc);
        // pripremimo rezultat ocenjivanja
       result = new RecordQualityReturnValue(
         task.getCommand().toString(),
       ((Facilitator)Facilitator).getConnectionManager().getMyAddr(),
         quality);
```

Listing 6.7: Deo listinga koda udaljenog agenta koji se bavi ocenjivanjem kvaliteta zapisa

```
      } else
       result = new RecordQualityReturnValue(
         task.getCommand().toString(),
       ((Facilitator)Facilitator).getConnectionManager().getMyAddr(),
        -1);
      // oslobodimo servis posle upotrebe
((Facilitator)Facilitator).getServiceManager().returnService(
        serviceId);
      // pošaljemo KQML poruku sa rezultatom
      // ocenjivanja glavnom agentu
      ((Facilitator)Facilitator).sendKQMLMessage(
      new KQMLMessage(
        "tell",
        message.getReceiver(),    // sender
        message.getSender(),      // receiver
        "", "",                   // from, to
        message.getReplyWith(),   // in-reply-to
        // reply-with
        "id"+((Facilitator)Facilitator).getUniqueId(),
        "KQML",                   // language
        "kqml-ontology",          // ontology
        result
        ));
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
```

Listing 6.7: Deo listinga koda udaljenog agenta koji se bavi ocenjivanjem kvaliteta zapisa – nastavak

I glavni i udaljeni agent vraćaju rezultat ocenjivanja u objektima klase **`RecordQualityReturnValue`**. Ova klasa sadrži adresu čvora bibliotečke mreže, ISBN broj zapisa i njegov kvalitet, kao što je prikazano u listingu 6.8.

```
public class RecordQualityReturnValue implements
Serializable {
  public String isbn;
  public String host;
  public int quality;

  public RecordQualityReturnValue (String isbn, String
host, int quality) {
    this.isbn = isbn;
    this.host = host;
    this.quality = quality;
  }

  public String toString() {
    return "Pogodak sa ISBN brojem: " + isbn + " na
ra\u010dunaru: " + host + " ima kvalitet: " + quality +
"%";
  }
}
```

Primer 6.8: Klasa **RecordQualityReturnValue**, koja čuva rezultat ocenjivanja zapisa

Za rad sa Bibliotečkim informacionim sistemom BISIS koristi se bibliotečki servis **LibraryService**. Ovaj servis omogućuje agentu rad sa zapisima unutar sistema BISIS. Listing 6.9 sadrži kod servisa **LibraryService**.

```
public class LibraryService implements Service {
  /** Konekcija sa bazom podataka. */
  Connection conn;
  /** BISIS tekst server za rad sa zapisima. */
  TextServer ts;

  public LibraryService() {
    try {
      // registrujemo drajver za pristup bazi podataka
      DriverManager.registerDriver(
        new oracle.jdbc.driver.OracleDriver());
      // otvorimo konekciju sa bazom podataka
      conn =
DriverManager.getConnection("jdbc:oracle:thin:@localhost:15
21:BIS9", "aaaaa", "bbbbb");
      conn.setAutoCommit(false);
      // pripremimo BISIS engine
```

Listing 6.9: Kod bibliotečkog servisa implementiranog klasom **LibraryService**

```
      Environment.setDriver("oracle.jdbc.driver.OracleDriver");
      Environment.setServer("localhost");
      Environment.setPort("1521");
      Environment.setSID("BIS9");
      Environment.setUsername("aaaaa");
      Environment.setPassword("bbbbb");
      Class.forName("bisis.edit.validation.NSPMFValidator");
      Environment.setValidator(
        ValidatorManager.getValidator());
      // inicijalizujemo okruženje bibliotekara
      LELibrarian lib = new LELibrarian(conn, "demo",
        "demo");
      // inicijalizujemo UNIMARC okruženje
      UFieldSet fs = new UFieldSet(conn);
      // inicijalizujemo tekst server
      ts = new TextServer(conn);
      Environment.setInternalTs(ts);
      // Odredimo adresu računara sa kojeg se izvršava
      // ovaj program
      String address = "";
     try { address =
InetAddress.getAllByName(InetAddress.getLocalHost().getHostAddress())[
0].getHostName(); } catch (UnknownHostException ex) { }
      Environment.init(lib, fs, ts, ts, address);
    } catch(Exception e) {
      e.printStackTrace();
    }
  }

  /** Pretražuje bazu podataka upotrebom DIALOG upita. */
  public int select(String command) throws Exception {
    return ts.select(command);
  }

  /** Vraća ID zapisa na osnovu rednog broja pogotka. */
  public int getDocId(int hit) throws Exception {
    return ts.getDocID(hit);
  }
  /** Vraća zapis u UNIMARC formatu na osnovu njegovog
   *  ID-a.
   */
  public String getDoc(int docId) throws Exception {
    return ts.getDoc(docId);
  }
```

Listing 6.9: Kod bibliotečkog servisa implementiranog klasom **LibraryService** – nastavak

```
  /** Vraća true ako zadato polje i potpolje postoji
   *  u zapisu.
   */
  private boolean exist(Vector fields, LESubfield lsf) {
    for (int i = 0; i < fields.size(); i++) {
      Field f = (Field) fields.elementAt(i);
      if (f.getName().equals(lsf.getField())) {
        Vector subfields = f.getSubfields();
        for (int j = 0; j < subfields.size(); j++) {
          Subfield sf = (Subfield)subfields.elementAt(j);
          if (sf.getName().equals(
            lsf.getSubfield(f.getName()))) {
            return true;
          }
        }
      }
    }
    return false;
  }
  /** Vraća kvalitet pogotka na osnovu spiska polja
   *  iz okruženja.
   */
  public int quality(String rezance) throws Exception {
    LELibrarian lib = Environment.getLib();
    Vector envFields = lib.getProcLESubfields();
    Vector fields = RezanceUtilities.unpackRezance(rezance);
    int counter = 0;
    for (int i = 0; i < envFields.size(); i++) {
      LESubfield lsf = (LESubfield)envFields.elementAt(i);
      if (exist(fields, lsf))
        counter++;
    }
    return (int)((float)counter/(float)envFields.size()*100.00f);
  }
  /** Poziva se prilikom vraćanja servisa. */
  public void remove() {
    try {
      conn.close();
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Listing 6.9: Kod bibliotečkog servisa implementiranog klasom **`LibraryService`** – nastavak

U konstruktoru klase **`LibraryService`** se otvara konekcija ka bazi podataka i iz baze podataka se inicijalizuje definicija UNIMARC standarda, okruženje bibliotekara i tekst server. Metode **`select()`**,

getDocId() i **getDoc()** su direktno preslikane iz tekst servera, odnosno, one direktno pozivaju istoimene metode iz tekst servera. Metoda **exist()** vraća **true** ako prosleđeno polje i potpolje postoji u definiciji dozvoljenih polja za zadati tip publikacije, a na osnovu UNIMARC definicije. Ova metoda je ključna za rad metode **quality()**, koja za svako polje i potpolje prosleđenog zapisa određuje kvalitet na osnovu broja dozvoljenih polja sa potpoljima. Ako zapis ima sva polja i potpolja dozvoljena UNIMARC standardom, kvalitet je 100%.

Proces ocenjivanja kvaliteta zapisa se inicira iz klijentske aplikacije BISIS, pozivanjem naredbe:

```
quality broj_pogotka
```

Ova naredba izvlači ISBN broj iz zapisa i inicira agentsku pretragu i ocenjivanje kvaliteta zapisa, kao što je prikazano u listingu 6.10.

```
public void execute() {
  int hit;
  try {
    if (Environment.getReadTs().getHitCount() == 0) {
      JOptionPane.showMessageDialog(null, "Nema pogodaka!",
       "Greska", JOptionPane.ERROR_MESSAGE);
      return;
    }
  } catch (Exception ex) {}
  if (params.size() == 1) {  // jedan parametar
    try {
      hit = Integer.parseInt((String)params.elementAt(0));
    } catch (NumberFormatException ex) {
      JOptionPane.showMessageDialog(null,
        "Neispravno unet broj!", "Greska",
        JOptionPane.ERROR_MESSAGE);
      return;
    }
    int docID = -1;
    // izvučemo ID dokumenta na osnovu broja pogotka
    try { docID = Environment.getReadTs().getDocID(hit); }
catch (Exception ex) {}
    if (docID == -1) {
      JOptionPane.showMessageDialog(null,
        "Pogre\u0161an broj pogotka!", "Greska",
        JOptionPane.ERROR_MESSAGE);
      return;
    }
```

Listing 6.10: Kod *Quality* komande klijentske aplikacije

```
    String rez = "";
    // izvučemo zapis na osnovu ID-a
    try { rez = Environment.getReadTs().getDoc(docID); }
catch (Exception ex) {}
    // napravimo hash mapu od zapisa
    HashMap mapa = prepareMap(rez);
    // izvučemo ISBN broj iz zapisa
    String isbn = (String)mapa.get("BN");
    // pripremimo pretragu
    mf.setAbortedSearch(false);
    // Kreiramo proksi ka agentskom centru
    FacilitatorProxy fp = new FacilitatorProxy();
    // pripremimo zadatak za izvršavanje
    LibraryRecordQualityTask task = new
LibraryRecordQualityTask(isbn, "");
    // pripremimo listener koji će čekati rezultate
    // rada agenata
    QualityServiceAgentListener al= new QualityServiceAgentListener(
      mf, this);
    // pokrenemo izvršenje zadatka
    fp.execute(task, al);
    // sačekamo rezultat
    synchronized (mf) {
      try {
        mf.wait();
      } catch(InterruptedException ex) {
        System.err.println("Thread interrupt error: " +
          ex.getMessage());
      }
    }
    // zustavimo proksi
    fp.stop();
    // završili smo pretragu
    mf.setAbortedSearch(false);
    // prepakujemo rezultate rada agenata u listu koju
    // ćemo prikazati na ekranu
    Vector lines = new Vector();
    Vector resV = (Vector)result.getContent();
    for (int i = 0; i < resV.size(); i++) {
     RecordQualityReturnValue rv = (RecordQualityReturnValue)
       resV.elementAt(i);
      lines.add(rv.toString());
    }
```

Listing 6.10: Kod *Quality* komande klijentske aplikacije – nastavak

```
    mf.hld.setTitle("Rezultat pregleda kvaliteta zapisa na
bibliote\u010dkoj mre\u017ei:");
    mf.hld.setList(lines);
    mf.hld.setVisible(true);
  } else {
    JOptionPane.showMessageDialog(null,
      "Pogre\u0161an broj parametara!", "Greska",
      JOptionPane.ERROR_MESSAGE);
  }
  mf.tfComLin.setText("");
}
/** Unutrašnja klasa koja implementira agentski listener
 *  i kojoj će agenti prijaviti rezultate rada.
 */
class QualityServiceAgentListener implements AgentListener
{
  MainFrame mf;
  QualityCommand qc;
  public QualityServiceAgentListener(MainFrame mf,
   QualityCommand qc) {
   this.mf = mf;
   this.qc = qc;
  }
  public void actionStarted(AgentEvent e) {
    System.out.println("%%% Startovao posao: " +
      e.getResult().getContent());
  }
  public void actionPerforming(AgentEvent e) {
    System.out.println("%%% Radim posao: " +
      e.getResult().getContent());
  }
  public void actionPerformed(AgentEvent e) {
    System.out.println("%%% Uradio posao: " +
      e.getResult().getContent());
    // sačuvamo rezultat rada agenata
    qc.result = e.getResult();
    // obavestimo klijentsku aplikaciju da je
    // gotovo ocenjivanje
    synchronized (mf) {
      mf.notify();
    }
  }
}
```

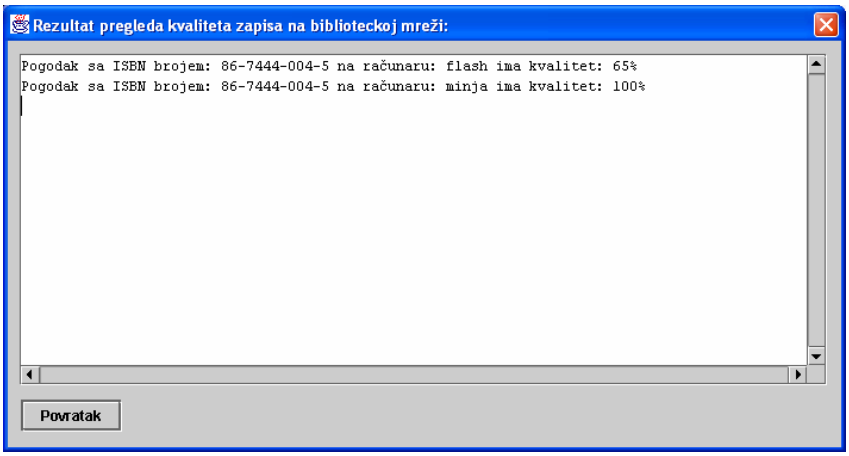
Listing 6.10: Kod *Quality* komande klijentske aplikacije – nastavak

```
/** Rezultat rada agenata. */
public AgentResult result;

/** Kreira HashMap objekat sa preslikavanjem
 *  prefiks->vrednost za dati zapis.
 */
public HashMap prepareMap(String rezance) {
  HashMap prefixMap = null;
  try { prefixMap = Environment.getWriteTs().getPrefixMap();
  } catch (Exception ex) {}
  DocumentParser dp = new DocumentParser(prefixMap,
    "\36", "\37");
  Vector pv = dp.parseDocument(rezance);
  PrefixPair pp = null;
  HashMap hMap = new HashMap(true);
  for (int h = 0; h < pv.size(); h++) {
    pp = (PrefixPair)pv.elementAt(h);
    hMap.add(pp.prefName, pp.value);
  }
  return hMap;
}
```

Listing 6.10: Kod *Quality* komande klijentske aplikacije – nastavak

Unutrašnja klasa **`QualityServiceAgentListener`** implementira interfejs **`AgentListener`** i obezbeđuje klijentskoj aplikaciji dostavljanje rezultata rada agenata. Agent po izvršenju zadatka izvršava metodu **`actionPerformed()`**, u okviru koje se poziva metoda **`notify()`**, čime se klijentska aplikacija obaveštava o završetku rada i prosleđuje joj se rezultat rada. Rezultat rada je vektor objekata klase **`RecordQualityReturnValue`**. Kvaliteti zapisa i adrese servera na kojima su pronađeni se ubacuju u spisak koji se prikazuje na ekranu BISIS klijentske aplikacije.

Iz BISIS klijenta se to vidi kao što je prikazano na slici 6.6:



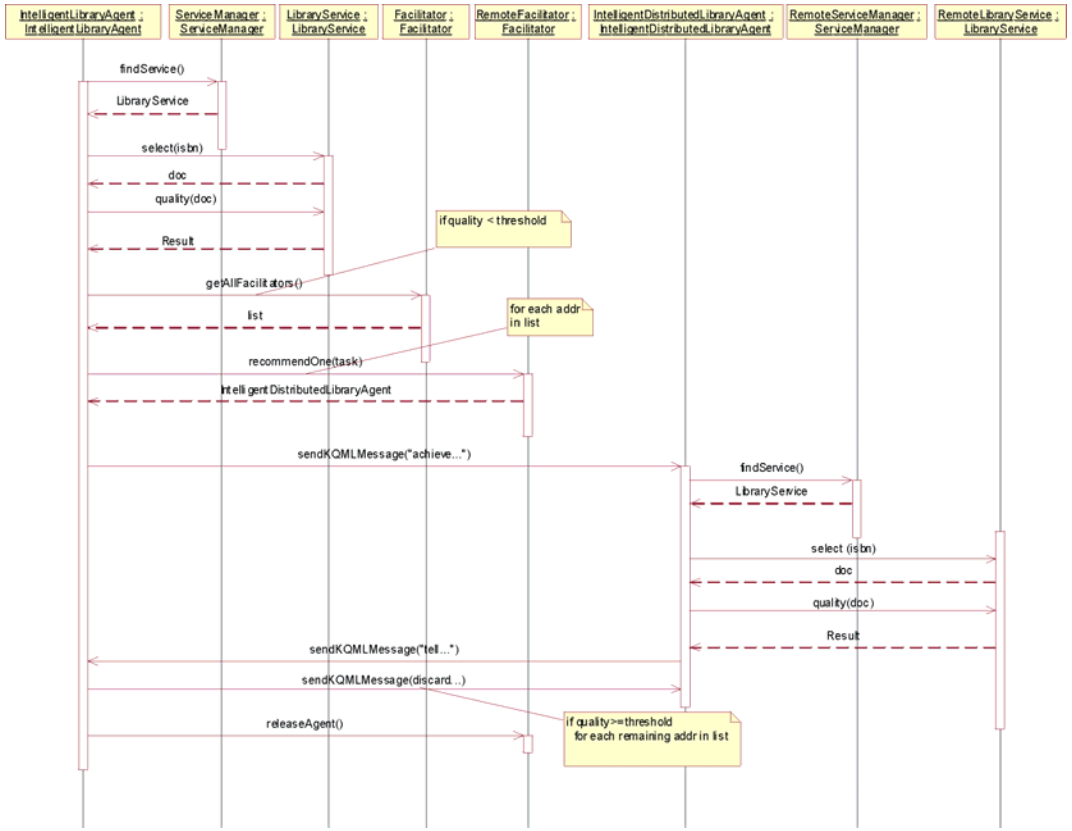
Slika 6.6: Spisak zapisa na bibliotečkoj mreži, sa njihovim kvalitetom

6.3 Inteligentna raspodela opterećenja prilikom pretrage

Prilikom ocenjivanja kvaliteta zapisa sakupljaju se rezultati pretrage svih agenata. To znači da svi agenti pretražuju bibliotečku mrežu i da se njihovi rezultati prikupljaju bez obzira na kvalitet zapisa koji su pronašli. Zadatak nije izvršen dok se ne prikupe rezultati svih agenata. Za ovakve situacije, moguće je modifikovati agente tako da se traži prvi zapis sa kvalitetom iznad određenog praga, a kada se takav zapis pronađe, ostalim agentima se javlja da prekidaju sa pretragom. Time je omogućeno da se do zadovoljavajućeg rezultata dođe u kraćem vremenu, uz smanjen angažman agenata i mrežnih resursa.

Za ilustraciju ovakvog pristupa razvijena su dva agenta: **`IntelligentLibraryAgent`** i **`IntelligentDistributedLibraryAgent`**. **`IntelligentLibraryAgent`** je agent koji prvo pretražuje lokalnu bazu zapisa i ako pronađe zapis sa kvalitetom iznad praga, vraća rezultat i prekida dalji rad.

Ako u lokalnoj bazi nije pronađen zapis zadovoljavajućeg kvaliteta, uzima se spisak svih raspoloživih agentskih centara i u svakom od njih se angažuje agent klase **`IntelligentDistributedLibraryAgent`** koji će pretraživati udaljenu bazu zapisa. Svaki agent vraća rezultat ocenjivanja glavnom agentu, a kada se pronađe prvi zapis zadovoljavajućeg kvaliteta, glavni agent će poslati poruku svim preostalim agentima da prekidaju pretragu. Dijagram sekvence na slici 6.7 ilustruje postupak pretrage.



Slika 6.7: Dijagram sekvence inteligentne raspodele opterećenja

Proces ocenjivanja kvaliteta započinje angažovanjem agenta implementiranog klasom **`IntelligentLibraryAgent`**. Angažovanje agenta podrazumeva prosleđivanje zadatka u okviru kojeg je ključan ISBN broj zapisa čiji kvalitet treba oceniti. Agent započinje svoj rad pronalaženjem bibliotečkog servisa. Po pronalaženju bibliotečkog servisa, njemu se zadaje da pronađe zapis sa zadatim ISBN brojem i zatim se inicira ocena kvaliteta zapisa. Ocenu kvaliteta zapisa obavlja bibliotečki servis u skladu sa definisanim skupom UNIMARC polja i potpolja koje svaki zapis mora da ima. Koristi se isti bibliotečki servis kao i u prethodnom primeru – **`LibraryService`**. Ocena se vraća agentu, a ovaj proverava da li kvalitet zapisa u lokalnoj bazi zadovoljava kriterijum. Ako zadovoljava, pretraga se prekida i vraća se rezultat.

Ako kvalitet lokalnog zapisa nije zadovoljavajući, od lokalnog agentskog centra se traži lista svih agentskih centara u mreži. Agent zatim prolazi kroz listu i od svakog udaljenog agentskog centra zahteva jednog agenta za ocenjivanje kvaliteta zapisa (implementiran u klasi **`IntelligentDistributedLibraryAgent`**). Ako ga dobije, šalje mu KQML poruku u kojoj mu zadaje da pronađe i oceni kvalitet zapisa, identifikovanog preko ISBN broja.

Udaljeni agent po prijemu KQML poruke pronalazi bibliotečki servis i uz pomoć njega pronalazi zapis. Po pronalaženju zapisa, agent inicira ocenjivanje kvaliteta zapisa i vraća ocenu glavnom agentu. Ako je prosleđeni kvalitet zapisa zadovoljavajući, glavni agent šalje KQML poruku sa komandom **`discard`** svim preostalim udaljenim agentima i vraća pronađeni kvalitet zapisa klijentu. Udaljeni agenti će primiti poruku o odustajanju i prekinuće rad. Listing 6.11 ilustruje deo koda **`LibraryRecordQualityAgent`** klase koji se bavi ocenom kvaliteta zapisa.

```
public AgentResult execute(AgentTask task,
  EJBObject facilitator, Object agentId) {

  AgentResult result;
  receivedResults = 0;
  allAgents = new Vector();

  try {
    // pripremimo zadatak za udaljene agente
    IntelligentDistributedLibraryTask t =
      new IntelligentDistributedLibraryTask(
        task.getCommand());
    // pripremimo rezultate rada;
    result = new AgentResult();
    // sada potražimo kod lokalnog Facilitatora...
    Object serviceId =
((Facilitator)facilitator).getServiceManager().findService(
"LibraryService");
    results = new Vector();
    // uzmemo servis na osnovu njegovog id-a
    Service service =
((Facilitator)facilitator).getServiceManager().getService(
  serviceId);

    String isbn = task.getCommand().toString();
    StringTokenizer st = new StringTokenizer(isbn, "-");
    String isbnQuery = "BN=";
    while (st.hasMoreTokens()) {
      String s = st.nextToken();
      if (st.hasMoreTokens())
        isbnQuery += s + " [W] BN=";
      else
        isbnQuery += s;
    }

    // izvršimo pretragu biblioteke
    int hits = ((LibraryService)service).select(isbnQuery);
    int quality = -1;
    if (hits == 1) {
      int docId = ((LibraryService)service).getDocId(1);
      String doc = ((LibraryService)service).getDoc(docId);
      quality = ((LibraryService)service).quality(doc);
    }
```

Listing 6.11: Deo koda **`IntelligentLibraryAgent`** klase koji se bavi inteligentnom raspodelom opterećenja prilikom ocenjivanja kvaliteta zapisa

```
    // oslobodimo servis posle upotrebe
 ((Facilitator)facilitator).getServiceManager().returnService(
       serviceId);
    if (quality <= THRESHOLD) {
      // lokalni zapis ne zadovoljava kriterijum
      // napravimo spisak dostupnih Facilitatora
      facilitators=((Facilitator)facilitator).getAllFacilitators();
      Object distributedId;
      for (int i = 0; i < facilitators.size(); i++) {
        String addr = (String)facilitators.elementAt(i);
        Facilitator f = new ProxyConnector(addr);
        // regrutujemo jednog library agenta.
        distributedId  = f.recommendOne(t);
        // dodamo regrutovanog agenta u listu
        // agenata od kojih očekujemo rezultat
        allAgents.addElement(distributedId);
        // dodamo ovog agenta u listu spoljnih agenata
        // udaljenog Facilitator-a
        f.addExternalAgentAddress(agentId,
        ((Facilitator)facilitator).getConnectionManager().getMyAddr());

/* Dodamo udaljenog agenta u listu spoljnih agenata. Ovo će
 * biti uklonjeno po prijemu KQML poruke od udaljenog
 * agenta (sa nađenim kvalitetom). */
 ((Facilitator)facilitator).getAgentManager().addExternalAgentAddress(
  distributedId, addr);
      // pošaljemo KQML poruku sa zadatkom udaljenom agentu
      ((Facilitator)facilitator).sendKQMLMessage(
        new KQMLMessage(
          "achieve",
          agentId,                // sender
          distributedId,          // receiver
          "", "",                 // from, to
          "",                     // in-reply-to
          // reply-with
          "id"+((Facilitator)facilitator).getUniqueId(),
          "KQML",                 // language
          "kqml-ontology",        // ontology
          t));
      } // for
```

Listing 6.11: Deo koda **`IntelligentLibraryAgent`** klase koji se bavi inteligentnom raspodelom opterećenja prilikom ocenjivanja kvaliteta zapisa – nastavak

```
      // sada čekamo na najbolji zapis
      synchronized (this) {
        wait();
      }
    } else
      // lokalni zapis zadovoljava kriterijum
      results.addElement(new RecordQualityReturnValue(
        task.getCommand().toString(),
       ((Facilitator)facilitator).getConnectionManager().getMyAddr(),
         quality));

    // ako nije nađen zapis, vraćamo kvalitet -1
    if (results.size() == 0)
      results.addElement(new RecordQualityReturnValue(
        task.getCommand().toString(),
 ((Facilitator)facilitator).getConnectionManager().getMyAddr(),
        -1));

    result.setContent(results);
    return result;
  } catch (Exception ex) {
      ex.printStackTrace();
      return null;
  }
}

/** Kada agent primi KQML poruku, ova metoda se izvrsava.
 */
public synchronized void onKQMLMessage(KQMLMessage message,
  EJBObject facilitator) {

  try {
    // uklonimo udaljenog agenta sa spiska spoljnih agenata
((Facilitator)facilitator).getAgentManager().removeExternalAgentAddress(
  message.getSender());
  } catch (Exception ex) {
    ex.printStackTrace();
  }
```

Listing 6.11: Deo koda **`IntelligentLibraryAgent`** klase koji se bavi inteligentnom raspodelom opterećenja prilikom ocenjivanja kvaliteta zapisa – nastavak

```
  // izbrišemo agenta sa spiska agenata od kojih se
  // očekuje rezultat
  int idx = allAgents.indexOf(message.getSender());
  if (idx != -1) {
    allAgents.remove(idx);
  }

  // pokupimo kvalitet iz odgovora
  RecordQualityReturnValue r =
(RecordQualityReturnValue)message.getContent();

  if (r.quality > THRESHOLD) {
    // ovaj rezultat zadovoljava kriterijum
    // zapamtimo ovaj rezultat
    results.addElement(message.getContent());

    // pošaljemo svim preostalim agentima poruku
    //da ne traže dalje, niti da šalju rezultat rada
    for (int i = 0; i < allAgents.size(); i++) {
      Object id = allAgents.elementAt(i);
      try {
        ((Facilitator)facilitator).sendKQMLMessage(
          new KQMLMessage(
            "discard",
            message.getReceiver(), // sender
            id,                    // receiver
            "", "",                // from, to
            "",                    // in-reply-to
            // reply-with
            "id"+((Facilitator)facilitator).getUniqueId(),
            "KQML",                // language
            "kqml-ontology",       // ontology
            ""
            ));
      } catch (Exception ex) {
        ex.printStackTrace();
      }
    }
    // iniciramo vraćanje rezultata
    synchronized (this) {
      notify();
    }
```

Listing 6.11: Deo koda **`IntelligentLibraryAgent`** klase koji se bavi inteligentnom raspodelom opterećenja prilikom ocenjivanja kvaliteta zapisa – nastavak

```
  } else if (++receivedResults == facilitators.size()) {
    // poslednji agent je poslao poruku
    // bez obzira da li kvalitet zadovoljava ili ne,
    // iniciramo vraćanje rezultata
    synchronized (this) {
      notify();
    }
  }
}
```

Listing 6.11: Deo koda **IntelligentLibraryAgent** klase koji se bavi inteligentnom raspodelom opterećenja prilikom ocenjivanja kvaliteta zapisa – nastavak

IntelligentLibraryAgent agent aktivira udaljenog agenta (**IntelligentDistributedLibraryAgent** klasa) slanjem KQML poruke. Po prijemu KQML poruke udaljeni agent započinje pretragu i ocenjivanje kvaliteta zapisa. Ako u međuvremenu primi KQML poruku o odustajanju, obustaviće rad i neće poslati rezultat glavnom agentu. Listing 6.12 ilustruje rad udaljenog agenta.

```
/** Kada agent primi KQML poruku, ova metoda se izvrsava.
 */
public void onKQMLMessage(KQMLMessage message,
  EJBObject Facilitator) {
  dontSend = false;
  if (message.getCommand().equals("discard")) {
    // ako primimo poruku o odustajanju,
    // postavimo atribut dontSend na true
    dontSend = true;
    return;
  }
  // uzmemo zadatak
  AgentTask task = (AgentTask)message.getContent();
  RecordQualityReturnValue result = null;
  // potražimo servis za rad sa bibliotekom
  try {
    Object serviceId =
((Facilitator)Facilitator).getServiceManager().findService(
"LibraryService");
```

Listing 6.12: Deo listinga koda udaljenog agenta koji se bavi ocenjivanjem kvaliteta zapisa

```
    // uzmemo servis na osnovu njegovog id-a
    Service service =
((Facilitator)Facilitator).getServiceManager().getService(serviceId);
    String isbn = task.getCommand().toString();
    StringTokenizer st = new StringTokenizer(isbn, "-");
    String isbnQuery = "BN=";
    while (st.hasMoreTokens()) {
      String s = st.nextToken();
      if (st.hasMoreTokens())
        isbnQuery += s + " [W] BN=";
      else
        isbnQuery += s;
    }
    if (!dontSend) {
      // izvrsimo pretragu biblioteke
      int hits = ((LibraryService)service).select(isbnQuery);
      if (hits == 1) {
        int docId = ((LibraryService)service).getDocId(1);
        String doc = ((LibraryService)service).getDoc(docId);
        int quality = ((LibraryService)service).quality(doc);
        result = new RecordQualityReturnValue(
          task.getCommand().toString(),
       ((Facilitator)Facilitator).getConnectionManager().getMyAddr(),
          quality);
      } else
        result = new RecordQualityReturnValue(
          task.getCommand().toString(),
      ((Facilitator)Facilitator).getConnectionManager().getMyAddr(),
          -1);
    }
    // oslobodimo servis posle upotrebe
    ((Facilitator)Facilitator).getServiceManager().returnService(
     serviceId);

    String msgId ="id"+((Facilitator)Facilitator).getUniqueId();
    if (!dontSend) {
      // samo ako mozemo da posaljemo
```

Listing 6.12: Deo listinga koda udaljenog agenta koji se bavi ocenjivanjem kvaliteta zapisa – nastavak

```
      ((Facilitator)Facilitator).sendKQMLMessage(
        new KQMLMessage(
          "tell",
          message.getReceiver(),     // sender
          message.getSender(),       // receiver
          "", "",                    // from, to
          message.getReplyWith(),    // in-reply-to
          msgId,                     // reply-with
          "KQML",                    // language
          "kqml-ontology",           // ontology
          result
          ));
       // sacekamo da poruka bude poslata
  while(((Facilitator)Facilitator).getMessageManager().getKQMLMessage(
     msgId) != null)
       ;
      }
    }
    // uklonimo ovog agenta sa spiska spoljnih agenata
    // u udaljenom Facilitator-u
    ((Facilitator)Facilitator).removeExternalAgentAddress(
      message.getSender());
    // agent oslobađa samog sebe
    ((Facilitator)Facilitator).releaseAgent(task,
      message.getReceiver());
  } catch (Exception ex) {
    ex.printStackTrace();
  }
}
```

Listing 6.12: Deo listinga koda udaljenog agenta koji se bavi ocenjivanjem kvaliteta zapisa – nastavak

Ako udaljeni agent u toku rada primi KQML poruku o prekidu rada, biće automatski pozvana metoda **`onKQMLMessage()`**, bez obzira što se agent već nalazi u njoj. U slučaju da je ovakva poruka stigla, atribut **`dontSend`** se postavlja na **`true`** i metoda završava rad, tj. agent nastavlja da izvršava zadatak zadat prethodnom KQML porukom. Pošto je u tom slučaju atribut **`dontSend`** promenjen sa **`false`** na **`true`**, agent će prestati da izvršava zadatak. Bilo da je izvršio zadatak ili ne, agent će sam sebe osloboditi u agentskom centru. Ovim je pronalaženje najkvalitetnijeg zapisa okončano.

6.4 Agentski server za korisničko pretraživanje

Preuzimanje zapisa u softverskom sistemu BISIS je implementirano i za korisnike preko web-a [Milosavljević00b]. Sistem pruža mogućnost da korisnik preuzme željene zapise preko posebne web stranice za preuzimanje podataka. Podrazumeva se da korisnik zna na kom će serveru tražiti zapis. I u ovom slučaju je poželjno unaprediti preuzimanje zapisa tako da korisnik ne mora da poznaje adrese svih BISIS web servera, već da iniciranjem pretrage sa jednog čvora bibliotečke mreže bude u mogućnosti da pronađe i preuzme sve raspoložive zapise sa svih čvorova. To znači da je upotrebom agentskog servera za korisničko pretraživanje moguće realizovati centralni katalog iako se BISIS sistem bazira na lokalnim bazama zapisa.

Agentski server za korisničko pretraživanje bi omogućio korisnicima da obavljaju pretragu bibliotečke mreže sa jednog mesta. Pretraga zapisa bi se svela na posetu web stranice centralnog kataloga, koja bi parametre za pretragu prosledila pojedinačnim agentima, a rezultati njihovog rada bi se sakupili i prikazali na web stranici zajedno sa adresom servera na kome je zapis pronađen. Time bi bilo omogućeno da se iniciranjem pretrage sa jednog mesta unutar bibliotečke mreže može pronaći svaka pojava željenog zapisa na svim serverima. U tom slučaju, nije potrebno da korisnik zna konkretne adrese čvorova bibliotečke mreže, jer se upotrebom hiperlinkova može obezbediti preuzimanje svih pronađenih zapisa sa jednog mesta.

Ovaj koncept na višem nivou apstrakcije nudi formiranje virtuelne mreže biblioteka, gde bi mreža agentskih okruženja povezanih sa bibliotečkim serverima definisala topologiju virtuelne bibliotečke mreže. Takođe, ovaj sistem ne bi bio vezan isključivo za jedan tip bibliotečkog servera, već bi omogućio formiranje mreže heterogenih bibliotečkih servera. Jedini uslov je da za svaki bibliotečki server postoji odgovarajući bibliotečki servis koji omogućuje agentima pristup bazi zapisa. Ovaj servis omogućuje agentu unificiran pristup bazi zapisa, nezavisno od tipa bibliotečkog servera. To znači da nije potrebno da se agenti prilagođavaju za rad sa konkretnim bibliotečkim serverima jer se bibliotečki servis nalazi između agenta i baze zapisa.

Zaključak

Tema doktorske teze pripada oblasti agentske tehnologije. Glavni rezultati dobijeni u tezi su sledeći:

1. Predložen je model agentskog okruženja baziran na tehnologiji distribuiranih komponenti. Ovaj model podržava FIPA specifikaciju i sledeće koncepte agentskog okruženja: razmenu poruka, mobilnost agenata, sigurnosne mehanizme i direktorijume agenata i servisa.
2. Implementacija agentskog okruženja je izvršena u J2EE tehnologiji.
3. Sistem podržava koncept *plug-in*-ova za sve bitne komponente agentskog okruženja (menadžere).
4. Dat je model i implementacija koncepta mobilnih zadataka.
5. Modeliran je i implementiran sistem međusobnog uređenja odnosa agentskih centara.
6. Izvršena je verifikacija agentskog okruženja na bibliotečkom softverskom sistemu.

Za modeliranje agentskog okruženja usvojen je objedinjeni jezik modeliranja (*Unified Modeling Language* – UML), a za implementaciju Java okruženje.

Opisana je agentska tehnologija i navedene dve najčešće tehnologije (CORBA i J2EE). Dat je i detaljan opis FIPA specifikacije. FIPA specifikacija obuhvata osnovne elemente agentskog okruženja: definiciju apstraktne arhitekture, direktorijum agenata i servisa, transport i razmenu poruka, kao i kriptovanje i validaciju poruka.

Dat je i pregled radova na temu agentskih okruženja, kao i analiza šest reprezentativnih agentskih okruženja. Svako okruženje je analizirano sa stanovišta najbitnijih faktora agentskog okruženja: razmena poruka, mobilnost agenata, sigurnost, direktorijum agenata i servisa i FIPA kompatibilnost.

Prikazan je spisak tehnika iz J2EE tehnologije koje se mogu upotrebiti u agentskoj tehnologiji. Koncepti koji se žele ostvariti su asinhrona razmena poruka, mobilnost agenata, direktorijumi i sigurnosni mehanizmi, a tehnologije potrebne za realizaciju su JMS sistem poruka, serijalizacija, JNDI tehnologija i J2EE sigurnosni mehanizmi.

Ključna poglavlja su četvrto i peto. U četvrtom poglavlju je data detaljna specifikacija informacionih zahteva proširivog agentskog okruženja, korišćenjem objedinjenog jezika modeliranja. Ovakvo agentsko okruženje je u potpunosti u saglasnosti sa FIPA specifikacijom. U petom poglavlju opisana je implementacija predloženog agentskog okruženja u J2EE tehnologiji.

Prikazano agentsko okruženje je verifikovano na bibliotečkom informacionom sistemu BISIS. Okruženje je verifikovano na sledećim primerima: agentsko pretraživanje bibliotečke mreže, sistem ocenjivanja kvaliteta zapisa i inteligentna raspodela opterećenja (šesto poglavlje).

Implementacija modelovanog agentskog okruženja, bazirana na J2EE tehnologiji, pruža određene pogodnosti, zato što većinu koncepata potrebnih za rad (udaljeno izvršenje, razmenu poruka, direktorijume, sigurnost, itd.) prepušta odgovarajućim elementima J2EE tehnologije.

Koncept međusobnog uređenja odnosa agentskih centara omogućuje jednostavnu eksploataciju agenata iz agentskih centara zato što programer ne mora da poznaje topologiju mreže, niti stvarne lokacije agentskih centara.

Koncept *plug-in*-ova omogućuje dodatnu fleksibilnost po pitanju instalacije, jer dozvoljava implementaciju različitih algoritama i koncepata u agentskim okruženjima, koji ne moraju biti prisutni u momentu kompajliranja koda.

Koncept mobilnih zadataka omogućuje pisanje specifičnih zadataka koji sadrže u sebi i program za rešavanje, čime se projektanti oslobađaju obaveze kreiranja specijalizovanih agenata, a svi agentski centri obaveze da poseduju sve vrste agenata. Dovoljno je da svi agentski centri poseduju generičke agente, a oni će izvršavati programe sadržane u samim zadacima.

Pokazalo se da implementacija agentskog okruženja i primera ne zavisi od konkretnog aplikacionog servera. Pošto je agentsko okruženje implementirano u programskom jeziku Java, njegova instalacija na određenom aplikacionom serveru se svela na podešavanje parametara u konfiguracionim datotekama samog aplikacionog servera.

Dalji pravci razvoja bi trebalo da budu u oblasti razvoja specijalizovanog agentskog jezika, kao i u oblasti razvoja svih menadžera. Posebna pažnja bi trebalo da bude posvećena razvoju sigurnosnog menadžera.

LITERATURA

[Agha86] Agha G., "ACTORS: A Model of Concurrent Computation in Distributed Systems", MIT Press, Cambridge, MA, 1986.

[Agha93] Agha G., Wegner P., Yonezawa A., "Research Directions in Concurrent Object-Oriented Programming", MIT Press, Cambridge, MA, 1993.

[Aglets] Aglets Home Page, http://www.trl.ibm.co.jp/aglets

[Allen90] Allen. J. F., Hendler J., Tate A., "Readings in Planning", Morgan Kaufman Publishers, San Mateo, CA, 1990.

[Aridor98] Aridor Y., Lange D., "Agent design patterns: elements of agent application design", Proceedings of the second international conference on Autonomous agents, Minneapolis, Minnesota, United States, 1998, ISBN:0-89791-983-1, pp. 108 – 115.

[Bellavista00a] Bellavista P., Corradi A., Stefanelli C., "A Mobile Agent Infrastructure for the Mobility Support", ACM Symposium on Applied Computing (SAC 2000), Italy, Como, Mar. 2000, ACM Press, pp. 539-546.

[Bellavista00b] Bellavista P., Corradi A., Tomasi A., "The mobile agent technology to support and to access museum information", Proceedings of the 2000 ACM symposium on Applied computing 2000, 2000 , Como, Italy, ISBN:1-58113-240-9, pp. 1006 – 1013.

[Bellifemine99] Bellifemine F., Poggi A., Rimassa G., "JADE – A FIPA-compliant agent framework", Proceedings of PAAM'99, London, April 1999, pagg.97-108.

[Benech97] Benech D., Desprats T., "A KQML-CORBA based Architecture for Intelligent Agents Communication in Cooperative Service and Network Management", IFIP/IEEE International Conference on Management of Multimedia Networks and Services, Canada, Montreal, July 1997.

[Bigus] Bigus J., "The Agent Building and Learning Environment", White Paper, available at http://www.research.ibm.com/able

[Bigus00] Bigus J., "The agent building and learning environment", Proceedings of the fourth international conference on Autonomous agents, Spain, Barcelona, June 2000.

[Binder02] Binder W., Roth V., "Secure mobile agent systems using Java: where are we heading?", Proceedings of the 2002 ACM symposium on Applied computing, 2002 , Madrid, Spain, ISBN:1-58113-445-2, pp. 115 – 119.

[Blixt00] Blixt K., Öberg R., "Software Agent Framework Technology", magistarska teza, Linköping University, Department of Computer and Information Science, 2000.

[Bowman94] Bowman C., Danzig P., Schwartz M., Manber U., "Scalable Internet Resource Discovery - Research Problems and Approaches", Communications of the ACM, July 94/Vol.37, No.7.

[Bradshaw97] Bradshaw J., " Software Agents", MIT Press: Cambridge, MA, 1997, ISBN 0-262-52234-9.

[Bratman88] Bratman M. E., Israel D. J., Pollack M.E., "Plans anbd resource-bounded practical reasoning", Computational Intelligence, 1988, vol. 4, pp. 349-355.

[Brooks86] Brooks R. A., "A robust layered control system for mobile robot", IEEE Journal of Robotics and Automation, vol2(1) , 1986., pp. 14-23.

[Brooks91a] Brooks R. A., "Intelligence without reason", Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91), Sydney, Australia, 1991., pp. 569-595.

[Brooks91b] Brooks R. A., "Intelligence without representation", Artificial Intelligence, 1991., vol 47., pp. 139-159.

[Brugali00] Brugali D., Sycara K, "Towards agent oriented application frameworks", ACM Computing Surveys (CSUR),Volume 32 , Issue , (March 2000) , ISSN:0360-0300, pp. 21-27.

[Chapman87] Chapman D., "Planning for conjuctive goals", Artificial Intelligence, 1987., vol 32, pp. 333-378.

[Chauhan98] Chauhan D., Baker A., "JAFMAS: a multiagent application development system", Proceedings of the second international conference on Autonomous agents, Minneapolis, Minnesota, United States, 1998, ISBN:0-89791-983-1, pp. 100 – 107.

[CORBA] OMG. Common Object Request Broker: Architecture and Specification. OMG Specification Revision 2.0, July 1995., http://www.omg.org

[CORBAMessaging] Notification/Java Message Service Interworking RFP, ab/01-07-01, http://www.omg.org

[Cowan02] Cowan D., Griss M., Burg B., "BlueJADE - A service for managing software agents", 1st International Workshop on "Challenges in Open Agent Systems", AAMAS'02, Bologna, Italy, July 2002.

[d'Inverno97] d'Inverno M., Luck M., "Development and Application of a Formal Agent Framework", In Proceedings of the First IEEE International Conference on Formal Engineering Methods, Hiroshima, Japan, 1997.

[DiPippo99] DiPippo L., Hodys E., Thuraisingham B., "Towards a Real-Time Agent Architecture - A Whitepaper", Fifth International Workshop on Object-Oriented Real-Time Dependable Systems, USA, California, Monterey, November 18 - 19, 1999., p. 59.

[EJB] EJB homepage, http://java.sun.com/products/ejb/

[Ferber99] Ferber J., "Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence", Addison-Wesley Pub. Co., Feb 1999, ISBN:021360489.

[Fikes71] Fikes R. E., Nilsson N., "STRIPS: A new approach to the application of theorem proving to problem solving"Artificial Intelligence, vol. 5(2), pp. 189-208, 1971.

[FIPA] FIPA Home Page, http://www.fipa.org

[FIPA00001] FIPA Abstract Architecture Specification, XC00001J, http://www.fipa.org/specs/fipa00001

[FIPA00007] FIPA Content Language Library Specification, http://www.fipa.org/specs/fipa00007

[FIPA00008] FIPA SL Content Language Specification, SC00008I, http://www.fipa.org/specs/fipa00008

[FIPA00023] FIPA Agent Management Specification, http://www.fipa.org/specs/fipa00023

[FIPA00061] FIPA Agent Communication Language Specification, http://www.fipa.org/specs/fipa00061

[FIPA00067] FIPA Agent Message Transport Service Specification, http://www.fipa.org/specs/fipa00067

[FIPA00073] FIPA Agent Message Transport Envelope Representation in String Specification, http://www.fipa.org/specs/fipa00073

[Ford95] Ford K. M., Glymour C., Hayes P. J., "Android Epistemology", Menlo Park, 1995., Calif., AAAI Press.

[Franklin96] Franklin S., Graesser A., "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents", In Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996.

[Fukuta01] Fukuta N., Takayuki I., Ozono T., Shintani T., "A Framework for Cooperative Mobile Agents and Its Case-Study on BiddingBot", the JSAI 2001 International Workshop on Agent-based Approaches in Economic and Social Complex Systems (AESCS 2001), 2001.

[Ganguly00] Ganguly P., Ray P., Low G., "Software agent based approach towards tele-electrocardiography". Proceedings of the 13th IEEE Symposium on Computer-Based Medical Systems: CBMS 2000, IEEE Computer Society Press, Los Alamitos, CA: 275-80.

[Genesereth87] Genesereth M.R., Nillson N.J., "Logical Foundations of Artificial Intelligence", Morgan Kaufmann Publishers, San Mateo, 1987.

[Genesereth92] Genesereth M.R., Fikes R., "Knowledge Interchange Format", Logic Group, Report Logic -92-1, 1992.

[Georgeff87] Georgeff M. P., Lansky A. L., "Reactive reasonong and planning", Proceedings of the sixth National Conference on Artificial Intelligence (AAAI-87), Seattle, WA, 1987., pp. 677-682.

[Gilbert95] Gilbert D., Aparicio M., Atkinson B., Brady S., Ciccarino J., Grosof B., O'Connor P.,Osisek D., Pritko S., Spagna R., Wilson L., "IBM Intelligent Agent Strategy", IBM Corporation, 1995.

[Gustavsson99] Gustavsson R., "Agents with Power", Communications of the ACM, March 1999/Vol. 42., No. 3., pp. 41-47.

[Hayes-Roth89] Hayes-Roth B., Hewett M., Washington R., Hewet R., Seiver A., "Distributing intelligence within an individual", Distributed Artificial Intelligence Volume II, Pitman Publishing: London and Morgan Kaufman: San Mateo, CA, 1989, pp. 385-412.

[He98] He Q., Sycara K., Finin T., "Personal Security Agent: KQML-Based PKI", Proceedings of the second international conference on Autonomous agents, United States, Minneapolis, 1998.

[Head94] Head J., "Adding Real Intelligence to Mutual Management", LAN Times, vol. 11, No. 2, Janualry 24, 1994, pp. 30-33.

[Horling00] Horling B., Lesser V., Vincent R., "Multi-Agent System Simulation Framework". In 16th IMACS World Congress 2000 on Scientific Computation, Applied Mathematics and Simulation. August, 2000.

[Horling02] Horling B., Lesser V., Vincent R., Wagner T., "The Soft Real-Time Agent Control Architecture". In Proceedings of the AAAI/KDD/UAI-2002 Joint Workshop on Real-Time Decision Support and Diagnosis Systems. April, 2002.

[Horling98] Horling B., "A Reusable Component Architecture for Agent Construction". In University of Massachusetts/Amherst CMPSCI Technical Report 1998-49. October, 1998.

[J2EE] Java 2 Enterprise Edition Homepage, http://java.sun.com/j2ee/

[JAF] Java Agent Framework Home Page, http://mas.cs.umass.edu

[JAT] Java Agent Template Home Page, http://java.stanford.edu

[Java] Java homepage, http://java.sun.com

[JBoss] JBoss Application Server Homepage, http://www.jboss.org

[Jennings96] Jennings N. R., Corera J., Laresgoiti I., Mamdani E. H., Perriolat F., Skarek P., Varga Z., "Using ARCHON to develop real-world DAI applications for electricity transportation management and particle acceleration control", December 1996, IEEE Expert, vol. 11(6), pp. 60-88.

[Jennings98] Jennings N., Sycara K., Georgeff M., "A Roadmap of Agent Research and Development", Autonomous Agents and Multi-Agent Systems, 1998, Volume 1, Number 1, ISSN: 1387-2532.

[Jeon00] Jeon H., Petrie C., Cutkosky M., "JATLite: A Java Agent Infrastructure with Message Routing", IEEE Internet Computing Mar/Apr 2000.

[JMS] Java Messaging System Homepage, http://java.sun.com/products/jms

[Kautz94] Kautz H. A., Selman B., Coen M., "Bottom-up Design of Software Agents", Communications of ACM, vol 37 (7), 1994., pp. 143-147.

[Kendall00] Kendall E, Krishna M., Suresh, C., Pathak C., "An application framework for intelligent and mobile agents", ACM

Computing Surveys (CSUR), Volume 32 , Issue 1, (March 2000), ISSN:0360-0300.

[Kim02] Kim Tan H., Moreau L., "Certificates for mobile code security", Proceedings of the 17th symposium on Proceedings of the 2002 ACM symposium on applied computing, 2002 , Madrid, Spain, ISBN:1-58113-445-2, pp. 76 – 81.

[Knapik98] Knapik M., Johnson J., "Developing Intelligent Agents for Distributed Systems", McGraw-Hill, 1998, pp. 3, 37-39.

[Krumel98] Krumel, A., "Revolutionary RMI: Dynamic class loading and behavior objects", Java World, December 1998., http://www.javaworld.com/javaworld/jw-12-1998/jw-12-enterprise.html

[Kuno02] Kuno H., Sahai A., "My Agent Wants to Talk to Your Service: Personalizing Web Services through Agents", 1st International Workshop on "Challenges in Open Agent Systems", AAMAS'02, Bologna, Italy, July 2002.

[Labrou97] Labrou J., Finin T., "A Proposal for a new KQML Specification", TR CS-97-03, February 1997.

[Lewis96] Lewis J., "NETSCAPE Gets Serious about Infrastructure", The Burton Group, 1996.

[Liang98] Liang S., Bracha G., "Dynamic class loading in the Java virtual machine", Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications, 1998 , Vancouver, British Columbia, Canada, ISSN:0362-1340, pp. 36 – 44.

[Lieberman95] Lieberman H., "Letizia: An agent that assists web browsing", Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95), Montreal, Canada, August 1995, pp. 924-929.

[Lubar93] Lubar S., "InfoCulture: The Smithsonian Book of Information and Inventions", Boston, 1993., Mass., Houghton Mifflin.

[Lupu96] Lupu E., Marriott D., Sloman M., Yialelis N., "A policy based role framework for access control", Proceedings of the first ACM Workshop on Role-based access control, 1996 , Gaithersburg, Maryland, United States, ISBN:0-89791-759-6, Article No. 11, pp. II-15 – II 24.

[Lyell02] Lyell M., "Interoperability, standards, and software agent systems", 23'rd Army Science Conference, Orlando, Florida, USA, December 2-5, 2002.

[Maamar00] Maamar Z., Moulin B., "An Overview of Software Agent-Oriented Frameworks", ACM Computing Surveys, Volume 32, Article No. 19, March 2000, ISSN:0360-0300.

[Maes94] Maes P., "Agents that reduce work and iformation overload", Communications of ACM, vol 37(7), July 1994, pp. 31-40.

[Maes99] Maes P., Guttman R., Moukas A., "Agents That buy and Sell", Communications of the ACM, March 1999/Vol. 42., No. 3., pp. 81-91.

[Menzes97] Menzes, A., van Oorschot, P., Vanstone, S., "Handbook of Applied Cryptography", CRC Press, 1997., pp. 1-48.

[Milosavljević00a] Milosavljević B., "Uzajamna katalogizacija: preuzimanje bibliografskih zapisa", Infoteka 1(2000)1, pp. 19-23.

[Milosavljević00b] Milosavljević B., Konjović Z., "Arhitektura WWW prezentacije za pretraživanje u bibliotečkom informacionom sistemu", InfoScience 4(2000), pp. 18-23.

[Milosavljević02] Milosavljević B., Vidaković M., "Java i Internet Programiranje", Grupa za Informacione Tehnologije, 2002.

[Nardi98] Nardi B., Miller J., Wright D., "Collaborative, programmable intelligent agents", Communications of the ACM, Volume 41 , Issue 3 (March 1998), pp. 96-104.

[Neches91] Neches R., Fikes R., Finin T., Gruber T., Patil R., Senator T., Swartout W. R., "Enabling Technology for Knowledge Sharing.", AI Magazine, 1991., vol. 12(3), pp. 36–55.

[Negroponte97] Negroponte N., "Being Digital", New York, 1997., Alfred Knopf.

[Newel61] Newel A., Simon H. A., "GPS: A program that simulates human thought.", Lernende Automaten, R. Oldenbourg, KG, 1961.

[Norman97] Norman D. A., "How Might People Interact with Agents? In *Software Agents*", Menlo Park, 1997., Calif., AAAI Press.

[Orion] Orion Application Server Homepage, http://www.orionserver.com

[Palaniappan92] Palaniappan M., Yankelovich N., Fitzmaurice G., Loomis A., Haan B., Coombs J., Meyrowitz N., "The envoy framework: an open architecture for agents", ACM Transactions on Information Systems, Volume 10 , Issue 3 (July 1992), ISSN:1046-8188, pp. 233 – 264.

[Papazoglou01] Papazoglou M., "Agent-oriented technology in support of e-business", Communications of the ACM, Volume 44 , Issue 4, (April 2001) , ISSN:0001-0782, pp. 71 – 77.

[Poggi01] Poggi A., Rimassa G., Tomaiuolo M., "Multi-User and Security Support for Multi-Agent Systems", Proceedings of WOA 2001 Workshop, Modena, Sep 2001.

[Rana00] Rana O., Stout K., "What is scalability in multi-agent systems?", Proceedings of the fourth international conference on Autonomous agents, pp. 56-63, Spain, Barcelona, 2000.

[Roman99] Roman, E., "Mastering Enterprise JavaBeans™", Willey Computer Publishing, 1999.

[Russel95] Russel S., Norvig P., "Artificial Intelligence: A Modern Approach", Prentice-Hall, 1995.

[Schelde93] Schelde P., "Androids, Humanoids, and Other Science Fiction Monsters", New York, 1993., New York University Press.

[Shehory01] Shehory O., Sturm A., "Evaluation of modeling techniques for agent-based systems", Proceedings of the fifth international conference on Autonomous agents, Canada, Montreal, 2001, pp. 624-631.

[Surla00] Surla D., Konjović Z., Milosavljević B., Vidaković M., "Bibliotečki informacioni sistem BISIS ver. 3.01", Deveta međunarodna konferencija "Informatika u obrazovanju, kvalitet i nove informacione tehnologije", Zrenjanin 2000. str 494-504.

[Tai99] Tai H., Kosaka K., "The Aglets project", Communications of the ACM March 1999, Volume 42 Issue 3, pp. 100-101.

[Thai03] Thai B., Wan R., Seneviratne A., Rakotoarivelo T., "Integrated personal mobility architecture: a complete personal mobility solution", Mobile Networks and Applications, Volume 8 , Issue 1 (February 2003), pp. 27-36.

[Unicode] The Unicode Standard, Unicode Consortium, http://www.unicode.org

[UNIMARC] UNIMARC manual: bibliographic format / International Federation of Library Association and Institutions, IFLA Universal Bibliographic Control and International MARC Programme, NEw Providence, London, 1994.

[Varadharajan00] Varadharajan V., "Security enhanced mobile agents", Proceedings of the 7th ACM conference on Computer and communications security, Greece, Athens, November 2000, pp. 200-209.

[Vidaković01] Vidaković M., Konjović Z. “Server za uzajamnu katalogizaciju u bibliotečkom informacionom sistemu BISIS ”, YuInfo 2001, Kopaonik 2001.

[Vidaković02a] Vidaković M., Konjović Z., "EJB Based Intelligent Agents Framework", The 6th IASTED International Conference on Software Engineering and Applications (SEA 2002), November 4-6, 2002, MIT, Cambridge, USA.

[Vidaković02b] Vidaković M., Milosavljević B., Konjović Z., "Implementacija Servera za Uzajamnu Katalogizaciju u EJB Tehnologiji", YuInfo 2002, Kopaonik 2002.

[Vidaković02c] Vidaković M., Milosavljević B., Konjović Z., Surla D., "Konverzija tekstualnih ekranskih formi u GUI aplikacije upotrebom automata stanja", XXIX Jugoslovenski simpozijum o operacionim istraživanjima, SIM-OP-IS 2002, Tara, 9.-12.oktobar 2002., pp. XXV-13.

[Vidaković98a] Vidaković M., Konjović Z. "Implementacija korisničkog interfejsa za pretraživanje bibliografskih podataka u java okruženju", YuInfo 98, Kopaonik 1998.

[Vidaković98b] Vidaković M., "Objekto orijentisana specifikacija i implementacija korisničkog interfejsa za bibliotečki informacioni sistem", magistarska teza, Univerzitet u Novom Sadu, Institut za Računarstvo, Automatiku i Merenja, 1998.

[Vincent01] Vincent R., Horling B., Lesser V., "An Agent Infrastructure to Build and Evaluate Multi-Agent Systems: The Java Agent Framework and Multi-Agent System Simulator". In Lecture Notes in Artificial Intelligence: Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems., Volume 1887. January, 2001., ISBN 3-540-42315-X.

[Voyager] Voyager ® Home Page, http://www.recursionsw.com/products/voyager/voyager.asp

[Vuong01] Vuong S., Fu P., "A security architecture and design for mobile intelligent agent systems", ACM SIGAPP Applied Computing Review archive, Volume 9 , Issue 3 Fall 2001, pp. 21 – 30.

[Wagner01] Wagner T., Horling B., "The Struggle for Reuse and Domain Independence: Research with TAEMS", DTC and JAF. In Proceedings of the 2nd Workshop on Infrastructure for Agents, MAS, and Scalable MAS (Agents 2001). June, 2001.

[Wang01] Wang D., Wang J., "Towards the distributed processing of mobile software agents", ACM SIGAPP Applied Computing Review July 2001, Volume 9 Issue 2.

[WebSphere] IBM WebSphere Application Server, http://www-3.ibm.com/software/info1/websphere/

[Wilson00] Wilson L., Burroughs D., Sucharitaves J., Kumar, A., "An agent-based framework for linking distributed simulations", Proceedings of the 32nd conference on Winter simulation, 2000 , Orlando, Florida, ISBN:1-23456-789-0, pp. 1713 – 1721.

[Wong99] Wong D., Paciorek N., Moore D., "Java-based Mobile Agents", COMMUNICATIONS OF THE ACM, March 1999-Volume 42, Number 3, pp. 92.

[Wooldridge95] Wooldridge M., Jennings N., "Agent Theories, Architectures, and Languages: a Survey", In *Intelligent Agents*, Berlin: Springer-Verlag, 1995, ISBN: 0387588558.

[Yuh-Jong01] Yuh-Jong H., "Some thoughts on agent trust and delegation", Proceedings of the fifth international conference on Autonomous agents, Canada, Montreal, 2001.

Spisak skraćenica

ABLE	*Agent Building and Learning Environment*
ACC	*Agent Communication Channel*
ACL	*Agent Communication Language*
AMS	*Agent Management System*
ARL	*Able Rule Language*
ATCI	*Agent Transport and Communication Interface*
ATP	*Agent Transfer Protocol*
CA	*Certificate Authority*
CORBA	*Common Object Request Broker Architecture*
COS	*CORBA Common Object Services*
DF	*Directory Facilitator*
EJB	*Enterprise JavaBeans*
FIPA	*Foundation for Intelligent Physical Agents*
HCI	*Human Computer Interface*
IIOP	*Internet Inter-ORB Protocol*
IPMT	*Internal Platform Message Transport*
J2EE	*Java 2 platform – Enterprise Edition*
JADE	*Java Agent DEvelpment framework*
JAF	*Java Agent Framework*
JAT	*Java Agent Template*
JAT	*Java Agent Template*
JCE	*Java Cryptography Extension*
JDBC	*Java Database Connectivity*
JMS	*Java Messaging Service*
JNDI	*Java Naming and Directory Interface*
JTA	*Java Transaction API*

JTS	*Java Transaction Service*
KIF	*Knowledge Interchange Format*
KQML	*Knowledge Query and Manipulation Language*
LDAP	*Lightweight Directory Access Protocol*
MASS	*Multi Agent System Simulator*
OMG	*Object Management Group*
ORB	*Object Request Broker*
OTS	*Object Transaction Service*
PRS	*Procedural Reasoning System*
QAF	*Quality Accumulation Function*
RMI	*Remote Method Invocation*
TAEMS	*Task Analysis, Environment Modeling and Simulation*

Biografija

Milan Vidaković je rođen u Novom Sadu 18. avgusta 1971. godine. Diplomirao je na Fakultetu tehničkih nauka Univerziteta u Novom Sadu 1995. godine, na odseku Elektrotehnika i računarstvo, smer Računarstvo i upravljanje sistemima, usmerenje Računarstvo. Diplomski rad sa temom "Miltimedijalna prezentacija Univerziteta u Novom Sadu" je odbranio sa ocenom 10 (deset).

Odmah po diplomiranju započeo je rad na Fakultetu tehničkih nauka kao stipendista Ministarstva za nauku i tehnologiju. Radni odnos je zasnovao 1998. godine na Institutu za računarstvo, automatiku i merenja Fakulteta tehničkih nauka u Novom Sadu.

Na poslediplomske studije, smer računarski sistemi, se upisao 1995/96. godine na Fakultetu tehničkih nauka u Novom Sadu. Odbranio je magistarski rad pod nazivom "Objektno orijentisana specifikacija i implementacija korisničkog interfejsa za bibliotečki informacioni sistem", 1998. godine.

U toku studija i rada na Fakultetu objavio je 25 radova u monografijama, domaćim i stranim zbornicima sa konferencija, sa ukupnim koeficijentom kompetencije $R = 25{,}4$. Pored toga, održao je jedno predavanje po pozivu, objavio jednu knjigu i učestvovao u izradi tri projekta.

Oženjen je i živi u Novom Sadu. Od stranih jezika govori engleski jezik.

Univerzitet u novom sadu
Fakultet tehničkih nauka

Ključna dokumentacijska informacija

Redni broj: RBR	
Identifikacioni broj: IBR	
Tip dokumentacije:	monografska dokumentacija
Tip zapisa: TZ	tekstualni štampani tekst
Vrsta rada: BR	doktorska disertacija
Autor: AU	Milan Vidaković
Mentor: MN	prof. dr Zora Konjović, vanr. profesor
Naslov rada: NR	Proširivo agentsko okruženje bazirano na Java tehnologiji
Jezik publikacije: JP	srpski (latinica)
Jezik izvoda: JI	srpski (latinica)/engleski
Zemlja publikovanja: ZP	Srbija i Crna Gora
Uže geografsko područje: UGP	Vojvodina
Godina: GO	2003
Izdavač:	autorski reprint

IZ

Mesto i adresa: MA	Fakultet tehničkih nauka Trg D. Obradovića 6, Novi Sad
Fizički opis rada: FO	7/182/118/13/56/0/0
Naučna oblast: NO	Računarske nauke
Uža naučna oblast: ND	agentska tehnologija
Ključne reči: PO	agent, agentsko okruženje, Java, UML
UDK:	
Čuva se: ČU	Biblioteka fakulteta tehničkih nauka Trg D. Obradovića 6, Novi Sad
Važna napomena: VN	
Datum prihvatanja od strane NN veća: DP	
Datum odbrane: DO	

Izvod:
IZ

Agentska paradigma predstavlja najprirodniji i najdosledniji postojeći pristup implementaciji distribuiranih sistema. Uz pomoć agenata moguće je u potpunosti realizovati koncept distribuiranih softverskih komponenti, koje će, osim rešenja zadatka na distribuiranom nivou, pružiti i određenu količinu autonomnosti i inteligencije da bi se zadati cilj ostvario. Agentsko okruženje predstavlja programsko okruženje koje upravlja životnim tokom agenata i obezbeđuje mu sve potrebne mehanizme za realizaciju zadatka. U ovoj doktorskoj disertaciji predložen je model agentskog okruženja baziran na tehnologiji distribuiranih komponenti, koji podržava FIPA specifikaciju i sledeće koncepte: razmenu poruka, mobilnost agenata, sigurnosne mehanizme i direktorijume agenata i servisa. Model agentskog okruženja je implementiran u J2EE tehnologiji. Podržan je sistem *plug-in*-ova za sve bitne komponente agentskog okruženja (menadžere). Modelovan je i implementiran koncept mobilnih zadataka. Dat je model i implementacija sistema međusobnog uređenja odnosa agentskih centara. Predloženo rešenje agentskog okruženja verifikovano je na bibliotečkom informacionom sistemu BISIS. Verifikacija je izvršena na sledećim agentskim zadacima: pretraživanje bibliotečke mreže, ocenjivanje kvaliteta zapisa i inteligentna raspodela opterećenja.

Komisija:
KO

Predsednik:	dr Surla Dušan, r. prof., PMF Novi Sad
član:	dr Seder Ivan, r. prof., Fachhochule Merseburg, University of Applied Sciences
član:	dr Konjović Zora, v. prof., FTN Novi Sad, mentor
član:	dr Malbaški Dušan, r. prof., FTN Novi Sad
član:	dr Hajduković Miroslav, r. prof., FTN Novi Sad

University of novi sad
Faculty of enigineering

Key word documentation

Accession number: ANO	
Identification number: INO	
Document type: DT	monograph documentation
Type of record: TR	textual printed material
Contents code: CC	doctoral dissertation
Author: AU	Milan Vidaković
Mentor: MN	Zora Konjović, Ph.D, assoc. prof.
Title: TI	Extensible Java based agent framework
Language of text: LT	Serbian (latin)
Language of abstract: LA	Serbian(latin)/English
Country of publication: CP	Serbia and Montenegro
Locality of publication: LP	Vojvodina
Publication year: PY	2003

Publisher: author's reprint
PU

Publ. place: Novi Sad, Faculty of Engineering,
PP Trg D. Obradovića 6

Physical description: 7/182/118/13/56/0/0
PD

Scientific field: computer science
SF

Scientific discipline: agent technology
SD

Subject/Key words: agent, agent framework, Java, UML
SKW

UDC:

Holding data: Library of Faculty of Engineering,
HD Trg D. Obradovića 6, Novi Sad

Note:
N

Accepted by Scientific Board on:
ASB

Defended:
DE

Abstract:
AB

Agent technology is one of the most consistent approaches to the distributed computing implementation. Agents can be used to fully implement distributed software component concept. Agents can solve distributed problems utilizing certain degree of autonomy and intelligence. Agent framework represents programming environment that controls agent life cycle and provides all necessary mechanisms for task execution. The subject of the dissertation is formal specification of an agent framework based on distributed component technology. This framework supports FIPA specification and following concepts: message interchange, agent mobility, security and agent and service directory. Agent framework is implemented in J2EE technology. Plug-in system is designed for all key elements of agent framework. Mobile tasks were specified and implemented. Also, inter-facilitator connectivity mechanism is specified and implemented. The framework is verified by a case study on the library information system BISIS. Following agent tasks were performed: library network search, library record quality estimation and intelligent load balansing.

Thesis defend board:
DB

president: prof. Surla Dušan, PhD, Faculty of Science, Novi Sad
member: prof. Seder Ivan, PhD, Fachhochule Merseburg, University of Applied Sciences
member: prof. Konjović Zora, PhD, Faculty of Engineering Novi Sad, menthor
member: prof. dr Malbaški Dušan, PhD, Faculty of Engineering Novi Sad
member: prof. dr Hajduković Miroslav, PhD, Faculty of Engineering Novi Sad